English

Fujitsu Software BS2000

COBOL Compiler

Reference Manual

Valid for:
COBOL2000 V1.6

Edition November 2022

Comments… Suggestions… Corrections…

The User Documentation Department would like to know your opinion on this manual. Your feedback helps us to optimize our documentation to suit your individual needs.

Feel free to send us your comments by e-mail to: bs2000.info@fujitsu.com.

Certified documentation according to DIN EN ISO 9001:2015

To ensure a consistently high quality standard and user-friendliness, this documentation was created to meet the regulations of a quality management system which complies with the requirements of the standard DIN EN ISO 9001:2015.

Copyright and Trademarks

Copyright © 2025 Fujitsu

All rights reserved.
Delivery subject to availability; right of technical modifications reserved.

All hardware and software names used are trademarks of their respective manufacturers.

Table of Contents

COBOL Compiler. Reference Manual. 15
1 Preface 16
1.1 Brief product description 17
1.2 Target group and summary of contents 18
1.3 Changes compared to the predecessor version 20
1.4 Acknowledgment 21
1.5 Readme file 22
2 Introduction to the COBOL language 23
2.1 Glossary 24
2.2 COBOL notation 54
2.3 Reference format 58
2.3.1 General description of the fixed form reference format 59
2.3.2 Rules for using the fixed form reference format 60
2.3.3 General description of the free form reference format 62
2.3.4 Rules for using the free form reference format 63
2.4 Language concepts 64
2.4.1 COBOL character set 65
2.4.2 Separators 66
2.4.3 COBOL words 67
2.4.4 Literals 82
2.4.5 PICTURE character-string 89
2.4.6 Types 90
2.4.7 Zero-length items 91
2.4.8 Concept of computer-independent data description 92
2.4.9 Implementor-dependent representation and alignment of data 101
2.5 Uniqueness of references 104
2.5.1 Qualification 105
2.5.2 Subscripting 107
2.5.3 Indexing 109
2.5.4 Function-identifier 111
2.5.5 Reference modification 112
2.5.6 Identifier 114
2.5.7 Object view 115
2.5.8 Predefined object references 116
2.5.8.1 NULL 117
2.5.8.2 SELF and SUPER 118
2.5.9 Predefined address NULL 119

2.5.10 Data address identifier 120
2.5.11 Program address identifier 121
2.5.12 BYTE-LENGTH OF 122
2.5.13 LENGTH OF 123
2.5.14 Condition-name 124
2.6 Table handling 125
2.6.1 Table definition 126
2.6.2 Subscripting 129
2.6.3 Indexing 131
2.6.4 Indexing and subscripting compared 133
2.7 Statements and sentences 134
2.7.1 Conditional statements and conditional sentences 135
2.7.2 Compiler-directing statements and compiler-directing sentences 136
2.7.3 Imperative statements and imperative sentences 137
2.7.4 Delimited scope statements 139
2.7.5 Scope of statements (scope terminators) 140
2.8 Processing a COBOL program 141
2.9 EBCDIC character set 142
3 Controlling the compiler 147
3.1 Statements for source text manipulation 148
3.1.1 COPY statement 149
3.1.2 REPLACE statement 153
3.2 Compiler directives 155
3.2.1 CALL-CONVENTION directive 158
3.2.2 DEFINE directive 160
3.2.3 EVALUATE directive 161
3.2.4 FLAG-85 directive 163
3.2.5 IF directive 164
3.2.6 IMP directive 165
3.2.6.1 IMP COMPILER-ACTION 166
3.2.6.2 IMP LISTING-OPTIONS 167
3.2.6.3 IMP PRINT-DIRECTIVES 168
3.2.6.4 IMP RUNTIME-ERRORS 169
3.2.6.5 IMP SET-DIRECTIVES 170
3.2.7 LISTING directive 171
3.2.8 PAGE directive 172
3.2.9 SOURCE FORMAT directive 173
3.2.10 TURN directive 174
4 Structure of a COBOL compilation group 175
4.1 General description 176
4.2 COBOL compilation group 177

4.3 END markers 180
5 Identification Division 181
5.1 General description 182
5.2 General format 183
5.3 Paragraphs 184
5.3.1 PROGRAM-ID paragraph 185
5.3.2 CLASS-ID paragraph 188
5.3.3 FACTORY paragraph 189
5.3.4 OBJECT paragraph 190
5.3.5 METHOD-ID paragraph 191
5.3.6 INTERFACE-ID paragraph 192
6 Environment Division 193
6.1 General description 194
6.2 CONFIGURATION SECTION 195
6.2.1 SOURCE-COMPUTER paragraph 196
6.2.2 OBJECT-COMPUTER paragraph 197
6.2.3 SPECIAL-NAMES paragraph 198
6.2.3.1 Implementor-name 200
6.2.3.2 ARGUMENT-NUMBER / ARGUMENT-VALUE / ENVIRONMENT-NAME / ENVIRONMENT-VALUE 202
6.2.3.3 ALPHABET clause 203
6.2.3.4 SYMBOLIC CHARACTERS clause 208
6.2.3.5 CLASS clause 209
6.2.3.6 CURRENCY SIGN clause 210
6.2.3.7 DECIMAL-POINT IS COMMA clause 211
6.2.4 REPOSITORY paragraph 212
6.3 INPUT-OUTPUT SECTION 214
6.3.1 FILE-CONTROL paragraph 215
6.3.1.1 SELECT clause 217
6.3.1.2 ASSIGN clause 220
6.3.1.3 ACCESS MODE clause 222
6.3.1.4 ALTERNATE RECORD KEY clause 224
6.3.1.5 FILE STATUS clause 225
6.3.1.6 ORGANIZATION clause 226
6.3.1.7 PADDING CHARACTER clause 227
6.3.1.8 RECORD DELIMITER clause 228
6.3.1.9 RECORD KEY clause 229
6.3.1.10 RESERVE clause 230
6.3.2 I-O-CONTROL paragraph 231
6.3.2.1 MULTIPLE FILE TAPE clause 232
6.3.2.2 RERUN clause 233

6.3.2.3 SAME AREA clause 235
7 Data Division 237
7.1 General description 238
7.1.1 Structure of a Data Division 239
7.1.2 General format 240
7.1.2.1 FILE SECTION 241
7.1.2.2 WORKING-STORAGE SECTION 242
7.1.2.3 LOCAL-STORAGE SECTION 243
7.1.2.4 LINKAGE SECTION 244
7.1.2.5 REPORT SECTION 245
7.1.2.6 SUB-SCHEMA SECTION 246
7.2 File description 247
7.2.1 Formats of the file description entry 248
7.2.2 Clauses for data description entries 251
7.2.2.1 BLOCK CONTAINS clause 252
7.2.2.2 CODE-SET clause 254
7.2.2.3 DATA RECORDS clause 255
7.2.2.4 EXTERNAL clause 256
7.2.2.5 GLOBAL clause 258
7.2.2.6 LABEL RECORDS clause 259
7.2.2.7 LINAGE clause 261
7.2.2.8 RECORD clause 264
7.2.2.9 RECORDING MODE clause 267
7.2.2.10 VALUE OF clause 268
7.3 Data description entry 269
7.3.1 General description 270
7.3.2 Data description entry formats 272
7.3.2.1 Level number 274
7.3.3 Clauses for data description 276
7.3.3.1 ANY LENGTH clause 277
7.3.3.2 BASED clause 279
7.3.3.3 BLANK WHEN ZERO clause 280
7.3.3.4 DYNAMIC clause 281
7.3.3.5 Data-name or FILLER clause 282
7.3.3.6 EXTERNAL clause 283
7.3.3.7 GLOBAL clause 285
7.3.3.8 GROUP-USAGE clause 286
7.3.3.9 JUSTIFIED clause 287
7.3.3.10 OCCURS clause 289
7.3.3.11 PICTURE clause 295
7.3.3.12 REDEFINES clause 306

7.3.3.13 RENAMES clause . . . 309
7.3.3.14 SIGN clause . . . 311
7.3.3.15 SYNCHRONIZED clause . . . 315
7.3.3.16 TYPE clause . . . 318
7.3.3.17 TYPEDEF clause . . . 319
7.3.4 USAGE clause . . . 320
7.3.4.1 DISPLAY phrase . . . 322
7.3.4.2 NATIONAL phrase . . . 324
7.3.4.3 BINARY phrase or COMPUTATIONAL phrase or COMPUTATIONAL-5 phrase . . . 325
7.3.4.4 COMPUTATIONAL-1 phrase . . . 326
7.3.4.5 COMPUTATIONAL-2 phrase . . . 327
7.3.4.6 COMPUTATIONAL-3 phrase or PACKED-DECIMAL phrase . . . 328
7.3.4.7 INDEX phrase . . . 331
7.3.4.8 OBJECT REFERENCE phrase . . . 332
7.3.4.9 POINTER phrase . . . 334
7.3.4.10 PROGRAM-POINTER phrase . . . 335
7.3.4.11 VALUE clause . . . 336
8 Procedure Division . . . 343
8.1 General description . . . 344
8.1.1 Structure . . . 345
8.2 Procedure Division header . . . 347
8.3 DECLARATIVES . . . 349
8.4 Arithmetic expressions . . . 350
8.5 Conditions . . . 353
8.5.1 Condition-name condition . . . 354
8.5.2 Class condition . . . 355
8.5.3 Switch-status condition . . . 357
8.5.4 Relation condition . . . 358
8.5.5 Sign condition . . . 364
8.5.6 OMITTED-ARGUMENT condition . . . 365
8.5.7 Complex conditions . . . 366
8.5.8 Implied subjects and relational operators . . . 368
8.6 Arithmetic statements . . . 370
8.7 Phrases in statements . . . 372
8.7.1 CORRESPONDING phrase . . . 373
8.7.2 GIVING phrase . . . 375
8.7.3 ROUNDED phrase . . . 376
8.7.4 ON SIZE ERROR phrase . . . 377
8.8 Overlapping operands . . . 379
8.9 Incompatible data . . . 380

8.10 Statements 381
8.10.1 ACCEPT statement 382
8.10.2 ADD statement 386
8.10.3 ALLOCATE statement 389
8.10.4 ALTER statement 391
8.10.5 CALL statement 392
8.10.6 CANCEL statement 402
8.10.7 CLOSE statement 404
8.10.8 COMPUTE statement 408
8.10.9 CONTINUE statement 409
8.10.10 DELETE statement 411
8.10.11 DISPLAY statement 412
8.10.12 DIVIDE statement 416
8.10.13 ENTRY statement 419
8.10.14 EVALUATE statement 421
8.10.15 EXIT statement 426
8.10.16 EXIT METHOD statement 427
8.10.17 EXIT PARAGRAPH statement 428
8.10.18 EXIT PERFORM statement 429
8.10.19 EXIT PROGRAM statement 431
8.10.20 EXIT SECTION statement 432
8.10.21 FREE statement 433
8.10.22 GOBACK statement 434
8.10.23 GO TO statement 435
8.10.24 IF statement 437
8.10.25 INITIALIZE statement 440
8.10.26 INSPECT statement 447
8.10.27 INVOKE statement 454
8.10.28 MERGE statement 457
8.10.29 MOVE statement 463
8.10.30 MULTIPLY statement 471
8.10.31 OPEN statement 473
8.10.32 PERFORM statement 477
8.10.33 RAISE statement 493
8.10.34 READ statement 494
8.10.35 RELEASE statement 499
8.10.36 RESUME statement 500
8.10.37 RETURN statement 501
8.10.38 REWRITE statement 503
8.10.39 SEARCH statement 506
8.10.40 SET statement 513

8.10.41 SORT statement 523
8.10.42 START statement 533
8.10.43 STOP statement 535
8.10.44 STRING statement 536
8.10.45 SUBTRACT statement 539
8.10.46 UNSTRING statement 542
8.10.47 USE statement 546
8.10.48 WRITE statement 558
9 Intrinsic functions 566
9.1 General 567
9.2 Overview of intrinsic functions 569
9.2.1 ACOS - Arccosine 573
9.2.2 ADDR - Address of an identifier 574
9.2.3 ANNUITY - Annuity 575
9.2.4 ASIN - Arcsine 577
9.2.5 ATAN - Arctangent 578
9.2.6 BYTE-LENGTH - Number of bytes 579
9.2.7 CHAR - Character in the alphanumeric collating sequence 580
9.2.8 CHAR-NATIONAL - Character in the national collating sequence 581
9.2.9 COS - Cosine 582
9.2.10 CURRENT-DATE - Current date 583
9.2.11 DATE-OF-INTEGER - Date conversion 584
9.2.12 DATE-TO-YYYYMMDD - year conversion 585
9.2.13 DAY-OF-INTEGER - Date conversion 587
9.2.14 DAY-TO-YYYYDDD - year conversion 588
9.2.15 DISPLAY-OF - alphanumeric string 590
9.2.16 EXCEPTION STATUS 592
9.2.17 FACTORIAL - Factorial 593
9.2.18 INTEGER - Next smaller integer 594
9.2.19 INTEGER-OF-DATE - Date conversion 595
9.2.20 INTEGER-OF-DAY - Date conversion 596
9.2.21 INTEGER-PART - Integer part of a floating-point value 597
9.2.22 LENGTH - Number of characters 598
9.2.23 LOG - Logarithm 599
9.2.24 LOG10 - Logarithm of base 10 600
9.2.25 LOWER-CASE - Lowercase letters 601
9.2.26 MAX - Value of maximum argument 602
9.2.27 MEAN - Arithmetic mean of arguments 604
9.2.28 MEDIAN - Median of arguments 605
9.2.29 MIDRANGE - Mean of minimum and maximum arguments 606
9.2.30 MIN - Value of minimum argument 607

9.2.31 MOD - Modulo 609
9.2.32 NATIONAL-OF - national character representation 610
9.2.33 NUMVAL - Numeric value of string 612
9.2.34 NUMVAL-C - Numeric value of string with optional currency sign 614
9.2.35 ORD - Ordinal position in collating sequence 616
9.2.36 ORD-MAX - Ordinal position of maximum argument 617
9.2.37 ORD-MIN - Ordinal position of minimum argument 618
9.2.38 PRESENT-VALUE - Present value (period-end amount) 619
9.2.39 RANDOM - Random number 620
9.2.40 RANGE - Difference value 623
9.2.41 REM - Remainder 624
9.2.42 REVERSE - Reverse order of string characters 625
9.2.43 SIN - Sine 626
9.2.44 SQRT - Square root 627
9.2.45 STANDARD-DEVIATION - Standard deviation of arguments 628
9.2.46 SUM - Sum of arguments 629
9.2.47 TAN - Tangent 630
9.2.48 UPPER-CASE - Uppercase letters 631
9.2.49 VARIANCE - Variance of arguments 632
9.2.50 WHEN-COMPILED - Date and time of compilation 633
9.2.51 YEAR-TO-YYYY year conversion 634
10 Report Writer 637
10.1 General description 638
10.1.1 General description of the Data Division 639
10.1.2 General description of the Procedure Division 641
10.2 Language elements of the Data Division 642
10.2.1 REPORT clause 643
10.2.2 REPORT SECTION 644
10.2.3 Report description entry 645
10.2.4 CODE clause 646
10.2.5 CONTROL clause 647
10.2.6 GLOBAL clause 650
10.2.7 PAGE LIMIT clause 651
10.2.8 Report group description entry 656
10.2.9 COLUMN clause 659
10.2.10 GROUP INDICATE clause 660
10.2.11 LINE clause 662
10.2.12 NEXT GROUP clause 665
10.2.13 PICTURE clause 667
10.2.14 SIGN clause 668
10.2.15 SOURCE clause 669

10.2.16 SUM clause 670
10.2.17 TYPE clause 677
10.2.18 USAGE clause 680
10.2.19 VALUE clause 681
10.3 Language elements of the Procedure Division 682
10.3.1 GENERATE statement 683
10.3.2 INITIATE statement 685
10.3.3 TERMINATE statement 686
10.3.4 USE BEFORE REPORTING statement 687
10.4 Special registers of the Report Writer 689
10.4.1 LINE-COUNTER special register 690
10.4.2 PAGE-COUNTER special register 691
10.4.3 PRINT-SWITCH special register 692
10.4.4 CBL-CTR special register 693
10.4.4.1 Function 1 of the CBL-CTR special register 694
10.4.4.2 Function 2 of the CBL-CTR special register 696
11 XML 697
11.1 General description 698
11.2 Language elements of the ENVIRONMENT DIVISION 700
11.2.1 FILE-CONTROL paragraph 701
11.2.1.1 SELECT clause 702
11.2.1.2 ASSIGN clause 703
11.2.1.3 ACCESS MODE clause 705
11.2.1.4 FILE STATUS clause 706
11.2.1.5 ORGANIZATION clause 707
11.3 Language elements of the DATA DIVISION 708
11.3.1 File description entry 710
11.3.1.1 EXTERNAL clause 711
11.3.1.2 GLOBAL clause 713
11.3.2 Data description entry 714
11.3.2.1 COUNT clause 716
11.3.2.2 IDENTIFIED clause 717
11.4 Language elements of the PROCEDURE DIVISION 720
11.4.1 CLOSE statement 722
11.4.2 CLOSE DOCUMENT statement 723
11.4.3 OPEN statement 724
11.4.4 OPEN DOCUMENT statement 725
11.4.5 READ statement 727
11.4.6 START statement 730
11.4.7 XML GENERATE statement 732
11.4.8 XML PARSE statement 734

11.5 Special registers for the XML PARSE statement 736
12 General concepts 739
12.1 File processing 740
12.1.1 Sequential file organization 741
12.1.1.1 Record sequential organization 742
12.1.1.2 Line sequential organization 743
12.1.1.3 I-O status 744
12.1.2 Relative file organization 747
12.1.2.1 Relative organization 748
12.1.2.2 Sequential access to records 749
12.1.2.3 Random access to records 750
12.1.2.4 Dynamic access to records 751
12.1.2.5 I-O status 752
12.1.3 Indexed file organization 755
12.1.3.1 Indexed organization 756
12.1.3.2 Sequential access to records 757
12.1.3.3 Random access to records 758
12.1.3.4 Dynamic access to records 759
12.1.3.5 I-O status 760
12.1.4 Input/output statements 763
12.1.5 Invalid key condition 764
12.1.6 At end condition 765
12.2 Exception conditions and exception statuses 766
12.3 Initial and "last-used" states 768
12.4 Inter-program communication 769
12.4.1 Concepts 770
12.4.2 Control of inter-program communication 772
12.4.2.1 Runtime control 773
12.4.3 Rules for program names 774
12.4.4 Initial state for inter-program communication 776
12.4.5 Using common data 778
12.4.5.1 External and internal data 779
12.4.5.2 Local and global names 780
12.4.6 Language elements for inter-program communication 783
12.4.6.1 Overview 784
12.5 Sorting records 785
12.5.1 Sorting and merging files 786
12.5.1.1 Sort processing 787
12.5.1.2 Merge processing 788
12.5.1.3 Sort and merge without input/output procedures 789
12.5.1.4 Sort with input/output procedures 790

12.5.1.5 Overview of language elements . . . 791
12.5.2 Special registers for files: SORT . . . 793
12.5.3 Sorting two-digit year numbers with a century window . . . 795
12.5.4 Sorting with extended character sets (XHCS) . . . 797
12.6 Character representation by UTF-16 . . . 799
12.6.1 National data . . . 800
12.6.2 Data structures, clauses . . . 801
12.6.3 National literals . . . 802
12.6.4 Moving national data items . . . 803
12.6.5 National data items in conditions . . . 804
12.6.6 Conversions between EBCDIC and UTF-16 representation . . . 805
12.6.7 Error handling in the event of conversion . . . 806
12.7 Object-oriented concepts . . . 807
12.7.1 Fundamentals of object-oriented programming . . . 808
12.7.2 Parameterized classes and interfaces . . . 813
12.7.3 Files in objects . . . 820
12.7.4 Conformance . . . 823
12.7.4.1 Conformance between interfaces . . . 824
12.7.4.2 Conformance for parameters and returning items . . . 826
12.7.5 The system class BASE . . . 832
12.7.5.1 New method . . . 833
12.7.5.2 FactoryObject method . . . 834
12.7.6 Automatic release of memory space (garbage collection) . . . 835
12.8 Data types . . . 836
12.8.1 Weakly typed data descriptions . . . 837
12.8.2 Strongly typed data descriptions . . . 838
12.9 Addresses and pointers . . . 839
12.9.1 Data addresses and data pointers . . . 840
12.9.2 Using data pointers . . . 841
12.9.3 Program addresses and program pointers . . . 843
12.9.4 Using program pointers . . . 844
12.9.5 Type-specific pointers . . . 845
12.10 Language elements for processing XML . . . 846
12.10.1 Structure-oriented processing . . . 847
12.10.1.1 XML document as a tree . . . 848
12.10.1.2 COBOL language elements for describing an XML document . . . 850
12.10.1.3 Definition of an XML document in a COBOL program . . . 856
12.10.1.4 Statements for XML processing . . . 859
12.10.1.5 OPEN, CLOSE . . . 863
12.10.1.6 OPEN DOCUMENT . . . 864
12.10.1.7 CLOSE DOCUMENT . . . 866

12.10.1.8 READ 867
12.10.1.9 START 879
12.10.1.10 Error handling 884
12.10.1.11 Namespace 888
12.10.2 Event-oriented processing 892
12.10.2.1 XML statement 893
12.10.2.2 Special registers 895
12.10.2.3 Processing procedure 897
12.10.3 XML Common Syntactic Constructs 902
12.11 Debugging 904
13 Segmentation 905
13.1 General description 906
13.1.1 Organization 907
13.1.2 Fixed portion of the program 908
13.1.3 Independent segments 909
13.2 General rules for segmentation 910
13.3 Language elements 912
13.3.1 Language elements of the Environment Division 913
13.3.1.1 SEGMENT-LIMIT clause 914
13.3.2 Language elements of the Procedure Division 915
13.3.2.1 Segment number 916
14 Summary of obsolete elements 917
15 Related publications 918

COBOL Compiler. Reference Manual.

1 Preface

1.1 Brief product description

In most cases, the solving of commercial problems involves processing large amounts of data. COBOL is particularly well suited to this task. COBOL programs are largely independent of the particular features of individual hardware systems. The language is laid down clearly and precisely in an official document issued by the American National Standards Institute (ANSI) under the title

> "American National Standard for Information Systems
> - Programming Language COBOL -
> ANSI X3.23-1985" and Addendum "ANSI X3.23a-1989, Intrinsic Function Module".

This is a revised version of the 1974 standard. The internal standard functions supported as of V2.1A of the compiler are described in the above Addendum.

The German standard version DIN 66028-1986 and the international standard version ISO 1989:1985 correspond to the American National Standard. The Intrinsic Functions by ANSI correspond to the international norm "ISO/IEC 1989 Amendment 1, Intrinsic Function Module".

For the purpose of description, the ANSI publication divides COBOL into a nucleus and eleven functional modules, of which five are optional (Report Writer, Communication, Debug, Segmentation, Intrinsic Functions). Each of these modules in turn contains one or two functional levels. The lower level of a module is a true subset of the higher level of the same module.

Since December 2002, the international standard ISO 1989:2002 has been valid for COBOL. This standard is similar to the ISO 1989:1985 standard except that it does not contain the previous subdivision into modules, and, on the other hand, Amendment 1, “Intrinsic Function Module“, and numerous new language elements have been added.

The COBOL2000 (BS2000) compiler supports the high ANS85 COBOL language set. The optional Report Writer and Segmentation language modules are also supported in accordance with the high level of ANS85.
In addition, the COBOL2000 (BS2000) compiler offers a large subset of the language functionality from the ISO 1989:2002 standard, which is now valid.

The optional Communication and Debug language modules, which have been dropped from the new standard, are not supported. In BS2000, these modules are replaced by the products openUTM and AID, respectively.

1.2 Target group and summary of contents

The present manual is aimed at programmers and training personnel. It is intended as a guide to the writing and maintenance of COBOL programs and as a complement to training manuals. It is neither a COBOL textbook nor a user guide.
Readers are assumed to have a sound general knowledge of programming and some basic knowledge of COBOL.

The operation of the compiler and the creation of an executable COBOL program are described in the "COBOL2000 User Guide" [1].

The manual includes all language elements which may be used when creating COBOL programs, organized according to function, format, syntax rules, general rules, and examples:

The **function** section offers a concise, general description of the individual language elements. If several formats are involved, the functional differences between them are explained in brief.

The **format** section defines the specific arrangement of character strings and separators required for a valid clause, statement, or compound structure. The occurrence of specific strings and separators and their order of appearance as shown in the format section are decisive.

The specific notation used for describing the formats is explained under the heading "General format".

Where more than one specific arrangement is permitted, the various formats are designated as "**Format 1**, **Format 2**" etc.

The **syntax rules** section describes the particular requirements and restrictions for a given function and offers additional explanations and application guidelines.

General rules describe the use of the language structure within the program context; that is, as a function of previous and subsequent as well as superior and subordinate structures and in conjunction with references and cross-references from other language elements which, strictly speaking, are independent of the described structure. Restrictions on the order of effects at program runtime are discussed. Generally speaking, all these considerations are concerned with those elements which do not appear directly in the format section.

Under **Example** you will find a concrete example of the language element that has just been described.

The structure is analogous to that used for the standard COBOL document.

Certain language elements are qualified by a colors, as follows:

Bluish green print	COBOL2000 compiler extensions to the 1985 COBOL language standard. These include: • implementor-defined extensions • extensions from the Journal of Development (JOD) • extensions from the X/OPEN Portability Guide • extensions rom the 2002 COBOL standard
Orange print	Language elements to be avoided in new programs, since they will not be supported by future COBOL standards (obsolete elements). It is advisable to remove them from old programs.

The "Contents" table gives an overview of the general structure and organization of the manual.

The "Index" enables rapid access to desired information.

The most important terms and concepts used in this manual are defined in alphabetical order in the "Glossary".

Other manuals are referred to in the text by their abbreviated titles. The full title of each publication mentioned is given at the back of the manual under "Related publications".

The "Glossary" and "Index" sections have been excluded from the color qualification system.

1.3 Changes compared to the predecessor version

Processing XML files

- IDENTIFIED and COUNT clauses
- New formats of the OPEN, CLOSE, READ and START statements
- XML PARSE statement
- XML GENERATE statement

1.4 Acknowledgment

The COBOL programming language described in this manual is based on the language defined in the standard document "American National Standard for Information Systems - Programming Language - COBOL, X.3.23-1985". In recognition of the efforts made to develop and standardize COBOL, it is customary to precede a description of COBOL with the following text:

"Any organization interested in reproducing the COBOL standard and specifications in whole or in part, using ideas from this document as the basis for an instruction manual or for any other purpose, is free to do so. However, all such organizations are requested to reproduce the following acknowledgment paragraphs in their entirety as part of the preface to any such publication (any organization using a short passage from this document, such as in a book review, is requested to mention "COBOL" in acknowledgment of the source, but need not quote the acknowledgment):

COBOL is an industry language and is not the property of any company or group of companies, or of any organization or group of organizations.

No warranty, expressed or implied, is made by any contributor or by the CODASYL COBOL Committee as to the accuracy and functioning of the programming system and language. Moreover, no responsibility is assumed by any contributor, or by the committee, in connection therewith.

The authors and copyright holders of the copyrighted materials used herein

FLOW-MATIC (trademark of Sperry Rand Corporation), Programming for the UNIVAC (R) I and II, Data Automation Systems copyrighted 1958, 1959, by Sperry Rand Corporation; IBM Commercial Translater Form No. F 28-8013, copyrighted 1959 by IBM; FACT, DSI 27A5260-2760, copyrighted 1960 by Minneapolis-Honeywell

have specifically authorized the use of this material in whole or in part, in the COBOL specifications. Such authorization extends to the reproduction and use of COBOL specifications in programming manuals or similar publications."

1.5 Readme file

Any functional changes or additions to the current product version as described in this user guide can be found in the product-specific readme file. The readme file is located on your BS2000/OSD computer under the file name SYSRME.COBOL2000-GEM.015.E. Please ask your system support staff for the user ID under which the readme file is stored.

The following command outputs the complete path name:

```
/SHOW-INSTALLATION-PATH INSTALLATION-UNIT=COBOL2000-GEM,LOGICAL-IDENTIFIER=SYSRME.E
```

You can view the readme file using the /SHOW-FILE command or open it in an editor. You can also output it at a standard printer using the following command:

```
/PRINT-DOCUMENT <pathname>, LINE-SPACING=*BY-EBCDIC-CONTROL
```

2 Introduction to the COBOL language

2.1 Glossary

This section contains definitions of the terms used to describe the COBOL language in this manual. These terms do not necessarily have the same meaning for other programming languages.

The definitions are brief summaries of basic characteristics. For detailed explanations and syntax rules consult the later chapters of this manual.

Access mode

The manner in which records are to be operated upon within a file.

Activated runtime element

A program or method that is identified in a CALL or INVOKE statement and forms a run unit with the calling division at runtime.

Activating runtime element

A program or method that contains the calling statement.

Actual decimal point

The physical representation, using either of the decimal point characters period (.) or comma (,), of the decimal point position in a data item.

Address

Addresses can apply to data or programs.

Alphabetic character

A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z and space.

Alphabet-name

A user-defined name, in the SPECIAL-NAMES paragraph of the Environment Division, that assigns a name to a specific character set and/or collating sequence.

Alphanumeric character

A character represented with the EBCDIC character set, irrespective of whether it has pictorial representation.

Alphanumeric group item

Every group item with the exception of national or strongly typed group items.

Alternate record key

A key, other than the prime record key, whose contents identify a record within an indexed file.

Argument

An identifier, a literal, or an arithmetic expression that specifies a value to be used in the evaluation of a function.

Arithmetic expression

An arithmetic expression can be:

- an identifier for a numeric elementary item
- a numeric literal
- two arithmetic expressions separated by an arithmetic operator
- an arithmetic expression enclosed in parentheses.

Arithmetic operator

A single character or a fixed two-character combination which belongs to the following set:

Character	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponentiation

Ascending key

A key upon the values of which data is ordered starting with the lowest value of key up to the highest value of key in accordance with the rules for comparing data items.

Assumed decimal point

A decimal point position which does not involve the existence of an actual character in a data item. The assumed decimal point has logical meaning but no physical representation.

At end condition

An at end condition may occur:

1. During execution of a sequential READ statement for a file.
2. During execution of a RETURN statement whenever there is no logical record for the sort or merge file.
3. During execution of a SEARCH statement whenever the search terminates before any of the WHEN conditions have been satisfied.

Binary search

A method of searching a table in ascending or descending order for a particular element. The search takes place by a process of halving the searched area. At each stage of the search, the middle element is compared to see whether it is greater than, less than, or equal to the element being sought. This process of halving and comparing continues until the checked element is identical to the element being sought.

Block

A physical unit of data that is normally composed of one or more logical records or a portion of a logical record. The size of a block has no direct relationship to the size of the file within which the block is contained or to the size of the logical record(s) that are either contained within the block or that overlap the block. The term is synonymous with physical record.

Body group

Generic name for a report group, control heading or control footing.

Character

The basic indivisible unit of the language.

Character-string

A sequence of contiguous characters which form a COBOL word, a literal, a PICTURE character-string, or a comment-entry.

Class (object-oriented)

A a set of objects with their attributes and methods, as described by the class definition.

Class condition

The class condition establishes whether the contents of a data item are

- completely numeric,
- completely alphabetic,
- completely uppercase,
- completely lowercase, or
- completely made up of characters defined by means of the class-name specified in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION.

Class definition (object-oriented)

A compilation unit that defines a class of objects.

Class-name (object-oriented)

A user-defined word that identifies a class.

Class-name

A user-defined word specified in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION and naming a character set defined by the user. The class-name is entered in the class condition for purposes of checking whether a data item consists entirely of characters from this character set.

Clause

A clause is an ordered set of consecutive COBOL character strings whose purpose is to specify an attribute of an entry.

COBOL character set

The set of characters with which the syntax of a COBOL compilation group can be written, with the exception of comments and the content of non-hexadecimal alphanumeric and non-hexadecimal national literals.

COBOL word

see "Word"

Collating sequence

The sequence in which the characters that are acceptable in a computer are ordered for purposes of sorting, merging and comparing.

Column

A character position within a print line. The columns are numbered from 1, starting at the leftmost character position of the print line and extending to the rightmost position of the print line.

Combined condition

A condition that is the result of connecting two or more conditions with the "AND" or the "OR" logical operator.

Common program

A contained program in a nested source unit, whose name is provided with the COMMON attribute. Such a program can be called by the directly superordinate program and also by any "sibling program" or its "descendants".

Compilation group

A set of compilation units that are compiled together.

Compilation unit

A source unit that is not nested within other source units (program prototype, program definition, class definition and interface definition). These units form the elements of a compilation group and can be compiled separately.

Compile time

The time taken by the compiler to compile a compilation unit.

Compiler directing statement

A statement, beginning with a compiler directing verb, that causes the compiler to take a specific action during compilation. The compiler directing statements are COPY, REPLACE and USE.

Complex condition

A condition in which one or more logical operators act upon one or more conditions.

Computer-name

A system-name that identifies the computer upon which the program is to be compiled or run.

Condition

A status of a program at run time for which a truth value can be determined. In this manual, the term "condition" (condition-1, condition-2, ...) represents either a simple condition or a combined condition consisting of the syntactically correct combination of simple conditions, logical operators, and parentheses, for which a truth value can be determined.

Condition-name

A user-defined word assigned to a specific value, set of values, or range of values, within the complete set of values that a conditional variable may possess; or the user-defined word assigned to a status of a task switch or a user switch.

Condition-name condition

Causes a conditional variable to be tested to see whether its value matches any of the values belonging to a condition-name.

Conditional expression

A simple condition or a complex condition specified in an IF, PERFORM, EVALUATE or SEARCH statement.

Conditional statement

A conditional statement specifies that the truth value of a condition is to be determined and that the subsequent action of the object program is dependent on this truth value.

Conditional variable

A data item whose value or values is or are assigned a condition name.

Conformance

for objects:

A feature that enables an object with a given interface A to also be used where an object with some other interface B is expected. The conformance between the interfaces ensures that every operation of interface B is also supported by the conforming interface A.

for parameters:

The requirements for current and corresponding formal parameters as well as returning parameters in calling and called run units.

Connective

A reserved word that is used to:

- associate a data-name, paragraph-name, condition-name, or text-name with its qualifier
- link two or more operands written in a series

- form conditions (logical connectives); see "Logical operator"

Contiguous items

Items that are described by consecutive entries in the Data Division, and that bear a definite hierarchic relationship to each other.

Control break

A change in the value of a data item that is referenced in the CONTROL clause.

More generally, a change in the value of a data item that is used to control the hierarchical structure of a report.

Control break level

The relative position within a control hierarchy at which the most major control break occurred.

Control data item

A data item, a change in whose contents may produce a control break.

Control data-name

A data-name that appears in a CONTROL clause and refers to a control data item.

Control footing

A report group that is presented at the end of the control group of which it is a member.

Control group

A contiguous set of data assigned to a control data item within the control hierarchy.

For a given control data item, the control group consists of the entire sequence of control headings, control footings, and their associated report groups.

Control heading

A report group that is presented at the beginning of the control group of which it is a member.

Control hierarchy

A designated sequence of report subdivisions defined by the positional order of FINAL and the data-names within a CONTROL clause.

Conversion

The implicit transformation of numeric values from one format to another, or of index values into table element numbers and vice versa.

- In the case of index values (binary numbers) and table element occurrence numbers, transformation occurs according to the formula:
  index value = (occurrence number - 1) * length of table element

  Hence, conversion depends on the table used.

- In cases where USAGEs vary from one numeric data item to another.

Counter

A data item used for storing numbers or number representations in a manner that permits these numbers to be increased or decreased by the value of another number, or to be changed or reset to zero or to an arbitrary positive or negative value.

Currency symbol

The character defined by the CURRENCY SIGN clause in the SPECIAL-NAMES paragraph. If no CURRENCY SIGN clause is present in a COBOL compilation unit, the currency symbol is identical to the currency sign ($).

Current record

The record which is available in the record area of a file.

Current record pointer

A pointer that is used in the selection of the next record.

Data-address

A data address is a conceptual data unit that identifies the storage location of an item of data. A data address can be stored in a data pointer.

Data clause

A clause that appears in a data description entry in the Data Division and provides information describing a particular attribute of a data item.

Data description entry

An entry in the Data Division that is composed of a level-number followed by a data-name, if required, and then followed by a set of data clauses, as required.

Data item

A unit of data (excluding literals) defined by a COBOL program or by the rules for function evaluation.

Data-name

A user-defined word that names a data item described in a data description entry in the Data Division. When used in the general formats, "data-name" represents a word which cannot be subscripted, indexed or qualified unless specifically permitted by the rules for that format.

Data pointer

A data pointer is a data element in which a data address is stored.

Debugging line

A debugging line is any line with "D" in its indicator area (column 7 in the COBOL fixed form reference format).

Declaratives

A set of one or more sections, written at the beginning of the Procedure Division, the first of which is preceded by the key word DECLARATIVES and the last of which is followed by the key words END DECLARATIVES. A declarative is composed of a section header, followed by a USE compiler directing sentence, followed by a set of zero, one or more associated paragraphs.

Declarative sentence

A compiler-directing sentence consisting of a single USE statement terminated by the separator period.

De-editing

Ascertaining the numeric value of numeric-edited data.

Delimited scope statement

Any statement which includes its explicit scope terminator.

Delimiter

A character or a sequence of contiguous characters that identify the end of a string of characters and separates that string of characters from the following string of characters. A delimiter is not part of the string of characters that it delimits.

Descending key

A key upon the values of which data is ordered starting with the highest value of key down to the lowest value of key, in accordance with the rules for comparing data items.

Direct indexing

With direct indexing, the index used is in the form of a direct subscript.

see "Direct subscripting"

Direct subscripting

With direct subscripting, the subscript is indicated either as an integral literal or as a data-name described as a numeric elementary item with no character positions to the right of the assumed decimal point.

Division

A set of zero, one, or more sections or paragraphs, called the division body, that are formed and combined in accordance with a specific set of rules. There are four (4) divisions in a COBOL program: Identification, Environment, Data and Procedure.

Division header

A combination of words followed by a period and a space that indicates the beginning of a division. The division headers are:

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
DATA DIVISION.
PROCEDURE DIVISION [USING {dataname-1]...].
```

Dynamic access

The method of switching between sequential and random access. This method can be specified for relative or indexed files only.

Editing character

A single character or a fixed two-character combination belonging to the following set:

Character	Meaning
B	Space
0	Zero
+	Plus
-	Minus
CR	Credit
DB	Debit
Z	Zero suppress
*	Check protect
$	Currency sign
,	Comma (decimal point)
.	Period (decimal point)
/	Stroke (virgule, slash)

Elementary item

A data item with no further logical subdivisions.

Element Position Vector (EPV)

A logical information unit which records which node from the tree presentation of an XML document is assigned to each data item for which an IDENTIFIED clause is specified.

End program header

An entry that indicates the end of a COBOL program. The end program header consists of the keywords END PROGRAM, followed by the program name and a terminating period.

Entry

Any descriptive set of consecutive clauses terminated by a period and written in the IDENTIFICATION DIVISION, ENVIRONMENT DIVISION, or DATA DIVISION of a COBOL compilation unit.

Exception

A situation at runtime which indicates that an error or a deviation from normal execution has occurred.

Exception condition

A triggered exception situation.

Exception name

Word which names an exception condition.

Execution time

The time at which an object program is executed.

Explicit scope terminator

A reserved word which terminates the scope of a particular Procedure Division statement.

Extend mode

The state of a file after execution of an OPEN statement, with the EXTEND phrase specified, for that file and before the execution of a CLOSE statement for that file.

Extended access

A method of switching to and from sequential and random access. This access method may only be specified for indexed files.

External data item

A data item which is described as part of an external record in one or more programs of a run unit and which itself may be referenced from any program in which it is described.

External data record

A logical record which is described in one or more programs of a run unit and whose constituent data items may be referenced from any program in which they are described.

External repository

An external library containing the interface descriptions of programs, classes and other interfaces.

Factory definition

The source unit that describes a factory object.

Factory object

Every class has only one factory object, which produces all other objects of the class. The factory object is specified by the factory definition of a class.

File

A collection of records.

File clause

A clause that appears as part of any of the following Data Division entries:

File description (FD)

Sort-merge file description (SD)

Report description (RD)

File connector

The storage area which contains information about a file and is used as the linkage between a file-name and a physical file and between a file-name and its associated record area.

File description entry

An entry in the FILE SECTION of the Data Division that is composed of the level indicator FD, followed by a file-name, and then followed by a set of file clauses as required.

File name

A user-defined word that names a file described in a file description entry or a sort-merge file description entry within the FILE SECTION of the Data Division.

File organization

The permanent logical file structure established at the time that a file is created.

File position indicator

A logical unit of information which contains a positional description of the record which is to be accessed on the next READ statement. If no such record exists, the file position indicator informs you of why the record does not exist, i.e. why it is “invalid”. The file position indicator is modified only by the statements CLOSE, OPEN, READ and START.

Format

A specific arrangement of character-strings and separators within a statement or clause.

Function

A temporary data item whose value is determined by invoking a mechanism provided by the implementor at the time the function is referenced during the execution of a statement.

Global name

A name that is declared in only one program but which can be referenced by any program contained directly or indirectly in this program. Global names can be: condition-names, data-names, file-names, record-names, report-names, type-names as well as certain special registers.

Hexadecimal digit

Character from the range of values 0..9, A..F and a..f

High order end

The leftmost character of a string of characters.

I-O mode

The state of a file after execution of an OPEN statement, with the I-O specified, for that file and before the execution of a CLOSE statement for that file.

I-O status

A value moved to a two-character data item to inform the COBOL program of the status of an input-output operation. This value is only moved if the FILE STATUS clause is specified in the FILE-CONTROL paragraph.

Identifier

A syntactically correct combination of character-strings and separators that names a data item, i.e. a combination of a data-name, with the appropriate qualifiers, subscripts, and reference modifiers, as required for uniqueness of reference. Identifiers of (intrinsic) functions are described separately under the term "Function-identifier".

Imperative statement

A statement that either begins with an imperative verb and specifies an unconditional action to be taken or is a conditional statement that is delimited by its explicit scope terminator (delimited scope statement). An imperative statement may consist of a sequence of imperative statements.

Implementor-name

A name taken from the following list:

CONSOLE*)	Literal
TERMINAL*)	Job variable name
SYSIPT*)	TSW-0 to TSW-31
PRINTER, PRINTER01-PRINTER99	USW-0 to USW-31
SYSOPT*)	COMPILER-INFO
ARGUMENT-NUMBER *)	CPU-TIME
ARGUMENT-NAME*)	PROCESS-INFO
ENVIRONMENT-NAME *)	TERMINAL-INFO
ENVIRONMENT-VALUE*)	DATE-ISO4
C01 to C08; C10, C11	

*) = "reserved words" within the ENVIRONMENT DIVISION.

Index

A computer storage area or register, the contents of which represent the identification of a particular element in a table.

Index data item

A data item in which the value associated with an index-name can be stored.

Index-name

A user-defined word that names an index associated with a specific table.

Indexed data-name

An identifier that is composed of a data-name, followed by one or more index-names enclosed in parentheses.

Indexed file

A file with indexed organization.

Indexed organization

The permanent logical file structure in which each record is identified by the value of one or more keys within that record.

Indicator area

Column 7 in the COBOL reference format.

Inheritance

for classes:

A mechanism by which the interface and implementation (code) of one or more classes can be used as a basis for another class. This subclass inherits from one or more superclasses. The interface of an inheriting class conforms to the interface of the inherited classes.

for interfaces:

A mechanism by which the specification of one or more interfaces serves as the basis for another interface. An inheriting interface conforms to the inherited interface.

Initial program

A program that is in the initial state whenever it is called within a run unit.

Initial state

The state of a program when it is first called within a run unit.

Input file

A file that is opened in the input mode.

Input mode

The state of a file after execution of an OPEN statement, with the INPUT phrase specified, for that file and before the execution of a CLOSE statement for that file.

Input-output file

A file that is opened in the I-O mode.

Input procedure

A set of statements that is executed each time a record is released to the sort file.

Integer

A numeric literal or a numeric data item that does not include any character positions to the right of the assumed decimal point.

Where the term "integer" appears in general formats, "integer" must be a numeric literal which is an integer, and it must be neither signed nor zero unless explicitly allowed by the rules for that format.

Integer function

A function whose category is numeric and whose return value to the right of the decimal point is always simply the digit 0 irrespectively of the value.

Interface (for method or program)

The information needed to call a method or a program, i.e., the name of the method or program, the defined order of the parameters with the associated definition and passing technique, and the returning parameter (if any) and its description.

Interface (language construct)

See interface

Interface definition

The source unit that defines an interface.

Internal data

The data that is described in a program, excluding all external data items and external files. Data-items that are defined in the LINKAGE SECTION of a program are treated as internal data.

Internal data item

A data item that is described in a program of a run unit. An internal data item can have a global name.

Internal file

A file that can only be accessed by a program of the run unit.

Invalid key condition

A condition occurring at the time of program execution when a particular value for a key of a relative or indexed file is invalid.

Key

A data item which identifies the location of one or more data items which serve to identify the ordering of data.

Key of reference

The key, either prime or alternate, currently being used to access records within an indexed file.

Key word

A reserved word or function-name whose presence is required when the format in which the word appears is used in a compilation unit.

Last exception status

The last exception status triggered in a run unit. This can also be the status "no exception status triggered".

Level indicator

Two alphabetic characters (FD, RD, SD, DB) that identify a specific type of file.

Level-numbers

A one or two digit number which, in the range 1 through 49, indicates the position of a data item in the hierarchical structure of a logical record or which, in the case of level-numbers 66, 77 and 88, identifies special properties of a data description entry.

Library-name

A user-defined word that identifies a compilation unit library, which may contain more than one COBOL text with various names.

Library-text

A sequence of character-strings and/or separators in a COBOL library.

Line

see "Report line"

Line number

An integer that denotes the vertical position of a report line on a page.

Line sequential organization

A sequential file organization derived from the X/Open standard.

Logical operator

One of the reserved words AND, OR, or NOT.

In the formation of a combined condition, AND and OR can be used as logical connectives. NOT can be used for logical negation.

Logical record

A data item at the highest level in the hierarchy (level-number 01) which does not occur in any other record.

Low order end

The rightmost character of a string of characters.

Mass storage

A storage medium on which data may be organized and maintained in both a sequential and nonsequential manner.

Mass storage file

A collection of records that is assigned to a mass storage medium.

Merge file

A collection of records to be merged by a MERGE statement. The merge file is created and can be used only by the merge function.

Method

Executable statements in the Procedure Division of a method definition.

Method definition

The source unit that defines a method.

Method-name

A user-defined word that identifies a method.

Mnemonic-name

A user-defined word that is associated in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION with a specified implementor-name.

National character

A character which is represented with the UTF-16 character set.

Native character set

In accordance with the character sets used to represent characters, there are:

- EBCDIC character set for representing alphanumeric characters (in 1 byte)
- UTF-16 character set for representing national characters (in 2 bytes)

Native collating sequence

In accordance with the character sets used to represent characters, there are 2 collating sequences:

- alphanumeric collating sequence in 1 byte
- national collating sequence in 2 bytes

In both cases the sequence is based on the binary representation of the characters.

Negated combined condition

The "NOT" logical operator immediately followed by a parenthesized combined condition.

Negated simple condition

The "NOT" logical operator immediately followed by a simple condition.

Nested source program

A COBOL program that contains other programs which in turn can contain further programs. It accordingly comprises an outer program with one or more programs contained in it.

Next executable sentence

The next sentence to which control will be transferred after execution of the current statement is complete.

Next executable statement

The next statement to which control will be transferred after execution of the current statement is complete.

Next record

The record which logically follows the current record of a file.

Noncontiguous items

Elementary data items, in the WORKING-STORAGE, LOCAL-STORAGE and LINKAGE SECTIONs, which bear no hierarchic relationship to other data items.

Non-numeric item

A alphanumeric or national data item whose description permits its contents to be composed of any combination of characters taken from the computer’s character set. Certain categories of non-numeric items may be formed from more restricted character sets.

Non-numeric literal

An alphanumeric or national literal.

Numeric character

A character that belongs to the following set of digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

Numeric function

A function whose class and category are numeric.

Numeric item

A data item whose value is represented by the digits "0" through "9".

Its sign, if required, must be represented by a permitted form of "+" or "-".

Numeric literal

A literal composed of one or more numeric characters that may also contain either a decimal point, or an algebraic sign, or both. The decimal point must not be the rightmost character. The algebraic sign, if present, must be the leftmost character.

Object

An object is defined by a class and consists of a combination of data and methods that act upon that data.

Object definition

The source unit that defines an object.

Object program

The result in machine language of the compilation of a COBOL compilation unit and a linkage run.

Object reference

An implicitly or explicitly defined data item that contains a unique reference to an object.

Object time

The time at which an object program is executed.

Object view

An object view causes the compiler to treat an object reference as if it were defined in the specified form.

Open mode

The state of a file after execution of an OPEN statement for that file and before the execution of a CLOSE statement for that file.

The precise open mode is specified in the OPEN statement either with INPUT, OUTPUT, I-O or EXTEND.

Operand

In general, an operand may be defined as "that entity which is operated upon". However, in this publication, any lowercase word (or words) that appears in an entry, paragraph, clause or statement format may be considered an operand.

Operational sign

An algebraic sign, associated with a numeric data item or a numeric literal, to indicate whether its value is positive or negative.

Optional file

A file which is declared as being not necessarily present each time the object program is executed. The object program causes an interrogation for the presence or absence of the file.

Optional word

A reserved word that is included in a specific format only to improve the readability of the language and whose presence is optional to the user when the format in which the word appears is used in a compilation unit.

Output file

A file that is opened in either the output mode or extend mode.

Output mode

The state of a file after execution of an OPEN statement, with the OUTPUT or EXTEND phrase specified, for that file and before the execution of a CLOSE statement for that file.

Output procedure

A set of statements to which control is given during execution of a SORT statement after the sort function is completed, or during execution of a MERGE statement after the merge function has selected the next record in merged order.

Padding character

An alphanumeric character used to fill the unused character positions of a physical record.

Page

A vertical division of a report representing a physical separation of report data, the separation being based on internal reporting requirements and/or external characteristics of the reporting medium.

Page body

That part of the logical page in which lines can be written and/or spaced.

Page footing

A report group that is presented at the end of a report page and is output prior to a page advance whenever this is caused by a page advance condition.

Page heading

A report group that is presented at the beginning of a report page and is output immediately after page advance whenever this is caused by a page advance condition.

Paragraph

In the PROCEDURE DIVISION: a paragraph-name followed by a period and a space and by zero, one or more sentences.
In the IDENTIFICATION DIVISION and ENVIRONMENT DIVISION: a paragraph header followed by zero, one, or more entries.

Paragraph header

A reserved word placed above the paragraphs in the Identification and Environment Divisions for identification purposes.

The permissible paragraph headers are:

In the IDENTIFICATION DIVISION:	In the ENVIRONMENT DIVISION:
CLASS-ID.	SOURCE-COMPUTER.
METHOD-ID.	OBJECT-COMPUTER.
OBJECT.	SPECIAL-NAMES.
FACTORY.	REPOSITORY.
INTERFACE-ID.	FILE-CONTROL.
PROGRAM-ID.	I-O-CONTROL.

Paragraph-name

A user-defined word that identifies a paragraph in the Procedure Division.

Phrase

A phrase is an ordered set of one or more consecutive COBOL character-strings that form a portion of a COBOL procedural statement or of a COBOL clause.

Physical record

see "Block"

Pointer

Pointers can point to data or programs.

Predefined object reference

An implicitly-generated data item referenced by one of the identifiers NULL, SELF, or SUPER.

Prime record key

A key whose contents uniquely identify a record within an indexed file.

Printable group

A report group that contains at least one print line.

Printable item

A data item, the extent and contents of which are specified by an elementary report entry. This elementary report entry contains a COLUMN NUMBER clause, a PICTURE clause, and a SOURCE, SUM or VALUE clause.

Procedure

A paragraph or group of logically successive paragraphs, or a section or group of logically successive sections, within the Procedure Division.

Procedure name

A user-defined word which is used to name a paragraph or section in the Procedure Division. It consists of a paragraph-name (which may be qualified) or a section-name.

Program address

A program address identifies the storage location of a program. A program address can be stored in a program pointer.

Program identification area

Columns 73 through 80 in the COBOL fixed form reference format.

Program-name

A user-defined word that identifies a COBOL program.

Program-pointer

A program pointer is a data element in which the address of a program can be stored.

Program-text area

Columns 8 through 72 (inclusive) in the COBOL fixed form reference format.

Prototype

The definition of the interface of a method or program alone, i.e., without executablestatements and only with definitions that are needed in the interface.

Pseudo-text

A sequence of text words, comment lines, or the separator space in a compilation unit or COBOL library bounded by, but not including, pseudo-text delimiters.

Pseudo-text delimiter

Two contiguous equal sign (==) characters used to delimit pseudo-text.

Punctuation character

A character that belongs to the following set:

Character	Meaning
,	Comma
;	Semicolon
.	Period
:	Colon
"	Quotation mark
’	Apostrophe
(	Left parenthesis
)	Right parenthesis
'BLANK'	Space
=	Equal sign

Qualified data-name

An identifier that is composed of a data-name followed by one or more sets of either of the connectives OF and IN followed by a data-name qualifier.

Qualifier

1. A data-name used in a reference together with another data-name at a lower level in the same hierarchy.
2. A section-name used in a reference together with a paragraph-name specified in that section.
3. A library-name used in a reference together with a text-name associated with that library.

Raising of an exception condition

Transition of an exception situation into an exception condition:

1. a check of the exception situation activated by the TURN directive.
2. execution of a RAISE statement for this exception situation.

Random access

An access mode in which the program-specified value of a key data item identifies the logical record that is obtained from, deleted from, or placed into a relative or indexed file.

Record

see "Logical record"

Record area

A storage area allocated for the purpose of processing the record described in a record description entry in the FILE SECTION.

Record description entry

The total set of data description entries associated with a particular record.

Record key

A key, either the prime record key or an alternate record key, whose contents identify a record within an indexed file.

Record name

A user-defined word that names a record described in a record description entry in the Data Division.

Record number

The ordinal number of a record in the file whose organization is sequential.

Reference format

Standard method for describing the two possible statement formats in a COBOL compilation unit, namely fixed form reference format and free form reference format, fixed form reference format corresponding to the traditional COBOL reference format.

Reference modification

Definition of a data item through specification of the leftmost character position and the length of the data item.

Reference-modifier

A syntactically correct combination of character-strings and separators that defines a unique data item. Reference modifiers consist of a delimiting left parenthesis separator, the leftmost character position at which the data item begins, a colon separator, the length of the data item, and a delimiting right parenthesis separator.

Relation

see "Relational operator"

Relation character

A character that belongs to the following set:

Character	Meaning
>	Greater than
<	Less than
=	Equal to
>=	Greater than or equal to
<=	Less than or equal to

Relation condition

A condition which can yield a truth value. A relation condition causes two operands to be compared. Either of these operands may be an identifier, a literal, or an arithmetic expression.

Relational operator

A reserved word, a relation character, a group of consecutive reserved words, or a group of consecutive reserved words and relation characters used in the construction of a relation condition. The permissible operators and their meaning are:

Character	Meaning
IS [NOT] GREATER THAN IS [NOT] >	Greater than or not greater than
IS [NOT] LESS THAN IS [NOT] <	Less than or not less than
IS [NOT] EQUAL TO IS [NOT] =	Equal to or not equal to
IS GREATER THAN OR EQUAL TO IS >=	Greater than or equal to
IS LESS THAN OR EQUAL TO IS <=	Less than or equal to

Relative file

A file with relative organization.

Relative indexing

With relative indexing, the name of the table element is followed by an index in the form (index name +|- integer).

Relative key

A key whose contents identify a logical record in a relative file.

Relative organization

The permanent logical file structure in which each record is uniquely identified by an integer value greater than zero, which specifies the record’s logical ordinal position in the file.

Relative record number

The ordinal number of a record in a file whose organization is relative. This number is treated as a numeric literal which is an integer.

Relative subscripting

With relative subscripting, the name of the table element is followed by a subscript in the form

```
(data-name + integer) or
(data-name - integer).
```

Report clause

A clause, in the REPORT SECTION of the Data Division, that appears in a report description entry or a report group description entry.

Report description entry

An entry in the REPORT SECTION of the Data Division that is composed of the level indicator RD, followed by a report name, followed by a set of report clauses as required.

Report file

An output file whose file description entry contains a report clause. The contents of a report file consist of records that are written under control of the Report Writer Control System.

Report footing

A report group that is presented only at the end of a report.

Report group

In the REPORT SECTION of the Data Division, a 01-level entry and its subordinate entries.

Report group description entry

An entry in the REPORT SECTION of the Data Division that is composed of the level-number 01, the optional data-name, a TYPE clause, and an optional set of report clauses.

Report heading

A report group that is presented only at the beginning of a report.

Report line

A division of a page representing one row of horizontal character positions.

Report-name

A user-defined word that names a report described in a report description entry within the REPORT SECTION of the Data Division.

Report Writer logical record

A record that consists of the Report Writer print line and associated control information necessary for its selection and vertical positioning.

Reserved word

A COBOL word which is specified in the list of reserved words and which may be used in the COBOL compilation unit in accordance with the formats and rules, but which must not appear in the programs as user-defined words or system names.

Restricted data pointer

A data pointer which is restricted to data of a particular type.

Run unit

A specific set of object programs that function as a unit at execution time.

Section

A section comprises a set of paragraphs or clauses. The contents are preceded by a section header. A section can be empty or contain one or more paragraphs.

Section header

A combination of words followed by a period and a space that indicates the beginning of a section in the ENVIRONMENT DIVISION, DATA DIVISION and PROCEDURE DIVISION. In the ENVIRONMENT DIVISION and DATA DIVISION, a section header is composed of reserved words followed by a period and a space.

The permissible section headers are:

In the Environment Division:

```
CONFIGURATION SECTION.
INPUT-OUTPUT SECTION.
```

In the Data Division:

```
FILE SECTION.
WORKING-STORAGE SECTION.
LOCAL-STORAGE SECTION
LINKAGE SECTION.
REPORT SECTION.
SUB-SCHEMA SECTION.
```

In the Procedure Division, a section header is composed of a section-name, followed by the reserved word SECTION, followed by a segment-number (optional), followed by a period and a space.

Section-name

A user-defined word which names a section in the PROCEDURE DIVISION.

Segment-number

A user-defined word which classifies sections in the Procedure Division for purposes of segmentation. Segment-numbers may contain only the characters "0", "1", ..., "9". A segment-number may be expressed either as a one or two digit number.

Sentence

A sequence of one or more statements, the last of which is terminated by a period followed by a space.

Separator

A character used to separate character-strings.

Sequence number area

Columns 1 through 6 in the COBOL fixed form reference format.

Sequential access

An access mode in which logical records are obtained from or placed into a file in a consecutive predecessor-to-successor sequence determined by the order of records in the file.

Sequential file

A file with sequential organization.

Sequential organization

A permanent logical file structure in which the records are arranged and read in the same order in which they were created.

Sign condition

The proposition, for which a truth value can be determined, that the algebraic value of a data item or an arithmetic expression is either less than, greater than, or equal to zero.

Simple condition

Any single condition chosen from the set:
Relation condition
Class condition
Condition-name condition
Switch-status condition
Sign condition

Sort file

A collection of records to be sorted by a SORT statement.
The sort file is created and can be used by the sort function only.

Sort-merge file description entry

An entry in the FILE SECTION of the Data Division that is composed of the level indicator SD, followed by a file-name, and then followed by a set of file clauses as required.

Source unit

A sequence of statements that begins with an Identification Division and ends with an associated END entry (may be nested, see also “Compilation unit”).

Special character

A character that belongs to the following set:

Character	Meaning	Character	Meaning
+	Plus sign	.	Period (decimal point)
-	Minus sign	:	Colon
*	Asterisk	"	Quotation mark
/	Stroke (virgule, slash)	’	Apostrophe
=	Equal sign	(	Left parenthesis
$	Currency sign	)	Right parenthesis
,	Comma (decimal point)	>	Greater than symbol
;	Semicolon	<	Less than symbol

Special-character word

A reserved word which is an arithmetic operator or a relation character.

Special registers

Compiler generated storage areas whose primary use is to store information produced in conjunction with the use of specific COBOL features.

Statement

A syntactically valid combination of words and symbols written in the Procedure Division beginning with a verb.

Strongly typed

The word STRONG is specified in the definition of the associated type.

Subclass

A class that inherits from another class.

Subprogram

A runtime element invoked by a CALL statement.

Subscript

An integer, a data-name or an arithmetic expression whose value identifies a particular element in a table or one of the data items subordinate to this element.
A subscript may be the word ALL when the subscripted identifier is used as a function argument.

Subscripted data-name

An identifier that is composed of a data-name followed by one or more subscripts enclosed in parentheses.

Sum counter

A signed numeric data item established by a SUM clause in the REPORT SECTION of the Data Division. The sum counter is used by the Report Writer in connection with summing operations.

Superclass

A class that is inherited by another class.

Switch-status condition

A condition which indicates whether a user or task switch has been set to "on" or "off". The test is positive if the status of the switch corresponds to the setting given in the condition-name.

Symbolic character

A user-defined word indicating a figurative constant defined by the user.

System name

A COBOL word which is used to communicate with the operating system.

Table

A set of logically consecutive items of data that are defined in the Data Division by means of the OCCURS clause.

Table element

A data item that belongs to the set of repeated items comprising a table.

Text name

A user-defined word which identifies library text.

Text-word

A character or sequence of contiguous characters in the program text area in a COPY library or in a compilation unit. Any of the following are text-words:

1. Separators, except for: space, pseudo-text delimiters, and the opening and closing literal delimiters. The colon and the opening or closing parenthesis are always treated as separators when used outside alphanumeric or national literals.
2. Alphanumeric and national literals including their literal delimiters.
3. Any other sequence of characters delimited by separators, except comments and the word "COPY".

Truth value

The representation of the result of the evaluation of a condition in terms of one of two values, "true" or "false".

Type

A pattern which contains all characteristics of the data description entry and its subordinate data items.

Type name

A user-defined name identifying a type that is described in the data description entry of the DATA DIVISION.

Unary operator

A plus (+) or a minus (-) sign, which precedes a variable or a left parenthesis in an arithmetic expression and which has the effect of multiplying the expression by +1 or -1 respectively.

User-defined word

A COBOL word that must be supplied by the user to satisfy the format of a clause or statement.

Variable

A data item whose value may be changed during execution of the object program. A variable used in an arithmetic expression must be a numeric elementary item.

Verb

A COBOL word that causes an action to be taken by the COBOL compiler and the object program.

Word

A character-string of not more than 31 characters which forms a user-defined word, a system-name, a reserved word, or a function-name.

XML file

A file or a memory area which contains an XML document

Zero-length item

A data item whose minimum permitted length is 0 and whose length at runtime is 0.

2.2 COBOL notation

1. Definition of a format

The specific arrangement of the elements within a clause or statement is referred to as a "general format". A clause or statement may be composed of various element types.

When more than one specific arrangement is permitted in a clause or statement, the general format is subdivided into numbered formats. Note that clauses must be written in the same sequence in which they are specified in the general format. In certain exceptional cases, departures from this rule are allowed. These cases, however, are identified as such.

The proper use of formats, the necessary application prerequisites, and the restrictions on their use are expressed in the form of rules.

2. Elements

Clauses or statements may be constructed from the following element types:

- uppercase words
- lowercase words
- uppercase and lowercase words
- level-numbers
- brackets
- braces
- connectives
- special characters

3. Words

Notation	Meaning
Uppercase	A word specially reserved for COBOL.
Uppercase, underlined	This word must be specified by the programmer as it is given in the format. It is a COBOL keyword.
Uppercase, not underlined	This word may be specified by the programmer at the location given in the format or it may be omitted. It is an optional COBOL word.
Lowercase	Generic term used to represent COBOL words, literals, picture-strings, comments, or a complete syntactical unit. It must be entered by the programmer at the location given in the format. If more than one generic term of the same kind occurs in the same format, an appended number or letter is used to uniquely qualify that term for the descriptions.

Table 1: Notation used for COBOL words

An entry consisting of one or more words in uppercase followed by the words "clause" or "statement" designates a clause or statement described elsewhere in this manual. In programs, all COBOL words can appear in uppercase and lowercase as well as in lowercase only.

4. Separators

The separators listed in the following table must be used as specified in the format.

Character	Meaning	Character	Meaning
'BLANK'	Space	"	Quotation marks *)
,	Comma	’	Apostrophe
;	Semicolon	(	Open parentheses
.	Period	)	Close parentheses
:	Colon	==	Pseudo-text delimiter

Table 2: Separators

The rules governing the use of separators are described in section "Separators"

5. Level indicators and level numbers

Level indicators and level numbers which occur in the format must be supplied at the appropriate point in the COBOL compilation unit. This manual uses the form 01, 02, ..., 09 to indicate level numbers 1, 2, ..., 9.

6. Brackets [ ]

A format specification placed in square brackets may be supplied or omitted at the option of the user. If two or more items are stacked within brackets, one or none of them may be specified.

7. Braces { }

If two or more items are stacked within braces, one of the enclosed items is required. If there is only one item, the braces perform only a combining function for a subsequent ellipsis (repetition symbol).

8. Vertical bars | |

The vertical bars are enclosed in either brackets or braces and have the following meaning:

1. Vertical bars within braces enclose optional entries. Here at least one entry must be selected, but it is also possible to select more than one. The order of the entries selected is freely definable. However, each alternative may only be used once.
2. Vertical bars within brackets enclose optional entries. Here the specification can be omitted, or more than one of the specified values can be selected. The order of the entries selected is freely definable. However, each alternative may only be used once.

9. Parentheses ( )

Format items appearing within parentheses refer to table item numbers (indices) which must be specified in order to differentiate the various items in a table.

10. Ellipsis ...

An ellipsis appearing in the text indicates the omission of one or more words when such an omission does not impair comprehension.

An ellipsis appearing in the format indicates that the immediately preceding unit may, if desired, be repeated any number of times after it has been specified once. A repeatable unit is either a single word or a group of words combined by brackets or braces. In the latter case, the ellipsis immediately follows the closing bracket or brace; the related opening bracket or brace determines the beginning of the unit to be repeated.

11. Space 'BLANK'

When used in examples and tables, this character stands for a space.

12. Special characters in formats

If the characters +, -, >, <, =, >= and <= appear in a format, they must be entered whenever the format is used. This applies even if these special characters are not underlined.

Example 2-1

```
ADD (1)   { (2) identifier-1 | literal-1 }(2)...(3)
          TO {identifier-2(4) [(5) ROUNDED ](5) }...
    [ON(6) SIZE ERROR imperative(7) statement-1]
    [NOT ON SIZE ERROR imperative statement-2]
    [END-ADD]
```

(1)	COBOL keyword: the indicated form is mandatory.
(2)	Braces: one of these options must be chosen.
(3)	Ellipsis: the preceding entry may be repeated any number of times.
(4)	Qualification: an appended number or letter is used to create a unique identification for an element.
(5)	Brackets: one or more of these options may be chosen.
(6)	Optional word: the word may be omitted, or specified for the sake of clarity.
(7)	Lowercase words: these must be entered by the programmer.

The following language elements are valid ADD statements derived from the above format; commas and semicolons are included for better readability.

```
ADD I TO J
ADD I-1, I-2, I-3 TO I-4 ROUNDED
ADD 1 TO I-1, I-2 ROUNDED, I-3
ADD I-1 TO I-2; SIZE ERROR PERFORM ADD-ERR END ADD
```

2.3 Reference format

The current COBOL standard permits two reference formats. The traditional reference format is referred to in the following as fixed form reference format or "fixed format" for short. The new format is the free form reference format, or "free format" for short. You switch between these two reference formats using the >>SOURCE FORMAT directive.

The following applies for both reference formats:
The following may not be separated but must be written in a separate line:

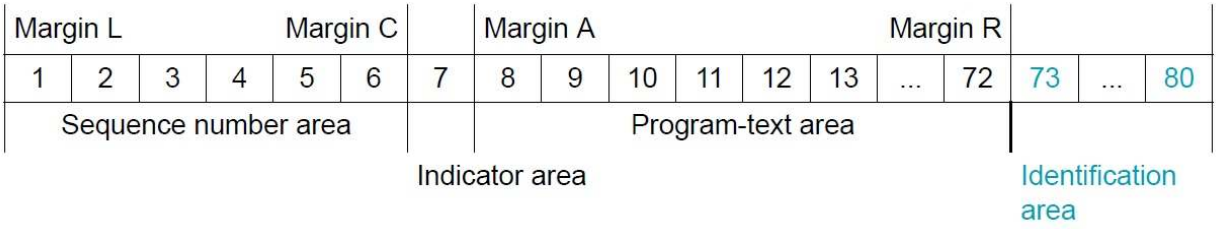
- Division headers
  - IDENTIFICATION DIVISION,
  - ENVIRONMENT DIVISION,
  - DATA DIVISION,
  - PROCEDURE DIVISION,
- Section headers
  - REPORT SECTION,
  - SUB-SCHEMA SECTION,
- PROGRAM-ID,
- CLASS-ID,
- INTERFACE-ID,
- METHOD-ID,
- paragraphs
  - FACTORY,
  - OBJECT,
- END entries.

A COBOL program must be written in the native alphanumeric character set.

2.3.1 General description of the fixed form reference format

The standardized fixed format for writing COBOL programs can be described in terms of a line consisting of 80 character positions. The compiler only accepts COBOL programs written in the reference format and generates a listing of the program in the same format.

A line is divided as follows:



Margin L
is located to the left of the leftmost character position in a line.

Margin C
is located between the sixth and seventh character position in a line.

Margin A
is located between the seventh and eight character position in a line.

Margin R
is located to the right of the rightmost character position in a line that is still relevant for the compiler.

Sequence number area
contains six character positions (columns 1 to 6) located between Margin L and Margin C.

Indicator area
the seventh character position in a line.

Program-text area
contains the character positions 12 through 72 and is located between Margin A and Margin R.

2.3.2 Rules for using the fixed form reference format

- **Sequence number area (columns 1-6)**

  This field may be used to label lines of a COBOL program.

  The content of the sequence number area is defined by the user and may consist of any character in the computer's character set.

- **Indicator area (column 7)**

  This field is used to designate continuation, comment, and debugging lines.

  A hyphen (-) in this field signifies that this is a **continuation line**, i.e. the previous line is being continued (see "Continuation of lines" further below). The absence of a hyphen in the indicator field is taken to indicate that the last character in area B (see below) of the previous line is followed by one space character.

  An asterisk (*) supplied in this field indicates a **comment line** (see "Comment line" further below).

  A slash (/) in this field indicates a special kind of comment line, which causes a form feed to be carried out in the program listing before this line is printed.

  A letter D in this field designates a **debugging line** (see section "Debugging").

- **Program-text area (columns 8-72)**

  This area is reserved for the beginning of the headers of the four COBOL program divisions, of the section headers and paragraph headings, for level indicators, and for certain level numbers. It is used to hold all clauses and statements.

- **Identification area (columns 73-80)**

  This area may be used to assign names to lines of a COBOL program. It may contain any characters from the computer's character set or may be blank. Its contents are not evaluated by the compiler.

- **Continuation of lines**

  A sentence or entry requiring more than one line may be continued on subsequent lines in the program-text area. The first line is called a **continued line**, the following lines are called **continuation lines**. If a sentence or entry spans more than two lines then all lines, except the first and last, are both continued and continuation lines.

  A word, PICTURE character-string, or literal may be continued in the next line. If this occurs, the following apply:

  - Continuation of non-numeric literals

    If a non-numeric literal is continued on the next line, a hyphen should be entered in the indicator area (column 7) of the continuation line.
    The continuation of the literal may begin anywhere in the program text area and must be introduced by the same quotation mark as that used in the opening literal delimiter.
    All blanks located at the end of the continued line or following the delimiter of the continuation line or ahead of the delimiter which closes the literal are regarded as part of the literal.

  - Continuation of words and numeric literals

    If a word or a numeric literal is continued on the next line, a hyphen must be entered in the indicator area (column 7) of the continuation line in order to show that the first non-blank character in the program-text area of the continuation line is the immediate successor of the last non-blank character of the continued line, i.e. without any intervening space.

- **Blank lines**

  A blank line is a line that contains only spaces in columns 7 through 72 or no characters at all. A blank line may appear anywhere within a compilation unit.

- **Comment lines**

  Explanatory comments may be included anywhere in the COBOL source unit in the form of comment lines by setting an asterisk or a slash in the indicator area (column 7). Any combination of characters from the character set of the data processing system may be used in areas A or B of these lines. The contents of the comment lines will be produced on the program listing (at the top of a new page if a slash has been entered in the indicator area) and have no effect on the program.

- **Debugging lines**

  Debugging lines are indicated by a "D" in the indicator area (column 7) (see section "Debugging").

- **Pseudo-text**

  The pseudo-text, which consists of character strings and delimiters, lies in the program-text area. If the indicator area of a line following the opening pseudo-text delimiter contains a hyphen, it is possible to continue at any position of the program text line. Text words are continued in accordance with the normal rules for the continuation of lines.

- **Inline comments**

  The asterisk and the greater-than sign (*>) in the program-text area precede a comment that extends to the end of the current line. A comment like this can stand alone in a line, but it can also be preceded by source text. In the latter case, the string '*>' must be preceded by a separator.

  In source code manipulation, comments like this are handled in the same way as spaces.

  Lines with an inline comment cannot be continued.

2.3.3 General description of the free form reference format

In free format the program text may be written as required to a line. Only a few rules for comments and continuation lines must be borne in mind. The entire line thus constitutes the program-text area.

The maximum length of a line is limited to 248 characters.

2.3.4 Rules for using the free form reference format

- **Continuation of lines**

  Non-numeric literals can be continued over several lines. The line continued in this way is called a "continued line", the following lines are called "continuation lines".

  The literal to be continued must be terminated by a closing literal delimiter followed immediately by a hyphen. This may only be followed by blanks. The first character in the continuation line which is not a blank must be the same quotation mark as that used in the opening literal delimiter. The character following this is interpreted as the next character of the continued literal.
  At least one character of the actual literal must be contained in the continued line and in the continuation line.

  All characters belonging to an indicator or separator which consists of several characters must be contained in the same line. They may not be separated by blanks. A duplicated delimiter which represents a single delimiter within a literal must also be contained in one line.

- **Blank lines**

  A blank line is a line that contains only blanks or no characters at all. A blank line may appear anywhere within a compilation unit.

- **Comments**

  A comment consists of the comment indicator ("*>") followed by any text. All characters after the comment indicator up to the end of the line are comments.
  Any character from the computer’s character set may be used in the comment text. Comments are only used for documentary purposes and have no effect on the meaning of the compilation group.
  A comment can be either a comment line or an inline comment.

  - **Comment lines**

    The first character in a comment line is a comment indicator. A comment line may occur anywhere in a compilation group.

  - **Inline comments**

    A comment indicator which is preceded by blanks constitutes an inline comment. An inline comment can occur in any line of a compilation group except in continued lines.

2.4 Language concepts

2.4.1 COBOL character set

The basic linguistic unit is the character. The COBOL character set consists of 26 uppercase letters, 26 lowercase letters, 10 digits, the space character, and any special characters.

Character	Meaning
0 to 9	Digit
A to Z, a to z	Letters
'BLANK'	Space
+	Plus
-	Minus (hyphen)
*	Asterisk
/	Slash
=	Equal sign
$ or €	Currency sign (dollar sign/Euro sign)
,	Comma (decimal point)
;	Semicolon
.	Period (decimal point)
:	Colon
"	Quotation mark
’	Apostrophe
(	Left parenthesis
)	Right parenthesis
>	Greater than
<	Less than

Table 3: COBOL character set

When non-numeric literals, comments or comment lines are used, the character set is extended to comprise the whole alphanumeric character set of the data processing system.

The characters which are permitted for use with each type of string and as separators (delimiters) are defined in the subsections to follow.

2.4.2 Separators

A separator consists of one or more contiguous characters. Each of the following is a separator unless it occurs in a literal or a PICTURE character-string:

1. A space is a separator. Anywhere where a space can be used as a separator or as part of a separator, it is also possible to use more than one space. All spaces which follow the separators comma, semicolon or period are part of these separators (i.e. they are not treated as a space separator).
2. Commas and semicolons can only be used as separators when they are immediately followed by a space. They can be used to improve the readability of the program anywhere where a space could also be used as a separator.
3. A period can only be used as a separator when immediately followed by a space. It may only be used to indicate the end of a sentence, or as shown in formats.
4. Left (opening) and right (closing) parentheses are separators. When used outside of pseudo-text, they must appear as balanced pairs of left and right parentheses used to delimit subscripts, lists of function arguments, reference modifiers, arithmetic expressions, conditions or the repetition factor of a PICTURE symbol.
5. Opening and closing literal delimiters are separators. Either a single quote (') or a double quote (“) can be used as quotation marks in the literal delimiter.

   Opening literal delimiters:

   - a quotation mark
   - the 2 characters N', N", X' and X"
   - the 3 characters NX' and NX"

   Closing literal delimiters:

   - a single quote if the opening literal delimiter used a single quote
   - a double quote if the opening literal delimiter used a double quote

   An opening literal delimiter must be immediately preceded either by a space, an opening parenthesis or an opening pseudo-text delimiter; a closing literal delimiter must be immediately followed by one of the separators space, comma, semicolon, period, closing parenthesis or closing pseudo-text delimiter. Separators which immediately precede the opening literal delimiter or immediately follow the closing literal delimiter are not part of the separator.
6. Pseudo-text delimiters are separators. An opening pseudo-text delimiter must be immediately preceded by a space; a closing pseudo-text delimiter must be immediately followed by one of the separators space, comma, semicolon or period.
   Pseudo-text delimiters may only appear in balanced pairs delimiting pseudo-text.
7. A colon is a separator and must be specified when required in the general formats.
8. A space used as a separator may immediately precede all separators unless
   - the reference format rules prohibit it
   - it is followed by the closing literal delimiter; in this case, a preceding space is considered part of the non-numeric literal and not as a separator.
9. A space used as a separator may immediately follow any separator except the opening literal delimiter. A space following the opening literal delimiter is considered part of the non-numeric literal and not as a separator.

2.4.3 COBOL words

A word consists of 1-30 characters from the following set:

A-Z, a-z, 0-9, - (hyphen)

No distinction is made between uppercase and lowercase letters.

A word may neither begin nor end with a hyphen, must not contain space characters, and must contain at least one letter.

Words are divided into four categories:

- user-defined words
- system-names
- reserved words
- function-names

1. User-defined words

A user-defined word is a COBOL word to be supplied by the programmer according to the format for a clause or statement. It refers to particular units of data at object time. The following subsections describe the types of user-defined words employed in COBOL programs, and state the rules for writing these names.

The 17 types of user-defined words are listed and defined in table 4.

All user-defined words except segment-numbers and level-numbers must be made unique. Either there must be no other user-defined word in the compilation unit with the same sequence of characters and punctuation marks, or the word must be qualified.

With the exception of paragraph-name, section-name, level-number, and segment-number, all user-defined words must contain at least one alphabetic character. Segment-numbers or level-numbers may be identical to other segment-numbers or level-numbers, or to paragraph-names and section-names.

alphabet-name	An alphabetical name located in the SPECIAL-NAMES paragraph of the Environment Division and connected with a character set and/or collating sequence.
class-name (object-oriented)	A user-defined word that identifies a class.
class-name	A name entered by the user in the CLASS clause of the SPECIAL-NAMES paragraph in the Environment Division to define a character set. This classname can be referenced in the class condition.
condition-name	The name assigned to a specific value, set of values, or range of values which an elementary data item may assume (hence, a condition of the data item). A condition-name is defined by an 88-level entry in the FILE SECTION, LINKAGE SECTION , WORKING-STORAGE SECTION or LOCAL-STORAGE SECTION.
data-name	A name identifying a data item in the Data Division. A data-name is defined by its appearance in a data description entry.

file-name	A name assigned to a set of input data or output data. A file-name is defined by its appearance in the SELECT clause of the FILE CONTROL paragraph and its use as the name of an FD entry. A special file-name is a sort-file-name that names a sort-file. A sort-file-name is defined by its appearance in the SELECT clause of the FILE CONTROL paragraph and its use to name an SD entry in the FILE SECTION.
index-name	A name of an index for a particular table. An index name is declared by its occurrence in the INDEXED BY phrase of the OCCURS clause.
interface-name	A user-defined word that identifies an interface.
level-number	A level-number indicates the position of a data item in the hierarchical structure of a record or indicates special properties of a data description entry. Level-numbers are defined by their appearance in a data description entry.
library-name	A name of an entry in the COBOL compilation unit library, which may contain more than one text with various names.
method-name	A user-defined word that identifies a method. The method name is defined by its use in the METHOD-ID paragraph of the Identification Division.
mnemonic-name	A fixed name, provided the programmer associated it with a particular implementor-name in the SPECIAL-NAMES paragraph of the Environment Division.
paragraph-name	A paragraph-name is used to name a paragraph in the Procedure Division. Paragraph-names are written starting at Area A.
parameter-name	A parameter-name identifies a formal parameter of a parameterized class or of a parameterized interface.
program-name	The name used to identify the program. The program-name is defined by its use in the PROGRAM-ID paragraph of the Identification Division. It may also appear in a CALL statement of a corresponding calling program.
program-proto-type-name	A user-defined word that identifies a program prototype.
record-name	The name of a record. A record is declared by a 01-level entry in the FILE SECTION, LINKAGE SECTION, WORKING-STORAGE SECTION, LOCAL-STORAGE SECTION or in the SUB-SCHEMA SECTION.
report-name	The name of a report. A report-name is defined by its occurrence in the REPORT clause of an FD entry; it is used to name an RD entry in the REPORT SECTION.
section-name	A section-name is used to name a section in the Procedure Division. A section-name is written starting at Area A and is followed by the word SECTION.
segment-number	A number to classify sections in the Procedure Division for purposes of segmentation. It is defined by its use in a section header.

symbolic-character	A name for a figurative constant defined by the user character in the SYMBOLIC-CHARACTERS clause of the SPECIAL-NAMES paragraph.
text-name	Name of an entry in the COBOL compilation unit library. The entry is copied from the library by the COPY statement.
type-name	A user-defined name to identify a type which is described in a data description entry of the DATA DIVISION.

Table 4: COBOL user-defined words

2. System-names

A system-name is a COBOL word which is used as an interface with the operating system environment. System-names are defined by the implementor and may vary from compiler to compiler. From the programmer's point of view, the system-names of a specific compiler are treated as reserved words.

The system names for COBOL2000 are:

Computer-name	in the SOURCE-COMPUTER and OBJECT-COMPUTER paragraphs.
Implementor-name	in the SPECIAL-NAMES paragraph and the ASSIGN clause.
Call-convention	in the CALL-CONVENTION directive.

3. Reserved words

COBOL includes a fixed number of reserved words, the COBOL words.

A reserved word serves a specific purpose and must be used only in the context specified in the formats; it must not occur in the compilation unit as a user-defined word or system-name.

A complete list of reserved words is supplied on the pages below. All reserved words marked with an asterisk (*) in this list are treated as reserved words only if DML (Data Manipulation Language) statements are being used for compilation; otherwise they may be employed as user-defined words. Compilation with DML statements occurs when

SUB-SCHEMA SECTION

is specified in the Data Division of a program (see the "UDS/SQL Reference Manual" [6]).

There are three types of reserved words:

- Required words
- Optional words
- Special purpose words

- **Required words**

  A required word is a word whose presence is required when the format in which the word appears is used in a compilation unit.

  Required words are of two types:

  Keywords

  Within each format, such words are uppercase and underlined. Keywords are only allowed in the formats indicated. Keywords may be grouped as shown below:

  - Verbs such as ADD, READ and CALL.
  - Required words which are encountered in statement and entry formats.
  - Words which have a specific functional significance, such as NEGATIVE, SECTION, etc.

  Some keywords may be abbreviated (e.g. PIC for PICTURE).

  Special character words

  These are the arithmetic operators and relation characters (see section "Glossary").

- **Optional words**

  Within each format, uppercase words which are not underscored are called "optional words". These words may be used at the option of the user. The presence or omission of an optional word has no effect on the meaning of the COBOL statement. However, an optional word must not be misspelled or replaced with another word.

- **Special purpose words**

  There are two types of special purpose words:

  - special registers
  - figurative constants

Special registers

Special registers are data items in which information produced with the use of certain COBOL features is stored. The attributes of these registers are predefined, and each register has a fixed name. Thus, the programmer does not have to define these registers in the Data Division. The eleven special registers are listed in table 5.

Register name	Description	Use
TALLY	5-digit unsigned data item with COMPUTATIONAL phrase (see section "USAGE clause")	TALLY may be used wherever a data item with an integral value can occur. For example, if the current value of TALLY is 3, the following statements are equivalent: ADD 3 TO ALPHA. ADD TALLY TO ALPHA.
LINE-COUNTER PAGE-COUNTER PRINT-SWITCH CBL-CTR	Used by the Report Writer (see chapter "Report Writer").	See section "Special registers of the Report Writer".

LINAGE-COUNTER	A 4-byte data item containing an unsigned integer whose value is less than or equal to integer-1 or the data item referenced by dataname-1 in the LINAGE clause	A LINAGE-COUNTER register is generated by the compiler for each file whose file description entry contains a LINAGE-clause (see section "LINAGE clause").
RETURN-CODE	8-digit signed data item with COMPUTATIONAL and SYNCHRONIZED phrase (corresponds to PIC S9(8) COMP-5 SYNC).	This data item exists only once for each run unit. The user can use this item to exchange information between COBOL modules which were compiled separately but linked into a single object program. This item can also be used to store the return value of a non-COBOL subprogram. When a COBOL subprogram terminates, the contents of the item can be made available to the calling non-COBOL program as a function value. If the contents of the RETURN-CODE special register are not 0 after the execution of STOP RUN, the operating system is informed that the program terminated abnormally.
SORT-RETURN SORT-FILE-SIZE SORT-CORE-SIZE SORT-MODE-SIZE SORT-EOW SORT-CCSN	Used by the sort section (see section "Sorting records".	See section "Special registers for files: SORT".
XML-EVENT XML-CODE XML-TEXT XML-NTEXT XML-NAMESPACE XML-NNAMESPACE XML-NAMESPACE-PREFIX XML-NNAMESPACE-PREFIX	Used by the XML part (see section "Language elements for processing XML").	See section "Special registers for the XML PARSE statement".

Table 5: COBOL special registers

Reserved words

The following table contains all the **reserved words**. All words marked with * are treated as reserved words only if DML (Data Manipulation Language) statements are being used for compilation; otherwise they may be employed as user-defined words. Compilation with DML statements occurs when SUB-SCHEMA SECTION is specified.

A '#' before the word means that this word is not treated as a reserved word if ENABLE-KEYWORDS = *COBOL85 is set in the SOURCE-PROPERTIES option.

```
<                       #B-AND              COMMUNICATION       DE
<=                      #B-NOT              COMP                DEBUG-CONTENTS
+                       #B-OR               COMP-1              DEBUG-ITEM
*                       #B-XOR              COMP-2              DEBUG-LINE
**                      #BASED              COMP-3              DEBUG-NAME
-                       BEFORE              COMP-5              DEBUG-SUB-1
/                       BEGINNING           COMPUTATIONAL       DEBUG-SUB-2
>                       BINARY              COMPUTATIONAL-1     DEBUG-SUB-3
>=                      #BINARY-CHAR        COMPUTATIONAL-2     DEBUGGING
:                       #BINARY-DOUBLE      COMPUTATIONAL-3     DECIMAL-POINT
=                       #BINARY-LONG        COMPUTATIONAL-5     DECLARATIVES
ACCEPT                  #BINARY-SHORT       COMPUTE             #DEFAULT
ACCESS                  #BIT                #CONDITION          DELETE
#ACTIVE-CLASS           BLANK               CONFIGURATION       DELIMITED
ADD                     BLOCK               *CONNECT            DELIMITER
#ADDRESS                #BOOLEAN            #CONSTANT           DEPENDING
ADVANCING               BOTTOM              CONTAINS            DESCENDING
AFTER                   BY                  CONTENT             DETAIL
ALL                     BYTE-LENGTH         CONTINUE            DISABLE
#ALLOCATE                                   CONTROL             DISC
ALPHABET                CALL                CONTROLS            *DISCONNECT
ALPHABETIC              CANCEL              CONVERTING          DISPLAY
ALPHABETIC-LOWER        *CASE               COPY                DIVIDE
ALPHABETIC-UPPER        CBL-CTR             CORR                DIVISION
ALPHANUMERIC            CF                  CORRESPONDING       #DOCUMENT
ALPHANUMERIC-EDITED     CH                  COUNT               DOWN
                        CHARACTER           CREATING            *DUPLICATE
ALSO                    CHARACTERS          #CRT                DUPLICATES
ALTER                   CHECKING            CURRENCY            DYNAMIC
ALTERNATE               CLASS               *CURRENT
AND                     #CLASS-ID           #CURSOR             EBCDIC
ANY                     CLOCK-UNITS                             #EC
#ANYCASE                CLOSE               DATA                ELSE
ARE                     CODE                #DATA-POINTER       *EMPTY
AREA                    CODE-SET            *DATABASE-EXCEPTION ENABLE
AREAS                   #COL                                    END
#AS                     COLLATING           DATABASE-KEY        END-ACCEPT
ASCENDING               #COLS               DATABASE-KEY-LONG   END-ADD
ASSIGN                  COLUMN              DATE                END-CALL
AT                      #COLUMNS            DATE-COMPILED       END-COMPUTE
AUTHOR                  COMMA               DATE-WRITTEN        END-DELETE
                        COMMIT              DAY                 END-DISPLAY
                        COMMON              DAY-OF-WEEK         END-DIVIDE
                                            *DB

                        FIRST                                   MOVE
                        #FLOAT-EXTENDED                         MULTIPLE
```

END-EVALUATE
END-IF
#END-INVOKE
END-MULTIPLY
END-OF-PAGE
#END-OPEN
END-PERFORM
END-READ
END-RECEIVE
END-RETURN
END-REWRITE
END-SEARCH
END-START
END-STRING
END-SUBTRACT
END-UNSTRING
END-WRITE
#END-XML
ENDING
ENTRY
ENVIRONMENT
#EO
EOP
EQUAL
ERASE
ERROR
*ESCAPE
EVALUATE
EVERY
EXCEPTION
#EXCEPTION-OBJECT
*EXCLUSIVE
EXIT
EXTEND
EXTENDED
EXTERNAL

#FACTORY
FALSE
FD
*FETCH
FILE
FILE-CONTROL
FILLER
FINAL
*FIND
*FINISH
#FLOAT-LONG
#FLOAT-SHORT
FOOTING
FOR
#FORMAT
FREE
FROM
FUNCTION
#FUNCTION-ID

GENERATE
GET
GIVING
GLOBAL
GO
GOBACK
GREATER
GROUP
#GROUP-USAGE

HEADING
HIGH-VALUE
HIGH-VALUES

I-O
I-O-CONTROL
ID
IDENTIFICATION
#IDENTIFIED
IF
IGNORING
IN
*INCLUDING
INDEX
INDEXED
INDICATE
#INHERITS
INITIAL
INITIALIZE
INITIATE
INPUT
INPUT-OUTPUT
INSPECT
INSTALLATION
#INTERFACE
#INTERFACE-ID
INTO
INVALID
#INVOKE
IS

JUST
JUSTIFIED

*KEEP
KEY

LABEL
LAST
LEADING
LEFT
LENGTH
LESS
LIMIT
*LIMITED
LIMITS
LINAGE
LINE
LINE-COUNTER
LINES
LINKAGE
#LOCAL-STORAGE
#LOCALE
LOCK
LOW-VALUE
LOW-VALUES

*MASK
*MATCHING
*MEMBER
*MEMBERS
*MEMBERSHIP
MEMORY
MERGE
MESSAGE
#METHOD
#METHOD-ID
#MINUS
MODE
*MODIFY
MODULES
MORE-LABELS
MULTIPLY

#NATIONAL
#NATIONAL-EDITED
NATIVE
NEGATIVE
#NESTED
NEXT
NO
NOT
#NULL
NUMBER
NUMERIC
NUMERIC-EDITED

#OBJECT
OBJECT-COMPUTER
#OBJECT-REFERENCE

*OCCURENCE
OCCURS
OF
OFF
OMITTED
ON
OPEN
OPTIONAL
#OPTIONS
OR
ORDER
ORGANIZATION
OTHER
OUTPUT
OVERFLOW
#OVERRIDE
*OWNER

PACKED-DECIMAL
PADDING
PAGE
PAGE-COUNTER
PERFORM
*PERMANENT
PF
PH
PIC

```
TYPE
#TYPEDEF

UNIT
UNITS
#UNIVERSAL
#UNLOCK
UNSTRING
UNTIL
UP
*UPDATE
UPON
USAGE
*USAGE-MODE
USE
#USER-DEFAULT
USING

#VAL-STATUS
#VALID
#VALIDATE
#VALIDATE-STATUS
VALUE
VALUES
VARYING
#VERSION-XML
*VIA

WHEN
WITH
*WITHIN
WORDS
WORKING-STORAGE
WRITE

#XML

ZERO
ZEROES
ZEROS
```

PICTURE	REPORTS	*SORTED
PLUS	#REPOSITORY	SOURCE
POINTER	RERUN	SOURCE-COMPUTER
POSITION	RESERVE	#SOURCES
POSITIVE	RESET	SPACE
#PRESENT	*RESULT	SPACES
PRINT-SWITCH	#RESUME	SPECIAL-NAMES
PRINTING	*RETAINING	STANDARD
*PRIOR	*RETRIEVAL	STANDARD-1
PROCEDURE	#RETRY	STANDARD-2
PROCEED	RETURN	START
PROGRAM	#RETURNING	STATUS
PROGRAM-ID	REVERSED	STOP
#PROGRAM-POINTER	REWIND	*STORE
#PROPERTY	REWRITE	STRING
*PROTECTED	RF	*SUB-SCHEMA
#PROTOTYPE	RH	SUBTRACT
PURGE	RIGHT	SUM
	ROLLBACK	#SUPER
QUOTE	ROUNDED	SUPPRESS
QUOTES	RUN	*SUPPRESSING
		SYMBOLIC
#RAISE	SAME	SYNC
#RAISING	#SCREEN	SYNCHRONIZED
RANDOM	SD	*SYSTEM
RD	SEARCH	#SYSTEM-DEFAULT
READ	SECTION	
*READY	SECURITY	TABLE
*REALM	SEGMENT-LIMIT	TALLY
*REALM-NAME	SELECT	TALLYNG
RECEIVE	*SELECTIVE	TAPE
RECORD	#SELF	TAPES
RECORDING	SEND	*TENANT
RECORDS	SENTENCE	TERMINAL
REDEFINES	SEPARATE	TERMINATE
REEL	SEQUENCE	TEST
REFERENCE	SEQUENTIAL	THAN
RELATIVE	SET	THEN
RELEASE	*SET-SELECTION	THROUGH
REMAINDER	*SETS	THRU
REMOVAL	#SHARING	TIME
RENAMES	SIGN	TIMES
REPEATED	SIZE	TO
REPLACE	SORT	TOP
REPLACING	SORT-MERGE	TRAILING
REPORT	SORT-TAPE	TRUE
REPORTING	SORT-TAPES	TRY

Reserved words for compiler directives

```
ALL
AND
AS

B-AND
B-NOT
B-OR
B-XOR
BYTE-LENGTH

CALL-CONVENTION
CHECKING
COBOL

DE-EDITING
DEFINE
DEFINED
DIVIDE

ELSE
END-IF
END-EVALUATE
EQUAL
EVALUATE

FIXED
FLAG-85
FLAG-NATIVE-ARITHMETIC

FORMAT
FREE
FUNCTION-ARGUMENT

GREATER

IF
IMP
IS

LEAP-SECOND

LESS
LISTING
LOCATION

MOVE

NOT
NUMVAL

OFF
ON
OR
OTHER
OVERRIDE

PAGE
PARAMETER
PROPAGATE

SET
SIZE
SOURCE

THAN
THROUGH
THRU
TO
TRUE
TURN

WHEN

ZERO-LENGTH
```

In addition, all the exception condition names from table 45 in section "Exception conditions and exception statuses" are reserved words for compiler directives.

Context-sensitive words

ALIGNED
ARITHMETIC
ATTRIBUTE
AUTO
AUTOMATIC

BACKGROUND-COLOR

BELL
BLINK

CENTER
CHECK
CLASSIFICATION
CYCLE

DISCARD
DTD

ELEMENT
ENTRY-CONVENTION

EOL
EOS
EXPANDS

FOREGROUND-COLOR

FOREVER
FULL

HIGHLIGHT

IMPLEMENTS
INITIALIZED
INTRINSIC

LINAGE-COUNTER
LOCALIZE
LOWLIGHT

MANUAL

NAMESPACE
NONE
NORMAL
NUMBERS

ONLY

PARAGRAPH
PARSE
PREVIOUS
PROCESSING

RAW
RECURSIVE
RELATION
REQUIRED
RETURN-CODE
REVERSE-VIDEO

SCHEMA
SECONDS
SECURE
SIGNED
SORT-CCSN
SORT-CORE-SIZE

SORT-EOW
SORT-FILE-SIZE

SORT-MODE-SIZE

SORT-RETURN
STACK
STATEMENT
STEP
STRONG
SYMBOL

UCS-2
UCS-4
UNDERLINE
UNSIGNED
UTF-16
UTF-8

VALIDITY

YYYYDDD
YYYYMMDD

4. Function-names

A function-name is a word that is one of a specified list of words which may be used in a COBOL program. The same word, in a different context, may appear in a program as a user-defined word (see section "General").

5. Exception-situation-names

An exception-situation-name is a COBOL word which identifies an exception condition (see table 45 in section "Exception conditions and exception statuses").

In other contexts these words can be used as user-defined names.

2.4.4 Literals

A literal is a character-string whose value is determined by the characters of which it is composed; alternatively, the string may represent a reserved word which corresponds to a figurative constant. Each literal belongs to one of the following classes and categories: alphanumeric, national or numeric.

Alphanumeric and national literals consist of a character string which is enclosed within literal delimiters. Valid quotation symbols in literal delimiters are quotation marks ("...") or apostrophes ('...'). You should note, however, that a literal should close with the same quotation symbol that was used to open it. Both types of quotation marks may be used in a source unit to form literals.

A hexadecimal literal consists of hexadecimal digits. These are the digits 0 .. 9 together with the letters A .. F and a .. f. The members of pairs of uppercase/lowercase letters (e.g. a, A) are considered to be equivalent.

1. Non-numeric literals

Alphanumeric literals are of the class and category alphanumeric.

Format 1 (alphanumeric)

```
{"character-1..." | 'character-1...'}
```

Syntax rules

1. An alphanumeric literal must contain at least one and at most 180 characters (excluding the literal delimiters). When an alphanumeric literal is used in conjunction with national data items, it may contain at most 90 characters.
2. If character-1 is the same as the quotation symbol of the literal delimiter then character1 must be specified twice if it is to be represented as a (single) character in the literal. The duplicated quotation symbol counts as one character for the purposes of the length of the literal.

General rule

1. An alphanumeric literal can contain all characters from the EBCDIC character set. The value of the alphanumeric literal is the string of the individual characters itself without literal delimiters.

Example 2-2

```
'CHARACTER'
"153.78"
"ADAM ""BDAM"" CDAM"
```

Format 2 (hexadecimal-alphanumeric)

```
{X | x} {"hexadecimal digit..." | 'hexadecimal digit...'}
```

Syntax rules

1. A hexadecimal-alphanumeric literal must contain at least 2 hexadecimal digits and at most 360 hexadecimal digits (excluding the literal delimiters).
   When a hexadecimal alphanumeric literal is used in conjunction with national data items, it may contain at most 180 hexadecimal digits.
2. There must be an even number of hexadecimal digits.

General rule

1. The value of the hexadecimal literal is the bit pattern that corresponds to the string of hexadecimal digits.

Example 2-3

```
X"12ab"
```

2. Numeric literals

Numeric literals are of the class and category numeric. There are two types of numeric literals: fixed-point literals and floating-point literals.

- **Numeric fixed-point literals**

  A fixed-point numeric literal is a string of characters chosen from the following set: the digits 0-9, the plus sign, the minus sign, and the decimal point.

  Fixed-point numeric literals must be formed according to the following rules:

  1. The literal may contain 1 to 31 digits.
  2. The literal may contain only one sign character. If a sign is used, it must be the leftmost character of the literal. An unsigned literal is assumed to be positive.
  3. The literal may contain only one decimal point. The decimal point may appear anywhere in the literal, except as the rightmost character. A decimal point designates an assumed decimal point location. (The assumed decimal point in any numeric literal or data item is the position where the compiler and the generated program assume the decimal point to be, though no internal memory position is reserved for a separate decimal point character.) A literal with no decimal point is an integer.

  The term **integer** is used to describe a numeric literal which is unsigned and greater than zero and which has no character positions to the right of the assumed decimal point.

Example 2-4

(Here, the assumed decimal point is represented by the character V.)

Literal	Location of assumed point	Internal sign	No. of digit positions assigned
+123	123V	+	3
3.765	3V765	+	4
-45.7	45V7	-	3

- **Numeric floating-point literals**

  A numeric floating-point literal must have the following format:

  mantissa exponent

  The mantissa consists of an optional sign followed by 1 to 16 digits with a decimal point. The decimal point may be specified anywhere in the mantissa.

  The exponent consists of the symbol E, followed by an optional sign and then by one or more digits (the exponent 0 can be written as 0 or 00).

  The literal must not contain blanks. The exponent must be specified immediately to the right of the mantissa.

  The sign is the only optional character in the format. An unsigned mantissa or an unsigned exponent is interpreted as positive.

  The value of the literal is the product of the mantissa and the power of 10 given by the exponent.

  The absolute value of a number represented by a floating-point literal must not exceed 7.2*10 .

Example 2-5

```
+1.5E-2=1.5*10-2
```

3. National Literals

National literals are of the class and category national.

Format 1 (national)

```
{N | n} {"character-1..." | 'character-1...'}
```

Syntax rules

1. A national literal must contain at least one and at most 90 characters (excluding the literal delimiters).
2. If character-1 is the same as the quotation symbol of the literal delimiter then character-1 must be specified twice if it is to be represented as a (single) character in the literal. The duplicated quotation symbol counts as one character for the purposes of the length of the literal.
3. character-1 must be an EBCDIC character which has a UTF-16 equivalent.

General rule

1. The value of the national literal is the string of national characters (without literal delimiters) which results from the conversion of the individual characters to UTF-16 representation.

Example 2-6

```
N’TSV’
n"1860"
```

Format 2 (hexadecimal-national)

```
{N | n} {X | x} {"hexadecimal digit..." | 'hexadecimal digit...'}
```

Syntax rules

1. A hexadecimal-national literal must contain at least 4 hexadecimal digits and at most 360 hexadecimal digits (excluding the literal delimiters).
2. The count of all hexadecimal digits must be a multiple of 4.

General rule

1. The value of the hexadecimal literal is the bit pattern that corresponds to the string of hexadecimal digits.

Example 2-7

```
NX"005400530056"
nx'0031003800360030'
```

The same literals are consequently represented in "Example 2-6" above.

4. Figurative constants

The values of figurative constants are produced by the compiler and are indicated by the reserved words listed in table 6.

Syntax rules:

1. The singular and plural forms of a figurative constant are equivalent and may be used optionally.
2. Except in the figurative constant ALL, literal, the word ALL has no function; it is used only to improve readability.
3. A figurative constant can be used wherever literal occurs in a format, with the following restrictions:
   - When literal is restricted to numeric literals, the only legal figurative constant is ZERO (ZEROS, ZEROES) without ALL specified.
   - When a syntax rule forbids the specification of figurative constants.
4. Literal must be an alphanumeric or national literal, but not a figurative constant.
5. Symbolic-character must be specified in the SYMBOLIC-CHARACTERS clause of the SPECIAL-NAMES paragraph.

General rules:

1. If one of the figurative constants ZERO, SPACE, HIGH-VALUE, LOW-VALUE or QUOTE is used in a context which requires national characters, the figurative constant represents a national value. In all other cases where the figurative constant represents a character value, the figurative constant stands for an alphanumeric value.
2. If a figurative constant represents a string of one or more characters, the compiler determines the length of the string according to the following rules (in this order):
   - If a figurative constant is specified in a VALUE clause or associated with another data item (e.g. moved to or compared with another data item), it is first duplicated to the right until the resultant string has at least as many character positions as the other data item.
     If this character-string has more character positions than the other data item following the duplication operation, the extra positions will be truncated from the right.
     If the data item has a length of zero, it is truncated to 1 character. Extension or truncation of the character-string of figurative constants takes place prior to and independently of any application of the JUSTIFIED clause to the other data item.

- The character-string always has a length of 1 whenever the figurative constants is not ALL literal, particularly whenever the figurative constants occur in a DISPLAY, STOP, STRING or UNSTRING statement.
- The length of the character-string is equal to the length of literal.

3. If the figurative constants HIGH-VALUE[S] or LOW-VALUE[S] are used in a compilation unit (except for the ALPHABET clause), the character currently associated with this constant is dependent on the collating sequence defined for the program and belonging to the character set (see "OBJECT-COMPUTER paragraph", and "SPECIAL-NAMES paragraph").
4. The figurative constant [ALL] symbolic-character stands for one or more of the characters specified as the value of symbolic-character in the SYMBOLIC-CHARACTERS clause of the SPECIAL-NAMES paragraph.
5. The figurative constant ZERO represents the numeric value 0 or one or more "0" characters depending on the context in which it is used.

Table 6 lists the figurative constants and indicates the values they represent.

Figurative constant	**Corresponding value**	**Example** [1]
[ALL] ZERO or [ALL] ZEROS or [ALL] ZEROES	One or more occurrences of the character 0 (X' F0' or X‘0030‘) or binary zero (X' 00'), depending on the description of the data item.	Statement: MOVE ZEROS TO FIELD. Contents of FIELD: • If FIELD is a binary item: X' 00000000' • If FIELD is an external decimal item: X' F0F0F0F0' (= C' 0000') • If FIELD is an internal decimal item: X' 0000000F' .
[ALL] SPACE or [ALL] SPACES	One or more occurrences of the character space (X' 40' or X‘0020‘).	Statement: MOVE SPACE TO FIELD. Contents of FIELD: X‘40404040‘(= C‘'BLANK''BLANK' 'BLANK' 'BLANK' 'BLANK' ‘)
[ALL] HIGH-VALUE or [ALL] HIGH-VALUES	With COLLATING SEQUENCE unspecified: One or more occurrences of the character that has the highest value in the EBCDIC or UTF-16 collating sequence (X'FF' or X' FFFF').	Statement: MOVE HIGH-VALUE TO FIELD. Contents of FIELD: X‘FFFFFFFF‘(=C‘~~~~‘)

	With COLLATING SEQUENCE specified: The character with the highest position in the program collating sequence.	Entry in SPECIAL-NAMES paragraph: ALPHABET ALPHATAB IS 193 THRU 1, 255 THRU 194. The highest position belongs to the character at the 194th position of the EBCDIC character set, i.e. the character A. A is assigned to HIGH-VALUE.
[ALL] LOW-VALUE or	With COLLATING SEQUENCE unspecified: One or more occurrences of the character that has the lowest value in the EBCDIC or UTF-16 collating sequence (X'00' or X' 0000').	Statement: MOVE LOW-VALUE TO FIELD. Contents of FIELD: X' 00000000'
[ALL] LOW-VALUES	With COLLATING SEQUENCE specified: The character with the lowest position in the program collating sequence.	Entry in SPECIAL-NAMES paragraph: ALPHABET ALPHATAB IS "0" "1" "2". The lowest position belongs to the character 0. 0 is assigned to LOW-VALUE.
[ALL] QUOTE or [ALL] QUOTES	One or more occurrences of the quotation mark. Note: The word QUOTE (or QUOTES) cannot be used in place of a quotation mark to enclose a non-numeric literal.	Statement: MOVE QUOTE TO FIELD. Contents of FIELD: X’7F7F7F7F’ [2]
ALL literal	One or more occurrences of the string of characters composing the literal. The literal must be non-numeric.	Statement: MOVE ALL "A" TO ALPHA. Contents of ALPHA: C' AAAA' Statement: MOVE ALL "12" TO ALPHA. Contents of ALPHA: C' 1212' Statement: MOVE ALL "ABC" TO ALPHA. Contents of ALPHA: C' ABCA'

[ALL] symbolic-character	One or more repetitions of the character specified as the value of symbolic-character in the SYMBOLIC-CHARACTER clause of the SPECIAL-NAMES paragraph.	Description: SYMBOLIC C0 IS 193 Statement: MOVE ALL C0 TO ALPHA. Contents of ALPHA: X‘C0C0C0C0‘

Table 6: COBOL figurative constants and values

[1] In these examples it is assumed that, unless otherwise specified, ALPHA and FIELD are defined as PIC X(4).

[2] Can be changed by option to X’7D’ (see "COBOL2000 User Guide" [1]).

2.4.5 PICTURE character-string

A PICTURE character-string consists of certain combinations of characters from the COBOL character set, which are used as symbols (see the section "PICTURE clause").

Any punctuation character within a PICTURE character-string is not interpreted as a punctuation character but rather as a symbol used in that PICTURE character-string.

2.4.6 Types

A type is a pattern containing all the characteristics of the data description and its subordinate data items. The principal characteristics of a type, which is identified by its type name, are the relative positions and lengths of its elementary items, which are defined in the type declaration, plus the BLANK WHEN ZERO clause, JUSTIFIED clause, PICTURE clause, SIGN clause, SYNCHRONIZED clause and USAGE clause which are specified for these elementary data items (for details on the clauses see the section "Data description entry").

A type is defined by specifying the TYPEDEF clause.

A type is referenced by a TYPE clause being specified in a data description. The typed data item thus defined has all the characteristics of the referenced type.

Group items can be weakly or strongly typed. A group item is strongly typed if:

- either its data description contains a TYPE clause which refers to a type definition with the specification STRONG, or
- it is subordinate to a group item containing a TYPE clause which refers to a type definition with the specification STRONG.

Two typed group items are of the **same type** if

- either the data descriptions contain TYPE clauses which reference equivalent type description entries, or
- data descriptions are subordinate group items in equivalent type description entries which start at the same position within the type description entries and are of the same length.

Two type description entries are regarded as equivalent if they have the same type name and for each elementary data item in one type description entry a corresponding elementary data item starting at the same position and of the same length exists in the other type description entry. Each pair of corresponding elementary data items must have the same BLANK WHEN ZERO clauses, JUSTIFIED clauses, PICTURE clauses, SIGN clauses, SYNCHRONIZED clauses and USAGE clauses.

The following exception applies here:

The *point* character in the associated PICTURE string is equivalent if the specification DECIMAL-POINT IS COMMA applies either for both or neither of the type description entries. The same applies for the *comma* character in the associated PICTURE string.

Weakly typed data descriptions

Weakly typed data descriptions take over the characteristics of their associated type description entries. These characteristics cannot be modified by the definitions of higher-ranking group items.

Weakly typed data descriptions can be used exactly like data descriptions without an associated type description.

Strongly typed data descriptions

Strongly typed data descriptions also take over the characteristics of their associated type descriptions. In addition, there are restrictions regarding their use in order to protect the integrity of the data.

Only group items, not data items, can be strongly typed.

2.4.7 Zero-length items

A zero-length item is a data item whose minimum permissible length is 0 and whose length at runtime is 0. A zero-length item is one of the following:

- A group item containing only data items which are described with OCCURS DEPENDING ON, and the number of repetitions is 0.
- A group item which is subordinate only to zero-length items.
- A data item which is defined with ANY LENGTH and corresponds to an argument or return value which is a zero-length item.
- A record of a file with variable record format in which the number of character positions is 0.
- An internal standard function which returns a zero-length result.
- Special registers XML-TEXT, XML-NAMESPACE, XML-NAMESPACE-PREFIX or XML-NTEXT, XML-NNAMESPACE, XML-NNAMESPACE-PREFIX if they are not assigned a value (see section "Special registers for the XML PARSE statement").

2.4.8 Concept of computer-independent data description

To make data as computer-independent as possible, the characteristics and properties of the data are described in a format which is largely independent of the concrete representation in the computer or on the external medium.

The following applies for literals:

- The content of alphanumeric non-hexadecimal literals is taken over unchanged for the runtime.
- The content of national non-hexadecimal literals is always converted to UTF-16 representation for the runtime.
- Hexadecimal literals always specify the final bit pattern that is to be used at runtime.

1. Concept of logical record and file

The logical characteristics of a record or a file differ from the way in which the data is physically stored in the computer.

- **Physical aspects of a file**

  The physical aspects of a file are determined by the way in which the data is stored on the input or output medium. They include such features as:

  - the grouping of logical records, taking into account the physical limitations of the storage medium.
  - the manner in which a file may be identified.

- **Conceptual characteristics of a file**

  The conceptual characteristics of a file are determined by the structures which the user specifies by data definitions. The input-output statements in a COBOL program refer to logical records.

  It is extremely important to distinguish between a physical record and a logical record.

  - A **physical record** (or **block**) is a unit of information whose size and recording mode provide for optimum data storage on an input or output medium for a particular computer installation. The size of a physical record is machine-dependent and bears no direct relationship to the size of the logical file information.
  - A **logical record** (or simply **record**) is a group of related data which can be uniquely identified and treated as a unit, and can be read from or written to a file. A block may contain several records.

  The term "record" is not restricted to data stored on an external data medium, but can be applied to the definition of working storage for data created internally during program execution.

  In this manual, references to "records" always mean to logical records.

2. Level concept

The level concept permits the structuring of a logical record. Data processed by a COBOL program can be described as elementary items, group items, records and files.

- **Elementary items**

  An elementary item is the smallest unit of data bearing a name, i.e. it is not divisible into further elementary items. An elementary item is mostly described with a PICTURE clause (for exceptions see section "Data description entry formats").

  The length of an elementary item must not exceed 131 071 bytes.

- **Group items**

  Several elementary items combined form a group item. Thus, a number of elementary items may be addressed simultaneously under the name of the group item. Each group consists of one or more elementary items. Groups, in turn, may be contain further more group items.Consequently, an elementary item may belong to more than one group item (see figure 1 and figure 2). The name of a group item must not be described with the PICTURE clause.

- **Records**

  A record is a data item which is not subordinate to another data item. It consists of one group items, or it is itself an elementary item. The description of a record must start in Area A.

- **Level-numbers**

  Data is divided into various levels. These levels are indicated by means of level-numbers. The numbers 01 to 49 are allowed as level-numbers. In addition, there are special level-numbers: 66, 77, and 88. In a compilation unit, every level-number must be given a separate entry.

  Since a record represents the largest organizational unit, level-numbers for records start at 01. Hierarchically subordinate items are assigned numerically higher level numbers (from 02 to 49). The level-number of a subordinate data item must be greater than that of a higher-ranking data item by one or more units. Once an elementary item has been described, only those level-numbers which have already appeared in the record description entry are permitted.

Example 2-8

right:

```
01 DATA RECORD.
   05  GROUP-ITEM-1.
       10  ELEMENTARY-ITEM-11 ...
       10  ELEMENTARY-ITEM-12 ...
       10  ELEMENTARY-ITEM-13 ...
   05  GROUP-ITEM-2.
       10  ELEMENTARY-ITEM-21 ...
       10  ELEMENTARY-ITEM-22 ...
```

wrong:

```
01 DATA RECORD.
   05 GROUP-ITEM-1.
       10  ELEMENTARY-ITEM-11 ...
       10  ELEMENTARY-ITEM-12 ...
       10  ELEMENTARY-ITEM-13 ...
   03  GROUP-ITEM-2.
       10  ELEMENTARY-ITEM-21 ...
       10  ELEMENTARY-ITEM-22 ...
```

There are three types of data for which no level concept exists. These are assigned the level numbers 66, 77, and 88:

- Level number 66 is given to the names of data items described with the RENAMES clause (see the section "RENAMES clause").
- Level number 77 is given to structure-independent data items of the WORKING-STORAGE SECTION or LINKAGE SECTION (see the section "Level number").
- Level number 88 is given to the explanation of condition-names (see the section "VALUE clause").

Further rules are described in the section "Level number".

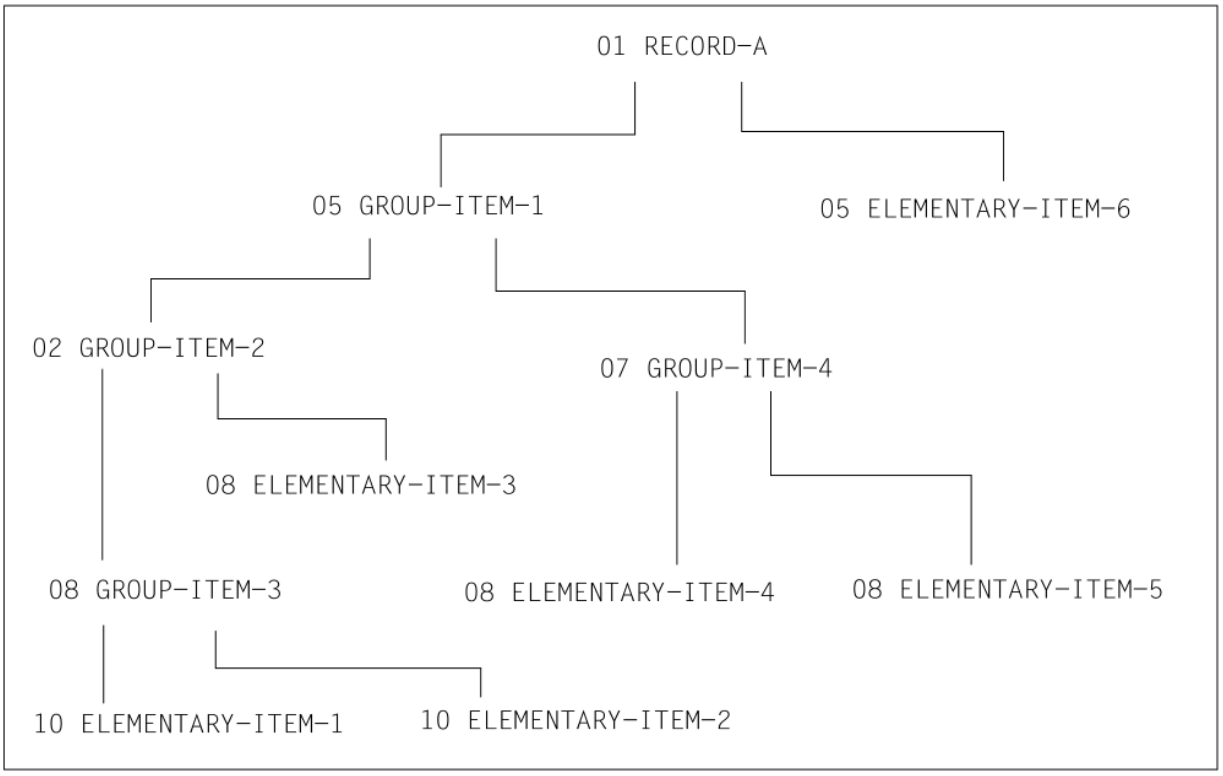


Figure 1: Relationship between group items and elementary items in a record

```
01 RECORD-A.
   05 GROUP-ITEM-1.
      07 GROUP-ITEM-2.
         08 GROUP-ITEM-3.
            10 ELEMENTARY-ITEM-1...
            10 ELEMENTARY-ITEM-2...
         08 ELEMENTARY-ITEM-3...
      07 GROUP-ITEM-4.
         08 ELEMENTARY-ITEM-4...
         08 ELEMENTARY-ITEM-5...
   05 ELEMENTARY-ITEM-6...
```

Figure 2: Group items and elementary items in a record

Figure 1 shows the structure of a sample record; Figure 2 demonstrates how to use level numbers to represent this structure in the record description entry. In this example, GROUP-ITEM-3 and ELEMENTARY-ITEM-3 are a subordinate part of GROUP-ITEM-2; similarly, GROUP-ITEM-2 and GROUP-ITEM-4 are a subordinate part of GROUP-ITEM-1.

3. Classes and categories of data and literals

Each data item is assigned a class and a category.

The class and category of a strongly typed group item are the type name which is specified in the TYPE clause of the group item.

The class and category of a group item which is not strongly typed are

- alphanumeric for an alphanumeric group
- national for a national group

An alphanumeric group is treated as if it has USAGE DISPLAY.

The class and category of a function are defined by the function type (see “Function types” in section "General").
The category of an elementary data item depends on its definition.
The classes and categories of literals are defined in the section "Literals".

Table 7 below illustrates the relationship between the classes and categories of elementary data items.

Class	Category
alphabetic	alphabetic
alphanumeric	numeric edited alphanumeric edited alphanumeric
index	index
national	national
numeric	numeric
object	object reference
pointer	data pointer program pointer

Table 7: Classes and categories of elementary items

The following subsections describe the data items in the various categories.

- **Alphabetic data items**

  An alphabetic data item is a data item whose contents can be any combination of the 52 uppercase and lowercase letters of the alphabet, plus the space character. Each alphabetic character is stored in its own byte in working storage.

  The PICTURE character-string for alphabetic items contains only the symbol A.
  The data format of alphabetic items is always DISPLAY.

  An alphabetic data item may be specified wherever an elementary alphanumeric item is permitted as an operand. Unless otherwise specified, it is treated as if it were an alphanumeric item.

- **Alphanumeric edited items**

  An alphanumeric edited item describes the editing of an alphanumeric value. When an alphanumeric edited item is a receiving item for a MOVE statement, the data being moved into the item is edited according to the PICTURE character-string specified for the item.

  The picture-string of an alphanumeric edited item is restricted to certain combinations of the following characters: A, / (slash), X, 9, 0 (zero), and B (see the section "PICTURE clause").

  The contents of an alphanumeric edited item are allowable characters chosen from the EBCDIC character set.

  The data format of an alphanumeric edited item is always DISPLAY.

- **Alphanumeric items**

  An alphanumeric item is one whose contents are any characters from the EBCDIC set.

  Its picture-string is restricted to combinations of the symbols A, X, and 9. The item is treated as if its picture-string contained all X's.

  A picture-string which contains all A's or all 9's does not define an alphanumeric item. The data format of an alphanumeric item is always DISPLAY.

- **Data pointer**

  A data pointer (also simply referred to as pointer) is an elementary item in which a data address can be stored.

- **Index items**

  Each of the following is an index item:

  - An elementary item which is described explicitly or implicitly with USAGE INDEX
  - An index function

- **National items**

  Each of the following is a national item:

  - An elementary item which is described as national by its PICTURE character-string
  - An elementary item without PICTURE clause which is described with a format 1 VALUE clause and a national literal
  - An elementary item which is defined explicitly or implicitly with USAGE NATIONAL
  - A group item which is described explicitly or implicitly with a GROUP-USAGE NATIONAL clause
  - A national function

- **Numeric edited items**

  A numeric edited item describes the editing of a numeric value. When a numeric edited item is a receiving item for a MOVE, the data coming into the item is edited according to the picture-string specified.

  The picture-string of a numeric edited item is restricted to certain combinations of the symbols B, / (slash), P, V, Z, 0 (zero), 9, , (comma), . (decimal point), *, +, -, CR, DB and $ (currency sign). The allowable combinations are determined from editing rules and the order or precedence of symbols (see the section "PICTURE clause"). The maximum number of digits that may be represented in a picture-string for a numeric edited item is 31.

  Data is stored one character per byte. The contents of a character position that represents a digit must be one of the numerals 0 through 9.

  The data format of a numeric edited data item is always DISPLAY.

- **Numeric items**

There are two types of numeric data items, fixed-point items and floating-point items. For the internal representation of numeric items, see table 17 in section "COMPUTATIONAL-3 phrase or PACKED-DECIMAL phrase".

Fixed-point data items

A fixed-point data item is a numeric data item in which the operational decimal point is assumed to be present in every value or to be maintained at a fixed position relative to the beginning or end of the storage area reserved for the data item. The contents of a fixed-point data item must be comprised of the digits 0 through 9, provided the SIGN clause is not specified. If the SIGN clause is specified, the contents may contain a +, - or other representations of the sign in addition to the above-mentioned digits. If the picture-string contains an S for a fixed-point data item, the contents of the data item are treated as positive or negative, depending on the operational sign. If the picture-string does not contain an S, the contents of the data item are treated as an absolute value. treated as an absolute value.

Picture-strings for fixed-point items may contain the symbolic characters 9, P, S and V only.

COBOL recognizes three types of fixed-point numbers:

external decimal	(USAGE IS DISPLAY)
binary	(USAGE IS COMPUTATIONAL or COMPUTATIONAL-5 or USAGE IS BINARY)
internal decimal	(USAGE IS COMPUTATIONAL-3 or USAGE IS PACKED-DECIMAL)

The differences between these three types are described under the section "USAGE clause".

Floating-point data items

A floating-point data item is a numeric data item whose decimal point is movable, i.e. it can be located at various positions and is defined by specifying a base-10 exponent. The base thus raised to a particular power serves as the coefficient of a fixed-point number (mantissa) thereby representing the floating-point number.

Floating-point data items are only used for data whose potential value range is too large for fixed-point representation.

There are two kinds of floating-point data items: external floating-point data items and internal floating-point data items.

External floating-point data items

The picture-string for an external floating-point data item can contain the characters 9, . (decimal point), V, E, + and -. Except for V, each character occupies one byte of memory, and is contained in each printout (see "PICTURE clause" and "USAGE clause" for further details).

The data format of an external floating-point data item is always DISPLAY.

Internal floating-point data items

There are two kinds of internal floating-point data items:

- single-precision (USAGE IS COMPUTATIONAL-1), with a length of 4 bytes and
- double-precision (USAGE IS COMPUTATIONAL-2), with a length of 8 bytes.

Both encompass the same value range. A single-precision data item permits precision to 7 decimal digits. A double-precision data item allows precision to 16 decimal digits.

A PICTURE clause is prohibited for internal floating-point data items; the length of this kind of data item is determined by its USAGE clause.

- **Object and object reference**

  An object reference is an implicitly or explicitly defined data item. The content of the object reference uniquely references an object and its associated information.

- **Program pointer**

  A program pointer is an elementary item in which the address of a program can be stored.

4. Algebraic signs

There are two categories of algebraic signs:

- operational signs, which are associated with signed numeric items and signed numeric literals to specify their algebraic properties
- editing signs, which occur e.g. in edited reports in order to indicate the sign of a data item.
  Editing signs are inserted in a data item by means of the sign control character of the relevant picture-string (see "PICTURE clause").

5. Alignment of data

The alignment of data within elementary data items depends on the category of the receiving item.

- **Numeric data items**

  If the receiving item is described as a numeric item, the data being sent is aligned on the decimal point and is moved to the digit positions of the receiving item. If the data being sent is shorter than the receiving item, the unused digit positions are filled with zeros. If the data being sent is longer than the receiving item, it is truncated from the left or right as appropriate.
  If an assumed decimal point is not supplied explicitly, the receiving item is treated as if it had an assumed decimal point immediately following its rightmost digit; alignment and moving are as described above.

- **Numeric edited data items**

  If the receiving item is a numeric edited item, alignment and moving of the data being sent take place as in the case of numeric receiving items; leading zeros can be replaced by other characters through special editing specifications.

- **Alphanumeric, alphanumeric edited, alphabetic and national data items**

  If the receiving item is alphanumeric (other than numeric edited), alphanumeric edited, alphabetic or national, then the data being sent is moved from left to right into the character positions of the receiving item. If the data being sent is shorter than the receiving item, the unused character positions are filled with spaces according the receiving item class. If the data being sent is longer than the receiving item, the excess characters of the data being sent are truncated.
  If the JUSTIFIED clause is specified for the receiving item, refer to the section "JUSTIFIED clause".

- **Data item alignment for accelerated program execution**

  Particular data (in arithmetic or subscripting operations) can be processed more rapidly if the data is aligned on natural boundaries (halfword, word, doubleword).
  The object program requires additional machine instructions for accessing and storing data if parts of two or more data items occur between two adjacent natural boundaries or if certain natural boundaries divide a single item.
  Data items whose alignment on these natural boundaries is such that they do not require additional machine instructions, are defined as "synchronized". Data items of the classes index, object and pointer are always aligned (see table 16 “Data item alignment” in section "SYNCHRONIZED clause").

  The user has two means of achieving this form of alignment:

  - using the SYNCHRONIZED clause (see the section "SYNCHRONIZED clause"),
  - suitably organizing the data, allowing for the natural boundaries.
    See the next section for details.

2.4.9 Implementor-dependent representation and alignment of data

Alignment by insertion of slack bytes

There are two types of slack bytes:

- Intra-record slack bytes (slack bytes within records) are unused character positions which precede every aligned data item in the record.
- Inter-record slack bytes (slack bytes between records) are unused character positions which are inserted between blocked logical records.

- **Intra-record slack bytes**

  For an output file or in the WORKING-STORAGE and LOCAL-STORAGE SECTION, the compiler inserts slack bytes within records to ensure that all aligned data items are justified on the appropriate boundaries. For an input file or in the LINKAGE section, the compiler expects any required slack bytes to be present in order to ensure proper alignment of a data item declared as SYNCHRONIZED.

  Since it is very important for the user to know the length of a record in a file, the algorithm that the compiler uses to determine whether slack bytes are required and, if they are required, how many slack bytes are to be added, is described as follows:

  The number of occupied bytes in all elementary data items which precede a data item in a record is computed, including any slack bytes previously added.

  This sum is to be divided by m, where:

  m = 2 for COMPUTATIONAL- or COMPUTATIONAL-5 or BINARY data items with a length of 4 digits or less;

  m = 4 for COMPUTATIONAL or COMPUTATIONAL-5 or BINARY data items with a length of 5 digits or more;

  m = 4 for COMPUTATIONAL-1 data items;

  m = 8 for COMPUTATIONAL-2 data items with a length of 18 digits or less;

  m = 4 for index data items;

  m = 8 for object references;

  m = 4 for pointers;

  m = 8 for typed group items.

  If the remainder r of this division is equal to zero, no slack bytes are required. If the remainder is unequal to zero, the number of slack bytes to be added is equal to m `-r`.

  These slack bytes are added to each record immediately following the elementary item that precedes the BINARY, COMPUTATIONAL, COMPUTATIONAL-1, COMPUTATIONAL-2, COMPUTATIONAL-5 or INDEX data item, the object reference, the pointer or the typed group item. They are declared as though they were a data item with a level number equal to that of the data item immediately preceding the aligned data item, and must be included in the size of the group where they are contained.

> Create items which require alignment at the start of substructures. This ensures that they always contain the same number of slack bytes.

Example 2-9

Slack bytes within records

```
01 A.
   02 B PICTURE X(5).
   02 C.
      03 D PICTURE XX.
     [03 slack byte PICTURE X. Inserted by the compiler.]
      03 E PICTURE S9(6) COMP SYNCHRONIZED.
```

Slack bytes are also added by the compiler when a group item is described with an OCCURS clause and contains an aligned data item defined with USAGE as BINARY, COMPUTATIONAL, COMPUTATIONAL-1, COMPUTATIONAL-2, COMPUTATIONAL-5 or INDEX. To decide whether to add slack bytes, the following steps are performed:

- The compiler calculates the size of the group including all intra-record slack bytes required.
- This sum is divided by the largest m required by any elementary item within the group.
- If the remainder r of this division is equal to zero, no slack bytes will be needed. If r is unequal to zero, m-r slack bytes must be added.

Insertion of slack bytes takes place at the end of each occurrence of the group item which contains the OCCURS clause, in order to ensure that all occurrences of table items begin at the same kind of boundary. In example 2-8 in section "Concept of computer-independent data description", all occurrences of D begin one byte beyond a double-word boundary.

Example 2-10

Occurrences of slack bytes in tables

```
01 A.
   02 B  PICTURE X.
   02 C  OCCURS 10 TIMES.
      03 D PICTURE X.
     [03 slack bytes PICTURE XX. Inserted by the compiler.]
      03 E PICTURE S9(4)V99 COMP SYNC.
      03 F PICTURE S9(4) COMP SYNC.
      03 G PICTURE X(5).
     [03 slack bytes PICTURE XX. Inserted by the compiler.]
```

If aligned data items defined as BINARY, COMPUTATIONAL, COMPUTATIONAL-1, COMPUTATIONAL-2, COMPUTATIONAL-5 or INDEX follow an entry with an OCCURS DEPENDING clause, then slack bytes are added on the basis of the item which is repeated with the maximum number. If the length of this item is not divisible by the m required by the data, then only certain values of the data-name used in the DEPENDING phrase produce a correct alignment of the items. The programmer should be aware of this situation and try to avoid it. These values are ones in which the length of the data item, multiplied by the number of occurrences plus the number of slack bytes calculated on the basis of the maximum number of occurrences, is divisible by m with no remainder.

Example 2-11

Occurrences of slack bytes in tables with the DEPENDING phrase

```
01 A.
   02 B PICTURE 99.
   02 C PICTURE X OCCURS 50 TO 99 TIMES
        DEPENDING ON B.
  [02 slack bytes PICTURE X. Inserted by the compiler.]
   02 D PICTURE S99 COMP SYNC.
```

In this example, when references to D are required, B is restricted to odd values.

```
01 A.
   02 B PICTURE 999.
   02 C PICTURE XX OCCURS 20 TO 99 TIMES
        DEPENDING ON B.
  [02 slack bytes PICTURE X. Inserted by the compiler.]
   02 D PICTURE S99 COMP SYNC.
```

In this example, all values of B provide correct references to D.

- **Alignment of records**
  All record descriptions (01-level entries) in all sections of the Data Division begin at doubleword boundaries.
- **Inter-record slack bytes**
  When records that contain aligned data items are to be blocked, the programmer must ensure that all records following the first record in the input-output storage area are properly boundary-aligned. This is only necessary, however, in cases where data is to be processed blockwise (locate mode). COBOL2000 does not use this mode.

2.5 Uniqueness of references

2.5.1 Qualification

Function

Every user-defined name explicitly referenced in a COBOL compilation unit must be unique. A name is unique when there is no other name consisting of the same sequence of characters and hyphens, or the name occurs in a hierarchy of names, so that it can be referenced unambiguously. This occurs by specifying one or more names on a higher level of the hierarchy. The higher levels are called qualifiers, and the process that causes the name to be unique is called qualification. A name must be qualified sufficiently to be unique; however, it is not absolutely necessary to specify all levels of the hierarchy. Within the DATA DIVISION, all data names used for qualification purposes must be given a level number or a level identifier. Thus, two identical data names cannot be subordinate elements of a single group item, unless they can be uniquely qualified. In the PROCEDURE DIVISION, two identical paragraph names are only allowed to occur in the same section if they are not referenced. If a paragraph name is referenced, it must be unique, i.e. it must be qualified when it occurs in more than one section.

In conjunction with types, the uniqueness of names is guaranteed if a name occurs only within a type definition and this type definition is not used more than once for defining data. In all other cases qualification is required.

In the qualification hierarchy, the names belonging to a level identifier are the most important, followed by the names belonging to level 01, then those belonging to level 02 to 49. A section name is the only qualifier available for paragraph names. The uppermost name in the hierarchy must be unique, and cannot be qualified. Subscripted or indexed data names and conditional variables, as well as procedure names and data names, can be made unique by means of qualification. The name of a conditional variable can be used as a qualifier for each of its condition names.

Format 1

```
{data-name-1 | condition-name} {   {{IN | OF} data-name-2}... [{IN | OF} filename]
                                 | {IN | OF} file-name
                               }
```

Format 2

```
paragraph-name {IN | OF} section-name
```

Format 3

```
text-name {IN | OF} library-name
```

Format 4

```
LINAGE-COUNTER {IN | OF} file-name
```

Format 5

```
{PAGE-COUNTER | LINE-COUNTER} {IN | OF} report-name
```

Format 6

```
data-name-1  {   {IN | OF} data-name-2 [{IN | OF} report-name]
              | {IN | OF} report-name
             }
```

Syntax rules

1. Each qualifier must be of a successively higher level and within the same hierarchy as the name it qualifies.
2. The same name must not appear on more than one level of the hierarchy.
3. A data name must not be subscripted or indexed when used as a qualifier.

General rules

1. A data-name or a condition-name, if assigned to more than one data item within the compilation unit, must be qualified whenever it is referenced in the Procedure, Environment or Data Division (except in the REDEFINES clause, where qualification is not needed and may be used).
2. A paragraph-name is only allowed to occur more than once within a section if it is not referenced. If it is referenced, it is only allowed to occur once within a section, or must be qualified when it occurs in more than one section. When a paragraph-name is qualified by a section-name, the word SECTION must not be used. A paragraph-name, when referenced from within the same section, need not be qualified.
3. A name may be qualified even when qualification is not required; if uniqueness may be ensured by more than one combination of qualifiers, then each such combination is permitted. The total set of the qualifiers for a given data-name must not be identical to a subset of qualifiers for another data-name.
4. If more than one COBOL library is available to the compiler at compile time, then every time text-name is referenced it must be qualified by library-name.
5. If data-name is qualified in a contained or containing program of a nested program, the same data-name must not be used for a unit of data (record or data item) that is declared as external or global in one of the group of nested programs.

2.5.2 Subscripting

Function

Subscripts are used when an individual element is to be accessed within a table (see the section "OCCURS clause").

Format 1

Describes subscripting without qualification.

```
{data-name-1 | condition-name} ({subscript-1}...)
```

Format 2

Describes subscripting with qualification.

```
{data-name-1 | condition-name} {IN | OF} data-name-2 [{IN | OF} data-name-3 ] ...
({subscript-1}...)
```

For a description of qualification with associated rules see the section "Qualification".

Syntax rules for both formats

1. data-name-1 is the name of the table element. Its data description entry must either contain an OCCURS clause, or it must be subordinate to a data item which contains an OCCURS clause.
2. subscript-1... may be represented by
   - an integer literal
   - a data-name with a positive integer as its value
   - relative subscripting
   - an arithmetic expression with a positive integer as its value
   - the word ALL.

   The data-name itself may be qualified but not indexed. ALL may be specified only if the subscripted identifier is specified as a function argument.
3. One subscript must be specified for each OCCURS clause which is subordinate to data-name. Since a table may have up to seven dimensions, references to an element in a table may require up to seven subscripts.
4. The subscript is enclosed in parentheses. The left parenthesis immediately follows the spaces after the name of the table element (data-name). When more than one subscript appears within a set of parentheses, these subscripts may be separated either by commas followed by at least one space, or by spaces only.

   In the case of relative subscripting, the operational signs between data-name and integer must also be delimited by spaces.
5. The subscript, or set of subscripts, identifies the table element which is to be referenced. A data-name to which one or more subscripts have been added is called a subscripted data-name or identifier.
6. When more than one subscript is used, they are entered proceeding from the outermost to the innermost table.

General rule for both formats

The **subscript** may contain a plus sign. The lowest valid subscript is 1. Consequently, neither zero nor negative numbers are permitted for subscripting. The highest allowable subscript value, in any particular case, is the maximum number of occurrences of the item, as specified in the OCCURS clause.

2.5.3 Indexing

Function

Indices are used when an individual element is to be accessed within a table (see the section "OCCURS clause").

Format 1

Describes indexing without qualification:

```
{data-name-1 | condition-name}  ({index-1 [{+ | -} integer ]} ... )
```

Format 2

Describes indexing with qualification:

```
{data-name-1 | condition-name} [{IN | OF} data-name-2] [{IN | OF} file-name]
        ({index-1 [{+ | -} integer]}... )
```

For a description of qualification with associated rules see section "Qualification".

Syntax rules for both formats

1. data-name-1 is the name of a table element.

   If a data-name-1 is used with an index-name, then the data description entry of data-name-1 must either itself contain an OCCURS clause with an INDEXED BY phrase, or data-name-1 must be subordinate to a group item containing an OCCURS clause with an INDEXED BY phrase.

   For example, the reference

   `TOTAL (INDEXA, INDEXB),`

   implies that TOTAL belongs to a structure with two levels of OCCURS clauses, each with an INDEXED BY phrase specified.
2. The index is enclosed in parentheses. The left parenthesis immediately follows the space after the name of the table element (data-name). When more than one index-name appears within a set of parentheses, these index-names may be separated either by commas followed by at least one space, or by spaces only.
3. When the + integer or - integer phrase is used, the + and - characters must be preceded and followed by spaces.
4. Index-names are written proceeding from the outermost to the innermost table.
5. The lowest valid occurrence number for index-name is 1; the highest is, in any particular case, the maximum number of occurrences of the item. This maximum number is defined in the OCCURS clause. This same rule also applies to relative indexing.
6. Referencing a table element, or an item within a table element, does not change the index-name associated with this table.
7. The use of relative indexing will not change the values of indices in the object program.

General rules for both formats

1. The values of indices may be stored without conversion (SET statement) in data items defined with the USAGE IS INDEX clause. These data items are then called index data items (see "USAGE clause" and "SET statement").
2. An index may be modified only by a SET, SEARCH or PERFORM statement (see the descriptions of these statements).

2.5.4 Function-identifier

A function-identifier is a syntactically correct combination of character-strings and separators that reference a function. The data item represented by a function is uniquely identified by a function-name with its arguments, if any. A function-identifier may include a reference-modifier.

Format

```
FUNCTION function-name-1 [({argument-1}...)]
```

Syntax rules

1. argument-1 must be an identifier, a literal, or an arithmetic expression. Specific rules governing the number, class, and category of argument-1 are given in the definition of each function (see chapter "Intrinsic functions").
2. A function-identifier which references an alphanumeric or national function may be specified anywhere in the general formats that an identifier is permitted and where the rules associated with the general formats do not specifically prohibit reference to functions, except as follows:
   a. as a receiving operand of any statement,
   b. where the rules associated with the general formats require the data item being referenced to have particular characteristics (such as class and category, usage, size, sign, and permissible values) and the evaluation of the function according to its definition and the particular arguments specified would not have these characteristics.
3. A function-identifier which references an integer or numeric function may be used only in an arithmetic expression.

General rules

1. The class and other characteristics of the function being referenced are determined by the function definition.
2. At the time reference is made to a function, its arguments are evaluated individually in the order specified in the list of arguments, from left to right. An argument being evaluated may itself be a function-identifier or may be an expression containing function-identifiers. There is no restriction preventing the function referenced in evaluating an argument from being the same function as that for which the argument is specified.

2.5.5 Reference modification

Function

Reference modification defines a data item through specification of the position of the leftmost character and the length of the data item.

Format

```
{identifier-1 | FUNCTION function-name-1 [({argument-1}... )] } (leftmost-char-
position: [length])
```

identifier-1 and FUNCTION function-name-1 are not part of the reference-modifier. They are included here for the sake of clarity.

Syntax rules

1. identifier-1 must be one of the following data items:
   - an elementary data item of the category alphanumeric or an alphanumeric group item.
   - an alphabetic data item.
   - a numeric or alphanumeric data item or a numeric data item with the USAGE DISPLAY phrase; in no cases may the data item be subordinate to a strongly typed group item.
   - a group data item which is not strongly typed.
   - a national data item.
   - a numeric edited or alphanumeric edited data item or a numeric data item with the USAGE DISPLAY phrase; in no cases may the data item be subordinate to a strongly typed group item.
2. leftmost-character-position and length must be arithmetic expressions.
3. Unless otherwise specified, reference modification may be used wherever an data item identifier of the class alphanumeric or national is permitted.
4. identifier-1 must not be reference modificated.
5. The function referenced by function-name-1 and its arguments (if any) must be an alphanumeric or national function.
6. identifier-1 may not refer to a group data item that contains data items of the class “object” or “pointer”.

General rules

1. Each character of identifier-1 or function-name-1 is assigned an ordinal number that is incremented stepwise by one from the leftmost position to the rightmost position. The number one is assigned to the leftmost position. If the data description entry for data-name-1 contains a SIGN IS SEPARATE clause, an ordinal number is likewise assigned to the sign position in this data item.
2. If identifier-1 is described as numeric, numeric edited, alphanumeric or alphanumeric edited, it is operated upon for the purposes of reference modification as if it were redefined as an alphanumeric data item of the same size.
3. If identifier-1 is subscripted and ALL is specified for a subscript, the reference-modifier refers to each of the implicitly referenced table elements.
4. Reference modification creates a unique data item that forms a subset of the data item referenced by identifier-1 or function-name-1. This unique data item is defined as follows:

- leftmost-character-position specifies the character position of identifier-1 at which the subfield is to begin. leftmost-character-position must give a positive nonzero integer value that is less than or equal to the number of character positions of identifier-1 or function-name-1 and its arguments (if any).
- length denotes the length of the unique data item. length must give a positive nonzero integer value.
- The sum of leftmost-character-position and length minus 1 must not exceed the number of characters of the data item referenced by identifier-1 or function-name-1. If "length" is not specified, the unique data item extends from the position denoted by leftmost-character-position to the last character (inclusive) of the data item referenced by identifier-1 or function-name-1.

Note:
In the case of national data it must be ensured that surrogate pairs are not separated by reference modification.

5. The unique data item is considered to be an elementary item without a JUSTIFIED clause. The generated subfield has the same class and category as identifier-1 or as function-name-1 except that the categories numeric, numeric-edited and alphanumeric-edited are regarded as the class and category alphanumeric and USAGE DISPLAY is assumed for an alphanumeric group.

Example 2-12

A data item CARREG contains a 10-character car registration, the last 6 characters of which are to be transferred to a subitem SHORTREG:

Program extract:

```
...
01  CARREG     PIC X(10).
01  SHORTREG   PIC X(6).
...
    MOVE CARREG (5:6) TO SHORTREG.
...
```

The "5" within the parentheses specifies that the MOVE operation is to take effect starting at the fifth character; the colon is the required separator; the "6" specifies that six characters are to be transferred to the item SHORTREG.

2.5.6 Identifier

Identifier is a term used to reflect that a data name, if not unique in a program, must be followed by a syntactically correct combination of qualifiers, subscripts or indices necessary to ensure uniqueness.

Format 1

```
FUNCTION function-name-1 [({argument-1}...)] [reference-modifier]
```

Format 2

```
data-name-1 [{IN | OF} data-name-2] ...  [{IN | OF} {file-name-1 | report-name-1}]
        [({subscript} ...)] [reference-modifier]
```

2.5.7 Object view

An object view causes the compiler to treat an object reference (e.g. within the framework of conformance checks) as if it were defined in the specified form. If required, runtime checks are performed to ensure that the current contents of the object reference have the required properties.

Format

```
identifier-1 AS {  [ FACTORY OF] class-name-1 [ONLY]
                 | interface-name-1
                 | UNIVERSAL
               }
```

Syntax rules

1. identifier-1 must be an object reference or a predefined object identifier, which must not be SUPER or NULL or a further object view.
2. identifier-1 may also be a class name.
3. The object view may not be specified as a receiving operand.

General rules

1. An object view returns an object reference that points to the same object as identifier-1 and assumes the same interface for conformance checks as specified in the AS phrase.
2. If class-name-1 is specified without either of the optional phrases, the implicit result is the same as for USAGE IS OBJECT REFERENCE class-name-1. If the object referenced by identifier-1 is not an object of class-name-1 or a subclass of class-name-1, an exception condition EC-OO-CONFORMANCE occurs.
3. If the FACTORY phrase is specified without the ONLY phrase, the implicit result is the same as for USAGE IS OBJECT REFERENCE class-name-1. If the object referenced by identifier-1 is not the factory object of class-name-1 or a subclass of class-name-1, an exception condition EC-OO-CONFORMANCE occurs.
4. If the ONLY phrase is specified and the FACTORY phrase is not specified, the implicit result is the same as for USAGE OBJECT REFERENCE class-name-1 ONLY. If the object referenced by identifier-1 is not an object of class-name-1, an exception condition EC-OO-CONFORMANCE occurs.
5. If both the FACTORY phrase and the ONLY phrase are specified, the implicit result is the same as for USAGE OBJECT REFERENCE class-name-1 ONLY. If the object referenced by identifier-1 is not the factory object of class-name-1, an exception EC-OO-CONFORMANCE condition occurs.
6. If interface-name-1 is specified, the implicit result is the same as for USAGE OBJECT REFERENCE interface-name-1. If the object referenced by identifier-1 does not conform to interface-name-1, an exception condition EC-OO-CONFORMANCE occurs.
7. If UNIVERSAL is specified, the implicit result is the same as for USAGE OBJECT REFERENCE without any of the optional phrases. No exception condition occurs.
8. If the exception condition EC-OO-CONFORMANCE has occurred and the check for this exception condition is activated, the associated exception status is triggered and control is transferred to the relevant USE procedure.

2.5.8 Predefined object references

General format

```
{ NULL | SELF | [class-name-1 OF] SUPER }
```

2.5.8.1 NULL

NULL is a predefined object reference that contains the “null” object reference value, which never references an object.

Format

NULL

Syntax rules

1. NULL may not be used as a receiving operand.
2. NULL is not a universal object reference.
3. NULL is of the class object.

2.5.8.2 SELF and SUPER

SELF and SUPER are predefined object references that reference the object on which the current method is executing.

Format

```
SELF
```

```
[class-name-1 OF] SUPER
```

Syntax rules

1. SELF and SUPER may only be used in a method definition.
2. SELF and SUPER may not be specified as a receiving operand.
3. SUPER may be specified only as the object for which a method is called with the INVOKE statement.
4. class-name-1 is the name of a class specified in the INHERITS clause of the containing class definition.
5. If the INHERITS clause of the containing class definition specifies more than one class-name, then class-name-1 must be specified.
6. If the INHERITS clause of the containing class definition specifies only one class-name, then class-name-1 need not be specified.
7. SELF and SUPER are both of the class object and are not universal object references.

General rules

1. SELF and SUPER both reference the object that was used to invoke the method in which the reference to SELF or SUPER appears.
2. If SELF is specified for a method invocation, the method resolution is based upon the set of methods defined for the runtime class of the object dynamically referenced by SELF.
3. If SUPER is specified for a method invocation, the method resolution ignores all the methods defined in the class containing the invocation and all the methods defined in any subclass of that class. Thus the invoked method can only be one that is inherited from a superclass.
4. If class-name-1 is specified, the search for the method will occur only in the methods defined for it.

2.5.9 Predefined address NULL

NULL is a predefined address of the “pointer” data class.

Format

```
NULL
```

Syntax rules

1. This format can only be used in the following cases:
   - as a parameter in a CALL statement with a program prototype
   - as a parameter in a method call
   - as the sending item in a SET statement or INITIALIZE statement
   - in a relation condition with a data pointer or program pointer

General rules

1. When used together with a data pointer, the predefined address NULL indicates a data item of the class “pointer”, which does not contain a valid address for data in COBOL.
2. When used together with a program pointer, the predefined address NULL indicates a data item of the data class “pointer”, which does not contain a valid address for programs in COBOL.

2.5.10 Data address identifier

Format

```
ADDRESS OF identifier-1
```

Syntax rules

1. identifier-1 references an elementary item which is defined in the FILE SECTION, WORKING-STORAGE SECTION, LOCAL-STORAGE SECTION or LINKAGE SECTION.
   identifier-1 may not be defined in the WORKING-STORAGE SECTION or FILE SECTION of an object or factory object.
2. identifier-1 may not reference an object reference, a pointer or an index. identifier-1 may not reference an elementary data item which is subordinate to a strongly typed group data item.
3. The data address identifier may not be used as a receiving item.
4. If identifier-1 references a strongly typed group data item, identifier-1 must be described on level number 01.

General rules

1. A data address identifier generates a specific elementary item of class “pointer” and category “data pointer” which contains the address of identifier-1.
2. If identifier-1 is strongly typed, the data address identifier is a data pointer to a data item of the same type as identifier-1.

2.5.11 Program address identifier

Format

```
ADDRESS OF PROGRAM {identifier-1 | literal-1}
```

Syntax rules

1. identifier-1 must belong to the data category “alphanumeric” and must not be defined in the REPORT SECTION.
   If identifier-1 is the program name of an individual program or of the outermost program of a nested program (programs nested within others are not permissible), it must begin with an alphabetic character and can only contain uppercase letters and digits. The permissible length of the program name depends on the module format (see the "COBOL2000 User Guide" [1]).
2. literal-1 must be an alphanumeric literal; its value must be a valid program name, as described in 1.
3. The program address identifier may not be used as a receiving item.

General rule

A program address identifier creates a specific elementary item of the class “pointer” and the category “program pointer” that contains the address of an entry point of the program. The name of the entry point is contained in identifier-1 or literal-1.
If an error occurs during processing of the program address identifier, the program is aborted with a runtime error message.

2.5.12 BYTE-LENGTH OF

Format

```
BYTE-LENGTH OF identifier-1
```

Syntax rules

1. BYTE-LENGTH OF identifier-1 can be specified at any point in the PROCEDURE DIVISION where a numeric literal is permitted. This includes subscripts and reference modifiers.
2. identifier-1 must not be subject to reference modification.
3. If identifier-1 is a table element then it is not necessary to specify a subscript. If the subscript is specified then the compiler simply evaluates the format

   > ¡ Since the value of the subscript is not relevant for the evaluation of the length, the compiler simply determines whether the sequence is correct within the expression and whether identifier-1 is defined. The data type is not checked.

4. identifier-1 must not be defined using the ANY LENGTH clause.

General rules

1. BYTE-LENGTH OF represents the length of identifier-1 in **bytes**.
2. BYTE-LENGTH OF identifier-1 is a constant which is calculated at compilation time.If identifier-1 is a table with a variable number of elements, then the upper threshold for the OCCURS clause is used to calculate the constant BYTE-LENGTH OF identifier-1.

   > ¡ Since the length of the data described with the ANY LENGTH clause is not known until runtime, the only way to determine the length is to use the FUNCTION BYTE-LENGTH clause.

2.5.13 LENGTH OF

Format

```
LENGTH OF identifier-1
```

Syntax rules

1. LENGTH OF identifier-1 can be specified at any point in the PROCEDURE DIVISION where a numeric literal is permitted. This includes subscripts and reference modifiers.
2. identifier-1 must not be subject to reference modification.
3. If identifier-1 is a table element then it is not necessary to specify a subscript. If the subscript is specified then the compiler simply evaluates the format

   > ¡ Since the value of the subscript is not relevant for the evaluation of the length, the compiler simply determines whether the sequence is correct within the expression and whether identifier-1 is defined. The data type is not checked.

4. identifier-1 must not be defined using the ANY LENGTH clause.

General rules

1. LENGTH OF represents the length of identifier-1 in **characters**.
2. LENGTH OF identifier-1 is a constant which is calculated at compilation time. If identifier-1 is a table with a variable number of elements, then the upper threshold for the OCCURS clause is used to calculate the constant LENGTH OF identifier-1.

   > ¡ Since the length of the data described with the ANY LENGTH clause is not known until runtime, the only way to determine the length is to use the FUNCTION LENGTH clause.
   >
   > The usage of LENGTH OF is suggestive only in combination with USAGE DISPLAY or USAGE NATIONAL.

2.5.14 Condition-name

If referenced explicitly, a condition-name must be unique or be made unique by means of qualification and/or subscripting. This is not necessary if the uniqueness of the reference is guaranteed by the naming conventions for the scope themselves.

If qualification is used in order to render a condition-name unique, the associated conditional variable can be used as the first qualifier. Also, in the case of qualification, the hierarchy of names that are assigned to the conditional variable must be used in order to render a condition-name unique.

If the reference to a conditional variable necessitates subscripting, the same combination of subscripts is required when referencing one of its condition-names.

With regard to the qualification and subscripting of condition-names, the same format and restrictions are applicable as for the identifiers except that data-name-1 is replaced by condition-name-1.

In the general format in the following sections, "condition-name-n" always refers to a condition-name that, depending on the requirements, is qualified or subscripted.

2.6 Table handling

A table is a series of data items of equal length. These items are the table elements, or table items. They all have an identical structure and are stored contiguously. The entire table itself also forms a data item in COBOL terms.

Problems arising during the processing of large amounts of identically structured data can often be solved more satisfactorily by putting this data into tabular form. This allows an effective interpretation and meaningful representation of the information involved.

The homogeneous structure of the individual table elements makes their relationship to one another readily apparent.

The individual table element occupies an easily determined physical location relative to the base of the table, i.e. to the start of the table in working storage. Thus every element can be referenced relative to the beginning of the table and does not need to be assigned a unique data-name. A table element is accessed with the aid of a table element number, or occurrence number (see section "Subscripting" and section "Indexing").

In addition, it is possible to determine the associated occurrence number for any given value of a table element (see the section "SEARCH statement").

The number of table elements in a table may be variable at object time (see “Example 7-6” in section "Data-name or FILLER clause").

2.6.1 Table definition

A table element is indicated in the data description entry by specifying the OCCURS clause. This clause defines how many elements the table contains. The name and description of the table item apply to each recurrence thereof. In the case of multi-dimensional tables, each dimension in the hierarchical structure must be given an OCCURS clause.

Example 2-13

```
01  TABLE1.
    02 TABLE-ELEMENT PIC XXX OCCURS 20 TIMES.
```

The data item TABLE1 comprises 20 data items of identical length. These items are given the name TABLE-ELEMENT:

```
TABLE1: 1. TABLE-ELEMENT (1)   PIC XXX.
        2. TABLE-ELEMENT (2)   PIC XXX.
        .
        .
        .
       20. TABLE-ELEMENT (20)  PIC XXX.
```

One-dimensional tables

The OCCURS clause is entered in the data description entry of the table element.

Example 2-14

```
01  TABLE2.
    02 TABLE-ELEMENT OCCURS 2 TIMES.
       03 ELEMENT-ITEM-1      PIC X(4).
       03 ELEMENT-ITEM-2      PIC X(4).
```

TABLE2 is the name of the table.

TABLE-ELEMENT is the element which occurs twice within the one-dimensional TABLE2.

ELEMENT-ITEM-1 and ELEMENT-ITEM-2 are elements which are subordinate to TABLE-ELEMENT.

Multi-dimensional tables

When a data item is subordinate to a table-element within a two-dimensional table and contains an OCCURS clause, then this data item is an element within a three-dimensional table.

Up to seven dimensions are allowed for a single table.

Example 2-15

```
01 TABLE3.
   02 BLK OCCURS 2 TIMES.
      03 RECORD  OCCURS 2 TIMES.
         04 ITEM  OCCURS 2 TIMES PIC X(10).
```

BLK is an element which occurs twice within a one-dimensional table.

RECORD is an element in a two-dimensional table. It occurs twice within each occurrence of BLK.

ITEM is an element in a three-dimensional table. It occurs twice within each occurrence of RECORD.

TABLE	BLK (1)	RECORD (1, 1)	ITEM (1, 1, 1)
			ITEM (1, 1, 2)
		RECORD (1, 2)	ITEM (1, 2, 1)
			ITEM (1, 2, 2)
	BLK (2)	RECORD (2, 1)	ITEM (2, 1, 1)
			ITEM (2, 1, 2)
		RECORD (2, 2)	ITEM (2, 2, 1)
			ITEM (2, 2, 2)

Figure 3: Schematic representation of TABLE

Initial values of table elements

A VALUE clause must not appear in a record description entry with an OCCURS clause, or in any record description entry subordinate to that entry. However, for the definition of condition-names, the VALUE clause is allowed and required here as well.

Initial values may be assigned to a table in the WORKING-STORAGE SECTION or LOCAL-STORAGE SECTION by using the VALUE clause.

Example 2-16

```
 WORKING-STORAGE SECTION.
****** 1. VALUE ON GROUP-LEVEL ******

 01  WOCHE  VALUE
            "MONTAG   DIENSTAG  MITTWOCH  DONNERSTAG
            "FREITAG  SAMSTAG   SONNTAG   ".
     02  TAG  PIC X(10) OCCURS 7 TIMES.

****** 2. REPEATED VALUE WITH OCCURS ******

 01  WEEK.
     02  WDAY  PIC X(10) OCCURS 7 TIMES VALUE FROM (1)
               "MONDAY" "TUESDAY" "WEDNESDAY" "THURSDAY"
               "FRIDAY" "SATURDAY" "SUNDAY".

****** 3. REPEATED VALUE SUBORDINATE TO OCCURS ******

 01  UGE.
     02  FILLER      OCCURS 7 TIMES.
         03  DAG   PIC X(10) VALUE FROM (1)
                   "MANDAG" "TISDAG" "ONSDAG" "TORSDAG"
                   "FREDAG" "LOERDAG" "SOENDAG".
```

References to table elements

All the elements within a table have the same data-name. To identify individual occurrences of table elements, occurrence numbers (indexes) enclosed in parentheses are appended to the data-name.

Example 2-17

```
01  TABLE4.
    02  ELEMENT  OCCURS 10 TIMES.
          .
          .
          .
    MOVE ELEMENT OF TABLE4 (8) TO ...
```

Here the eighth table element is accessed.
An occurrence number must be supplied for each dimension.

There are two techniques for referencing table elements:

- subscripting
- indexing

2.6.2 Subscripting

One method of specifying occurrence numbers is to append one or more subscripts to the data-name. A subscript is an integer whose value represents the occurrence number of a table element or one of the items subordinate to that table element. The subscript may be represented

- by an integer literal
- by a data-name defined as a numeric elementary data item without any character positions to the right of the assumed decimal point
- by an arithmetic expression that is neither a direct nor a relative subscript.

In either case, the subscript must be enclosed in parentheses and must be written immediately following any qualification for the name of the table element. The referenced table element must have appended to it as many subscripts as the associated table has dimensions. A subscript must therefore be supplied for each OCCURS clause, including the OCCURS clause which contains the data-name within the defined hierarchy.

In example 2-13 in section "Table definition" (three-dimensional table), the following are required:

- one subscript for references to BLK
- two subscripts for references to RECORD
- three subscripts for references to ITEM.

Subscripts are written proceeding from the outermost to the innermost table.

Thus, for example,

ITEM (1, 2, 2)

identifies the second element ITEM

within the second element RECORD

within the first element BLK.

A reference to a data item must not be subscripted unless the data item is a table element, or unless it is an item or condition-name within a table element.

There are three forms of subscripting:

- direct subscripting
- relative subscripting
- subscripting by means of an arithmetic expression

Direct subscripting

With direct subscripting, the subscript is specified either by an integer literal or by a data-name. The data-name must be defined as a numeric elementary item with no character positions to the right of the assumed decimal point. In the preceding example, direct subscripting was used.

Relative subscripting

If the name of the table element is followed by a subscript in the form:

```
(data-name + integer-1),
```

then the occurrence number required to complete the reference is calculated from the value of data-name at object time, plus integer-1.

If it takes the form:

```
(data-name - integer-2),
```

the occurrence number is obtained by subtracting integer-2 from data-name.

Relative subscripting is treated in the same manner as relative indexing. For further details see section "Indexing".

Subscripting by means of an arithmetic expression

A subscript can consist of an arithmetic expression that supplies an integer as its result.

If a subscript consists of an arithmetic expression that is neither a direct nor a relative subscript, then the required occurrence number is calculated from the value of the arithmetic expression at object time.

At both compile time and object time, arithmetic expressions are processed more slowly than direct or relative subscripts. For this reason, swapping of data-name and integer in relative subscripts should be avoided, as too should the enclosure of a direct or relative subscript in parentheses since such expressions are considered to be arithmetic expressions.

If a subscript ends with a data-name or an index, then an immediately following subscript must not begin with a left parenthesis since this would initiate subscripting of the data-name or index.

2.6.3 Indexing

Another technique for referencing table elements is indexing. Indexing is made possible by supplying the INDEXED BY phrase in the OCCURS clause.

The index does not require its own data description entry. At object time, the value of an index is a binary value representing a displacement from the beginning of the table. The value of this binary number is calculated from the number and length of the table element as follows:

binary value of index = (occurrence number - 1) * length of table element

The value of an index may only be set using the SET, SEARCH or PERFORM statement. The initial value is undefined and must be set explicitly.

There are two forms of indexing:

- direct indexing
- relative indexing

Direct indexing

Direct indexing obtains when an index is used in the manner of a direct subscript.

Example 2-18

```
01 TABLE1.
   02  TABLE-A PIC XX OCCURS 10 TIMES INDEXED BY INDEX-A.
   02  TABLE-B PIC X(3) OCCURS 5 TIMES INDEXED BY INDEX-B.
          .
          .
   SET INDEX-A TO 7.
          .
   MOVE "X7" TO TABLE-A (INDEX-A).
```

Two tables are defined here:

- TABLE-A with 10 elements, each 2 bytes long
- TABLE-B with 5 elements, each 3 bytes long

INDEX-A is declared for TABLE-A and INDEX-B is declared for TABLE-B by means of the INDEXED BY phrase. Indices may only be used with the corresponding elementary item, e.g. TABLE-A(INDEX-A) or TABLE-B(INDEX-B). The SET statement sets the index to a value that points to the seventh element of TABLE-A. The displacement from the start of the table, i.e. the internal binary contents of INDEX-A, is (7-1) * 2 = 12. Thus, the MOVE statement transfers X7 to the seventh table element.

Relative indexing

When the name of a table element is followed by an index in the form

(index + integer-1),

then the required occurrence number is calculated from the value of index-name at object time, plus integer-1.

If the form

(index - integer-2),

is used, then the new occurrence number is obtained by subtracting integer-2 from the corresponding current occurrence number.

The use of relative indexing will not change the values of the index-names in the object program.

Permissible value ranges for indices

As specified in the standard, the value of an index should correspond to a valid occurrence number of the associated table. This compiler also permits corresponding occurrence numbers for 0, ZERO, or negative numbers, and values beyond the maximum permissible occurrence numbers, if the binary value of the index remains within the range (representable in 4 bytes) -2^{31} to $+2^{31}-1$. In these cases, the index must be set (e.g. with SET UP or SET DOWN) to a valid occurrence number before it is used, or corresponding relative indexing must be used to ensure that only valid table elements are addressed.

2.6.4 Indexing and subscripting compared

Availability of occurrence numbers for the user

Subscripting:

The occurrence number is immediately available.

Indexing:

The occurrence number is available only if preceded by a SET, SEARCH or PERFORM statement. It is calculated as follows:

value of index divided by length of table element plus 1.

References to table elements

Subscripting:

At object time, the address of the table element must be calculated anew each time from subscripts (except in the case of literal subscripts), i.e. subscripted data items cannot be referenced as quickly as data items outside the table.

Indexing:

When indexing is used, references to a table element are faster than subscripting using identifiers, since the displacement from the start of the table is already stored in the index.

Changing the index

Subscripting:

Changing a subscript in the form of a data-name (using MOVE, ADD etc.) is faster than changing an index-name using SET, since the SET statement requires that the occurrence number must first be converted to the displacement from the start of the table. This applies when the index is not being set up or down by a fixed integer value.

Indexing:

It is faster to change an index using PERFORM or SEARCH statements than to change a subscript.

Validity

Subscripting:

A subscript can also be used for other table elements.

Indexing:

An index may only be used with its associated table element (except in SET, PERFORM and SEARCH statements).

2.7 Statements and sentences

There are four types of statements:

- conditional statements
- compiler-directing statements
- imperative statements
- delimited scope statements

There are three types of sentences:

- conditional sentences
- compiler-directing sentences
- imperative sentences

2.7.1 Conditional statements and conditional sentences

- A **conditional statement** is used to determine the truth value of a condition and to specify the subsequent action in the object program on the basis of this value.

  Conditional statements include:

  - IF, EVALUATE, SEARCH and RETURN statements;
  - OPEN DOCUMENT statements which contain the (NOT) AT-END phrase;
  - READ statements which contain the (NOT) AT END or (NOT) INVALID KEY phrase;
  - WRITE statements which contain the (NOT) INVALID KEY or (NOT) END-OF-PAGE phrase;
  - START, REWRITE or DELETE statements which contain the (NOT) INVALID KEY phrase;
  - ADD, COMPUTE, DIVIDE, MULTIPLY or SUBTRACT statements which contain the (NOT) ON SIZE ERROR phrase;
  - STRING and UNSTRING statements which contain the (NOT) ON OVERFLOW phrase;
  - CALL statements which contain the ON OVERFLOW or (NOT) ON EXCEPTION phrase;
  - ACCEPT or DISPLAY statements which contain the (NOT) ON EXCEPTION phrase;
  - an INVOKE statement containing the (NOT) ON EXCEPTION phrase.

- A **conditional sentence** is a conditional statement which may optionally be preceded by an imperative statement and which is terminated by a period followed immediately by a space.

2.7.2 Compiler-directing statements and compiler-directing sentences

A **compiler-directing statement** consists of one of the compiler-directing verbs COPY, REPLACE or USE and the associated operands or one of the following compiler directives:

- >>CALL-CONVENTION
- >>DEFINE
- >>EVALUATE
- >>FLAG-85
- >>IF
- >>IMP
- >>LISTING
- >>PAGE
- >>SOURCE

A compiler-directing statement causes the compiler to perform certain actions during compilation.

2.7.3 Imperative statements and imperative sentences

- An imperative statement causes a specific action to be carried out in the object program.
- An imperative statement is a statement which begins with an imperative verb and specifies that an action is to be executed unconditionally or a conditional statement which is delimited by its explicit scope terminator.
- An imperative statement may consist of a sequence of imperative statements, each separated from the next by a COBOL separator.
  The imperative statements are:

ACCEPT (7)	FREE	READ (4)
ADD (1)	GENERATE	RELEASE
ALLOCATE	GO TO	RESUME
ALTER	INITIALIZE	REWRITE (2)
CALL (6)	INITIATE	SET
CANCEL	INSPECT	SORT
CLOSE	INVOKE (7)	START (2)
CLOSE DOCUMENT	MERGE	STOP
COMPUTE (1)	MOVE	STRING (3)
CONTINUE	MULTIPLY (1)	SUBTRACT (1)
DELETE (2)	OPEN	TERMINATE
DISPLAY (7)	OPEN DOCUMENT (8)	UNSTRING (3)
DIVIDE (1)	PERFORM	WRITE (5)
EXIT	RAISE	XML (7)

  (1) without the (NOT) ON SIZE ERROR phrase

  (2) without the (NOT) INVALID KEY phrase

  (3) without the (NOT) ON OVERFLOW phrase

  (4) without the (NOT) AT END or (NOT) INVALID KEY phrase

  (5) without the (NOT) INVALID KEY or (NOT) END-OF-PAGE phrase

  (6) without the ON OVERFLOW or (NOT) ON EXCEPTION phrase

  (7) without the (NOT) ON EXCEPTION phrase

  (8) without the (NOT) AT END phrase

- Whenever "imperative-statement" occurs in the general format of statements, it also refers to a sequence of imperative statements terminated either by a period or by a phrase that belongs to a statement that contains an imperative statement. For example, the statement

  ```
  DIVIDE A INTO B.
  ```

  is an imperative statement because of the period that concludes it,

  as is `DIVIDE` in

```
READ D AT END
       DIVIDE A INTO B
       NOT AT END ...
```

because of “NOT ...” .

- An imperative sentence is an imperative statement which is terminated by a period followed immediately by a space.

2.7.4 Delimited scope statements

A delimited scope statement is any statement which includes an explicit scope terminator.

2.7.5 Scope of statements (scope terminators)

Scope terminators delimit the scope of certain Procedure Division statements.Statements which include their explicit scope terminators are termed delimited scope statements.
The scope of statements which are contained within other statements (nested) may also be implicitly terminated. When statements are nested within other statements, a separator period which terminates the sentence also implicitly terminates all nested statements.

Whenever any statement is contained within another statement, the next phrase of the containing statement following the contained statement terminates the scope of any unterminated contained statement.
When a delimited scope statement is nested within another delimited scope statement with the same verb, each explicit scope terminator terminates the statement begun by the most recently preceding, and as yet unterminated, occurrence of that verb.
When statements are nested within other statements which allow optional conditional phrases, any optional conditional phrase encountered is considered to be the next phrase of the nearest preceding unterminated statement.
An unterminated statement is one which has not been previously terminated either explicitly or implicitly.

In addition to the separator period (implicit scope terminator), the following explicit scope terminators can be used to support structured programming:

END-ACCEPT	END-DIVIDE	END-PERFORM	END-START
END-ADD	END-EVALUATE	END-READ	END-STRING
END-CALL	END-IF	END-RECEIVE	END-SUBTRACT
END-COMPUTE	END-INVOKE	END-RETURN	END-UNSTRING
END-DELETE	END-MULTIPLY	END-REWRITE	END-WRITE
END-DISPLAY	END-OPEN	END-SEARCH	END-XML

Explicit scope terminators are reserved COBOL words.

2.8 Processing a COBOL program

The COBOL compiler, with the aid of a linkage editor, generates an executable program from a COBOL compilation unit. This executable program is also called an **object program** or **object module**.

The **linkage editor** links the modules generated by the compiler with the necessary runtime subroutines and, if applicable, with further compiler-generated modules.

The **runtime subroutines**, which are supplied as modules, are used for performing special COBOL functions such as input-output operations.

The interactive debugging aid AID is available for symbolic and hardware-oriented testing of COBOL programs.

See the "COBOL2000 User Guide" [1] for further details.

2.9 EBCDIC character set

Version of the 8-bit code used by the compiler

Decimal	Hexadecimal	EBCDIC	Printer graphics	
0	00	0000 0000		(LOW-VALUE)
...	...	...		
64	40	0100 0000		(space)
...	...	...		
74	4A	0100 1010	c	(cents)
75	4B	0100 1011	.	(period)
76	4C	0100 1100	<	(less than)
77	4D	0100 1101	(	(left parenthesis)
78	4E	0100 1110	+	(plus)
79	4F	0100 1111	|	(vertical)
80	50	0101 0000	&	(ampersand)
...	...	...		
90	5A	0101 1010	!	(exclamation mark)
91	5B	0101 1011	$	(dollar sign)
92	5C	0101 1100	*	(asterisk)
93	5D	0101 1101	)	(right parenthesis)
94	5E	0101 1110	;	(semicolon)
95	5F	0101 1111	?	(logical NOT)
96	60	0110 0000	-	(minus)
97	61	0110 0001	/	(slash)
98	62	0110 0010	§	(paragraph symbol)
99	63	0110 0011	[	(left square bracket)
100	64	0110 0100	]	(right square bracket)
...	...	...		
103	67	0110 0111	ß	(special German character)
...	...	...		
106	6A	0110 1010	^	(logical AND)

107	6B	0110 1011	,	(comma)
108	6C	0110 1100	%	(percent)
109	6D	0110 1101	_	(underline)
110	6E	0110 1110	>	(greater than)
111	6F	0110 1111	?	(question mark)
....	....	....	...	
122	7A	0111 1010	:	(colon)
123	7B	0111 1011	#	(number sign)
124	7C	0111 1100	@	(commercial at)
125	7D	0111 1101	'	(apostrophe)
126	7E	0111 1110	=	(equals)
127	7F	0111 1111	"	(quotionmark)
...	...	...		
129	81	1000 0001	a	
130	82	1000 0010	b	
131	83	1000 0011	c	
132	84	1000 0100	d	
133	85	1000 0101	e	
134	86	1000 0110	f	
135	87	1000 0111	g	
136	88	1000 1000	h	
137	89	1000 1001	i	
138	8A	1000 1010		
139	8B	1000 1011	Ä	
140	8C	1000 1100	Ö	
141	8D	1000 1101	Ü	
...	...	...		
145	91	1001 0001	j	
146	92	1001 0010	k	
147	93	1001 0011	l	

148	94	1001 0100	m
149	95	1001 0101	n
150	96	1001 0110	o
151	97	1001 0111	p
152	98	1001 1000	q
153	99	1001 1001	r
...	...	...	
159	9F	1001 1111	general currency symbol or €
...	...	...	(depending on character set)
162	A2	1010 0010	s
163	A	1010 0011	t
164	A4	1010 0100	u
165	A5	1010 0101	v
166	A6	1010 0110	w
167	A7	1010 0111	x
168	A8	1010 1000	y
169	A9	1010 1001	z
170	AA	1010 1010	
171	AB	1010 1011	ä
172	AC	1010 1100	ö
173	AD	1010 1101	ü
...	...	...	
192	C0	1100 0000	{
193	C1	1100 0001	A
194	C2	1100 0010	B
195	C3	1100 0011	C
196	C4	1100 0100	D
197	C5	1100 0101	E
198	C6	1100 0110	F
199	C7	1100 0111	G
200	C8	1100 1000	H

201	C9	1100 1001	I
...	...	...	
208	D0	1101 0000	}
209	D1	1101 0001	J
210	D2	1101 0010	K
211	D3	1101 0011	L
212	D4	1101 0100	M
213	D5	1101 0101	N
214	D6	1101 0110	O
215	D7	1101 0111	P
216	D8	1101 1000	Q
217	D9	1101 1001	R
...	...	...	
226	E2	1110 0010	S
227	E3	1110 0011	T
228	E4	1110 0100	U
229	E5	1110 0101	V
230	E6	1110 0110	W
231	E7	1110 0111	X
232	E8	1110 1000	Y
233	E9	1110 1001	Z
...	...	...	
240	F0	1111 0000	0
241	F1	1111 0001	1
242	F2	1111 0010	2
243	F3	1111 0011	3
244	F4	1111 0100	4
245	F5	1111 0101	5
246	F6	1111 0110	6
247	F7	1111 0111	7

248	F8	1111 1000	8	
249	F9	1111 1001	9	
...	...	...		
255	FF	1111 1111	~	(Tilde) (HIGH-VALUE)

Note

COBOL2000 supports the Euro symbol to the extent that the Euro symbol is allowed in character strings and comments of a COBOL program. The Euro symbol is also allowed in the CURRENCY-SIGN clause, and its use is supported in the PICTURE clause. More details on this can be found in the BS2000/OSD Softbooks [12].

3 Controlling the compiler

Both statements for source text manipulation and compiler directives are available for controlling the compiler. The tables below show which special statements and compiler directives exist and the logical compilation step in which they are effective (for details of the phases see the "COBOL2000 User Guide" [1].

Statements for source text manipulation	**Phase**
COPY statement	Source text manipulation
SUPPRESS option	Listing generation phase
REPLACE statement	Source text manipulation

Compiler directives	**Phase**
CALL-CONVENTION	Compilation phase
DEFINE	Source text manipulation
EVALUATE	Source text manipulation
FLAG-85	Compilation phase
IF	Source text manipulation
IMP	Compilation phase and listing generation phase (depending on the operand)
LISTING	Listing generation phase
PAGE	Listing generation phase
SOURCE FORMAT	Source text manipulation
TURN	Compilation phase

3.1 Statements for source text manipulation

The COPY and REPLACE statements can be used to extend and/or modify a source text with texts stored in a library. These two statements, which are effective in the source text manipulation phase can be used either independently or together.

A COPY statement extends an existing source text with COBOL text lines taken from a library element. The compiler then treats the inserted text as part of the compilation unit. The REPLACING phrase offers the additional possibility of replacing each occurrence of a literals, an identifier, a word or a group of words with another text.

The REPLACE statement causes parts of the source text to be replaced with new texts. With a REPLACE statement, compilation units written by the programmer in his self-defined notation can be converted at compile time into syntactically correct phrases, clauses and statements.

The compiler does not check the source text until execution of all COPY and REPLACE statements has been completed.

3.1.1 COPY statement

Function

The COPY statement inserts texts from a library into a compilation unit. At the same time, it permits parts of the new texts to be replaced.

Format

```
COPY text-name [{OF | IN} library-name] [SUPPRESS]

[REPLACING {  {  word-1             BY  {  word-2             }... ].
              | identifier-1              | identifier-2
              | literal-1                 | literal-2
              | ==pseudo-text-1==         | ==pseudo-text-2==
             }                           }
```

Syntax rules

1. text-name is the name of the text which is to be inserted in the compilation unit. The text is stored as a library element ("member") with this name in a library.
   text-name is a user-defined word with a length of 1 to 31 characters.
2. If text-name is qualified by library-name, the specified library is searched for the text. If text-name is not qualified by library-name, up to ten libraries are searched for the text. Assignment of libraries at compile time is described in more detail in the "COBOL2000 User Guide" [1].
3. The COPY statement must be preceded by a space or some other separator symbol.
4. pseudo-text-1 must contain at least one text word, (see “Text-word” in section "Glossary") while pseudo-text-2 may be empty (====) or may contain only spaces, comma, semicolon and comment lines.
5. Text words in pseudo-text-1 and pseudo-text-2 and text words which form identifier-1, identifier-2, literal-1, literal-2, word-1, word-2 may be continued on the next line. The pseudo-text separator (==), on the other hand, must not be continued.
6. word-1 and word-2 may be any single COBOL word except "COPY".
7. Lowercase letters that are used in text words are equivalent to the corresponding uppercase letters.
8. A COPY statement is recognized anywhere within a program except:
   - in comment lines
   - within non-numeric literals.
9. The text generated by one COPY statement must not contain another COPY statement. However, the compiler described here permits this in the outermost COPY:If a COPY statement does not contain the REPLACING clause, the related library text may contain COPY statements; these statements may contain the REPLACING clause.

General rules

1. Execution of a COPY statement causes the library text specified via text-name to be copied to the compilation unit. The library text then replaces the entire COPY statement (including the terminating separator period).
2. If SUPPRESS is specified, the library text is not included in the compilation unit list.
3. If REPLACING is not specified, the library text is copied unchanged.

COPY...REPLACING...

4. If REPLACING is specified, the library text is copied and the text preceding BY (A-operand) is replaced by the text following BY (B-operand).
5. Text is replaced only if each text word in the text preceding BY matches, character for character, the corresponding library text.
6. identifier-1, word-1 and literal-1 are treated as pseudo-text containing nothing but identifier-1, word-1 and literal-1.
7. The comparison of the library text with the text preceding BY (A-operand), the result of which decides whether or not a text is replaced, is carried out as follows:
   - Everything which precedes the first text word in the library text is copied into the compilation unit. Starting with the first of the A-operands (pseudo-text-1, identifier-1, word-1, literal-1), each A-operand is in turn compared with the corresponding number of library text words, starting each time with the first library word.
   - Each of the separators comma (,'BLANK'), semicolon (;'BLANK') or space in pseudo-text-1 or in the library text is treated as a single space. Each string of one or more spaces is treated as a single space.
   - If there is no match, the comparison is repeated with the next A-operand. This process is continued until a match is found or until there are no more A-operands.
   - When all A-operands of the REPLACING phrase have been compared with the library text without finding a match, the first word of the library text is copied into the compilation unit. The next library text word is then regarded as the first word and the comparison operation is started again with the first A-operand.
   - Whenever a match occurs between an A-operand and the library text, the corresponding B-operand (pseudo-text-2, identifier-2, word-2 or literal-2) is placed in the compilation unit and replaces the library text corresponding to the A-operand. The library text word following the last text word which was replaced is then regarded as the first text word and the comparison operation is repeated starting with the first A-operand.
   - The comparison cycle is repeated until the last library text word has been either copied or replaced.
8. Comment lines or empty lines in the library text or pseudo-text-1 are treated as space characters for the comparison. Any comment lines or empty lines in pseudo-text-2 are moved to the compilation unit unchanged. Comment lines and empty lines in the library text are copied unchanged into the compilation unit unless they are located within a text word sequence which matches pseudo-text-1 and which is therefore replaced.
9. Debugging lines are permitted within a library text or pseudo-text. Text words in a debugging line are subjected to the comparison rules as if there were no ’D’ in the indicator area (column 7) of the debugging line. A debugging line lies within a pseudo-text if the opening pseudo-text separator appears on a line which precedes the debugging line.
10. Each word copied from the library, but not replaced, is copied so that it starts in the same area of the line as in the library text.
    If replaced by pseudo-text, each text word begins in the same area as specified in pseudo-text-2. Text from a library is copied to the same area of the compilation unit which it occupied in the library. For this reason, it must comply with the rules of the COBOL reference format.
11. If there is a COPY statement in a debugging line, the copied text is treated as if it were also in debugging lines. This also applies to additional lines created by replacements in the compilation unit.
12. If a COPY statement with the REPLACING phrase causes a line to be lengthened such that additional lines are required, corresponding continuation lines are generated. These lines may contain a continuation character in column 7.

13. If parts of a text word are to be replaced, the string to be replaced must be enclosed between appropriate separators (e.g. colons or parentheses) in the library text and in the A-operand of the REPLACING phrase, thus identifying it as a separate text word (see "Example 3-4").
14. The reference format which applies for the COPY statement must be the same as that which also applies for the parts of the library text which may need to be replaced.

Example 3-1

Assume a German library text named ADR, stored in library ADRLIB and consisting of the following lines:

```
05  STRASSE    PIC X(20).
05  PLZ        PIC 9(5).
05  ORT        PIC X(20).
05  LAND       PIC X(20).
```

In a compilation unit, the COPY statement is written as follows:

```
01  ADRESSE.
    COPY ADR OF ADRLIB.
```

After execution of the COPY statement, the compilation unit contains:

```
01  ADRESSE.
    COPY ADR OF ADRLIB.    ————————————————————————— (1)
    05 STRASSE     PIC X(20).
    05 PLZ         PIC 9(5).
    05 ORT         PIC X(20).
    05 LAND        PIC X(20).
```

(1) Though appearing in the source listing, the COPY statement is treated as comments during compilation.

Example 3-2

In order to modify library text, the REPLACING phrase is specified in the COPY statement:

```
    COPY ADR OF ADRLIB
     REPLACING ADRESSE BY ADDRESS
               STRASSE BY STREET
               ==PLZ PIC 9(5)== BY ==POST CODE PIC X(8)==
               ORT BY TOWN
               LAND BY COUNTRY.
```

After execution of this COPY statement, the compilation unit contains:

```
01  ADRESSE.
    COPY ADR OF ADRLIB                                    (1)
     REPLACING ADRESSE BY ADDRESS                          |
               STRASSE BY STREET                           |
               ==PLZ PIC 9(5)== BY ==POST CODE PIC X(6)==  |
               ORT BY TOWN                                 |
               LAND BY COUNTRY.                         (1)
```

```
    05  STREET PIC X(20).
    05  POST CODE PIC X(6).
    05  TOWN PIC X(20).
    05  COUNTRY PIC X(20).
```

(1) Though appearing in the source listing, the COPY statement is treated as comments during compilation.

The changes only affect this compilation unit; the text in the library remains unchanged.

Example 3-3

In order to replace the following string in the library text:

```
... PIC X(30).
```

by the string

```
... PIC X(40).
```

a space must be inserted after the period in pseudo-text-1 and pseudo-text-2. Otherwise a search will be conducted for a period, but the library text contains a separator period:

```
...REPLACING ==PIC X(30). 'BLANK' == BY ==PIC X(40).'BLANK'==
```

Example 3-4

The library text in ELEM

```
01 FILLER
   02 :prefix:-name PIC X(30).
   02 :prefix:-address PIC X(30).
```

is expanded by the COPY statements

```
   ...
   COPY ELEM OF COPYLIB REPLACING ==:prefix:== BY ==in==.
   COPY ELEM OF COPYLIB REPLACING ==:prefix:== BY ==out==.
   ...
```

to

```
   ...
   01 FILLER
      02 in-name PIC X(30).
      02 in-address PIC X(30).
   01 FILLER
      02 out-name PIC X(30).
      02 out-address PIC X(30).
   ...
```

3.1.2 REPLACE statement

Function

The REPLACE statement is used to replace source text.

Format 1

```
REPLACE {==pseudo-text-1== BY ==pseudo-text-2==}... .
```

Format 2

```
REPLACE OFF.
```

Syntax rules

1. If a REPLACE statement is not the first statement in a separately compiled program, it must be preceded by the period separator character or some other separator symbol. One REPLACE statement must be terminated before another REPLACE statement is started.
2. A REPLACE statement is recognized anywhere within a program except:
   - in comment lines
   - within non-numeric literals
3. pseudo-text-1 must contain at least one word, while pseudo-text-2 may be empty (====) or may contain only spaces, comma, semicolon, and comment lines.

General rules

1. Format 1 of the REPLACE statement specifies that each occurrence of pseudo-text-1 is to be replaced by pseudo-text-2.
2. Format 2 of the REPLACE statement terminates the current text replacement operation.
3. A REPLACE statement remains effective from the point where it is specified up to the next REPLACE statement or up to the end of a separately compiled program.
4. All REPLACE statements in a compilation unit or in a library text are executed only after all COPY statements in the compilation unit or library text have been executed. The operands of a REPLACE statement cannot be modified with the REPLACING phrase of a COPY statement.
5. The text which results from execution of a REPLACE statement must not contain either a REPLACE statement or a COPY statement.
6. Lowercase letters that are used in text words are equivalent to the corresponding uppercase letters.
7. The comparison operation whose results determine whether or not a text is to be replaced is executed as follows (the term "source text" is used here to mean both the compilation unit text and the library text):
   - Starting with the first text word of the source text after the REPLACE statement and the first word of pseudo-text-1, pseudo-text-1 is compared with the corresponding number of consecutive text words.
   - pseudo-text-1 matches the source text only if the sequence of text words in pseudo-text-1 is identical, character for character, with the corresponding sequence of text words in the source text.
   - If no match is found, the comparison is repeated with each subsequent occurrence of pseudo-text-1 until a match is found or until there is no further pseudo-text-1.

- When each specified pseudo-text-1 has been compared with the source text without finding a match, the next word of the source text is regarded as the first text word and the comparison operation starts again with the first pseudo-text-1.
- If a match between pseudo-text-1 and the source text is found, this part of the source text is replaced with the corresponding pseudo-text-2. The text word in the source text which immediately follows the replaced text is then regarded as the first text word and the comparison operation starts again with the first pseudo-text-1.
- The comparison operation continues until the last text word affected by the REPLACE statement has been replaced or has been compared as the first text word with every pseudo-text-1.

8. Comment lines and empty lines in the source text or in pseudo-text-1 are treated as spaces for the comparison operation. The order of the text words in the source text and in pseudo-text-1 is defined by the reference format (see section "Statements and sentences").
   Any comment lines or empty lines in pseudo-text-2 are transferred unchanged to the compilation unit. Any comment line or empty line in the source text which lies within a sequence of text words which matches pseudo-text-1 will not appear in the final version of the program text.
9. Debugging lines are permitted in the pseudo-text. Text words in pseudo-text-1 which lie in a debugging line are treated, during comparison, as if there were no ’D’ in the indicator area (column 7).
10. Except for the COPY and REPLACE statements themselves, the syntax of the source text cannot be checked until all COPY and REPLACE statements have been fully executed.
11. Text words introduced by a REPLACE statement are placed in the program text in accordance with the reference format. When text words from pseudo-text-2 are moved to the program text, additional spaces are inserted only where they existed in the original text or in pseudo-text-2. This includes the implicit space between lines of the compilation unit.
12. If a REPLACE statement introduces additional lines to the compilation unit, the indicator area of each of these lines is marked with the character specified in the first line which was replaced. The only exception to this is where the original compilation unit line contained a hyphen: in this case, the inserted line contains a space.
    If a COPY statement with the REPLACING phrase causes a line to be lengthened such that additional lines are required, corresponding continuation lines are generated. These lines may contain a continuation character in column 7.
    If the replacement operation requires continuation of a literal in a debugging line, there is an error in the compilation unit.
13. The reference format which applies for the REPLACE statement must be the same as that which also applies for the parts of the library text which may need to be replaced.

3.2 Compiler directives

General

COBOL programmers can use compiler directive to specify compilation options and control the manipulation of the source text. In particular, compiler directives can be used to control conditional compilation.

Syntax rules for all compiler directives

1. A compiler directive can be preceded by any number of space characters.
2. A compiler directive must be specified in the program-text area (relative to the reference format) and may be followed only by space characters.
3. A compiler directive begins with the character string “>> “, followed by an optional space and the actual directive.
4. A compiler directive must be specified alone on a single line, except for the EVALUATE ("EVALUATE directive") and IF ("IF directive") directives.
5. A compiler directive may also be specified in library text (see COPY, section "Statements for source text manipulation").
6. EVALUATE and IF directives in COPY elements within the copied text must be terminated with the associated >>END-EVALUATE or >>END-IFCOPY elements.
7. The maximum nesting depth for directives is 256.
8. A compiler directive may be specified anywhere except in the following cases:
   - the use is restricted for certain directives by the syntax rules
   - within a source text manipulation statement (see section "Statements for source text manipulation")
   - between a continued line and a continuation line
   - in a debugging line

General rules for all compiler directives

1. A compiler directive is not affected by a "replacing action" of a COPY or REPLACE statement.
2. Compiler directives apply either in the source text manipulation phase, in the listing generation phase or in the compilation phase (see "Controlling the compiler").
   During the compilation phase the compiler directives are processed in the order in which they occur in the compilation group.
3. Compiler directives apply for all subsequent source texts and library texts and are independent of the program flow.
4. Compiler directives are regarded as coming **before** a compilation unit if this is not specified between the outermost IDENTIFICATION DIVISION (or, if this is missing, the ID paragraph) and the associated END entry.
5. If a compilation unit contains more than 65535 lines (this includes comment and empty lines and compiler directives which are specified before the IDENTIFICATION DIVISION and are not assigned to this compilation unit), directives which are normally effective during the compilation phase are ignored if they are specified inside this compilation unit.

Arithmetical expression

An arithmetical expression may consist of integers, constant names and the characters “(“, “)”, “+”, “-”, “*”, “/”. The range of values must be representable as a 32-bit number for all intermediate results. Decimal places may be truncated as necessary.

The following must apply for the range of values of the literal represented by the arithmetic expression as well as for the range of values of all the intermediate results derived during the calculation of the arithmetic expression:

-2^{31} < range of values < $2^{31}-1$

Boolean expression

The following constitute boolean expressions:

- A relation condition in which both operands are literals or arithmetic expressions which are formed from literals. Names of compilation variables from DEFINE directives can also be used in place of literals.
- DEFINED condition
- Expressions consisting of the logical operators AND, OR, NOT and any bracketed boolean expressions.

  Expressions in directives which contain logical operands (AND or OR) need to be fully bracketed in order to ensure that the logical operators are processed in the correct order. Only when partial logical expressions are used can additional bracketing be omitted and the usual rules of priority be applied.

Conditional compilation

Some compiler directives can be used to include or omit selected lines of source code in the compilation. This is called conditional compilation.
The DEFINE, EVALUATE and IF directives are intended for this purpose.

DEFINED condition

Format

```
compilation-variable-name-1 IS [ NOT ] DEFINED
```

Syntax rule

1. compilation-variable-name-1 must not be a word reserved for the compiler directives (see “Reserved words for compiler directives” in section "COBOL words").

General rules

1. If IS DEFINED is specified and compilation-variable-name-1 is defined, the condition will evaluate to “true”.
   if NOT DEFINED is specified and compilation-variable-name-1 is not defined, the condition will still evaluate to “true”.

   compilation-variable-name-1 is defined if the following conditions are satisfied:

   - compilation-variable-name-1 is specified in a DEFINE directive (see section "DEFINE directive") without an OFF clause.

- If the PARAMETER clause is specified in the DEFINE directive and the value was provided by the operating system (see the "COBOL2000 User Guide" [1]).

3.2.1 CALL-CONVENTION directive

The CALL-CONVENTION directive controls the handling of program and method names and determines the interface generation in the event of program and method calls.

Format

```
>>CALL-CONVENTION {  COMPATIBLE
                  | COBOL
                  | ILCS
                  | ILCS-SET-RETURN-CODE
                 }
```

General rules

1. If the CALL-CONVENTION directive is specified within a statement, it applies for the first time for the next statement.

   > The same CALL-CONVENTION phrase applies for WHEN phrases in EVALUATE and SEARCH statements as for the EVALUATE and SEARCH statements themselves.

2. If the CALL-CONVENTION directive is not specified, >>CALL-CONVENTION COMPATIBLE is assumed.
3. The table below shows the effect of the CALL-CONVENTION directive's operands.

Overview for CALL-CONVENTION

Effective for	Convention			
	COMPATIBLE	COBOL	ILCS	ILCS-SET-RETURN-CODE
Program name of external program (CALL...AS prototype (a))	converted to uppercase	converted to uppercase	converted to uppercase	converted to uppercase
Program name of external program (CALL otherwise (b), CANCEL, ADDRESS OF PROGRAM)	unchanged	converted to uppercase	unchanged	unchanged
Program name of nested program (CALL...AS NESTED)	converted to uppercase	converted to uppercase	converted to uppercase	converted to uppercase
Program name of nested program (CALL otherwise (b), CANCEL)	converted to uppercase	converted to uppercase	- (c)	- (c)
Method name (INVOKE with object reference (d))				

Method name (INVOKE otherwise (e))	unchanged	converted to uppercase	unchanged	unchanged
Set uppermost bit in last parameter	in accordance with option	no	no	no
Register1 transferred to special register RETURN-CODE after return from called register	in accordance with option ACTIVATE-XPG4-RETURNCODE	no	no	no

(a) Prototype name created from COBOL program without PROTOTYPE phrase and repository information generated by COBOL2000 compiler version > V1.2A

(b) Without prototype name or prototype name created from COBOL program with PROTOTYPE phrase or repository information generated by COBOL2000 compiler version <= V1.2A

(c) Call of nested programs only possible with AS NESTED

(d) Object reference SELF, SUPER or defined as USAGE OBJECT REFERENCE with class nname or ACTIVE-CLASS

(e) Universal object reference or object reference defined as USAGE OBJECT REFERENCE with interface name

3.2.2 DEFINE directive

The DEFINE directive defines a symbolic name for numeric literals or string literals which are used in an IF directive, an EVALUATE directive or the constant definition of a DEFINE directive.

Format

```
>>DEFINE compilation-variable-name AS {{arithmetical-expression | literal | PARAMETER
} [OVERRIDE] | OFF}
```

Syntax rules

1. If neither the OFF nor the OVERRIDE phrase is used, one of the following conditions must apply:
   - compilation-variable-name must not yet be defined within the compilation group.
   - The last preceding DEFINE directive that refers to compilation-variable-name must specify the same value as the OFF clause.
2. literal may be an integer numeric literal or a non-hexadecimal alphanumerical literal (see section "Literals"). The length of such a string literal is limited by the space available in a line.
3. arithmetical-expression must be formed as described on "Compiler directives".

General rules

1. The DEFINE directive is processed concurrently with the execution of the COPY and REPLACE statement.
2. The defined compilation variables may only be used in other directives in place of literals, but not in program text.
3. The OVERRIDE phrase can be used to define a new value for compilation-variable-name and ignore any previous value.
4. The PARAMETER phrase passes the value of the compilation variable to the compiler at start time (see the "COBOL2000 User Guide" [1]).
5. If OFF is specified or if an error occurs when the associated compilation variable is passed then compilation-variable-name has the status “undefined”.

3.2.3 EVALUATE directive

The EVALUATE directive supports the conditional compilation of multiple alternatives.

Format 1

```
>>EVALUATE {arithmetical-expression-1 | boolean-expression-1}

{>>WHEN {arithmetical-expression-2 | boolean-expression-2}
    [{THROUGH | THRU} arithmetical-expression-3]
        [source-text-1]}...

    [>>WHEN OTHER
        [source-text-2]]

    >>END EVALUATE
```

Format 2

```
>>EVALUATE TRUE

    {>>WHEN boolean-expression-1
        [source-text-1]}...

    [>>WHEN OTHER
        [source-text-2]]

    >>END-EVALUATE
```

Syntax rules

All formats:

1. THRU is the abbreviated form of THROUGH.
2. The parts of the compiler directive introduced with >> must be specified together with the succeeding operands, each in a separate line.
   source-text-1 and source-text-2 must begin on a new line.
3. source-text-1 and source-text-2 may contain any source code lines, including compil-text-1 and source-text-2 may each span multiple lines.
4. source-text-1 and source-text-2 may contain a maximum of 1 COPY statement in any line.

Format 1

1. All operands of one EVALUATE directive must be of the same category. For this rule, an arithmetic expression is of category “numeric” and a boolean expression is of category “boolean”.
2. If the THROUGH phrase is specified, all selection items must be of category “numeric”.

General rules

All formats

1. The EVALUATE directive is processed during the execution of the COPY and REPLACE statements.
2. As soon as a WHEN phrase evaluates to TRUE, the associated source-text-1 is compiles, including >>END-EVALUATE, are ignored. If none of the WHEN phrases evaluate to TRUE, then source-text-2, if specified, is compiled, and all other lines are ignored.

Format 1

1. arithmetic-expression-1 and boolean-expression-1 are compared with the values in the WHEN phrase in accordance with the following rules:
   - If THROUGH is not specified, the value ‘true’ is returned when arithmetic expression-1 or boolean-expression-1 is equal to arithmetic-expression-2 or boolean-expression-2,
   - If THROUGH is specified, the value ‘true’ is returned when arithmetic-expression-1 lies in the value range of arithmetic-expression-2 and arithmetic-expression-3.

3.2.4 FLAG-85 directive

The FLAG-85 directive is used to have language resources displayed by the compiler whose behavior has changed in comparison to the ANS85 standard or has been defined exactly.

Format

```
>>FLAG-85 ZERO-LENGTH {ON | OFF}
```

General rules

1. If the FLAG-85 directive is specified within a statement, it applies for the first time for the next statement.

   > The same FLAG-85 phrase applies for WHEN phrases in EVALUATE and SEARCH statements as for the EVALUATE and SEARCH statements themselves.

2. If no FLAG-85 directive is specified, ">>FLAG-85 ZERO-LENGTH OFF" is assumed.
3. If ON is specified or assumed, the check of the language elements is activated for the subsequent source text. The check remains activated until it is deactivated by another FLAG-85 directive with the OFF phrase.
4. If OFF is specified, the check of the language elements is deactivated for the subsequent source text. The check remains deactivated until it is activated again by another FLAG-85 directive with the ON phrase.
5. When the FLAG-85 directive is activated, compiler messages are issued in the following situations:
   - A READ statement which can supply a record with the length 0.
   - A statement in which a potential zero-length item is addressed.

3.2.5 IF directive

The IF directive supports the conditional compilation of one or two alternatives.

Format

```
>>IF boolean-expression-1
     [source-text-1]

   [>>ELSE
      [source-text-2]]

    >>END-IF
```

Syntax rules

1. The parts of the compiler directive introduced with >> must be specified together with the succeeding operands, each in a separate line.
   source-text-1 and source-text-2 must begin on a new line.
2. The source lines must be entered as specified in the format, and both source-text-1 and source-text-2 may span multiple lines. The source texts may also include compiler directives.
3. source-text-1 and source-text-2 may contain a maximum of 1 COPY statement in any line.

General rules

1. The IF directive is processed during the execution of COPY and REPLACE statements.
2. If boolean-expression-1 is true, source-text-1 is compiled if it exists, and source-text-2 is ignored.
3. If boolean-expression-1 is not true, source-text-1 is ignored, and source-text-2 is compiled if it exists.

3.2.6 IMP directive

The IMP directive enables the compiler to be controlled by being specified directly in the source program.

3.2.6.1 IMP COMPILER-ACTION

This directive enables actions of the compiler to be controlled during module generation (see the "COBOL2000 User Guide" [1]).

3.2.6.2 IMP LISTING-OPTIONS

This directive enables the values of compiler options which influence the listings generated by the compiler to be modified (see the "COBOL2000 User Guide" [1]).

3.2.6.3 IMP PRINT-DIRECTIVES

This directive enables the values of directives to be output in the source listing (see the "COBOL2000 User Guide" [1]).

3.2.6.4 IMP RUNTIME-ERRORS

This directive permits control of the checking and handling of runtime errors (see the "COBOL2000 User Guide" [1]).

3.2.6.5 IMP SET-DIRECTIVES

This directive enables directive values to be reset to their default.

Format

```
>>IMP SET-DIRECTIVES {ALL | IMPLEMENTOR-DEFINED} TO DEFAULT
```

Syntax rule

1. The directive may only be specified before a compilation unit.

General rules

1. The directive applies in the compilation phase.
2. When ALL is specified, all directives for which a default value exists are reset to the default.
3. When IMPLEMENTOR-DEFINED is specified, all IMP directives for which a default value exists are reset to the default.
4. The default value of directives which correspond to a compiler option is the value which was specified when the compiler was called for this option (ON/OFF instead of YES/NO).

3.2.7 LISTING directive

The LISTING directive controls whether or not source text is to be listed.

Format

```
>>LISTING {ON | OFF}
```

General rules

1. If the compiler does not produce a source text listing (see "COBOL2000 User Guide" [1]), the LISTING directive is ignored; otherwise, the following rules apply:
2. The LISTING directive is executed after the COPY and REPLACE statements.
3. The >>LISTING ON/OFF directive is always listed, even if the listing itself is suppressed by a LISTING directive.
4. If OFF is specified then, except for another LISTING OFF directive, source text will not be listed until a LISTING ON directive is encountered.
5. If ON is specified or implied, source text will be listed until either a LISTING OFF directive is encountered or the end of the compilation group is reached.

3.2.8 PAGE directive

The PAGE directive generates a page feed and thus supports the documentation of the source listing.

Format

```
>>PAGE [comment]
```

Syntax rules

1. comment-text may contain any characters in the computer's coded character set.
2. comment-text is not checked syntactically.

General rules

1. comment-text is only intended for documentation purposes.
2. If a source listing is being produced, a PAGE directive generates a page feed; if no listing is being produced, a blank line is generated.

3.2.9 SOURCE FORMAT directive

The SOURCE FORMAT directive shows the reference format of the following source or library text.

Format

```
>>SOURCE [FORMAT IS] {FREE | FIXED }
```

General rules

1. If FIXED is specified, the subsequent source text or library text is treated as fixed format text. If FREE is specified, it is treated as free format text.
2. The default reference format of a compilation group is fixed format.
3. The reference format set for the COPY statement applies for the library text to be read in until a SOURCE FORMAT directive is specified in the library text.
4. SOURCE FORMAT directives in a library text apply until another SOURCE FORMAT directive is specified or the end of the library text is reached. Once processing of the library element has been completed, the reference format which applied for the initiating COPY statement applies again.

> The SOURCE FORMAT directive should be used in newly created COPY elements so that these COPY elements can in future also be used regardless of the format of the containing source program.

3.2.10 TURN directive

The TURN directive is used to activate or deactivate the checking of exception situations.

Format

```
>>TURN {exception-situation-name}... CHECKING {ON | OFF}
```

Syntax rules

1. exception-situation-name must be one of the names listed in table 45 in section "Exception conditions and exception statuses".
2. No exception-situation-name may be specified more than once in a TURN directive.

General rules

1. If the TURN directive is specified within a statement, it applies for the first time for the next statement.

   > The same TURN phrase applies for WHEN phrases in EVALUATE and SEARCH statements as for the EVALUATE and SEARCH statements themselves.

2. If no TURN directive is specified for an exception-situation-name, ">>TURN exception-situation-name CHECKING OFF" is assumed for this name.
3. If ON is specified or assumed, the check of the exception situation designated by exception-situation-name is activated for the subsequent source text. The check remains activated until it is deactivated by another TURN directive for exception-situation-name with the OFF phrase.
4. If OFF is specified, the check of the exception situation designated by exception-situation-name is deactivated for the subsequent source text. The check remains deactivated until it is activated again by another TURN directive for exception-situation-name with the ON phrase.

4 Structure of a COBOL compilation group

4.1 General description

A COBOL compilation group may contain one or more compilation units.
Apart from the source text manipulation statements and END markers, all statements, entries, paragraphs and sections of a COBOL compilation unit are grouped into four divisions, which are organized as follows:

1. Identification Division
2. Environment Division
3. Data Division
4. Procedure Division

The end of a COBOL compilation unit is indicated either by the END marker or by the absence of further compilation unit lines.

The beginning of a division is indicated by the corresponding program division header or the corresponding ID or FACTORY or OBJECT. The end of a division is indicated by

- the division header or ID or FACTORY or OBJECT of a succeeding division in the program or
- an END marker or
- the absence of further source lines.

Although a COBOL program may have an unlimited number of lines, the compiler will only number these lines uniquely up to a value of 65536.

4.2 COBOL compilation group

The following formats illustrate the general structure of a COBOL compilation group and the structure of the compilation units contained in it. A compilation unit can be compiled separately.

Format of the compilation group

```
{compilation unit}...
```

Compilation unit:

```
{  Program-prototype
 | Program-definition
 | Class-definition
 | Interface-definition
}
```

Program prototype:

```
[IDENTIFICATION DIVISION.]
PROGRAM-ID. {program-prototype-name-1} IS PROTOTYPE.
[environment-division]
[data-division]
[procedure-division]
END PROGRAM program-prototype-name-1.
```

Program definition; nested program definition:

```
[IDENTIFICATION DIVISION.]
PROGRAM-ID. program-name-1 [IS {| COMMON | {INITIAL |  RECURSIVE } |}  PROGRAM ].
[environment-division]
[data-division]
[procedure-division [nested program-definition]...]
[END PROGRAM program-name-1.]
```

Class definition:

```
[IDENTIFICATION DIVISION.]

CLASS-ID. class-name-1 [IS FINAL]
   [ INHERITS  FROM {class-name-2}...]
   [ USING  {parameter-name-1}...].

[environment-division]

[ [IDENTIFICATION DIVISION.]
  FACTORY.
  [environment-division]
  [data-division]
  [object-oriented procedure-division]
  END FACTORY.
]

[ [IDENTIFICATION DIVISION.]
  OBJECT.
  [environment-division]
  [data-division]
  [object-oriented procedure-division]
  END OBJECT.
]


END CLASS class-name-1.
```

Interface definition:

```
[IDENTIFICATION DIVISION.]

INTERFACE-ID. interface-name-1
   [INHERITS FROM {interface-name-2}...]
   [USING {parameter-name-1}...].

[environment-division]

[object-oriented procedure-division]

END INTERFACE interfacename-1.
```

Object-oriented Procedure Division

```
PROCEDURE DIVISION.
[{methods-definition}...]
```

Methods definition:

```
[IDENTIFICATION DIVISION.]

METHOD-ID. method-name-1 [OVERRIDE] [IS FINAL].

[environment-division]

[data-division]

[procedure-division]

END METHOD method-name-1.
```

Syntax rules

1. Within a compilation group, the program-prototype-definitions must precede all other compilation units.
2. If a compilation group contains a program definition and a program prototype definition with the same name, the interfaces of these two compilation units must be identical.
3. The Data Division of a method within an interface definition may contain only the Linkage Section.
4. The Procedure Division of a method within an interface definition may contain only a Procedure Division header.

General rules

1. In the program prototype compilation unit, all sections in the Data Division except for the Linkage Section and everything in the Procedure Division except for the Procedure Division header are ignored. The ignored parts must still contain the correct COBOL syntax.
2. The beginning of a compilation unit is indicated by the corresponding Identification Division. The end is indicated by one of the following:
   - the END marker
   - the end of the entire source code.
3. When an END marker indicates the end of an individual compilation unit, it is either followed by no further source code or by the Identification Division of the next individual compilation unit (for a sequence of compilation units).
4. The assignment of source lines between compilation units, i.e. to preceding or succeeding compilation units, is undefined.

4.3 END markers

Function

END markers indicate the end of a known source unit, where a source unit is a sequence of statements that begins with an Identification Division and ends with an END marker.

Format

```
END  {  PROGRAM program-prototype-name-1
      | PROGRAM program-name-1
      |  CLASS class-name-1
      | FACTORY
      | OBJECT
      | METHOD [method-name-1]
      | INTERFACE interface-name-1
     }.
```

Syntax rules

1. An END marker must be present in every source unit that contains, is contained in, or precedes another source unit.
2. If a PROGRAM-ID paragraph for program-name-2 is stated between a PROGRAM-ID paragraph for program-name-1 and its associated END marker, then the END marker for program-name-2 must precede the END marker for program-name-1.
3. program-name-1 must be identical to the program name declared in the PROGRAM-ID paragraph of the program to which the END marker refers.
4. class-name-1 must be identical to the class name declared in the corresponding CLASS-ID paragraph.
5. method-name-1 must be identical to the method name declared in the corresponding METHOD-ID paragraph.
6. interface-name-1 must be identical to the interface name declared in the corresponding INTERFACE-ID paragraph.
7. program-prototype-name-1 must be identical to one of the program prototype names declared in the corresponding PROGRAM-ID paragraph.

5 Identification Division

5.1 General description

A source unit begins with the IDENTIFICATION DIVISION and is specified in the associated paragraph header. This may involve a program, class, factory object, object, method or interface.

5.2 General format

```
[ {IDENTIFICATION DIVISION. | ID DIVISION.} ]

{  program-id paragraph
 | class-id paragraph
 | factory paragraph
 | object paragraph
 | method-id paragraph
 | interface-id paragraph
}
```

5.3 Paragraphs

5.3.1 PROGRAM-ID paragraph

Function

The PROGRAM-ID paragraph contains the name by which a program or program prototype is identified. The program can be assigned the appropriate attributes with the INITIAL, COMMON or RECURSIVE clause.

Format 1

```
PROGRAM-ID . program-name [ IS {| COMMON | { INITIAL | RECURSIVE } |}  PROGRAM].
```

Format 2

```
PROGRAM-ID. program-prototype-name-1 IS PROTOTYPE.
```

Syntax rules

1. The program-name must be a user-defined name with a length of 1-30 characters, beginning with a letter.

   Note:
   A program name should not begin with the letter "I" to avoid potential conflicts with the names of COBOL runtime modules or modules of other runtime systems.
2. The programs of a nested program must have different names.
3. The COMMON clause may only be specified for the contained programs of a nested program. It must not be specified in the outermost containing program.
4. The INITIAL clause may not be specified if any program that is directly or indirectly contained in a recursive program.
5. The RECURSIVE clause may not be specified if any program that is directly or indirectly contained in a program containing the INITIAL clause.

General rules for Format 1

1. The effect of the INITIAL clause is that the program is set to its initial state each time it is called (see section "Initial state for inter-program communication").
2. The COMMON clause causes the program to have the COMMON attribute. Such a program can be called not only by the directly superordinate program but also by its "sibling programs" and their "descendants".
3. The RECURSIVE clause specifies that the program and all programs contained within it are recursive. The program may be called while it is active and may call itself. If the RECURSIVE clause is not specified in a program or implied for a program, the program may not be called while it is active.

General rule for Format 2

1. program-prototype-name-1 specifies the program prototype.

Additional rules, depending on the module format

The following applies to program-name and program-prototype-name-1 when generating the *OMF format (see the "COBOL2000 User Guide" [1]):

1. The 8th character must not be a hyphen.

2. The operating system uses only the first eight characters of the module identification. Therefore, these characters should be unique for every name in a particular module/program library.
3. Only the first 7 characters of the name are used when generating shared code.

Example 5-1

of the effect of the INITIAL clause

Calling program:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MAIN.
...
DATA DIVISION.
WORKING-STORAGE SECTION.
...
PROCEDURE DIVISION.
...
    CALL "UPROG1" USING...  (1)
...
...
    CALL "UPROG1" USING...  (2)
```

Called program:

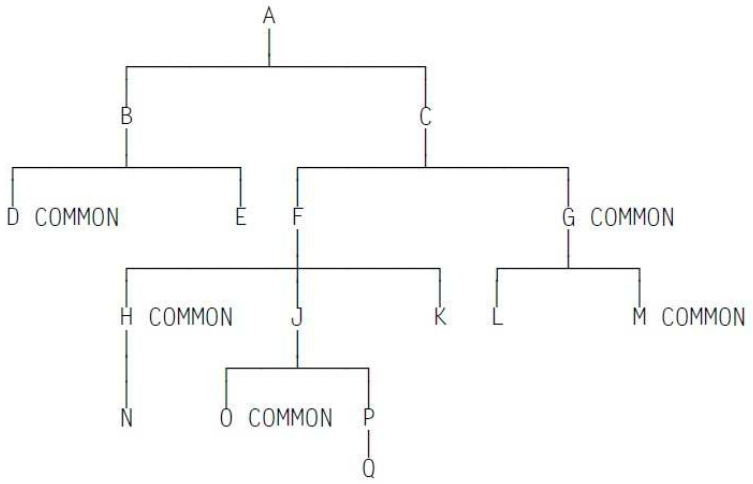
```
IDENTIFICATION DIVISION.
PROGRAM-ID. UPROG1 IS INITIAL PROGRAM.
...
DATA DIVISION.
WORKING-STORAGE SECTION.
01 USER-DATE PIC X(8) VALUE SPACE.
...
PROCEDURE DIVISION USING ...
...
    MOVE "26.10.49" TO USER-DATE.
...
    EXIT PROGRAM.
```

(1) UPROG1 is called for the first time by MAIN.

(2) UPROG1 is called for the second time by MAIN. UPROG1 is again in its initial state: the content of USER-DATE, for example, is again SPACE, even if it was changed during the first call of UPROG1.

Example 5-2

of the effect of the COMMON clause
Structure of a nested program A:



Valid call options within the nested program shown above:

Called program	Calling program															
	A	B	C	D	E	F	G	H	J	K	L	M	N	O	P	Q
A																
B	x															
C	x															
D		x			x											
E		x														
F			x													
G			x			x		x	x	x			x	x	x	x
H						x			x	x				x	x	x
J						x										
K						x										
L							x									
M							x				x					
N								x								
O									x						x	x
P									x							
Q															x	

x = permitted call

5.3.2 CLASS-ID paragraph

Function

The CLASS-ID paragraph indicates that this Identification Division introduces a class definition and specifies the name of the class and its attributes.

Format

```
CLASS-ID. class-name-1 [IS FINAL]
   [INHERITS FROM {class-name-2}...]
   [USING {parameter-name-1}...].
```

Syntax rules

1. class-name-1 must not be BASE.
2. class-name-2 is the name of a class, which must be specified in the REPOSITORY paragraph.
3. class-name-2 must not be the same as class-name-1.
4. class-name-2 may not inherit directly or indirectly from a class which is an expansion of class-name-1.
5. class-name-2 may not be defined with the FINAL clause.
6. If two or more different methods with the same name are inherited, none of them may be specified with the FINAL clause. If the same method is inherited from one superclass through two or more intermediate superclasses, it may be specified with the FINAL clause.
7. A given class-name-2 may not appear more than once in an INHERITS clause.
8. If a method of the same name is inherited from more than one inherited class, and if these methods have different interfaces, then a method with the same name of this method must be declared in this class, and its interface must satisfy syntax rule 5 listed under the METHOD-ID paragraph.
9. parameter-name-1 is the name of a class or interface which must be defined in the REPOSITORY paragraph.
10. The same parameter-name-1 may not occur more than once in a USING clause.
11. parameter-name-1 must be different from class-name-1.

General rules

1. class-name-1 names the class declared by this class definition.
2. The INHERITS clause specifies the names of classes that are inherited by class-name-1.
3. If the FINAL clause is specified, the class cannot be the superclass for any other class.
4. If the same class is inherited more than once, then only one copy of the data for that class is added to class-name-1.
5. The USING clause specifies this class as a parameterized class and lists the names of the parameters.

5.3.3 FACTORY paragraph

Function

The FACTORY paragraph indicates that this Identification Division introduces a factory definition.

Format

```
FACTORY. [IMPLEMENTS {interfacename-1}... .]
```

Syntax rules

1. interfacename-1 must be the name of an interface definition that is specified in the REPOSITORY paragraph.
2. The interface of the factory object must be compatible with all interfacename-1 interfaces.

General rules

If the IMPLEMENTS clause is specified, during compilation of a class the compiler checks whether the interface of its factory object is compatible with interfacename-1. If the IMPLEMENTS clause is missing, this check cannot be carried out until the object reference for the factory object is used in connection with interfacename-1.

5.3.4 OBJECT paragraph

Function

The OBJECT paragraph indicates that this Identification Division introduces an object definition.

Format

```
OBJECT. [IMPLEMENTS {interfacename-1}... .]
```

Syntax rules

1. interfacename-1 must be the name of an interface definition that is specified in the REPOSITORY paragraph.
2. The interface of the objects must be compatible with all interfacename-1 interfaces.

General rules

If the IMPLEMENTS clause is specified, during compilation of a class the compiler checks whether the interface of its factory object is compatible with interfacename-1. If the IMPLEMENTS clause is missing, this check cannot be carried out until the object reference for the factory object is used in connection with interfacename-1.

5.3.5 METHOD-ID paragraph

Function

The METHOD-ID paragraph indicates that this Identification Division introduces a method definition and specifies its name and its attributes.

Format

```
METHOD-ID. method-name-1 [OVERRIDE] [IS FINAL].
```

Syntax rules

1. If the OVERRIDE phrase is specified, there must be a method with the same method-name-1 in one of the inherited classes. The method in the superclass may not be defined with the FINAL clause.
2. If the OVERRIDE phrase is not specified, there should not be a method with the same method-name-1 in any of the inherited classes.
3. The OVERRIDE clause may not be specified in an interface method.
4. The FINAL clause may not be specified in an interface definition.
5. If method-name-1 is the same as a method-name inherited by the classes, the interface of method-name-1 must be conform with the interface of the inherited method(s).
6. The names of the methods in a class must be unique.

General rules

1. method-name-1 is the name of the method specified by the method definition.
2. The OVERRIDE phrase indicates that the method method-name-1 overrides any inherited method of the same name.
3. The FINAL clause indicates that this method may not be overridden in any inherited class using OVERRIDE.
4. method-name-1 can be used in the call to a method for an object of the class in which this method is defined.
5. If a given user-defined word is defined in the Data Division of this method definition and in the Data Division of the containing factory definition or object definition, the use of that word in this method refers to the declaration in this method. The declaration in the containing factory object or object definition is inaccessible to this method.

5.3.6 INTERFACE-ID paragraph

Function

The INTERFACE-ID paragraph indicates that this Identification Division introduces an interface definition and specifies the interface name and its attributes.

Format

```
INTERFACE-ID. interface-name-1
   [INHERITS FROM {interface-name-2}...]
   [USING {parameter-name-1}...].
```

Syntax rules

1. interface-name-1 must not be either BASEFACTORYINTERFACE or BASEINTERFACE.
2. interface-name-2 is the name of an interface, which must be specified in the REPOSITORY paragraph.
3. interface-name-2 may not inherit directly or indirectly from interface-name-1.
   interface-name-2 may not inherit directly or indirectly from an interface which is an expansion of interface-name-1.
4. If a given method-name is inherited from more than one interface, then each of these methods must have the same interface.
5. parameter-name-1 is the name of a class or interface which must be defined in the REPOSITORY paragraph.
6. The same parameter-name-1 may not occur more than once in a USING clause.
7. parameter-name-1 must be different from interface-name-1.

General rules

1. interface-name-1 names the interface declared by this interface definition.
2. The INHERITS phrase specifies the names of interfaces that are inherited by interface-name-1 according to the rules for interface inheritance.
3. The USING clause specifies this interface as a parameterized interface and lists the names of the parameters.

6 Environment Division

6.1 General description

The Environment Division provides a standard method for describing those aspects of a data processing problem which depend on the physical characteristics of a given computer installation. This division can be used to define the equipment configuration of the data processing system on which the program is to be compiled and executed. It also provides an opportunity for specifying input/output control, specific machine characteristics, and control techniques.

The Environment Division must be included in a COBOL compilation unit.

It consists of two optional sections:

1. CONFIGURATION SECTION
2. INPUT-OUTPUT SECTION.

The CONFIGURATION SECTION deals with the characteristics of the computers used for compiling and executing the program (source computer and object computer, respectively).

This section is divided into the following paragraphs:

1. the SOURCE-COMPUTER paragraph, which describes the equipment configuration of the data processing system on which the compilation unit is to be compiled
2. the OBJECT-COMPUTER paragraph, which describes the equipment configuration of the data processing system on which the object program is to be executed
3. the SPECIAL-NAMES paragraph, which (among other things) relates the implementor-names used by the compiler to the mnemonic-names used in the compilation unit.
4. the REPOSITORY paragraph, which specifies the interfaces, classes and program prototype names used in programs.

The INPUT-OUTPUT SECTION deals with the required information for controlling the transmission of data between external devices and the object program.

This section is divided into two paragraphs:

1. the FILE-CONTROL paragraph, which names the files and assigns them to external devices
2. the I-O-CONTROL paragraph, which describes special control techniques to be used in the object program.

General format

```
ENVIRONMENT DIVISION.

[configuration-section]
[input-output-section]
```

6.2 CONFIGURATION SECTION

Function

The CONFIGURATION SECTION makes up part of the Environment Division of a compilation unit and provides a means to do the following:

- describe the computer configuration on which the program is to be compiled or executed
- declare a currency sign
- choose the decimal point
- specify symbolic names for characters
- relate implementor-names to user-specified mnemonic-names
- relate alphabet-names to character sets or collating sequences
- relate class-names to user-defined sets of characters

Format

```
CONFIGURATION SECTION.

[SOURCE-COMPUTER.  [source-computer-entry [debugging-mode].]]
[OBJECT-COMPUTER.  [object-computer-entry.]]
[SPECIAL-NAMES.    [special-names-entry.]]
[REPOSITORY.       [specifier]]
```

Syntax rules

1. The CONFIGURATION SECTION and its associated paragraphs are optional.
2. If this paragraphs are specified, the sequence indicated must be observed.
3. The SOURCE-COMPUTER, OBJECT-COMPUTER, SPECIAL-NAMES and REPOSITORY paragraphs may not be specified in a FACTORY or OBJECT definition.
4. The CONFIGURATION SECTION may not be specified in a program that is contained within another program.
5. The CONFIGURATION SECTION may not be specified in a method definition.

6.2.1 SOURCE-COMPUTER paragraph

Function

The SOURCE-COMPUTER paragraph identifies the data processing system on which the compilation unit is to be compiled; it may also be used to specify debugging aids.

Format

```
SOURCE-COMPUTER. [computer-name [WITH DEBUGGING MODE]. ]
```

Syntax rule

1. computer-name must be a user-defined COBOL word.

General rules

1. All clauses of this paragraph apply to the program in which they are explicitly or implicitly specified.
2. When the paragraph is not specified, or if it is specified but the computer-name is not, the computer upon which the compilation unit is being compiled is the compilation computer.
3. If the WITH-DEBUGGING-MODE clause is specified, all debugging lines are compiled as specified in the rules described in section "Debugging lines" of chapter "Debugging".
4. If the WITH-DEBUGGING-MODE clause is not specified in a program, any debugging lines are compiled as if they were comment lines.

6.2.2 OBJECT-COMPUTER paragraph

Function

The OBJECT-COMPUTER paragraph describes the data processing system on which the program is to be executed.

Format

```
OBJECT-COMPUTER. [computer-name
    [MEMORY SIZE integer {WORDS | CHARACTERS | MODULES}]
    [PROGRAM COLLATING SEQUENCE IS alphabet-name].]
```

Syntax rules

1. computer-name and the MEMORY-SIZE clause are used for documentation purposes only and are treated as comments.
2. computer-name must be a user-defined COBOL word.
3. If the PROGRAM COLLATING SEQUENCE clause is specified, the collating sequence associated with alphabet-name (see section "SPECIAL-NAMES paragraph") is used to determine the truth value of any alphanumeric comparisons:
   a. explicitly specified in relation conditions
   b. explicitly specified in condition-name condition
   c. implicitly specified by the presence of a CONTROL clause in a report description entry (see the section "CONTROL clause").
4. If the PROGRAM COLLATING SEQUENCE clause is not specified, the native collating sequence is used.
5. The PROGRAM COLLATING SEQUENCE clause is also applied to any alphanumeric merge or sort keys unless the COLLATING SEQUENCE phrase of the respective MERGE or SORT statement is specified (see section "MERGE statement" and section "SORT statement").

For examples on the use of the PROGRAM COLLATING SEQUENCE and ALPHABET clauses (see section "SPECIAL-NAMES paragraph").

6.2.3 SPECIAL-NAMES paragraph

Function

The SPECIAL-NAMES paragraph provides a means of

1. relating system-names to mnemonic-names specified by the user
2. relating alphabet-names to character sets and/or collating sequences
3. defining symbolic names for characters
4. relating class-names to character sets
5. supplying a character which is to be used as the currency sign in picture-strings
6. exchanging the functions of the comma and the decimal point in picture-strings and in numeric literals.

Format

```
SPECIAL-NAMES.

[  [implementor-name  { IS mnemonic-name-1 [ON STATUS IS condition-name-1
                                               [OFF STATUS IS condition-name-2]]

                      | IS mnemonic-name-2 [OFF STATUS IS condition-name-2
                                               [ON STATUS IS condition-name-1]]

                      | ON STATUS IS condition-name-1
                                           [OFF STATUS IS condition-name-2]

                      | OFF STATUS IS condition-name-2
                                           [ON STATUS IS condition-name-1]
                      }

   ]...

   [[ARGUMENT-NUMBER IS mnemonic-name-3]
    [ARGUMENT-VALUE IS mnemonic-name-4]
    [ENVIRONMENT-NAME IS mnemonic-name-5]
    [ENVIRONMENT-VALUE IS mnemonic-name-6]
   ]

   [ALPHABET alphabet-name-1 IS     { STANDARD-1
                                    | STANDARD-2
                                    | NATIVE
                                    | EBCDIC
                                    | {literal-1 [ {THROUGH | THRU} literal-2
                                                   {ALSO literal-3}... ]}...
                                    }

   ]...

   [SYMBOLIC CHARACTERS { {symbolic-character-1}... {IS | ARE} {integer-1}...
   }...
   [IN alphabet-name-2]
   ]...
```

```
    [CLASS class-name IS { literal-4 [{THROUGH | THRU} literal-5] }... ]...
    [CURRENCY SIGN IS literal-6]
    [DECIMAL-POINT IS COMMA].

]
```

Syntax rules

1. In a class definition, only the CURRENCY SIGN clause may be specified in the SPECIAL-NAMES paragraph.
2. In an interface definition, only the ALPHABET, CURRENCY SIGN and DECIMAL-POINT clauses may be used in the SPECIAL-NAMES paragraph.
3. The words THROUGH and THRU are equivalent.
4. The following rules apply to literal-1 to literal-5:
   a. If the literals are numeric, they must be unsigned integers with a value from 1 to 256.
   b. If the literals are alphanumeric and associated with the THROUGH, THRU or ALSO phrase, each literal must be one character long.
   c. The literals must not specify a symbolic-character figurative constant.

General rule

1. The individual clauses of the SPECIAL-NAMES paragraph must, if they are used, be specified in the order given in the format.

The individual clauses of the SPECIAL-NAMES paragraph are described below.

6.2.3.1 Implementor-name

Syntax rules

1. implementor-name is a system-name and must be a name from the left column of the following table. Implementor-names and their meanings:

Implementor-name	Meaning
CONSOLE	System or main console or subconsole
TERMINAL	The user' s data display unit
SYSIPT	System logical input file
PRINTER PRINTER01-PRINTER99	System logical printer file
SYSOPT	System logical output file
C01 to C08	Skip to channel 1 through 8
C10 to C11	Skip to channel 10 or 11
JV-job-variable-name	Job variable describing the link name of a job variable (see below)
TSW-0 to TSW-31	Task switches
USW-0 to USW-31	User switches
COMPILER-INFO	Compiler information
CPU-TIME, PROCESS-INFO, TERMINAL-INFO DATE-ISO4	Operating system information

Table 8: Implementor-names and their meanings

2. job-variable-name indicates a BS2000 job variable. It is a COBOL word of up to 7 characters and is used to form the link name *job-variable-name and for accessing the job variable (see Example 6-1)

General rules

1. If implementor-name is a user or task switch, at least one condition-name must be associated with it. The status of the switches is described under "Condition-names", and can be interrogated by testing the condition-name (see the section "Switch-status condition").

   The status of a switch may be altered by using a format 3 SET statement (see section "SET statement").

2. C01 through C08, C10 and C11 will not be supported in the next version of the COBOL2000 compiler.
   If C01 through C08, C10 or C11 is specified as implementor-name, the associated mnemonic-name may be used only in a WRITE statement with ADVANCING phrase.

Example 6-1

Use of job variables:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. JVTEST.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
    JV-JV1 IS JOB-VAR-1.
    ...
PROCEDURE DIVISION.
    ...
    DISPLAY "xyz" UPON JOB-VAR-1.
```

Prior to the program call:

```
/SET-JV-LINK LINK-NAME=*JV1,JV-NAME=JV1TEST
```

6.2.3.2 ARGUMENT-NUMBER / ARGUMENT-VALUE / ENVIRONMENT-NAME / ENVIRONMENT-VALUE

are extensions from the X/OPEN Portability Guide. They are used only in conjunction with ACCEPT and DISPLAY statements and are described under those statements.

6.2.3.3 ALPHABET clause

Syntax rules

1. If the literal phrase is specified in the ALPHABET clause, any given character for literal-1, literal-2 etc. which is referenced by alphabet-name in the PROGRAM COLLATING SEQUENCE clause (see section "OBJECT-COMPUTER paragraph") or in the COLLATING SEQUENCE phrase of the SORT or MERGE statement (see chapter "Procedure Division") may be used once only (see Example 6-10 and Example 6-11).
2. The words THROUGH and THRU are equivalent.
3. The NATIVE and EBCDIC phrases mean the same thing in BS2000.

General rules

1. The ALPHABET clause provides a means for relating a name to a particular character set and/or alphanumeric collating sequence. When referenced in the PROGRAM COLLATING SEQUENCE clause (see section "OBJECT-COMPUTER paragraph") or the COLLATING SEQUENCE phrase of a SORT or MERGE statement (see chapter "Procedure Division") alphabet-name specifies a collating sequence. When alphabet-name-1 is referenced in the SYMBOLIC-CHARACTERS clause or in a CODE-SET clause of a file description entry (for sequentially organized files), the ALPHABET clause specifies a character set.
   a. If the STANDARD-1 phrase is specified, the character set or collating sequence is that defined in the American National Standard Code for Information Interchange (ASCII), X3.4-1968.
   b. If the STANDARD-2 phrase is specified, the character set identified is the International Reference Version of the ISO 7-bit code, as defined in International Standard 646, "7-Bit Coded Character Set for Information Processing Interchange". Each character of the standard character set is associated with a corresponding character from the native character set.
   c. If the NATIVE or EBCDIC phrase is specified, the native character set or native collating sequence is used (EBCDIC).
   d. If the literal phrase of the ALPHABET clause is specified, the alphabet-name must not be referenced in a CODE-SET clause (see the section "CODE-SET clause").
      - The value of the literal specifies the ordinal number of a character (beginning with 1) within the native character set, if the literal is numeric. This value must not exceed the number of characters in the native character set (256).
      - The value of the literal specifies the actual character within the native character set, if the literal is alphanumeric. If the value of the alphanumeric literal contains multiple characters, each character in the literal is inserted into the collating sequence in the order specified (see Example 6-2).
      - The order in which the literals appear in the ALPHABET clause specifies, in ascending sequence, the ordinal number of the character within the collating sequence being specified (see Example 6-3).
      - All characters within the EBCDIC collating sequence which are not specified explicitly with the literal specification have a higher position in the collating sequence than each of the explicitly defined characters. The relative order within the set of these unspecified characters is unchanged from the EBCDIC collating sequence.
      - If the THROUGH/THRU phrase is specified, the set of contiguous characters in the EBCDIC character set beginning with the character specified by the value of literal-1, and ending with the character specified by the value of literal-2, is assigned a successive ascending position in the collating sequence being specified. In addition, the set of contiguous characters specified by a given THROUGH/THRU phrase may contain characters of the EBCDIC character set in either ascending or descending sequence (see Example 6-4).

- If the ALSO phrase is specified, the characters of the EBCDIC character set specified by the value of literal-1 and literal-3 are assigned to the same ordinal position in the collating sequence being specified or in the character set (see Example 6-5).
  If alphabet-name-1 is referenced in a SYMBOLIC CHARACTERS clause, only literal-1 is used to represent the character in the EBCDIC character set.

2. The character that has the highest ordinal position in the program collating sequence specified is associated with the figurative constant HIGH-VALUE. If more than one character has the highest position in the program collating sequence, the last character specified is associated with the figurative constant HIGH-VALUE (see Example 6-6 and Example 6-7).
3. The character that has the lowest ordinal position in the program collating sequence specified is associated with the figurative constant LOW-VALUE. If more than one character has the lowest position in the program collating sequence, the first character specified is associated with the figurative constant LOW-VALUE (see Example 6-8 and Example 6-9).

Example 6-2

```
ALPHABET ALPHATAB IS "AJKCDF".
```

First character is A
Second character is J
.
.
Sixth character is F

Example 6-3

```
ALPHABET ALPHATAB IS "A" "C" "D" "Z".
```

First character in the collating sequence is "A"
Second character in the collating sequence is "C"
Third character in the collating sequence is "D"
Fourth character in the collating sequence is "Z"

Example 6-4

```
ALPHABET ALPHATAB IS "A" THRU "I".
```

First character is A
Second character is B
Third character is C
.
.
Eighth character is H
Ninth character is I

Example 6-5

```
ALPHABET ALPHATAB IS "A" ALSO "B" ALSO "C" ALSO "D".
```

The characters A, B, C, and D will be associated with the lowest ordinal positions in the collating sequence.

Example 6-6

```
ALPHABET ALPHATAB IS 193 THRU 1, 255 THRU 194.
```

The highest ordinal position in the collating sequence is occupied by the character that appears in the 194th position of the EBCDIC character set, i.e. the character A.A is associated with the figurative constant HIGH-VALUE.

Example 6-7

```
ALPHABET ALPHATAB IS 193 THRU 1, 255 THRU 197, "A" ALSO "B" ALSO "C".
```

Positions 1 through 193 of the collating sequence are associated with the characters which appear at positions 193 to 1 of the EBCDIC character set.

Positions 194 through 253 of the collating sequence are associated with the characters which appear at positions 255 to 197 of the EBCDIC character set.

Position 254 is assigned the characters A, B, C; with this all characters in the EBCDIC character set are associated with a position in the collating sequence. The highestorder position (254) is occupied by the characters A, B, C. Being the character specified last, C is associated with the figurative constant HIGH-VALUE.

Example 6-8

```
ALPHABET ALPHATAB IS "0" "1" "2".
```

The lowest ordinal character in the collating sequence is 0. Hence 0 is associated with the figurative constant LOW-VALUE.

Example 6-9

```
ALPHABET ALPHATAB IS "A" ALSO "B" ALSO "C".
```

The lowest ordinal position in the collating sequence is occupied by the characters A, B, C. The character A, which was specified first, is associated with the figurative constant LOW-VALUE.

Example 6-10

PROGRAM COLLATING SEQUENCE and ALPHABET clauses:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. ABC.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
OBJECT-COMPUTER.
    PROGRAM COLLATING SEQUENCE IS ALPHATAB.
SPECIAL-NAMES.
    TERMINAL IS T
    ALPHABET ALPHATAB IS "X" "Y" "Z".
DATA DIVISION.
WORKING-STORAGE SECTION.
77  ITEM-1 PIC X(3) VALUE "ABC".
```

```
77  ITEM-2 PIC X(3) VALUE "XYZ".
PROCEDURE DIVISION.
MAIN.
    IF ITEM-1 > ITEM-2
    THEN
      DISPLAY "Collating sequence ok" UPON T
    END-IF
    STOP RUN.
```

With the definition of the alphabet-name ALPHATAB in the SPECIAL-NAMES paragraph, the character X was assigned to the first position in the collating sequence, Y to the second and Z to the third.

All remaining characters of the EBCDIC character set are assigned a position in the collating sequence implicitly, since their positions in the collating sequence are higher than those of the specified characters X, Y, Z and their order in the collating sequence was taken from the EBCDIC character set without alteration.

Positions 1 through 231 in the EBCDIC character set correspond to positions 4 through 234 in the collating sequence.

Positions 235 through 256 in the EBCDIC character set correspond to positions 235 through 256 in the collating sequence.

Thus, A occupies position 197, B position 198, and C position 199.

Hence, the relation ITEM-1 > ITEM-2 is true.

Example 6-11

PROGRAM COLLATING SEQUENCE and ALPHABET clauses:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. ALPH.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
OBJECT-COMPUTER.
    PROGRAM COLLATING SEQUENCE IS ALPHA.
SPECIAL-NAMES.
    TERMINAL IS T
    ALPHABET ALPHA 1 THRU 247, 251 THRU 256
                   "7" ALSO "8" ALSO "9".
DATA DIVISION.

WORKING-STORAGE SECTION.
77  ITEM-1 PIC X(3) VALUE HIGH-VALUE.
77  ITEM-2 PIC X(3) VALUE "789".
PROCEDURE DIVISION.
P1 SECTION.
COMPARISON.
    IF ITEM-1 = ITEM-2
    THEN
      DISPLAY "First relation ok" UPON T
    ELSE
      DISPLAY "First relation not ok" UPON T
    END-IF
    IF ITEM-2 = HIGH-VALUE
```

```
    THEN
      DISPLAY "Second relation ok" UPON T
    ELSE
      DISPLAY "Second relation not ok" UPON T
    END-IF.
FINISH-PAR.
    STOP RUN.
```

Characters less than 7 remain as in native collating sequence. Characters greater than 9 are then appended, thereby becoming less than 7.
The characters 7, 8, 9 are set at the highest ordinal position, with 9, being the last character specified, corresponding to "HIGH-VALUE".

Result:

First relation OK
Second relation OK

6.2.3.4 SYMBOLIC CHARACTERS clause

Syntax rules

1. No symbolic name for a character may be used more than once in the SYMBOLIC CHARACTERS clause.
2. The relationship between each separate symbolic name and its corresponding integer results from the sequence within the SYMBOLIC CHARACTERS clause: symbolic-character-1 is paired with integer-1, symbolic-character-2 with integer-2, and so on.
3. An integer must be specified for each symbolic name which is specified.
4. The position specified by integer-1 must exist in the alphanumeric native character set. If IN is specified, the position must exist in the character set named by alphabet-name-2.

General rules

1. symbolic-character is a figurative constant.
2. If the IN phrase is omitted, symbolic-character-1 represents the character whose position is identified by integer-1 in the collating sequence of the native character set.
3. If the IN specification exists, character position integer-1 selects a character from the character set specified by alphabet-name-2. The internal representation of symboliccharacter-1 is the same as that of the corresponding character in the native character set.

Example 6-12

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SYMCHAR.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
    TERMINAL IS T
    SYMBOLIC CHARACTERS HEX-0A IS 11.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 PRINT-RECD.
   02 CNTRLBYTE        PIC X.
   02 PRINT-LINE       PIC X(132).
PROCEDURE DIVISION.
MAIN SECTION.
P1.
    MOVE HEX-0A TO CNTRLBYTE.
    DISPLAY CNTRLBYTE UPON T.
STOP RUN.
```

The symbolic name HEX-0A is assigned to the eleventh character of the EBCDIC character set (this character corresponds to the hexadecimal value 0A).
The MOVE statement uses this symbolic name in order to move the hexadecimal value 0A into the control byte.

6.2.3.5 CLASS clause

Syntax rules

1. The CLASS clause enables the user to associate a name with the character set defined in this clause. This class-name may be referenced in a class condition only. Characters specified by the values of literal-4, literal-5, ... form the exclusive character set named by class-name.

   The value of each literal specifies:

   a. The ordinal number of a character in the alphanumeric native character set, if the literal is numeric.

   b. The actual character in the native character set, if the literal is alphanumeric. If the value of this literal contains more than one character, each of these characters is included in the character set named by class-name.

2. If THROUGH is specified, the contiguous characters in the native character set beginning with literal-4 and ending with literal-5 are included in the special character set identified by class-name. The THROUGH phrase may be used to specify this character string in either ascending or descending order.

Example 6-13

```
SPECIAL-NAMES.
    CLASS HEXADECIMAL-CHARACTERS
    194 THRU 199, 241 THRU 250.
```

194 through 199 corresponds to the letters A through F
241 through 250 corresponds to the digits 0 through 9

6.2.3.6 CURRENCY SIGN clause

Syntax rules

1. literal-6 must be an alphanumeric literal. However, it may not be a figurative constant. It must consist of one character which is not one of the following:
   - digits 0 through 9
   - uppercase letters A, B, C, D, E, N, P, R, S, V, X, Z, or the space
   - lowercase letters a - z
   - special characters

     * (asterisk)

     + (plus)

     - (minus)

     , (comma)

     . (period)

     : (colon)

     ; (semicolon)

     ((left parenthesis)

) (right parenthesis)

     " (quotationmark)

     ’ (apostrophe)

     / (slash)

     = (equal sign)
2. If the CURRENCY SIGN clause is not used, only the currency sign $ may be specified as the currency symbol in a PICTURE string.
3. The Euro sign € is also allowed in the CURRENCY SIGN clause.

6.2.3.7 DECIMAL-POINT IS COMMA clause

Syntax rule

1. The DECIMAL-POINT IS COMMA clause means that the functions of comma and period are exchanged in the character-string of the PICTURE clause and in numeric literals.

General rule

1. The DECIMAL-POINT IS COMMA clause is used to exchange the functions of the decimal point and the comma in numeric literals and in picture-strings. When this clause is used, the decimal point required in a numeric literal or in a picture-string must be represented by a comma. The decimal point, in turn, must be used for the function normally performed by the comma.

6.2.4 REPOSITORY paragraph

The REPOSITORY paragraph defines the class and interface names that may be used within the scope of the corresponding Environment Division.

Format

```
REPOSITORY.
[{  CLASS class-name-1

       [EXPANDS class-name-2 USING {class-name-3 | interfacename-3}... ]

  | INTERFACE interfacename-1

       [EXPANDS interfacename-2 USING {class-name-4 | interfacename-4}... ]

 }... .
]
```

Syntax rules

1. If class-name-1 is identical to the name of the class definition in which the REPOSITORY paragraph is specified, this entry is ignored.
2. If interfacename-1 is identical to the name of the interface definition in which the REPOSITORY paragraph is specified, this entry is ignored.
3. If program-prototype-name-1 is identical to the name of the program definition in which the REPOSITORY paragraph is specified, this entry is ignored.
4. In COBOL Standard 2002 the EXPANDS phrase is forbidden within a class or interface definition which uses the USING clause in its ID paragraph. However, this compiler permits another nesting level for the EXPANDS phrase. To avoid name conflicts and to ensure clarity and portability of such programs, this nesting should only be used when there is proper justification.
5. class-name-2, class-name-3 and interfacename-3 must be defined in the same REPOSITORY paragraph as class-name-1.
6. class-name-4, interfacename-4 and interfacename-2 must be defined in the same REPOSITORY paragraph as interfacename-1.
7. class-name-2 and interfacename-2 may only be referenced in an EXPANDS phrase.
8. The expansion of a parameterized class may not be used either directly or indirectly as a current parameter of the same parameterized class.
9. The expansion of a parameterized interface may not be used either directly or indirectly as a current parameter of the same parameterized interface.

General rules

1. Within a class/interface definition no class/interface names may be specified in the REPOSITORY paragraph which either directly or indirectly reference the currently defined class in their REPOSITORY paragraph.
2. class-name-1 is the class name which applies for the entire source unit containing the relevant Environment Division.

3. interfacename-1 is the interface name which applies for the entire source unit containing the relevant Environment Division.
4. program-prototype-name-1 is the program prototype name which applies for the entire source unit containing the relevant Environment Division.
5. If there is no EXPANDS phrase, then data must be contained in the external repository which defines class-name-1/interfacename-1.
6. If a program prototype exists for program-prototype-name-1 in the compilation group, the use of this prototype definition depends on the control of repository access (see the "COBOL2000 User Guide" [1]).
7. class-name-3 and interfacename-3 are the current parameters of the parameterized class class-name-2.
8. class-name-4 and interfacename-4 are the current parameters of the parameterized interface interfacename-2.
9. The number of current parameters in the USING clause must match the number of formal parameters in the USING clause of the CLASS-ID of class-name-2. The specification (as class or interface) of the corresponding current and formal parameters must also match.
   class-name-1 is generated from the parameterized class class-name-2 by replacing all formal parameters with the corresponding current parameters and by replacing class-name-2 with class-name-1.
10. The number of current parameters in the USING clause must match the number of formal parameters in the USING clause of the INTERFACE-ID of interfacename-1. The specification (as class or interface) of the corresponding current and formal parameters must also match.
    interfacename-1 is generated from the parameterized interface interfacename-2 by replacing all formal parameters with the corresponding current parameters and by replacing interfacename-2 with interfacename-1.

6.3 INPUT-OUTPUT SECTION

Function

The INPUT-OUTPUT SECTION deals with the following:

- the definition of each file used in the program,
- the assignment of these files to external devices, and
- the transmission and handling of data between external devices and the object program.

The INPUT-OUTPUT SECTION is divided into two paragraphs:

- the FILE-CONTROL paragraph, which names the files used in the program and assigns them to external devices, and
- the I-O-CONTROL paragraph, which defines special input/output techniques.

Format

```
 INPUT-OUTPUT SECTION.

 FILE-CONTROL. {file-control-entry}...
[I-O-CONTROL. [input-output-control-entry]]
```

Syntax rules

1. The entire INPUT-OUTPUT SECTION is optional. If the INPUT-OUTPUT SECTION is defined, the FILE-CONTROL paragraph must also be specified.
2. The INPUT-OUTPUT SECTION may not be within an interface definition.

6.3.1 FILE-CONTROL paragraph

Function

The FILE-CONTROL paragraph is used to give each file a name. The files are assigned to one or more external devices, and the information required for file processing is made available. This information indicates how the data is organized and how it is to be accessed.

Format 1 File control entries for sequential file organization

```
FILE-CONTROL.

SELECT clause
ASSIGN clause
[ORGANIZATION clause]
[PADDING CHARACTER clause]
[RECORD DELIMITER clause]
[ACCESS MODE clause]
[RESERVE clause]
[FILE STATUS clause].
```

Format 2 File control entries for relative file organization

```
FILE-CONTROL.

SELECT clause
ASSIGN clause
[ACCESS MODE clause]
[FILE STATUS clause]
ORGANIZATION clause
[RESERVE clause].
```

Format 3 File control entries for indexed file organization

```
FILE-CONTROL.

SELECT clause
ASSIGN clause
[ACCESS MODE clause]
[ALTERNATE RECORD KEY clause]
[FILE STATUS clause]
ORGANIZATION clause
RECORD KEY clause
[RESERVE clause].
```

Syntax rule

1. The SELECT clause must be the first file control entry. All other clauses may appear in **any order**.
2. If the GLOBAL clause is specified in the file description entry, the data items referenced in the clauses of the FILE-CONTROL paragraph may not be defined in a LOCAL-STORAGE SECTION.

3. Only the SELECT and ASSIGN clauses may be specified for sort files.

In the pages that follow, the SELECT and ASSIGN clauses are described first, followed by the remaining clauses in alphabetical order.

6.3.1.1 SELECT clause

Function

The SELECT clause is used to give a name to each file in the program.

Format 1

```
SELECT [ OPTIONAL ] file-name
```

Format 2

```
SELECT file-name
```

Syntax rule for format 1 and format 2

1. file-name stands for the name by which a file is referenced in the compilation unit (internal file-name). Each file-name used in a program, a factory, an object or a method may only occur in one SELECT clause.

Syntax rules for format 1

2. Each file specified in a SELECT clause must have precisely one file description (FD) entry in the DATA DIVISION of the same program, the same method, the same factory or the same object.
3. The OPTIONAL phrase is required for files that are not necessarily present at object time. When a file is not present at object time, the first READ statement for that file passes control to the associated at end condition. The OPTIONAL phrase is permitted only for files opened in INPUT, I-O or EXTEND mode.

Syntax rules for format 2

4. file-name designates a sort file.
5. For each sort file specified in a SELECT clause, there must be precisely one sort file description entry (SD) in the DATA DIVISION of the same program or the same method.
6. Within a class definition, this format may only be specified in a method definition.

> ¡ A maximum of 30 characters of the file name are used at runtime (e.g. for messages and default settings).

General rules for format 1

1. If no SET-FILE-LINK command is given for a file when the program is executed, the runtime system creates a file named FILE.COB85.linkname at OPEN OUTPUT. The link name is formed from the specifications in the ASSIGN clause (see the section "ASSIGN clause").
2. If the OPTIONAL phrase refers to an *external* file, OPTIONAL must be specified in all programs that describe this external file.

General rules for format 2

3. No further clauses other than the ASSIGN clause are permitted.
4. Each sort-file described in the Data Division must be designated once, and only once, as a file-name in the FILE-CONTROL paragraph.

5. The following file-names must not be used in SELECT clauses within a program that uses SORT:

```
MERGExx(xx = 01,...,99)
SORTIN
SORTINxx(xx = 01,...,99)
SORTOUT
SORTWK
SORTWKx(x = 1,...,9)
SORTWKxx(xx = 01,...,99)
SORTCKPT
```

6.3.1.2 ASSIGN clause

Function

The ASSIGN clause assigns an external device to a file of the COBOL program. One ASSIGN clause is required for each file in the program.

Format 1 for sequential file organization

```
ASSIGN { TO { PRINTER [literal-1] | implementor-name-1 | literal-2 }...
           | USING data-name-1 }
```

Format 2 for relative and indexed file organization

```
ASSIGN {TO literal-1... | USING data-name-1}
```

Format 3 for sort files

```
ASSIGN TO {implementor-name-1 | literal}
```

Syntax rules

The following applies only to **sequentially organized files**:

1. PRINTER specified without literal-1 refers to the logical system file SYSLST.

   PRINTER literal-1 refers to a print file.

   In both cases, the compiler reserves the feed control character, which is not accessible to the user.
2. implementor-name-1 refers to devices which are named and assigned as follows:

Device name	Assigned system file
PRINTER01 - PRINTER99	SYSLST01 - SYSLST99
SYSIPT	SYSIPT
SYSOPT	SYSOPT

3. Files assigned by means of PRINTER or implementor-name-1 must not be external files.

The following applies for **sequential, relative and indexed file organization**:

4. literal-1, literal-2 must be alphanumeric literals and may not be figurative constants. dataname-1 must be alphanumeric.
5. literal-1, literal-2 or the content of data-name-1 specify the line name for the file.
6. When a literal is specified, the link name is formed from the first 8 characters. These must be unique within the program. If the last character of the link name thus formed is a hyphen (-), then it is replaced by a # character.
7. data-name-1 must not be qualified.

8. data-name-1 must be defined as an alphanumeric data item in the WORKING-STORAGE SECTION, LOCAL-STORAGE SECTION or LINKAGE SECTION.

   The following applies to **sort files**:

9. implementor-name is DISC
10. literal is SORTWK.

General rules

1. The type of file organization must be specified in the ORGANIZATION clause (see section "ORGANIZATION clause").
2. Only the first entry in the ASSIGN clause is evaluated; all other entries are ignored (PRINTER literal-1 is regarded as one entry).
3. If the ASSIGN clause refers to an *external* file, the ASSIGN clause must be used in the same form in all programs that describe this external file. The contents of the literal or data-name may, however, be different.

6.3.1.3 ACCESS MODE clause

Function

The ACCESS MODE clause determines the manner in which the records of a file are to be accessed.

Format 1 for sequential file organization

```
ACCESS MODE IS SEQUENTIAL
```

Format 2 for relative file organization

```
ACCESS MODE IS {  SEQUENTIAL [RELATIVE KEY IS data-name-1]
                | {RANDOM | DYNAMIC} RELATIVE KEY IS data-name-1
               }
```

Syntax rules

1. The RELATIVE KEY phrase specifies the key field whose content is used for retrieving or placing a logical record in a file.
2. data-name-1 must not be subscripted or indexed.
3. The data item referenced by data-name-1 must be defined as an unsigned integer. The symbol "P" may not be specified in its PICTURE character string.
4. data-name-1 must not be specified within the record description entry of the associated file.

Format 3 for indexed file organization

```
ACCESS MODE IS {SEQUENTIAL | RANDOM | DYNAMIC}
```

General rules

For **sequential, relative and indexed file organization:**

1. If the ACCESS MODE clause is not specified, ACCESS MODE IS SEQUENTIAL is assumed.
2. SEQUENTIAL means that records are read or written sequentially, i.e. the next logical record of the file is made available when a READ statement is executed, or the next logical record is placed on that file when a WRITE statement is executed.
3. If the ACCESS MODE clause refers to an *external* file, an equivalent ACCESS MODE clause must be specified in all other programs that describe this external file.
   Furthermore, in the case of **files with relative organization**, data-name-1 must be an external data item identical in all programs.

   The following also applies to **relative and indexed file organization**:
4. RANDOM means that records are randomly read or written on the basis of a key, i.e. access is achieved via the RELATIVE KEY.
5. DYNAMIC means that random and sequential access may be combined.

   The following applies only to **relative file organization**:

6. The data item referenced by data-name-1 is used for communicating a relative record number between the user and the input/output system.

   All records which exist on a file are uniquely identified by their relative record numbers. The relative record number of a given record specifies the logical position of that record within the file record sequence. The first logical record is relative record number one (1), and the following logical records have the relative record numbers 2, 3, 4... .

7. The RELATIVE KEY must not be zero.

6.3.1.4 ALTERNATE RECORD KEY clause

Function

The ALTERNATE RECORD KEY clause defines a further, alternative key, in addition to the primary key, which can be used to access the records of an indexed file.

Format

```
ALTERNATE RECORD KEY IS data-name [WITH DUPLICATES]
```

Syntax rules

1. data-name must specify an alphanumeric or national data item in a record description entry, and this record description entry must be associated with the file name to which the ALTERNATE RECORD KEY clause is subordinate.
2. data-name may be any fixed-length field, with a length of 1 to 127 bytes, within the record. data-name may be qualified, but not subscripted or indexed.
3. data-name may not reference a group item which contains an element with an OCCURS clause with DEPENDING ON phrase.
4. If the file contains variable-length records, the alternate key must be contained within the first x character positions of the record, where x equals the minimum record size specified for the file (see section "RECORD clause"), i.e. the length of the alternate key must fit into the shortest possible record.
5. Several (up to 30) alternate record keys may be specified for one file.
6. data-name must not refer to a data item whose start address (the leftmost character position) is identical with the start address of the primary record key or of any other alternate record key. Apart from this restriction, the primary record key and any alternate record key(s) may overlap.

General rules

1. The data description of data-name and the relative position of the data item defined as a key must be the same as those specified when the file was created. The number of alternate record keys must match the number specified when the file was created.
2. WITH DUPLICATES specifies that the value of the related alternate record key may occur in more than one record. Records with identical key values are also called "duplicates". If WITH DUPLICATES is not specified, the value of the related alternate key must be unique, i.e. it may not occur more than once in the file.
3. If there are several record description entries in a file, data-name needs to be described in only one of these entries. The position of the key field in the record defined by data-name is used implicitly for all keys in the other record description entries.
4. If the ALTERNATE RECORD KEY clause(s) refers/refer to an *external* file, the same number of ALTERNATE RECORD KEY clauses must be specified in any program that uses this external file; the length and position of the key fields in the record and, where applicable, the DUPLICATES phrase, must also be defined so as to be consistent.

6.3.1.5 FILE STATUS clause

Function

The FILE STATUS clause specifies a data item that indicates the status of input/output operations during processing. In addition, by specifying a further item, an additional error code is made available.

Format

```
FILE STATUS IS data-name-1 [, data-name-2]
```

Syntax rules

1. data-name-1 and data-name-2 must be defined in the WORKING-STORAGE SECTION, LOCAL-STORAGE SECTION or LINKAGE SECTION of the Data Division.
2. data-name-1 must be a two-byte alphanumeric data item.
3. data-name-2 must be a 6-character group item with the following format:

```
01 data-name-2.
  02 data-name-2-1 PIC 9(2) COMP.
    02 data-name-2-2 PIC X(4).
```

General rules

1. If the FILE STATUS clause is specified, the runtime system copies the I-O status to data-name-1.
2. If specified, data-name-2 is assigned as follows:
   a. If data-name-1 has the value 0, the contents of data-name-2 are undefined.
   b. If data-name-1 has a non-zero value, data-name-2 contains the additional error code. The value 64 in data-name-2 indicates that this code is the (BS2000) DMS code; the value 96 in data-name-2 indicates that the code is the (POSIX) SIS code. The command HELP DMS <contents-of-data-name-2-2> or HELP SIS <contents-of-data-name-2-2> supplies more detailed information on the corresponding error code.
3. The I-O status is copied during the execution of each OPEN, CLOSE, READ, WRITE, REWRITE or START statement that references the specified file, and prior to the execution of each corresponding USE procedure (see chapter "General concepts").

6.3.1.6 ORGANIZATION clause

Function

The ORGANIZATION clause defines the logical structure of a file.

Format 1 for sequential file organization

```
[ORGANIZATION IS] { SEQUENTIAL | LINE SEQUENTIAL }
```

Format 2 for relative file organization

```
[ORGANIZATION IS] RELATIVE
```

Format 3 for indexed file organization

```
[ORGANIZATION IS] INDEXED
```

General rules

1. File organization is defined at the time a file is created and cannot be changed subsequently.
2. If the ORGANIZATION clause is omitted, ORGANIZATION IS SEQUENTIAL is assumed.
3. For an *external* file, the same ORGANIZATION clause must be defined in all programs that describe this external file.

6.3.1.7 PADDING CHARACTER clause

Function

The PADDING CHARACTER clause is used to specify a padding character.

Format

```
PADDING CHARACTER IS {data-name | literal}
```

The PADDING CHARACTER clause is treated as a comment by the compiler.

6.3.1.8 RECORD DELIMITER clause

Function

The RECORD DELIMITER clause indicates the method of determining the length of a variable-length record on the external medium.

Format

```
RECORD DELIMITER IS {STANDARD-1 | BS2000}
```

The RECORD DELIMITER clause is treated as a comment by the compiler.

6.3.1.9 RECORD KEY clause

Function

The RECORD KEY clause defines the primary record key used for accessing the records in an indexed file.

Format

```
RECORD KEY IS data-name
```

Syntax rules

1. data-name must be an alphanumeric or national data item defined within a record description entry. This record description entry must be assigned to the file-name to which the RECORD KEY clause is subordinate.
2. data-name may be any fixed-length item within the record. The item may be 1 through 255 bytes in length. The data-name may be qualified, but not indexed or subscripted.
3. data-name may not reference a group item which contains an element with an OCCURS clause with DEPENDING ON phrase.
4. If the file contains variable-length records, the record key must be contained in the first x bytes in the record, where x is the minimum record length for this file (see section "RECORD clause"). This means that the length of the record key must be fully included within the shortest record.

General rules

1. Record key values must be unique for all records within the file.
2. The data description and the relative position within a record of data-name must be the same as those specified when the file was created. The number of alternate record keys must match the number specified when the file was created.
3. If there are several record description entries in a file, data-name needs to be described in only one of these entries. The position of the key field in the record defined by data-name is used implicitly for all keys in the other record description entries.
4. If the RECORD KEY clause refers to an *external* file, the equivalent RECORD KEY clause must be specified in any program that uses this external file; in particular, the length and position of the key fields in the record must be defined so as to be consistent.

6.3.1.10 RESERVE clause

Function

The RESERVE clause allows the user to modify the number of input/output areas (buffers) to be allocated to the program by the compiler.

Format

```
RESERVE integer [AREA | AREAS]
```

The RESERVE clause is treated as a comment by the compiler.

6.3.2 I-O-CONTROL paragraph

Function

The I-O-CONTROL paragraph defines the events which, if they occur, cause restart points to be established. It may also specify a memory area that is to be shared by different files. In addition, it defines special input/output conditions and, in the case of **sequentially organized files**, also the location of multiple file tapes.

Format 1 for sequential file organization

```
I-O-CONTROL.

    [MULTIPLE FILE TAPE clause]...
    [RERUN clause]...
    [SAME AREA clause]...
```

Format 2 for relative and indexed file organization

```
I-O-CONTROL.

     [RERUN clause]...
     [SAME AREA clause]...
```

Syntax rule

1. Within a class definition, an I-O-CONTROL may only be specified in a method definition.

6.3.2.1 MULTIPLE FILE TAPE clause

Function

The MULTIPLE FILE TAPE clause is required when more than one file shares the same physical reel of tape.

Format

```
MULTIPLE FILE TAPE CONTAINS {file-name-1 [POSITION integer-1]}...
```

Syntax rule

1. integer can have any value between 1 and 3315.

General rules

1. When all file-names are supplied in the same order in which they appear on a reel, the POSITION phrase may be omitted.
2. If any of the files is not specified, then the positions must be supplied as relative to the beginning of the tape.
3. Irrespective of the number of files that share the reel of tape, only those used by the program need be specified.
4. Only one file may be opened on the same reel at any given time.
5. REWIND can be carried out when the last file of the tape has been processed.
6. If the MULTIPLE FILE TAPE clause is specified for an *external* file, then the MULTIPLE FILE TAPE clause must be specified in all programs that describe this external file. If position numbers are specified, these must be the same in all programs.

For further information, see the "COBOL2000 User Guide" [1].

6.3.2.2 RERUN clause

Function

A RERUN clause indicates where and when restart-point records are to be issued. A restart-point record describes the status of a program at a specified point during program execution. It is produced automatically by the operating system upon the request of the program and contains all information necessary to restart the program from that point. The RERUN clause controls such requests by the COBOL object.
For further information on the RERUN clause, see "COBOL2000 User Guide" [1].

The RERUN clause in the I-O CONTROL paragraph enables programs containing a SORT or MERGE statement to be restarted.

Format 1 for sequential file organization

```
RERUN [ON {file-name-1 | implementor-name}]
       EVERY {  { [END OF] {REEL | UNIT} | integer-1 RECORDS } OF file-name-2
              | integer-2 CLOCK-UNITS
              | condition-name
             }
```

Format 2 for relative and indexed file organization

```
RERUN [ON {implementor-name | file-name-1}]
       EVERY {  integer-1 RECORDS OF file-name-2
              | integer-2 CLOCK-UNITS
              | condition-name
             }
```

Syntax rules

1. file-name-1 must be the name of a sequential tape file; it must be described in the FILE SECTION.
2. implementor-name has the format
   SYSnnn
   where 000 <= nnn <= 244
   nnn determines the processing mode.
   If nnn >= 200, restart points are written alternately to two restart files. This enables the restart to be made from the two most recent restart points.
3. If the same implementor-name is also used in the SELECT clause, it must be associated with a tape file.
4. The END OF REEL/UNIT phrase may be used only when file-name-2 describes a sequential file. REEL and UNIT are synonymous.
5. implementor-name must be specified in the RERUN clause when integer-1 RECORDS or integer-2 CLOCK-UNITS is used.
6. Subject to the following restrictions, more than one RERUN clause may be specified for the same file-name-2:
   a. When specifying more than one integer-1 RECORDS phrase, the same file-name-2 must appear only once in them.
   b. When specifying more than one END OF REEL or END OF UNIT phrase, the same file-name-2 must appear only once in them.

7. file-name-1 or implementor-name specifies the file to which the restart points are to be output.
8. Restart points are written as follows:
    a. If file-name-1 is specified, the restart points are output to each reel or volume assigned to file-name-1.
    b. If implementor-name is specified, the restart points are written as follows:When implementor-name is assigned to a file-name from the SELECT clause, the file involved must be a tape file. The effect on restart point generation is the same as if the file-name had been specified directly in the RERUN clause (see 8a).

       When implementor-name is not assigned to a file from a SELECT clause, a disk storage file must be assigned at runtime; otherwise, a file with the name progid.RERUN.implementor-name is assigned by the runtime system. Only in this case will restart points be written to disk storage.
    c. If the same SYS numbers are specified in the ASSIGN clause and in the RERUN END OF REEL clause, the restart point is written to the end of the output tape.
9. There are four types of RERUN clauses, depending on the conditions under which restart points are requested.
    a. END OF REEL or END OF UNIT without specification of the ON clause: the restart points are written to file-name-2, which must specify an output file.
    b. END OF REEL or END OF UNIT with file-name-1 specified in the ON clause: the restart points are written to file-name-1, which must refer to an output file. Additionally, the usual end-of-reel handling is carried out for file-name-2. file-name-2 may be an input or output file.
    c. END OF REEL or END OF UNIT with implementor-name in the ON clause: The restart points are written to a separate file (see 8b). file-name-2 may be an input or an output file.
    d. integer-1 RECORDS: The restart points are written to the file specified by file-name1 or implementor-name, respectively, whenever integer-1 records have been processed. File-name-2 may be either an input or an output file with any organization or type of access; it must not be referenced in the USING/GIVING phrase or in an INPUT/OUTPUT procedure during sort (see section "Sorting records").
10. The CLOCK-UNITS phrase is treated by the compiler as a comment.
11. The condition-name phrase is also treated by the compiler as a comment.
12. If an *external* file is specified by file-name-1, the behavior is undefined.

Format 3 for sorting records

```
RERUN ON {file-name-1 | implementor-name} EVERY SORT [OF file-name-2 ... ]
```

Syntax rules

1. file-name-2... must be defined in an FD entry.
2. file-name-1 must not be defined in an FD entry.
3. implementor-name must not appear in a SELECT clause.

General rules

1. When the OF phrase is not specified, the checkpoint is written prior to each sort operation.
2. When file-name-2 is specified, checkpoints are written for this specific sort operation only.

6.3.2.3 SAME AREA clause

Function

The SAME AREA clause indicates that a number of files are to share a specified input/output area during program execution.

Format 1	applies to all files except sort files, unless RECORD is specified
Format 2	is suitable for sort files

Format 1 for all types of file organization

```
SAME [RECORD] AREA FOR file-name-1 {file-name-2}...
```

Syntax rules

1. More than one SAME AREA clause may be included in a program. In this case, the following must be observed:

   A specific file-name must not appear in more than one SAME AREA clause. The same is true of the SAME RECORD AREA clause.

   A specific file-name may concurrently appear in a SAME RECORD AREA clause. In this case, all file-names appearing in the SAME AREA clause must also appear in the SAME RECORD AREA clause. The SAME RECORD AREA clause may also contain other file-names that do not appear in the SAME AREA clause.
2. The SAME AREA clause indicates that the specified files (no sort-files) are to share the input/output areas assigned to them.
3. The SAME RECORD AREA clause indicates that the specified files are to share the same storage areas for processing the current logical record.

   A logical record in the SAME RECORD AREA clause is considered a logical record of all files which are opened for OUTPUT and whose names are supplied in that SAME RECORD AREA clause. It is also a logical record of the file (in this clause) from which the most recent input occurred. This is equivalent to an implicit overwriting of all record areas, where the records are aligned on the leftmost character position.
4. file-name-1 and file-name-2 may not be files with organization XML.

General rules

1. If SAME AREA is used, only one file may be open at any given time.
2. If the RECORD phrase is used, all specified files may be open at the same time.
3. General rule 1 applies in the case of files that are specified in the SAME AREA clause as well as the SAME RECORD AREA clause.
4. The SAME [RECORD] AREA clause must not be specified for *external* files.

Format 2 for sorting records

```
SAME {RECORD | SORT | SORT-MERGE} AREA FOR file-name-1 {file-name-2}...
```

Syntax rules

1. SORT and SORT-MERGE are equivalent.
2. If SAME SORT AREA or SAME SORT-MERGE AREA is used, at least one of the filenames must refer to a sort or merge file. Files which are not sort or merge files may also be specified in the clause.
3. More than one SAME AREA clause may appear in a given program, but the following restrictions must be observed:
   a. A given file-name must not occur in more than one SAME RECORD AREA clause.
   b. A file-name representing a sort-file must not appear in more than one SAME SORT AREA or SAME SORT-MERGE AREA clause.
   c. If a file-name which does not represent a sort-file appears in a SAME AREA clause and in one (or more) SAME SORT AREA or SAME SORT-MERGE AREA clause(s), then all files specified in the SAME AREA clause must also be supplied in the SAME SORT AREA or SAME SORT-MERGE AREA clause.
4. The files that appear in the SAME SORT AREA, SAME SORT-MERGE AREA or SAME RECORD AREA clause need not all have the same organization or access mode.
5. file-name-1 and file-name-2 may not be files with organization XML.

General rule

1. The SAME RECORD AREA clause indicates that two or more files should share the storage area in which the current logical record is being processed. All of the files can be open at the same time. A logical record in the "same record area" is considered as a logical record of each opened output file whose file-name occurs in the SAME RECORD AREA clause; it is also considered as a logical record of the last read input file whose file-name occurs in the SAME RECORD AREA clause. This is equivalent to an implicit redefinition of the internal storage area, where the records are aligned on the leftmost character position.

7 Data Division

7.1 General description

Two types of data are processed by a COBOL compilation unit:

1. data stored on some external medium
2. data generated internally during program execution.

The first type of data is combined into records in files; the second type of data must be declared by the user as records or subordinate data items.

To ensure maximum independence of data from its specific representation on external volumes and in data processing systems, the properties or contents of data are described with respect to a standard data format rather than a system-oriented format.

The data description method used allows a distinction to be made between the physical and system-dependent properties on the one hand, and conceptual characteristics on the other.

Physical and certain system-specific properties of data on an external medium are defined in a COBOL compilation unit in order that efficient use may be made of special techniques.

1. The term "physical properties of data" refers to
    a. the grouping of logical records within the physical boundaries of the external volume
    b. the recording mode in which the data is stored on the external volume.
2. The term "system-dependent properties of data" refers to the description under which a file is identified on some external medium.

The conceptual characteristics of data on an external volume pertain to the logical units of data, the logical records, and are not associated with any physical or system-specific properties.

These characteristics are described for files in the file description entries and for records in the record description entries.

A data description in a COBOL compilation unit is separate from the declaration of execution procedures. This permits the programmer a great number of options for modifying a data description entry without any change to the procedures which are related to that entry. Therefore, to a certain extent, the procedures of a COBOL compilation unit may be seen as data-independent.

7.1.1 Structure of a Data Division

The Data Division is one of the divisions of a compilation unit and is divided into the following SECTIONs. A SECTION need not be specified if its logical function is not required. If SECTIONs are used, the order in which they are shown in the general format must be maintained.

7.1.2 General format

```
    DATA DIVISION.

[ FILE SECTION.
  [  file-description-entry.{record-description-entry}..
   | sort-file-description-entry.{record-description-entry}...
   | report-file-description-entry.
  ] ...
]

[ WORKING-STORAGE SECTION.
  [  77-level-description-entry.
   | record-description-entry.
  ] ...
]

[ LOCAL-STORAGE SECTION.
  [  77 - level-description-entry.
   | record-description-entry.
  ] ...
]

[ LINKAGE SECTION.
  [  77-level-description-entry.
   | record-description-entry.
  ] ...
]

[ REPORT SECTION.
  [report-description-entry. {report-group-description-entry.}...]...
]

[ SUB-SCHEMA SECTION.
  database-description-entry.
]
```

7.1.2.1 FILE SECTION

All data which is, or is meant to be, stored on external media must first be described in the FILE SECTION before it can be processed by a COBOL program. The FILE SECTION is used to define the structure of files. Each file is defined by a file description entry and one or more record description entries. Record description entries are written immediately following the file description entry.

7.1.2.2 WORKING-STORAGE SECTION

Information intended for internal use must be described in the WORKING-STORAGE SECTION. The WORKING-STORAGE SECTION describes records and structurally noncontiguous data items (refer to "General format") which are not part of external files. Data described in the WORKING-STORAGE SECTION may be in the “last-used” or initial state (see section "Sorting records") on calling the source unit. The WORKING-STORAGE SECTION may be specified in a program, a factory or object definition or a method definition within a class definition.

7.1.2.3 LOCAL-STORAGE SECTION

Data described in the LOCAL-STORAGE SECTION constitutes automatic data and is initialized with values specified in the associated VALUE clause. The LOCAL-STORAGE SECTION may occur in both recursive and non-recursive legal source units. Data in this section is always in the initial state on calling the source unit. The LOCAL-STORAGE SECTION may not contain any EXTERNAL clause and can be specified in the DATA DIVISION of a program or a method.

Note:
Information concerning internal use can be described in the WORKING-STORAGE SECTION or the LOCAL-STORAGE SECTION.

7.1.2.4 LINKAGE SECTION

Information transmitted from one source unit to another must be described in the LINKAGE SECTION. The LINKAGE SECTION is located in a called source unit and defines data items in the calling source unit which may reference the called source unit. The LINKAGE SECTION is primarily meaningful in methods and in programs that are called as subprograms. Furthermore, data is defined in the LINKAGE-SECTION which is assigned storage space only at runtime (see section "BASED clause"). If a data item from the LINKAGE SECTION of a program that was not called from another source unit or to which no storage space has as yet been allocated, the behavior is undefined.
The structure and contents of the WORKING-STORAGE SECTION, LOCAL-STORAGE SECTION and LINKAGE SECTION are basically identical.

7.1.2.5 REPORT SECTION

The contents and appearance of all listings created by the Report Writer must be described in the REPORT SECTION.
(see chapter "Report Writer")

7.1.2.6 SUB-SCHEMA SECTION

All information pertaining to the description of database structures must be entered in the SUB-SCHEMA SECTION. The SUB-SCHEMA SECTION may not be specified in a program for which the RECURSIVE clause has been defined. It is otherwise ignored.

(For further details see the "UDS/SQL Reference Manual" [6]).

7.2 File description

Function

The **file description (FD) entry** specifies the physical structure and the record names for a given file. A **sort-file description (SD) entry** provides information concerning the physical structure, identification, and size of the records on a sort-file.

A file description entry must be written for each file to be processed by the program. The information contained in this entry generally pertains to the physical aspects of the file, that is, the description of the data as it appears on the input or output medium.

7.2.1 Formats of the file description entry

Format 1 for sequential file organization

```
FD file-name

    [BLOCK CONTAINS clause]
    [CODE-SET clause]
    [DATA RECORDS clause]
    [EXTERNAL clause]
    [GLOBAL clause]
    [LABEL RECORDS clause]
    [LINAGE clause]
    [RECORD clause]
    [RECORDING MODE clause]
    [VALUE OF clause].
```

Format 2 for relative file organization

```
FD file-name

    [BLOCK CONTAINS clause]
    [DATA RECORDS clause]
    [EXTERNAL clause]
    [GLOBAL clause]
    [LABEL RECORDS clause]
    [RECORD clause]
    [VALUE OF clause].
```

Format 3 for indexed file organization

```
FD file-name

    [BLOCK CONTAINS clause]
    [DATA RECORDS clause]
    [EXTERNAL clause]
    [GLOBAL clause]
    [LABEL RECORDS clause]
    [RECORD clause]
    [VALUE OF clause].
```

Format 4 for sort files

```
SD sort-file-name
```

```
[DATA RECORDS clause]
[LABEL RECORDS clause]
[RECORD clause]
[RECORDING MODE clause]
```

Syntax rules for format 1, 2 and 3

1. The level indicator FD identifies the beginning of a file description entry.
2. file-name must be identical to the file-name given in a SELECT clause.
3. Clauses that follow the file-name may appear in any order.
4. The file description entry must be followed by one or more record description entries.
5. The associated record description entries must be of the category alphabetic, alphanumeric, national or numeric.
6. If the GLOBAL clause is specified in the file description entry, the data items referenced in the clauses of the file description entry may not be defined in a LOCAL-STORAGE SECTION.

General rules for formats 1, 2 and 3

1. The following Table provides a summary of the functions of clauses used in file description entries.

Clause	Function
BLOCK CONTAINS	Specifies physical block length
CODE-SET	Specifies the character code set used to represent data on the extern media
DATA RECORDS	Specifies the names of the records in the file
EXTERNAL	Declares a file as external
GLOBAL	Declares a file as global
LABEL RECORDS	Gives the names and values of the label records contained in the file
LINAGE	Specifies the size of the logical page. It can also be use to specify a page body and a page footer.
RECORD	Specifies the logical record size.
RECORDING MODE	Specifies the format of the logical records.
VALUE OF	Specifies the values of some data items of a label.

Table 9: Functions of file description clauses

2. The CODE-SET clause applies only to **sequential file organization**.

Syntax rules for format 4

1. The level indicator SD identifies the beginning of the sort-file description entry and must precede the file-name.

2. The clauses following the name of the file are optional and may appear in any order.
3. One or more record description entries for data-name-1,... must follow the sort-file description entry.
4. The associated record description entries must be of the category alphabetic, alphanumeric, national or numeric.
5. If the GLOBAL clause is specified in the file description entry, the data items referenced in the clauses of the sort description entry may not be defined in a LOCAL-STORAGE SECTION.

General rules for format 4

1. sort-file-name indicates the sort-file.
2. data-name-1... in the DATA RECORDS clause refer to records described in the record description entries associated with this sort-file description (SD) entry.
3. The FILE SECTION must contain a sort-file description entry for each sort-file, i.e. for each file that is supplied as the first operand within a SORT or MERGE statement.
4. The RECORDING MODE clause specifies the organization format of data on external devices.
5. The LABEL RECORDS clause is optional; however, if it is not specified LABEL RECORDS ARE OMITTED is assumed. This means that existing labels will be overwritten.
6. When the LABEL RECORDS ARE STANDARD clause is specified, the work tapes must have standard labels for sorting purposes; however, label handling is not performed. In this case, the labels are retained intact.
7. The DATA RECORDS clause specifies the names of the records on the file to be sorted.
8. If more than one data-name is present, the file contains two or more types of records. These records may differ in length, format etc. They may be listed in any order.
9. If more than one record description is specified for the logical record of a file, these records automatically occupy the same internal storage area; this is equivalent to an implicit redefinition of the area.
10. The RECORDING MODE, DATA RECORDS and RECORD CONTAINS clauses are optional, as the compiler can determine the modes, names and sizes of the records from the associated record descriptions.
11. If any of the record descriptions associated with the sort-file description contains an OCCURS clause with the DEPENDING ON phrase, variable-length records are assumed (for further information on assumptions made by the compiler when the RECORDING MODE clause is omitted, see "RECORDING MODE clause").
12. Sort-file-names may only be used in the SORT, MERGE and RETURN statements.

7.2.2 Clauses for data description entries

7.2.2.1 BLOCK CONTAINS clause

Function

The BLOCK CONTAINS clause specifies the maximum size of a physical block.

Format

```
BLOCK CONTAINS [integer-1 TO ] integer-2 {CHARACTERS | RECORDS }
```

Syntax rules

1. A block must contain at least 20 bytes and must not exceed a maximum length of 32763 bytes.
2. The CHARACTERS or RECORDS phrase indicates whether block length is to be specified as a multiple of bytes or logical records.
3. If neither CHARACTERS nor RECORDS is specified, CHARACTERS will be assumed.
4. integer-1 TO integer-2 indicates the number of bytes or records in a given block, depending on the phrase used (either CHARACTERS or RECORDS).
5. When only integer-2 is specified, it refers to the maximum size of the block. When integer-1 and integer-2 are specified, they refer to the minimum and maximum size of the block, respectively. However, integer-1, when specified, is used only for documentation purposes and is treated as a comment by the compiler.

   The **maximum block size** specified by this clause has the following meaning:

   The blocks of the file must not be longer but can be shorter than the specified length. This is frequently the case with unblocked or blocked records whose length is variable.
6. When the CHARACTERS phrase is used, the block size has to be specified in terms of the number of bytes contained within the block. In this case, both integer-1 and integer2 must include slack bytes and 4 bytes for the record length field of each record of the block.
7. When the CHARACTERS phrase is used, and only integer-2 is specified, integer-2 indicates the exact length of the physical block. When both integer-1 and integer-2 are specified, they refer to the minimum or maximum physical block length, respectively.
8. When the RECORDS phrase is used, the block size is specified in terms of logical records. In this case, the compiler calculates the block size by multiplying the number of bytes in the maximum size logical record by the value specified in integer-2. In the case of variable-length records, 4 bytes are added for the record length field.
9. When the BLOCK CONTAINS clause is omitted, the compiler assumes that the records are not blocked, i.e. BLOCK CONTAINS 1 RECORDS is assumed.

   Note:
   The BLOCK CONTAINS clause may therefore be omitted when **all** blocks of the file contain one, and only one, record.

General rules

1. The following table shows how the compiler calculates the block sizes in terms of bytes and includes the appropriate phrases contained in the BLOCK CONTAINS clause.

Legend for table 10

F = fixed-length records

V = variable-length records

BL = block length

RL = record length

RL_{max} = maximum record length

BLF = block length field (has the value 4)

RLF = record length field (has the value 4)

n = integer

Record format	**BLOCK CONTAINS [integer-1 TO] integer-2**	
	CHARACTERS	**RECORDS**
Fixed length	BL = integer-2 (integer-2 = n * RL)	BL = integer-2 * RL
Variable length	BL = integer-2 + BLF	BL = integer-2 * (RL_{max} + RLF) + BLF

Table 10: Calculation of the maximum block size

2. If the BLOCK CONTAINS clause is specified for an *external* file, the BLOCK CONTAINS clause must be specified in all programs that describe this external file; the block length calculated from the information specified in the BLOCK CONTAINS clause must be the same, regardless of whether it results from the number of "RECORDS" or the number of "CHARACTERS".

Note:
The CHARACTERS phrase should be used if the RECORDS phrase would produce a block size that is not accurate enough.
Let us assume, for example, that a block contains 4 records consisting of one 50-character record and three 100-character records. If `BLOCK CONTAINS 4 RECORDS` is specified, and the maximum record length is defined as 100 bytes, the compiler will calculate a block size of 4*(100+4)+4 = 420 bytes (including the block length field). In this case, however, since each block actually requires only (50+4)+3*(100+4)+4 = 366 bytes (without the block length field), the exact block size could be specified by using the following clause instead:
`BLOCK CONTAINS 366 CHARACTERS.`

7.2.2.2 CODE-SET clause

Function

The CODE-SET clause specifies the character code set used to represent data on the external media.

Format

```
CODE-SET IS alphabet-name
```

Syntax rules

1. When the CODE-SET clause is specified for a file, all data in that file must be described as USAGE IS DISPLAY, and any signed numeric data must be described with the SIGN IS SEPARATE clause.
2. The alphabet-name clause referenced by the CODE-SET clause must not specify the literal phrase (see "SPECIAL-NAMES paragraph").

General rules

1. If the CODE-SET clause is specified, alphabet-name specifies the character code convention used to represent data on the external media. It also specifies the algorithm for converting the character codes on the external media from/to the native character codes. This code conversion occurs during the execution of an input or output operation (see "SPECIAL-NAMES paragraph").
2. If the CODE-SET clause is not specified, the native character code set (EBCDIC) is assumed for data on the external media.
3. If the CODE-SET clause refers to an *external* file, an identical CODE-SET clause must be specified in all programs that describe this external file.

7.2.2.3 DATA RECORDS clause

Function

The DATA RECORDS clause is used only for documentation. It specifies the names of the records in a file.

Format

```
DATA {RECORD | RECORDS}  {IS | ARE} {data-name-1}...
```

Syntax rules

1. data-name-1 is the name of a record. It must be preceded by level number 01 in the file description entry.
2. The presence of more than one data-name indicates that the file contains more than one type of record. These records may be of differing sizes, formats etc. The order in which they are listed is not significant.

7.2.2.4 EXTERNAL clause

Function

With the EXTERNAL clause, a file can be defined as external. External files can be accessed by any program in which the file is described.

Format

```
IS EXTERNAL
```

Syntax rule

1. Names of external files can have a maximum length of 30 characters.

General rules

1. If a file is defined as external, the records in that file are also implicitly external.
2. If the file description entry for a sequential file contains the LINAGE clause and the EXTERNAL clause, the LINAGE-COUNTER special register is implicitly an external data item.
3. The following may not be used as names for external files:
   - external record-names from the WORKING-STORAGE SECTION of other compilation units in the run unit,
   - PROGRAM-ID names of the run unit, except for program names of contained programs of a nested program,
   - names used as entry points in the ENTRY statement,
   - names that identify interfaces (LZS-name, etc.).
4. The effect of the FILE STATUS clause for external files is always local to the program, i.e. the file status is supplied only by I-O operations in the program that contains a corresponding specification in the file description entry.
5. The EXTERNAL clause may not be specified in file or record description entries for files that use a common I-O area (SAME RECORD AREA clause).
6. The EXTERNAL clause may not be specified for files that are assigned to the system devices SYSIPT, SYSOPT, PRINTER or PRINTERnn.
7. The EXTERNAL clause may not be specified for files for which user labels and corresponding USE procedures are defined.
8. An external file must be essentially described in the same manner via explicit clauses or implicit default values in all programs that wish to access the file. The following table shows how and to what extent the descriptions must match:

Clauses / specifications	**In all programs**
Name of external file	same to full length (30 characters)
OPTIONAL phrase (SELECT clause)	same specification*)
ASSIGN TO data-name	same form of assignment
ASSIGN TO PRINTER literal	same form of assignment
ORGANIZATION clause	same form of organization

ACCESS MODE clause	same access method
RELATIVE KEY phrase	same number of digits
RECORD KEY clause	same length and position
ALTERNATE RECORD KEY clause	same number, position, length and DUPLICATES phrase
BLOCK CONTAINS clause	same block size in bytes
MULTIPLE FILE TAPE clause	same position number
RECORD clause	same minimum and maximum record length
LABEL RECORDS clause	same specification*)
REPORT clause (Report Writer)	same specification*)
LINAGE clause	same specification*)
CODE SET clause	same specification*)
RECORDING MODE clause	same specification*)

*) Same specification means that the relevant clause may either be specified in none of the programs or must be specified the same in all programs.

All programs that access the same external file must have been compiled with the same value of the compiler option ENABLE-UFS-ACCESS or with the same module format (see the "COBOL2000 User Guide" [1]).

> ℹ If a file is defined as external, this does not mean that the associated file-name is implicitly a global name.

Additional rules, depending on the module format

The following applies to the names of external files when generating the *OMF format (see the "COBOL2000 User Guide" [1]):

1. The eighth character must not be a hyphen.
2. Only the first 7 characters of the name are used for identification. These characters should therefore be unique for each external name in the run unit.

7.2.2.5 GLOBAL clause

Function

The GLOBAL clause can only be used within a nested program. It defines a file-name (data-name or report-name; see section "Data description entry" and chapter "Report Writer" for details) as global. A global name can be accessed by the program defining it and by any other program contained directly or indirectly in that program.

Format

```
IS GLOBAL
```

Syntax rules

1. The GLOBAL clause may only be specified in file description entries.
2. The GLOBAL clause must not be specified in file or record description entries for files that use a common I-O area (SAME RECORD AREA clause).
3. The GLOBAL clause may only be specified in a method definition.

General rules

1. A data-name, file-name or report-name whose description contains a GLOBAL clause is a global name. All data-names subordinate to a global name and all condition-names associated with a global name are global names.
2. Any program contained in the program that describes the global name may access a global name. The global name does not need to be described again in the program that references it. In the case of references to identical names, the local names have priority (see “Determining the valid name” in "Local and global names").
3. If both the LINAGE and GLOBAL clauses are specified in a description of a sequential file, then the LINAGE-COUNTER special register is also global.

7.2.2.6 LABEL RECORDS clause

Function

The LABEL RECORDS clause specifies whether labels are present, and identifies them if they are.

Format 1 for sequential file organization

```
LABEL {RECORD | RECORDS} {IS | ARE} {OMITTED | STANDARD | {data-name-1}...}
```

Format 2 for relative ad indexed file organization

```
LABEL {RECORD | RECORDS} {IS | ARE} STANDARD
```

Syntax rules

1. For data-name-1, record description entries must be present for the file concerned. data-name-1... must not appear as operands in the DATA RECORDS clause of this file.
2. The OMITTED phrase specifies that there are either no unique labels for this file, or the existing labels are non-standard, and the user does not wish to use a USE procedure for label processing (e.g. he may want to process the labels as records).
3. The STANDARD phrase specifies that labels are present for the file and that these labels are in accordance with system conventions (see the "Introductory Guide to DMS" [9]).
4. When data-name-1 is specified, this indicates either the presence of user labels in addition to standard labels, or the presence of nonstandard labels. data-name-1 defines the name of a user label record.

   When processing user labels, data-name-1 may be defined for any files except unit-record files.

General rules

1. OMITTED may not be specified for files which are assigned with "ASSIGN TO literal".
2. For the format of system labels, see the "Introductory Guide to DMS" [9].
3. User labels are formatted as follows:
   a. Each user label is 80 bytes long.
   b. Positions 1-3 of a user header label must contain the characters UHL.
   c. Positions 1-3 of a user trailer label must contain the characters UTL.
   d. Position 4 shows the relative location of the label in a sequence of header or trailer labels; in other words, this position must contain a digit from 1 to 9. If only one label (UHL and/or UTL) is present, position 4 must contain the character "1".
   e. Positions 5-80 are formatted according to user specifications.

   For further details, see the "Introductory Guide to DMS" [9].
4. User header labels follow standard file header labels of the system, however, they precede the first record.
5. User trailer labels follow standard end-of-file labels of the system.
6. Nonstandard labels can be from 1 to 4095 bytes long. Their format and contents are defined by the user.
7. If data-name-1 is specified, then all Procedure Division references to the specified data-names or to items subordinate to these data-names must appear within user declaratives (USE procedures).

8. If the LABEL RECORDS clause refers to an *external* file, an equivalent LABEL RECORDS clause must be specified in all programs that describe this external file.

7.2.2.7 LINAGE clause

Function

The LINAGE clause provides a means of specifying, for an output file, the size of a logical page in terms of the number of lines. It can also be used to specify the size of the top and bottom margins on the logical page and the line number, within the page body, at which the footing area is to begin.

Format

```
LINAGE IS {data-name-1 | integer-1}
     LINES [WITH FOOTING AT {data-name-2 | integer-2}]
    [LINES AT TOP {data-name-3 | integer-3}]
    [LINES AT BOTTOM {data-name-4 | integer-4}]
```

Syntax rules

1. data-name-1, data-name-2, data-name-3, and data-name-4 must be elementary unsigned numeric integer data items.
2. data-name-1, data-name-2, data-name-3, and data-name-4 may be qualified.
3. The value of integer-1 or of the data item referenced by data-name-1 must be greater than zero.
4. The value of integer-2 or of the data item referenced by data-name-2 must be greater than zero, but not greater than integer-1 or the value of the data item referenced by data-name-1.
5. The value of integer-3 and integer-4, or the data items referenced by data-name-3 and data-name-4, may be 0.
6. The LINAGE clause is not permitted for files which are opened with OPEN EXTEND.
7. The LINAGE clause is permitted only for files which are assigned to PRINTER literal-1 or literal-2.
8. The LINAGE clause provides a means of specifying the size of a logical page in terms of the number of lines. The logical page size is the sum of the values of each phrase in the LINAGE clause except the FOOTING phrase. If the LINES AT TOP or LINES AT BOTTOM phrases are not specified, the value for this function is zero. If the FOOTING phrase is not specified, no footing area is defined (see table 11).

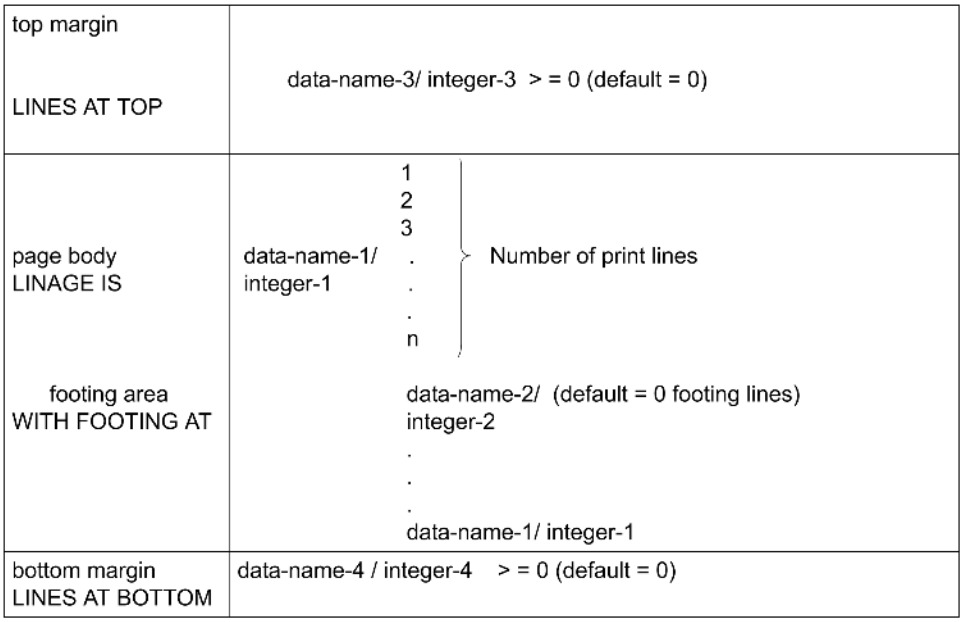


Table 11: Setup of a logical page

9. The size of a logical page must not necessarily correspond to the size of a physical page.
10. The value of integer-1 or the data item referenced by data-name-1 specifies the number of lines that can be written and/or spaced on the logical page. This part of the logical page, in which these lines can be written and /or spaced, is called the page body.
11. The value of integer-3 or the data item referenced by data-name-3 specifies the number of lines forming the top margin of the logical page. This area is left blank.
12. The value of integer-4 or the data item referenced by data-name-4 specifies the number of lines forming the bottom margin of the logical page. This area is left blank.
13. The value of integer-2 or the data item referenced by data-name-2 specifies the line number within the page body at which the footing area begins.
14. The footing area comprises the area of the logical page between the line specified by integer-2 or data-name-2 and the line represented by integer-1 or data-name-1.
15. The values of integer-1, integer-3 and integer-4 (or the values of data items data-name-1, data-name-3 and data-name-4, if specified) are used at object time by an OPEN statement (with the OUTPUT phrase) to specify the number of lines in each of the indicated parts of the first logical page. At the same time, the value of integer-2 or of data item referenced by data-name-2 (if specified) is used to define the footing area.

    If a page overflow occurs during execution of a WRITE statement with the ADVANCING phrase, the values of integer-1, integer-3 and integer-4 are used to specify the number of lines that comprise each of the indicated sections of the next logical page.

    The value of integer-2 or of data item referenced by data-name-2 (if specified) is then used to define the footing area of the next logical page.

General rules

1. The COBOL register LINAGE-COUNTER is generated whenever a LINAGE clause occurs. The LINAGE-COUNTER value at any given time represents the line number at which the printer is positioned within the current page body. The first printable line on each logical page has the number 1.

    A separate LINAGE-COUNTER is supplied for each file described in the FILE SECTION whose file description entry contains a LINAGE clause.
2. The LINAGE-COUNTER may be accessed, but must not be modified, by Procedure Division statements. Since more than one LINAGE-COUNTER may exist in a program, the user must qualify LINAGE-COUNTER by file-name when necessary.
3. The LINAGE-COUNTER is automatically updated during the execution of a WRITE statement for the file:
    a. If the ADVANCING PAGE phrase is specified in the WRITE statement, the LINAGE-COUNTER is automatically reset to the value 1.
    b. If ADVANCING integer or ADVANCING identifier-2 is specified in the WRITE statement, the LINAGE-COUNTER is incremented by integer or by the value of the data item referenced by identifier-2.
    c. If the ADVANCING phrase is omitted in the WRITE statement, the LINAGE-COUNTER is automatically incremented by the value 1.
    d. The LINAGE-COUNTER is automatically reset to the value 1 when the printer is positioned to the first line to be written on the next logical page (see "WRITE statement").
    e. The LINAGE-COUNTER is automatically set to the value 1 at the time an OPEN statement is executed for the file.
4. If the LINAGE clause refers to an *external* file, an equivalent LINAGE clause must be specified in all programs that describe this external file. In contrast to the Standard, the compiler described here requires specifications

only of the same type (i.e. either only data-name specifications or only integer specifications). The contents of the data items or the numerical values may be different.

7.2.2.8 RECORD clause

Function

The RECORD clause defines the length of the records in a file and controls the external record format (for **sequentially organized files**, see also the "RECORDING MODE clause").

Format 1	Indicates fixed-length records by specifying the number of bytes in a record.
Format 2	Indicates variable-length records, whereby the size of the record must lie within a certain range.
Format 3	Indicates variable-length records, whereby the minimum and maximum number of bytes are specified.

Format 1

```
RECORD CONTAINS integer-1 CHARACTERS
```

Syntax rules

1. The length of each record is precisely defined by its record description entry. The length specification in the RECORD clause is ignored in this respect.
2. The number of bytes in each record description entry for the file must be equal to integer-1.
3. integer-1 must be at least 1 and at most 32767. When the RECORD clause is used in a sort file description entry, the maximum permitted value for integer-1 is equal to 32755 minus the sort key length.

Format 2

```
RECORD IS VARYING IN SIZE [[FROM integer-2] [TO integer-3] CHARACTERS]
    [DEPENDING ON file-name-1]
```

Syntax rules

1. In any record description entry, it is impermissible for the length specification to be less than integer-2 or more than integer-3.
2. integer-3 must be larger than integer-2.
3. integer-2 must be larger than 0 or equal 0, while integer-3 can be at most 32763. When the RECORD clause is used in a sort file description entry, the maximum permitted value for integer-3 is equal to 32751 minus the sort key length.
4. data-name-1 must be described as an unsigned integer data item in the WORKING-STORAGE, LOCAL-STORAGE or LINKAGE SECTION.
5. This format must not be specified for **relative organized files** which are to be processed using the DMS access method UPAM (see "COBOL2000 User Guide" [1]).

General rules

1. If integer-2 is omitted, it is assumed that the length of the shortest record is 1. For each SORT or MERGE statement, the sort key must lie completely within the range of this minimal length.
2. If integer-3 is omitted, the length of the longest record in the record description entry for this file is assumed. This number also provides the block size.
3. If data-name-1 is specified, the number of bytes in the record must be moved to data-name-1 before a RELEASE, REWRITE or WRITE statement is executed for this file.
4. If data-name-1 is specified, its contents remain unchanged when a RELEASE, REWRITE or WRITE statement is executed or when a READ or RETURN statement terminates abnormally.
5. During execution of a RELEASE, REWRITE or WRITE statement, the record length is determined as follows:
   a. If data-name-1 is specified: by the contents of the data item referenced by data-name-1.
   b. If data-name-1 is omitted and the record description entry has no OCCURS clause with DEPENDING ON phrase: by the number of bytes in the record.
   c. If data-name-1 is not specified and the record description entry contains no OCCURS clause with DEPENDING ON phrase: by the fixed portion of the record description entry (i.e. all data description entries without the DEPENDING ON phrase) and by the number of occurrences of the table elements during execution of the statement.
6. If data-name-1 is specified and a READ statement has been successfully executed, data-name-1 contains the number of bytes of the record just read. This number is, however, not greater than integer-3, but it may be larger than the largest record description entry.
7. If INTO is specified in a READ or RETURN statement, the number of bytes in the current record, which is the source field in the implicit MOVE, is determined as follows:
   a. If data-name-1 is specified: by the contents of the data item referenced by data-name-1.
   b. If data-name-1 is not specified: by the value that would have been transferred had data-name-1 been specified.
8. If the number of bytes determined is 0, the record is a zero-length item.

Format 3

```
RECORD CONTAINS integer-4 TO integer-5 CHARACTERS
```

Syntax rules

1. integer-4 describes the number of bytes in the shortest record; integer-5 the number in the longest.
2. In no record description entry for the file may the length be less than specified in integer-4 or greater than specified in integer-5.
3. integer-5 must be greater than integer-4.
4. integer-4 must be larger than 0 or equal to 0, integer-5 must not exceed 32763. When the RECORD clause is used in a sort file description entry, the maximum permitted value for integer-5 is 32751 minus the sort key length.
5. This format must not be specified for **relative organized files** which are to be processed using the DMS access method UPAM (see "COBOL2000 User Guide" [1]).

General rules for all formats

1. The length of each record is precisely defined by its record description entry. If the RECORD clause is specified, the length is compared with the specifications given in the clause (for **sequentially organized files**, see also the "RECORDING MODE clause").
2. The length of a record is determined by the sum of the bytes of all its elementary items and the slack bytes generated by the compiler. If the record contains a table, the minimum or maximum number of table elements is taken into account when calculating the length. For further details, see section "SYNCHRONIZED clause", section "USAGE clause" and section "Implementor-dependent representation and alignment of data".
3. If the RECORD clause is not specified, the minimum or maximum record length results from the specifications in the corresponding record description entry. If a record description entry contains an OCCURS clause with the DEPENDING phrase, the value 1 is assumed as the minimum record length for this.
4. If the RECORD clause is specified for an *external* file, a RECORD clause of the same format must be specified in all programs that describe this external file; the minimum and maximum record lengths calculated from the specifications in the RECORD clause must match those from the corresponding record description entries.
5. The following table shows which specifications in the formats of the RECORD clause are decisive in calculating the minimum and maximum record lengths.

	Format 1	**Format 2**	**Format 3**	**No RECORD clause**
Minimum record length	Longest record of record description entry	integer-2; if not specified: 1	integer-4	Shortest record of record description entry; with OCCURS DEPENDING: 1
Maximum record length	as above	integer-3; if not specified: longest record of record description entry	integer-5	Longest record of record description entry

Table 12: Specifications in the RECORD clause

6. The specified maximum limits are reduced by 8 bytes for **files with relative organization**.
7. If the number of bytes determined is 0, the record is a zero-length item.

7.2.2.9 RECORDING MODE clause

Function

The RECORDING MODE clause specifies that the format of logical records in the file is "undefined".

Format

```
RECORDING MODE IS U
```

Syntax rule

1. Specifying U means that the file may contain any combination of fixed-length or variable-length records. Records in mode U cannot be blocked and have no preceding control data item (record length field, RLF). If RECORDING MODE IS U is specified, there is no need to specify the BLOCK CONTAINS clause.

General rules

1. The record formats "F" (fixed-length records) and "V" (variable-length records) are defined by the RECORD clause:

   fixed record length by a RECORD clause in format 1, variable record length by a RECORD clause in format 2.
2. If the RECORDING MODE clause refers to an *external* file, an equivalent RECORDING MODE clause must be specified in all programs that describe this external file.

7.2.2.10 VALUE OF clause

Function

The VALUE OF clause particularizes the description of data items in a label record.

Format

```
VALUE OF {{IDENTIFICATION | ID} IS {data-name-1 | literal-1}} ...
```

The VALUE OF clause is treated as a comment by the compiler.

7.3 Data description entry

7.3.1 General description

Organization of entries in the Data Division

Data description entry is the general term for the description of every single data item in the Data Division; the entry is composed of the level number, followed by a data-name (if necessary) and several data clauses.

The **record description entry** is used to define all data description entries which are associated with a particular record; that is, the record description entry describes all properties of that record. The type description entry is a special form of record description entry. It is used as a pattern which can be reused in other record description entries.

Level numbers are used in structuring a logical record so that subdivisions of the record may be referenced. Once a record has been subdivided, this structuring may be carried further to permit even more detailed data references.

Table 13 shows the permitted level numbers and their associated Data Division entries.

Level-number	Use
01	Record description entries
02 - 49	Data description entries describing subdivisions of a record
77	Description entries for independent or noncontiguous data items which are not subdivisions of other items and are not themselves subdivided
66	Elementary items or group items described by the RENAMES clause for the purpose of regrouping data items (see section "RENAMES clause")
88	Condition-name entries to specify condition-names associated with particular values of a conditional variable (see section "VALUE clause")

Table 13: Meaning of level numbers

Consecutive data description entries may have the same format as the first such entry or may be intended according to their level numbers. While indentation is helpful for documentation purposes, it does not affect the compiler.

Multiply-defined 01- and 77-level record description entries are not treated as errors, provided they are not used in the Procedure Division. Multiply-defined type description entries are not treated as errors if they are not used in a TYPE clause or in the USAGE clause for type-specific data pointers.

The concept of level is contained in the structure of a logical record. This concept arises from the need of assigning names to the parts of a record in order to access them. Once a record has been thus subdivided, the subdivision can be carried further to permit even more detailed data references.

A "report group" is to the REPORT SECTION what a "record" is to other sections of the Data Division. The **report group description entry** describes all data description entries associated with a particular report group. Within a report group description entry, a distinction is made between the first and the subsequent data description entries (see chapter "Report Writer").

Those components of a record which are not further subdivided are called **elementary items**; a record thus either consists of a sequence of elementary items or is itself an elementary item.

An elementary item may be at the most 131071 characters long.

In order to reference a number of elementary items at one time, these items are arranged into **"groups"** or **"group items"**. These groups may in turn be arranged into sets of two or more groups. Consequently, an elementary item may belong to more than one group.

The word **"data item"** is used in those cases where there is no need to distinguish between elementary and group items.

Data items that bear no hierarchical relationship to one another are defined as independent elementary items in conjunction with the level number 77.

In addition to the mandatory clauses, additional data clauses can be used to complete the data description entry. Their use is described in section "Clauses for data description".

7.3.2 Data description entry formats

Function

A data description entry describes the attributes of a single data item.

Format 1

```
level-number   [data-name | FILLER]

               [REDEFINES clause]
               [ANY LENGTH clause]
               [BASED clause]
               [BLANK WHEN ZERO clause]
               [DYNAMIC clause]
               [EXTERNAL clause]
               [GLOBAL clause]
               [GROUP-USAGE clause]
               [JUSTIFIED clause]
               [OCCURS clause]
               [PICTURE clause]
               [SIGN clause]
               [SYNCHRONIZED clause]
               [TYPE clause]
               [TYPEDEF clause]
               [USAGE clause]
               [VALUE clause]
```

Syntax rules

1. level-number must be a number from 01 through 49 or 77.

   Descriptions of level 77 specify data items that are not hierarchically related and are not subdivided into smaller parts.
2. The clauses may be written in any order, except for the FILLER and data-name phrases and the REDEFINES and TYPEDEF clauses. The FILLER or data-name phrase must directly follow the level number. If the REDEFINES clause if specified, it must be the first clause ahead of all other clauses. The TYPEDEF clause may not be specified together with the REDEFINES clause. If the TYPEDEF clause is specified, it must follow immediately after the level number and data-name phrase.
3. The PICTURE clause must not be specified for elementary items that are defined with USAGE COMP-1, COMP-2, INDEX, OBJECT REFERENCE, POINTER or PROGRAM-POINTER. The PICTURE clause must be specified for all other elementary items, with the following exception:
   The PICTURE clause need not be specified for an elementary item if a VALUE clause of format 1 is specified with an alphanumeric or national literal and it is not followed by an entry with the level number 88.
   The compiler then assumes a PICTURE clause PICTURE X(length) or PICTURE N(length), where length is the number of characters represented by the literal.
4. The OCCURS clause must not be specified in a data description entry that has a level-number of 01, 66, 77 or 88.
5. If the ANY LENGTH clause is specified, then only the PICTURE clause and USAGE clause may be specified in addition to the level number and data name.
6. The following specifications are required in each data description entry

- data-name
- the PICTURE clause
  or
- the USAGE clause with the INDEX, COMPUTATIONAL-1, COMPUTATIONAL-2, OBJECT-REFERENCE, POINTER or PROGRAM-POINTER
  or
- a VALUE clause with an alphanumeric or national literal or a TYPE clause

Example 7-1

for the structure of a record with description of group items

```
01 DATA-RECORD. ——————————————— Logical record
   02 REF-NO PIC ... ————————— Elementary item
   02 CUSTMR-NO PIC ... ———————— Elementary item
   02 MAILING-ADDRESS. ———————— Group item                    |
      03 FIRST-NAME PIC ... ——— Elementary item               |
      03 LAST-NAME PIC ... ———— Elementary item               |
      03 STATE PIC ... ———————— Elementary item               | Group item
      03 CITY. ——————————————— Group item       |           |
         04 ZIP-CODE PIC ... —— Elementary item  | Group item |
         04 CITY-NAME PIC ... — Elementary item  |           |
      03 STREET PIC ... ————————— Elementary item             |
   02 ART-NO PIC ... ————————— Elementary item
   02 PRICE. ————————————————— Group item       |
      03 DOMES PIC ... ———————— Elementary item  | Group item
      03 FORGN PIC ... ———————— Elementary item  |
```

The group item contains no information on data class or size of item. Definitions (e.g. REDEFINES, OCCURS) can, however, follow the group item name. The entry ends with a period.

Format 2

```
66 data-name-1 RENAMES data-name-2 [{THROUGH | THRU} data-name-3].
```

For syntax rules and general rules, see section "RENAMES clause".

Format 3

```
88 condition-name {VALUE | VALUES} {IS | ARE} {literal-1 [{THROUGH | THRU} literal-2
]}...
```

For syntax rules and general rules, see “Format 2” in section "VALUE clause".

7.3.2.1 Level number

Function

The level number indicates the position of a data item within the hierarchical structure of a logical record. It also identifies entries for data items within the WORKING-STORAGE, LOCAL-STORAGE and LINKAGE SECTION, as well as for condition-names and data-items in the RENAMES clause.

Format

```
level-number
```

Syntax rules

1. The level number is a special numeric literal consisting of one to two digits. A level number which is less than 10 may be written either as a single digit or with a leading zero.
2. Data description entries subordinate to an FD or SD entry must have level numbers with the values 01 to 49, 66, or 88.
3. Data description entries subordinate to an RD entry may have the level numbers 01 and 02 only.
4. The first element in every data description entry must be a level number.

General rules

1. Level number 01 identifies the first entry of each record description or report group description.
2. Special level numbers are assigned to certain kinds of entries for which there is no real concept of hierarchy. These numbers include:

   Level number 66 is used to identify renaming entries. It may be used only as described in the RENAMES clause.

   Level number 77 is used to identify structurally noncontiguous data items in the WORKING-STORAGE, LOCAL-STORAGE and LINKAGE SECTION. It may be used only as described under "77-level description entry".

   Level number 88 refers to entries which define condition-names associated with a conditional variable. It may be used only as described in format 2 of the VALUE clause.
3. Multiple level-01 entries which are subordinate to a given level indicator (except RD) represent implicit redefinitions of the same area.

Example 7-2

```
01  ADDRESS.
    02  NAME.
        03  FIRST-NAME           PIC X(18).
        03  LAST-NAME            PIC X(20).
    02  STREET ADDRESS.
        03  ZIP-CODE.
            04  DIGIT-1          PIC 9.
            04  DIGIT-2          PIC 9.
            04  DIGIT-3          PIC 9.
            04  DIGIT-4          PIC 9.
            04  DIGIT-5          PIC 9.
```

```
        03  CITY                 PIC X(19).
        03  STREET               PIC X(16).
        03  HOUSE-NUMBER         PIC XXX.
```

The statement

```
    MOVE ADDRESS TO...
```

will move the entire group.

The statement

```
    MOVE NAME TO...
```

will move the first and last names etc.

7.3.3 Clauses for data description

7.3.3.1 ANY LENGTH clause

Function

The ANY LENGTH clause ensures that the length of a method's formal parameter is determined by the length of the argument.

Format

ANY LENGTH

Syntax rules

1. In data entries, the ANY LENGTH clause can only be specified with level number 01 or 77 in a method's LINKAGE SECTION.
2. The data entry must contain a PICTURE clause with a picture string consisting of a single "X" oder "N".

General rules

1. The ANY LENGTH clause defines the elementary item as being of variable length. The current length of the item corresponds to the length of the call argument or of the return value.
2. If the call argument or return value corresponding to the elementary item is a zero length item, the elementary item is also a zero-length item.

Example 7-3

```
METHOD-ID. SEARCH-CHAR.
DATA DIVISION.
LINKAGE SECTION.
01    PAR1    PIC X ANY LENGTH.
01    PAR2    PIC X.
01    IDX     PIC 9(9) USAGE COMP-5.
PROCEDURE DIVISION USING PAR1 PAR2 RETURNING IDX.
  PERFORM VARYING IDX FROM 1 BY 1 UNTIL IDX > FUNCTION LENGTH(PAR1)
    IF PAR1(IDX:1) = PAR2
        EXIT METHOD           *> character found ---->>>>
     END-IF
  END-PERFORM
  MOVE 0 TO IDX.            *> character not found
  EXIT METHOD.
END METHOD SEARCH-CHAR.
```

Extract from a program in which the SEARCH-CHAR method is called:

```
...
01 OBJ      USAGE IS OBJECT REFERENCE.
01 I        PIC 9(9) USAGE IS COMP-5.
01 F1       PIC x(50).
01 F2
   02 ELEM  OCCURS 100 DEPENDING ON I.
...
```

```
INVOKE OBJ "SEARCH-CHAR" USING F1,"X" RETURNING I. ————————— (1)
INVOKE OBJ "SEARCH-CHAR" USING F2,"Y" RETURNING I. ————————— (2)
...
```

(1) The current length of parameter PAR1 is 50. The length does **not** have to be passed by the user

(2) The current length of parameter PAR1 is dynamic and known only at runtime.

Changes to the RETURNING parameter IDX in the SEARCH-VAR method do not affect the length of the current F2 parameter until after a return from the SEARCH-CHAR method.

The different lengths of the current F1 and F2 parameters do **not** violate the conformance rules because of the presence of the ANY LENGTH clause.

7.3.3.2 BASED clause

The BASED clause identifies a record as being a simple template to which no storage space is allocated.

Format

BASED

Syntax rules

1. The BASED clause may be specified only for data definitions in the LINKAGE SECTION with level number 01 or 77.
2. It is not permitted to specify USAGE OBJECT REFERENCE, POINTER and PROGRAM-POINTER. In addition, data descriptions that are subordinate to the data description containing the BASED phrase must also not contain USAGE OBJECT REFERENCE, POINTER and PROGRAM-POINTER.
3. The BASED and REDEFINES phrases may not be used together.

General rule

1. The start address of this record is set to the predefined address NULL. An address is not assigned until it is explicitly set.

Example 7-4

```
WORKING-STORAGE SECTION.
    01 POINTER-A USAGE POINTER.
LINKAGE SECTION.
    01 DSECT-A BASED.
    02 ...
PROCEDURE DIVISION.
...
    SET ADDRESS OF DSECT-A TO POINTER-A.
```

7.3.3.3 BLANK WHEN ZERO clause

Function

The BLANK WHEN ZERO clause specifies that an item is to be set to blanks when its value is zero.

Format

```
BLANK WHEN ZERO
```

Syntax rules

1. The BLANK WHEN ZERO clause may be specified only at the elementary level for numeric-edited or numeric items.
2. The numeric or numeric edited data description entry to which the BLANK WHEN ZERO clause applies must be described, either implicitly or explicitly, as USAGE IS DISPLAY.

General rules

1. When the BLANK WHEN ZERO clause is used, the item will contain only blanks if the value of the item is zero.
2. When the BLANK WHEN ZERO clause is used for numeric data items, the category of the item is considered to be “numeric-edited”.
3. If the BLANK WHEN ZERO clause and the PICTURE clause with asterisk (*) (for zero suppression) are used simultaneously in a data description entry, the zero suppression editing function overrides the function of the BLANK WHEN ZERO clause (see section "PICTURE clause").

Example 7-5

```
IDENTIFICATION DIVISION.
PROGRAM-ID. BWHENZ.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
    TERMINAL IS T.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 PURCHASE-EXAMPLE.
   02 PURCHASE PICTURE $Z.99 BLANK WHEN ZERO.
PROCEDURE DIVISION.
MAIN SECTION.
P1.
    MOVE ZERO TO PURCHASE.
    DISPLAY PURCHASE UPON T.
    STOP RUN.
```

Value of PURCHASE after the MOVE statement:

'BLANK' 'BLANK' 'BLANK' 'BLANK' 'BLANK' (5 spaces)

7.3.3.4 DYNAMIC clause

Function

The DYNAMIC clause enables the dynamic provision of memory in a scope defined by the user.

Format

```
01 data-name IS DYNAMIC.
```

(level-number and data-name are not part of the DYNAMIC clause; they are specified here simply to improve clarity.)

Syntax rules

1. The DYNAMIC clause may only be specified in level 01 record description entries in the WORKING-STORAGE SECTION.
2. If the DYNAMIC clause is specified for a data item, no other clause may be specified for this data item.

General rule

1. The data item to which the DYNAMIC cause is applied is set up in main memory at object time and begins on a 4-Kbyte boundary.

7.3.3.5 Data-name or FILLER clause

Function

A data-name specifies the data being described. The reserved word FILLER specifies an elementary or group item which is never referenced explicitly and therefore need not be given a name.

Format

```
level-number [data-name | FILLER]
```

(The level number is not part of the data-name or FILLER clause; it is shown here merely for purposes of clarity.)

Syntax rules

1. data-name must be formed according to the rules for user-defined words.
2. In the FILE, WORKING-STORAGE, LOCAL-STORAGE and LINKAGE SECTION, the first word in a data description entry following the level number must be a data-name or the reserved word FILLER.
3. The reserved word FILLER is used to give a name to an elementary item or group item which is never referenced in the program, and therefore does not require a data-name. A FILLER data item cannot be referenced directly.
4. If the data-name or FILLER entry is omitted, FILLER is assumed.

General rule

1. All referenced 01 and 77-level entries in the WORKING-STORAGE, LOCAL-STORAGE and LINKAGE SECTION must, if they are to be referenced, be given unique data-names, since neither type of entry can be qualified. A subordinate data-name need not be unique if it can be qualified in a manner which makes it unique.

Example 7-6

```
01 REC.
02 FIRST-NUMBER     PICTURE 9(8).
02 SECOND-NUMBER    PICTURE 9(12).
02 FILLER           PICTURE X(60).
```

Here, a record is identified by the data-name RECORD, and its first two fields are identified by the data-names FIRST-NUMBER and SECOND-NUMBER. Since the third field is not referenced in the program, its level number is followed by the reserved word FILLER.

7.3.3.6 EXTERNAL clause

Function

With the EXTERNAL clause, a record can be defined as external. External files and records can be accessed by any program in which the file or record is described.

Format for file description entries

```
IS EXTERNAL
```

Syntax rules

1. The EXTERNAL clause may be included only in the record description entry of the WORKING-STORAGE SECTION.
2. A data-name declared as external may not be declared again within the same program.
3. The EXTERNAL clause may refer only to a data description entry with level number 01.
4. Names of external records can have a maximum length of 30 characters.
5. The following must not be used as the name of an external record:
   - names of external files from other compilation units in the run unit
   - PROGRAM-ID names of the run unit, except names of contained programs in a nested program
   - names used as entry points in an ENTRY statement
   - names which identify interfaces (LZS-name, etc.).
6. The EXTERNAL clause and the REDEFINES clause must not be used together in the same data description entry.
7. The EXTERNAL clause and the TYPEDEF clause may not be used in the same data description entry.
8. The EXTERNAL clause may not be specified for a data item of the class “object”, “pointer” or "strongly typed".

General rules

1. An external record that is described in several programs of a run unit must have the same name in each of these programs and must also have the same length. The compiler permits the use of different length definitions, but this possibility should not be used if CALL identifier is used in the program.
2. If a record is defined as external, the associated file is not implicitly an external file.
3. VALUE clauses which are specified in a data description which is assigned or subordinate to a data description with EXTERNAL clause are only meaningful when an INITIALIZE statement is executed.
   If, in more than one program, VALUE clauses are assigned or subordinate to an external record which is defined in several programs of a run unit, these VALUE clauses must be identical. The compiler also permits different VALUE clauses.

Additional rules, depending on the module format

The following applies to the name of any external record when generating the *OMF format (see the "COBOL2000 User Guide" [1]):

1. The eighth character must not be a hyphen.

2. Only the first 8 characters of the name are used for identification. These characters should therefore be unique for each external name in the run unit.

7.3.3.7 GLOBAL clause

Function

The GLOBAL clause can only be used within a nested program. It defines a file-name, data-name or report-name as global. A global name can be accessed by the program defining it and by any other program contained directly or indirectly in this program.

Format

```
IS GLOBAL
```

Syntax rules

1. The GLOBAL clause may only be specified in entries with level number 01:
   - for data description entries in the WORKING-STORAGE or FILE SECTION.
   - for type description entries in the WORKING-STORAGE, LOCAL-STORAGE, LINKAGE or FILE SECTION.
2. If two data items are defined in the same Data Division with the same name, the GLOBAL clause must not be specified in any of the corresponding data description entries.
3. The GLOBAL clause may not be specified in a method, factory or object definition.

General rules

1. A data-name, whose description contains a GLOBAL clause is a global name. All data-names subordinate to a global name and all condition-names associated with a global name are global names.
2. Any program contained in the program that describes the global name may access a global name. The global name does not need to be described again in the program that references it. In the case of references to identical names, the local names have priority (see "Determining the valid name").
3. If a data description entry also contains the REDEFINES clause in addition to the GLOBAL clause, the redefined data item is not necessarily global by implication.
4. If a global data item contains a variable-length table, then the corresponding DEPENDING ON element in the same Data Division must also be described as global.
5. The data-names in the CONFIGURATION SECTION are always implicitly global.
6. The index of an indexed table assigned to a global data item is also global.

7.3.3.8 GROUP-USAGE clause

Function

The GROUP-USAGE clause enables a group item to be treated like an elementary national item.

Format

```
GROUP-USAGE IS NATIONAL
```

Syntax rules

1. The GROUP-USAGE clause may only be specified for group items. They may not be strongly typed.
2. The specification implies USAGE NATIONAL for the data entry. A USAGE clause may not be specified (in addition).
3. All subordinate elementary items must be described explicitly or implicitly with USAGE NATIONAL. All subordinate group items must be described explicitly or implicitly with GROUP-USAGE NATIONAL.
4. The GROUP-USAGE clause may not be specified in data description entries with the level number 01 in the FILE SECTION.

General rules

1. The clause makes the group a national group. Its class and category are national.
2. If in a particular case nothing else is specified, a national group is always treated like an elementary item with USAGE NATIONAL and PICTURE N(m), m being the length of the group.
3. GROUP-USAGE NATIONAL is assumed implicitly for all subgroups.
4. If a GROUP-USAGE clause is not specified explicitly or implicitly, a strongly typed group is always an alphanumeric group.

> The GROUP-USAGE clause is required when or group containing only national data items is also to be treated as a group like a national item, e.g. when filling with blanks when moves take place.

7.3.3.9 JUSTIFIED clause

Function

The JUSTIFIED clause permits non-numeric data to be aligned within a non-numeric receiving item in an alternative manner to the standard.

Format

```
{JUSTIFIED | JUST}  RIGHT
```

Syntax rules

1. JUST is the abbreviation of JUSTIFIED.
2. The JUSTIFIED clause can be specified for elementary items only.
3. The JUSTIFIED clause can only be specified for alphabetic, alphanumeric or national data items.
4. The JUSTIFIED clause must not be specified for data items with level number 66 or 88.
5. The JUSTIFIED clause must not be specified for a receiving item of a STRING statement (see "STRING statement").

General rules

1. If the JUSTIFIED clause is specified for the receiving item, and the sending item is longer than the receiving item, the data is aligned at the rightmost character position, and the leftmost characters of the sending item are truncated.

   If the JUSTIFIED clause is specified for the receiving item, and the receiving item is longer than the sending item, the data is aligned at the rightmost character position, and the leftmost character positions are filled with blanks according to the class of the receiving item.
2. When the JUSTIFIED clause is omitted, the standard rules for data alignment within an elementary item are applicable (see the section "Concept of computer-independent data description").

Example 7-7

Normal alignment (without JUSTIFIED):

Sending item smaller than Receiving item

Sending item larger than or equal to Receiving item

Alignment when JUSTIFIED clause is specified:

Sending item smaller than Receiving item

Sending item larger than Receiving item

7.3.3.10 OCCURS clause

Function

The OCCURS clause is used to define tables. It specifies how many elements a table is to have, i.e. how often a data item is to recur. All elements have the same format. The size of the table may be variable. Furthermore, indices can be supplied.

Format 1	specifies the exact number of occurrences of a data item.
Format 2	specifies a variable number of occurrences of a given data item, ranging between a **maximum** and a **minimum** number of occurrences. The minimum number may be omitted.

Format 1

```
OCCURS integer-2 TIMES
    [{ASCENDING | DESCENDING} KEY IS {data-name-2}... ] ...
    [INDEXED BY {index-1}...]
```

Format 2

```
OCCURS [integer-1 TO] integer-2 TIMES DEPENDING ON data-name-1
    [{ASCENDING | DESCENDING} KEY IS {data-name-2}... ] ...
    [INDEXED BY {index-1}...]
```

Syntax rules for both formats

1. The OCCURS clause must not be specified in a data description entry that:
   a. has a level number of 01, 66, 77 or 88, or
   b. describes an item whose size is variable (the size of an item is variable if the data description entry of any item subordinate to it contains an OCCURS clause with the DEPENDING phrase).
2. data-name-1, data-name-2,... may be qualified with OF or IN (see section "Qualification").
3. data-name-2 may either be the subject of the OCCURS clause (in which case data-name-2,... must be specified at first), or it must be subordinate to the group item referenced by the OCCURS clause.
4. If data-name-2 does not match the subject of the OCCURS clause, data-name-2 must not be described with an OCCURS clause. Furthermore, it must not be subordinate to an entry which contains an OCCURS clause.
5. data-name-2 is subject to the following additional rules:
   a. Up to 12 key fields may be specified for a given table element.
   b. The sum of the lengths of all key fields associated with a table element must not exceed 256 bytes.
   c. The key fields may not be of the class object or pointer.
6. The OCCURS clause may not be specified together with an IDENTIFIED clause or in an entry which is directly subordinate to an entry with an IDENTIFIED clause.

Syntax rules for format 2

7. integer-1 must be a positive integer or 0.
8. When used together, integer-1 must be less than integer-2.
9. data-name-1 must be defined as an integer numeric data item.
10. If data-name-1 appears in the same record as the table whose occurrences it controls, it must appear before the variable portion of that record. In other words, the data item defined by data-name-1 must precede the record portion described by the OCCURS clause with the DEPENDING ON phrase.
11. If the OCCURS clause is specified in a data description entry included in a record description entry containing the EXTERNAL or GLOBAL clause, data-name-1, if specified, must reference a data item possessing the EXTERNAL or GLOBAL attribute which is described in the same Data Division.
12. A data description entry containing an OCCURS clause with the DEPENDING ON phrase may only be followed, within that record description, by data description entries which are subordinate to it. The COBOL2000 compiler also allows data description entries which are independent of data hierarchy (see general rule 15).
13. The OCCURS clause may not be specified in the record description of a file with organization XML.
14. Any entry that contains an OCCURS clause with the DEPENDING ON phrase, or has a subordinate entry with an OCCURS... DEPENDING ON clause, must not contain a REDEFINES clause.

General rules for both formats

1. integer-2 represents the exact number of occurrences.
2. Except for the OCCURS clause itself, all data description clauses associated with the item whose description contains that OCCURS clause apply to each occurrence of the item described.
3. The subject of an OCCURS clause must be indexed or subscribed whenever it is referenced in a statement.

   Exception: SEARCH and SORT (Format 2) statement.

   If the subject is the name of a group item, then all data-names belonging to the group must be indexed or subscribed whenever they are used as operands.

   An indexed or subscribed data-name references one particular table element. When used in a SEARCH and SORT (Format 2) statement, the data-name refers to the entire table.
4. The ASCENDING/DESCENDING phrase in conjunction with the INDEXED BY phrase is used in the execution of a SEARCH ALL and SORT (Format 2) statement.
5. Before an index-name may be used as an index, it must be initialized (using a SET, SEARCH ALL or PERFORM statement with VARYING phrase).
6. The ASCENDING/DESCENDING KEY phrase defines whether the elements in the table are to be arranged in ascending or descending order, according to the values contained in data-name-2. The data-names must be listed in descending order of significance. This order is presupposed in the SEARCH ALL statement (In this case the user is responsible for seeing that the table elements are sorted properly).
7. The INDEXED BY phrase indicates that indexing may be used to reference the data-name which is the subject of the OCCURS clause, or any entry subordinate to that data-name. The storage allocation and format for such defined indices are automatically defined by the compiler.
8. Each index contains a binary value that represents a displacement from the beginning of the table, corresponding to an occurrence number. The value is calculated as the occurrence number minus one, multiplied by the length of the entry that is indexed by the index-name (see "Indexing").
9. If a data item within a table element requires a particular alignment, the compiler ensures that this alignment also applies for every repetition of the table element (see section "Implementor-dependent representation and alignment of data").

General rules for format 2

10. integer-1 and integer-2 specify the minimum and maximum number of occurrences, respectively. The value of the data item referenced by data-name-1 must range between integer-1 and integer-2.
11. During an access to the table the current value of data-name-1 must not exceed that of integer-2.
12. When used in the DEPENDING ON phrase of the OCCURS clause for a data area of variable length, data-name-1 must be supplied with a value before this area is used in a MOVE operation as a sending or receiving item. The same data-name-1 may be used by the sending and receiving items (see Example 7-8).
13. The DEPENDING ON phrase specifies that the data item described by the OCCURS clause has a variable number of occurrences. The number of occurrences is controlled at object time by the value of data-name-1.
14. When reference is made to a group item to which an entry with an OCCURS DEPENDING ON clause is subordinate, the part of the table area used in the operation is determined as follows:
    a. If the data item referenced by data-name-1 is outside the group, only that part of the table area that is specified by the value of the data item referenced by data-name-1 at the start of the operation will be used. If no other entries are subordinate to the group item and the content of data-name-1 is 0, the group item is a zero-length item.
    If only entries which are defined with an OCCURS clause with DEPENDING ON phrase are subordinate to the group item and the content of all data items identified by data-name-1 is 0, the group item is a zero-length item.
    b. If the data item referenced by data-name-1 is included in the same group and the group data item is referenced as a sending item, only that part of the table area that is specified by the value of the data item referenced by data-name-1 at the start of the operation will be used in the operation. If the group is a receiving item, the maximum length of the group will be used.
15. If, within a record description entry, a data area follows data items with the DEPENDING ON phrase, but is not subordinate to those items, then its position depends on the current values of data-name-1 in the preceding DEPENDING ON phrases.

    If the value of data-name-1 is changed (i.e. change of table length), the position of these data areas are shifted accordingly. However, their original contents are **not** shifted (see Example 7-8).

> ¡ If the value of data-name-1 is reduced at object time, the contents of the data items with occurrence numbers greater than the new value of data-name-1 are undefined.

Example 7-8

Supplying a value to data-name-1 in OCCURS DEPENDING ON, using a MOVE operation. Given the following data definition:

```
WORKING-STORAGE SECTION.
01  ITEM-A.
    02  COUNTER-A PIC 9.
    02  DATA-AVALUE "ABCDEFGHI".
        03  CHARACTER-A PIC X OCCURS 1 TO 9
            DEPENDING ON COUNTER-A.
01  ITEM-B.
    02  COUNTER-B PIC 9.
    02  DATA-B.
        03  CHARACTER-B PIC X OCCURS 1 TO 9
            DEPENDING ON COUNTER-B.
```

The following MOVE operations are to be performed:

Case a) Sending item longer than receiving item

```
MOVE 6 TO COUNTER-A,
MOVE 3 TO COUNTER-B,
MOVE ITEM-A TO ITEM-B.
```

Contents following MOVE operation:

```
ITEM-A: 6ABCDEF
ITEM-B: 6ABCDEF
```

MOVE DATA-A TO DATA-B would result in:

```
ITEM-A: 6ABCDEF
ITEM-B: 3ABC
```

Case b) Sending item shorter than receiving item (contents of both items as they were before case a)

```
MOVE 3 TO COUNTER-A,
MOVE 6 TO COUNTER-B,
MOVE ITEM-A TO ITEM-B.
```

Contents following MOVE operation:

```
    ITEM-A: 3ABC
    ITEM-B: 3ABC
```

MOVE DATA-A TO DATA-B would result in:

```
    ITEM-A: 3ABC
    ITEM-B: 6ABC'BLANK''BLANK''BLANK
```

The MOVE operations proceed according to the rules for alphanumeric moves (see "MOVE statement").

Example 7-9

`OCCURS DEPENDING ON data-name-1`

Given the following data definition:

```
WORKING-STORAGE SECTION.
01  DATA-RECORD.
    02  TABLE1.
        03  LEN         PIC 9.
        03  TAB-ELEM    PIC X OCCURS 1 TO 9
                        DEPENDING ON LEN.
    02  ITEM PIC X.
```

If the current value of LEN is 9, the following starting position of the items results:

DATA-RECORD	TABLE	LEN	9
		ELEM (1)	A
			B . . H
		ELEM (9)	I
	ITEM		J

After MOVE 1 TO LEN

the length of the table and hence the position of ITEM is changed:

<table>
<tr><td rowspan="3">DATA-RECORD</td><td rowspan="2">TABLE</td><td>LEN</td><td>1</td></tr>
<tr><td>ELEM (1)</td><td>A</td></tr>
<tr><td colspan="2">ITEM</td><td>B</td></tr>
<tr><td colspan="3">currently unused portion of DATA-RECORD</td><td>.
.
H
I
J</td></tr>
</table>

The data item LEN now has the value 1; the data item ITEM has the value B.

7.3.3.11 PICTURE clause

Function

The PICTURE clause describes the general characteristics and editing requirements of an elementary data item.

Format

```
{PICTURE | PIC}  IS character-string
```

Syntax rules

1. PIC is the abbreviation of PICTURE.
2. A character-string consists of certain allowable combinations of the characters in the COBOL character set. These combinations determine the category of data to which an elementary item belongs.
3. There are 18 characters or symbols that may be used in a character-string: A, N, comma (,), X, 9, P, Z, *, B, 0, +, minus (-), currency symbol ($), slash (/), S, V, period (.), credit (CR), and debit (DB). The functions of each character and symbol are described below under "Summary of characters and symbols in the PICTURE character-string".
4. The characters S, V, ., CR, and DB may appear only once in a PICTURE clause.
5. An integer enclosed in parentheses can follow the symbols A, N, comma (,), X, 9, P, Z, *, $, B, slash (/), 0, minus (-), and plus (+), to indicate the number of consecutive occurrences of the symbol. The number in parentheses must be at least 1 and must not exceed 131071.
6. At least one of the symbols A, N, X, Z, 9, or *, or at least two of the symbols +, -, or CS [1)] must be present in a character-string.
7. The maximum number of characters allowed in a character-string is 50. This does not limit the number of characters in the described elementary item which may be much more than 50.
8. The allowable combinations of symbols used in a character-string are shown in table 14.
   An X at an intersection means that, in a character-string, the "second symbol" specified in the associated column may be located at any position to the **right** of the "first symbol" located at the start of the row. The leftmost column and uppermost row for each symbol represent its use to the left of the decimal point position (l). The rightmost column and lowermost row for each symbol represent its use to the right of the decimal point position (r).

		Second symbol																			
		Non-floating insertion symbols							Floating insertion symbols						Other symbols						
		B 0 /	,	.	+ -	+ -	CR DB	CS[1)]	Z *	Z *	+ -	+ -	CS[1)]	CS[1)]	9	A X	S	V	P	P	N
First symbol					l	r			l	r	l	r	l	r					l	r	
Non-float. insertion symbols	B 0 /	X	X	X		X	X		X	X	X	X	X	X	X	X		X	X		
	,	X	X	X		X	X		X	X	X	X	X	X	X			X	X		
	.	X	X			X	X			X		X		X	X						

	+ -	l	X	X	X				X	X	X			X	X	X			X	X	X	
	+ -	r																				
	CR / DB																					
	CS[1)]		X	X	X		X	X		X	X	X	X			X			X	X	X	
Float. insertion symbols	Z *	l	X	X	X		X	X		X	X					X			X	X		
	Z *	r	X	X			X	X			X											
	+ -	l	X	X	X							X	X			X			X	X		
	+ -	r	X	X									X									
	CS[1)]	l	X	X	X	X		X						X	X	X				X	X	
	CS[1)]	r	X	X	X			X							X							
Other symbols	9		X	X	X		X	X								X	X		X	X		
	A X		X													X	X					
	S															X			X	X	X	
	V		X	X			X	X			X		X		X	X					X	
	P	l					X	X											X	X		
	P	r	X	X			X	X			X		X		X	X					X	
	N																					X

Table 14: Precedence of symbols used in the PICTURE clause

[1)]CS is the abbreviation for currency symbol.

9. The number of characters specified in the character-string is used to determine the size of the item. However, the actual internal storage requirements are determined by the combination of the PICTURE and USAGE clauses (see section "USAGE clause").
10. Data in following categories may be described with the PICTURE clause:
    - alphabetic
    - alphanumeric
    - alphanumeric edited
    - national
    - numeric
    - numeric edited
11. There are two general methods of performing editing in the PICTURE clause: by insertion, or by suppression and replacement.

    The following types of insertion editing are available:
    - simple insertion
    - special insertion
    - fixed insertion

- floating insertion

Two types of suppression and replacement editing are available:

- zero suppression and replacement with spaces
- zero suppression and replacement with asterisks (*).

General rule

1. The PICTURE clause is permitted only for elementary items.

Summary of characters and symbols in a PICTURE character-string

The characters and symbols that are permitted in a character-string have the following meaning:

A Each A in the character-string represents a character position that may contain only a letter or a space.

B Each B in the character-string represents a character position into which a space character will be inserted. Each space is counted in the size of the item.

N Each N in the PICTURE character-string represents a character position which can contain any character from the UTF-16 character set.

P P represents a numeric digit position which is counted in the number of character positions; however, storage space is not reserved for it. At the same time it shows the position of the conceptual assumed decimal point:

- to the left of the Ps if Ps are the leftmost characters of the character-string
- to the right of the Ps if the Ps are the rightmost characters of the character-string

In some operations which address an elementary item whose PICTURE character-string contains the character P, the actual content of the data item is replaced by its algebraic value. For the algebraic value each P is treated as if it contained a zero and the decimal point were at the specified position.
Operations which use the algebraic value:

- Every operation which requires a numeric sending item.
- An elementary MOVE statement (elementary move) with numeric sending item whose PICTURE character-string contains one or more Ps.
- A MOVE statement with numerically edited sending item whose character-string contains one or more Ps and a numeric or numeric-edited receiving item.
- A comparison operation in which both operands are numeric.

In all other operations the digit positions described with the character P are ignored and not included in the size of the data item.

S The character S indicates the presence but not the location or mode of representation of an operational sign. If used, it must be the leftmost character of the character-string. The S is not counted in the size of the item unless the entry is subject to a SIGN clause which specifies the SEPARATE CHARACTER phrase.

V The character V indicates the position of an assumed decimal point. Since a numeric item cannot contain a printed decimal point, an assumed decimal point simply provides the compiler with information about the

scaling alignment of items involved in computations. Storage is never reserved for the character V; therefore, V is not counted in the size of the item. If the assumed decimal point is the rightmost character in the character-string, the character V need not be supplied.

X — Each X in the character-string represents a character position which may contain any allowable character from the EBCDIC set.

Z — Each Z in the character-string represents a leading numeric character position. If such a character position contains a zero, then the zero is replaced by a space. Each Z is counted in the size of the item.

9 — Each 9 in the character-string represents a character position that contains a numeral and is counted in the size of the elementary data item.

0 — Each zero in the character-string represents a character position into which the numeral zero will be inserted. Each zero is counted in the size of the item.

/ — Each slash (/) in the character-string represents a character position into which a slash will be inserted. Each slash is counted in the size of the item.

, — Each comma (,) in the character-string represents a character position into which a comma is inserted. Each comma is counted in the size of the elementary data item.

. — A period (.) in the character-string is an editing symbol and represents the decimal point used for alignment of the elementary data item. Additionally, it represents a character position into which a period is inserted. The period is counted in the size of the elementary data item.

Note

In a given program, the functions of the period and the comma are exchanged if the DECIMAL-POINT IS COMMA clause is specified in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION. The rules for the period then apply to the comma, and vice versa, whenever they appear in a PICTURE character-string.

+, -, CR, DB — These symbols are used as editing sign control symbols. When used, each represents the character position into which the editing sign control will be placed. These symbols are mutually exclusive in any one character-string, and each character used in the symbol is counted in determining the size of the elementary data item. Editing sign control symbols produce different results for positive and negative elementary data items, depending on the value (see "Fixed insertion editing").

* — This symbol is a check protection symbol. Each asterisk (*) in the character-string represents a leading numeric character position into which an asterisk will be placed if that position contains a zero. Each asterisk (*) is counted in the size of the item.

If the asterisk is used together with the BLANK WHEN ZERO clause in a data description entry, the print editing routine cancels the effect of the BLANK WHEN ZERO clause since it suppresses any zeros.

$ — The currency symbol ($) in the character-string represents a character position into which a currency sign is to be placed. The currency symbol in a character-string is represented either by the symbol $ or by the single character specified in the CURRENCY SIGN clause in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION. The currency symbol is counted in the size of the item.

Alphabetic elementary data items

Syntax rules

1. Only the symbol A is allowed in the character-string of an alphabetic item.
2. Any combination of the 52 letters of the alphabet and the space may be specified as the contents of an alphabetic elementary data item.

Example 7-10

Picture	Value
`PICTURE AAA`	`NEW`

Alphanumeric elementary data items

Syntax rules

1. The character-string for an alphanumeric elementary data item is restricted to certain combinations of the following symbols: A, X, 9.

   An alphanumeric item is treated as if its character-string contained all Xs, with each X representing one character position.
2. A character-string that contains all As or 9s does not define an alphanumeric item.

Example 7-11

The alphanumeric value AB1234 could be represented by any of the following character-strings:

```
PICTURE XXXXXX
PICTURE AAXXXX
PICTURE AA9999
PICTURE A(2)X(4)
```

National elementary data items

Syntax rule

1. The character-string for a national elementary data item is restricted to certain the symbol N.

Numeric elementary data items

There are two types of numeric elementary data items: fixed-point items and floating-point items.

Fixed-point items

There are three types of fixed-point items: external decimal, binary, and internal decimal (see section "USAGE clause").

Syntax rules

1. The character-string for a fixed-point elementary data item may contain any permissible combination of the following symbols: 9, V, P, S.

2. It can contain from 1 up to and including 31 digit positions.
3. If the symbol S has not been specified, a elementary data item may contain a combination of the digits 0 through 9.
4. If the symbol S has been specified, the elementary data item may contain, in addition to the above digits, a +, - or other representation of an arithmetic sign (see "SYNCHRONIZED clause").

Example 7-12

Valid combinations for fixed-point items:

```
PICTURE 9999
PICTURE S99
PICTURE S99V9
PICTURE PPP999
PICTURE S999PPP
```

Example 7-13

Let 8735 be the contents of a elementary data item.

For PICTURE P(4)9(4), the arithmetic value of this item is .00008735.

For PICTURE 9(4)P(2), the arithmetic value of this item is 873500.

Floating-point elementary data items

There are two types of floating-point elementary data items: internal and external floating-point items (see section "USAGE clause").

Syntax rules

1. The character-string for an external floating-point item has the following format:

   { + | - } mantissaE { + | - } exponent

   where the following rules are to be observed for the elements of the character-string:

   A positive or negative sign must be written immediately in front of the mantissa and the exponent in the character-string.

   + indicates that a plus sign is to represent positive values and a minus sign is to represent negative values.

   - indicates that a blank is to represent positive values and a minus sign is to represent negative values.

   mantissa

   The mantissa is the decimal part of the number after the decimal point; it consists of 1 to 18 ’9’s (each 9 representing a numeric character) and a leading, embedded, or trailing decimal point (.) or V. The decimal point indicates an actual (printed) decimal point, and the V indicates an assumed decimal point; these two characters are mutually exclusive.

   E

   immediately follows the mantissa and indicates that an exponent follows.

   exponent

   The exponent immediately follows the second sign and consists of two consecutive 9s.

2. The PICTURE clause **must not** be specified for an internal floating-point item.

Example 7-14 External floating-point item

```
PICTURE   -9V99E-99
PICTURE   +9999.99E+99
PICTURE   -9(16)VE+99
PICTURE   +9(16).E-99
```

Alphanumeric edited elementary data items

Syntax rules

1. The character-string for an alphanumeric edited elementary data item is restricted to certain combinations of the following symbols: A, X, 9, 0, B, / (slash).
2. An alphanumeric edited character-string must contain at least one A or X, and at least one B or 0 or / (slash).
3. Only one type of editing is performed on alphanumeric edited elementary data items: simple insertion editing using the characters zero (0), slash (/) and space (B) (see rules for simple insertion editing, "PICTURE clause").

Example 7-15

Picture	Value
PICTURE BAAAB	'BLANK'NEW'BLANK'

Numeric edited elementary data items

Syntax rules

1. The character-string for a numeric edited elementary data item is restricted to certain combinations of the following symbols:
   B, / (slash), P, V, Z, 0, 9, , (comma), . (period), *, +, -, CR, DB, $.
2. The character-string must contain at least one of the symbols:
   0, B, / (slash), Z, *, +, , (comma), . (period), -, CR, DB or $.
3. The maximum number of digit positions in the character-string is 31.
4. The maximum length of a numeric edited elementary data item is 127 characters.
5. Allowable combinations of these characters are governed by the editing rules and the symbol precedence rules (see following syntax rules and table 14).

Simple insertion editing

Syntax rules

1. In simple insertion editing, the following insertion characters are used: , (comma), B (space), 0 (zero), and / (slash).
2. The insertion characters are counted in the size of the item, and represent the positions within the item into which they will be inserted.

Example 7-16

Category of data	PICTURE string receiving item	Data being moved	Edited result

Numeric edited	999,999 99B99B99 99B99B00 99/99/99	54321 654321 654321 654321	054,321 65'BLANK'43'BLANK'21 43'BLANK'21'BLANK'00 65/43/21
Alphanumeric edited	XXBXXX 000X(5) XX/XX	123AA A5CD3 CD05	12'BLANK'3AA 000A5CD3 CD/05

Special insertion editing

Syntax rules

1. Special insertion editing is performed by using the period (.) as an insertion character.

   In addition to being used as the insertion character, the period is also used as the decimal point for alignment purposes.
2. The assumed decimal point (represented by the character V) and the actual (printed) decimal point cannot be used in the same character-string.
3. Special insertion editing may be used on numeric edited elementary data items only.
4. As a result of special insertion editing, the insertion character (decimal point) appears in the item in the same position as shown in the character-string; thus, the insertion character is the actual decimal point. The actual decimal point is counted in the size of the item.

Example 7-17

PICTURE string of receiving item	Data being moved *)	Edited result
999.99	123&4	123.40
999.99	12&34	012.34
999.99	1&234	001.23
999.99	&1234	000.12

*) & designates the position of the assumed decimal point, which does not appear in the MOVE operation.

Fixed insertion editing

Syntax rules

1. The editing symbols used for fixed insertion editing are:
   + (plus), - (minus), CR (credit), DB (debit), and $ / € (currency symbol).
2. Only one currency symbol and only one of the editing sign control symbols (+, -, CR, DB) may be used in a given character-string.
3. The currency symbol may be preceded only by a plus or a minus symbol; otherwise, it must be the leftmost character.
4. The symbols CR or DB, when specified, must be either the leftmost or the rightmost character.
5. The plus or minus symbol, when specified, must be either the leftmost or the rightmost character.

6. Fixed insertion editing results in the editing character occupying the same character position in the edited item as in the character-string.
7. The symbols CR or DB, when used, represent two character positions which are counted in the size of the elementary data item. All fixed insertion editing characters are counted in the size of the elementary data item.
8. Editing sign control symbols produce the results listed in table 15, depending on the value of the elementary data item (see table 15).

Editing symbol in PICTURE character-string	elementary data item positive or zero	elementary data item negative
+ - CR DB	+ space 2 spaces 2 spaces	- - CR DB

Table 15: Editing sign control symbols and their results

Example 7-18

PICTURE string of receiving item	Data being moved *)	Edited result
+999.99	+123&45	+123.45
+999.99	-123&45	-123.45
-999.99	+123&45	123.45
-999.99	-123&45	-123.45
$999.99CR	+123&45	$123.45
$999.99CR	-123&45	$123.45CR
$999.99DB	+123&45	$123.45
$999.99DB	-123&45	$123.45DB

*) & designates the position of the assumed decimal point, which does not appear in the MOVE operation.

Floating insertion editing

Syntax rules

1. In floating insertion editing, the currency symbol ($) and the editing sign control symbols (+ and -) are used as insertion characters. These characters are mutually exclusive as floating insertion characters in the same character-string.

   Floating insertion editing is indicated in a character-string by the use of a sequence of at least two of the allowable insertion characters to represent the leftmost numeric character positions into which the insertion characters can be floated. Any of the simple insertion characters (, B 0 /) embedded in the sequence of floating insertion characters or to the immediate right of this sequence are part of this floating string.
2. Only two types of floating insertion editing may be specified in a character-string.:
   - Some or all of the leading numeric character positions to the left of the decimal point may be represented by insertion characters.

- All of the numeric character positions of the character-string may be represented by insertion characters.

3. The result of floating insertion editing depends on the representation in the character-string:
   - If the insertion characters are specified only to the left of the decimal point, a single insertion character is placed into the character position which immediately precedes the decimal point, or the first non-zero digit to the left of the character-string, and which is located inside the data represented by the insertion symbol string. The character positions preceding the insertion character are replaced with spaces.
   - If each of the numeric character positions in the character-string is represented by the insertion character, the result depends on the value of the elementary data item concerned. If the value is zero, the entire data will contain spaces. If the value of the item is not zero, the result is the same as that occurring when the insertion characters are specified only to the left of the decimal point.
4. Every floating insertion character is counted in the size of the elementary data item.
5. To avoid truncation of character positions, the programmer must form the character-string for the receiving item according to the following rule:
   - The minimum size of the character-string must equal the number of non-floating insertion characters which are used for editing in the receiving item, plus one floating insertion character.

Example 7-19

Receiving area PICTURE	Data being moved*)	Edited result
$$$$.99	123&12	$123.12
$$$$.99	3&12	$3.12
$$$$.99	&12	$.12
$,$$$.99	123&12	$123.12
$,$$$.99	3&12	$3.12
$,$$$.99	&12	$.12
+,+++.99	123&12	+123.12
+,+++.++	123&12	+123.12
$,$$$.99	-123&12	$123.12
-,---.99	-123&12	-123.12
-,---.99	123&12	'BLANK'123.12
$$,$$$.99	1234&56	
+,+++,999.99	-123456&78	$1,234.56
+,+++,+++.++	000&00	-123,456.78
		(blank)

*) & designates the position of the assumed decimal point which does not appear in the MOVE operation.

Zero suppression and replacement editing

Syntax rules

1. Suppression of leading zeroes in numeric character positions is indicated by the use of the alphabetic character Z or the character * (asterisk) as suppression symbols in a PICTURE character-string. These characters are mutually exclusive in the same character-string. If Z is used, the replacement character is a space. If * is used, the replacement character is a space. If * is used, the replacement character is * (asterisk).
2. Zero suppression and replacement editing in a character-string is achieved by using a string of one or more of the permissible symbols (* or Z) to represent leading numeric character positions which are to be filled with the

replacement characters when the associated character positions in the data contain zeros. Any of the simple insertion characters (, B 0 /) embedded in the string of symbols or to the immediate right of the string are part of the string; each of these simple insertion characters works like an * or a Z, until a non-zero character is encountered. The simple insertion characters (, B 0 /) and fixed insertion characters ($ + -) to the left of the suppression string are not subject to the rules for zero suppression and replacement.

3. Two types of zero suppression editing may be used in a character-string.
   - Some or all of the leading numeric character positions to the left of the decimal point may be represented by suppression symbols.
   - All of the numeric character positions in the character-string may be represented by suppression symbols.
4. The result of zero suppression and replacement editing depends on the representation in the character-string:
   - If the suppression symbols appear only to the left of the decimal point, any leading zero in the data which corresponds to a suppression symbol in the string is replaced by the replacement character. Suppression terminates at the first non-zero digit in the data represented by the suppression symbol string or at the decimal point.
   - If all numeric character positions in the character-string are represented by suppression symbols and the value of the elementary data item is not zero, the result is the same as if the suppression characters were only to the left of the decimal point. If the value is zero and the suppression symbol is Z, the entire elementary data item is replaced by spaces. If the value is zero and the suppression symbol is *, all characters in the elementary data item, except for the decimal point, are replaced by asterisks; in this case, zero suppression editing overrides the BLANK WHEN ZERO clause, if the latter is specified.
5. Each suppression character is included in the size of the elementary data item.

Example 7-20

Receiving area PICTURE	Data being moved*)	Editing result
ZZZZ.ZZ	0000&00	'BLANK'(x7)
****.**	0000&00	****.**
ZZZZ.99	0000&00	'BLANK''BLANK''BLANK''BLANK'.00
****.99	0000&00	****.00
ZZZZ.ZZ	+135&00	'BLANK'135.00
$**,***.**BDB	-2135&00	$*2,135.00'BLANK'DB
$BB****,**.99BBCR	-2135&00	$'BLANK''BLANK'**21,35.00'BLANK''BLANK'CR

*) & designates the position of the assumed decimal point which does not appear in the MOVE operation.

7.3.3.12 REDEFINES clause

Function

The REDEFINES clause allows the programmer to define different data description entries for the same area of computer storage.

Format

```
level-number  [data-name-1 | FILLER] REDEFINES data-name-2
```

(The level number, data-name-1 and FILLER are not part of the REDEFINES clause, and are shown here only for clarity.)

Syntax rules

1. The REDEFINES clause, when used, must immediately follow data-name-1.
2. The level numbers of data-name-1 and data-name-2 must be identical, but must not be 66 or 88.
3. The length of the data item of data-name-1 must be less than or equal to the length of the data item of data-name-2, if the associated level-number is not equal to 01. There is no such restriction in effect at level 01.
4. Data-name-2 may be qualified but not indexed or subscripted.
5. The data description entry for data-name-2 must not contain an OCCURS clause; however, data-name-2 may be subordinate to a data item which contains an OCCURS clause. In this case, the reference to data-name-2 in the REDEFINES clause must not be indexed or subscripted. A data item that is subordinate to data-name-2 may contain an OCCURS clause without the DEPENDING ON phrase (see "OCCURS clause").
6. Data-name-1 or any data item subordinate to data-name-1 may contain an OCCURS clause without the DEPENDING ON phrase. If data-name-1 contains an OCCURS clause, the size of data-name-1 is calculated by multiplying the length of one table element by the number of occurrences of the table element.
7. The REDEFINES clause must not appear in 01-level entries in the FILE SECTION (implicit redefinition is provided there automatically at 01-level).
8. Multiple redefinitions of the storage area are permitted but must all refer to the data-name supplied in the original definition.
9. Except for condition-name entries, the entries giving a new description of a storage area must not contain a VALUE clause.
10. No entries having level numbers numerically lower than that of data-name-1 and data-name-2 may occur between the descriptions of data-name-1 and data-name-2.
    Data entries having level numbers numerically equal to that of data-name-1 and data-name-2 may only occur between data-name-1 and data-name-2 if they also contain a REDEFINES clause.
11. The REDEFINES clause may be specified for an item subordinate to a redefined item, or for a data item which is subordinate to an item containing a REDEFINES clause.
12. The REDEFINES clause may not be located within a type description entry.
13. The REDEFINES clause may not be specified for data of the class "object" or "pointer" or for group items which contain such data.
14. data-name-2 may not be of the class "object" or "pointer" in a group item which contains such data or is a strongly typed group item.
15. data-name-2 may not be ANY LENGTH clause.

16. The REDEFINES clause may not be specified in the record description entry of a file with organization XML.

General rules

1. data-name-1 is the name of the data area associated with the redefinition.
   data-name-2 is the name of the original definition of the data area to be redefined.
2. Redefinition starts at data-name-2 and ends when a level number less than or equal to that of data-name-2 is encountered.
3. When an area is redefined, all descriptions of that area remain in effect. For example, if A and B are two separate data items sharing the same storage area, the procedure statements MOVE ALPHA TO A or MOVE BETA TO B could be executed at any point in the program. In the first case, ALPHA would be moved to A and would take the form specified by the description of A. In the second case, BETA would be moved to the same physical area and would take the form specified by the description of B. If both MOVE statements were executed successively in the order specified, the value BETA would overlay the value ALPHA; however, redefinition of an area does not erase any data and does not supersede a previous description.
4. Moving a data item from A to B when B is a redefinition of A amounts to moving an item to itself, and the result of such a move is unpredictable. The same is true of the opposite type of move; that is, moving A to B when A redefines B.
5. The use of data items defined by the PICTURE and USAGE clauses within an area can be redefined. Altering the use of an area by the REDEFINES clause does not, however, change any existing data.
6. When the SYNCHRONIZED clause is specified in a data entry that is redefining a previous data entry, the user should ensure that the area being redefined begins on the proper boundary: halfword, fullword, or doubleword.

Example 7-21

```
02  ALPHA.
    03  A-1 PICTURE X(3).
    03  A-2 PICTURE X(2).
02  BETA REDEFINES ALPHA PICTURE 9(5).
02  GAMMA.
```

BETA is data-name-1; ALPHA is data-name-2. BETA redefines the area assigned to ALPHA (that is, the area occupied by A-1 and A-2). Redefinition starts at BETA and ends at the next level number 02 (the number preceding GAMMA).

Example 7-22

(Multiple redefinitions)

```
02 ALPHA PICTURE 9(3).
02 BETA REDEFINES ALPHA PICTURE X(3).
02 GAMMA REDEFINES ALPHA PICTURE A(3).
```

Example 7-23

```
01  SAMPLE-AREA-1.
    02  FIRST-DEFINITION  PICTURE 99 VALUE 12.
```

```
    02  SECOND-DEFINITION REDEFINES FIRST-DEFINITION
        USAGE COMPUTATIONAL  PICTURE S9(4).
```

In this example, FIRST-DEFINITION is a 2-byte unsigned external decimal number with the value 12. This means that the contents of the two bytes in hexadecimal is X'F1F2'. SECOND-DEFINITION is also a number, and occupies the same two bytes; but it does not have the value 12. The data in these two bytes (X'F1F2') is unchanged by the redefinition; and, since SECOND-DEFINITION is a signed, binary number, this data has the value -3598.

Example 7-24

```
01 SAMPLE-AREA-2.
   02 FIRST-DEFINITION.
      03 ALPHA                     PICTURE X(3).
      03 BETA                      PICTURE X(5).
      03 GAMMA REDEFINES BETA      PICTURE 9(5).
      03 FILLER                    PICTURE X(10).
   02 SECOND-DEFINITION REDEFINES FIRST-DEFINITION PICTURE X(18).
```

In this example, one of the items subordinate to FIRST-DEFINITION is redefined: GAMMA REDEFINES BETA. This is permitted, and is not blocked by the fact that FIRST-DEFINITION is itself later redefined by SECOND-DEFINITION.

Example 7-25

```
01 SAMPLE-AREA-3.
   02 FIRST-DEFINITION PICTURE S9(7).
   02 SECOND-DEFINITION REDEFINES FIRST-DEFINITION.
      03 A-1 PICTURE A.
      03 N-1 REDEFINES A-1 PICTURE 9.
      03 FILLER PICTURE X(6)
```

In this example, one of the data items subordinate to SECOND-DEFINITION is redefined; N-1 REDEFINES A-1. This is permitted, and is not blocked by the fact that SECOND-DEFINITION itself is a redefinition.

7.3.3.13 RENAMES clause

Function

The RENAMES clause permits alternative, possibly overlapping, groupings of elementary data items. This clause assigns a new name to an item or items established by a record description. Unlike REDEFINES, the RENAMES clause does not redefine existing data descriptions but merely allows data to be accessed and/or grouped under alternative names while maintaining the previously defined data descriptions.

Format

```
66 data-name-1 RENAMES data-name-2 [ {THRU | THROUGH} data-name-3 ]
```

(The level number 66 and data-name-1 are not part of the RENAMES clause, and are shown only to improve clarity.)

Syntax rules

1. All entries of the RENAMES clause which refer to data items within a given logical record must immediately follow the last data description entry of the associated record description entry.
2. data-name-2 must precede data-name-3 in the record description. After each redefinition, the beginning point of the area described by data-name-3 must logically follow the beginning point of the area defined by data-name-2.
3. data-name-2 and data-name-3 must be the names of elementary items or groups of elementary data items in the associated logical record, and cannot be the same data-name.
4. The beginning of the area defined by data-name-3 must not lie to the left of the beginning of the area defined by data-name-2. The end of the area defined by data-name-3 **must** lie to the right of the end of the area defined by data-name-2. Hence, data-name-3 cannot be subordinate to data-name-2. Neither data-name-2 and data-name-3 nor the data that lies between them may be of the class “object” or “pointer”. Neither data-name-2 nor data-name-3 may come from a type description entry.
5. None of the data items within the area of data-name-2 and data-name-3, when specified, may have a variable size as described in the OCCURS clause (see "OCCURS clause" with DEPENDING ON phrase).
6. data-name-1 cannot be used as a qualifier and can be qualified only by the names of the associated 01-level, SD, or FD entries.
7. data-name-2 and data-name-3 may be qualified.
8. Neither data-name-2 nor data-name-3 may contain an OCCURS clause in its data description entry, nor may it be subordinate to a data item which contains an OCCURS clause in its data description entry.
9. The RENAMES clause may neither refer to another 66-level entry nor to a 77-level, 88-level, or 01-level entry.
10. The RENAMES clause may not be specified within a type description entry.
11. The RENAMES clause may not be specified in the record description entry of a file with organization XML.

General rules

1. More than one RENAMES clause may be written for the same logical record.
2. data-name-1 specifies an alternative definition for one or more data items.
3. data-name-2 or data-name-3 specifies the data item(s) to be renamed.
4. When data-name-3 is specified, data-name-1 is a group item that includes all elementary items:
   - starting with data-name-2 (if this is an elementary data item); or starting with the first elementary item within data-name-2 (if this is a group item).

- concluding with data-name-3 (if this is an elementary data item); or concluding with the last elementary item within data-name-3 (if this is a group item).

5. If data-name-3 is not specified, then data-name-2 may be either a group item or an elementary item. If data-name-2 is a group item, data-name-1 is treated as a group item; if data-name-2 is an elementary item, data-name-1 is treated as an elementary data item.

Example 7-26

The following example shows how a RENAMES clause may be used in an actual program:

```
01  INPUT-RECORD.
    02  ARTICLE-1.
        03  ARTICLE-NO            PIC 99.
        03  PRICE                 PIC 9999.
    02  ARTICLE-2.
        03  ARTICLE-NO            PIC 99.
        03  PRICE                 PIC 9999.
    02  ARTICLE-3.
        03  ARTICLE-NO            PIC 99.
        03  PRICE                 PIC 9999.
    66  ART-ONE RENAMES ARTICLE-1.
    66  ART-TWO RENAMES ARTICLE-1 THRU ARTICLE-2.
    66  ART-THREE RENAMES ARTICLE-1 THRU ARTICLE-3.
```

In this case, each reference to ART-ONE would access group item ARTICLE-1; each reference to ART-TWO, the group items ARTICLE-1 and ARTICLE-2; each reference to ART-THREE, the group items ARTICLE-1, ARTICLE-2 and ARTICLE-3.

7.3.3.14 SIGN clause

Function

The SIGN clause specifies the position and the mode of representation of the operational sign for numeric data items.

Format

```
[SIGN IS] {LEADING | TRAILING} [SEPARATE CHARACTER]
```

Syntax rules

1. The SIGN clause may be specified only for a numeric data description entry whose PICTURE contains the character S, or a group item containing at least one such numeric data description entry.
2. The numeric data description entries to which the SIGN clause applies must be described, explicitly or implicitly, as USAGE IS DISPLAY.
3. If a SIGN clause is specified for either a group item or an elementary numeric item subordinate to a group item for which a SIGN clause is also specified, the SIGN clause of the subordinate group or numeric data item takes precedence for that item.
4. If the CODE-SET clause is specified, any signed numeric data description entries must be described with the SIGN IS SEPARATE clause.
5. The SIGN clause specifies the position and the mode of representation of the operational sign. If entered for a group item, it applies to each numeric data description entry subordinate to that group. The SIGN clause applies only to numeric data description entries whose PICTURE contains the character S; the S indicates the presence, but not the mode of representation, of the operational sign.
6. A numeric data description entry whose PICTURE contains the character S, but to which no SIGN clause applies, has an operational sign, but neither the representation nor, necessarily, the position of the operational sign is specified by the character S. (For representation of the operational sign see section "USAGE clause").

General rules

1. If the SEPARATE CHARACTER phrase is not present, then:
   a. The letter S in a PICTURE character-string is not counted in determining the size of the item.
   b. The operational sign will be presumed to be associated with the leading (or, respectively, trailing) digit position of the elementary numeric data item. The TRAILING phrase is taken as this compiler’s default value.
   c. For the compiler, the operational sign is the half-byte C for positive and the half-byte D for negative.
2. If the SEPARATE CHARACTER phrase is present, then:
   a. The letter S in a PICTURE character-string is counted in determining the size of the item.
   b. The operational sign will be presumed to be the leading (or, respectively, trailing) character position of the elementary numeric data item; this character position is not a digit position.
   c. The operational signs for positive and negative are the standard data format characters + and -, respectively.
3. Every numeric data description entry whose PICTURE character-string contains the character S is a signed numeric data description entry. If a SIGN clause applies to such an entry and conversion is necessary for purposes of computation or comparisons, conversion takes place automatically.

Example 7-27

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SIGNEXPL.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
    TERMINAL IS T.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  FIELD1         PIC S999  SIGN IS LEADING SEPARATE.
01  GROUP1         USAGE IS DISPLAY.
    02  FIELD2     PIC S9(5) SIGN IS TRAILING SEPARATE.
    02  FIELD3     PIC X(15).
    02  FIELD4     PIC S99   SIGN IS LEADING.
01  FIELD5         PIC S9(9) SIGN IS TRAILING.
PROCEDURE DIVISION.
MAIN SECTION.
P1.
    MOVE ZEROES TO FIELD1,FIELD2,FIELD3,FIELD4,FIELD5.
    MOVE 3 TO FIELD4.
    MOVE -2 TO FIELD5.
    MOVE FIELD4 TO FIELD2.
    MOVE FIELD2 TO FIELD3.
    MOVE FIELD5 TO FIELD4.
    DISPLAY "Field1 = " FIELD1 UPON T.
    DISPLAY "Field2 = " FIELD2 UPON T.
    DISPLAY "Field3 = " FIELD3 UPON T.
    DISPLAY "Field4 = " FIELD4 UPON T.
    DISPLAY "Field5 = " FIELD5 UPON T.
    STOP RUN.
```

The contents of all fields after each MOVE statement are shown below.

After the first MOVE statement:

FIELD1	decimal	+	0	0	0										
	hexadecimal	4E	F0	F0	F0										
FIELD2	decimal	0	0	0	0	0	+								
	hexadecimal	F0	F0	F0	F0	F0	4E								
FIELD3	decimal	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	hexadecimal	F0	F0	F0	F0	F0	F0	F0	F0	F0	F0	F0	F0	F0	F0	F0
FIELD4	decimal	0+	0													
	hexadecimal	C0	F0													
FIELD5	decimal	0	0	0	0	0	0	0	0	0+						
	hexadecimal	F0	F0	F0	F0	F0	F0	F0	F0	C0						

After the second MOVE statement:

FIELD4	decimal	+	3
	hexadecimal	C0	F3

After the third MOVE statement:

FIELD5	decimal	0	0	0	0	0	0	0	0	2-
	hexadecimal	F0	F0	F0	F0	F0	F0	F0	F0	D2

After the fourth MOVE statement:

FIELD2	decimal	0	0	0	0	3	+
	hexadecimal	F0	F0	F0	F0	F3	4E

After the fifth MOVE statement:

FIELD3	decimal	0	0	0	0	3										
	hexadecimal	F0	F0	F0	F0	F3	40	40	40	40	40	40	40	40	40	40

After the sixth MOVE statement:

FIELD4	decimal	0-	2
	hexadecimal	D0	F2

7.3.3.15 SYNCHRONIZED clause

Function

The SYNCHRONIZED clause specifies the alignment of an elementary item on a natural boundary of the computer memory to ensure efficiency during the performance of arithmetic operations on the item.

As an extension to standard COBOL, the compiler described in this publication allows the SYNCHRONIZED clause to be specified at group level; this has the effect of aligning the elementary items subordinate to the group.

Format

```
{SYNCHRONIZED | SYNC} [LEFT | RIGHT]
```

Syntax rules

1. SYNC is the abbreviation of SYNCHRONIZED.
2. LEFT and RIGHT are treated as comments.
3. The SYNCHRONIZED clause may not be specified for data of the classes national, object and pointer.

General rules

1. When items are boundary-aligned because of the presence of a SYNCHRONIZED clause, it is sometimes necessary for the compiler to insert slack bytes. Slack bytes are unused character positions inserted into a record immediately ahead of an item requiring boundary alignment. Such slack bytes are included in the length of the group item containing the aligned item.
2. The actual boundary at which a synchronized item is placed depends on the USAGE clause for the item.
3. If the SYNCHRONIZED clause is **omitted** for binary data items or internal floating-point items, no slack bytes are generated. However, if arithmetic operations are performed on these items, the compiler will generate the statements necessary to move these items to auxiliary items which are properly aligned for the arithmetic operation.
4. If the SYNCHRONIZED clause is specified for a group item, then all elementary items with USAGE COMPUTATIONAL, COMPUTATIONAL-5, BINARY, COMPUTATIONAL-1 or COMPUTATIONAL-2 will be aligned as if the SYNCHRONIZED clause had been specified in the data description entries of these items.
5. If the SYNCHRONIZED clause is specified then the following actions will be performed:
   - For a data item with USAGE COMPUTATIONAL, COMPUTATIONAL-5 or BINARY:
     a. For the area S9 through S9(4) in the PICTURE clause, the data item is aligned on a halfword boundary (2 bytes).
     b. For the area S9(5) through S9(31) in the PICTURE clause, the data item is aligned on a fullword boundary (multiple of 4).
   - For a data item with USAGE COMPUTATIONAL-1, the item is aligned on a fullword boundary.
     For a data item with USAGE COMPUTATIONAL-2, the item is aligned on a doubleword boundary (8 bytes).

     Notes:
     - For a data item with USAGE DISPLAY, or COMPUTATIONAL-3 or PACKED-DECIMAL, the SYNCHRONIZED clause is treated as a comment, as no alignment is necessary in these cases.
     - For a data item with USAGE INDEX, the SYNCHRONIZED clause is also treated as a comment, as alignment in these cases is always performed on word boundaries.

For a summary of the details that have to be observed on the alignment of data items, refer to table 16:

USAGE	SYNC permitted	Alignment without SYNC	Alignment with SYNC
BINARY, COMP, COMP-5	yes	B	HW[1] W[2]
COMP-1	yes	B	W
COMP-2	yes	B	DW
COMP-3, PACKED-DECIMAL	yes[3]	B	B
DISPLAY	yes[3]	B	B
INDEX	yes[3]	W	W
NATIONAL	no	B	-
OBJECT REFERENCE	no	DW	-
POINTER	no	W	-
PROGRAM-POINTER	no	W	-

Table 16: Data item alignment

B	Alignment on byte boundary	W	Alignment on word boundary
HW	Alignment on half-word boundary	DW	Alignment on double word boundary

[1)] 1-4 digits
[2)] >= 5 digits
[3)]Permitted but has no influence on alignment

6. When the SYNCHRONIZED clause is specified within a table (described by the OCCURS clause), each table element will be aligned. (This process is described under "Alignment by insertion of slack bytes".)
7. When specifying the SYNCHRONIZED clause in conjunction with a REDEFINES clause, the programmer must ensure that the element being redefined is aligned (see Example 7-28).
8. The SYNCHRONIZED clause does not alter the length of an elementary data item. Each unused internal memory location (slack bytes) is included in the size of the group to which the elementary item is subordinate, and must be allowed for an internal memory allocation if the group item was the object of a REDEFINES clause (see Example 7-29).
9. All record descriptions (01-level entries) in all sections of the Data Division begin at doubleword boundaries.
10. When blocking records that contain elementary items with the SYNCHRONIZED clause specified, the user must add the necessary slack bytes to ensure proper alignment after the first record within the block. (This process is described under "Alignment by insertion of slack bytes").
11. For the purpose of aligning elementary items with USAGE COMPUTATIONAL, COMPUTATIONAL-5, BINARY, COMPUTATIONAL-1, COMPUTATIONAL-2, specified in the LINKAGE section, all 01-level elementary items

are assumed to be aligned on doubleword boundaries. Consequently, the user must ensure that these operands are appropriately aligned in the USING phrase when he writes a CALL statement.

Example 7-28

In the following example, A has to be aligned on a fullword boundary:

```
02  A  PICTURE X(4).
02  B  REDEFINES A PICTURE S9(9) USAGE BINARY SYNC.
```

Example 7-29

```
01 RECORD.
   02  A.
       03  G PICTURE X(5).
       03  H PICTURE S9(9) SYNC USAGE BINARY.
   02  B   REDEFINES A.
       03  I PICTURE X(12).
```

Here, elementary item G occupies 5 bytes, and elementary item H occupies 4 bytes.

The SYNCHRONIZED and USAGE clauses indicate that the elementary data item H is aligned on fullword boundary; elementary item H is therefore preceded by 3 slack bytes. As data item A as a whole occupies 12 bytes, the subject of the REDEFINES clause (data item B) must also occupy 12 bytes.

7.3.3.16 TYPE clause

Function

The TYPE clause defines that the data description entry is defined by a type description entry.

Format

```
TYPE TO type-name-1
```

Syntax rules

1. Neither a subordinate data description entry nor a condition name may follow a data description entry with TYPE clause.
2. The current data description entry may not be renamed as a whole or in part using the RENAMES clause.
3. The current data description entry may not be redefined either implicitly or explicitly either as a whole or in part.
4. Higher-ranking group items may not contain a SIGN clause, a USAGE clause or GROUP-USAGE clause.
5. If type-name-1 contains the STRONG phrase, the current data description entry must have either the level number 01 or must be subordinate to a type description entry with the specification STRONG.
6. If the TYPE clause is specified in a data description entry with the level number 77, type-name-1 must describe an elementary data item.
7. A data description entry may contain the following clauses in addition to the TYPE clause: BASED, EXTERNAL, GLOBAL, OCCURS and TYPEDEF clauses.

General rules

1. The TYPE clause defines that the current data description entry is determined by type-name-1. The TYPE clause acts as if the data description entry of type-name-1 applies at the place where the TYPE clause is located with the exception of the level number, the name and the GLOBAL and TYPEDEF clauses which are specified intype-name-1.
2. If type-name-1 defines a group item, the following applies:
   a. The current data description entry is a group item whose subordinate elements have the same name, description and hierarchy as the subordinate elements of type-name-1.
   b. The level numbers of the data description entries which are subordinate to the current data description entry are adapted if necessary in order to retain the hierarchy of type-name-1.
   c. Consequently level numbers in the resultant hierarchy can exceed 49.
   d. The current data description entry is aligned to doubleword boundary.

7.3.3.17 TYPEDEF clause

Function

The TYPEDEF clause defines that the data description entry is a type description entry.

Format

```
01 type-name-1 IS TYPEDEF [STRONG]
```

Syntax rules

1. The TYPEDEF clause may only be used in a data description entry with level number 01. The TYPEDEF clause must be the first clause.
2. Neither the current data description entry nor the description of a subordinate data item may be a data item of type-name-1 either directly or indirectly.
3. If the current data description entry is a data item, the STRONG phrase is not permitted.
4. The phrases TYPEDEF and REDEFINES may not be used together.
5. A type description entry may not contain an OCCURS clause.
6. Data types which are subordinate to a strongly typed type description entry must them selves be strongly typed or elementary.

General rules

1. If the TYPEDEF clause is specified, the data description entry is a type description entry. Subordinate data description entries and condition names are part of the type description entry. The data names of these descriptions can only be referenced as subordinate descriptions of group items which use the type name. If more than one group item references the type name, qualification with the name of the group is required. If there is no group with this characteristic, every reference to a data name which is subordinate to the current type description entry points to other data description entries, provided some exist, or is an invalid reference.
2. No storage is assigned to a type declaration entry.
3. The GLOBAL clause refers to the scope of the type name. All other clauses of the data description entry and subordinate data description entries refer to data which is defined with the aid of the type description entry.
4. Further details and restrictions are described in the section "Data types" .

7.3.4 USAGE clause

Function

The USAGE clause specifies the format in which an elementary item is represented in the computer's internal storage.

Format

```
[USAGE IS] {  BINARY
            | COMPUTATIONAL
            | COMP
            | COMPUTATIONAL-1
            | COMP-1
            | COMPUTATIONAL-2
            | COMP-2
            | COMPUTATIONAL-3
            | COMP-3
            | COMPUTATIONAL-4
            | COMP-4
            | COMPUTATIONAL-5
            | COMP-5
            | DISPLAY
            | INDEX
            | NATIONAL
            | OBJECT REFERENCE [  interface-name-1
                                 | [FACTORY OF] ACTIVE-CLASS
                                 | [FACTORY OF] class-name-1 [ONLY]
                               ]
            | PACKED-DECIMAL
            | POINTER [TO type-name-1]
            | PROGRAM-POINTER
          }
```

Syntax rules

1. COMP is the abbreviation for COMPUTATIONAL.
   COMP-1 is the abbreviation for COMPUTATIONAL-1.
   COMP-2 is the abbreviation for COMPUTATIONAL-2.
   COMP-3 is the abbreviation for COMPUTATIONAL-3.
   COMP-5 is the abbreviation for COMPUTATIONAL-5.
2. If the USAGE clause is not specified for an elementary item, or for a group item, then:
   a. USAGE NATIONAL is assumed if the explicit or implicit PICTURE character-string contains the character N
   b. USAGE DISPLAY is assumed in all other cases.
3. For a description of the various categories of data see chapter "Introduction to the COBOL language".

General rules

1. The USAGE clause may be written at any data description level. If it is specified at group level, it applies to each elementary data item of the group.

2. The USAGE of an elementary item must not conflict with the USAGE of the group item to which the elementary item belongs.
3. An elementary item described with USAGE BINARY, COMPUTATIONAL, COMPUTATIONAL-1, COMPUTATIONAL-2, COMPUTATIONAL-3, COMPUTATIONAL-5 or PACKED-DECIMAL represents a value for use in arithmetic operations and must therefore be numeric. If any of these phrases is specified for a group item, it refers only to the elementary items of that group; the group item itself must not be used in arithmetic operations.
4. The USAGE clause does not affect the use of the data item. Although the specifications for some statements in the procedure division may restrict the USAGE clause of the operands referred to.
5. The internal representation of the numeric data items is shown in table 17 in section "COMPUTATIONAL-3 phrase or PACKED-DECIMAL phrase".

7.3.4.1 DISPLAY phrase

Syntax rules

1. The type of elementary item for which the DISPLAY phrase is written is defined by the character-string in the PICTURE clause.

General rules

1. The DISPLAY phrase specifies that the data item is to be stored in EBCDIC data format; that is, each character position is represented by one byte.
2. External decimal data items are internally represented as follows:

   Each digit of a number is represented by a single byte. The four high-order bits of each byte are the zone portion. The zone portion of the low-order or high-order byte (depending on the SIGN clause) represents the sign of the number, assuming that a sign exists. The four low-order bits contain the value of the digit.

   The maximum length of an external decimal item is 31 digits.
3. External floating-point items consist of a mantissa, which represents the decimal part of the number, and an exponent with the base 10

   The value of an external floating-point item is calculated by multiplying the mantissa by 10 to the power of exponent.
   The magnitude of a number represented by a floating-point item must be greater than $5.4 * 10^{-79}$ and must not exceed $7.2 * 10^{79}$ and must not exceed $7.2 * 10^{75}$.

   An external floating-point item, when used as a numeric operand, is checked at object time and is converted into an internal floating-point item. It is used in this form in arithmetic operations (see notes on COMPUTATIONAL-1 and COMPUTATIONAL-2).

Example 7-30

Data formats for USAGE IS DISPLAY

Data category	Value	PICTURE description	Internal representation*)									
alphabetic	ABCD	AAAA	C1	C2	C3	C4						
alphanumeric	A1B2	XXXX	C1	F1	C2	F2						
alphanumeric edited	123AB	XXBXXX	F1	F2	40	F3	C1	C2				
numeric edited	54321	99,999	F5	F4	6B	F3	F2	F1				
numeric external decimal												
	+1234	9999	F1	F2	F3	F4						
	+6879	S9999	F6	F8	F7	C9						
	-6879	S9999	F6	F8	F7	D9						
external floating-point	6879	+99.99E-99	4E	F6	F8	4B	F7	F9	C5	40	F0	F2

	.6879	+99.99E-99	4E	F6	F8	4B	F7	F9	C5	60	F0	F2

*) Each box represents one byte.

7.3.4.2 NATIONAL phrase

Syntax rules

1. An elementary item which contains a USAGE NATIONAL clause or is subordinated to a group item with a USAGE NATIONAL clause must describe a national item with its PICTURE character-string.
2. The GROUP-USAGE clause may not be used together with the NATIONAL phrase.

General rules

1. The NATIONAL phrase specifies that the data item is to be stored in UTF-16 data format; that is, each character position is represented by 2 bytes.

> All national characters are of the same length for COBOL. "Surrogate Pairs" are therefore regarded as 2 characters.

Example 7-31

Data formats for USAGE IS NATIONAL

Data category	Value	PICTURE description	Internal representation*)					
national	ABC	NNN	00	41	00	42	00	43

*) Each box represents 1 byte.

7.3.4.3 BINARY phrase or COMPUTATIONAL phrase or COMPUTATIONAL-5 phrase

Syntax rules

1. These phrases specify binary data items.
2. The PICTURE clause of a binary data item must contain no other characters but 9s, the operational sign S, the assumed decimal point V, and one or more Ps (see section "PICTURE clause").
3. The data items are stored in a halfword, fullword, or doubleword, and are aligned only if the SYNCHRONIZED clause was specified.

General rules

1. If a data item described with USAGE IS BINARY is used as a receiving data item, a check is made to determine whether the value to be transferred to this data item exceeds the maximum possible value indicated by the PICTURE character-string in the PICTURE clause. If this is the case, the value is made to conform by truncation.If a receiving data item is described with USAGE IS COMPUTATIONAL or COMPUTATIONAL-5, this check and any subsequent truncation which may be required are not performed.
2. The PICTURE string specified in the PICTURE clause is also evaluated if USAGE IS COMPUTATIONAL or COMPUTATIONAL-5 is specified in a DISPLAY statement. This evaluation serves to check the overflow condition.
3. The storage requirements for binary items vary depending on the number of decimal digits specified in the PICTURE clause, as follows:

Decimal digits in the PICTURE clause	Bytes required in computer storage	Alignment
1-4	2	Halfword
5-9	4	Word
10-18	8	Word
19-31	16	Word

4. The leftmost bit of a binary data item is the operational sign. The remaining bits represent the value.

For examples of the BINARY, COMPUTATIONAL or COMPUTATIONAL-5 phrases see table 17, "Internal representation of internal data items" in section "COMPUTATIONAL-3 phrase or PACKED-DECIMAL phrase".

7.3.4.4 COMPUTATIONAL-1 phrase

Syntax rules

1. This phrase specifies internal floating-point items, which are equivalent to external floating-point items in terms of capacity and application (see “Classes and categories of data and literals” in section "Concept of computer-independent data description").
2. For a COMPUTATIONAL-1 data item, the PICTURE clause is prohibited.
3. The COMPUTATIONAL-1 phrase indicates that a data item is stored in single-precision floating-point format.
4. A COMPUTATIONAL-1 data item has a length of 4 bytes and is aligned on a fullword boundary if the SYNCHRONIZED clause is specified.

General rules

1. A COMPUTATIONAL-1 data item is represented in storage as follows:

   Here, S is the sign of the mantissa.
   Characteristic = exponent + 32
2. An internal single-precision floating-point item permits representation with a precision of seven decimal digits.
3. The following applies to the value that may be represented in a COMPUTATIONAL-1 data item:

   value = 0 or the absolute value of it may range from $5.4 * 10^{-79}$ to $7.2 * 10^{75}$.

For examples of the COMPUTATIONAL-1 phrase see table 17 in section "COMPUTATIONAL-3 phrase or PACKED-DECIMAL phrase".

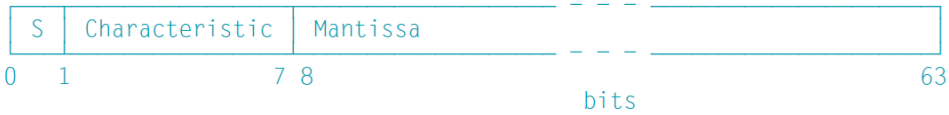
7.3.4.5 COMPUTATIONAL-2 phrase

Syntax rules

1. This phrase specifies internal floating-point items, which are equivalent to external floating-point items in terms of capacity and application (see “Classes and categories of data and literals” in section "Concept of computer-independent data description").
2. For a COMPUTATIONAL-2 item, the PICTURE clause is prohibited.
3. The COMPUTATIONAL-2 phrase indicates that a data item is to be stored in double precision floating-point format.
4. A COMPUTATIONAL-2 item has a length of 8 bytes and is aligned on a doubleword boundary if the SYNCHRONIZED clause is specified.

General rules

1. A COMPUTATIONAL-2 data item is represented in storage as follows:



   Here, S is the sign of the mantissa.

   Characteristic = exponent + 64
2. A double-precision internal floating-point item permits representation with a precision of 16 decimal digits.
3. The following applies to the value that may be represented in a COMPUTATIONAL-2 data item:

   value = 0 or the absolute value of it may range from $5.4 * 10^{-79}$ to $7.2 * 10^{75}$.

For examples of the COMPUTATIONAL-2 phrase see table 17 in section "COMPUTATIONAL-3 phrase or PACKED-DECIMAL phrase".

7.3.4.6 COMPUTATIONAL-3 phrase or PACKED-DECIMAL phrase

Syntax rules

1. The COMPUTATIONAL-3 and PACKED-DECIMAL phrases are identical in meaning.
2. The phrases indicate that the data item is stored in internal decimal format (i.e. in packed form).
3. The PICTURE clause of a COMPUTATIONAL-3 or PACKED-DECIMAL item may contain no characters other than 9s, the operation sign S, the assumed decimal point V, and one or more Ps (see section "PICTURE clause").

General rule

1. Internal decimal data items are represented by 2 digits per byte; the sign is contained in the four low-order bits of the low-value byte.

   For internal decimal data items whose PICTURE clause contains no S, the represen-tation of the absolute value corresponds to the number.

For examples of the COMPUTATIONAL-3 or PACKED-DECIMAL table 17.

Format	**PICTURE clause**	**USAGE and SIGN phrase**	**Value in** external representation	**Value in internal representation** [4]	**Bytes required**	**Conversion for arithmetic operation** Alignment	**Alignmen if SYNC is specified**
External decimal (zoned)	9999	DISPLAY	1234	F1F2F3F4	1 byte /digit	Yes, in order to conform to format of other operands or COMP-3 or PACKED-DECIMAL	
	S9999		+1234	F1F2F3C4[1)2)]			
	S9999		-1234	F1F2F3D4[1)2)]			
	S9999	DISPLAY SIGN TRAILING	1234+	F1F2F3C4			
			1234-	F1F2F3D4			
	S9999	DISPLAY SIGN TRAILING SEPARATE	1234+	F1F2F3F44E	+ 1 byte for sign		
			1234-	F1F2F3F460			
	S9999	DISPLAY SIGN LEADING	+1234	C1F2F3F4			
			-1234	D1F2F3F4			
	S9999	DISPLAY SIGN LEADING SEPARATE	+1234	4EF1F2F3F4	+ 1 byte for sign		
			-1234	60F1F2F3F4			

Internal decimal (packed)	9999	COMP-3 or PACKED-DECIMAL	+1234	01234F[2)]	2 digits per byte, except for low-order byte which contains a digit and the sign	No, except when other operand is binary and conversion to binary would be more advantageous.	None
	9999		-1234	01234F[2)]			
	S9999		+1234	01234C[2)]			
	S9999		-1234	01234D[2)]			
Binary	S9999	BINARY or COMP or COMP-5	+1234	04D2	2 bytes for 1-4 digits	No, except when used in mixed-form computations to maintain common formats, or if COMP-3 or PACKED-DECIMAL would be more advantageous.	Halfword
					4 bytes for 5-9 digits		Word
					8 bytes for 10-18digits		Word
					16 bytes for 19-31digits		Word
	S9999		-1234	FB2E	2		Halfword[3]
Ext. floating point	+99.99E-99	DISPLAY	+12.34E+2	4EF1F26BF3F4C540F0F2	1 byte per character	Yes. To internal floating-point	None
Int. floating point	None allowed	COMP-1	+12.34E+2	434D2000	4	No	Word
	None allowed	COMP-2	-12.34E-2	C01F972474538EF3	8	No	Doublewo

Table 17: Internal representation of internal data items

[1)] One byte per digit, except for the low-order byte which contains the sign in the first halfbyte and the last digit in the second halfbyte.

[2)] Mode of sign representation:

F = non-printable plus sign (treated as an absolute value)
C = internal equivalent of plus sign
D = internal equivalent of minus sign.

[3)] See rules for binary data items.

[4)] Each character (letter/digit) represents a half byte.

7.3.4.7 INDEX phrase

An elementary item described with USAGE IS INDEX is called an index data item. This is a data item (not necessarily associated with any table) which may be used to save values of index-names for future reference. An index data item is assigned the value of an index by the SET statement. The value of an index data item is not an occurrence number.

General rules

1. The USAGE clause with INDEX phrase may be written at any level. If a group item is described with USAGE IS INDEX, the elementary items in the group are all index data items; the group itself is not an index data item.
2. An index data item can be referenced directly only in a SEARCH statement, in a SET statement, in a relation condition, in the USING phrase of the Procedure Division header, or in the USING phrase of a CALL statement.
3. An index data item cannot be a conditional variable.
4. An index data item may be part of a group which is referenced in a MOVE statement or an input/output statement. When such statements are executed, however, the contents of the index data item are not converted.
5. SYNCHRONIZED, PICTURE or VALUE clauses cannot be used to describe group items or elementary items described with USAGE IS INDEX.
   However, the compiler allows the SYNCHRONIZED clause to be used with the USAGE IS INDEX clause.

Example 7-32

```
02  ALPHA PICTURE X(9) OCCURS 5 INDEXED BY A-NAME.
...
77  A-INDEX USAGE IS INDEX.
...
    SET A-NAME TO 3.
    ...
    SET A-INDEX TO A-NAME.
```

Here the index data item A-INDEX is set to the current value of the index-name A-NAME, i.e. the occurrence number (3) minus 1, multiplied by the length of the entry (9) = 18.

7.3.4.8 OBJECT REFERENCE phrase

Syntax rules

1. The USAGE OBJECT REFERENCE clause should only be specified on level 01 or within a type description entry with the attribute STRONG. Deviations are permitted as an extension.
2. The USAGE OBJECT REFERENCE clause may not be specified on the group level in data definitions.
3. The USAGE OBJECT REFERENCE clause may not be specified in the FILE SECTION.
4. The ACTIVE-CLASS phrase may be specified only in a factory definition, an object definition, or a method definition.
5. The USAGE OBJECT REFERENCE clause may not be specified in structures in which the EXTERNAL or DYNAMIC clauses are used on level 01.
6. No VALUE clauses may be specified for data items defined with a USAGE OBJECT REFERENCE clause.
7. Data items to which a USAGE OBJECT REFERENCE clause applies may have neither a REDEFINES clause, nor be referenced directly from any such clause. In addition, these data items may not be indirectly contained in any redefined or redefining structure.
8. Data items and structures to which a USAGE OBJECT REFERENCE clause applies may not be renamed (with a RENAMES clause).
9. For groups that have defined data items with a USAGE OBJECT REFERENCE clause, no VALUE clause may be specified on the group level.
10. No condition name (level number 88) may be specified for data items with a USAGE OBJECT REFERENCE clause.
11. The data item specified in the KEY phrase of an OCCURS clause may not be defined with a USAGE OBJECT REFERENCE clause.

General rules

A data item described with a USAGE OBJECT REFERENCE clause is an object reference. The following rules apply:

1. If none of the optional phrases is specified, this data item is called a universal object reference. Its content may be a reference to any object.
2. interface-name and class-name must be the names of the containing interface and class definitions, respectively, or must be listed accordingly in the REPOSITORY paragraph.
3. If interface-name-1 is specified, the object referenced by the data item be an object whose interface conforms with interface-1.
4. If class-name-1 is specified, the object referenced by this data item must be an object of the class identified by class-name-1 or of a subclass thereof, subject to the following rules:
    a. If the ONLY phrase is not specified, but the FACTORY phrase is used, the object referenced by this data item must be the **factory object** of the associated class or a subclass thereof. If FACTORY is not specified, the object referenced by this data item must be an **object** of the specified class or a subclass thereof.
    b. If the ONLY phrase is specified, then the same rules apply as given under a), except that no reference to a subclass is possible.

5. If ACTIVE-CLASS is specified, the object referenced by this data item must be an object of the class of the object that was used to invoke the method in which the object reference is defined. If FACTORY is specified, then the object reference refers to the **factory object** of that class; otherwise, to an **object** of that class.

7.3.4.9 POINTER phrase

The POINTER phrase in the USAGE clause is used to define data of the category “data pointer”.

Syntax rules

1. The USAGE POINTER clause should only be specified on level 01 or within a type description entry with the attribute STRONG. Deviations from this rule are permitted as extensions.
2. The USAGE POINTER clause must not be specified at group level in definitions.
3. The USAGE POINTER clause must not be specified in the FILE SECTION .
4. The USAGE POINTER clause must not be specified in structures in which the EXTERNAL or DYNAMIC clauses are used on level 01.
5. In the case of data items defined with a USAGE POINTER clause, no VALUE clause may be specified.
6. Data items for which there is a USAGE POINTER clause may not possess a REDEFINES clause and may also not be directly referenced by such clauses. In addition, such data items may also not be present indirectly in any redefined or redefining structure.
7. Data items and structures for which there is a USAGE POINTER clause may not be named using a different name (RENAMES clause).
8. No VALUE clause may be specified at group level in the case of groups that contain data items with a USAGE POINTER clause.
9. No condition name (level 88) may be specified for data items having a USAGE POINTER clause.
10. The data item specified in the KEY phrase of an OCCURS clause may not be defined using a USAGE POINTER clause.
11. If type-name-1 is specified, the data description entry must also contain a TYPEDEF clause.

General rule

1. A data item described using the USAGE POINTER clause has the category “data pointer” and contains the address of a data item. The assumed start value is equal to NULL. If type-name-1 is specified, a type-specific pointer is involved. A type-specific pointer has only the predefined value NULL as its content or the address of a typed data item of the type type-name-1.

7.3.4.10 PROGRAM-POINTER phrase

The PROGRAM-POINTER phrase in the USAGE clause is used to define data of the category “program pointer”.

Syntax rules

1. The USAGE PROGRAM-POINTER clause should only be specified on level 01. Deviations from this rule are permitted as extensions.
2. The USAGE PROGRAM-POINTER clause must not be specified at group level in definitions.
3. The USAGE PROGRAM-POINTER clause must not be specified in the FILE SECTION.
4. The USAGE PROGRAM-POINTER clause must not be specified in structures in which the EXTERNAL or DYNAMIC clauses are used on level 01.
5. In the case of data items defined with a USAGE PROGRAM-POINTER clause, no VALUE clause may be specified.
6. Data items for which there is a USAGE PROGRAM-POINTER clause may not possess a REDEFINES clause and may also not be directly referenced by such clauses. In addition, such data items may also not be present indirectly in any redefined or redefining structure.
7. Data items and structures for which there is a USAGE PROGRAM-POINTER clause may not be named using a different name (RENAMES clause).
8. No VALUE clause may be specified at group level in the case of groups that contain data items with a USAGE PROGRAM-POINTER clause.
9. No condition name (level 88) may be specified for data items having a USAGE PROGRAM-POINTER clause.
10. The data item specified in the KEY phrase of an OCCURS clause may not be defined using a USAGE PROGRAM-POINTER clause.

General rule

1. A data item described using the USAGE PROGRAM-POINTER clause has the category “program pointer” and contains the address of a data item. The assumed start value is equal to NULL.

7.3.4.11 VALUE clause

Function

The VALUE clause defines the initial value of a data item in the WORKING-STORAGE SECTION and LOCAL-STORAGE SECTION, the value of a printable data item of the REPORT SECTION. The VALUE clauses in the WORKING-STORAGE SECTION, the LOCAL-STORAGE SECTION, in the FILE SECTION and in the LINKAGE SECTION are meaningful when an INITIALIZE statement with the BY VALUE phrase is executed.

In the case of condition names, the VALUE clause determines the value or range of values which is assigned to this condition name.

Format 1 of the VALUE clause is specified to define the initial value of a data item or the value of a printable data item or the value of the sending item for the INITIALIZE statement with the BY VALUE phrase.

Format 2 of the VALUE clause is specified to define the value or range of values associated with a condition-name. The optional addition "WHEN SET TO FALSE IS LITERAL-4" defines the value to which the related data item is set when the "SET condition-name TO FALSE" statement is executed.

Format 3 serves to initialize table elements.

Format 1

```
VALUE IS literal
```

Syntax rules

1. The literal specified can be replaced by a figurative constant.
2. A numeric literal must have a size which is within the number of positions specified by the PICTURE clause, and must not have a value which would require truncation of nonzero digits.
3. A non-numeric literal must not exceed the size specified in the PICTURE clause.
4. If the category of the item being described is alphabetic, alphanumeric, alphanumeric edited or numeric edited, the literal in the VALUE clause must be alphanumeric.The length of alphanumeric literals in the VALUE clause of an elementary item may not be greater than the length which is defined by an explicitly specified PICTURE clause.The length of alphanumeric literals in the VALUE clause of an alphanumeric group may not be greater than the length of the group.
5. A signed numeric literal must be associated with a PICTURE clause which provides a signed numeric character-string.
6. If the category of the item being described is numeric, the literal in the VALUE clause must be a numeric literal. The value of the literal must be in the range described by the PICTURE clause. All digits at positions which correspond to the PICTURE symbol P must be 0.
7. If the VALUE clause is used in an entry at the group level, the literal must be a figurative constant or a non-numeric literal. The VALUE clause cannot be stated at the subordinate levels within the group.
8. The VALUE clause must not be specified for a group item containing subordinate items with descriptions that include JUSTIFIED, SYNCHRONIZED or USAGE (other than USAGE DISPLAY or USAGE NATIONAL).
9. No VALUE clause may be specified for a strongly typed group item.
10. The VALUE clause is prohibited for external floating-point data items.

11. If the category of the data item is national, all literals in the VALUE clause must be national.
    The length of national literals in the VALUE clause of an elementary item may not be greater than the length which is defined by an explicitly specified PICTURE clause.
    The length of national literals in the VALUE clause of an alphanumeric group may not be greater than the length of the group.

General rules

1. If a VALUE clause is specified in a data description entry of a data item which is associated with a variable-length data item, the initialization of the data item behaves as if the value of the data item referenced by the DEPENDING ON phrase in the OCCURS clause specified for the variable-length data item had the maximum possible value. A data item is associated with a variable-length data item in any of the following cases:
   a. It is a group data item which contains a variable-length data item.
   b. It is a variable-length data item.
   c. It is a data item that is subordinate to a variable-length data item.
2. If the VALUE clause is used in an entry at the group level, the group area will be initialized without consideration for the individual elementary or group items contained within the group.
3. The literal is aligned in the data item as if the data item had been described as alphanumeric. The JUSTIFIED clause has no influence here and no editing takes place.
4. Initialization of a data item is not affected by any BLANK WHEN ZERO or JUSTIFIED clause that may be specified.
5. A VALUE clause specified in a data description entry that contains an OCCURS clause or in an entry that is subordinate to a data description entry that contains an OCCURS clause causes every occurrence of the associated data item to be assigned the specified initial value.
6. In the WORKING-STORAGE and LOCAL-STORAGE SECTION, the VALUE clause may be used to specify the initial value of any data item; in this case, the clause causes the item to be initialized to the specified value at the start of program execution. If the VALUE clause is not specified in the description of a data item, the initial value of that item is undefined.
7. A data item of the class “object“ or “pointer” cannot have a VALUE clause; it is always initialized with NULL (undefined).

Example 7-33

```
77 FELD PICTURE IS AA VALUE IS "AA"
```

Here the value of FIELD is initialized to AA.

Format 2

```
{VALUE | VALUES} {IS | ARE} {literal-1 [{THRU | THROUGH} literal-2]} ...
                            [WHEN SET TO FALSE IS literal-4]
```

Syntax rules

1. A format 2 VALUE clause may be used only in connection with condition-names (level-number 88).
2. Level number 88 applies to declarations of condition-names which are associated with a conditional variable; these declarations are called condition-name declarations. A conditional variable is a data item which is

followed by one or more condition-name declarations. A condition-name defines a value or a range of values which is to be queried as the contents of the conditional variable at the execution time of the program. A condition-name can then be "true" or "false" during program execution. A condition-name is not a data item and requires no storage space (see "Condition-name condition").

Condition names may not follow a data item for which an ANY LENGTH clause is specified.
Condition names may not follow a type definition with the STRONG phrase or a group item which is subordinate to a type definition with the STRONG phrase.

3. The specified literals may be replaced by figurative constants.
4. All numeric literals must have a length which is within the number of positions specified by the PICTURE clause for the related elementary item (conditional variable), and must not have a value which would require truncation of non-zero digits.
5. Non-numeric literals must not exceed the size specified in the PICTURE clause for the related elementary item (conditional variable).
6. If the category of the item being described is alphabetic, alphanumeric, alphanumeric edited or numeric edited, the literal in the VALUE clause must be alphanumeric.The length of alphanumeric literals in the VALUE clause of an elementary item may not be greater than the length which is defined by an explicitly specified PICTURE clause.
7. If the category of the item being described is numeric, all literals in the VALUE clause must be numeric literals. The value of the literal must be in the range of values described by the PICTURE clause. All digits at positions which correspond to the PICTURE symbol P must be 0.
8. A signed numeric literal must be associated with a PICTURE clause which provides a signed numeric character-string.
9. When the THRU/THROUGH phrase is used, the literal preceding THRU/THROUGH must be less than the literal which follows it.
10. The THRU/THROUGH phrase assigns a range of values to the specified condition-name.
11. literal-4 may not be equal to any literal-1.
12. The following must apply to each literal-1, literal-2 pair: literal-4 must be less than literal-1 or greater than literal-2.
13. If the category of the data item is national, all literals in the VALUE clause must be national.
The length of national literals in the VALUE clause of an elementary item may not be greater than the length which is defined by an explicitly specified PICTURE clause.

General rules

1. The VALUE clause is prohibited for external floating-point data items and for data of the class object or pointer.
2. The VALUE clause must not be specified for items whose size, whether explicitly or implicitly, is variable.
3. The VALUE clause must not conflict with other clauses in the data description of an item or in the data description within the hierarchy of an item. The following rules are applicable:
4. The value is aligned in the data item as if the data item had been described as alphanumeric.
5. Format 2 of the VALUE clause is only allowed in the FILE, WORKING-STORAGE, LOCAL-STORAGE and LINKAGE SECTION. It must not be specified in the REPORT SECTION.
6. The specification of FALSE in the VALUE clause is relevant only when the condition-name is used in a SET-TO-FALSE statement.

Example 7-34

```
02 CITIES PICTURE 9.
        88  BERLIN VALUE 1.
        88  HAMBURG VALUE 2.
        88  MUNICH VALUE 3.
        88  COLOGNE VALUE 4.
```

Here, CITIES is the conditional variable, and BERLIN, HAMBURG, MUNICH, and COLOGNE are the condition-names. If a statement IF MUNICH GO TO TEST-C were written in the Procedure Division, then the value of the conditional variable CITIES would be compared to the value 3; this statement would be equivalent to the statement IF CITIES IS EQUAL TO 3 GO TO TEST-C.

Example 7-35

```
02 AGE PICTURE 99.
    88 TWENTIES VALUE 20 THRU 29.
    88 THIRTIES VALUE 30 THRU 39.
```

If the statement IF TWENTIES... were to be written in the Procedure Division, the value of the conditional variable AGE would be compared to the values 20, 21, ... and 29. This statement would be equivalent to the statement IF AGE NOT LESS THAN 20 AND NOT GREATER THAN 29...

Example 7-36

```
02 NAME-OF-DAY PICTURE X(3).
88  BEGINNING-WEEK VALUE "MON" "TUE" "WED".
88  END-OF-WEEK    VALUE "THRU" "FRI".
88  WEEKEND       VALUE "SAT" "SUN".
```

If the statement IF BEGINNING-OF-WEEK... were to be written in the Procedure Division, the conditional variable NAME-OF-DAY would be compared with "MON", "TUE" and "WED". This statement would be equivalent to IF NAME-OF-DAY IS EQUAL TO "MON" OR "TUE" OR "WED"...

Format 3

```
{{VALUE | VALUES} [FROM ({subscript-1}...)] [IS | ARE]
       {literal-2}... [REPEATED {integer-1 TIMES | TO END}]} ...
```

Syntax rules

1. All numeric literals in a VALUE clause of an item must have a value which is within the range of values indicated by the associated PICTURE clause, and must not have a value which would require truncation of non-zero digits.
2. Non-numeric literals in a VALUE clause of an item must not exceed the size indicated by the associated PICTURE clause.
3. If the category of the item being described is alphabetic, alphanumeric, alphanumeric edited or numeric edited, all literals in the VALUE clause must be alphanumeric literals.The length of alphanumeric literals in the VALUE clause of an elementary item may not be greater than the length which is defined by an explicitly specified

PICTURE clause. The length of alphanumeric literals in the VALUE clause of an alphanumeric group may not be greater than the length of the group.

4. If the VALUE clause is used in an entry at the group level, the literal must be a figurative constant or a non-numeric literal, and the group area is initialized without consideration for the individual elementary or group items contained within this group. The VALUE clause must not be stated at the subordinate levels within this group.
5. The VALUE clause must not be specified for a group item containing data items with descriptions including JUSTIFIED, SYNCHRONIZED or USAGE (other than USAGE DISPLAY or USAGE NATIONAL).
6. When format 3 is specified, the data description entry must contain an OCCURS clause or be subordinate to a data description entry that contains an OCCURS clause.
7. Subscript-1 must be a numeric literal that is an integer. If all subscripts have the value 1, no subscripts need be specified; otherwise, all subscripts required to reference an individual element in a table must be specified.
8. The number of table elements to be initialized is determined as follows:
   a. If integer-1 is not specified, it is the number of repetitions of literal-2.
   b. If integer-1 is specified, it is the number of repetitions of literal-2 times integer-1.

   The number of table elements to be initialized must not exceed the maximum number of occurrences in the table from the point of reference to the end of the table.
9. If multiple format 3 VALUE clauses are specified in an entry:
   a. The TO END phrase may be specified only once.
   b. A specified table element may be referenced only once.
10. If the category of the data item is national, all literals in the VALUE clause must be national.
    The length of national literals in the VALUE clause of an elementary item may not be greater than the length which is defined by an explicitly specified PICTURE clause.
    The length of national literals in the VALUE clause of an alphanumeric group may not be greater than the length of the group.

General rules

1. All formats of the VALUE clause can be used in one table.
2. Within the same data description entry, if more than one VALUE clause references the same table element, the value defined by the last specified VALUE clause in the data description entry is assigned to the table element.
3. A format 3 VALUE clause initializes a table element to the value of literal-2. The table element initialized is identified by subscript-1. Consecutive table elements are initialized, in turn, to the successive occurrences of the value of literal-2. Consecutive table elements are referenced by augmenting by 1 the subscript that represents the least inclusive dimension of the table. When any reference to a subscript, prior to augmenting it, is equal to the maximum number of occurrences specified by its corresponding OCCURS clause, that subscript is set to 1 and the subscript for the next most inclusive dimension of the table is augmented by 1.
4. If the REPEATED phrase is specified, all occurrences of literal-2 are reused, in the order specified.
   If the TO END phrase is specified, this reuse occurs until the end of the table is reached. If the integer-1 TIMES phrase is specified, the occurrences of literal-2 are reused, in the order specified, integer-1 times.
   If the REPEATED phrase is not specified, the occurrences of literal-2 are used, in the order specified, only once.
5. If a VALUE clause is specified in a data description entry of a data item which is associated with a variable-occurrence data item, the initialization of the data item behaves as if the value of the data item referenced by

the DEPENDING ON phrase in the OCCURS clause specified for the variable-occurrence data item is set to the maximum number of occurrences as specified by that OCCURS clause. A data item is associated with a variable-occurrence data item in any of the following cases:

a. It is a group data item which contains a variable-occurrence data item.

b. It is a variable-occurrence data item.

c. It is a data item that is subordinate to a variable-occurrence data item.

6. The VALUE clause must not conflict with other clauses in the data description of the item or in the data description within the hierarchy of the item. The following rules apply:

   If the category of the item is “numeric”, all literals in the VALUE clause must be numeric. The literal is aligned in the data item according to the standard alignment rules.

   The literal is aligned in the data item as if the data item had been described as alphanumeric. The JUSTIFIED clause has no influence here and no editing takes place.

   A data item is initialized regardless of whether a BLANK WHEN ZERO or JUSTIFIED clause was specified.

Example 7-37

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. TAB.
  ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
  SPECIAL-NAMES.
      TERMINAL IS T.
  DATA DIVISION.
 *****
  WORKING-STORAGE SECTION.
  01 FIELD1.
     02 A OCCURS 20.
       03 B OCCURS 4.
         49 PIC X(01)
            VALUE FROM (5 2) IS "1" "2" "3"
            REPEATED 4.
 *
  01 FIELD2.
     02 Z PIC 99.
     02 A OCCURS 1 TO 78 DEPENDING ON Z.
        49 PIC X VALUE "x".
 *
  01 FIELD3.
     02 A OCCURS 20
          VALUE FROM (1) IS "ab" "c"
          REPEATED 10 TIMES.
        03 B OCCURS 4.
           49 PIC X.
  PROCEDURE DIVISION.
  MAIN SECTION.
  P1.
      MOVE 78 TO Z.
      DISPLAY FIELD1 UPON T.
      DISPLAY FIELD2 UPON T.
      DISPLAY FIELD3 UPON T.
      STOP RUN.
```

This results in the following field assignments:

```
FIELD1:  B(5,2) = "1"  B(6,1) = "1"  B(7,1) = "2"  B(8,1) = "3"
         B(5,3) = "2"  B(6,2) = "2"  .
         B(5,4) = "3"  B(6,3) = "3"  .
                       B(6,4) = "1"  .
```

All other table elements are not assigned

```
FIELD2   78 times "x"
FIELD3:  A(1) = "ab'BLANK"BLANK"'     A(2) = "c'BLANK"BLANK"BLANK"'
         A(3) = "ab'BLANK"BLANK"'     A(4) = "c'BLANK"BLANK"BLANK"'
         ...                          ...
         A(19) = "ab'BLANK"BLANK"'    A(20) = "c'BLANK"BLANK"BLANK"'
```

8 Procedure Division

8.1 General description

The PROCEDURE DIVISION of a program or method contains the specific instructions for solving a given data processing problem.

The PROCEDURE DIVISION in an object, factory or interface definition contains the invocable methods (for interfaces, only the method prototypes) which apply to an object, a factory object or an interface, respectively.

COBOL instructions are written in the form of statements.

A **statement** is a syntactically valid combination of words and symbols, beginning with a COBOL verb.

Example of a statement

```
MOVE A TO B
```

Several statements may be combined into a sentence, groups of sentences into paragraphs, and one or more paragraphs into a section.

Normally, a COBOL statement refers to user-defined data or procedures by means of data-names or procedure-names. References to user-defined words must be unique (see section "Qualification").

A logical subset of the program, consisting of one or more successive paragraphs or one or more successive sections of the Procedure Division, is called a procedure. A procedure-name is a word which is used for referring to a paragraph or a section; it consists of a paragraph-name (which may be qualified by a section-name) or a section-name.

There are two types of procedures in the Procedure Division:

- Declaratives, which cannot be executed within the normal sequence of statements in the Procedure Division.
- Nondeclarative procedures that contain statements for normal execution when there are no special exceptional conditions.

The execution of the source unit begins with the first PROCEDURE DIVISION statement following the declaratives. Statements are then executed in the order in which they are presented for compilation, except where the rules for a given statement indicate some other order.

If program segmentation is used, the programmer must divide the entire Procedure Division into named sections. Program segmentation is discussed in chapter "Segmentation".

8.1.1 Structure

General format

Format 1 (with sections)

```
PROCEDURE DIVISION [USING-phrase] [RETURNING data-name-2].

[ DECLARATIVES.
  { section-name SECTION.
      USE statement.
 [paragraph-name.
      [sentence]...]...}...
  END DECLARATIVES.

]

 {section-name SECTION [segment-number].
 [paragraph-name.
     [sentence]... ]... }...
```

Format 2 (without sections)

```
PROCEDURE DIVISION [USING-phrase] [RETURNING data-name-2].

[sentence]...
[paragraph-name.
     [sentence]...]...
```

where the **USING** phrase is defined as follows:

```
USING {[BY REFERENCE] {[OPTIONAL] data-name-1}... | BY VALUE {data-name-1}... }...
```

Format 3

Only applies to an object, factory or interface definition.

```
PROCEDURE DIVISION.

[{ method-definition}...]
```

General rules

1. The Procedure Division header of a program or method is followed, optionally, by the declarative portion containing declarative procedures (DECLARATIVES), which is followed, in its turn, by normal executable procedures. Each of these procedures consists of statements, sentences, paragraphs, and/or sections in a syntactically valid format.
2. For a description of the declarative subdivision, see section "DECLARATIVES".

3. If sections are used within the Procedure Division, format 1 must be applied. Otherwise, format 2 may be used.
4. A **section** consists of a section header followed either by zero, one, or more successive paragraphs. (The section header consists of a section-name, followed by the word SECTION and a period; if program segmentation is desired, a space and a segment-number followed by a period may be inserted after the word SECTION.) A section ends immediately before the next section, at the end of the Procedure Division; or in the declaratives subdivision of the Procedure Division immediately before the next section or at the keywords END DECLARATIVES.

   Multiple definitions of section-names, or of paragraph-names within a section, are not reported as errors by the compiler as long as they are not referenced.
5. A **paragraph** must contain at least a paragraph-name followed by a period and a space, followed by zero, one or more successive sentences. A paragraph ends immediately before the next paragraph or section, or at the end of the PROCEDURE DIVISION, or, in the declaratives portion of the PROCEDURE DIVISION, immediately before the next paragraph or section, or at the keywords END DECLARATIVES.
6. If one paragraph is in a section, then all of the paragraphs must be in sections.
7. A **sentence** consists of one or more statements, optionally separated by semicolons, spaces, or commas, and is terminated by a period, followed by spaces.
8. A **statement** consists of a syntactically valid combination of words and symbols, and must begin with a COBOL word.

8.2 Procedure Division header

Function

In a called program (subroutine) or method, the Procedure Division header determines the standard entry point. Optionally, data-names may be specified if data is transferred from the calling program as a parameter.

Format 1

```
PROCEDURE DIVISION [USING-phrase] [RETURNING data-name-2].
```

where the USING phrase is defined as follows:

```
USING {[BY REFERENCE] {[OPTIONAL] data-name-1}... | BY VALUE {data-name-1}... }...
```

Format 2

For object, factory or interface definitions.

```
PROCEDURE DIVISION.
```

Syntax rules

1. The USING phrase may be written only if the program is called by a CALL statement or a method by an INVOKE statement and the CALL /INVOKE statement in the calling source unit includes a USING phrase. For calls as subprograms from “programs written in other languages“, see the "CRTE" manual [2].
2. The RETURNING phrase may only be written if the program is called by a CALL statement or a method by an INVOKE statement and the CALL /INVOKE statement in the calling source unit includes a RETURNING phrase.
3. Each data-name supplied in the USING or RETURNING phrase of the Procedure Division header must be defined in the LINKAGE SECTION of the source unit containing this header and must have the level number 01 or 77.
   The data description of data-name-1 or data-name-2 must not contain a REDEFINES clause or BASED clause.
4. Each data-name-1 defined in the BY VALUE phrase may identify only one data item of the class “numeric”, “alphanumeric” or “object”.
5. The RETURNING phrase may be specified in a method, program or program prototype definition.
6. data-name-2 must not be identical with data-name-1.

General rules

1. The standard entry point within a called program or method is determined by the Procedure Division header. In order to transfer control from a calling source unit to that entry point, the calling source unit must contain a CALL or INVOKE statement. The name supplied in this CALL or INVOKE statement must be the same as the name specified in the PROGRAM-ID or METHOD-ID paragraph of the Identification Division of the called program or method.
2. The USING phrase, when specified, has the effect that data-name-1 of the Procedure Division header in the called program/method and identifier-2 or identifier-5 in theUSING phrase of the CALL or INVOKE statement in the calling source unit refer to the same set of data, which is equally available both to the called and the calling source unit. It is not necessary for the names to be identical.

A data-name may appear only once in the Procedure Division header of the called program or method, whereas the same identifier may occur several times in the USING phrase of the CALL or INVOKE statement.

3. In the called program/method, the operands of the USING phrase are treated according to the data description supplied in the LINKAGE SECTION.
4. A source unit may run both as a called source unit and as a calling source unit at execution time. An exception to this is the first source unit (called for execution by the system); it **must not** contain a USING phrase in the Procedure Division header.

Example 8-1

```
Calling program:

IDENTIFICATION DIVISION
PROGRAM-ID. A-PROG.
...
WORKING-STORAGE SECTION.
01  ALPHA ...
01  BETA ...
77  GAMMA ...
...
PROCEDURE DIVISION.
...
    CALL "B-PROG" USING ALPHA BETA GAMMA.        (1)
...

Called program:

IDENTIFICATION DIVISION.
PROGRAM-ID. B-PROG.
...
LINKAGE SECTION.
01  DELTA ...
01  EPSILON ...
77  THETA ...
...
PROCEDURE DIVISION USING DELTA EPSILON THETA.    (1)
...
```

(1) The parameters of the USING phrases relate to one another in pairs, i.e. ALPHA and DELTA, BETA and EPSILON, GAMMA and THETA respectively each relate to the same data item.

> For details on passing parameters between ICLS refer to the "CRTE" User Guide [2].

8.3 DECLARATIVES

Function

The DECLARATIVES subdivision is an optional portion of the Procedure Division. It contains a group of procedures, called declarative procedures, which are not executed within the normal sequence of statements in the Procedure Division but only when a particular condition occurs.

Declarative procedures are used for performing the following functions:

- input/output label handling
- handling of input/output errors
- special Report Writer functions.

Format (General format in the Procedure Division)

```
PROCEDURE DIVISION [USING {data-name-1}... ].

[ DECLARATIVES.
  { section-name SECTION.
      USE statement.
 [paragraph-name.
      [sentence]...]...}...
 END DECLARATIVES.]

  {section-name SECTION [segment-number].
  [paragraph-name.
      [sentence]...]...}...
```

Syntax rules

1. Declarative procedures must be placed at the beginning of the Procedure Division, preceded by the keyword DECLARATIVES and followed by a period and a space. Declarative procedures are terminated by the keyword END DECLARATIVES, followed by a period and a space.
2. As indicated in the general format of the Procedure Division, the DECLARATIVES subdivision must be divided into sections. These sections are called declarative sections. Each declarative section contains a group of related procedures, and is preceded by a section header, immediately followed by a USE statement with subsequent period and space.
3. The USE statement defines the type of declarative procedures according to the three functions listed above. The formats of the USE statement are described in detail in chapter "USE statement" and chapter "USE BEFORE REPORTING statement".
4. The USE statement itself is never executed; rather, it defines the conditions for executing the declarative procedures specified in the associated section.

8.4 Arithmetic expressions

Function

Arithmetic expressions allow the user to combine arithmetic operations.

Format

An **arithmetic expression** may be one of the following:

- An identifier of a numeric elementary item.
- A numeric literal.
- Two arithmetic expressions separated by an arithmetic operator or an arithmetic expression enclosed in parentheses.

Any arithmetic expression may be preceded by a unary plus (+) or a unary minus (-).

Arithmetic operators

The following **operators** may be used in arithmetic expressions:

Binary arithmetic operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponentiation

Unary arithmetic operator	Meaning
+	The effect of multiplication by the numeric literal +1
-	The effect of multiplication by the numeric literal -1

As defined by the standard, a binary arithmetic operator must always be preceded and followed by a space. However, the compiler allows all these operators, with the exception of the addition and subtraction operators, to be used without the enclosing spaces. The subtraction operator (-) must always be preceded and followed by a space.

The addition operator (+) must be followed by a space if it occurs before an unsigned numeric literal. Both operators may be immediately preceded and followed by a parenthesis.

A unary + must be followed by a space, if it is before an unsigned literal.
A unary - always must be followed by a space.

Rules for the formation and evaluation of expressions

1. An arithmetic expression may begin only with a left parenthesis, a unary +, a unary -, an identifier, or a literal; and it may end only with a right parenthesis or a variable (identifier or literal).

2. There must be a one-to-one correspondence between left and right parentheses in an arithmetic expression.
3. Table 18 shows the permissible combinations of operators, variables and parentheses in arithmetic expressions.

First symbol	Second symbol				
	identifier, literal	arithmetic operator	unary operator	(	)
identifier, literal	-	P	-	-	P
arithmetic operator	P	-	P	P	-
unary operator	P	-	-	P	-
(	P	-	P	P	-
)	-	P	-	-	P

Table 18: Valid symbol combinations in arithmetic expressions

P indicates that the two symbols may appear consecutively (permissible pair).
- indicates that the two symbols must not appear consecutively (invalid pair).

4. Parentheses may be used in arithmetic expressions in order to indicate the order in which the elements are to be evaluated.
5. Expressions within parentheses are evaluated first. When nested parentheses are used, evaluation proceeds from the innermost to the outermost set of parentheses.
6. When no parentheses are used, or parenthesized expressions are at the same level of inclusiveness, the following hierarchical order of execution is implied:
   a. Unary plus or minus (evaluated first)
   b. Exponentiation
   c. Multiplication and division
   d. Addition and subtraction (evaluated last)
7. If consecutive operations of the same hierarchical level occur, they are evaluated from left to right.

General rules

1. Parentheses are used either to eliminate ambiguities in logic where consecutive operations of the same hierarchical level appear, or to modify the normal hierarchical sequence of execution.
2. Arithmetic expressions are used in arithmetic and conditional statements.

Example 8-2

Expression: A + (B - C) * D

Evaluation:
1. B - C (denote result by x)
2. x * D denote result by y)
3. A + y (final result)

Example 8-3

Expression:	A + ((B / C) + (D ** E) * F) - G	
Evaluation:	1. B / C	(denote result by z)
	2. D ** E	(denote result by x)
	3. x * F	(denote result by y)
	4. z + y	(denote result by a)
	5. A + a	(denote result by b)
	6. b - G	(final result)

8.5 Conditions

General description

A condition enables the program to select between two alternative paths of execution, depending upon the truth value of a test. There are two categories of conditions: simple conditions and complex conditions.

Simple conditions

1. Condition-name condition
2. Class condition
3. Switch-status condition
4. Condition-name condition
5. Sign condition
6. Omitted-argument condition

Each of these conditions may be enclosed in parentheses.

Complex conditions

Complex conditions are formed by combining simple conditions and/or complex conditions with the logical operators AND and OR or by negating these conditions with the logical operator NOT.

8.5.1 Condition-name condition

Function

The condition-name condition causes a conditional variable to be tested to determine whether or not its value is equal to one of the values associated with a specified condition-name. (See section "VALUE clause").

Format

```
condition-name
```

Syntax rules

1. condition-name specifies the condition-name to be used in the test.
2. If a condition-name is associated with a single value, then the related test is true, only if the value corresponding to the condition-name equals the value of its associated conditional variable.
3. If the condition-name is associated with one or more ranges of values, then the conditional variable is tested to determine whether or not its value falls in the range, including the end values.
4. The condition-name condition is a shorthand form of the relation condition (see "Example 8-4").
   See also “Format 4” in section "SET statement".

Example 8-4

```
02  PAY-CLASS PICTURE  9.
    88  HOURLY   VALUE 1.
    88  WEEKLY   VALUE 2.
    88  MONTHLY  VALUE 3.
         ...
    IF  HOURLY GO TO HOUR-PROCEDURE.
```

Here, PAY-CLASS is a conditional variable, and HOURLY, WEEKLY, and MONTHLY are condition-names. If the current value of PAY-CLASS is 1, the result of the test in the IF statement is true. Otherwise, the result is false.

As noted above, the condition-name is a shorthand form of the relation condition. The following statement, which contains a relation condition, is equivalent to the above IF statement:

```
IF PAY-CLASS = 1 GO TO HOUR-PROZEDUR.
```

8.5.2 Class condition

Function

The class condition determines whether an operand is numeric, alphabetic, alphabeticlower, alphabetic-upper, or whether it contains only characters from a character set specified with class-name in the CLASS clause of the SPECIAL-NAMES paragraph of the Environment Division.

Format

```
identifier IS [NOT] {NUMERIC | ALPHABETIC | ALPHABETIC-LOWER | ALPHABETIC-UPPER |
class-name}
```

Syntax rules

1. identifier must be a data item described implicitly or explicitly with USAGE DISPLAY or USAGE NATIONAL or COMPUTATIONAL-3 or PACKED-DECIMAL.
2. Table 19 lists all categories of identifiers which are permissible in the tests:

Tests	Permissible for identifiers of the categories
NOT] ALPHABETIC [NOT] ALPHABETIC-LOWER [NOT] ALPHABETIC-UPPER	Alphabetic Alphanumeric Alphanumeric-edited Numeric-edited National
[NOT] NUMERIC	Alphanumeric Alphanumeric-edited Numeric-edited Numeric National
class-name	Alphabetic Alphanumeric Alphanumeric-edited Numeric-edited Numeric (only USAGE DISPLAY)

Table 19: Valid formats of the class condition

General rules

1. identifier specifies the data item to be tested.
2. NUMERIC, ALPHABETIC, ALPHABETIC-LOWER, ALPHABETIC-UPPER and class-name (possibly negated by NOT) indicate which characteristic is to be tested.
3. If identifier is a zero-length item, the value of the class condition without a NOT phrase is "false".
4. identifier is treated as numeric when
    a. its numeric category is

- identifier is described with USAGE DISPLAY and it contains only the sign corresponding to its description and the digits 0 through 9.
- identifier is described with USAGE COMP-3 or USAGE PACKED-DECIMAL and its content is numeric.

If no sign is defined in the PICTURE character-string, only F is permitted as a sign in the internal representation.
If a sign is defined, you will find detailed information in section "SIGN clause".

b. its category is not numeric and it contains only the digits 0 through 9.

5. identifier is treated as alphabetic when its contents consist of any combination of the characters A through Z and /or a through z and the space character.
6. identifier is treated as alphabetic-lower when its contents consist of a combination of the lowercase letters a through z and the space character.
7. identifier is treated as alphabetic-upper when its contents consist of a combination of the uppercase letters A through Z and the space character.
8. identifier corresponds to class-name when its contents consist solely of a combination of those characters which were defined by means of class-name in the SPECIAL-NAMES paragraph.

8.5.3 Switch-status condition

Function

The switch-status condition tests the setting of an implementor-defined user or task switch. The implementor-name and the ON or OFF value associated with the condition must appear in the SPECIAL-NAMES paragraph of the Environment Division.

Format

```
condition-name
```

Syntax rules

1. The result of the test is true if the switch is set to the position corresponding to condition-name.
2. The status of a switch can be changed by means of a format 3 SET statement (see “Format 3” in section "SET statement").

8.5.4 Relation condition

Function

A relation condition causes a comparison of two operands, each of which may be a literal, an index or an arithmetic expression.

Format

```
{  identifier-1                  relational-          {  identifier-2
 | literal-1                     operator              | literal-2
 | arithmetic-expression-1                             | arithmetic-expression-2
 | index-1                                             | index-2
}                                                     }
```

Syntax rules

1. The first and the second operand of a relational condition must not both be literals. Nor may both operands be NULL at the same time.
2. Relational-operator must be one of the operators listed in Table 20. It must be preceded and followed by a space.

Operator	Meaning
IS [NOT] GREATER THAN IS [NOT] >	[Not] greater than
IS [NOT] LESS THAN IS [NOT] <	[Not] less than
IS [NOT] EQUAL TO IS [NOT] =	[Not] equal to
IS GREATER THAN OR EQUAL TO IS >=	Greater than or equal to
IS LESS THAN OR EQUAL TO IS <=	Less than or equal to

Table 20: Relational operators

The special symbols <, > and = are not underlined in could be mistaken for other symbols.

3. The relational operator specifies the type of comparison to be made in a relation condition.
4. If data of the class object or pointer is involved in a relation condition, only relational operators with the meaning "equal to" or "not equal to" may be used.
5. If data of the class object or pointer is involved in a relation condition, the left and right operands must be of the same category.

6. Strongly typed group items may not be compared. Only the elementary items from strongly typed group items may be compared with each other.
7. All allowable comparisons and the type of comparison are shown in table 21:

Right operand **Left operand**	Group	Alphabetic	Alphanumeric	National	Numeric	Expression	Index name	Index data item	Object	Poin
Group	an	an	an	-	an	-	-	-	-	-
Alphabetic	an	an	an	ant	-	-	-	-	-	-
Alphanumeric	an	an	an	ant	anu,i,d	-	-	-	-	-
National	-	ant	ant	nt	nnu,i	-	-	-	-	-
Numeric	an	-	anu,i,d	nnu,i	nu	nu	ix2,i	-	-	-
Expression	-	-	-	-	nu	nu	ix2,i	-	-	-
Index name	-	-	-	-	ix2,i	ix2,i	ix1	ix3	-	-
Index data item	-	-	-	-	-	-	ix3	ix3	-	-
Object	-	-	-	-	-	-	-	-	ob	-
Pointer	-	-	-	-	-	-	-	-	-	pt

Table 21: Permissible comparisons between operands [1)]

1) Table entries:

i = Only for integers

d = In the case of alphanumeric literals, only permitted for numeric items with USAGE DISPLAY

- = Comparison is not permitted

nu = Comparison of numeric operands

an = Comparison of alphanumeric operands

nt = Comparison of national operands

ix = Comparison of index names or index data items

ob = Comparison of object references

pt = Comparison of pointers

ant = Comparison of alphanumeric and national operands

anu = Comparison of alphanumeric and integral numeric operands

nnu = Comparison of national and integral numeric operands

Comparison of numeric operands (nu)

When two numeric operands are compared, their algebraic values are compared regardless of the data formats (USAGE clause). Their lengths (that is, the number of digits they contain) are not significant.
Unsigned numeric operands are considered to be positive for purposes of comparison.
Zero is considered to be a unique value, regardless of sign.

Example 8-5

-50 is less than +5
+75 is greater than +5
-100 is less than -10
-0 is equal to + 0

Comparison of alphanumeric operands with integral numeric operands (anu)

The numeric operand must be an integral literal or an integral data item with USAGE DISPLAY. However, the compiler described here also permits other data formats.

a. If the alphanumeric operand is an **elementary item** or an alphanumeric literal, the numeric operand is treated as if it had been moved to an alphanumeric elementary item in accordance with the rules of the MOVE statement. The length of the alphanumeric data item corresponds to the number of digits in the description of the numeric data item; any scaling position character P are not counted.
b. If the alphanumeric operand is a **group item**, the numeric operand is treated as if it had been moved to a group item of the same size as the numeric data item in accordance with the rules of the MOVE statement.

In both cases the conceptual data item is then compared with the alphanumeric operand.
This is done in accordance with the rules for the comparison of alphanumeric operands (an).

Comparison of national operands with integral numeric operands (nnu)

The numeric operand must be an integral literal or an integral data item. It is treated as if it had been moved to a national elementary item in accordance with the rules of the MOVE statement. The length of the national data item corresponds to the number of digits in the description of the numeric data item; any scaling position characters P are not counted.

The conceptual national data item is then compared with the national operand. This is done in accordance with the rules for comparing national operands (nt).

Comparison of alphanumeric operands (an)

When two non-numeric operands are compared, the comparison is made with respect to the collating sequence of the PROGRAM COLLATING SEQUENCE (see section "OBJECT-COMPUTER paragraph").

There are two cases to consider:

a. Comparison of operands of equal size

The program compares characters in corresponding character positions, starting at the high-order end (that is, leftmost), and continuing until it encounters two unequal characters or until it reaches the low-order end of the operands.
If all pairs of corresponding characters are equal, the operands are considered to be equal. Two operands with the size zero are equal, too.
If the object program encounters a pair of unequal characters, it determines which character has a higher position in the collating sequence. The operand containing the higher character is considered to be the greater operand.

Example 8-6

left operand	**right operand**	**no COLLATING SEQUENCE or COLLATING SEQUENCE NATIVE**	**COLLATING SEQUENCE ALPHATAB (see “Example 6-5” in** section "ALPHABET clause" **)**
+-*	ABC	’less than’ The 1st character left operand "+" is less than the 1st character right operand "A".	’greater than’ The 1st character left operand "+" is greater than the 1st character right operand "A" because "A" has the position 1 in the collating sequence ALPHATAB.
RADE	RABE	’greater than’ The 3rd character left operand "D" is greater than the 3rd character right operand "B".	’equal to’ The 3rd character left operand "D" is equal to the 3rd character right operand "B" because "D" and "B" have the position 1 in the collating sequence ALPHATAB.
XYZ	XYZ	’equal to’ All pairs of corresponding characters are equal.	’equal to’ All pairs of corresponding characters are equal.

b. Comparison of operands of unequal size

The comparison proceeds as though the shorter operand were extended on the right by sufficient alphanumeric spaces to make the operands of equal size.

Then follow the rules described in a) "Comparison of operands of equal size".

Example 8-7

left operand	**right operand**	**no COLLATING SEQUENCE or COLLATING SEQUENCE NATIVE**	**COLLATING SEQUENCE ALPHATAB (see “Example 6-5” in section "ALPHABET clause")**
SMITH	SMITHY	’less than’ The left operand is filled with blanks to "SMITH’BLANK'". The 6th character left operand "'BLANK'" is less than the 6th character right operand "Y".	’less than’ The left operand is filled with blanks to "SMITH’ BLANK'". The 6th character left operand "'BLANK’ " is less than the 6th character right operand "Y".
AB	ABBA	’less than’	’greater than’

		The left operand is filled with blanks to "AB'BLANK' 'BLANK'". The 3rd character left operand "'BLANK'" is less than the 3rd character right operand "B".	The left operand is filled with blanks to "AB'BLANK''BLANK'". The 3rd character left operand "'BLANK'" is greater than the 3rd character right operand "B" because "B" has position 1 in the collating sequence ALPHATAB.

Comparison of national operands (nt)

There are two cases to consider:

a. Comparison of operands of equal size

The program compares characters in corresponding character positions, starting at the high-order end (that is, leftmost), and continuing until it encounters two unequal characters or until it reaches the low-order end of the operands.
If all pairs of corresponding characters are equal, the operands are considered to be equal. Two operands with the size zero are equal, too.
If the object program encounters a pair of unequal characters, it determines which character has a higher position in the national collating sequence. The operand containing the higher character is considered to be the greater operand.

b. Comparison of operands of unequal size

The comparison proceeds as though the shorter operand were extended on the right by sufficient national spaces to make the operands of equal size.

Then follow the rules described in a) "Comparison of operands of equal size".

Comparison of alphanumeric and national operands (ant)

The alphanumeric operand is treated as if it has been moved and converted to a national elementary item in accordance with the rules of the MOVE statement. The national data item has exactly as many character positions as the alphanumeric data item.

The conceptual data item is then compared with the national operand. This is done in accordance with the rules for comparing national operands (nt).

Example 8-8

left, alphanumeric operand	right, national operand	Result of the comparison
123	XYZ	'less than' The left operand is converted to national representation. The 1st character left operand "1" is less than the 1st character right operand "X" in the national collating sequence.
AB	ABBA	'less than' The left operand is converted to national representation and filled with blanks to "AB 'BLANK' 'BLANK'". In the national collating sequence the 3rd character left operand "'BLANK'" is less than the 3rd character right operand "B".

Comparison of index names or index data items (ix)

1. If the two operands are index names, the two table element numbers which correspond to the two indices are compared with each other (ix1).
2. If one operand is numeric (integer only), this number is regarded as a table element number and compared with the table element number which corresponds to the index name (ix2).
3. If one operand is an index data item, the current values of the items are compared (without conversion to table element numbers!) (ix3).

Comparison of object references (ob)

The condition ’equal’ is only true if identifier-1 and identifier-2 reference the same object.

Comparison of pointers (po)

The condition ’equal’ is only true if identifier-1 and identifier-2 reference the same memory location.

8.5.5 Sign condition

Function

The sign condition determines whether or not the algebraic value of a numeric operand (that is, an item described as numeric) is less than, greater than, or equal to zero.

Format

```
{identifier | arithmetic-expression} IS [NOT] {POSITIVE | NEGATIVE | ZERO}
```

Syntax rule

1. identifier must refer to a numeric data item.

General rules

1. identifier or arithmetic-expression identifies the operand to be tested.
2. POSITIVE, NEGATIVE or ZERO specifies the test to be made.
3. An operand is positive if its value is greater than zero, negative if its value is less than zero, and zero if its value is equal to zero.

8.5.6 OMITTED-ARGUMENT condition

The OMITTED-ARGUMENT condition determines whether an argument has been provided for a program or a method.

Format

```
data-name-1 IS [NOT] OMITTED
```

Syntax rules

1. data-name-1 must be defined in the LINKAGE SECTION with level number 01 or 77. The data description entry for data-name-1 must not contain a REDEFINES clause.
2. data-name-1 must be specified in the USING phrase of the Procedure Division header of the source element in which this condition is contained.

General rules

1. The result of the IS OMITTED test is true:
   a. if the OMITTED phrase is specified as the current value for the formal parameter data-name-1 of the called program or called method.
   b. if no current value was specified for the formal parameter data-name-1; this is only possible for parameters at the end of the USING list, even if no value was specified for all subsequent parameters.
2. If NOT is specified, the truth value as per item No. 1 is reversed.

8.5.7 Complex conditions

Function

A complex condition consists of a combination of two or more simple conditions.

Format

```
condition {AND | OR} [NOT] condition  [{AND | OR} [NOT] condition] ...
```

Syntax rules

1. condition specifies a simple condition.
2. Parentheses may be used within a complex condition to improve readability or to modify the normal hierarchical sequence of execution.
3. The simple conditions within a complex condition are separated from each other by logical operators, according to the specified rules. The logical operators must be preceded by a space and followed by a space.
4. A complex condition may comprise up to 60 simple conditions.
5. Table 22 lists the logical operators and their meanings.

Operator	Meaning	Example
OR	Logical inclusive Or (either or both)	The expression A OR B is true if A is true, or B is true, or both A and B are true.
AND	Logical conjunction (both)	The expression A AND B is true only if both A and B are true.
NOT	Logical negation	The expression "NOT" A is true only if A is false.

Table 22: Logical operators

6. The ways in which conditions and logical operators may be combined are shown in Table 23.

First symbol	Second symbol					
	simple-condition	OR	AND	NOT	(	)
simple-condition	-	P	P	-	-	P
OR	P	-	-	P	P	-
AND	P	-	-	P	P	-
NOT	P	-	-	-	P	-
(	P	-	-	P	P	-
)	-	P	P	-	-	P

Table 23: Valid symbol pairs of conditions and logical operators[1]

[1)] P indicates that the two symbols may be used as a pair.

7. ## Rules of precedence for evaluation of expressions

The evaluation of complex conditions starts with the innermost pair of parentheses and proceeds through to the outermost pair of parentheses.

If the order of evaluation is not determined by parentheses, the expression is evaluated according to the following precedence (hierarchical levels):

- Arithmetic expressions
- Relational operators
- NOT conditions
- AND and its associated conditions are evaluated from left to right.
- OR and its associated conditions are evaluated last, also proceeding from left to right.
- If consecutive expressions have the same hierarchical level, they are evaluated from left to right.

Example 8-9

Consider this expression:

```
A IS NOT GREATER THAN B OR A + B IS EQUAL TO C AND D IS POSITIVE
```

This expression is evaluated as if the following parentheses had been supplied:

```
(A IS NOT GREATER THAN B) OR (((A+B) IS EQUAL TO C) AND (D IS POSITIVE)).
```

Example 8-10

Table 24 shows some of the relationships between logical operators and simple conditions.

Operands	Value of A [1)]	True	False	True	False
	Value of B [1)]	True	True	False	False
Combinations	NOT A	False	True	False	True
	A AND B	True	False	False	False
	A OR B	True	True	True	False
	NOT (A AND B)	False	True	True	True
	NOT A AND B	False	True	False	False
	NOT (A OR B)	False	False	False	True
	NOT A OR B	True	True	False	True

Table 24: Results of logical operators

[1)] A and B represent simple conditions.

8.5.8 Implied subjects and relational operators

Function

When a complex condition is written without parentheses, any relation condition except the first may be abbreviated as follows:

- the subject of the relation condition may be omitted.
- the subject and relational operator of the relation condition may be omitted.

However, the compiler permits the use of parentheses in relation subjects and relation objects which are arithmetic expressions, and in order to affect the sequence in which the logical operators AND and OR are evaluated.

Format of implied subject

```
...subject relational-operator object {AND | OR} [NOT] relational-operator object...
```

Format of implied subject and relational operator

```
...subject relational-operator object {AND | OR} [NOT] object...
```

Syntax rules

1. Within a sequence of relation conditions, both forms of abbreviation may be used. The effect of using such abbreviations is the same as if the omitted subject were replaced by the most recently stated subject, or the omitted relational-operator were replaced by the most recently stated relational-operator.
2. NOT in an abbreviated complex condition is interpreted as follows:
   a. If the word NOT is followed by one of the relational operators
   GREATER, >, LESS, <, EQUAL, =,
   then NOT is considered as part of the relevant relational operator.
   b. If the word NOT is followed by one of the other relational operators, then NOT is considered as a logical operator to negate the relevant relation condition.
3. A NOT appearing in front of a left parenthesis remains in effect up to the associated right parenthesis (see "Example 8-15").

Example 8-11

Implied subjects

```
IF X = Y OR > W OR < Z
```

is equivalent to

```
IF X = Y OR X > W OR X < Z
```

In this example, the implied subject is the most recently stated subject, i.e. X.

Example 8-12

Implied subjects and relational operators

```
IF X = Y OR Z OR W
```

is equivalent to

```
IF X = Y OR X = Z OR X = W
```

In this example, the implied subject is the most recently stated subject, i.e. X; and the implied relational operator is the most recently stated operator, i.e. =.

Example 8-13

Implied subject, and implied subject with relational operator

```
X = Y AND > Z OR A
```

is equivalent to

```
X = Y AND X > Z OR X > A
```

Here, since X is the only stated subject, it is substituted in both simple conditions. The most recently stated operator, >, is substituted in the third simple condition.

Example 8-14

```
A > B AND NOT > C OR D
```

is equivalent to

```
A > B AND NOT A > C OR NOT A > D
```

or

```
((A > B) AND (A NOT > C)) OR (A NOT > D)
```

Example 8-15

```
A NOT = "A" AND NOT ("B" OR NOT "C")
```

is equivalent to

```
A NOT = "A" AND NOT (A NOT = "B" OR NOT A NOT = "C")
```

or

```
A NOT = "A" AND A = "B" AND A NOT = "C"
```

or

```
A = "B".
```

8.6 Arithmetic statements

Syntax rules

1. All identifiers used in arithmetic statements must be defined as numeric data in the Data Division.
2. All literals used in arithmetic statements must be numeric. They may be floating-point literals.
3. The maximum size of any operand (identifier or literal) is 31 decimal digits.
4. The maximum size of all results after decimal point alignment is 31 decimal digits.
5. When several operands occurring in an arithmetic statement are "overlapped" in a hypothetical data item, aligned on their decimal points, then the maximum size of the data item required (i.e. the composite of operands) is 31 decimal digits (see "ADD statement" and "SUBTRACT statement").
6. A maximum of 100 operands may be supplied in one arithmetic statement or arithmetic expression. The number of right and left parentheses () must not exceed 250.
7. The format of any data item involved in computations (for example, an addend, a subtrahend, or a multiplier) cannot contain editing symbols. Operational signs and implied decimal points are not considered to be editing symbols.
8. Identifiers which are used only to receive the result of an arithmetic statement (for example, the identifier used with the GIVING phrase) may be numeric-edited items (see section "GIVING phrase").
9. Condition-names cannot appear as operands.

General rules

1. The operands need not have the same data description; any necessary conversion and decimal point alignment is supplied throughout the calculation (see "Rules for numeric moves" on MOVE statement).
2. If the sending or receiving items of an arithmetic statement, or of an INSPECT, MOVE, SET, STRING or UNSTRING statement, share the same storage area (that is, if the operands overlap), the result of the execution of such a statement is undefined.
3. The results are also undefined if the identifiers contain any data other than numeric data at object time.

   > ℹ If the input operands for an arithmetic statement do not contain valid numeric data, a data error will occur at program runtime.

4. The following rules apply to evaluation of exponentiation in an arithmetic expression:
   a. If the value of an expression to be raised to a power is zero, the exponent must have a value greater than zero. Otherwise, the size error condition exists.
   b. If the evaluation yields both a positive and a negative real number, the value returned as the result is the positive number.
   c. If no real number exists as the result of the evaluation, the size error condition exists.
5. In evaluating arithmetic statements, the compiler generates a number of arithmetic operations. Depending on the relationship between the various operands, the compiler will generate one or more intermediate result items. These intermediate result items are retained until required for solving the final result of that statement.

   Table 25 shows the number of the integer and decimal digits which are stored in the result item according to the operation executed. From this table, it is possible to determine the optimum operand size for the desired precision to be achieved in the statement. On the basis of the decimal places contained in each operand, and by reference to the formulae supplied in the table, the programmer may determine the exact number of positions which the compiler will make available to the result item. However, the result placed in the result item

is aligned on the decimal point. Hence decimal point alignment is likewise important in determining the precision of the result.

If one of the operands has the data format COMPUTATIONAL or COMPUTATIONAL-5, then special rules apply that cannot be shown in table 25.

Statement type	**Operation**	**Decimal places in intermediate result (d)**	**Integer places in intermediate result (i)**
Arithmetic	Addition or subtraction (+) or (-)	MAX (Ad, Bd)	MAX (Ai+1, Bi+1)
	Multiplication (*)	Ad+Bd	Ai+Bi
	Division (/)	MAX (Fd+1, Ad)	Ai+Bd
IF or PERFORM	Addition or subtraction (+) or (-)	MAX (Ad, Bd)	MAX(Ai+1, Bi+1)
	Multiplication(*)	Ad+Bd	Ai+Bi
	Division (/)	Ad	Ai+Bd

Table 25: Calculating the integer and decimal places in intermediate results

i	= Calculated integer places
Ai	= Integer places in first operand
Bi	= Integer places in second operand
d	= Calculated decimal places
Ad	= Decimal places in first operand
Bd	= Decimal places in second operand
Fd	= Decimal places in final result
MAX	= The greater value of the specified operands in each case

If the result item (i+d) contains more than 31 places then floating point arithmetic is generally used to perform the calculation.

The following are also performed using floating point arithmetic

- Potentiation
- Divisions specified in the arguments of internal standard functions

8.7 Phrases in statements

8.7.1 CORRESPONDING phrase

The CORRESPONDING phrase enables the user to write one statement to perform operations on several elementary items of the same name in different groups.

1. The word CORRESPONDING may be abbreviated as CORR.
2. All identifiers must refer to group items. If an identifier refers to a national group item, this is not processed like an elementary item but like a group.
3. Pairs of data items correspond if the following conditions exist; all other items are ignored for the operation:
4. Both data items have the same name and qualification, up to, but not necessarily including, identifier-1 and identifier-2.
5. None of the data items is declared with FILLER.
6. In the case of MOVE CORRESPONDING, at least one of the data items is an elementary item, and the MOVE statement resulting from it applies in accordance with the rules of the MOVE statement; both data items are elementary items in the case of ADD CORRESPONDING or SUBTRACT CORRESPONDING.
7. A data item that is subordinate to identifier-1 or identifier-2 and which is defined with a REDEFINES, RENAMES, OCCURS, or USAGE IS INDEX clause or is of class “object” or “pointer” will be ignored; any items which are subordinate to such items are also ignored. However, identifier-1 or identifier-2 may be defined with REDEFINES or OCCURS clauses or be subordinate to data items defined with REDEFINES or OCCURS clauses.
8. The CORRESPONDING phrase cannot be applied to identifiers that are subjected to reference modification.

Example 8-16

In this example, elementary items in EMPLOYEE-RECORD are subtracted from corresponding items in PAYROLL-CHECK.

Procedure Division statement:

```
    SUBTRACT CORRESPONDING EMPLOYEE-RECORD FROM PAYROLL-CHECK.
```

Data Division entries:

```
01  EMPLOYEE-RECORD.
    02  EMPLOYEE-NUMBER.
        03  PLANT-LOCATION...
        03  CLOCK-NUMBER.
            04  SHIFT-CODE...
            04  CONTROL-NUMBER...
    02  WAGES.
        03  HOURS-WORKED...
        03  PAY-RATE...
    02  FICA-RATE...
    02  DEDUCTIONS...
```

```
01  PAYROLL-CHECK.
    02  EMPLOYEE-NUMBER.
        03  CLOCK-NUMBER...
        03  FILLER...
    02  DEDUCTIONS.
        03  FICA-RATE...
        03  WITHHOLDING-TAX...
        03  PERSONAL-LOANS...
    02  WAGES.
        03  HOURS-WORKED...
        03  PAY-RATE...
    02  NET-PAY...
    02  EMPLOYEE-NAME.
        03  SHIFT-CODE...
```

According to the rules for the CORRESPONDING phrase, the following subtractions take place:

1st operand	2nd operand
HOURS-WORKED OF WAGES OF EMPLOYEE-RECORD	HOURS-WORKED OF WAGES OF PAYROLL-CHECK
PAY-RATE OF WAGES OF EMPLOYEE-RECORD	PAY-RATE OF WAGES OF PAYROLL-CHECK

The following items are not subtracted, for the reasons stated:

Item	Reason
EMPLOYEE-NUMBER	Item not elementary item in either group
PLANT-LOCATION OF EMPLOYEE-NUMBER OF EMPLOYEE-RECORD	Name does not appear under PAYROLL-CHECK
CLOCK-NUMBER OF EMPLOYEE-NUMBER	Item is not elementary in one group
SHIFT-CODE OF CLOCK-NUMBER OF EMPLOYEE-NUMBER OF EMPLOYEE-RECORD	Qualification is not identical in PAYROLL-CHECK
CONTROL-NUMBER OF CLOCK-NUMBER OF EMPLOYEE-NUMBER OF EMPLOYEE-RECORD	Name does not appear under PAYROLL-CHECK
WAGES	Name is not elementary in either group
TAX-RATE OF EMPLOYEE-RECORD	Qualification is not identical in PAYROLL-CHECK
DEDUCTIONS	Item not elementary in one group

8.7.2 GIVING phrase

Syntax rules

1. The identifier following the word GIVING may be a numeric-edited item since it is not itself involved in the calculation.
2. When the GIVING phrase is supplied, the result of the arithmetic operation is assigned to the specified identifier.
3. The result stored in the identifier replaces its previous contents. Therefore, it is not necessary to reset the identifier to zero.

Example 8-17

```
ADD A B GIVING C.
```

The value C is set to the sum of A + B, and A and B are not changed.

8.7.3 ROUNDED phrase

Syntax rules

1. If, after decimal point alignment, the number of places following the decimal point in the result of an arithmetic operation is greater than the number of decimal places provided in the resultant identifier, truncation is performed according to the size of this identifier.
   If rounding is specified, the absolute value of the last significant digit of the resultant identifier is incremented by 1 if the most significant digit of those to be truncated is greater than or equal to 5.
2. If rounding is not desired but truncation of excess digits is required, the last digit of the resultant identifier remains unchanged.
3. When the least significant digits of a resultant identifier are represented by P in the PICTURE character-string for that identifier, rounding or truncation takes place relative to the rightmost digit position for which internal storage is allocated (see Example 8-18).

   ROUNDED is assumed for COMPUTATIONAL-1 or COMPUTATIONAL-2 result items, and need not be specified for them.

Example 8-18

Calculated result [1)]	Description of result item	Description after rounding	Result without rounding
03&2627	PIC 99 PIC 99.9 PIC 99.99 PIC 99.999	03 03.3 03.26 03.263	03 03.2 03.26 03.262
123788&6	PIC S999PPP	124000	123000

[1)] & represents the operational decimal point.

8.7.4 ON SIZE ERROR phrase

A size error condition exists if, after decimal point alignment, the integer digits in the computed result exceed the number of places provided for them and thus cause an overflow.

Syntax rules

1. Violation of the rules for evaluation of exponentiation always terminates the arithmetic operation and always causes a size error condition (see section "Arithmetic statements" et seq.).
2. The ON SIZE ERROR phrase contains an imperative-statement which specifies what actions are to be taken in the event of a size error.
3. The size error condition applies only to the final results of an arithmetic operation and not to intermediate results, except in the case of the MULTIPLY and DIVIDE statements.
4. If the ROUNDED phrase is specified, rounding takes place before the size error check.
5. If the ON SIZE ERROR phrase is specified and a size error condition exists after the execution of the arithmetic statement, the values of the affected resultant identifiers remain unchanged from the values they had before execution of the arithmetic statement. The values of resultant identifiers for which no size error condition exists are the same as they would have been if the size error condition had not resulted for any of the resultant identifiers. After completion of the arithmetic operations, control is transferred to the imperative-statement specified in the ON SIZE ERROR phrase and execution continues according to the rules for each statement specified in that imperative-statement. If a procedure branching or conditional statement which causes explicit transfer of control is executed, control is transferred in accordance with the rules for that statement; otherwise, upon completion of the execution of the imperativestatement specified in the ON SIZE ERROR phrase, control is transferred to the end of the arithmetic statement and the NOT ON SIZE ERROR phrase, if specified, is ignored.
6. If ON SIZE ERROR is not specified and a size error condition exists after the execution of the arithmetic operations specified by an arithmetic statement, the values of the affected resultant identifiers are undefined. The values of resultant identifiers for which no size error condition exists are the same as they would have been if the size error condition had not resulted for any of the resultant identifiers. After completion of the arithmetic operations, control is transferred to the end of the arithmetic statement, and the NOT ON SIZE ERROR phrase, if present, is ignored.
7. If the size error condition does not exist, control is transferred to the end of the arithmetic statement or to the imperative-statement specified in the NOT ON SIZE ERROR phrase if it is specified. In the latter case, execution continues according to the rules for each statement specified in that imperative-statement. If a procedure branching or conditional statement which causes explicit transfer of control is executed, control is transferred in accordance with the rules for that statement; otherwise, upon completion of the execution of the imperative-statement specified in the NOT ON SIZE ERROR phrase, control is transferred to the end of the arithmetic statement.
8. For an ADD statement with the CORRESPONDING phrase or a SUBTRACT statement with the CORRESPONDING phrase, if any of the individual operations produces a size error condition, the imperative-statement specified in the ON SIZE ERROR phrase is not executed until all of the individual additions or subtractions are completed.
9. Division by zero always causes a size error condition.

   For COMPUTATIONAL-1 or COMPUTATIONAL-2 data items, division by zero will cause the imperative-statement in the ON SIZE ERROR phrase to be executed.

Example 8-19

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SE.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
    TERMINAL IS T.
DATA DIVISION.
WORKING-STORAGE SECTION.
77  A PIC 99 VALUE ZERO.
77  B PIC 99 VALUE ZERO.
PROCEDURE DIVISION.
MAIN SECTION.
P1.
    MOVE 44 TO A.
    MOVE 72 TO B.
    ADD A TO B
    ON SIZE ERROR
      PERFORM PROC-A
    END-ADD
    STOP RUN.
PROC-A.
    DISPLAY "Size error!" UPON T.
    DISPLAY A UPON T.
    DISPLAY B UPON T.
```

Current value of A: 44
Current value of B: 72
Calculated result: 116

The result item B is too small to accommodate the calculated result and a size error condition occurs. Since ON SIZE ERROR is specified, the statement PERFORM PROC-A is executed. Result item B is unchanged.

8.8 Overlapping operands

The following rule applies to all statements:

If a sending and a receiving item in any statement share a part or all of the same storage areas, yet are not defined by the same data description entry, the result of the execution of such a statement is undefined. With certain statements, the results will also be undefined if the sending and receiving items are defined by the same data description entry. Further information is contained in the rules associated with the individual statements.

8.9 Incompatible data

Except for the class condition test, the following applies when a data item is referenced in the Procedure Division: If the content of a data item is not compatible with the data class defined for that data item by its PICTURE clause, the result of the operation is undefined.

General rule

1. Every operation involving a numeric data item which may possibly have non-numeric contents (e.g. due to a redefinition of the data item or following a MOVE statement using a group item as operand) should be preceded by the IF NUMERIC class test. The operation can only be performed successfully if the class test yields the truth value TRUE.

8.10 Statements

8.10.1 ACCEPT statement

Function

The ACCEPT statement transfers small amounts of data to a data item. The data is either read from a system file or made available by the compiler or operating system.

Format 1 reads user input by means of appropriate mnemonic names or supplies information of the operating system and compiler.

Format 2 supplies date and time specifications of the operating system.

Format 3 are used to access the command line of the POSIX subsystem.
Format 4

Format 5 supplies the contents of a BS2000 or POSIX environment variable or the contents of a specific argument from the POSIX command line.

Format 1

```
ACCEPT identifier [ FROM mnemonic-name ]
```

Syntax rules

1. identifier can be an alphanumeric group item or an alphabetic, alphanumeric, external decimal data item or external floating-point data item.
   If mnemonic-name is linked to operating system information (COMPILER-INFO, PROCESS-INFO, TERMINAL-INFO or DATE-ISO4), identifier may also be a national data item.
2. mnemonic-name must be specified in the SPECIAL-NAMES paragraph and be associated with one of the following implementor-names:

   ```
   SYSIPT
   TERMINAL
   CONSOLE
   job-variable-name (BS2000 job variable)
   COMPILER-INFO
   CPU-TIME, PROCESS-INFO, TERMINAL-INFO, DATE-ISO4
   ```

3. SYSIPT, TERMINAL, or CONSOLE specifies the system file from which data is to be read.
   SYSIPT refers to the system file of that name.
   TERMINAL refers to the system file SYSDTA (normally assigned to the data terminal).
   CONSOLE refers to the system console.
4. When entered for a job-variable-name, mnemonic-name references the associated operating system job variable which is to be read in.
5. When entered for COMPILER-INFO, CPU-TIME, PROCESS-INFO, or TERMINAL-INFO, mnemonic-name specifies the information which is to be requested.
   COMPILER-INFO refers to information provided by the compiler.
   CPU-TIME, PROCESS-INFO, TERMINAL-INFO and DATE-ISO4 refer to information provided by the operating system.

6. If the FROM phrase is omitted, data is read by default from the logical input file SYSIPT.
   Data can also be read from the logical input file SYSDTA by means of an appropriate compiler directive (see "COBOL2000 User Guide" [1]).
7. identifier must not be defined using the ANY LENGTH clause.
8. The execution of ACCEPT statements and the structure of the information provided for the individual functions are described in the "COBOL2000 User Guide" [1].

General rules

1. Data is stored aligned to the left in the area indicated by identifier, regardless of the PICTURE character-string associated with the identifier. Incoming data is not edited, and no error checking is performed.
   The only exception to this is CPU-TIME: the CPU time is moved in accordance with the rules of the MOVE statement from a field with the description PIC 9(6)V9(4).
   If identifier is a zero-length item, no data is moved.
2. If the system file specified for an ACCEPT statement is the same as one designated for a READ statement, the results will be unpredictable.
3. An ACCEPT statement for job-variable will be rejected with an error message at object time if the job variables are not present in the operating system. If the job variable cannot be read in for some reason, the runtime system issues an error message, and the program is then either continued or aborted, as determined by an appropriate compiler directive (see "COBOL2000 User Guide" [1]).
   In the former case, the literal "/*" is assigned to the identifier according the general rule 1.

Example 8-20

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES:
    TERMINAL IS T.
  .
  .
PROCEDURE DIVISION.
  ...
    ACCEPT INPUT-DATA FROM T.
```

The mnemonic-name T is linked to the implementor-name TERMINAL in the SPECIAL-NAMES paragraph. The subsequent ACCEPT statement requests data from the system file SYSDTA, which is assigned to TERMINAL and moves this data to the data item called INPUT-DATA.

Format 2

```
ACCEPT identifier FROM {DATE | DAY | DAY-OF-WEEK | TIME}
```

Syntax rules

1. The data item specified by identifier must not be an alphabetic elementary item.
2. The ACCEPT statement causes the requested information to be moved to the data item specified by the identifier, in accordance with the rules governing the MOVE statement.

DATE, DAY, DAY-OF-WEEK and TIME are special data items and thus are not described in the compilation unit.

General rules

1. DATE is composed of the data elements "year" of century, "month" of year, and "day" of month. The sequence of these conceptual elementary items is as follows, from left to right: year, month, day. Thus, for example, April 1, 1996 would be expressed as 960401. DATE, if referenced in a COBOL program, is interpreted as if it had been described as an unsigned elementary numeric integer data item, six digits in length (PIC 9(6)).
2. DAY is composed of the data elements "year" of century, and "day" of year. The sequence of these conceptual elementary items is as follows, from left to right: year, day. Thus, for example, April 1, 1998 would be expressed as 98091. DAY, if referenced in a COBOL program, is interpreted as if it had been described as an unsigned elementary numeric integer data item, five digits in length (PIC 9(5)).
3. DAY-OF-WEEK is composed of a single data element whose content represents the day of the week. If referenced in a COBOL program, DAY-OF-WEEK is interpreted as if it had been described as an unsigned elementary numeric integer data item with a length of one digit (PIC 9). In DAY-OF-WEEK, the value 1 represents Monday, 2 represents Tuesday, ... , 7 represents Sunday.
4. TIME is composed of the data elements "hours", "minutes", "seconds" and "hundredths of a second". TIME is based on the 24-hour clock; thus 2:41 pm, for example, would be expressed as 14410000. If referenced in a COBOL program, TIME is interpreted as if it had been described as an unsigned elementary numeric integer data item, 8 digits in length (PIC 9(8)). The minimum value for TIME is 00000000, the maximum value is 23595900. The last two digits are not supplied by the system, and are therefore always set to zero.

The following three formats of the ACCEPT statement are extensions from the X/Open Portability Guide. They allow access to environment variables and command lines. Access to command lines is meaningful only if the object program is executing in the POSIX subsystem available as of BS2000/OSD V2.0. Execution of the COBOL2000 compiler and of any programs generated by it under POSIX is described in the "COBOL2000 User Guide" [1].

Format 3

This supplies the current number of arguments in the command line.

```
ACCEPT identifier-1 [ FROM mnemonic-name-3 ]
    [END-ACCEPT]
```

Syntax rule

1. identifier-1 must refer to an elementary item that is described as an unsigned integer.
2. mnemonic-name-3 must be linked in the SPECIAL-NAMES paragraph with the implementor-name ARGUMENT-NUMBER.

Format 4

This supplies (consecutively) the contents of the arguments in the command line.

```
ACCEPT identifier-2 [ FROM mnemonic-name-4 ]

    [ON EXCEPTION imperative-statement-1 [NOT ON EXCEPTION imperative-statement-2]]

    [END-ACCEPT]
```

Syntax rules

1. identifier-2 must refer to an alphanumeric elementary item defined without the ANY LENGTH clause.
2. mnemonic-name-4 must be linked in the SPECIAL-NAMES paragraph with the implementor-name ARGUMENT-VALUE.
3. NOT ON EXCEPTION can only be specified if ON EXCEPTION is also specified.

Format 5

This supplies the contents of an environment variable or the contents of a specific argument from the command line. The name of the specified environment variable or the number of the specified argument in the command line must have been defined before by an appropriate DISPLAY statement.

```
ACCEPT identifier-2 [ FROM mnemonic-name-6 ]

    [ON EXCEPTION imperative-statement-1 [NOT ON EXCEPTION imperative-statement-2]]

    [END-ACCEPT]
```

Syntax rules

1. identifier-2 must refer to an alphanumeric elementary item defined without the ANY LENGTH clause.
2. mnemonic-name-6 must be linked in the SPECIAL-NAMES paragraph with the implementor-name ARGUMENT-VALUE or ENVIRONMENT-VALUE.
3. NOT ON EXCEPTION can only be specified if ON EXCEPTION is also specified.

> A detailed example illustrating access to command lines and environment variables can be found in chapter 13 of the "COBOL2000 User Guide" [1].

8.10.2 ADD statement

Function

The ADD statement causes two or more numeric operands to be summed and the result to be stored.

Format 1 of the ADD statement stores the sum in one of the operand items. More than one addition may be expressed by specifying more than one result item in the same ADD statement.

Format 2 of the ADD statement stores the sum in a separate result item.

Format 3 of the ADD statement adds the data items of one group item to the corresponding data items of another group item.

Format 1

```
ADD {identifier-1 | literal-1}... TO {identifier-2 [ROUNDED]}...
    [ON SIZE ERROR imperative-statement-1]
    [NOT ON SIZE ERROR imperative-statement-2]
    [END-ADD]
```

Syntax rules

1. Each identifier must refer to an elementary numeric item.
2. The composite of operands is determined by using all of the operands in a given statement and must not contain more than 31 digits (see "Arithmetic statements").
3. The values of the operands preceding the word TO are added together and the sum is added to the current value of identifier-2... The result is stored in identifier-2 ...

Additional rules are given under "Options in arithmetic statements" (Phrases in statements et seq.), where the ROUNDED phrase and the (NOT) ON SIZE ERROR phrase are described.

Example 8-21

Statement	PICTURE IS of result item	Calculation
ADD A, B TO C, D		A + B + C stored in C A + B + D stored in D
ADD A, B, C TO D	S9999V99	A + B + C + D stored in D as SnnnnVnn
ADD A, 14 TO C ROUNDED	99999	A + 14 + C stored in C as nnnnn; rounded if necessary

Format 2

```
ADD {identifier-1 | literal-1}... TO {identifier-2 | literal-2} GIVING {identifier-
3 [ROUNDED]}...
```

```
    [ON SIZE ERROR imperative-statement-1]

    [NOT ON SIZE ERROR imperative-statement-2]

    [END-ADD]
```

Syntax rules

1. Each identifier preceding the GIVING phrase must refer to an elementary numeric item.
2. identifier-3 may refer either to an elementary numeric item or to an elementary numericedited data item.
3. The composite of operands is determined by using all of the operands in a given statement, excluding the data items which follow the word GIVING, and must not contain more than 31 digits (see "Arithmetic statements").
4. The values of the operands preceding the word GIVING are added together, and the sum is stored as the new value of identifier-3.

Additional rules are given under "Options in arithmetic statements" (Phrases in statements et seq.), where the GIVING, ROUNDED, and (NOT) ON SIZE ERROR phrases are described.

Example 8-22

Statement	PICTURE IS of result item	Calculation
ADD A, B, C GIVING D.	9999.99	A + B + C stored in D as nnnn.nn
ADD A, B, 43.6 GIVING D ON SIZE ERROR GO TO O-FLOW END-ADD.	99V99	A + B + 43.6 stored in D. If the integer result is greater than 2 digits, the size error condition occurs and the GO TO statement specified in the SIZE ERROR phrase is executed.

Format 3

```
ADD {CORR | CORRESPONDING} identifier-1 TO identifier-2 [ROUNDED]

    [ON SIZE ERROR imperative-statement-1]

    [NOT ON SIZE ERROR imperative-statement-2]

    [END-ADD]
```

Syntax rules

1. Each identifier must refer to a group item.
2. The composite of operands is determined separately for each pair of corresponding data items, and must not contain more than 31 digits (see "Arithmetic statements").
3. Elementary items within the first operand (identifier-1) are added to the corresponding elementary items in the second operand (identifier-2 ...). The results are stored in the items of the second operand.

Additional rules are given under "Options in arithmetic statements" (Phrases in statements et seq.), where the CORRESPONDING, ROUNDED, and (NOT) ON SIZE ERROR phrases are described.

Example 8-23

Refer to the description of the CORRESPONDING phrase for an example of the use of this option (CORRESPONDING phrase).

8.10.3 ALLOCATE statement

Function

The ALLOCATE statement allocates dynamic storage.

Format 1 If storage is requested for a data description (data-name-1) for which the BASED clause is specified, the address of the requested storage is allocated to this data description.
If the RETURNING phrase is present, the data pointer identifier-2 also contains the address of the allocated storage.

Format 2 If storage is requested as a number of characters, the address of the allocated storage is returned in the data pointer identifier-2.

Format 1

```
ALLOCATE data-name-1
    [INITIALIZED] [RETURNING identifier-2]
```

Syntax rule

1. data-name-1 must be an elementary item which is specified with the BASED clause.
2. If identifier-2 is a type-specific data pointer, data-name-1 must also be type-based and the associated types must match.
3. If data-name-1 is strongly typed, identifier-2 must also be a type-specific pointer which is allocated the same type as data-name-1.
4. identifier-2 must reference an elementary item of the category "data pointer".

Format 2

```
ALLOCATE arithmetical-expression CHARACTERS
    [INITIALIZED] RETURNING identifier-2
```

Syntax rule

1. identifier-2 may not be a type-specific data pointer.
2. identifier-2 must reference an elementary item of the category "data pointer".

General rules for formats 1 and 2

1. arithmetical-expression defines the number of bytes in storage which are requested. The result of the evaluation of arithmetical-expression is placed in an internal data item which is defined with `PIC S9(9) BINARY` .
2. If the result of the evaluation is less than or equal to NULL, no storage is requested and identifier-2 contains the predefined value NULL.
3. With format 1 the size of the requested storage is determined by how much space is required for a data description data-name-1. If a data description which is subordinate to data-name-1 contains an OCCURS DEPENDING clause, the maximum length of the record is requested.

4. If the requested storage space is available, it is allocated. With format 1 the address of data-name-1 is set to the address of the allocated storage. The data pointer identifier-2 from the RETURNING phrase is set to the address of the allocated storage.
5. If the requested storage space is not available, the address of data-name-1 and/or the data pointer identifier-2 from the RETURNING phrase are set to NULL. The exception condition EC-STORAGE-NOT-AVAIL occurs. If the check for EC-STORAGE-NOT-AVAIL is activated, the associated exception condition is triggered and control is transferred to the corresponding USE procedure. After the return from the USE procedure, processing continues with the executable statement following the ALLOCATE statement.
6. The INITIALIZED phrase in format 1 causes an INITIALIZE statement INITIALIZE dataname-1 WITH FILLER ALL TO VALUE THEN TO DEFAULT to be executed for dataname-1.
7. If the INITIALIZED phrase is missing in format 1, an INITIALIZE statement INITIALIZE data-name-1 WITH FILLER REPLACING DATA-POINTER BY NULL, PROGRAM-POINTER BY NULL, OBJECT-REFERENCE BY NULL is executed.
8. The INITIALIZED phrase in format 2 causes the allocated storage to be initialized with binary NULLS.
9. If the INITIALIZED phrase is missing in format 2, the content of the allocated storage is undefined.
10. The storage remains allocated until it is explicitly released with a FREE statement or the program run is terminated.

8.10.4 ALTER statement

Function

The ALTER statement modifies one or more GO TO statements, thereby altering a predetermined sequence of operations.

Format

```
ALTER {procedure-name-1 TO [PROCEED TO] procedure-name-2}...
```

Syntax rules

1. procedure-name-1... must be names of paragraphs which contain only one sentence consisting of a GO TO statement without the DEPENDING phrase.
2. procedure-name-2... must be paragraph names or section names in the PROCEDURE DIVISION.
3. During the execution of the program, the ALTER statement modifies the GO TO statement specified under procedure-name-1... so that subsequent executions of the modified GO TO statement cause control to be transferred to procedure-name-2... (see section "GO TO statement").

General rules

1. A GO TO statement in a section whose segment number is greater than or equal to 50 must not be referenced by an ALTER statement in a section with a different segment number.
2. A GO TO statement in a section whose segment number is less than 50 may be referenced by an ALTER statement in any section, even if the GO TO statement thus referred to is contained in a program segment which has not yet been called for execution.

8.10.5 CALL statement

Function

The CALL statement passes control to a called program. Optionally, operands may be specified to enable the called program to access data of the calling program. The CALL statement (format 4) can also be used to execute BS2000 system commands from a COBOL program.

Format 1

```
CALL {identifier-1 | literal-1} [USING {[BY REFERENCE] {identifier-2|file-name-1}...
                                       | BY VALUE {identifier-5}...
                                       | BY CONTENT {identifier-2|literal-2}...
                                       }...]
  [RETURNING identifier-3]
  [ON OVERFLOW imperative-statement-1]
  [END-CALL]
```

Format 2

```
CALL {identifier-1 | literal-1} [USING {[BY REFERENCE] {identifier-2|file-name-1}...
                                       | BY VALUE {identifier-5}...
                                       | BY CONTENT {identifier-2|literal-2}...
                                       }... ]
  [RETURNING identifier-3]
  [ON EXCEPTION imperative-statement-1]
  [NOT ON EXCEPTION imperative-statement-2]
  [END-CALL]
```

Syntax rules for formats 1 and 2

1. literal-1 must be a alphanumeric literal. However, it may not be a figurative constant.

   If literal-1 is the program-name of an individual program or of the outermost containing program of a nested program, it must begin with an alphabetic character and may contain only uppercase letters and digits. The length of the name is dependent on the module format (see the "COBOL2000 User Guide" [1]).
   If literal-1 is the program-name of a contained program of a nested program, it must begin with an alphabetic character, may contain uppercase letters, lowercase letters and digits, and must not be longer than 30 characters.
2. identifier-1 must be defined as a program pointer or an alphanumeric data item. The contents of the alphanumeric data item must be a valid program name, as described under point 1. The contents of the program pointer are interpreted as the address of the entry point.
3. The USING phrase in a CALL statement may be supplied only if a USING phrase has been written either after the associated Procedure Division header or in the ENTRY statement of the called program. Each USING phrase must have the same number of operands, otherwise the result will be unpredictable.
4. Every identifier-1 specified in the USING phrase must have been defined in the FILE SECTION, WORKING-STORAGE SECTION, LOCAL-STORAGE SECTION, LINKAGE SECTION or SUB-SCHEMA SECTION. It may have level number 01 or level number 77. However, the compiler allows every level number except 88. In order to align elementary items with USAGE INDEX, BINARY, COMPUTATIONAL, COMPUTATIONAL-5,

COMPUTATIONAL-1, COMPUTATIONAL-2 in the LINKAGE SECTION, all 01-level items included are aligned on doubleword boundaries. Consequently, the user must ensure that these operands are aligned accordingly in the USING phrase.

5. file-name-1 is appropriate only if the called program is written in a language other than COBOL.
6. identifier-2 must not be a function-identifier.
7. identifier-3 must not be defined in the REPORT SECTION.
8. identifier-3 is a receiving item.
9. Object references, pointer and strongly typed data items may also be used as parameters. They may only be passed BY CONTENT as USING parameters.
10. identifier-5 must have been defined as a 2- or 4-byte data item with USAGE COMP-5 or as a 1-byte data item. If this is not the case, the result of the parameter transfer is undefined. This type of parameter transfer is not a good idea unless the called program was written in a language other than COBOL (see CALL statement).
11. literal-2 must be an alphanumeric or national literal or one of the figurative constants SPACE, LOW-VALUE or HIGH-VALUE.
12. identifier-2, identifier-3 and identifier-5 must not be defined with the ANY LENGTH clause.

General rules for formats 1 and 2

1. literal-1 or identifier-1 contains the name of the called program. The program which contains a CALL statement is the calling program. If the called program is a COBOL program, literal-1 or identifier-1 must contain the program-name specified in the PROGRAM-ID paragraph or in the ENTRY statement. If identifier-1 is defined as a program pointer, it contains the address of an entry point into a program.
2. When the CALL statement is executed, control is passed to the called program. When control is returned to the calling program, imperative-statement-2 (if specified) is executed and the program then branches to the end of the CALL statement. If NOT ON EXCEPTION is omitted, the program branches immediately to the end of the CALL statement.
3. If the called program is not available or already active during execution of the CALL statement, one of the following actions is executed:
   - If ON OVERFLOW or ON EXCEPTION is specified, control is passed to imperativestatement-1. After completion of imperative-statement-1, control is passed to the end of the CALL statement.
   - If ON OVERFLOW or ON EXCEPTION is not specified, an error message is issued and program execution is aborted.
4. If two programs in a run unit have the same name, then at least one of them must be a contained program within a nested program. A program having a multiply used program name can only be successfully called under the following conditions:
   - The called program is directly contained in the calling program.
   - The called program has the COMMON attribute and is called by the directly superordinate program or by one of the latter's sibling programs or their descendants.
   - The calling program is a contained program within a nested program and calls a separately compiled program of a run unit.
5. The data to be passed as parameters from the calling program to the called program are specified in the USING phrase of the CALL statement with identifier-2... .
   The number and sequence of the parameters must match the specifications in the USING phrase of the Procedure Division header or the ENTRY statement. Excepted here are the indices assigned to tables (INDEXED BY phrase): The indices in a calling program and a called program always point to separate indices.

6. If file-name-1 is specified in the list of the USING phrase, the starting address of the system file control block of that file is supplied to the called program.
7. BY CONTENT and BY REFERENCE may be used together. The phrase BY CONTENT or BY REFERENCE applies to all subsequent parameters until another BY CONTENT or BY REFERENCE phrase is encountered. If neither BY CONTENT nor BY REFERENCE is specified, the default is BY REFERENCE.
8. If a parameter is passed BY REFERENCE, it occupies the same memory location in the calling program and the called program. The description of the item in the called program must specify the same number of characters as the description of the corresponding item in the calling program.
9. If a parameter is passed BY CONTENT, the calling program makes a copy of the parameter and passes this copy BY REFERENCE.
   The data description of the corresponding parameter in the called program must be chosen as follows:
   - if identifier-4 is specified: the same as that of identifier-4
   - if figurative constants are specified: PIC X
   - if an alphanumeric literal is specified: PIC X(n). Repetition factor n may not exceed the length of the literal. In contrast with the behavior for other transfers, the literal at the end may not be filled with blanks to the length of the corresponding parameter from the subprogram.
   - if an national literal is specified: PIC N(n). Repetition factor n may not exceed the length of the literal. In contrast with the behavior for other transfers, the literal at the end may not be filled with blanks to the length of the corresponding parameter from the subprogram.
10. Specifying BY VALUE enables the direct, C-compliant transfer of values to C programs. If the parameter value “by value” is specified, this means that only the value of the parameter is passed to the called C program. The called program can access this value; it can also modify it, in which case the value remains unchanged in the COBOL program.
11. A called program may contain CALL statements, but must not execute any CALL statement that directly or indirectly calls the calling program via the standard entry point or an entry point generated by means of the ENTRY statement.
12. If the RETURNING phrase is defined, the result of the current program is placed in identifier-5.

Format 3

```
CALL { identifier-1 | literal-1 } AS { NESTED | program-prototype-name-1 }
       [USING { [BY REFERENCE] { identifier-2 | OMITTED }
              | [BY  CONTENT] { identifier-4 | arithmetic-expression-1 | literal-2 }
              | [BY  VALUE] { identifier-4 | arithmetic-expression-1 | literal-2 }
              }...]
[RETURNING identifier-3]
[ON EXCEPTION imperative-statement-1]
[NOT ON EXCEPTION imperative-statement-2]
[END-CALL]
```

Syntax rules

1. literal-1 must not be an alphanumeric literal. However, it may not be a figurative constant.

   If literal-1 is the program name of an individual program or the outermost program of a nested program, it must begin with an alphabetic character and can only contain uppercase letters and digits. The length of the name depends on the module format (see the "COBOL2000 User Guide" [1]).

If literal-1 is the program name of a program inside a nested program, it must begin with an alphabetic character, can contain uppercase and lowercase letters and digits and can have a length of up to 30 characters.

2. identifier-1 must be defined as a program pointer or an alphanumeric data item. As an alphanumeric data item, its value must be a valid program name, as described in 1. The content of the program pointer is interpreted as the address of the entry point.
3. The NESTED phrase may only be specified in a program definition.
4. The NESTED phrase may only be specified together with literal-1.
   literal-1 must be the same as the program-name specified in a PROGRAM-ID paragraph of a nested program.
5. If program-prototype-name-1 is specified, there must be an entry for this name in the REPOSITORY paragraph.
6. identifier-4 and any identifier specified in arithmetic-expression-1 is a sending operand.
7. If BY CONTENT or BY REFERENCE is specified for an argument, the BY REFERENCE phrase must be specified for the corresponding formal parameter in the PROCEDURE DIVISION header.
8. BY CONTENT must not be omitted when identifier-4 is an identifier that is permitted as a receiving operand.
9. If BY VALUE is specified for an argument, the BY VALUE phrase must be specified for the corresponding formal parameter in the PROCEDURE DIVISION header.
10. If identifier-4 or its corresponding formal parameter is specified with a BY VALUE phrase in the Procedure Division header, identifier-4 must be of class “numeric”, “object” or ”pointer”.
11. If OMITTED is specified or an argument is omitted, the OPTIONAL phrase must be specified for the corresponding formal parameter in the PROCEDURE DIVISION header.
12. identifier-2 and identifier-3 must not be defined using the ANY LENGTH clause.
13. If identifier-2 or the corresponding formal parameter is specified with a BY VALUE phrase then it may only be of the class “numeric”, “object” or “pointer”.
14. identifier-2 must reference an address identifier or elementary item defined in the File Section, Working-Storage Section, Local-Storage Section or Linkage Section.
15. If the BY REFERENCE phrase is specified or implied for identifier-2, which is not an address identifier, then identifier-2 represents both the sending and the receiving item.
    Otherwise identifier-2 is a sending item.

    Note:
    BY REFERENCE ADDRESS OF data-name is processed in the same way as BY CONTENT ADDRESS OF data-name.
16. identifier-3 must not be defined in the REPORT SECTION.
17. identifier-3 is a receiving item.
18. Conformity of parameters and return elements (see "Conformance for parameters and returning items") must be ensured.

General rules

In general, the rules for format 1 and 2 apply. The following rules apply in addition:

1. The BY REFERENCE, BY CONTENT and BY VALUE phrases refer **only** to the argument that follows the phrase immediately.
2. If an argument is specified without any of the keywords BY REFERENCE, BY CONTENT, or BY VALUE, that argument is handled as follows:

   a. BY REFERENCE is assumed if the BY REFERENCE phrase is specified or implied for the corresponding formal parameter in the PROCEDURE DIVISION header and if the argument is an identifier that is permitted as the receiving item.
   b. BY CONTENT is assumed if the BY REFERENCE phrase is specified or implied for the corresponding formal parameter and if the argument is a literal, an arithmetic expression or any other identifier that is not permitted as the receiving item.
   c. BY VALUE is assumed if the BY VALUE phrase is specified or implied for the corresponding formal parameter in the Procedure Division header.
3. An argument at the end of the USING list may also be omitted totally (i.e. the OMITTED phrase need not be specified for it) if all the following arguments are omitted as well (Note syntax rule 11).
4. If an OMITTED phrase is specified or a trailing argument is omitted, the omitted-argument condition for that argument must be true in the called program.
5. If an argument for which the omitted-argument condition is true is referenced in a called program (except in the omitted-argument condition), the behavior is undefined.
6. When the CALL statement is executed, control is passed to the called program. After control is returned by the called program, imperative-statement-2 is executed, if present, and a branch is made to the end of the CALL statement. If NOT ON EXCEPTION is missing, a branch is made to the end of the CALL statement.
7. If the called program is not available during execution of the CALL statement, or if it is already active without being recursive, one of the following actions is executed:
   a. If ON EXCEPTION is specified, controlo is passed to imperative-statement-1. After termination of imperative-statement-1, control passes to the end of the CALL statement.
   b. If ON EXCEPTION is not specified, program execution is aborted after an error message is output.

Format 4

```
CALL UPON SYSTEM USING { identifier-1 | literal-1 } [identifier-2]

   [STATUS identifier-3]

   [ON EXCEPTION imperative-statement-1]
   [NOT ON EXCEPTION imperative-statement-2]

       [END-CALL]
```

Syntax rules

1. identifier-1 and identifier-2 must be alphanumeric data items.
2. literal-1 must be an alphanumeric literal.
3. identifier-3 must be a numeric data item described with PIC S9(8) SYNC and USAGE COMP, USAGE COMP-5 or USAGE BINARY.
4. identifier-2 may not be a function identifier.

General rules

1. literal-1 or identifier-1 contains the BS2000 command to be executed.
   A slash (/) can be specified in front of the command. Lower-case letters are *not* converted into upper-case.

2. identifier-2 specifies an area for recording system output.
   This response area should be large enough to record the complete output of the specified BS2000 command.
   If identifier-2 is variable in length and contains the DEPENDING ON item, the maximum length of identifier-2 is used.
3. If identifier-2 is not specified or its current length is 0, the output is written to SYSOUT.
4. If an error occurs during execution of the command, the response area contains the system's error message text, provided it is large enough.
5. identifier-3 can be used to specify a status item in which specific values show the result of the execution of the command. The following values can occur.

00	Command executed successfully
04	The command was executed, but one or more records could not be entered because the response area is too small.
30	Error during command execution; no more information available
34	This command may not be specified in a program .
40	The current length of identifier-1 is invalid (<= 0 or > 32767 bytes).
41	The current length of identifier-2 is invalid (< 0).
90	There is not enough main memory available to execute the command; workaround: specify smaller areas (identifier-1 or identifier-2)

6. If an error occurs while the CALL statement is being executed, one of the following actions is performed:
   a. If ON EXCEPTION is specified, imperative-statement-1 is executed. When imperative-statement-1 has been completed, the program continues at the end of the CALL statement.
   b. If ON EXCEPTION is not executed, the program sequence is resumed without an error message.
7. If the CALL statement is executed successfully, one of the following actions is performed:
   a. If NOT ON EXCEPTION is specified, imperative-statement-2 is executed.
   b. If NOT ON EXCEPTION is not specified, a branch is made to the end of the CALL statement.

i The system output consists of variable records with length items and special characters for feed control, etc.; a record is usually represented by several lines on the screen. Only complete records are entered in the response area; the rest of the response area which is not used is deleted. For information on the structures and further processing of the response area, see User Guide "Executive Macros - CMD macro" [12].

Example 8-24

of the use of the CALL statement in the format CALL identifier-1

Main program:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MAIN.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
    TERMINAL IS T.
DATA DIVISION.
WORKING-STORAGE SECTION.
01   CALL-OPERAND PIC X(8) VALUE SPACE.
01   TABLE-FUNCTION.
     02  FUNCTION-1 PIC X(8) VALUE "ADDREC".
     02  FILLER     PIC X(72) VALUE SPACE.
     02  FUNCTION-2 PIC X(8) VALUE "DELREC".
     02  FILLER     PIC X(72) VALUE SPACE.
     02  FUNCTION-3 PIC X(8) VALUE "CHGREC".
     02  FILLER     PIC X(72) VALUE SPACE.
01   RECORD-NUMBER PIC 9(8).
PROCEDURE DIVISION.
P1 SECTION.
PMAIN.
    PERFORM UNTIL CALL-OPERAND = FUNCTION-1 OR FUNCTION-2
                                            OR FUNCTION-3
       DISPLAY "Please enter desired function"
              UPON T
       DISPLAY TABLE-FUNCTION UPON T
       ACCEPT CALL-OPERAND FROM T
    END-PERFORM
    PERFORM UNTIL RECORD-NUMBER NUMERIC
       DISPLAY "Please enter record number"
               "(numeric, 8 digits)" UPON T
       ACCEPT RECORD-NUMBER FROM T
    END-PERFORM
    CALL CALL-OPERAND USING RECORD-NUMBER
    END-CALL
    STOP RUN.
```

Subprogram "ADDREC":

```
IDENTIFICATION DIVISION.
PROGRAM-ID. ADDREC.
ENVIRONMENT DIVISION.
DATA DIVISION.
...
LINKAGE SECTION.
01 RECORD-NUMBER PIC 9(8).
PROCEDURE DIVISION USING RECORD-NUMBER.
    ...
    EXIT PROGRAM.
```

Subprogram "DELREC":

```
IDENTIFICATION DIVISION.
PROGRAM-ID. DELREC.
ENVIRONMENT DIVISION.
DATA DIVISION.
...
LINKAGE SECTION.
01  RECORD-NUMBER PIC 9(8).
PROCEDURE DIVISION USING RECORD-NUMBER.
    ...
    EXIT PROGRAM.
```

Subprogram "CHGREC":

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CHGREC.
ENVIRONMENT DIVISION.
DATA DIVISION.
...
LINKAGE SECTION.
01  RECORD-NUMBER PIC 9(8).
PROCEDURE DIVISION USING RECORD-NUMBER.
    ...
    EXIT PROGRAM.
```

Example 8-25

for the use of CALL ... USING BY VALUE

Main program:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MAIN.
ENVIRONMENT DIVISION.
    CONFIGURATION SECTION.
    SPECIAL-NAMES.
        TERMINAL IS T.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  C  PIC  9(4) USAGE COMP-5 VALUE 1.
01  D  PIC  9(9) USAGE COMP-5 VALUE 1.
01  E  PIC  S9(4) USAGE COMP-5 VALUE -1.
01  F  PIC  S9(9) USAGE COMP-5 VALUE -1.
01  RTC  PIC S9(10) SIGN IS LEADING SEPARATE.
PROCEDURE DIVISION.
1ST SECTION.
1.
CALL "C1" USING BY VALUE C, D.
    MOVE RETURN-CODE TO RTC.
    DISPLAY "RETURN-CODE = " RTC UPON T.
    CALL "D1" USING BY VALUE E, F.
    MOVE RETURN-CODE TO RTC.
    DISPLAY "RETURN-CODE = " RTC UPON T.
    MOVE 0 TO RETURN-CODE.
    STOP RUN.
```

Subprogram C1:

```
long C1(unsigned short c, unsigned long d)
{
long ret_val;
if (c && d)
   ret_val = 1;
else
   ret_val = -1;
return ret_val;
}
```

Subprogram D1:

```
long D1(signed short c, signed long d)
{
long ret_val;
if (c && d)
   ret_val = 1;
else
   ret_val = -1;
return ret_val;
}
```

Example 8-26

File name assignment with the SET-FILE-LINK command in dialogs and status queries:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  CMD.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
    TERMINAL IS T.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  LINKFILE.                                                  (1)
  02  FILLER   PIC X(30) VALUE "ADD-FILE-LINK LINK=XXX,F-NAME=".
  02  FILNAM   PIC X(54).
01  RTC        PIC S9(8) SYNC BINARY.                          (2)
01  RTC-ED     PIC Z(6)99.
01  TFT-CMD    PIC X(40) VALUE "SHOW-FILE-LINK".
01  RESP       PIC X(2000) VALUE ALL SPACE.                    (3)
PROCEDURE DIVISION.
MAIN SECTION.
PARA.
    DISPLAY "PLEASE ENTER FILE NAME X(54)" UPON T.
    ACCEPT FILNAM FROM T.
    CALL UPON SYSTEM USING LINKFILE
                           STATUS RTC
      ON EXCEPTION
         MOVE RTC TO RTC-ED
         DISPLAY "ERROR DURING COMMAND CAL, STATUS= "
                     RTC-ED UPON T
    END-CALL
    CALL UPON SYSTEM USING TFT-CMD
                           RESP
    END-CALL.
    DISPLAY "RESPONSE AREA RESP" RESP UPON T.
FIN.
    STOP RUN.
```

(1) Description of identifier-1 with substructures for BS2000 system command and file name.

(2) Description of identifier-3 (status item RTC)

(3) Description of identifier-2 (response area RESP)

8.10.6 CANCEL statement

Function

The CANCEL statement causes the specified program to be set to its initial state the next time it is called.

Format

```
CANCEL { identifier-1 | literal-1 } ...
```

Syntax rules

1. literal-1 must be an alphanumeric literal and must be a valid program name. However, it may not be a figurative constant.

   If literal-1 is the program-name of a separately compiled program or of the outermost containing program of a nested program, it must begin with an alphabetic character and may contain only uppercase letters and digits. The length of the name depends on the module format.
   If literal-1 is the program-name of a contained program of a nested program, it must begin with an alphabetic character, may contain uppercase letters, lowercase letters and digits, and must not be longer than 30 characters.
2. identifier-1 must be defined as an alphanumeric data item so that its value may be a valid program name, as described under point 1.

General rules

1. literal-1 or the content of identifier-1 specifies the program which is to be set to its initial state.
2. Successful execution of the CANCEL statement closes the files in the specified program. If the program is called again in the same run unit or in a nested program after successful execution of a CANCEL statement, this program is in its initial state.
3. The program name specified as literal-1 or contained in identifier-1 must be unique within the run unit or the nested program unless it is a program name that may be used multiply under certain conditions (see section "CALL statement", general rule 4).

   The program name must not be the same as the first 7 characters of the program name in the PROGRAM-ID paragraph of the program that contains the CANCEL statement.
4. Called programs may contain CANCEL statements, but a called program may not execute any CANCEL statement that either directly or indirectly references the calling program.
5. The logical connection to a program which is initialized with a CANCEL statement is established again only if this program is subsequently called again with a CALL statement.
6. A CANCEL statement has no effect if one of the following situations exists:
   - The program concerned is already in the initial state because it has not yet been called in the active run unit or in the nested program.
   - The program concerned has already been initialized with a CANCEL statement.
   - The program concerned or (with nested programs) a superordinate program has the INITIAL attribute..

   In these cases, the program continues at the next executable statement after the CANCEL statement.
7. During execution of a CANCEL statement, an implicit CLOSE statement (without any optional phrases) is executed for each open internal file assigned to the program. Any USE procedures specified for these files are not executed.

8. A CANCEL statement may only reference such programs for which the call is permitted within the call hierarchy.
9. If an explicit or implicit CANCEL statement is executed, then all contained programs in the program referenced by the CANCEL statement are also cancelled.

Example 8-27

Main program:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MAIN.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 ERROR-CODE PIC 9.
   88  O-K VALUE 0.
  ...
PROCEDURE DIVISION.
P1 SECTION.
MAIN.
    PERFORM WITH TEST AFTER UNTIL O-K
      CALL "UPROG1" USING ERROR-CODE
      IF NOT O-K
      THEN
        CANCEL "UPROG1"
      END-IF
    END-PERFORM
    STOP RUN.
```

Subprogram:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. UPROG1.
ENVIRONMENT DIVISION.
DATA DIVISION.
   ...
LINKAGE SECTION.
01 ERROR-CODE  PIC 9.
    88  O-K  VALUE 0.
PROCEDURE DIVISION USING ERROR-CODE.
    ...
    IF INTERN-ERROR
    THEN
      CONTINUE
    ELSE
      SET O-K TO TRUE
    END-IF
    EXIT PROGRAM.
```

UPROG1 is called repeatedly until the value 0 is returned. As long as a value other than 0 is returned, UPROG1 is set to its initial state and the loop is continued.

8.10.7 CLOSE statement

Function

For **sequential file organization:** The CLOSE statement terminates the processing of input/output reels, units and files, with optional rewind and/or lock functions.

For **relative and indexed file organization:** The CLOSE statement terminates the processing of files, with optional lock functions.

Format 1 for sequential file organization

```
CLOSE {file-name [ [{ REEL | UNIT }] [FOR REMOVAL]
                 | WITH { NO REWIND | LOCK }
                 ]
      } ...
```

Format 2 for relative and indexed file organization

```
CLOSE {file-name-1 [WITH LOCK]}...
```

Syntax rules

1. The files referenced in the CLOSE statement need not have the same organization or access type.
2. The terms 'REEL' and 'UNIT' are synonymous and completely interchangeable within a CLOSE statement. The handling of sequential mass storage files is logically equivalent to the handling of files on magnetic tape or similar sequential media.
3. In the case of line sequential files only the specification WITH LOCK is permitted.

General rules

1. A CLOSE statement may be issued only for a file which is open.
2. The meanings of the options in the CLOSE statement are given below:

REEL	The current tape reel is to be closed.
UNIT	The current mass-storage unit is to be closed.
NO REWIND	No repositioning to the beginning of the reel is to be performed after a tape file is closed.
LOCK	The file cannot be reopened in the same program run.
REMOVAL	The current reel of a magnetic tape file is to be unloaded.

3. In order to show the effect of various CLOSE statements as applied to various storage media, all input/output files are divided into the following categories:
   a. **UNIT RECORD volume file (unit record file):**
   a file assigned to an input or output medium for which the concepts rewinding, units, and reels have no meaning.
   b. **Sequential single-volume file:**
   a sequential file that is entirely contained on one volume. There may be more than one file on this volume.

c. **Sequential multi-volume file:**
a sequential file contained on more than one volume.

4. The results of executing each CLOSE statement for each type of file are summarized in the following table:

CLOSE statement	File type			
	Sequentially organized files			Relative/indexed
	Unit record	Sequential single-volume	Sequential multi-volume	Single-/multi-volume
CLOSE	C	C, G	C, G, A	C
CLOSE WITH LOCK	C, E	C,G,E	C, G, E, A	C,E
CLOSE WITH NOREWIND	C, H	C, B	C, B, A	—
CLOSE FOR REMOVAL	C, H	G, D	F, G, D	—
CLOSE REEL/UNIT	J	F, G	F, G	—
CLOSE REEL/UNIT FOR REMOVAL	J	F, G, D	F, G, D	—

Table 26: Relationship between types of files and the formats of the CLOSE statement

The definitions of the symbols used in the table are given below. When the definition of the symbol depends on whether the file is an input or output file, alternate definitions are given; otherwise, a definition applies to files opened as INPUT, OUTPUT and I-O.

A **Previous volumes unaffected**

All volumes of the file processed prior to the current volume were processed according to standard volume switching procedures, except those volumes controlled by a prior CLOSE REEL/UNIT statement.

B **No rewind of current reel**

The current volume is positioned at the logical end of the file on the volume.

C **Standard close file**

For files opened with the INPUT or I-O phrase:

if the file is positioned at its end and a LABEL RECORDS clause was supplied, then (if a USE procedure is present) the standard trailer label procedure and user trailer label procedure are executed. The order in which these two routines are executed is specified by the USE procedure. The standard system closing procedures are then performed.

If the file is positioned at its end, but a LABEL RECORDS clause was not supplied, then only the standard system closing procedures are performed.

If the file is positioned other than at its end, only the standard system closing proce-dures are performed. Even if USE procedures are supplied, no label processing will take place in this case. (An INPUT or I-O file is considered to be at its end if the imperative statement in the AT END phrase of the READ statement, if entered, has been executed, and no CLOSE statement has been executed.)

For files opened with the OUTPUT phrase:

if a LABEL RECORDS clause was specified for this file, the standard ending label procedures and the user ending label procedures (provided such were specified by a USE procedure) are performed. The order in which these two procedures are executed is defined by the USE statement. The standard system closing procedures are then performed.

If label records are not specified for the file, only the standard system closing proce-dures are performed.

D **Unload current reel**

The REMOVAL option, when specified, causes the current reel of a magnetic tape file to be unloaded. Further processing of the file requires an appropriate continuation reel. After executing a CLOSE statement without the REEL/UNIT phrase specified, and an OPEN statement for this file, this reel may be processed again.

E **Standard file lock**

If the LOCK phrase is used, the file cannot be reopened in the same program run.

F **Standard close volume**

The following operations are performed for files opened with the INPUT or I-O phrase:

- When the current volume is the last or only volume for the file:

  Disk storage files

  Processing of the current data block is terminated. The next READ statement makes the first record of the following data block available.

  Tape files

  Processing of the reel is terminated. The next READ statement leads to an at end condition.
- When other volumes exist for the file:
  a. Volume swapping.
  b. USE procedures for processing standard and user header labels (if specified by a USE statement). The order in which these procedures are executed is defined by the USE statement.
  c. The next record on the new volume is made available for the subsequent read operation.

For files opened as OUTPUT, the following operations are carried out:

- *Disk storage files*

  Processing of the current data block is terminated. The next WRITE statement creates a new data block.
- *Tape files*
  a. Standard and user trailer label procedures (if specified by a USE statement). The order in which these procedures are executed is defined by the USE statement.
  b. Volume swapping.
  c. Standard and user header label procedures (if specified by a USE statement). The order in which these procedures are executed is defined by the USE statement.

G **Rewind**

The current volume is positioned at its beginning (for a magnetic tape file, this is the beginning of the reel; for disk storage files, this is the beginning of the file concerned).

H I/O status 07 is set.

The optional phrases are ignored.

J I/O status 07 is set.

The CLOSE statement is ignored.

5. The execution of a CLOSE statement causes the contents of the data item specified in the FILE STATUS clause (if such a clause exists) to be updated (see section "FILE STATUS clause").
6. After successful execution of a CLOSE statement, the record area assigned to the file is no longer accessible. The unsuccessful execution of such a CLOSE statement leaves the availability of the record area undefined.
7. If an optional input file does not exist, no end-of-file processing and no reel/volume processing is executed.
8. If more than one file-name has been specified, the effect is the same as if a separate CLOSE statement had been written for each file-name.
9. All files that are still open on completion of a task are closed.

8.10.8 COMPUTE statement

Function

The COMPUTE statement is used to assign the value of a data item, literal, or arithmetic expression to a data item.

Format

```
COMPUTE {identifier-1 [ROUNDED]}... = { identifier-2
                                      | literal-1
                                      | arithmetic-expression }

       [ON SIZE ERROR imperative-statement-1]

       [NOT ON SIZE ERROR imperative-statement-2]

       [END-COMPUTE]
```

Syntax rules

1. identifier-1... must refer to an elementary numeric item or an elementary numeric-edited data item.
2. identifier-2 must refer to an elementary numeric item.
3. The arithmetic-expression specified in the COMPUTE statement permits the use of any meaningful combination of identifiers (which must satisfy the general rules for datanames in simple arithmetic operations), literals, and arithmetic operands; if necessary, they may also be in parentheses (see section "Arithmetic expressions").
4. If identifier-2 or literal-1 is specified, the value of identifier-1 is set equal to the value of identifier-2 or literal-1.
5. When an arithmetic-expression is used, the value of that arithmetic-expression is first calculated and then stored as the new value of identifier-1...
6. The COMPUTE statement allows the user to combine arithmetic operations without the restrictions on the composite of operands and/or receiving data items which are imposed by the ADD and SUBTRACT statements.
7. Up to 50 data-names may be specified in a COMPUTE statement.

Additional rules are given under "Options in arithmetic statements" (Phrases in statements et seq.), where the ROUNDED and (NOT) ON SIZE ERROR phrases are described.

Example 8-28

Statement	Calculation
COMPUTE A = (B + C) / D - E.	The value of the expression (B + C) / D - E is assigned to A. The precedence rules for evaluating expressions apply when calculating values.
COMPUTE A = 2.	The value 2 is assigned to A.

8.10.9 CONTINUE statement

The CONTINUE statement is a no operation statement. It indicates that no executable statement is present. Processing is continued with the next executable statement.

Format

```
CONTINUE
```

Syntax rule

1. The CONTINUE statement may be used anywhere a conditional statement or an imperative-statement may be used.

General rule

1. The CONTINUE statement has no effect on the execution of the program.

Example 8-29

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CONT1.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
    TERMINAL IS T.
DATA DIVISION.
WORKING-STORAGE SECTION.
77 N PIC 9.
77 K PIC 9(3).
77 Z PIC 9(6) VALUE ALL ZERO.
77 E PIC 9(3).
PROCEDURE DIVISION.
PROC SECTION.
INPUT-PAR.
    DISPLAY "Enter upper limit N" UPON T.
    ACCEPT N FROM T.
    IF N NUMERIC
    THEN
     CONTINUE
    ELSE
     DISPLAY "Incorrect entry" UPON T
     PERFORM INPUT-PAR
    END-IF.
COMPUTATION.
    PERFORM WITH TEST BEFORE VARYING K FROM 1 BY 1 UNTIL K > N
      COMPUTE E = K ** 3
      ADD E TO Z
    END-PERFORM
    DISPLAY "Result = " Z UPON T.
FINISH-PAR.
    STOP RUN.
```

The effect of CONTINUE is to make the IF statement syntactically correct even though the THEN branch does not contain an executable statement.

Example 8-30

```
READ INPUT-DATEI AT END CONTINUE.
```

AT END is used in order to avoid program abortion at the end of the file; CONTINUE specifies the unconditional statement which is required by the statement syntax, even though nothing is to be done at this point in the program.

8.10.10 DELETE statement

Applies to **relative and indexed file organization**

Function

A DELETE statement logically removes a record from a disk storage file.

Format

```
DELETE file-name RECORD
  [INVALID KEY imperative-statement-1]
  [NOT INVALID KEY imperative-statement-2]
  [END-DELETE]
```

Syntax rules

1. The INVALID KEY phrase must not be specified for a DELETE statement which references a file in sequential access mode.
2. If a file is not in sequential access mode and no applicable USE procedure has been specified, the INVALID KEY phrase is mandatory.

General rules

1. The file referenced in the DELETE statement must be open in I-O mode during the execution of this statement (see also "OPEN statement").
2. After successful execution of the DELETE statement, the identified record is deleted from the file.
3. Execution of a DELETE statement does not affect the contents of the record area associated with the file or the content of the data item referenced by the data-name specified in the DEPENDING ON phrase of the RECORD clause.
4. For a file in sequential access mode, the last input/output statement entered prior to the DELETE statement must be a successfully completed READ statement. The RELATIVE KEY (for **relative file organization**) or RECORD KEY (for **indexed file organization**) must not be changed between reading and deletion. In this case, the DELETE statement causes the record read by the preceding READ statement to be logically removed from that file.
5. After execution of the DELETE statement, the contents of the data item specified in the FILE STATUS clause for the file are updated (see "FILE STATUS clause").
6. Continuation of the execution of the DELETE statement depends on whether the INVALID KEY or NOT INVALID KEY phrase is specified (see "Invalid key condition").

8.10.11 DISPLAY statement

Function

Format 1 is used to output small quantities of data.

Format 2 is used to access a POSIX command line.

Format 3 are used to access a BS2000 or POSIX environment variable.
Format 4

Format 1

```
DISPLAY { literal-1 | identifier-1 }... [ UPON mnemonic-name ] [WITH NO ADVANCING]
```

Syntax rules

1. If literal-1 is numeric, it must be an unsigned integer.

   identifier-1 must not be of the class national, pointer or object.
2. mnemonic-name must be specified in the SPECIAL-NAMES paragraph and be associated with one of the following implementor-names:
   CONSOLE, PRINTER, PRINTER01 - PRINTER99, SYSOPT, TERMINAL, job-variable-name (BS2000 job variable).
3. The mnemonic-name for SYSOPT, TERMINAL, CONSOLE, PRINTER and PRINTER01 - PRINTER99 specifies the system file into which the data is to be written.

   SYSOPT specifies the system file with the same name.

   TERMINAL specifies the system file SYSOUT.

   CONSOLE specifies the system operator console.

   PRINTER specifies the system file SYSLST, and PRINTER01 - PRINTER99 refer to the system files SYSLST01 - SYSLST99.
4. If the UPON phrase is omitted, the data is written by default to the logical output file SYSLST. Data can also be written to the logical output file SYSOUT by means of an appropriate compiler directive (see the "COBOL2000 User Guide" [1]).
5. identifier-1 must not be defined with the ANY LENGTH clause.

General rules

1. literal-1 or identifier-1 serves to specify the operands in the order they are to be output. If necessary, the contents of the data item specified by "identifier" are converted to external formats according to the following rules:
   - Internal decimal and binary items are converted to external decimal data items.
   - Internal floating-point data items are converted to external floating-point data items.

   No other data items require conversion.

   If one of the operands is the figurative constant ALL literal, the literal is output once.

   If one of the operands is a figurative constant (except ALL literal), it is output with length 1.
2. A maximum logical record size is assumed for each device. These sizes are listed in table 27.

Device	Maximum record size
CONSOLE	180 characters
PRINTER PRINTER01_PRINTER99	132 characters + 1 control byte
SYSOPT	80 characters: 72 data bytes; bytes 73-80 contain the first 8 bytes of the PROGRAM-ID name.
TERMINAL	8192 characters

Table 27: Maximum logical record size for the DISPLAY statement

3. When a DISPLAY statement contains more than one operand, the contents of the specified operands and literals are displayed adjacent to each other, from left to right.
   Zero-length items are not displayed. If all operands are zero-length items, the DISPLAY statement has no effect.
4. The compiler treats the NO ADVANCING phrase as a comment.
5. For output to printer, the following entries cause a line feed: DISPLAY, WRITE and WRITE AFTER ADVANCING. A WRITE statement without ADVANCING phrase and a WRITE statement with BEFORE ADVANCING phrase causes the printer to space after printing. Therefore, mixed use of DISPLAY and WRITE statements on the same device within the same program may cause two or more lines to overprint. Overprinting is not possible on laser printers.
6. The maximum record length for job variables is 256 characters.
   If the total number of characters in the operands exceeds the maximum record length, the record will be truncated to the maximum length.
7. When a job variable is used as a monitoring job variable (MONJV), the system protects the first 128 bytes (system portion) of this job variable against write access. Therefore, only that portion of a record that begins at position 129 will be written from position 129 of the monitoring job variable. In all other respects, general rule 4 applies for monitoring job variables.

For further information see the "COBOL2000 User Guide" [1].

Example 8-31

```
SPECIAL-NAMES.
     TERMINAL IS SPECIAL-OUTPUT.
     ...
PROCEDURE DIVISION.
     ...
     DISPLAY OUTPUT-MESSAGE UPON SPECIAL-OUTPUT.
```

Here, the mnemonic-name SPECIAL-OUTPUT is associated with the implementor-name TERMINAL in the SPECIAL-NAMES paragraph. The DISPLAY statement writes the current contents of OUTPUT-MESSAGE on SYSOUT.

Example 8-32

```
DISPLAY "Hello world".
```

Since the UPON phrase is omitted, the literal "Hello world" is written to the logical output device SYSLST. If the compiler directive COMOPT REDIRECT-ACCEPT-DISPLAY=YES (in SDF: ACCEPT-DISPLAY-ASSGN=*TERMINAL) is specified, the literal is written to the output file SYSOUT (see the "COBOL2000 User Guide" [1]).

The following three formats of the DISPLAY statement are extensions from the X/Open Portability Guide. They allow access to environment variables and command lines. Access to command lines is meaningful only if the object program is executing in the POSIX subsystem available as of BS2000/OSD V2.0. Execution of the COBOL2000 compiler and of any programs generated by it under POSIX is described in the "COBOL2000 User Guide" [1].

Format 2

This format sets the number of the argument in the command line which is subsequently accessed via an ACCEPT statement.

```
DISPLAY { identifier-3 | integer-1 } UPON mnemonic-name-3 [END-DISPLAY]
```

Syntax rules

1. identifier-3 must refer to an elementary item that is described as an unsigned integer.
2. integer-1 must be unsigned.
3. mnemonic-name-3 must be linked in the SPECIAL-NAMES paragraph with the implementor-name ARGUMENT-NUMBER.

Format 3

This format sets the name of the environment variable which is subsequently accessed by an ACCEPT or DISPLAY statement.

```
DISPLAY { identifier-4 | literal-1 } UPON mnemonic-name-5 [END-DISPLAY]
```

Syntax rules

1. identifier-4 must refer to an alphanumeric elementary item which is not defined with the ANY LENGTH clause.
2. literal-1 must be an alphanumeric literal.
3. mnemonic-name-5 must be linked in the SPECIAL-NAMES paragraph with the implementor-name ENVIRONMENT-NAME.

Format 4

This format writes to the environment variable specified previously in a format-3 DISPLAY statement.

```
DISPLAY {identifier-2 | literal-2 } UPON mnemonic-name-6
   [ON EXCEPTION imperative-statement-1]
   [NOT ON EXCEPTION imperative-statement-2]
   [END-DISPLAY]
```

Syntax rules

1. identifier-2 must refer to an alphanumeric elementary item which is not defined with the ANY LENGTH clause.
2. literal-2 must be an alphanumeric literal.
3. mnemonic-name-6 must be linked in the SPECIAL-NAMES paragraph with the implementor-name ENVIRONMENT-VALUE.
4. NOT ON EXCEPTION can only be specified if ON EXCEPTION is also specified.

A detailed example of accessing command lines and environment variables can be found in chapter 12 of the COBOL User Guide [1].

8.10.12 DIVIDE statement

Function

The DIVIDE statement is used to divide one numeric operand by another and store the result.

Format 1 of the DIVIDE statement stores the quotient in the dividend item.

Format 2 of the DIVIDE statement stores the quotient in more than one separate result item.

Format 3 of the DIVIDE statement uses the GIVING phrase for storing the quotient and generates
the division remainder by means of the REMAINDER phrase.

Format 1

```
DIVIDE { identifier-1 | literal-1 } INTO {identifier-2 [ROUNDED]}...
    [ON SIZE ERROR imperative-statement-1]
    [NOT ON SIZE ERROR imperative-statement-2]
    [END-DIVIDE]
```

Syntax rules

1. Each identifier must refer to an elementary numeric item.
2. The value of identifier-2 is divided by the value of identifier-1 or literal-1. The quotient then replaces the current value of identifier-2 and so on.
3. The maximum size of the quotient is 31 decimal digits.
4. Division by zero always results in overflow (SIZE ERROR).
5. In the case of division with ON SIZE ERROR, it is still possible for a DIVIDE-ERROR to occur since no test is made for quotient overflow (only for division by zero).

Additional rules are given under "Phrases in statements", where the ROUNDED, (NOT) ON SIZE ERROR, and GIVING phrases are described.

Example 8-33

Statement	PICTURE of result item	Calculation
DIVIDE A INTO B	9(4)V9(2)	B / A stored as nnnnVnn in B

Format 2

```
DIVIDE {identifier-1 | literal-1} {INTO | BY} {identifier-2 | literal-2}
        GIVING {identifier-3 [ROUNDED]}...
       [ON SIZE ERROR imperative-statement-1]
       [NOT ON SIZE ERROR imperative-statement-2]
```

```
[END-DIVIDE]
```

Syntax rules

1. identifier-1 or identifier-2 must refer to an elementary data item.
2. identifier-3... may refer to an elementary numeric item or to an elementary numericedited item.
3. When the INTO phrase is used, the value of identifier-2 or literal-2 is divided by the value of identifier-1 or literal-1; when the BY phrase is used, the value of identifier-1 or literal-1 is divided by the value of identifier-2 or literal-2. The quotient is stored in identifier-3... .
4. The maximum size of the quotient is 31 decimal digits.
5. Division by zero always results in overflow (SIZE ERROR).
6. In the case of division with ON SIZE ERROR, it is still possible for a divide error to occur since no test is made for quotient overflow (only for division by zero).

Additional rules are given under "Phrases in statements" where the ROUNDED, (NOT) ON SIZE ERROR, and GIVING phrases are described.

Example 8-34

Statement	PICTURE IS of result item (C):	Calculation
DIVIDE A INTO B GIVING C ROUNDED	S999V99 for C	B / A stored in C as nnnVnn after rounding, if necessary
DIVIDE A BY B, GIVING C, D ROUNDED	9(5) for C 9(4) for D	A / B stored in C as nnnnn, in D as nnnn, after rounding the rightmost character,if necessary.

Format 3

```
DIVIDE {identifier-1 | literal-1} {INTO | BY} {identifier-2 | literal-2} GIVING
 identifier-3 [ROUNDED]

        REMAINDER identifier-4

       [ON SIZE ERROR imperative-statement-1]

       [NOT ON SIZE ERROR imperative-statement-2]

       [END-DIVIDE]
```

Syntax rules

1. identifier-1 or identifier-2 must refer to an elementary numeric data item.
2. identifier-3 or identifier-4 may refer to an elementary numeric data item or to an elementary numeric-edited data item.

3. When INTO is used, the value of identifier-2 or literal-2 is divided by the value of identifier-a or literal-1. When the BY phrase is used, the value of identifier-1 or literal-1 is divided by the value of identifier-2 or literal-2. The quotient is stored in identifier-3.
4. When the REMAINDER phrase is used, the remainder of the division is stored in identifier-4.
   The remainder is calculated by subtracting the product of the quotient and the divisor from the dividend.
   If identifier-3 is defined as an elementary numeric-edited item, then the remainder is calculated by using an intermediate item for the quotient, containing the value in an unedited format.
   If both the ROUNDED and the REMAINDER phrases are supplied, then the remainder is calculated by using an intermediate item for the quotient, containing the quotient of the DIVIDE statement in a truncated rather than rounded format.
5. If ON SIZE ERROR is specified, and overflow occurs in the quotient, then the remainder will not be calculated. In this case, the contents of the data items referenced by identifier-3 and identifier-4 are therefore unchanged. If overflow occurs in the remainder, the value of the data item referenced by identifier4 is not changed.
6. The precision of the data item required for the REMAINDER phrase (identifier-4) is determined by the calculations described above. Appropriate decimal point alignment and truncation (rather than rounding) are performed as necessary for the contents of the data item referenced by identifier-4.
7. The maximum size of the quotient is 31 decimal digits.
8. Division by zero always results in overflow (SIZE ERROR).

Additional rules are given under "Phrases in statements", where the ROUNDED, (NOT) ON SIZE ERROR, and GIVING phrases are described.

Example 8-35

Statement	Result item (C) PICTURE IS:	Calculation
DIVIDE A BY B, GIVING C REMAINDER D	9(5) for C9(2) for D	A / B stored in C as nnnnn, the remainder, e.g. A - C * B, stored in D as nn.

8.10.13 ENTRY statement

Function

The ENTRY statement is used in a COBOL subprogram of a run unit for specifying the entry point by which the subprogram may be called (as opposed to the standard entry point supplied by the Procedure Division header). Optionally, data-names may be specified, if data is to be transferred from the calling program as a parameter.

Format

```
ENTRY literal [USING {data-name-1}... ].
```

Syntax rules

1. literal must be an alphanumeric literal.

   literal must be a valid program-name, i.e. it must begin with an alphabetic character, must not contain any characters other than letters and digits, and have a maximum length of 7 characters.
2. The program-name specified by the literal must be unique in the programs which are linked to form a run unit. Also, it must not be the same as the first 7 characters of the program-name entered in the PROGRAM-ID paragraph of the program containing this ENTRY statement.
3. The USING phrase may be written only if the associated CALL statement in the calling program also contains a USING phrase. The number of operands in the corresponding USING phrases must be identical; otherwise the result will be unpredictable.
4. Each data-name-1... supplied in the USING phrase of the ENTRY statement must be defined as a data item in the LINKAGE SECTION of the program containing this ENTRY statement; its level number must be either 01 or 77.
5. The ENTRY statement must not be used in the programs of a nested program.

General rules

1. The ENTRY statement determines the entry point in the called program. The name of the entry point is specified by the literal. A branch to this entry point is effected by a CALL statement in another program which references this entry point.
2. The USING phrase has the effect that, at execution time, data-name-1 in the ENTRY statement of the called program and identifier-1 in the USING phrase of the CALL statement in the calling program refer to the same set of data, which is equally available both to the called program and to the calling program. The names need not be the same.

   In the USING phrase of the ENTRY statement in the called program, a data-name may occur only once; in the USING phrase of the CALL statement, however, the same identifier-1 may be specified more than once.
3. In the called program, the operands of the USING phrase are treated according to the data descriptions given in the LINKAGE SECTION.
4. An ENTRY statement may be executed only once (upon entry to the program). The remainder of the program may contain no further ENTRY statements.
5. If a data item is passed in the CALL statement as a BY CONTENT parameter, the value of this data item is transferred to a memory area which has the characteristics specified for identifier-2 in the CALL statement

before the CALL statement is executed. The data type and length of each parameter in the BY CONTENT phrase of the CALL statement must be the same as those of the corresponding parameter in the USING phrase in the Procedure Division header.

8.10.14 EVALUATE statement

Function

The EVALUATE statement describes a multi-branch, multi-join structure. It can cause multiple conditions to be evaluated. The subsequent action of the program depends on the results of these evaluations.

Format

```
EVALUATE {identifier-1  | literal-1  | expression-1  | TRUE | FALSE }
 [ALSO {identifier-2 | literal-2  | expression-2 | TRUE | FALSE} ]...
{ { WHEN    { ANY | condition-1 |  subcondition-1  | TRUE | FALSE
              | [NOT]   { {identifier-3 | literal-3 | arithm-expression-1}
                          [ {THROUGH | THRU}
                            {identifier-4 | literal-4 | arithm-expression-2}
                          ]
                        }
            }
     [ ALSO { ANY | condition-2 | subcondition-2 | TRUE | FALSE
              | [ NOT ] { {identifier-5 | literal-5 | arithm-expression-3}
                          [ { THROUGH | THRU }
                            {identifier-6 | literal-6 | arithm-expression-4}
                          ]
                        }
            }
     ]...
   }...
   imperative-statement-1
 } ...
[WHEN OTHER imperative-statement-2]
[END-EVALUATE]
```

Syntax rules

1. The operands or the words TRUE and FALSE which appear before the first WHEN phrase of the EVALUATE statement are referred to individually as selection subjects and collectively, for all those specified, as the set of selection subjects.

2. The operands or the words TRUE, FALSE and ANY which appear in a WHEN phrase of an EVALUATE statement are referred to individually as selection objects and collectively, for all those specified in a single WHEN phrase, as the set of selection objects.
3. The words THROUGH and THRU are equivalent.
4. Two operands connected by a THROUGH phrase must be of the same class. The two operands thus connected constitute a single selection object. The connected operands may not be of the class “object” or the class “pointer”.
5. The number of selection objects within each set of selection objects must be equal to the number of selection subjects.
6. Each selection object within a set of selection objects must correspond to the selection subject having the same ordinal position within the set of selection subjects according to the following rules:
   a. Identifiers, literals, or arithmetic expressions appearing within a selection object must be valid operands (according to the rules for relation conditions) for
   comparison with the corresponding operand in the set of selection subjects.
   b. condition-1, condition-2, or the words TRUE or FALSE appearing as a selection object must correspond to a conditional expression or the words TRUE or FALSE
   in the set of selection subjects.
   c. The word ANY may correspond to a selection subject of any type.
   d. subcondition-1 or subcondition-2 may not start with an arithmetic operator.
   e. subcondition-1 or subcondition-2 appearing within a selection object must correspond to an identifier, a literal or an arithmetic expression in the set of selection subjects. They must be specified in such a way that a valid, simple condition can be created by inserting the corresponding subject in front of the subcondition.

General rules

1. The execution of the EVALUATE statement operates as if each selection subject and selection object were evaluated and assigned a numeric or non-numeric value, a range of numeric or non-numeric values, or a truth value. These values are determined as follows:
   a. Any selection subject specified by identifier-1, identifier-2, and any selection object specified by identifier-3, identifier-5, without either the NOT or the THROUGH phrase, are assigned the value and class of the data item referenced by the identifier.
   b. Any selection subject specified by literal-1, literal-2, and any selection object specified by literal-3, literal-5, without either the NOT or the THROUGH phrase, are assigned the value and class of the specified literal. If literal-3, literal-5, is the figurative constant ZERO, it is assigned the class of the corresponding selection subject.
   c. Any selection subject in which expression-1, expression-2, is specified as an arithmetic expression and any selection object, without either the NOT or the THROUGH phrase, in which arithmetic-expression-1, arithmetic-expression-3, is specified are assigned a numeric value according to the rules for evaluating an arithmetic expression.
   d. Any selection subject in which expression-1, expression-2, is specified as a conditional expression and any selection object in which condition-1, condition-2, is specified are assigned a truth value according to the rules for evaluating conditional expressions.
   e. Any selection subject or any selection object specified by the words TRUE or FALSE is assigned a truth value. The truth value "true" is assigned to those items specified with the word TRUE, and the truth value "false" is assigned to those items specified with the word FALSE.
   f. Any selection specified by the word ANY is not further evaluated.

g. If the THROUGH phrase is specified for a selection object, without the NOT phrase, the range of values includes all permissible values of the selection subject that are greater than or equal to the first operand and less than or equal to the second operand according to the rules for comparison.

h. If the NOT phrase is specified for a selection object, the values assigned to that item are all permissible values of the selection subject not equal to the value, or not included in the range of values, that would have been assigned to the item had the NOT phrase not been specified.

i. If a subcondition is specified for a selection object, it is assigned the truth value of the condition resulting from the addition of the object to the result of the evaluation of the related subject.

2. The execution of the EVALUATE statement then proceeds as if the values assigned to the selection subjects and selection objects were compared to determine if any WHEN phrase satisfies the set of selection subjects. This comparison proceeds as follows:

   a. Each selection object within the set of selection objects for the first WHEN phrase is compared with the selection subject having the same ordinal position within the set of selection subjects. One of the following conditions must be satisfied if the comparison is to be satisfied:

      - If the items being compared are assigned numeric or non-numeric values, or a range of numeric or non-numeric values, the comparison is satisfied if the value, or one of the range of values, assigned to the selection object is equal to the value assigned to the selection subject according to the rules for comparison (see section "Relation condition").
      - If the object is a subcondition, the comparison is satisfied if the object was assigned the truth value "true".
      - If the items being compared are assigned truth values, the comparison is satisfied if the items are assigned the identical truth value.
      - If the selection object being compared is specified by the word ANY, the comparison is always satisfied regardless of the value of the selection subject.

   b. If the above comparison is satisfied for every selection object within the set of selection objects being compared, the WHEN phrase containing that set of selection objects is selected as the one satisfying the set of selection subjects.

   c. If the above comparison is not satisfied for one or more selection objects within the set of selection objects being compared, that set of selection objects does not satisfy the set of selection subjects.

   d. This procedure is repeated for subsequent sets of selection objects, in the order of their appearance in the compilation unit, until either a WHEN phrase satisfying the set of selection subjects is selected or until all sets of selection objects are exhausted.

3. After the comparison operation is completed, execution of the EVALUATE statement proceeds as follows:

   a. If a WHEN phrase is selected, execution continues with the first imperative-statement following the selected WHEN phrase. There is a branch to END-EVALUATE after the last imperative-statement is executed.

   b. If no WHEN phrase is selected and a WHEN OTHER phrase is specified, execution continues with the imperative-statement of the WHEN OTHER phrase.

   c. An EVALUATE statement is terminated when one of the following conditions is satisfied:

      - imperative-statement-1 of the selected WHEN phrase is executed,
      - imperative-statement-2 is executed,
      - no WHEN phrase is selected and no WHEN OTHER phrase is specified.

Example 8-36

```
IDENTIFICATION DIVISION.
PROGRAM-ID. EVAL1.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
    TERMINAL IS T.
DATA DIVISION.
WORKING-STORAGE SECTION.
77 A PIC 9.
77 B PIC 9.
77 C PIC 9.
77 D PIC 9.
PROCEDURE DIVISION.
PROC SECTION.
DIALOG.
    DISPLAY "Enter value for A" UPON T.
    ACCEPT A FROM T.
    DISPLAY "Enter value for B" UPON T.
    ACCEPT B FROM T.
    DISPLAY "Enter value for C" UPON T.
    ACCEPT C FROM T.
    DISPLAY "Enter value for D" UPON T.
    ACCEPT D FROM T.
TEST1.
    EVALUATE A + B ALSO C + D
    WHEN 5 ALSO 5
      DISPLAY "Values correct" UPON T
    WHEN OTHER
      DISPLAY "Values incorrect" UPON T
    END-EVALUATE.
FINISH-PAR.
    STOP RUN.
```

Example 8-37

```
IDENTIFICATION DIVISION.
PROGRAM-ID. BSP.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
    TERMINAL IS T.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  TYPE-OF-ORDER PIC 9.
    88 ON-SITE       VALUE 1.
    88 IN-WRITING    VALUE 2 THRU 4.
01  CUSTOMER-TYPE PIC X.
    88 PRIVATE        VALUE "1".
    88 BUSINESS       VALUE "2".
01  WEIGHT        PIC 9999.
01 SHIPPING-MODE  PIC 9.
   88 PICK-UP     VALUE 1.
   88 MAIL        VALUE 2.
   88 RAIL        VALUE 3.
```

```
   88 UPS         VALUE 4.

PROCEDURE DIVISION.
PROC SECTION.
DIALOG.
    DISPLAY "Enter type-of-order" UPON T.
    DISPLAY "  on-site = 1, in-writing = 2-4 " UPON T.
    ACCEPT TYPE-OF-ORDER FROM T.
    DISPLAY "Customer-type" UPON T.
    DISPLAY "Business = 2 , Private = 1 " UPON T.
    ACCEPT CUSTOMER-TYPE FROM T.
    DISPLAY "Enter weight" UPON T.
    ACCEPT WEIGHT FROM T.
DETERMINATION-SHIPPING-MODE.
    EVALUATE TRUE ALSO TRUE ALSO TRUE
       WHEN PRIVATE ALSO ON-SITE ALSO ANY
       WHEN BUSINESS ALSO ON-SITE ALSO ANY
           SET PICK-UP TO TRUE
       WHEN PRIVATE ALSO IN-WRITING ALSO WEIGHT < 5
           SET MAIL TO TRUE
       WHEN BUSINESS ALSO IN-WRITING ALSO WEIGHT < 10
           SET UPS TO TRUE
       WHEN OTHER SET RAIL TO TRUE
    END-EVALUATE.
OUTPUT-PAR.
    DISPLAY "Shipping-mode = " SHIPPING-MODE UPON T.
    STOP RUN.
```

Explanation

The selection objects ON-SITE, IN-WRITING, PRIVATE, BUSINESS (condition-names) are evaluated with respect to their truth value. The selection subjects are represented by the three TRUEs; all three selection subjects (= set of selection subjects) have the truth value "true". A set of selection objects satisfies the condition of the set of selection subjects when all selection objects in the set (except those specified by ANY) are assigned the truth value "true" (a WHEN phrase corresponds to a set of selection objects). The statement which follows this WHEN phrase will then be executed. If none of the specified sets of selection objects satisfies the comparison, the statement of the WHEN OTHER phrase is executed. The ANY phrase must be used in the first two WHEN phrases, because only two condition-names are tested for their truth value in these WHEN phrases, whereas three condition-names are tested for their truth value in the other WHEN phrases, and the number of selection objects within the set of selection objects must correspond to the number of selection subjects.

8.10.15 EXIT statement

Function

The EXIT statement provides a general exit at the end of a series of procedures.

Format

```
EXIT.
```

Syntax rules

1. The EXIT statement must be preceded by a paragraph-name. It must be the only statement in the paragraph.
2. The EXIT statement is used only to assign a procedure-name to a given point in the program. It has no other effect on the execution of the program.

General rules

1. The EXIT statement, when supplied at the end of a series of procedures, enables the normal execution of that sequence of procedures to be interrupted, passing control directly to the end of the procedure sequence.
2. If control reaches an "EXIT paragraph" and no associated PERFORM or USE statement is active, then control passes to the first sentence of the next paragraph.

Example 8-38

```
PROCEDURE DIVISION.
    ...
    PERFORM X-PAR THRU Y-PAR.
    ...
X-PAR.
    ...
    IF A IS ZERO, GO TO Y-PAR.
    ...
Y-PAR.
    EXIT.
Z-PAR.
    ...
```

Here, the "EXIT paragraph" is the last procedure covered by the PERFORM statement. If the value of A is zero, then the GO TO statement interrupts the normal flow of execution of the statements ranging from X-PAR to Y-PAR, and passes control directly to the end of the range of procedures specified by the PERFORM statement. After this, program execution resumes with the next statement after PERFORM.

8.10.16 EXIT METHOD statement

The EXIT METHOD statement identifies the logical end of a called method.

Format

```
EXIT METHOD
```

Syntax rule

1. The EXIT METHOD statement may only be specified in the PROCEDURE DIVISION of a method.

General rule

1. The execution of an EXIT METHOD statement causes the method to terminate, and control to return to the calling program which continues processing after the INVOKE statement.

8.10.17 EXIT PARAGRAPH statement

The EXIT PARAGRAPH statement results in a branch to an assumed CONTINUE statement which is located at the end of the current paragraph.

Format

```
EXIT PARAGRAPH
```

Syntax rule

1. The EXIT PARAGRAPH statement can only be specified within a paragraph.

8.10.18 EXIT PERFORM statement

Function

The EXIT PERFORM statement makes it possible to branch to the end of the PERFORM statement or to a repetition of the loop from an in-line PERFORM statement.

Format

```
EXIT {[TO TEST OF] PERFORM | PERFORM CYCLE}
```

Syntax rules

1. An EXIT [TO TEST OF] PERFORM statement can only be specified within an in-line PERFORM.
2. An EXIT TO TEST OF PERFORM statement can only refer to format 2, 3, or 4 PERFORM statements (see section "PERFORM statement").
3. EXIT PERFORM CYCLE is a synonym for EXIT TO TEST OF PERFORM.

General rules

1. The associated in-line PERFORM statement is exited during an EXIT PERFORM.
2. Depending on the format of the PERFORM statement, different branches are made as the result of EXIT TO TEST OF PERFORM:
   a. In a format 2 PERFORM, control passes to the test of "end of loop".
   b. In a format 3 PERFORM, control passes to the test of "UNTIL condition".
   c. When TEST AFTER is specified in format 4, control passes to the test of "UNTIL condition". If the condition is satisfied, the PERFORM statement is terminated. If the condition is not satisfied, augmentation is carried out.
   d. When TEST BEFORE is specified in format 4, control is passed to the increment counter. The test of "UNTIL condition" then follows.

Example 8-39

```
IDENTIFICATION DIVISION.
     PROGRAM-ID. EXITP.
     ENVIRONMENT DIVISION.
     CONFIGURATION SECTION.
     SPECIAL-NAMES.
        TERMINAL IS T.
     DATA DIVISION.
     WORKING-STORAGE SECTION.
     01  INPUTA          PIC 99.
     01  INPUT-STATUS    PIC X VALUE LOW-VALUE.
       88  INPUT-FINISH  VALUE HIGH-VALUE.
     01  AMOUNT          PIC 9(3).
     01  COUNTER         PIC 99.
     01  AVERAGE         PIC Z9.9(2).
     PROCEDURE DIVISION.
     PROC SECTION.
     COMPUTATION.
         MOVE 0 TO COUNTER AMOUNT
```

```
       DISPLAY "Calculation of the average of numbers" UPON T
       PERFORM WITH TEST AFTER UNTIL INPUT-FINISH
         DISPLAY "Input numbers (2 digits, end at input 00)" UPON T
         ACCEPT INPUTA FROM T
         IF INPUTA IS NOT NUMERIC
         THEN
           DISPLAY "Input is not numeric or not with 2 digits!" UPON T
           EXIT TO TEST OF PERFORM
         END-IF
         IF INPUTA = 0
         THEN
           SET INPUT-FINISH TO TRUE
           EXIT TO TEST OF PERFORM
         END-IF
         ADD 1 TO COUNTER
         ADD INPUTA TO AMOUNT
      END-PERFORM
      IF COUNTER > 0
      THEN
        COMPUTE AVERAGE ROUNDED = AMOUNT / COUNTER
        DISPLAY "Average = " AVERAGE UPON T
      ELSE
        DISPLAY "No calculation of average performed" UPON T
      END-IF
      STOP RUN.
```

The program calculates the average of numbers that have been input at the terminal. End criterion is the input of 00; invalid input is reported.

8.10.19 EXIT PROGRAM statement

Function

The EXIT PROGRAM statement marks the dynamic end of a called program.

Format

```
EXIT PROGRAM
```

Syntax rules

1. If an EXIT PROGRAM statement is one of several imperative-statements within asentence, it must be the last statement in this sentence.
2. The EXIT PROGRAM must not be used in a global USE procedure.

General rules

1. Execution of the EXIT PROGRAM statement in a called program causes a transfer of control back to the calling statement. An EXIT PROGRAM statement in a program that was not called as a subprogram has no effect.
2. The program state of the calling program remains unchanged and is identical with the state which existed when the CALL statement was executed in the calling program. This does not apply to data which was passed to the called program with the CALL statement and which has been modified by this program.
3. The EXIT PROGRAM statement in a program with the INITIAL attribute has the same effect as a CANCEL statement for this program.
4. In the called program, the control mechanisms for all PERFORM statements are initialized.

8.10.20 EXIT SECTION statement

The EXIT SECTION statement results in a branch to an assumed CONTINUE statement which is located at the end of the last paragraph of the current section (SECTION).

Format

```
EXIT SECTION
```

Syntax rule

1. The EXIT SECTION statement may only be specified within a section (SECTION).

8.10.21 FREE statement

Function

The FREE statement releases storage which was requested beforehand with an ALLOCATE statement (see ALLOCATE statement).

Format

```
FREE {data-name-1}...
```

Syntax rule

1. data-name-1 must be of the category "data pointer".

General rules

The FREE statement is executed as follows:

1. If the data pointer data-name-1 points to the start of the storage which was requested beforehand with the ALLOCATE statement, this storage is released and the data pointer data-name-1 is set to NULL. The length of the released storage is the same as that which was requested with ALLOCATE. The content of any data item within the released storage area is undefined.
2. If the data pointer data-name-1 contains the predefined address NULL, no storage is released.
3. In all other cases the exception condition EC-STORAGE-NOT-ALLOC occurs. data-name-1 is not modified. If the check for EC-STORAGE-NOT-ALLOC is activated, the associated exception condition is triggered and the corresponding USE procedure is branched to if it is available. If the check of the exception condition is not activated or if the USE procedure is executed with RESUME AT NEXT STATEMENT, the program run is continued in accordance with rule 4.

   > The FREE statement can lead to a crash if the check of the exception condition EC-STORAGE-NOT-ALLOC is deactivated.

4. If more than one data-name-1 is specified, this applies as if a separate FREE statement had been specified for each data-name-1 in the specified order. If the exception condition EC-STORAGE-NOT-ALLOC occurs as a result of one of these implicit FREE statements, the program run (as required after a return from the relevant USE procedure) is continued with the next implicit FREE statement, if there is one.

8.10.22 GOBACK statement

Function

The GOBACK statement identifies the logical end of a program.

Format

```
GOBACK
```

Syntax rule

1. The GOBACK statement must not be used in a global USE procedure or a method.

General rules

1. The GOBACK statement has the same effect as the following sequence:
   EXIT PROGRAM
   STOP RUN
2. For GOBACK the same rules therefore apply as for EXIT PROGRAM and STOP RUN.

8.10.23 GO TO statement

Function

The GO TO statement is used to transfer control from one part of the Procedure Division to another.

Format 1 of the GO TO statement passes control to a specific procedure.

Format 2 of the GO TO statement passes control to one of a series of specified procedures, depending on the value of a data item.

Format 1

```
GO TO [ procedure-name ]
```

Syntax rules

1. If the GO TO statement appears in a consecutive sequence of imperative statements within a sentence, it must appear as the last statement within that sequence.
2. A GO TO statement referenced by an ALTER statement must be preceded by a paragraph-name, and it must be the only statement in this paragraph (see section "ALTER statement").
3. A GO TO statement without specification of procedure-name must be preceded by a paragraph-name, and it must be the only statement in this paragraph.

General rules

1. When a GO TO statement is executed, control is transferred to procedure-name.
2. If procedure-name is not specified, an ALTER statement, referring to this GO TO statement, must be executed prior to the execution of this GO TO statement.
3. A GO TO statement in which procedure-name is not specified must be supplied with a procedure-name destination by the ALTER statement. Otherwise, the program will terminate with error message 9142.

Example 8-40

```
GO TO END-ROUTINE.
       ...
END-ROUTINE.
    CLOSE CARD-FILE, PRINTER-FILE.
    STOP RUN.
```

In this example, the GO TO statement transfers control to the procedure named END-ROUTINE; the CLOSE statement is executed immediately after the GO TO statement.

Format 2

```
GO TO {procedure-name-1}...

      DEPENDING ON identifier
```

Syntax rule

1. identifier is the name of an elementary numeric data item described as an integer. The USAGE must be either DISPLAY, COMPUTATIONAL, COMPUTATIONAL-5, BINARY, COMPUTATIONAL-3 or PACKED-DECIMAL.

General rule

1. When a GO TO statement is executed, control is transferred to procedure-name-1..., depending on the value of the identifier, which may be 1, 2, ... n.

   If the identifier has any value other than 1, 2, ... n, no transfer of control takes place, and processing continues with the next statement in the normal sequence for execution (n represents the specified number of procedure-names).

Example 8-41

```
77  A PICTURE 9.
       ...
    MOVE 3 TO A.
       ...
    GO TO X-PAR Y-PAR Z-PAR DEPENDING ON A.
```

Since the value of A is 3 when the GO TO statement is executed, control is transferred to Z-PAR, the third procedure-name in the series.

8.10.24 IF statement

Function

The IF statement causes a condition to be evaluated (see under section "Conditions"). The subsequent action of the program depends upon whether the condition is true or false.

Format 1

```
IF condition THEN statement-1 [ELSE statement-2]
    END-IF
```

Format 2

```
IF condition THEN {statement-1 | NEXT SENTENCE} [ELSE {statement-2 | NEXT SENTENCE}]
```

Syntax rules

1. statement-1 and statement-2 may consist of one or more imperative statements and/or one conditional statement.
2. An IF statement with the NEXT SENTENCE phrase must not be contained within statement-1 or statement-2 of a format 1 IF statement which is immediately terminated with END-IF.

General rules

1. statement-1 and/or statement-2 may contain an IF statement. In this case the IF statement is said to be nested.
2. The scope of the IF statement may be terminated by any of the following:
   a. An END-IF phrase at the same level of nesting.
   b. A separator period.
   c. If nested, by an ELSE phrase associated with an IF statement at a higher level of nesting.
3. IF statements within IF statements may be considered as paired IF, ELSE, and END-IF combinations, proceeding from left to right. Thus, any ELSE or END-IF encountered is considered to apply to the immediately preceding IF that has not been already paired with an ELSE or END-IF.
4. When an IF statement is executed, the following transfers of control occur:
   a. If the condition is true and statement-1 is specified, control is transferred to the first statement of statement-1 and execution continues according to the rules for each statement specified in statement-1. If a procedure branching or conditional statement is executed which causes an explicit transfer of control, control is explicitly transferred in accordance with the rules of that statement. Upon completion of the execution of statement-1, the ELSE phrase, if specified, is ignored and control passes to the end of the IF statement.
   b. If the condition is true and the NEXT SENTENCE phrase is specified instead of statement-1, the ELSE phrase, if specified, is ignored and control passes to the next executable sentence.
   c. If the condition is false and statement-2 is specified, statement-1 or its surrogate NEXT SENTENCE is ignored, control is transferred to the first statement of statement-2, and execution continues according to the rules for each statement specified in >statement-2. If a procedure branching or conditional statement is

executed which causes an explicit transfer of control, control is explicitly transferred in accordance with the rules of that statement. Upon completion of the execution of statement-2, control passes to the end of the IF statement.

d. If the condition is false and the ELSE phrase is not specified, statement-1 is ignored and control passes to the end of the IF statement.

e. If the condition is false and the ELSE NEXT SENTENCE phrase is specified, statement-1 is ignored and control passes to the next executable sentence.

Example 8-42

```
IF A = B
THEN
  statement-1
END-IF
statement-2.
```

If A = B is true, statement-1 and statement-2 are executed.

If A = B is false, only statement-2 is executed.

Example 8-43

```
IF A = B
THEN
  statement-1
ELSE
  statement-2
END-IF
statement-3.
```

If A = B is true, statement-1 and statement-3 are executed.

If A = B is false, statement-2 and statement-3 are executed.

Example 8-44

```
IF A = B
THEN
  CONTINUE
ELSE
  statement-1
END-IF
statement-2.
```

If A = B is true, statement-2 is executed.

If A = B is false, statement-1 and statement-2 are executed.

Example 8-45

```
|-> IF MALE
|   |-> IF MARRIED
|   |      ADD 1 TO MALES-MARRIED
|   |-> ELSE
|   |   |-> IF DIVORCED
|   |   |      ADD 1 TO MALES-DIVORCED
|   |   |-> ELSE
|   |   |      ADD 1 TO MALES-SINGLE
|   |   |-> END-IF
|   |-> END-IF
|-> ELSE
|      ADD 1 TO FEMALE
|-> END-IF
    next statement
```

This is an example of the structure of a nested IF statement. The arrows are used to indicate the IF, ELSE and END-IF assignments.

8.10.25 INITIALIZE statement

Function

The INITIALIZE statement enables selected data items to be supplied with specific values.

Format

```
INITIALIZE {identifier-1} ... [WITH FILLER]

    [{ ALL | category-name...} TO VALUE  ]

    [ THEN REPLACING {category-name DATA BY {identifier-2 | literal-1}}... ]

[THEN TO DEFAULT ]
```

category-name stands for:

```
{  ALPHABETIC
 | ALPHANUMERIC
 | ALPHANUMERIC-EDITED
 | DATA-POINTER
 | NATIONAL
 | NUMERIC
 | NUMERIC-EDITED
 | OBJECT-REFERENCE
 | PROGRAM-POINTER
}
```

Syntax rules

1. identifier-1 is the receiving item of the INITIALIZE statement. literal-1 and identifier-2 are sending items.
2. The following applies for the sending items:
   - If DATA-POINTER is named in the REPLACING phrase, identifier-2 must be a valid sending item of a SET statement whose receiving item belongs to the category "data pointer".
   - If PROGRAM-POINTER is named in the REPLACING phrase, identifier-2 must be a valid sending item of a SET statement whose receiving item belongs to the category "program pointer".
   - If OBJECT-REFERENCE is named in the REPLACING phrase, identifier-2 must be a valid sending item of a SET statement whose receiving item belongs to the category "object reference".
   - Any other data category specified in the REPLACING phrase must be permitted as a data category for a receiving item in a MOVE statement in which identifier-2 or literal-1 represent the sending field.
3. Each data category may only be named once in the VALUE phrase and in the REPLACING phrase.
4. The data description entry of identifier-1 or a data item subordinate to identifier-1 may not contain the DEPENDING phrase of the OCCURS clause.
5. An index data item may not occur as the operand of an INITIALIZE statement.
6. The data description entry of identifier-1 may not contain the RENAMES clause.
7. If identifier-1 is a table element or if it contains tables, the VALUE phrases for the table elements are ignored. However, if the INITIALIZE statement contains REPLACING or DEFAULT phrases, these are effective instead.

General rules

1. If identifier-1 is a national group, it is processed like a group. If identifier-2 is a national group, it is processed like an elementary item.
2. The keywords in category-name correspond to the data categories (see "data categories" in section "Concept of computer-independent data description"). The word ALL in the VALUE phrase means the same as if every category from category-name had been specified.
3. If more than one identifier-1 is specified, this has the same effect as if a separate INITIALIZE statement had been issued in the specified sequence for each identifier-1.
   If the exceptional condition EC-DATA-CONVERSION arises from one of these implicit INITIALIZE statements, the program run is stopped after a USE procedure which is exited with RESUME AT NEXT STATEMENT has been executed, and it is resumed at the next implicit INITIALIZE statement, if one is issued.
4. Regardless of whether identifier-1 represents an elementary item or a group item, all move operations are performed as though a sequence of MOVE or SET statements had been specified, each of them with an elementary item as receiving item. The receiving item of these implicit statements are determined in rule 3, the sending items in rule 4.

   Initialization is performed as follows:

   - If the category is "data pointer", "program pointer" or "object reference", then identifier-2 acts as the sending item of an implicit SET statement whose receiving item in each case is an elementary receiving item that is subordinate to identifier-1.
   - If the category is not "data pointer", "program pointer" or "object reference", then identifier-2 or literal-1 acts as the sending item of an implicit MOVE statement whose receiving item in each case is an elementary receiving item that is subordinate to identifier-1.
5. The receiving item in each implicit MOVE or SET statement is determined by the use of the following steps in the defined order:
   a. To determine the elementary receiving items the following elementary items are first of all excluded:
      i. Index data items.
      ii. Data items that are subordinate to an identifier-1 and whose data description entry contains the REDEFINES clause, and data items that are subordinate to such an item.
      iii. Data items that are subordinate to an identifier-1 and whose data description entry contains the RENAMES clause.
      iv. Elementary FILLER data items if the WITH FILLER phrase is omitted.

      Note:
      The data description entry of identifier-1 itself, however, may contain the REDEFINES clause or be subordinate to a data item with a REDEFINES clause.
   b. An elementary item is a potential receiving item if:
      i. identifier-1 is an elementary item,
      ii. or if it is subordinate to identifier-1. If the elementary item is a table element, it is a potential receiving item each time it occurs.
   c. In fact each potential receiving item is really a receiving field if at least one of the following conditions applies in this order:
      i. The INITIALIZE statement contains the VALUE phrase, the category of the elementary item is explicitly or implicitly contained in the VALUE phrase and one of the following conditions is satisfied:
         1. The category of the elementary item is "data pointer", "object reference" or "program pointer", or

2. a VALUE clause (defined by format 1) is contained in the data description of the elementary item.

ii. The INITIALIZE statement contains the REPLACING phrase and the category of the elementary item is one of the categories from the REPLACING phrase, or

iii. the INITIALIZE statement contains the DEFAULT phrase, or

iv. The INITIALIZE statement contains neither the VALUE phrase nor the REPLACING phrase.

The first condition which applies for an elementary item takes effect - the subsequent conditions are then no longer evaluated for this elementary item.

6. The following rules apply when determining the elementary sending items:

a. If the receiving item qualifies itself as a result of the VALUE phrase, the following applies:

i. If the receiving item is of the category "data pointer" or "program pointer", the sending item is the predefined address NULL.

ii. If the receiving item is of the category "object reference", the sending item is the predefined object reference NULL.

iii. In all other cases the sending item is the literal which is specified in the VALUE clause in the definition of the receiving item (see the section "VALUE clause" format 1 or format 3).

b. If the receiving item qualifies iteself as a result of the REPLACING phrase, the sending item is literal-1 or identifier-2 which are allocated to the category of the receiving item in the REPLACING phrarse.

c. If neither rule a) nor rule b) apply for the receiving item, the value of the sending item depends on the category of the receiving item as shown in the table below:

Category of the receiving item	**Send operand**
alphabetic	Figurative constant alphanumeric SPACES
alphanumeric	Figurative constant alphanumeric SPACES
alphanumeric-edited	Figurative constant alphanumeric SPACES
national	Figurative constant national SPACES
numeric	Figurative constant ZEROES
numeric-edited	Figurative constant ZEROES
data-pointer	Predefined address NULL
program-pointer	Predefined address NULL
object-reference	Predefined object reference NULL

7. The elementary items represented by identifier-1 are initialized from left to right in the order in which they occur in the INITIALIZE statement. If identifier-1 is a group, the relevant elementary itemswithin this group are initialized in the order in which they were defined in the group.

8. If identifier-1 and identifier-2 occupy the same storage space, the result of the execution of this statement will be undefined (see "Overlapping operands").

Example 8-46

```
IDENTIFICATION DIVISION.
PROGRAM-ID. INIT1.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
    TERMINAL IS T.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  WAGE-RATE.
    02  SURNAME    PIC X(30).
    02  NAME       PIC X(30).
    02  ADDRESS    PIC X(30).
    02  DATE-OF-BIRTH.
        03  BDAY   PIC 99.
        03  MONTH  PIC 99.
        03  YEAR   PIC 99.
    02  HIRING-DATE.
        03  HDAY    PIC 99.
        03  MONTH   PIC 99.
        03  YEAR    PIC 99.
    02  NO-OF-HOURS   PIC 9(3).
    02  HOURLY-RATE   PIC 9(2)V99.
PROCEDURE DIVISION.
MAIN SECTION.
P1.
    INITIALIZE WAGE-RATE.
    DISPLAY WAGE-RATE UPON T.
    STOP RUN.
```

The statement INITIALIZE WAGE-RATE means:

```
MOVE SPACE TO SURNAME NAME ADDRESS
MOVE ZERO TO BDAY OF DATE-OF-BIRTH
             MONTH OF DATE-OF-BIRTH
             YEAR OF DATE-OF-BIRTH
             HDAY OF HIRING-DATE
             MONTH OF HIRING-DATE
             YEAR OF HIRING-DATE
             NO-OF-HOURS, HOURLY-RATE
```

Example 8-47

```
01 Data-structure.
   02 alphaItem       PIC X(20).
   02 numItem         PIC 9(15).
   02 pointerA        POINTER.
   02 FILLER          PIC 9(5).99.
INITIALIZE Data-structure FILLER
      REPLACING ALPHANUMERIC BY HIGH VALUE
                NUMERIC BY 5
      DEFAULT
```

The statement INITIALIZE means:

```
MOVE HIGH-VALUE TO AlphaItem
MOVE 5 TO NumItem
SET PointerA TO NULL                (1)
MOVE ZERO TO <FILLER-data-item>     (2)
```

(1) Because of the “Default” specification

(2) Because of the FILLER specification and the “Default” specification

Example 8-48

```
01 Data-structure.
   02 alphaitem      PIC X(20).
   02 numItem        PIC 9(15) value 1860.
   02 pointerA       POINTER.
   02 FILLER         PIC 9(5).99.
   02 objref         OBJECT REFERENCE.
INITIALIZE Data-structure FILLER
     ALL TO VALUE
     REPLACING ALPHANUMERIC BY LOW-VALUE
                  NUMERIC BY 5
     DEFAULT.
```

The statement INITIALIZE means:

```
MOVE LOW-VALUE TO alphaitem
MOVE 1860 TO numItem               (1)
SET pointerA TO NULL               (2)
MOVE ZERO TO <FILLER-data-item>    (3)
SET objref TO NULL                 (2)
```

(1) Because of the VALUE phrase;
the REPLACING phrase for NUMERIC is not used in this case

(2) Because of the VALUE phrase

(3) Because of the FILLER phrase and the DEFAULT phrase

Example 8-49

```
01 Data-structure.
   02 alphaitem      PIC X(20).
   02 numItem        PIC 9(15) value 1860.
   02 pointerA       POINTER.
   02 FILLER         PIC 9(5).99.
   02 objref         OBJECT REFERENCE.

INITIALIZE Data-structure
     DATA-POINTER TO VALUE
     REPLACING ALPHANUMERIC BY HIGH-VALUE
```

```
           NUMERIC BY 5
      DEFAULT.
```

The statement INITIALIZE means:

```
MOVE HIGH-VALUE TO alphaitem
MOVE 5 TO numItem
SET pointerA TO NULL              (1)
SET objref TO NULL                (2)
```

(1) Because of the VALUE phrase

(2) Because of the DEFAULT phrase

Example 8-50

```
01 PTR USAGE POINTER.
...
INITIALIZE PTR DATA-POINTER TO VALUE
```

is a more complicated way of writing:

```
SET PTR TO NULL.
```

Example 8-51

```
01 Datastructure VALUE "XXXX1234".
   02 alphaitem         PIC X(4).
   02 numItem           PIC 9(4).

INITIALIZE Data-structure ALL TO VALUE DEFAULT
```

The statement INITIALIZE means:

```
MOVE SPACE TO alphaitem
MOVE ZERO TO numItem
```

The VALUE phrase in the INITIALIZE statement has no effect because the VALUE clause is specified at the group item level and not at the level of the elementary items.

8.10.26 INSPECT statement

Function

The INSPECT statement enables single characters or groups of characters within a data item to be tallied, replaced, or tallied and replaced.

Format 1 (tallying)

```
INSPECT identifier-1 TALLYING

{identifier-2 FOR { { {ALL | LEADING} {identifier-3 | literal-1} | CHARACTERS}
  [{BEFORE | AFTER} INITIAL {identifier-4 | literal-2}]... }... }...
```

Format 2 (replacing)

```
INSPECT identifier-1 REPLACING

{ CHARACTERS BY {identifier-5 | literal-3} [{BEFORE | AFTER} INITIAL {identifier-4 |
literal-2}]...

 |{ALL | LEADING | FIRST} { {identifier-3 | literal-1}
   BY {identifier-5 | literal-3} [{BEFORE | AFTER} INITIAL { identifier-4 | literal-
2} ]... }...
}...
```

Format 3 (tallying and replacing)

```
INSPECT identifier-1 TALLYING

{identifier-2 FOR {{ {ALL | LEADING} {identifier-3 | literal-1} | CHARACTERS}
   [{BEFORE | AFTER} INITIAL {identifier-4 | literal-2}]... }... }...

REPLACING

{ CHARACTERS BY {identifier-5 | literal-3}
   [{BEFORE | AFTER} INITIAL {identifier-4 | literal-2 }]...

 |{ALL | LEADING | FIRST} { {identifier-3 | literal-1}
     BY {identifier-5 | literal-3}
   [{BEFORE | AFTER} INITIAL {identifier-4 | literal-2}]... }... }...
```

Format 4 (converting)

```
INSPECT identifier-1 CONVERTING {identifier-6 | literal-4}
  TO {identifier-7 | literal-5}
  [{BEFORE | AFTER} INITIAL {identifier-4 | literal-2}]...
```

Syntax rules for all formats

1. identifier-1 may reference either an alphanumeric or national group item or an elementary item, which is described (implicitly or explicitly) with USAGE DISPLAY or USAGE NATIONAL.
2. identifier-3, ..., identifier-n may reference either an alphanumeric or national group item or an elementary item, which is described (implicitly or explicitly) with USAGE DISPLAY or USAGE NATIONAL.
3. literal-1, ..., literal-5 must be non-numeric literals. They may not be figurative constants which begin with ALL.

   If literal-1, literal-2 or literal-4 is a figurative constant, it stand for an implicitly singlecharacter item. If the class of identifier-1 is national, the class of the figurative constant is also national. Otherwise the class of the figurative constant is alphanumeric.
4. If the class of any literal-1, ..., literal-5, identifier-1, ..., identifier-7 is national, the class of all must be national.
5. Only one BEFORE and/or AFTER entry may be assigned to each individual ALL, LEADING, CHARACTERS, FIRST, or CONVERTING phrases.

Syntax rule for formats 1 and 3

6. identifier-2 must be an elementary numeric data item.

Syntax rules for formats 2 and 3

7. literal-3 or identifier-5 must be equal in size to literal-1 or identifier-3. When a figurative constant is used as literal-3, it is implicitly equal in size to literal-1 or identifier-3.
8. When CHARACTERS is used, literal-3 and identifier-5 must each be one character in length.

Syntax rules for format 4

9. literal-5 or identifier-7 must be equal in size to literal-4 or identifier-6.

   When a figurative constant is used as literal-5, it is implicitly equal in size to literal-4 or identifier-6.
10. No character may appear more than once in literal-4 or identifier-6.

General rules for all formats

1. The length of identifier-1 is calculated in the same way as for sending items (see section "OCCURS clause"). If identifier-1 is a zero-length item:
   a. identifier-1 and identifier-2 remain unchanged.
   b. The runtime control passes to the end of the INSPECT statement.
2. identifier-1 is processed from left to right regardless of its data class.
3. The contents of the data items indicated by identifier-1, identifier-3, identifier-4, identifier-5, identifier-6, identifier-7 are treated as follows:
   a. If any of these identifiers is described as alphabetic, alphanumeric or national its contents will be treated as a character-string of the according category.
   b. If any of these identifiers is described as alphanumeric-edited, numeric-edited, or unsigned numeric, it will be treated as though it had been redefined as alphanumeric.
   c. If any of these identifiers is described as signed numeric, it will be treated as though it had been moved to an unsigned numeric data item of the same length (not counting sign position) and this item had then been redefined as alphanumeric (see section "MOVE statement").

If identifier-1 is described as a signed numeric data item, its original sign will be retained until the INSPECT statement has executed.

4. If any of these identifiers is indexed, the index value for these items will be calculated once only, namely immediately following execution of the INSPECT statement.
5. In general rules 5 to 15, everything that applies to literal-1, literal-2, literal-3, literal-4, or literal-5 applies equally to identifier-3, identifier-4, identifier-5, identifier-6 or identifier-7.

General rules for formats 1, 2 and 3

6. While the contents of identifier-1 are being checked, each occurrence of literal-1 will be tallied (format 1) or all characters matching literal-1 will be replaced by literal-3 (format 2). If CHARACTERS is used, the characters in identifier-1 are tallied or replaced by literal-3 one at a time, depending on where the comparison operation is currently positioned.
7. The TALLYING or REPLACING operands are processed from left to right in the order in which they were specified in the INSPECT statement (comparison cycle). The first comparison cycle starts at the leftmost character in identifier-1.
8. If BEFORE or AFTER are omitted, the comparison operation to determine the occurrences of literal-1 in identifier-1 takes place as follows:
   a. The first literal-1 is compared to a series of contiguous characters within identifier1 starting with the leftmost character, where the length of this series is equal to the length of literal-1. Only if literal-1 and this portion of identifier-1 are identical, character-for-character, does a match occur.
   b. If no match between literal-1 and identifier-1 occurs, the comparison is repeated with each successive literal-1 until either a match is found or there is no next successive literal-1.
   c. If no match whatsoever occurs between literal-1 and identifier-1, the character position within identifier-1 is shifted one position to the right and the comparison cycle starts again with the first literal-1.
   d. Whenever a match occurs, the comparison cycle is terminated. Identifier-2 is incremented by 1 and/or the characters in identifier-1 which match literal-1 are replaced by literal-3. The character position within identifier-1 is then shifted to the right by the number of characters in literal-1 and the comparison cycle starts again with the first literal-1.
   e. The comparison operation continues until the rightmost character in identifier-1 has either participated successfully in a match or is the character at which a comparison cycle begins.
   f. If ALL is used, points a) to e) apply without restrictions.

      If LEADING is used, the corresponding literal-1 is always involved in the first runthrough of the comparison cycle. It only takes part is subsequent comparison cycles if the preceding cycle has produced a match with literal-1.

      If CHARACTERS is used, an implicit single-character operand takes part in the comparison cycle as though it were entered as literal-1. However, no comparison with the contents of identifier-1 takes place; instead, this operand is always considered to match the character in identifier-1 at which the comparison cycle is currently positioned.
9. If BEFORE or AFTER is used, the following restrictions apply to point 7 above:
   a. If BEFORE is used, the operation proceeds as follows: literal-1 or (if CHARACTERS has been specified) the implicit operand participates only in those comparison cycles which would have been performed if identifier-1 had ended with the character located immediately in front of the first occurrence of literal-2 within identifier-1.

If literal-2 does not occur at all within identifier-1 or identifier-4 is a zero-length item, the comparison proceeds as if BEFORE had never been entered.

b. If AFTER is used, the same considerations apply as in a). That is, a search is made in identifier-1 for literal-2. If literal-2 is located, the record pointer is positioned to the character within identifier-1 which is located immediately to the right of literal-2. From this point on, literal-1 or (if CHARACTERS has been specified) the implicit operand participates in the subsequent comparison cycles.

   If literal-2 does not occur at all within identifier-1 or identifier-4 is a zero-length item, literal-1 or (if CHARACTERS has been specified) the implicit operand is not involved in a comparison cycle.

General rules for formats 1 and 3

10. The contents of identifier-2 are not initialized when the INSPECT statement is executed.
11. If identifier-1, identifier-3 or identifier-4 occupies the same memory area as identifier-2, the results of the INSPECT statement will be unpredictable, even when these identifiers are defined in the same data description entry (see section "Overlapping operands").

General rules for formats 2 and 3

12. The mandatory words ALL, LEADING and FIRST are adjectives which apply to all subsequent BY phrases until the next adjective is entered.
13. If FIRST is used, literal-1 will be replaced by literal-3 within identifier-1 only at the position where it occurs for the first time. This rule applies to all successive FIRST phrases, regardless of the value of literal-1.
14. If identifier-3, identifier-4 or identifier-5 occupies the same memory area as identifier-1, the results of the INSPECT statement will be unpredictable, even when these identifiers are described in the same data description entry (see section "Overlapping operands").

General rule for format 3

15. A format 3 INSPECT statement is executed as though it were two successive INSPECT statements referring to the same identifier-1, namely one format 1 INSPECT statement (with TALLYING phrase) and a format 2 INSPECT statement (with REPLACING phrase). The rules given for formats 1 and 2 apply accordingly. Subscripting associated with any identifier in the format 2 statement is evaluated only once before executing the format 1 statement.

General rules for format 4

16. A format-4 INSPECT statement is interpreted and executed as though it were a format-2 INSPECT statement containing a series of ALL phrases (one for each character in literal-4 or identifier-6) which refer to the same identifier-1.

    Thus, each character in literal-4 or identifier-6 and the corresponding character in literal-5 or identifier-7 is interpreted as though it were a self-contained literal-1 (or identifier-3) or literal-3 (identifier-5) in format 2.

    The unique assignment of characters from literal-4 (identifier-6) and literal-5 (identifier-7) results from their position within the data item.
17. If identifier-4, identifier-6 or identifier-7 occupies the same memory area as identifier-1, the results of the INSPECT statement will be unpredictable, even when the identifiers are defined in the same data description entry (see section "Overlapping operands").

Example 8-52

for all formats:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  INSP.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
    TERMINAL IS T.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 COUNTER1 PIC 99 VALUE ZEROES.
01 COUNTER2 PIC 99 VALUE 0.
```

```
01 COUNTER3 PIC 99 VALUE 0.
01 FIELD PIC X(20) VALUE SPACES.
   ...

PROCEDURE DIVISION.
PROC SECTION.
COUNT-PAR.
    MOVE "BBYZYZBBYZAXBXBBX" TO FIELD.
    INSPECT FIELD TALLYING
       COUNTER1 FOR ALL "X" AFTER INITIAL "A"
       COUNTER2 FOR LEADING "YZ" AFTER INITIAL "BB"            (1)
       COUNTER3 FOR CHARACTERS BEFORE INITIAL "A".
    DISPLAY "After INSPECT" UPON T.
    DISPLAY "Counter1 = *" COUNTER1 "*" UPON T.
    DISPLAY "Counter2 = *" COUNTER2 "*" UPON T.
    DISPLAY "Counter3 = *" COUNTER3 "*" UPON T.
REPLACE-1.
    MOVE "MR. COBOLUSER" TO FIELD.
    DISPLAY "Before INSPECT" UPON T
    DISPLAY "Field = *" FIELD "*" UPON T.
    INSPECT FIELD REPLACING
       CHARACTERS BY "X" AFTER INITIAL "MR. "
                         BEFORE INITIAL "U".
    DISPLAY "After INSPECT" UPON T.
    DISPLAY "Field = *" FIELD "*" UPON T.
REPLACE-2.
    MOVE "ALGOL-PROGRAM" TO FIELD.
    DISPLAY "Before INSPECT" UPON T.
    DISPLAY "Field = *" FIELD "*" UPON T.
    INSPECT FIELD REPLACING
       ALL "A" BY "C" BEFORE INITIAL "P"
       ALL "L" BY "O" BEFORE INITIAL "G"
       ALL "G" BY "B" BEFORE INITIAL "P".
    DISPLAY "After INSPECT" UPON T.
    DISPLAY "Field = *" FIELD "*" UPON T.
REPLACE-3.
    MOVE "XXYZYZXXYZ-XYZXYZ" TO FIELD.
    DISPLAY "Before INSPECT" UPON T.
    DISPLAY "Field = *" FIELD "*".
    INSPECT FIELD REPLACING
       LEADING "YZ" BY "AB" BEFORE INITIAL "-"
                            AFTER INITIAL "XX"                 (2)
       FIRST "YZ" BY "CD" AFTER INITIAL "-".
    DISPLAY "After INSPECT" UPON T.
    DISPLAY "Field = *" FIELD "*" UPON T.
CONVERT.
    MOVE "CE#CGDHDEF-CD#F" TO FIELD.
    DISPLAY "Before INSPECT" UPON T.
    DISPLAY "Field = *" FIELD "*" UPON T.
    INSPECT FIELD CONVERTING                                   (3)
       "CDEF" TO "UVWU" AFTER "#"
                        BEFORE "-".
    DISPLAY "After INSPECT" UPON T.
    DISPLAY "Field = *" FIELD "*" UPON T.
ENDE.
    STOP RUN.
```

Result:

After INSPECT (1)

```
COUNTER1 = *03*
COUNTER2 = *02*
COUNTER3 = *06*
```

Before INSPECT	After INSPECT
FIELD = *MR. COBOLUSER * FIELD = *ALGOL-PROGRAM * FIELD = *XXYZYZXXYZ-XYZXYZ *	FIELD = *MR. XXXXXUSER * FIELD = *COBOL-PROGRAM * (2) FIELD = *XXABABXXYZ-XCDXYZ *
FIELD = *CE#CGDHDEF-CD#F *	FIELD = *CE#UGVHVWU-CD#F * (3)

Explanation

(1) In the case of COUNTER2, the instances of YZ underlined in the following string were tallied in FIELD:

BBYZYZBBYZAXBXBBX

In other words, with each of the underlined YZ pairs there was a match in the sense of general rule 7f. Hence, on both occasions the compare cycle resumed with the first literal-1, i.e. with COUNTER1.

The result is as follows:

COUNTER3 is not incremented for the underlined YZ pairs. Hence, only the following underlined characters are tallied:

BBYZYZBBYZAXBXBBX

Thus, COUNTER3 is equal to 6.

(2) The replacement of leading YZ pairs by AB is caused by entering AFTER INITIAL "XX". BEFORE INITIAL "-" has no effect, because on account of the LEADING phrase each string that is not equal to YZ prevents further replacements.

(3) This INSPECT statement has the same effect as the following statement:

```
INSPECT FIELD REPLACING
    ALL "C" BY "U" AFTER "#" BEFORE "-"
    ALL "D" BY "V" AFTER "#" BEFORE "-"
    ALL "E" BY "W" AFTER "#" BEFORE "-"
    ALL "F" BY "U" AFTER "#" BEFORE "-".
```

8.10.27 INVOKE statement

Function

The INVOKE statement causes a method to be invoked.

Format

```
INVOKE identifier-1 {identifier-2 | literal-1}
       [USING {  [BY REFERENCE] {identifier-3 | OMITTED}
               | [BY CONTENT] {identifier-5 | arithmetic-expression-1 | literal-2}
               | [BY VALUE] {identifier-5 | arithmetic-expression-1 | literal-2}
              }... ]
[ RETURNING identifier-4 ]
[ON EXCEPTION imperative-statement-1]
[NOT ON EXCEPTION imperative-statement-2]
[END-INVOKE]
```

Syntax rules

1. identifier-1 must be an object reference or a class-name.
2. literal-1 must be an alphanumeric literal. However, it may not be a figurative constant.
3. If identifier-1 references a universal object reference, neither the BY CONTENT nor BY VALUE phrase must be specified and the BY REFERENCE phrase, if not specified explicitly, is assumed implicitly.
4. identifier-2 must be an alphanumeric data item.
5. identifier-3 must reference a data item defined in the FILE SECTION, WORKING-STORAGE SECTION, LOCAL-STORAGE SECTION or LINKAGE SECTION.
6. If identifier-5 or its corresponding formal parameter (in the PROCEDURE DIVISION header) is specified with the BY VALUE phrase, identifier-5 may only be of class “numeric”, “object” or “pointer”.
7. identifier-4 must reference a data item defined in the FILE SECTION, WORKING-STORAGE SECTION, LOCAL-STORAGE SECTION or LINKAGE SECTION.
8. If identifier-2 is specified, identifier-1 must be a universal object reference.
9. If identifier-1 is not a universal object reference, the rules for conformance as specified for parameters and returning items apply.
10. If identifier-1 is not a universal object reference and a BY CONTENT or BY REFERENCE phrase is specified for an argument, a BY REFERENCE phrase must be specified for the corresponding formal parameter in the Procedure Division header.
11. BY CONTENT may not be omitted if identifier-5 is permitted as a receiving item.
12. If a BY VALUE phrase is specified for an argument, a BY VALUE phrase must also be specified for the corresponding formal parameter in the Procedure Division header.

13. If an OMITTED phrase is specified, an OPTIONAL phrase must also be specified for the corresponding formal parameter in the Procedure Division header.
14. If identifier-1 is a universal object reference then the formal parameters and formal return element must not be defined with the ANY LENGTH clause.
15. If identifier-3 or identifier-4 are defined with the ANY LENGTH-clause, then the corresponding formal parameter must also be defined with the ANY LENGTH clause. If identifier-5 is defined with the ANY LENGTH clause then the corresponding formal parameter must **not** be defined with the ANY LENGTH clause.
16. If the formal parameter corresponding to identifier-5 is defined with the ANY LENGTH clause the identifier-5 must be of the class “alphabetic” or “alphanumeric”.
17. identifier-5 and any identifier specified in arithmetic-expression-1 are send operands.
18. identifier-4 is a receiving item.
19. identifier-3 must reference an address identifier or an elementary data item which is defined in the FILE SECTION, WORKING-STORAGE SECTION, LOCAL-STORAGE SECTION or LINKAGE-SECTION.
20. If the BY REFERENCE phrase is specified or implied for identifier-3, which is not an address identifier, then identifier-3 represents both a sending and a receiving item. In all other cases, identifier-3 specifies a sending item.

    Note:
    The specification BY REFERENCE ADDRESS OF data-name is processed in the same way as BYCONTENT ADDRESS OF data-name.

General rules

1. identifier-1 identifies an object instance. If a class-name is specified as identifier-1, it identifies the factory object of that class-name. Literal-1 or the content of the data item referenced by identifier-2 identifies a method of that object that will act upon that object instance.
   literal-1 or the content of the identifier-2 is the name of the method to be invoked as specified in the corresponding METHOD-ID paragraph (as a COBOL word).
2. The sequence of arguments in the USING phrase of the INVOKE statement and the corresponding formal parameters in the USING phrase of the invoked method’s Procedure Division header determines the correspondence between arguments and formal parameters. This correspondence is positional and not by name equivalence. The first argument corresponds to the first formal parameter, the second to the second parameter, and the nth to the nth parameter.
3. An argument that consists merely of a single identifier or literal is regarded as an identifier or literal rather than an arithmetic expression.
4. If identifier-1 is null, an exception condition EC-OO-NULL occurs, no method is activated, and the execution proceeds as specified in general rule 6f.
5. If identifier-1 is not a universal object reference and an argument without any of the keywords BY REFERENCE, BY CONTENT, or BY VALUE is specified, that argument is handled as follows:
   a. BY REFERENCE is assumed if the BY REFERENCE phrase is specified or implied for the corresponding formal parameter and if the argument is an identifier that is permitted as the receiving item.
   b. BY CONTENT is assumed if the BY REFERENCE phrase is specified or implied for the corresponding formal parameter and if the argument is a literal, an arithmetic expression or any other identifier that is not permitted as the receiving item.
   c. BY VALUE is assumed if the BY VALUE phrase is specified or implied for the corresponding formal parameter.

6. Execution of the INVOKE statement proceeds as follows:
   a. arithmetic-expression-1, identifier-1, identifier-2, identifier-3, and identifier-5 are evaluated and item identification is done for identifier-4 at the beginning of the execution of the INVOKE statement. If an exception condition exists, no method is invoked and execution proceeds as specified in general rule 6f.
   b. The runtime system attempts to locate the invoked method. If the method cannot be located or the resources necessary to execute the method are not available, the exception condition EC-OO-METHOD occurs. The method is not activated, and execution continues as specified in general rule 6f.
   c. If identifier-1 is a universal object reference, the conformance rules for parameters and returning items must apply. If a violation of these rules is detected, an exception condition EC-OO-UNIVERSAL, the method is not activated, and execution continues as specified in general rule 6f.
   d. The method specified by the INVOKE statement is made available for execution and control is transferred to the invoked method (in conformance with the calling conventions).
   e. After control is returned from the invoked method, imperative-statement-2 is executed if present, and control is transferred to the end of the INVOKE statement.
   f. If one of the exception conditions EC-OO-NULL, EC-OO-METHOD or EC-OO-UNIVERSAL has occurred, the following procedure results:
      - If ON EXCEPTION is specified, control is transferred to imperative-statement-1 and then to the end of the INVOKE statement. The associated exception condition is not triggered.
      - If ON EXCEPTION is not specified and the check of the exception condition is activated, the associated exception condition is triggered and control is transferred to the relevant USE procedure. After RESUME NEXT StATEMENT has been executed in the USE procedure, imperative-statement-2 is ignored and processing continues with the next executable statement after the end of the INVOKE statement.
      - If ON EXCEPTION is not specified and the check of the exception condition is not activated but NOT ON EXCEPTION is specified, imperative-statement-2 is ignored and processing continues with the next executable statement after the end of the INVOKE statement.
      - If neither ON EXCEPTION nor NOT ON EXCEPTION is specified and the check of the exception condition is not activated, the program run is aborted.

   If an exception condition other than EC-OO-NULL, EC-OO-METHOD or EC-OO-UNIVERSAL has occurred (e. g. EC-OO-CONFORMANCE), and if the check of the exception condition is activated, the associated exception condition is triggered and control is transferred to the relevant USE procedure. After returning from the USE procedure, imperative-statement-1 and imperative-statement-2 are ignored and processing continues with the next executable statement after the end of the INVOKE statement.
   If the check of the exception condition is not activated, the program run is aborted.
7. If a RETURNING phrase is specified, the result of the invoked method is placed in identifier-4.
8. If an OMITTED phrase is specified or an argument is completely omitted, the omitted-argument condition for that parameter is true in the invoked method.
9. If a parameter for which the omitted-argument condition is true for a parameter and this is referenced in an invoked method (except in an omitted-argument condition), the behavior is undefined.

8.10.28 MERGE statement

Function

The MERGE statement creates a sort-file into which records are accepted from two or more similarly sorted input files. It merges the records in the sort-file on the basis of a set of specified data items (keys) and, once this merge operation is finished, makes each record from the sort-file available to an output procedure or to output files.

Format

```
MERGE sort-file-name
    {ON {DESCENDING | ASCENDING} {KEY |  KEY-YY } {data-name-1}... }...
    [COLLATING SEQUENCE IS alphabet-name]
    USING {file-name-1}...
    { OUTPUT PROCEDURE IS section-name-1 [{THRU | THROUGH} section-name-2]
     |GIVING {file-name-2}...
    }
```

Syntax rules

1. sort-file-name must be defined in a sort-file description (SD) entry in the Data Division.
2. sort-file-name must correspond to the sort-file-name defined in the SELECT clause (format 2).
3. The file names specified in the USING phrase must not be specified in the GIVING phrase.
4. data-name-1... are key data-names. A key is that part of a record which is used as a basis for sorting. Key data-names must be defined in a record description belonging to an SD description entry. They are subject to the following rules:
   a. The data items identified by key data-names must not be of variable length.
   b. The data-names describing the keys may be qualified (see section "Qualification").
   c. When two or more record descriptions are supplied, the keys need only be described in one of these descriptions. If a key is defined in more than one record description, the descriptions of that key must be identical, and must ensure that the key appears in the same position within each record.
   d. A key must not be defined with an OCCURS clause and must not be subordinate to a data item defined with an OCCURS clause.
   e. If the sort-file contains variable length records, all the data items identified by key data-names must be contained within the first n character positions of the record, where n equals the minimum record size specified for the sort-file.
   f. A maximum of 64 keys may be specified for any file.
   g. Each key must lie within the first 4096 bytes.
   h. Keys are always listed from left to right in the order of their decreasing significance, regardless of whether they are ascending or descending. Hence, the first occurrence of data-name-1 would be the principal sort-key and the second occurrence of data-name-1 the subsidiary key.
   i. A key, when expressed as a packed decimal, may have no more than 16 digits.

   j. The keys following the KEY-YY specification must be defined either with PIC 99 USAGE DISPLAY or USAGE PACKED-DECIMAL.

5. section-name-1 identifies the first or only section in the output procedure. section-name-2 identifies the last section in the output procedure. It is required only if the output procedure consists of more than one section.
6. file-name-1..., file-name-2... must be defined in a file description (FD) entry in the Data Division.
7. The size of the records that can be passed to a MERGE operation or written to an output file is dependent on the record format of the sort-file (variable or fixed) and will be discussed in more detail in the "General rules" when the USING/GIVING phrases are mentioned.
8. At least one ASCENDING/DESCENDING phrase must be specified in a MERGE statement.
9. The MERGE statement may be written anywhere in the program except
   a. in the declaratives area,
   b. in an input/output procedure belonging to a SORT/MERGE statement.
10. The sort- and input-files named in the MERGE statement must not be specified together in the same SAME AREA, SAME SORT AREA or SAME SORT-MERGE AREA (see section "SAME AREA clause").
11. If file-name-2 references an indexed file, the first specification of data-name-1 must be associated with an ASCENDING phrase and the data item referenced by that data-name-1 must occupy the same character positions in its record as the record key occupies within the file referenced by file-name-2.

General rules

1. The collating sequence is set by means of the ASCENDING/DESCENDING option:
   a. When ASCENDING is specified, sorting proceeds from the lowest to the highest value of the key, i.e. in ascending order.
   b. When DESCENDING is specified, sorting proceeds from the highest to the lowest value of the key, i.e. in descending order.

      The collating sequence is governed by the same rules as apply to the comparison of operands in a relation condition (see section "Relation condition").
2. When, according to the rules for the comparison of operands in a relation condition, the contents of all the key data items of one record are equal to the contents of the corresponding key data items of one or more other records, the order of return of these records:
   a. Follows the order of the associated input files as specified in the MERGE statement.
   b. Is such that all records associated with one input file are returned prior to the return of records from another input file.
3. When the program is executed, the collating sequence for the comparison of alphanumeric sort items is set as follows:
   a. If COLLATING SEQUENCE has been specified in the MERGE statement, this entry is used as a sort criterion.
   b. If COLLATING SEQUENCE was not specified in the MERGE statement, the program-specific collating sequence will be used (see section "OBJECT-COMPUTER paragraph").

   The national (native) collating sequence is used for comparisons of national collating sequences.
4. All records from the input files (file-name-1...) are transferred to the sort-file designated by sort-file-name. At the start of execution of the MERGE statement, the input files must not be in the open mode. The MERGE statement is executed for each of the referenced files in the following way:

a. The processing of the file is initiated. The initiation is performed as if an OPEN INPUT statement had been executed.

b. The logical records are obtained and released to the merge operation. Each record is obtained as if a READ statement with the NEXT and the AT END phrases had been executed. If the input file contains the RECORD clause with the DEPENDING phrase, the associated DEPENDING ON data item will not be provided for this READ operation. Relative files must be described in the FILE CONTROL paragraph with ACCESS MODE IS SEQUENTIAL.
If the sort-file contains variable length records, the size which the record had when it was input is used as the length for a record when it is transferred to a MERGE operation. This length must be within the range defined for the sort-file in the RECORD clause (see section "RECORD clause"). If the sort-file has a fixed length record format, records shorter than the specified format length will be supplied with blanks, longer records will not be permitted.

c. The processing of the file is terminated. The termination is performed as if a CLOSE statement without optional phrases had been executed.

These implicit functions are performed such that any associated USE AFTER EXCEPTION/ERROR procedures are executed.

5. OUTPUT PROCEDURE indicates that the Procedure Division contains an output procedure to process records after they have been merged. If OUTPUT PROCEDURE is specified, control passes to it after the sort-file has been processed by the MERGE statement. During RETURN statement processing, the output procedure accepts the records from the sort-file. The compiler inserts a return mechanism at the end of the last section of the output procedure. When control passes to the last statement in the output procedure, the return mechanism provides for termination of the sort, and then passes control to the statement following the MERGE statement.

   The following rules apply to the output procedure, which is a self-contained section within the Procedure Division:

   a. It must consist of one or more sections that are written consecutively.

   b. It must contain at least one RETURN statement, to make merged records available for processing.

   c. It must not lead to execution of a MERGE, RELEASE, or SORT statement.

   d. It may include any procedures needed to select, modify, or copy records.

   e. A branch may be made from the output procedure if the programmer makes sure that a transfer of this type is followed by a return to the output procedure in order to effect a proper exit from this procedure (i.e. to processing its last statement).

   f. A branch may be made from points outside an output procedure to procedure names within that procedure if the branch does not involve a RETURN statement or the end of the output procedure.

6. When the OUTPUT PROCEDURE phrase is used, control is passed from the specified procedure as though a format-1 PERFORM statement is executing. That is, all sections constituting the procedure are executed once, and execution of the procedure is terminated after its last statement has been processed. Thus, any procedure may be terminated by using an EXIT statement.

7. If MERGE statements are supplied in segmented programs, the following restrictions apply:

   a. If a MERGE statement appears in a section which is outside any independent segment, then all input or output procedures referred to by that MERGE statement must be either wholly contained within one fixed segment, or wholly contained in a single independent segment.

   b. If a MERGE statement appears in an independent segment, then all input or output procedures referred to by that MERGE statement must be either wholly contained within one fixed segment, or wholly contained in the same independent segment as the MERGE statement.

   These restrictions do not apply to the compiler discussed in this manual.

8. If the GIVING phrase is specified, all the merged records are written on the output file (file-name-3...). At the start of execution of the MERGE statement, the output file must not be in the open mode. The MERGE statement is executed for each of the referenced files in the following way:
   a. The processing of the file is initiated. The initiation is performed as if an OPEN statement with the OUTPUT phrase had been executed.
   b. The merged logical records are returned and written onto the output file as if a WRITE statement without any optional phrases had been executed. The length of these records must be within the range defined for the output file (see section "RECORD clause").
      For a relative file, the relative key data item for the first record returned contains the value 1; for the second record returned, the value 2, etc. After execution of the MERGE statement, the content of the relative data item indicates the last record returned to the file. The file must be defined in the FILE-CONTROL paragraph with ACCESS MODE IS SEQUENTIAL.
   c. The processing of the file is terminated. The termination is performed as if a CLOSE statement without optional phrases had been executed. If the output file contains the RECORD clause with DEPENDING ON phrase, the associated DEPENDING ON data item is not evaluated when a record is written. The current record length is used instead.

Example 8-53

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MERGE1.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT INPUT-FILE-1 ASSIGN TO "INPUT-FILE-1".
    SELECT INPUT-FILE-2 ASSIGN TO "INPUT-FILE-2".
    SELECT OUTPUT-FILE-1 ASSIGN TO "OUTPUT-FILE-1".
    SELECT OUTPUT-FILE-2 ASSIGN TO "OUTPUT-FILE-2".
    SELECT SORT ASSIGN TO "SORTWK".
DATA DIVISION.
FILE SECTION.
FD INPUT-FILE-1 LABEL RECORD STANDARD.
01 INPUT-RECORD-1.
    02  I10          PIC X.
    02  I11          PIC 9(4).
    02  I12          PIC 9(4).
    02  I13          PIC 9(4).
FD INPUT-FILE-2 LABEL RECORD STANDARD.
01 INPUT-RECORD-2.
    02  I20          PIC X.
    02  I21          PIC 9(4).
    02  I22          PIC 9(4).
    02  I23          PIC 9(4).
FD OUTPUT-FILE-1 LABEL RECORD STANDARD.
01 O1RECORD PIC X(13).
FD OUTPUT-FILE-2 LABEL RECORD STANDARD.
01 O2RECORD PIC X(13).
SD SORT LABEL RECORD STANDARD.
01 SRECORD.
    02  S0           PIC X.
    02  S1           PIC 9(4).
    02  S2           PIC 9(4).
    02  S3           PIC 9(4).
PROCEDURE DIVISION.
```

```
MAIN SECTION.
H01.
    MERGE SORT ON ASCENDING S1 S2 S3       (1)
    USING INPUT-FILE-1 INPUT-FILE-2         |
    GIVING OUTPUT-FILE-1 OUTPUT-FILE-2.    (1)
H02.
    STOP RUN.
```

(1) The records of two files sorted in the same way are output to two identical files in a sorted sequence.

All files are sorted in ascending order in accordance with the sort terms S1 S2 S3.

Example 8-54

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MERGE2.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT INPUT-FILE-1 ASSIGN TO "INPUT-FILE-1".
    SELECT INPUT-FILE-2 ASSIGN TO "INPUT-FILE-2".
    SELECT OUTPUT-FILE-1 ASSIGN TO "OUTPUT-FILE-1".
    SELECT OUTPUT-FILE-2 ASSIGN TO "OUTPUT-FILE-2".
    SELECT SORT ASSIGN TO "SORTWK".
DATA DIVISION.
FILE SECTION.
FD INPUT-FILE-1 LABEL RECORD STANDARD.
01 INPUT-RECORD-1.
    02  I10          PIC X.
    02  I11          PIC 9(4).
    02  I12          PIC 9(4).
    02  I13          PIC 9(4).
FD INPUT-FILE-2 LABEL RECORD STANDARD.
01 INPUT-RECORD-2.
    02  I20          PIC X.
    02  I21          PIC 9(4).
    02  I22          PIC 9(4).
    02  I23          PIC 9(4).
FD OUTPUT-FILE-1 LABEL RECORD STANDARD.
01 O1RECORD PIC X(13).
FD OUTPUT-FILE-2 LABEL RECORD STANDARD.
01 O2RECORD PIC X(13).
SD SORT LABEL RECORD STANDARD.
01 SRECORD.
    02  S0           PIC X.
    02  S1           PIC 9(4).
    02  S2           PIC 9(4).
    02  S3           PIC 9(4).
WORKING-STORAGE SECTION.
01 MERGE-STATUS PIC X VALUE LOW-VALUE.
    88  MERGE-END VALUE HIGH-VALUE.
PROCEDURE DIVISION.
MAIN SECTION.
H01.
    OPEN OUTPUT OUTPUT-FILE-1 OUTPUT-FILE-2.
H02.
    MERGE SORT ON ASCENDING S1 S2 S3       (1)
```

```
    USING INPUT-FILE-1 INPUT-FILE-2          |
    OUTPUT PROCEDURE IS OUT1.               (1)
H03.
    CLOSE OUTPUT-FILE-1 OUTPUT-FILE-2.
    STOP RUN.
            .
            .
            .
OUT1 SECTION.
001.
    PERFORM UNTIL MERGE-END                 (2)
      RETURN SORT                            |
      AT END                                 |
        SET MERGE-END TO TRUE                |
      NOT AT END                             |
        WRITE 01RECORD FROM SRECORD          |
        WRITE 02RECORD FROM SRECORD         (2)
      END-RETURN
    END-PERFORM.
```

(1) The records from the files sorted in the same way are transferred to the sort file in their sorted order. Control then passes to the output procedure (OUT1 SECTION).

(2) The sort file is taken record by record and written to the output files.

8.10.29 MOVE statement

Function

The MOVE statement transfers data from one data item to one or more other data items.

Format 1 of the MOVE statement moves one data item to one or more other data items.

Format 2 of the MOVE statement moves corresponding data items from one group to another.

Format 1

```
MOVE {identifier-1 | literal} TO {identifier-2}...
```

Syntax rules

1. An index data item, a data pointer or a data item of class “object” must not appear as an operand of a MOVE statement.
2. identifier-2 may not be a group data item containing items of the class "object" or "pointer". If identifier-2 is a strongly typed group data item, identifier-1 must be defined as a group data item of the same type.
3. identifier-1 resp. literal are sending items.
4. identifier-2 is a receiving item.
5. The figurative constant SPACE may not be moved to numeric or numeric-edited elementary items.
6. The figurative constant ZERO may not be moved to alphabetic elementary items.
7. table 28 shows the moves that are permitted in a MOVE statement and their type.

Receiving item **Sending item**		Group	Group Alphabetic	Alphanumeric	National	External decimal, Binary, Internal decimal, Numeric literal	Alphanumeric edited	External fl. point, Internal fl. point	Strongly typed
Group		nn0	nn0	nn0	-	nn0	nn0	nn0	-
Alphabetic		nn0	nn	nn	nn1	-	nn	-	-
Alphanumeric, Alphanumeric literal		nn0	nn	nn	nn1	nu2	nn	nu2	-
National, National literal		-	-	-	nn	-	-	-	-
External decimal, Binary, Internal decimal,	integer	nn0	-	nn	nn1	nu	nn	nu	-
	non-integer	nn0	-	-	-	nu	-	nu	-

Numeric literal								
Numeric edited	nn0	-	nn	nn1	nu1	nn	nu1	-
Alphanumeric edited	nn0	nn	nn	nn1	-	nn	-	-
External fl. point	nn0	-	-	-	nu	-	nu	-
Internal fl. point, Fl. point literal	nn0	-	-	-	nu3	-	nu	-
Strongly typed	nn0	nn0	nn0	-	nn0	nn0	nn0	nn0[1)]

Table 28: Legal moves in the case of elementary and group moves[2)]

[1)] Only for a receiving item of the same type

[2)] table entries:

-- = Move illegal

nn = Non-numeric move for elementary items

nn0 = Non-numeric move for groups (without conversion, without editing, without deediting)

nn1 = Non-numeric move with conversion of the representation of a character from EBCDIC to UTF-16

nu = Numeric move

nu1 = Numeric move after deediting of the sending item

nu2 = Numeric move, the sending item being treated as an external decimal elementary item

nu3 = Numeric move with rounding

General rules

1. identifier-1 or literal is moved to every data item addressed by identifier-2 in the specified order.
2. Any subscripting, indexing, reference modification and length calculation associated with identifier-2 is performed immediately before the data is moved to the relevant data item. If identifier-2 is a zero-length item, it is not modified by the MOVE statement.
3. Any subscripting, indexing, reference modification and length calculation associated with identifier-1 is performed only once before the first move. If identifier-1 is a zerolength item, the MOVE statement is executed as if SPACE were specified as a sending item.
4. Any necessary conversion of data from one form of internal representation to another takes place during legal moves, along with any editing specified for the receiving data item.
5. Each move in which the sending item is a literal or an elementary item and the receiving item is also an elementary item is an **elementary move**. In a MOVE statement in format 1 a national group is treated like an elementary item.

   Any other move is a **group move**. A group move corresponds to an elementary move with alphanumeric sending and receiving item with the difference that no conversion takes place, i.e. the receiving item is filled without taking into account the group items/data items contained in the sending and receiving items.

(For special aspects associated with the OCCURS DEPENDING phrase, see general rule 14 of the OCCURS clause on OCCURS clause.)

6. Any editing specified for the receiving data item takes place during an elementary move (some examples are given below).
7. Deediting takes place only if the sending item is numeric-edited and the receiving item is numeric or numeric-edited.
8. There are two types of elementary moves:
   a. A **non-numeric move** takes place when the receiving item is an alphanumeric edited, alphanumeric, alphabetic, or national item.
      - Alignment and filling with blanks take place in accordance with section "Alignment of data" in section "Concept of computer-independent data description".
      - If the sending item is described as numeric with sign, the sign is not moved. If the sign occupies a character position of its own, it is not moved and the length of the sending item is assumed to be reduced by 1.
      - If the USAGE clause of the sending item differs from that of the receiving item, the sending item is converted for internal representation of the receiving item.
      - If the sending item is numeric and contains the PICTURE symbol 'P', all these character positions are counted in the size of the sending item and it is assumed that they contain the value 0.
   b. A **numeric move** takes place when the receiving item is a numeric or a numeric edited item.
      - Alignment and filling with blanks take place in accordance with section "Alignment of data" in section "Concept of computer-independent data description".
      - If the sending item is numeric-edited, it is deedited in order to determine its numeric value. This value is moved to the receiving item.
      - If the receiving item is described with a sign, the sign of the sending item is moved. If the sending item has no sign, a positive sign is generated in the receiving item.
      - If the receiving item as no sign, the absolute value of the sending item is moved and no sign is generated in the receiving item.
      - If the sending item is alphanumeric, it is treated like an external decimal elemtary item. It may only contain numeric characters. If it is longer than 31 characters, only the rightmost 31 characters are used.
9. Every conversion between different types of internal representation takes place during legal elementary moves. Conversion of alphanumeric to national representation also takes place. If no corresponding national character exists for an alphanumeric character in the sending item in this case, the replacement character "period" is used for this in the receiving item and the exceptional condition EC-DATA-CONVERSION occurs.

   > When the exceptional condition EC-DATA-CONVERSION occurs, the receiving item is always modified before any USE procedure that may exist is activated (see also section "NATIONAL-OF - national character representation").

10. If the sending or receiving operands of a MOVE statement share the same storage area (i.e. if the operands "overlap"), then execution of that statement produces unpredictable results.
11. Figurative constants are treated like numeric, alphanumeric or national literals depending on the type of receiving item. The table below shows the resultant category of the literal.

Figurative constant	Type of receiving item	Category of the figurative constant

ALL Alphanumeric literal	Independent of this	Alphanumeric
ALL National literal	Independent of this	National
ALL Symbolic-Character	Independent of this	Alphanumeric

HIGH-VALUE, LOW-VALUE, QUOTE	Group, Alphanumeric, Alphanumeric edited	Alphanumeric
	National	National
SPACE	Group, Alphanumeric, Alphabetic, Alphanumeric edited	Alphanumeric
	National	National
ZERO	Group, Alphanumeric, Alphanumeric edited	Alphanumeric
	National	National
	External decimal, Binary, Internal decimal, Numeric edited, External fl. point, Internal fl. point	Numeric

Table 29: Moving figurative constants

Example 8-55

Sending item		Receiving item		Statement
PICTURE IS	Value	PICTURE IS	Value	
XXX	M8N	XXXXX	M8N'BLANK''BLANK'	If no JUSTIFIED RIGHT clause is specified for the receiving item, the characters are moved from left to right. As the data to be moved does not fill the receiving item completely, the remaining positions are filled with blanks .
AAAAAA	XYZABC	AAA AAA JUST RIGHT	XYZ ABC	As the sending item is longer than the receiving item, the move is aborted as soon as the receiving item is filled. The remaining characters are ignored. If the JUSTIFIED RIGHT clause is specified for the receiving item, the characters are moved into the receiving item right-justified.
999PPP	456	XXXXX	45600	Zeros are assumed at positions of the PICTURE symbol ’P’. The sending field consequently has 6 characters. These are moved into the receiving item left-justified. The remaining characters are ignored.
S999	-333	XXX	333	If the sending item has a sign, the absolute value is moved.
9V999	8765	V99	76	The items are aligned on the decimal point. The sending item has more positions to both the left and the right of the decimal point than the receiving item. Positions are truncated at both ends.

9V9	12	99V99	0120	If the receiving item is greater than the sending item, the character positions not used are filled with zeros or editing characters.
99V99	6789	$$$99.99	'BLANK''BLANK' $67.89	
S999	-333	9999	333	If the sending item has a sign and the receiving field has none, the absolute value of the sending item is moved.
99	15	999	015	If no assumed decimal point exists, the data is moved to the receiving item right-justified.
XXX	123	99V9	230	The sending item is treated like an integer numeric item with 3 digit positions. There is no space for the leading hundreds positions in the receiving item and it is truncated.

Format 2

```
MOVE {CORRESPONDING | CORR} identifier-1 TO {identifier-2} ...
```

Syntax rules

The following rules for identifier-2 also apply to any additional identifiers which follow identifier-2.

1. CORR is the abbreviation for CORRESPONDING.
2. identifier-1 specifies a group item which contains the elementary data items to be moved.
3. identifier-2 specifies a group item which contains the receiving items for the move.
4. The data items selected from the first operand (identifier-1) are moved to corresponding data items within the second operand (identifier-2). Data items from each group are considered to be corresponding when both data items have the same name and qualification up to, but not necessarily including, identifier-1 and identifier-2.
5. Any subscripting or indexing associated with identifier-2 is evaluated immediately before the data is moved to the relevant data item.
6. Any necessary conversion of the data from one form of internal representation to another takes place during legal moves, along with any editing specified for the receiving data item. This is described more fully in the general rules which follow.

General rules

1. The results of the MOVE CORRESPONDING statement are the same as if the user had specified each pair of corresponding identifiers in a separate MOVE statement of format1 (for further rules see section "CORRESPONDING phrase").
2. At least one of the data items in each pair of corresponding data items must be an elementary data item (note that the MOVE statement differs from arithmetic statements in this respect: in arithmetic statements using the CORRESPONDING phrase, both corresponding items must be elementary data items).
3. A data item that is subordinate to identifier-1 or identifier-2 and contains an OCCURS, REDEFINES, USAGE IS INDEX, USAGE IS POINTER, USAGEIS PROGRAM-POINTER, USAGE IS OBJECT REFERENCE or RENAMES clause will be ignored. However, identifier-1 or identifier-2 themselves may have REDEFINES or OCCURS clauses or be subordinate to data items with REDEFINES or OCCURS clauses.

4. In a MOVE statement in format 2 a national group is treated like a group (and not like an elementary item).

Example 8-56

Procedure Division statement:

```
MOVE CORRESPONDING EMPLOYEE-RECORD TO PAYROLL-CHECK.
```

Data Division entries:

```
01 EMPLOYEE-RECORD.
   02 EMPLOYEE-NUMBER.
      03 PLANT-LOCATION...
      03 CLOCK-NUMBER.
         04 SHIFT-CODE...
         04 CONTROL-NUMBER...
   02 WAGES.
      03 HOURS-WORKED...
      03 PAY-RATE...
   02 FICA-RATE...
   02 DEDUCTIONS...
```

```
01 PAYROLL-CHECK.
   02 EMPLOYEE-NUMBER.
      03 CLOCK-NUMBER...
      03 FILLER...
   02 DEDUCTIONS.
      03 FICA-RATE...
      03 WITHHOLDING-TAX...
      03 PERSONAL-LOANS...
   02 WAGES.
      03 HOURS-WORKED...
      03 PAY-RATE...
   02 NET-PAY...
   02 EMPLOYEE-NAME...
      03 SHIFT-CODE...
```

According to the MOVE CORRESPONDING rules, the following data items would be moved:

Sending area	Receiving area
CLOCK-NUMBER OF EMPLOYEE- NUMBER OF EMPLOYEE-RECORD	CLOCK-NUMBER OF EMPLOYEE-NUMBER OF PAYROLL- CHECK
HOURS-WORKED OF WAGES OF EMPLOYEE-RECORD	HOURS-WORKED OF WAGES OF PAYROLL-CHECK
PAY-RATE OF WAGES OF EMPLOYEE-RECORD	PAY-RATE OF WAGES OF PAYROLL-CHECK
DEDUCTIONS OF EMPLOYEE-RECORD	DEDUCTIONS OF PAYROLL-CHECK

The following items are not moved, for the reasons stated:

Field	Reason
EMPLOYEE-NUMBER	Item is not elementary in either group.
PLANT-LOCATION OF EMPLOYEE-NUMBER OF EMPLOYEE-RECORD	Item does not appear in PAYROLL-CHECK.
SHIFT-CODE OF CLOCK-NUMBER OF EMPLOYEE-NUMBER OF EMPLOYEE-RECORD	Qualification is not identical in PAYROLL-CHECK.
CONTROL-NUMBER OF CLOCK-NUMBER OF EMPLOYEE-NUMBER OF EMPLOYEE-RECORD	Item does not appear in PAYROLL-CHECK.

WAGES	Item is not elementary in either group.
FICA-RATE OF EMPLOYEE-RECORD	Qualification is not identical in PAYROLL-CHECK.

8.10.30 MULTIPLY statement

Function

The MULTIPLY statement is used to perform multiplication of two numeric operands and store the result.

Format 1 of the MULTIPLY statement stores the products in the specified multiplier.

Format 2 of the MULTIPLY statement uses the GIVING phrase.

Format 1

```
MULTIPLY {identifier-1 | literal-1} BY {identifier-2 [ROUNDED]}...
    [ON SIZE ERROR imperative-statement-1]
    [NOT ON SIZE ERROR imperative-statement-2]
    [END-MULTIPLY]
```

Syntax rules

1. Each identifier must refer to a numeric elementary item.
2. The value of identifier-1 or literal-1 is multiplied by the value of identifier-2. The multiplication product replaces the current value of identifier-2.
3. The maximum size of the product is 31 decimal digits.

Additional rules are given under "Phrases in statements" et seq., where the ROUNDED, (NOT) ON SIZE ERROR, and GIVING phrases are discussed.

Example 8-57

Statement	PICTURE of result item	Calculation
`MULTIPLY A BY B`	`999`	`A * B` stored in B as nnn

Format 2

```
MULTIPLY {identifier-1 | literal-1} BY {identifier-2 | literal-2}
    GIVING {identifier-3 [ROUNDED]}...
    [ON SIZE ERROR imperative-statement-1]
    [NOT ON SIZE ERROR imperative-statement-2]
    [END-MULTIPLY]
```

Syntax rules

1. Each identifier preceding the word GIVING must refer to a numeric elementary item.
2. identifier-3... may refer to a numeric elementary item or a numeric edited elementary item.
3. The value of identifier-1 or literal-1 is multiplied by the value of identifier-2 or literal-2, and the product is stored in identifier-3 (the same applies to additional receiving items).

4. The maximum size of the product is 31 decimal digits.

Additional rules are given under "Phrases in statements" et seq., where the ROUNDED, (NOT) ON SIZE ERROR, and GIVING phrases are discussed.

Example 8-58

Statement	PICTURE of result item (C)	Calculation
`MULTIPLY A BY B GIVING C`	`9(5)`	`A * B` stored in `C` as nnnnn

8.10.31 OPEN statement

Function

The OPEN statement opens files for processing and performs label checking and output for **sequentially organized files**.

Format 1 for sequential file organization

```
OPEN {  INPUT {file-name-1 [ REVERSED | WITH NO REWIND]}...
      | OUTPUT {file-name-1 [WITH NO REWIND]}...
      | I-O {file-name-1}...
      | EXTEND {file-name-1}...
     }...
[END-OPEN]
```

Syntax rules

1. The EXTEND phrase must not be used for multi-file volumes (see section "I-O-CONTROL paragraph").
2. EXTEND may be specified only for files for which no LINAGE clause has been specified.
3. A file-name must not appear more than once in an OPEN statement.
4. The I-O phrase is permitted for disk storage files only.
5. The files specified in a given OPEN statement may have different organizations or access modes.
6. In the case of line sequential files, the only permissible open modes are OPEN INPUT and OPEN OUTPUT without the specifications REVERSED and NO REWIND.

Format 2 for relative and indexed file organization

```
OPEN {  INPUT {file-name-1}...
      | OUTPUT {file-name-1}...
      | I-O {file-name-1}...
      | EXTEND {file-name-1}...
     }...
[END-OPEN]
```

Syntax rules

1. The same file-name must not appear more than once in an OPEN statement.
2. The files specified in a given OPEN statement need not all have the same organization or access mode.
3. The EXTEND phrase may be specified only for files with sequential access.

General rules

For **sequential, relative and indexed file organization**:

1. After successful execution of an OPEN statement, the file specified by file-name-1 is available in the open mode.
2. After successful execution of an OPEN statement, the associated record area is available to the program. Only one record area exists for an external file; this area is available to all programs that describe this file.
3. Before successful execution of an OPEN statement for a file, no statement (except for a SORT or MERGE statement with the USING or GIVING phrase) may be executed that would either explicitly or implicitly reference that file.
4. INPUT means that the file is to be processed as an input file (input mode).
5. OUTPUT indicates that the file is to be processed as an output file (output mode).
6. I-O indicates that input and output operations (i.e. read, write and update operations) are to be performed on the file (update mode). Since this phrase implies the existence of the file, it cannot be used if the file is being initially created.
7. The EXTEND phrase positions the file immediately after the last logical record in the file. For sequentially organized files, this is the last record written in the file; for relative organization, the record with the highest relative key, and for indexed file organization, the record with the highest value of the primary key. Subsequent WRITE statements for this file causes records to be appended to the file exactly as if it had been opened in OUTPUT mode (extend mode).
8. All of the INPUT, OUTPUT, I-O or EXTEND phrases may be used, within the same program, in different OPEN statements for a given file. Following the initial execution of an OPEN statement for a file, each subsequent OPEN statement for the same file must be preceded by the execution of a CLOSE statement. The LOCK phrase must not be specified in this CLOSE statement.
9. The minimum and maximum record lengths for a file are defined when the file is created and must not be modified subsequently.
10. If an optional file does not exist, successful execution of an OPEN statement with I-O or EXTEND phrase causes the file to be created.
11. After execution of an OPEN statement, the contents of the data item indicated in the FILE STATUS clause (if specified) will be updated (see section "FILE STATUS clause").
12. If more than one file-name is specified, the result is the same as if a separate OPEN statement had been written for each file-name.
13. If an optional file opened with OPEN INPUT is not present then the input/output status is set to the corresponding value.
14. If files are opened with INPUT or I-O then the following applies:
    - In the case of relative and sequential files, the file position indicator is set to the first record,
    - In the case of indexed files, the file position indicator is set to the smallest possible value for the primary record key.

    For **sequentially organized files**, the following also applies:
15. REVERSED indicates that records in the file are to be read in reversed order, i.e. starting with the last record of the file.

Note:
The REVERSED phrase should not be specified for a file with nonstandard labels unless the last such label is followed by a tape marker. In all other cases, the system will read label records as if they were records.

16. NO REWIND may be specified for a tape file or for a disk storage file which was opened OUTPUT. The actions taken for tape and disk storage files are listed below:

    Tape files
    NO REWIND indicates that the tape reel is not to be rewound when the file is opened.

    Disk storage files
    NO REWIND indicates that no records existing on the file are to be overwritten. NO REWIND exists only for reasons of compatibility. OPEN OUTPUT WITH NO REWIND has the same function as OPEN EXTEND.

17. If the storage medium containing the file permits the rewind function, the following rules apply:
    a. If neither the REVERSED, nor the EXTEND, nor the NO REWIND phrase is specified, execution of the OPEN statement causes the file to be positioned at its beginning.
    b. If the NO REWIND phrase is specified, execution of the OPEN statement does not cause the file to be repositioned; instead, the file is expected to be at its beginning already.
    c. If the REVERSED phrase is specified, the records of the file are made available in reversed order; that is, starting with the last record of the file.

18. Label handling procedures are performed by the OPEN statement for all files, except those for which LABEL RECORDS ARE OMITTED is specified in the file description entry.

    If the INPUT phrase of the OPEN statement is specified, the following steps are taken:
    a. If system labels are present, they are checked.
    b. If user labels or nonstandard labels are present and an applicable USE procedure is declared, that USE procedure is executed.
    c. The file is positioned so that the first record can be read.

       If the OUTPUT phrase of the OPEN statement is supplied, execution takes place as follows:
    d. If standard labels are specified for the file, they are created and written.
    e. If an applicable USE procedure is declared, it is executed; and user or nonstandard labels are written.
    f. The record area is made available to the program in order to receive data.

       If the I-O phrase of the OPEN statement is specified, the following steps are taken:
    g. If system labels are present, they are checked.
    h. If user labels are present and an applicable USE procedure is declared, that USE procedure is executed.
    i. The file is positioned for data read or replacement operations.

       If the EXTEND phrase as well as the LABEL RECORDS STANDARD/data-name clause are specified for an OPEN statement, then execution of the OPEN statement involves the following steps:
    j. The file header labels are processed only if it is a single-reel or single-volume file.
    k. The reel/volume header labels on the last existing reel/volume are processed as in the case of the OPEN statement with INPUT phrase.
    l. The existing file trailer labels are processed as in the case of the OPEN statement with INPUT phrase. These label records are then deleted.
    m. Further processing is then the same as with OPEN OUTPUT.

    Overview of the statements permitted in OPEN mode

ACCESS clause	Statement	Open mode			
		INPUT	OUTPUT	I-O	EXTEND
SEQUENTIAL	READ	R,I,S,L		R,I,S	
	WRITE		R,I,S,L		R,I,S
	REWRITE			R,I,S	
	START	R,I		R,I	
	DELETE			R,I	
RANDOM	READ	R,I		R,I	
	WRITE		R,I	R,I	
	REWRITE			R,I	
	START				
	DELETE			R,I	
DYNAMIC	READ	R,I		R,I	
	WRITE		R,I	R,I	
	REWRITE			R,I	
	START	R,I		R,I	
	DELETE			R,I	

Table 30: Permitted input/output statements in each OPEN mode

The **permitted entries** are identified as follows:

S for sequential file organization

L for line-sequential file organization

R for relative file organization

I for indexed file organization

8.10.32 PERFORM statement

Function

The PERFORM statement is used to execute one or more procedures or a set of statements.

Format 1 of the PERFORM statement executes the specified procedures statements one time.

Format 2 of the PERFORM statement executes the specified procedures or statements a specified number of times.

Format 3 of the PERFORM statement executes the specified procedures or statements until a specified condition is true.

Format 4 of the PERFORM statement changes the values of one or more identifiers or index names in ascending or descending order, and executes a series of procedures or the specified statements one or more times, based on this action.

Format 1

"out-of-line"

```
PERFORM procedure-name-1 [{THRU | THROUGH} procedure-name-2]
```

"in-line"

```
PERFORM [imperative-statement END-PERFORM]
```

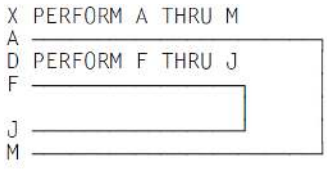
Syntax rules

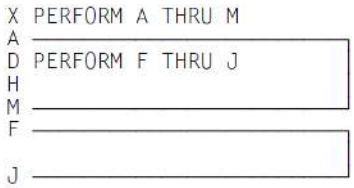
1. The words THROUGH and THRU are equivalent.
2. Where both procedure-name-1 and procedure-name-2 are specified, and either is the name of a procedure within the declarative sections, then both procedure-names must be within the same declarative section.

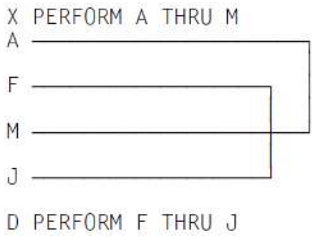
General rules

1. The END-PERFORM phrase delimits the scope of the in-line PERFORM statement.
2. An in-line PERFORM statement functions according to the following general rules for an otherwise identical out-of-line PERFORM statement, with the exception that the statements contained within the in-line PERFORM statement are executed in place of the statements contained within the range of procedure-name-1 (through procedure-name-2 if specified). Unless specially qualified by the word in-line or out-of-line, all the general rules which apply to the out-of-line PERFORM statement also apply to the in-line PERFORM statement.
3. When a PERFORM statement is executed, control is transferred to the first statement of the procedure named by procedure-name-1. An implicit return to the next executable statement following the PERFORM statement is made as follows:
   a. If procedure-name-1 is a paragraph-name and procedure-name-2 is not specified, then the return takes place after the last statement of procedure-name-1.

   b. If procedure-name-1 is a section-name, and procedure-name-2 is not specified, then the return is after the last statement of the last paragraph in procedure-name-1.
   c. If procedure-name-2 is specified and it is a paragraph-name, then the return is after the last statement of the paragraph.
   d. If procedure-name-2 is specified and it is a section-name, then the return is after the last statement of the last paragraph in the section.
   e. If an in-line PERFORM statement is specified, an execution of the PERFORM statement is completed after the last statement contained within it has been executed.
4. There is no necessary relationship between procedure-name-1 and procedure-name-2, except that a consecutive sequence of operations is to be executed beginning at the procedure named procedure-name-1 and ending with the execution of the procedure named procedure-name-2. Even GO TO and PERFORM statements may occur between procedure-name-1 and the end of procedure-name-2. If there are two or more logical paths to the return point, then procedure-name-2 may be the name of a paragraph consisting of the EXIT statement to which all of these paths must lead.
5. The statements within the range of the PERFORM statement are executed once, and control returns to the statement following the PERFORM statement.
6. The range of a PERFORM statement consists logically of all those statements that are executed as a result of executing the PERFORM statement through execution of the implicit transfer of control to the end of the PERFORM statement. The range includes all statements that are executed as the result of a transfer of control by CALL, EXIT, GO TO, and PERFORM statements in the range of the PERFORM statement, as well as all statements in declarative procedures that are executed as a result of the execution of statements in the range of the PERFORM statement. The statements in the range of a PERFORM statement need not appear consecutively in the compilation unit.
7. Statements executed as the result of a transfer of control caused by executing an EXIT PROGRAM statement are not considered to be part of the range of the PERFORM statement when:
   a. That EXIT PROGRAM statement is specified in the same program in which the PERFORM statement is specified, and
   b. The EXIT PROGRAM statement is within the range of the PERFORM statement.
8. If the range of a PERFORM statement includes another PERFORM statement, the sequence of procedures associated with the included PERFORM must itself either be totally included in, or totally excluded from, the logical sequence referred to by the first PERFORM. Thus, an active PERFORM statement, whose execution point begins within the range of another active PERFORM statement, must not allow control to pass to the exit of the other active PERFORM statement; furthermore, two or more such active PERFORM statements must not have a common exit.
   See the following illustrations for examples of legal PERFORM constructs:







9. The following information on segmentation is relevant here (for further details, refer to chapter "Segmentation").
   a. If a PERFORM statement appears in a section whose segment number is less than the value specified in the SEGMENT-LIMIT clause, then it may contain within its range only the following procedures:
      - either sections all having a segment number less than 50, or
      - sections which are wholly contained in a single independent segment whose segment number is greater than 49.
   b. If a PERFORM statement appears in a section whose segment number is greater than 49, then it may contain within its range only the following procedures:
      - either sections which all have the same segment number as the section containing the PERFORM statement or
      - sections all having a segment number less than 50.
   c. If a procedure-name in a segment with a segment number greater than 49 is referenced by a PERFORM statement in a segment with a different segment number, then the referenced segment is made available in its initial state (that is, any GO TO statements in this segment which have been modified by the execution of ALTER statements will be reset to their original values).

Example 8-59

```
PERFORM X-PAR.
ADD A TO B.
```

Let X-PAR be a paragraph-name. In this case, all statements in the paragraph named X-PAR are executed, and control is then returned to the ADD statement following the PERFORM statement.

Example 8-60

```
    ...
    PERFORM X1-PAR THRU X3-PAR.
    ...
X-KAP SECTION.
X1-PAR.
    ...
X2-PAR.
    ...
X3-PAR.
    ...
Y-KAP SECTION.
    ...
```

The PERFORM statement has the effect that all statements in the paragraphs named X1-PAR, X2-PAR and X3-PAR are executed.

Example 8-61

The same effect would result from the execution of the statement:

```
PERFORM X-KAP.
```

Format 2

"out-of-line"

```
PERFORM procedure-name-1 [{THRU | THROUGH} procedure-2] {identifier-1 | integer-1}
TIMES
```

"in-line"

```
PERFORM {identifier-1 | integer-1} TIMES imperative-statement END-PERFORM
```

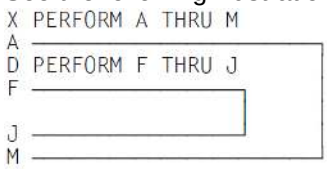
Syntax rules

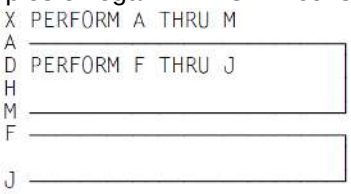
1. identifier-1 specifies an elementary numeric data item described as an integer.
2. The words THROUGH and THRU are equivalent.
3. Where both procedure-name-1 and procedure-name-2 are specified, and either is the name of a procedure in the declaratives portion of the Procedure Division, then both procedure-names must be within the same declarative section.

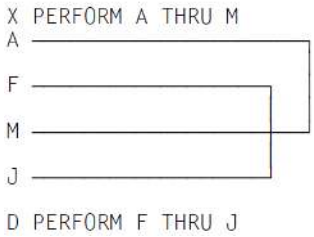
General rules

1. The END-PERFORM phrase delimits the scope of the in-line PERFORM statement.
2. An in-line PERFORM statement functions according to the following general rules for an otherwise identical out-of-line PERFORM statement, with the exception that the statements contained within the in-line PERFORM statement are executed in place of the statements contained within the range of procedure-name-1 (through procedure-name-2 if specified). Unless specifically stated to the contrary, all the general rules which apply to the out-of-line PERFORM statement also apply to the in-line PERFORM statement.
3. When a PERFORM statement is executed, control is transferred to the first statement of the procedure named by procedure-name-1. An implicit return to the next executable statement following the PERFORM statement is made as follows:
   a. If procedure-name-1 is a paragraph-name and procedure-name-2 is not specified, then the return takes place after the last statement of procedure-name-1.
   b. If procedure-name-1 is a section-name, and procedure-name-2 is not specified, then the return is after the last statement of the last paragraph in procedure-name-1.
   c. If procedure-name-2 is specified and it is a paragraph-name, then the return is after the last statement of the paragraph.
   d. If procedure-name-2 is specified and it is a section-name, then the return is after the last statement of the last paragraph in the section.
   e. If an in-line PERFORM statement is specified, an execution of the PERFORM statement is completed after the last statement contained within it has been executed.
4. procedure-name-2, except that a consecutive sequence of operations is to be executed beginning at the procedure named procedure-name-1 and ending with the execution of the procedure named procedure-name-2. Even GO TO and PERFORM statements may occur between procedure-name-1 and the end of procedure-name-2. If there are two or more logical paths to the return point, then procedure-name-2 may be the name of a paragraph consisting of the EXIT statement to which all of these paths must lead.

5. The specified set of statements is performed the number of times specified by integer-1 or by the initial value of the data item referenced by identifier-1 for that execution. Following the execution of the specified set of statements, control is transferred to the end of the PERFORM statement.
6. If at the time of the execution of a PERFORM statement, the value of the data item referenced by identifier-1 is equal to zero or is negative, control passes to the end of the PERFORM statement.
7. The range of a PERFORM statement consists logically of all those statements that are executed as a result of executing the PERFORM statement through execution of the implicit transfer of control to the end of the PERFORM statement. The range includes all statements that are executed as the result of a transfer of control by CALL, EXIT, GO TO, and PERFORM statements in the range of the PERFORM statement, as well as all statements in declarative procedures that are executed as a result of the execution of statements in the range of the PERFORM statement. The statements in the range of a PERFORM statement need not appear consecutively in the compilation unit.
8. Statements executed as the result of a transfer of control caused by executing an EXIT PROGRAM statement are not considered to be part of the range of the PERFORM statement when:
   a. That EXIT PROGRAM statement is specified in the same program in which the PERFORM statement is specified, and
   b. The EXIT PROGRAM statement is within the range of the PERFORM statement.
9. If the range of a PERFORM statement includes another PERFORM statement, the sequence of procedures associated with the included PERFORM must itself either be totally included in, or totally excluded from, the logical sequence referred to by the first PERFORM. Thus, an active PERFORM statement, whose execution point begins within the range of another active PERFORM statement, must not allow control to pass to the exit of the other active PERFORM statement; furthermore, two or more such active PERFORM statements must not have a common exit.
   See the following illustrations for examples of legal PERFORM constructs:







10. The following information on segmentation is relevant here (for further details, refer to chapter "Segmentation".
    a. If a PERFORM statement appears in a section whose segment number is less than the value specified in the SEGMENT-LIMIT clause, then it may contain within its range only the following procedures:
       - either sections all having a segment number less than 50, or
       - sections which are wholly contained in a single independent segment whose segment number is greater than 49.
    b. If a PERFORM statement appears in a section whose segment number is greater than 49, then it may contain within its range only the following procedures:
       - either sections which all have the same segment number as the section containing the PERFORM statement or

- sections all having a segment number less than 50.

c. If a procedure-name in a segment with a segment number greater than 49 is referenced by a PERFORM statement in a segment with a different segment number, then the referenced segment is made available in its initial state (that is, any GO TO statements in this segment which have been modified by the execution of ALTER statements will be reset to their original values).

Example 8-62

```
    PERFORM X-PAR 5 TIMES.
```

All statements in the paragraph named X-PAR are performed five times. Control then passes to the statement following the PERFORM statement.

Example 8-63

```
...
77 A PICTURE 9.
...
    MOVE 3 TO A.
    ...
    PERFORM X-PAR A TIMES.
    ...
X-PAR.
    ...
    ADD 1 TO A.
Y-PAR.
    ...
```

Since the value of A is 3 when the PERFORM statement is executed, the paragraph named X-PAR is executed three times.

The reference to A within X-PAR has no effect on the PERFORM statement.

Format 3

"out-of-line"

```
PERFORM procedure-name-1 [{THRU | THROUGH} procedure-name-2]
    [WITH TEST {BEFORE | AFTER}] UNTIL condition-1
```

"in-line"

```
PERFORM [WITH TEST {BEFORE | AFTER}] UNTIL condition-1
    imperative-statement END-PERFORM
```

Syntax rules

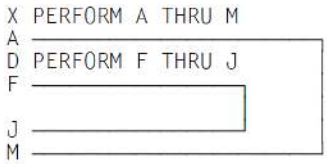
1. The words THROUGH and THRU are equivalent.

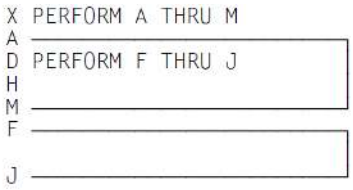
2. When both procedure-name-1 and procedure-name-2 are specified, and either is the name of a procedure within the declaratives portion of the Procedure Division, then both procedure-names must be within the same declarative section.
3. If neither TEST BEFORE nor TEST AFTER is specified, TEST BEFORE is assumed.

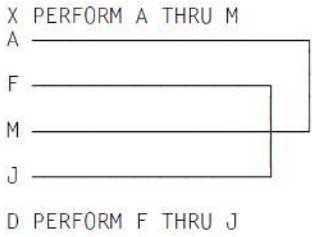
General rules

1. The END-PERFORM phrase delimits the scope of the in-line PERFORM statement.
2. An in-line PERFORM statement functions according to the following general rules for an otherwise identical out-of-line PERFORM statement, with the exception that the statements contained within the in-line PERFORM statement are executed in place of the statements contained within the range of procedure-name-1 (through procedure-name-2 if specified). Unless specially qualified by the word in-line or out-of-line, all the general rules which apply to the out-of-line PERFORM statement also apply to the in-line PERFORM statement.
3. When a PERFORM statement is executed, control is transferred to the first statement of the procedure named by procedure-name-1. An implicit return to the next executable statement following the PERFORM statement is made as follows:
    a. If procedure-name-1 is a paragraph-name and procedure-name-2 is not specified, then the return takes place after the last statement of procedure-name-1.
    b. If procedure-name-1 is a section-name, and procedure-name-2 is not specified, then the return is after the last statement of the last paragraph in procedure-name-1.
    c. If procedure-name-2 is specified and it is a paragraph-name, then the return is after the last statement of the paragraph.
    d. If procedure-name-2 is specified and it is a section-name, then the return is after the last statement of the last paragraph in the section.
    e. If an in-line PERFORM statement is specified, an execution of the PERFORM statement is completed after the last statement contained within it has been executed.
4. There is no necessary relationship between procedure-name-1 and procedure-name-2, except that a consecutive sequence of operations is to be executed beginning at the procedure named procedure-name-1 and ending with the execution of the procedure named procedure-name-2. Even GO TO and PERFORM statements may occur between procedure-name-1 and the end of procedure-name-2. If there are two or more logical paths to the return point, then procedure-name-2 may be the name of a paragraph consisting of the EXIT statement to which all of these paths must lead.
5. The statements within the range of the PERFORM statement are executed once, and control returns to the statement following the PERFORM statement.
6. The range of a PERFORM statement consists logically of all those statements that are executed as a result of executing the PERFORM statement through execution of the implicit transfer of control to the end of the PERFORM statement. The range includes all statements that are executed as the result of a transfer of control by CALL, EXIT, GO TO, and PERFORM statements in the range of the PERFORM statement, as well as all statements in declarative procedures that are executed as a result of the execution of statements in the range of the PERFORM statement. The statements in the range of a PERFORM statement need not appear consecutively in the compilation unit.
7. Statements executed as the result of a transfer of control caused by executing an EXIT PROGRAM statement are not considered to be part of the range of the PERFORM statement when:
    a. That EXIT PROGRAM statement is specified in the same program in which the PERFORM statement is specified, and
    b. The EXIT PROGRAM statement is within the range of the PERFORM statement.

8. If the range of a PERFORM statement includes another PERFORM statement, the sequence of procedures associated with the included PERFORM must itself either be totally included in, or totally excluded from, the logical sequence referred to by the first PERFORM. Thus, an active PERFORM statement, whose execution point begins within the range of another active PERFORM statement, must not allow control to pass to the exit of the other active PERFORM statement; furthermore, two or more such active PERFORM statements must not have a common exit.
   See the following illustrations for examples of legal PERFORM constructs:







9. The specified set of statements is performed until the condition specified by the UNTIL phrase is true. When the condition is true, control is transferred to the end of the PERFORM statement. If the condition is true when the PERFORM statement is entered, and the TEST BEFORE phrase is specified or implied, no transfer to procedure-name-1 takes place, and control is passed to the end of the PERFORM statement.
10. If the TEST AFTER phrase is specified, the PERFORM statement functions as if the TEST BEFORE phrase were specified except that the condition is tested after the specified set of statements has been executed.
11. Any subscripting or reference modification associated with the operands specified in condition-1 is evaluated each time the condition is tested.
12. The following information on segmentation is relevant here (for further details, refer to chapter "Segmentation").
    a. If a PERFORM statement appears in a section whose segment number is less than the value specified in the SEGMENT-LIMIT clause, then it may contain within its range only the following procedures:
       - either sections all having a segment number less than 50, or
       - sections which are wholly contained in a single independent segment whose segment number is greater than 49.
    b. If a PERFORM statement appears in a section whose segment number is greater than 49, then it may contain within its range only the following procedures:
       - either sections which all have the same segment number as the section containing the PERFORM statement or
       - sections all having a segment number less than 50.
    c. If a procedure-name in a segment with a segment number greater than 49 is referenced by a PERFORM statement in a segment with a different segment number, then the referenced segment is made available in its initial state (that is, any GO TO statements in this segment which have been modified by the execution of ALTER statements will be reset to their original values).

Example 8-64

```
    PERFORM X-PAR UNTIL A GREATER THAN 3.
    ...
X-PAR.
    ...
    COMPUTE A = A + 1.
    ...
Y-PAR.
    ...
```

Assume A = 1, when the PERFORM statement is initiated. In this case, the statements in the paragraph named X-PAR are performed three times:

The first time X-PAR is performed, A is set to 2.

The second time X-PAR is performed, A is set to 3.

The third time X-PAR is performed, A is set to 4.

Since 4 is greater than 3, the condition specified in the PERFORM statement is true. Thus, control passes to the statement which follows the PERFORM statement.

Format 4

"out-of-line"

```
PERFORM procedure-name-1 [{THRU | THROUGH} procedure-name-2] [WITH TEST {BEFORE |
AFTER}]

     VARYING {index-1 | identifier-2} FROM {index-2 | literal-1 | identifier-3} BY
     {literal-2 | identifier-4} UNTIL condition-1

    [AFTER {index-3 | identifier-5} FROM {index-4 | literal-3 | identifier-6} BY
     {literal-4 | identifier-7} UNTIL condition-2]...
```

"in-line"

```
PERFORM [WITH TEST {BEFORE | AFTER}]

    VARYING {index-1| identifier-2} FROM {index-2 | literal-1 | identifier-3} BY
    {literal-2 | identifier-4} UNTIL condition-1

    imperative-statement END-PERFORM
```

Syntax rules

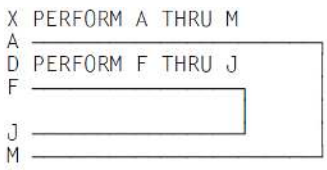
1. The words THROUGH and THRU are equivalent.
2. When both procedure-name-1 and procedure-name-2 are specified, and either is the name of a procedure within the declaratives portion of the Procedure Division, then both procedure-names must be within the same declarative section.
3. If neither TEST BEFORE nor TEST AFTER is specified, TEST BEFORE is assumed.

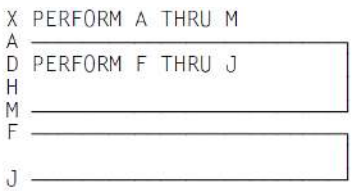
4. Each identifier represents a numeric item described as an integer. However, the compiler allows each identifier to be described as a non-integer numeric item.
5. Each literal must be numeric.
6. The literal in the associated BY phrase must be a non-zero integer.
7. If an index-name is specified in the VARYING or AFTER phrase, then:
   a. The identifier in the associated FROM and BY phrases must reference an integer data item.
   b. The literal in the associated FROM phrase must be a positive integer.
   c. The literal in the associated BY phrase must be a non-zero integer.
8. If an index-name is specified in the FROM phrase, then:
   a. The identifier in the associated VARYING or AFTER phrase must reference an integer data item.
   b. The identifier in the associated BY phrase must reference an integer data item.
   c. The literal in the associated BY phrase must be an integer.
9. condition-1, condition-2 may be any conditional expression.
10. A maximum of 6 AFTER phrases are permitted in the out-of-line PERFORM statement.

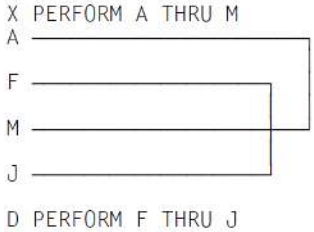
General rules

1. The data items referenced by identifier-4 and identifier-7 must not have a zero value.
2. If an index-name is specified in the VARYING or AFTER phrase, and an identifier is specified in the associated FROM phrase, the data item referenced by the identifier must have a positive value.
3. Format 4 of the PERFORM statement is used to increment or decrement the values of one or more identifiers or index-names in a specific manner while the PERFORM statement is executing. Execution depends on the number of identifiers or index-names that are varied. The following rules describe what happens when one, two and three identifiers or index-names are varied.
4. The END-PERFORM phrase delimits the scope of the in-line PERFORM statement.
5. An in-line PERFORM statement functions according to the following general rules for an otherwise identical out-of-line PERFORM statement, with the exception that the statements contained within the in-line PERFORM statement are executed in place of the statements contained within the range of procedure-name-1 (through procedure-name-2 if specified). Unless specially qualified by the word in-line or out-of-line, all the general rules which apply to the out-of-line PERFORM statement also apply to the in-line PERFORM statement.
6. When a PERFORM statement is executed, control is passed to the first statement of the procedure named by procedure-name-1. An implicit return to the next executable statement following the PERFORM statement is made as follows:
   a. If procedure-name-1 is a paragraph-name and procedure-name-2 is not specified, then the return takes place after the last statement of procedure-name-1.
   b. If procedure-name-1 is a section-name, and procedure-name-2 is not specified, then the return is after the last statement of the last paragraph in procedure-name-1.
   c. If procedure-name-2 is specified and it is a paragraph-name, then the return is after the last statement of the paragraph.
   d. If procedure-name-2 is specified and it is a section-name, then the return is after the last statement of the last paragraph in the section.
   e. If an in-line PERFORM statement is specified, an execution of the PERFORM statement is completed after the last statement contained within it has been executed.

7. There is no necessary relationship between procedure-name-1 and procedure-name-2, except that a consecutive sequence of operations is to be executed beginning at the procedure named procedure-name-1 and ending with the execution of the procedure named procedure-name-2. Even GO TO and PERFORM statements may occur between procedure-name-1 and the end of procedure-name-2. If there are two or more logical paths to the return point, then procedure-name-2 may be the name of a paragraph consisting of the EXIT statement to which all of these paths must lead.
8. In the following discussion, every reference to identifier as the object of the VARYING, AFTER, and FROM (current value) phrases also refers to index-names.
   a. If index-name-1 or index-name-3 is specified, the value of the associated index at the beginning of the PERFORM statement must be set to an occurrence number of an element in the table.
   b. If index-name-2 or index-name-4 is specified, the value of the data item referenced by identifier-2 or identifier-5 at the beginning of the PERFORM statement must be equal to an occurrence number of an element in a table associated with index-name-2 or index-name-4.
   c. Subsequent augmentation, as described below, of index-name-1 or index-name-3 must not result in the associated index being set to a value outside the range of the table associated with index-name-1 or index-name-3; except that, at the completion of the PERFORM statement, the index associated with index-name-1 may contain a value that is outside the range of the associated table by one increment or decrement value.
   d. If identifier-2 or identifier-5 is subscripted, the subscripts are evaluated each time the content of the data item referenced by the identifier is set or augmented.
   e. If identifier-3, identifier-4, identifier-6, or identifier-7 is subscripted, the subscripts are evaluated each time the content of the data item referenced by the identifier is used in a setting or augmenting operation.
9. Statements executed as the result of a transfer of control caused by executing an EXIT PROGRAM statement are not considered to be part of the range of the PERFORM statement when:
   a. That EXIT PROGRAM statement is specified in the same program in which the PERFORM statement is specified, and
   b. The EXIT PROGRAM statement is within the range of the PERFORM statement.
10. If the range of a PERFORM statement includes another PERFORM statement, the sequence of procedures associated with the included PERFORM must itself either be totally included in, or totally excluded from, the logical sequence referred to by the first PERFORM. Thus, an active PERFORM statement, whose execution point begins within the range of another active PERFORM statement, must not allow control to pass to the exit of the other active PERFORM statement; furthermore, two or more such active PERFORM statements must not have a common exit.
    See the following illustrations for examples of legal PERFORM constructs:







11. If the TEST BEFORE phrase is specified or implied:

    When the data item referenced by **one** identifier is varied:

    - The content of the data item referenced by identifier-2 is set to literal-1 or the current value of the data item referenced by identifier-3 at the point of initial execution of the PERFORM statement.
    - Then, if the condition of the UNTIL phrase is false, the specified set of statements is executed once.
    - The value of the data item referenced by identifier-2 is augmented by the specified increment or decrement value (literal-2 or the value of the data item referenced by identifier-4) and condition-1 is evaluated again .
    - This cycle continues until this condition is true, at which point control is transferred to the end of the PERFORM statement.
    - If condition-1 is true at the beginning of execution of the PERFORM statement, control is transferred to the end of the PERFORM statement.

    When the data items referenced by **two** identifiers are varied:

    - At the start of execution of the PERFORM statement, the content of the data item referenced by identifier-2 is set to literal-1 or the current value of the data item referenced by identifier-3.
    - Then the content of the data item referenced by identifier-5 is set to literal-3 or the current value of the data item referenced by identifier-6.
    - After the contents of the data items referenced by the identifiers have been set, condition-1 is evaluated; if true, control is transferred to the end of the PERFORM statement.
    - If condition-1 is false, condition-2 is evaluated.
    - If condition-2 is false, the specified set of statements is executed once.
    - Then the content of the data item referenced by identifier-5 is augmented by literal-4 or the content of the data item referenced by identifier-7 and condition-2 is evaluated again.
    - This cycle of evaluation and augmentation continues until condition-2 is true.
    - When condition-2 is true, the content of the data item referenced by identifier-2 is augmented by literal-2 or the content of the data item referenced by identifier-4, and the content of the data item referenced by identifier-5 is set to literal-3 or the current value of the data item referenced by identifier-6.
    - condition-1 is reevaluated.
    - The PERFORM statement is completed if condition-1 is true; if not, the cycle continues until condition-1 is true.

    At the termination of the PERFORM statement, the data item referenced by identifier-5 contains literal-3 or the current value of the data item referenced by identifier-6. The data item referenced by identifier-2 contains a value that exceeds the last used setting by one increment or decrement value, unless condition-1 was true when the PERFORM statement was entered, in which case the data item referenced by identifier-2 contains literal-1 or the current value of the data item referenced by identifier-3.

12. If the TEST AFTER phrase is specified or implied:

    When the data item referenced by **one** identifier is varied:

    - The content of the data item referenced by identifier-2 is set to literal-1 or the current value of the data item referenced by identifier-3 at the point of execution of the PERFORM statement.
    - Then the specified set of statements is executed once and condition-1 of the UNTIL phrase is tested.
    - If the condition is false, the value of the data item referenced by identifier-2 is augmented by the specified increment or decrement value (literal-2 or the value of the data item referenced by identifier-4) and the specified set of statements is executed again.

- The cycle continues until condition-1 is tested and found to be true, at which point control is transferred to the end of the PERFORM statement.

When the data items referenced by **two** identifiers are varied:

- The content of the data item referenced by identifier-2 is set to literal-1 or the current value of the data item referenced by identifier-3.
- Then the content of the data item referenced by identifier-5 is set to literal-3 or the current value of the data item referenced by identifier-6.
- The specified set of statements is then executed.
- condition-2 is then evaluated.
- If condition-2 is false, the content of the data item referenced by identifier-5 is augmented by literal-4 or the content of data item referenced by identifier-7 and the specified set of statements is again executed.
- The cycle continues until condition-2 is again evaluated and found to be true.
- At this time, condition-1 is evaluated.
- If condition-1 is false, the content of the data item referenced by identifier-2 is augmented by literal-2 or the content of the data item referenced by identifier-4, and the content of the data item referenced by identifier-5 is set to literal-3 or the current value of the data item referenced by identifier-6.
- The specified set of statements is then executed again.
- This cycle continues until condition-1 is again evaluated and found to be true, at which time control is transferred to the end of the PERFORM statement.

After the completion of the PERFORM statement, each data item varied by an AFTER or VARYING phrase contains the same value it contained at the end of the most recent execution of the specified set of statements.

13. During the execution of the specified set of statements associated with the PERFORM statement, any change to the VARYING variable (the data item referenced by identifier-2 and index-name-1), the BY variable (the data item referenced by identifier-4), the AFTER variable (the data item referenced by identifier-5 and index-name-3), or the FROM variable (the data item referenced by identifier-3 and index-name-2) will be taken into consideration and will affect the operation of the PERFORM statement.

    When the data items referenced by two identifiers are varied, the data item referenced by identifier-5 goes through a complete cycle (FROM, BY, UNTIL) each time the content of the data item referenced by identifier-2 is varied. When the contents of three or more data items referenced by identifiers are varied, the mechanism is the same as for two identifiers except that the data item being varied by each AFTER phrase goes through a complete cycle each time the data item being varied by the preceding AFTER phrase is augmented.

14. The following information on segmentation is relevant here (for further details, refer to chapter "Segmentation").
    a. If a PERFORM statement appears in a section whose segment number is less than the value specified in the SEGMENT-LIMIT clause, then it may contain within its range only the following procedures:
        - either sections which all have a segment number less than 50, or
        - sections which are wholly contained in a single independent segment whose segment number is greater than 49.
    b. If a PERFORM statement appears in a section whose segment number is greater than 49, then it may contain within its range only the following procedures:
        - either sections which all have the same segment number as the section containing the PERFORM statement or
        - sections all having a segment number less than 50.

c. If a procedure-name in a segment with a segment number greater than 49 is referenced by a PERFORM statement in a segment with a different segment number, then the referenced segment is made available in its initial state (that is, any GO TO statements in this segment which have been modified by the execution of ALTER statements will be reset to their original values).

Example 8-65

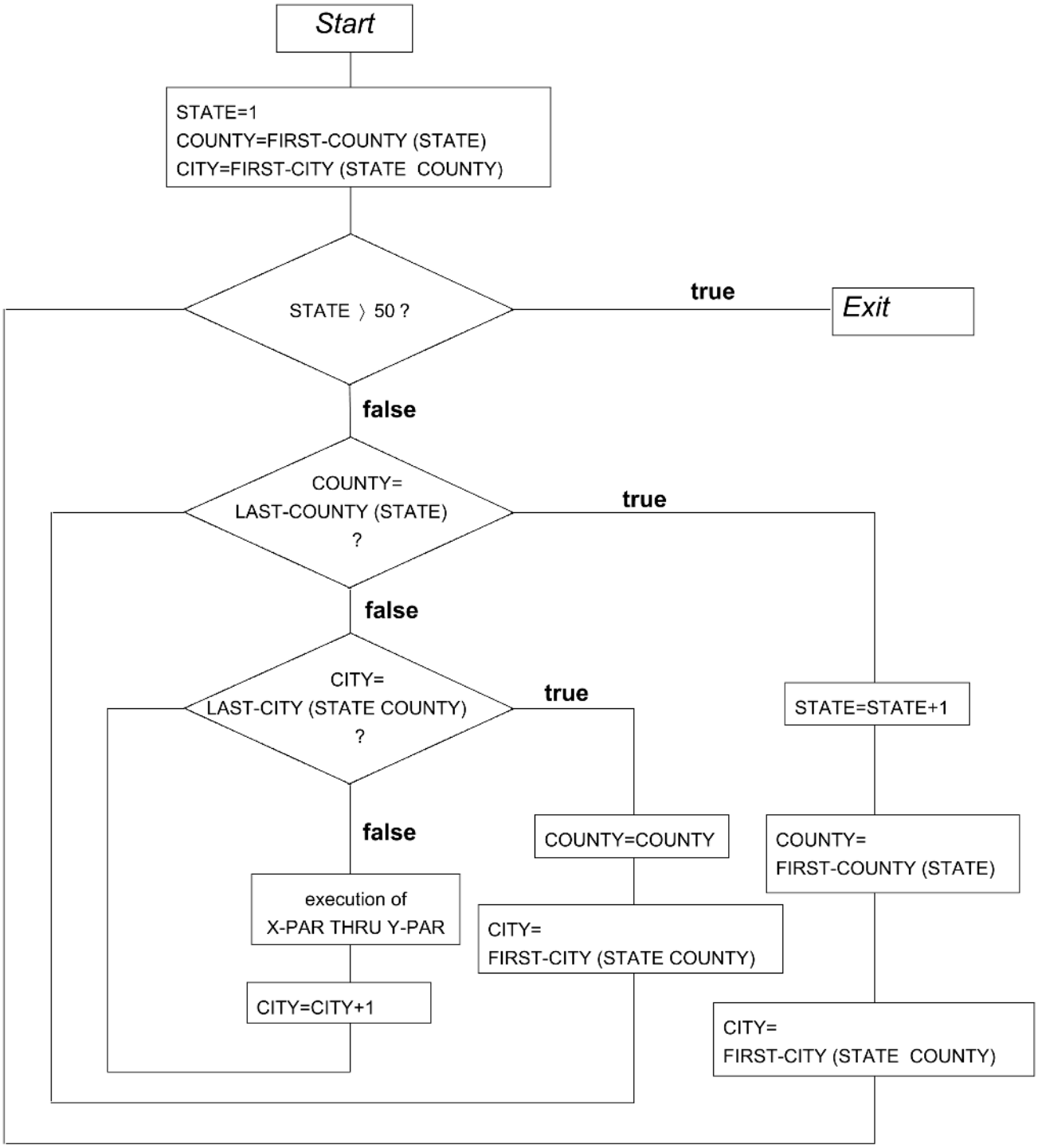
```
    ...
01  STATES.
    02  COUNTIES OCCURS 51 INDEXED STATE.
       03  FIRST-COUNTY     PIC 9(3).
       03  LAST-COUNTY     PIC 9(3).
       03  CITIES OCCURS 100 INDEXED COUNTIES.
           04  FIRST-CITY   PIC 9(3).
           04  LAST-CITY    PIC 9(3).
01  CITY                    PIC 9(3).
     ...
PROCEDURE DIVISION.
K1.
    PERFORM X-PAR THRU Y-PAR
       VARYING STATE FROM 1 BY 1
           UNTIL STATE IS GREATER THAN 50;
       AFTER COUNTY FROM FIRST-COUNTY (STATE) BY 1
           UNTIL COUNTY IS EQUAL TO LAST-COUNTY (STATE)
       AFTER CITY FROM FIRST-CITY (STATE COUNTY)
           UNTIL CITY IS EQUAL TO LAST-CITY (STATE COUNTY)
. . .
.
```

Example 8-66

for format 3 and 4, WITH TEST AFTER/BEFORE

```
IDENTIFICATION DIVISION.
PROGRAM-ID. PWTA.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
    TERMINAL IS T.
DATA DIVISION.
WORKING-STORAGE SECTION.
77 N PIC 9.
77 K PIC 9(3).
77 Z PIC 9(4).
77 E PIC 9(4).
PROCEDURE DIVISION.
P1 SECTION.
```

Flowchart:


Start
STATE=1
COUNTY=FIRST-COUNTY (STATE)
CITY=FIRST-CITY (STATE COUNTY)
STATE 〉 50 ?
true
Exit
false
COUNTY=
LAST-COUNTY (STATE)
?
true
false
CITY=
LAST-CITY (STATE COUNTY)
?
true
STATE=STATE+1
false
COUNTY=COUNTY
COUNTY=
FIRST-COUNTY (STATE)
execution of
X-PAR THRU Y-PAR
CITY=
FIRST-CITY (STATE COUNTY)
CITY=CITY+1
CITY=
FIRST-CITY (STATE COUNTY)

```
COMPUTATION.
    MOVE 0 TO Z.
    DISPLAY "Enter single-digit number as upper bound N" UPON T.
    ACCEPT N FROM T.
    PERFORM WITH TEST AFTER VARYING K FROM 1 BY 1 UNTIL K >= N
        COMPUTE E = K ** 3
        ADD E TO Z
    END-PERFORM
    DISPLAY "Result =" Z UPON T.
FIN.
    STOP RUN.
```

The program calculates the nth sum of the cube of an integer K. Only after the COMPUTE and ADD statements are executed is there a test to see whether the termination condition is satisfied.

If TEST BEFORE is specified, the termination condition is tested first. If K <= N, the in-line statements are executed. If K > N, the PERFORM statement is terminated with END-PERFORM.

8.10.33 RAISE statement

Function

The RAISE statement causes a particular exception condition to be triggered.

Format

```
RAISE EXCEPTION exception-condition-name
```

Syntax rule

1. exception-condition-name must be one of the names listed in table 45 in section "Exception conditions and exception statuses".

General rule

1. The exception condition corresponding to exception-condition-name is triggered.

8.10.34 READ statement

Function

The READ statement makes the following available to the program:

- for sequential access: the next record in the file (for **sequential, relative and indexed file organization**).
- for random access: a record of a file (for **relative and indexed file organization**) specified by the key.

Format 1 **sequential file access**
applies to sequential file organization; used with relative and indexed files to read records sequentially in sequential or dynamic access mode.

Format 2 **random file access**
applies to relative and indexed file organization; used for random reading of records (RELATIVE KEY for **relative files**; RECORD KEY or ALTERNATE RECORD KEY for **indexed files**) in random or dynamic access mode.

> The extension (WITH NO LOCK) in both formats, which is effective during shared updating of files, is described in the "COBOL2000 User Guide" [1].

Format 1 sequential file access

```
READ file-name [WITH NO LOCK] {NEXT | PREVIOUS } RECORD [INTO identifier]
    [AT END imperative-statement-1]
    [NOT AT END imperative-statement-2]
    [END-READ]
```

Syntax rules

1. In the READ statement, the receiving operand references the same storage area as the record description entry.
2. If no USE procedure is declared for the file, AT END must be specified in the READ statement.
3. PREVIOUS may only be specified for indexed and relative files.
4. If ACCESS MODE RANDOM is entered for the file in the FILE-CONTROL paragraph then AT END, NOT AT END, NEXT and PREVIOUS must not be specified.
5. If ACCESS MODE SEQUENTIAL is specified for the file in the FILE-CONTROL paragraph and neither NEXT nor PREVIOUS is specified then NEXT is assumed.
6. If the following conditions hold simultaneously, NEXT is assumed:
   - ACCESS MODE DYNAMIC is entered for the file in the FILE-CONTROL paragraph.
   - Neither NEXT nor PREVIOUS is specified.
   - AT END or NOT AT AND is specified.

General rules

The following applies for **sequential, relative and indexed file organization**:

1. An OPEN statement with the INPUT or I-O phrase must be executed for a file before the READ statement can be executed.
2. The execution of a READ statement causes the contents of the data item specified in the FILE STATUS clause of the related file description entry to be updated (see section "FILE STATUS clause").
3. When the logical records of a file are described with more than one record description, these records automatically share the same storage area; this is equivalent to an implicit redefinition of the area. The contents of any data items which lie beyond the range of the current record are undefined after execution of the READ statement.
4. The INTO phrase may be specified in a READ statement:
   - if only one record description is subordinate to the file description entry, or
   - if all record-names associated with the file-name and the data item referenced by the identifier represent group items or alphanumeric elementary items.

   If identifier is a strongly typed group item, only one record description may be subordinate to the file description entry. This record description must be a strongly typed group item of the same type as identifier.
5. If, after a READ statement without the INTO phrase, the user wishes to explicitly address the input area, then he is responsible for using the correct record description (i.e. the one which matches the length of the record which was read).
6. If no exception condition exists, the record is available in the record area, and any implicit move as a result of the INTO phrase is executed.
7. Whether execution of the READ statement continues depends on whether AT END or NOT AT END is specified (see section "At end condition").
8. After unsuccessful execution of a READ statement, the content of the input area belonging to the file and the file position indicator are undefined and the I/O status indicates that no valid next record has been read.
9. If the number of character positions in a read record is less than the minimum length specified in the record descriptions, the contents to the right of the last valid character will be unpredictable.
   If the number of character positions is greater than the maximum length specified in the record descriptions, the record will be truncated to the right of the maximum length.
   In either case the read operation is successful, but a FILE STATUS is set to indicate the occurrence of a record length conflict.
10. The execution of a READ statement with the INTO phrase is equivalent to:

    ```
    READ file-name
    MOVE record-name TO identifier
    ```

    The MOVE operation takes place according to the rules for the MOVE statement without the CORRESPONDING phrase. After the READ statement with INTO phrase has been successfully executed, the record is available both in the input area and in the area specified by the identifier. The length of the source field is determined by the length of the record that is read (see section "RECORD clause").
    If the file description entry contains a RECORD-IS-VARYING clause, the implicitly executed transfer operation is a group item transfer.
    If the execution of the READ statement was unsuccessful, the implied MOVE statement does not occur.
    The index for the identifier is calculated after execution of the READ statement and immediately before the data transfer.
11. The file position indicator determines whether a record is supplied and, if so, which record this is:
    a. If the file position indicator indicates “invalid” then the READ statement is unsuccessful.
    b. If the file position indicator was set by a preceding OPEN or START statement then the following result is supplied:

- If NEXT was specified either explicitly or implicitly then the first record of the file with a record number or record key file position indicator is supplied.
- If PREVIOUS was specified then the first record of the file with a record number or record key file position indicator is supplied.

  Note:
  After OPEN, READ PREVIOUS for a relative file supplies the first record and for indexed files it usually supplies the End condition.

c. If the file position indicator was set by a preceding READ statement then the following result is supplied:
   - If NEXT was specified either explicitly or implicitly then the first record of the file with a record number or record key > file position indicator is supplied.
   - If PREVIOUS was specified then the first record of the file with a record number or record key < file position indicator is supplied.

If a record is supplied, it is available in the record area of the file file-name. The file position indicator is set to the record number or record key. If no record is supplied, the file position indicator indicates “invalid” and processing continues in accordance with rule 5.

For **sequentially organized files**, the following also applies:

12. After the at end condition has occurred, no READ statement may be issued for the file until a successful CLOSE statement has been carried out, followed by a successful OPEN statement for that file.
13. If the physical end of a reel or disk storage volume is encountered during execution of a READ statement for a multi-volume file, then the following operations take place:
    a. The standard volume trailer label routine and, if specified by an appropriate USE procedure, the user volume trailer label routine are executed. The order in which the two routines are performed is specified by the USE procedure.
    b. A volume switch is executed.
    c. The standard volume header label routine and, if specified, the user volume header label routine are executed. Again, the order in which these routines are executed is specified by the USE procedure.

For **relative and indexed files**, the following also applies:

14. If the RELATIVE KEY phrase is specified, the execution of the READ statement moves the relative record number of the record made available to the relative key data item according to the rules for the MOVE statement.
15. After the at end condition has occurred, no further READ statement may be issued for the file until either repositioning has been effected with START or the file has been successfully closed and reopened.
16. In the case of an indexed file with sequential access, the sequence of records whose keys permit duplicates is defined by the order in which these duplicate keywords were generated by means of WRITE or REWRITE statements. When such files are read with an explicit or implicit NEXT, the read operation starts at the next written record. If PREVIOUS is specified then the operation starts at the previously written record.

Format 2 random file access

```
READ file-name [WITH NO LOCK] RECORD [INTO identifier]
    [KEY IS data-name]
    [INVALID KEY imperative-statement-1]
```

```
[NOT INVALID KEY imperative-statement-2]

[END-READ]
```

Syntax rules

1. In the READ statement, the receiving operand references the same storage area as the record description entry.
2. If no USE procedure is present for the file, INVALID KEY must be specified.
3. The `KEY IS data-name` phrase is only allowed for **indexed file organization**.
4. data-name must be the name of a RECORD KEY or ALTERNATE RECORD KEY field declared for this file.
5. data-name may be qualified.

General rules

1. An OPEN statement with the INPUT or I-O phrase must be executed for a file before the READ statement can be executed.
2. When the logical records of a file are described with more than one record description, these records automatically share the same storage area; this is equivalent to an implicit redefinition of the area.
3. The INTO phrase may be specified in a READ statement:
   - if only one record description is subordinate to the file description entry, or
   - if all record-names associated with the file-name and the data item referenced by the identifier represent group items or alphanumeric elementary items.

   If identifier is a strongly typed group item, only one record description may be subordinate to the file description entry. This record description must be a strongly typed group item of the same type as identifier.
4. The execution of a READ statement with the INTO phrase is equivalent to:

   ```
   READ file-name
   MOVE record-name TO identifier
   ```

   The MOVE operation takes place according to the rules for the MOVE statement without the CORRESPONDING phrase. After the READ statement with INTO phrase has been successfully executed, the record is available both in the input area and in the area specified by the identifier. The length of the source field is determined by the length of the record that is read (see section "RECORD clause"). If the execution of the READ statement was unsuccessful, the implied MOVE statement does not occur.

   The index for the identifier is calculated after execution of the READ statement and immediately before the implicit MOVE.
5. If the input area is to be explicitly referenced following a READ statement without the INTO phrase, it is the user's responsibility to ensure that the correct record description entry (i.e. corresponding to the length of the read record) is used.
6. If an invalid key condition does not occur during the execution of a READ statement, the INVALID KEY phrase is ignored, if specified, and the following actions occur:
   a. The I-O status for file-name is updated.
   b. If some other exception condition occurs, control is transferred to the USE procedure.
   c. If no exception condition exists, the record is made available in the record area, and any implicit move as a result of the INTO phrase is executed. Control is transferred to the end of the READ statement or to imperative-statement-2 of the NOT INVALID KEY phrase, if specified. In the latter case, execution

continues according to the rules for the specified imperative-statement, and control is transferred to some other branch in the program or to the end of the READ statement.

7. Following an unsuccessful READ statement, the contents of the input area associated with the file and the file position indicator are undefined.
8. If the number of character positions in a record is less than the minimum length specified in the record descriptions, the contents to the right of the last valid character will be unpredictable.
   If the number of character positions is greater than the maximum length specified in the record descriptions, the record will be truncated to the right of the maximum length.In either case the read operation is successful, but a FILE STATUS is set to indicate the occurrence of a record length conflict.

   For **relative file organization**, the following also applies:
9. If a file is read randomly, a search is executed for a record whose relative record number is equal to the value of the RELATIVE KEY field for this file. If there is no such record in the file, an invalid key condition occurs and execution of the READ statement is unsuccessful (see section "Invalid key condition").
10. If an optional file does not exist, the invalid key condition occurs and execution of the READ statement is unsuccessful.
    For **indexed file organization**, the following also applies:
11. Execution of the READ statement sets the file position indicator to value in the key of reference. This value is compared with the value contained in the corresponding data item of the stored records in the file until the first record having an equal value is found. In the case an alternate key is used as key of reference and if duplicates are available for this alternate key, the first record written is made available with this very key value. The record thus found is made available in the record area associated with file-name. If no record can be identified in this way, the invalid key condition occurs, and execution of the READ statement is unsuccessful (see "Invalid key condition").
12. If KEY is specified, data-name is established as the key of reference for the current read operation. In the case of dynamic access, this key of reference is also used for all subsequent format 1 READ statements until a different key of reference is established for the file.
13. If KEY is not specified, the primary key is established as the key of reference. In the case of dynamic access, this key of reference is also used for all subsequent format 1 READ statements until a different key of reference is established for the file.

8.10.35 RELEASE statement

Function

The RELEASE statement transfers records from the input file to the sort-file. It can only be used during a sort operation.

Format

```
RELEASE sort-record-name [FROM identifier]
```

Syntax rules

1. sort-record-name must be the name of a logical record in the associated sort-file description.
2. sort-record-name and identifier must not refer to the same internal storage area.

General rules

1. Execution of a RELEASE statement has the effect that the record specified by sort-record-name is transferred to the sort-file, to be further processed by the SORT routine of the system.
2. The FROM phrase makes the RELEASE statement equivalent to a MOVE statement followed by a RELEASE statement.
   When this phrase is written, data is moved from the area specified by identifier to the area identified by sort-record-name, and is then released to the sort-file. The move takes place according to the rules which govern the MOVE statement without the CORRESPONDING phrase.
3. A RELEASE statement may be used only within an input procedure in connection with a SORT statement for the sort file containing sort-record-name.
4. After a RELEASE statement is executed, the logical record is no longer available in the record area, unless the associated sort-file, appears in a SAME RECORD AREA clause. The logical record is, also at program execution time, available as a record of other files specified in the SAME RECORD AREA clause as the associated sort-file; and it is available to the file associated with sort-record-name. When control is transferred to the input procedure, that file consists of all those records which were passed by means of the execution of RELEASE statements.
5. After a RELEASE statement with the FROM phrase is executed, the record is still available in "identifier".

8.10.36 RESUME statement

Function

The RESUME statement causes a branch to an explicit branch address or to the statement after the statement which led to a USE procedure being executed.

Format

```
RESUME AT {NEXT STATEMENT | procedure-name}
```

Syntax rules

1. The RESUME statement may only be specified in a USE procedure of format 2 or 4.
2. procedure-name must be defined outside the declaratives.

General rules

1. If execution of the RESUME statement would lead to a global USE procedure being exited, the RESUME statement is treated like CONTINUE.
2. The RESUME statement with the NEXT STATEMENT phrase may only be executed if a USE procedure was activated by an exception condition or an input/output exception condition being triggered, i.e. the statement may not be executed by a PERFORM statement from outside the declaratives.
3. If NEXT STATEMENT is specified, the program run is continued with an assumed CONTINUE statement. If not explicitly defined differently for a statement, this is assumed to be directly behind the statement which activated the USE procedure. For statements with scope terminators the CONTINUE statement is assumed to be directly behind the explicit or implicit scope terminator.
4. If procedure-name is specified, the program run is continued at procedure-name.

8.10.37 RETURN statement

Function

The RETURN statement obtains individual records in sorted order from the sort-file.

Format

```
RETURN sort-file-name RECORD [INTO identifier]
     AT END imperative-statement-1
    [NOT AT END imperative-statement-2]
    [END-RETURN]
```

Syntax rules

1. sort-file-name must be defined in a sort-file description entry.
2. The storage area associated with identifier and the record area associated with sort-file-name must not be the same storage area.

General rules

1. Execution of a RETURN statement has the effect that the next record is made available according to the order specified by the keys of the SORT/MERGE statement. Then it can be processed in the record areas of the sort-file.
2. If more than one record description is supplied for the logical record of a file, these records will automatically share the same storage area; this corresponds to an implicit redefinition of the area. The contents of any data item which may be outside the area of the current record will be undefined after a RETURN statement is executed.
3. The INTO phrase may be specified in two cases:
   - if only one record description is subordinate to the sort-merge file description entry,
   - if all record names associated with file-name and the data item referenced by identifier describe a group item of an elementary alphanumeric item.

   If identifier is a strongly typed group item, only one record description may be subordinate to the sort-merge file description entry. This record description must be a strongly typed group item of the same type as identifier.
4. The INTO phrase, when specified, makes the RETURN statement equivalent to a RETURN statement followed by a MOVE statement.
5. When this phrase is written, records are returned from the sort-file and then moved to the area specified by identifier; identifier must not refer to a data item within the record of that sort-file. The move is made according to the rules specified for a MOVE statement without CORRESPONDING phrase. The size of the current record is determined by rules specified for the RECORD clause (see section "RECORD clause").
   If the sort file description entry contains a RECORD IS VARYING clause, the implied move is a group move.
6. The MOVE statement is not carried out if an at end condition occurs.
7. Any subscripting or indexing of identifier is evaluated after the record has been obtained and immediately before it is moved into the data area.
8. If the INTO phrase was used, the data will be available both in the record area and in the data item specified by identifier.

9. The execution of the RETURN statement causes the next existing record in the file referenced by file-name-1, as determined by the keys listed in the SORT or MERGE statement, to be made available in the record area associated with file-name-1.If no next logical record exists in the file referenced by file-name-1, the at end condition exists. Execution continues according to the rules for each statement specified in imperative-statement-1. If a procedure branching or conditional statement which causes explicit transfer of control is executed, control is transferred according to the rules for that statement; otherwise, upon completion of the execution of imperative-statement-1, control is transferred to the end of the RETURN statement and the NOT AT END phrase is ignored, if specified. When the at end condition occurs, execution of the RETURN statement is unsuccessful and the contents of the record area associated with file-name-1 are undefined. After the execution of imperative-statement-1 in the AT END phrase, no RETURN statement may be executed as part of the current output procedure.
10. If an at end condition does not occur during the execution of a RETURN statement, then after the record is made available and after executing any implicit move resulting from the presence of an INTO phrase, control is transferred to imperative-statement-2, if NOT AT END was specified; otherwise, control is transferred to the end of the RETURN statement.
11. A RETURN statement may be used only within the range of the output procedure associated with a SORT /MERGE statement for sort-file-name.

8.10.38 REWRITE statement

Function

The REWRITE statement replaces a logical record on a disk storage file.

Format 1 for sequential file organization

```
REWRITE record-name [FROM identifier] [END-REWRITE]
```

Format 2 for relative and indexed file organization

```
REWRITE record-name [FROM identifier]
    [INVALID KEY imperative-statement-1]
    [NOT INVALID KEY imperative-statement-2]
    [END-REWRITE]
```

Syntax rules

1. record-name and identifier must not refer to the same storage area.
2. record-name must be associated with a file description (FD) entry in the Data Division of the program and may be qualified.
3. The INVALID KEY phrase must be specified for files with relative and indexed file organization unless an appropriate USE procedure has been declared. INVALID KEY may not be specified for relative files with sequential access.
4. The REWRITE statement is not permitted for line sequential files.

General rules

The following applies for **sequential, relative and indexed file organization**:

1. record-name identifies the record to be replaced.
2. The file associated with record-name must be a disk file and be open in the I-O mode at the time the REWRITE statement is executed.
3. Execution of a REWRITE statement with the FROM phrase is equivalent to execution of the following statements:

   ```
   MOVE identifier TO record-name
   REWRITE record-name.
   ```

   The content of the storage area described by record-name before the implicit MOVE statement is performed has no effect on the execution of the REWRITE statement.
   Thus, when the FROM phrase is used, data is transferred from identifier to record-name and then released to the appropriate file.
   The identifier may be used to reference any data outside the current file description entry. Transfer of data through the implicit MOVE statement takes place in accordance with the rules for a MOVE statement without the CORRESPONDING phrase. After the REWRITE statement is executed, the content of the record is still available in the area specified by identifier, although it is no longer available in the area specified by record-name.

4. The following rules apply to all files whose access mode is sequential:
   a. A REWRITE statement must be preceded by a successful READ statement as the last input/output statement for the associated file.
   b. Execution of the REWRITE statement causes the record accessed by the preceding READ statement to be replaced on that file.
   c. The contents of the data item specified by the RELATIVE KEY phrase (RELATIVE KEY for **relative files**; RECORD KEY for **indexed files**) or the RECORD KEY clause must not be modified between the READ and REWRITE statements.
5. A REWRITE statement must be preceded by a successful READ statement as the last input/output statement for the associated file.
6. Execution of the REWRITE statement causes the record accessed by the preceding READ statement to be replaced on that file.
7. The record rewritten by a successful REWRITE statement is no longer accessible in the record area; an exception to this rule is the use of the SAME RECORD AREA clause. In this case, the record is available to all other files specified in the SAME RECORD AREA clause as well as to the current file.
8. Execution of a REWRITE statement causes the contents of the data item that was specified in the FILE STATUS clause of the related file description entry to be updated (see also section "FILE STATUS clause").

   For **relative and indexed file organization**, the following also applies:
9. The length of the record to be replaced on the file may be modified.
10. In the case of files whose access mode is random or dynamic, the replaced record is the one accessed by the contents of the RELATIVE KEY data item associated with the file (for relative files) or by the contents of the data item specified by the RECORD KEY (for indexed files).
11. The number of character positions in the record indicated by record-name must not be greater than the largest number of character positions or less than the smallest number of character positions permitted by the associated RECORD IS VARYING clause. Otherwise, the REWRITE statement will be unsuccessful, the update operation will not take place, the content of the record area remains unchanged and the I-O status of the file associated with record-name is set to a value indicating a record length conflict (see chapter "General concepts").

    For **sequential file organization**, the following also applies:
12. The record addressed by the record name must be of the same length as the record to be replaced on the file.
13. If the number of character positions specified in the record referenced by record-name is not equal to the number of character positions in the record being replaced, execution of the REWRITE statement is unsuccessful, the updating operation does not take place, the content of the record area is unaffected and the I-O status of the file is set to a value indicating that the record limits have been exceeded.
14. Rewriting a record does not cause the record contents on the associated disk storage file to be changed until the next block of the file is read or that file is closed.

    For **relative file organization**, the following also applies:
15. If the file does not contain a record corresponding to the contents of the RELATIVE KEY item, the invalid key condition occurs (see "Invalid key condition"). In this case, execution of the REWRITE statement is unsuccessful, no updating is executed, the contents of the record area remain unchanged, and the I-O status is set to a value which indicates an invalid key condition.

16. The continuation of processing of the REWRITE statement depends on whether INVALID KEY or NOT INVALID KEY is specified (see "Invalid key condition").
17. When rewriting a record, the user should ensure that its first byte does not contain the value X’FF’, since this will cause logical deletion of the record.

    For **indexed file organization**, the following also applies:

18. The invalid key condition occurs in one of the following situations:
    a. in sequential access mode: if the primary key value of the record to be rewritten does not match the primary key value of the last record read from the file;
    b. in dynamic or random access mode: if the primary key value of the record to be rewritten does not match the primary key value of any record in the file;
    c. if the alternate record key value of the record to be rewritten is the same as the alternate record key value of a record which already exists in the file and duplicate keys are not permitted.
19. For a record with an alternate key, the REWRITE statement is executed as follows:
    a. If the value of a specific alternate key is not changed, the order in which records are retrieved remains unchanged when that key is the key of reference.
    b. If the value of a specific alternate key is changed, the order in which records are retrieved may be changed when that key is the key of reference. If duplicates are permitted, the record becomes the last of the set of records which have the same alternate key value as the record to be rewritten.

8.10.39 SEARCH statement

Function

The SEARCH statement is used to search a table for a table element that satisfies a specified condition, and to adjust the value of the associated index to indicate that table element, i.e. to set it to the corresponding occurrence number.

Format 1 is used to perform a serial search of a table. The search for identifier-1 begins at the current value of the index assigned to identifier-1.

Format 2 is used to perform a binary search of a table.

Format 1

```
SEARCH identifier-1 [VARYING {index-1 | identifier-2}]

    [AT END imperative-statement-1]

    {WHEN condition-1 {imperative-statement-2 | NEXT SENTENCE}}...

[END-SEARCH]
```

Syntax rules

1. identifier-1 specifies the table to be searched.
2. The data description entry of identifier-1 must include an OCCURS clause with an INDEXED BY phrase.
3. identifier-1 must not be subscripted or indexed or subjected to reference modification.
4. index-1 or identifier-2 specifies an item whose value is to be varied during execution of the SEARCH statement.
5. index-1 may be one of the indices for identifier-1, or an index for another table entry.
6. identifier-2 must therefore be described as an index data item (USAGE IS INDEX), or it must be a fixed-point numeric item described as an integer.
7. The AT END phrase specifies a statement which is to be executed if the search is unsuccessful.
8. Each condition (condition-1, condition-2,...) must be a valid condition (see section "Conditions").
9. condition-1... specify the conditions to be satisfied during the execution of the SEARCH statement.
10. imperative-statement-2... or NEXT SENTENCE specifies an action to be taken when the associated WHEN condition is satisfied: control passes to the imperative-statement or to the next sentence (that is, the statement following the SEARCH statement), depending on the option specified.
11. A serial search of a table begins at the table element pointed to by the index associated with identifier-1.
12. If, at the start of a SEARCH statement, the value of the index associated with identifier-1 is greater than the highest permissible occurrence number for identifier-1, the search will terminate immediately. If the AT END phrase is specified, imperative-statement-1 is executed. If this phrase is omitted, processing continues with the next statement.
13. If, at the start of a SEARCH statement, the value of the index associated with identifier-1 corresponds to a valid occurrence number for identifier-1, the serial search takes place as follows:
    a. The WHEN conditions are evaluated in the order in which they are written.

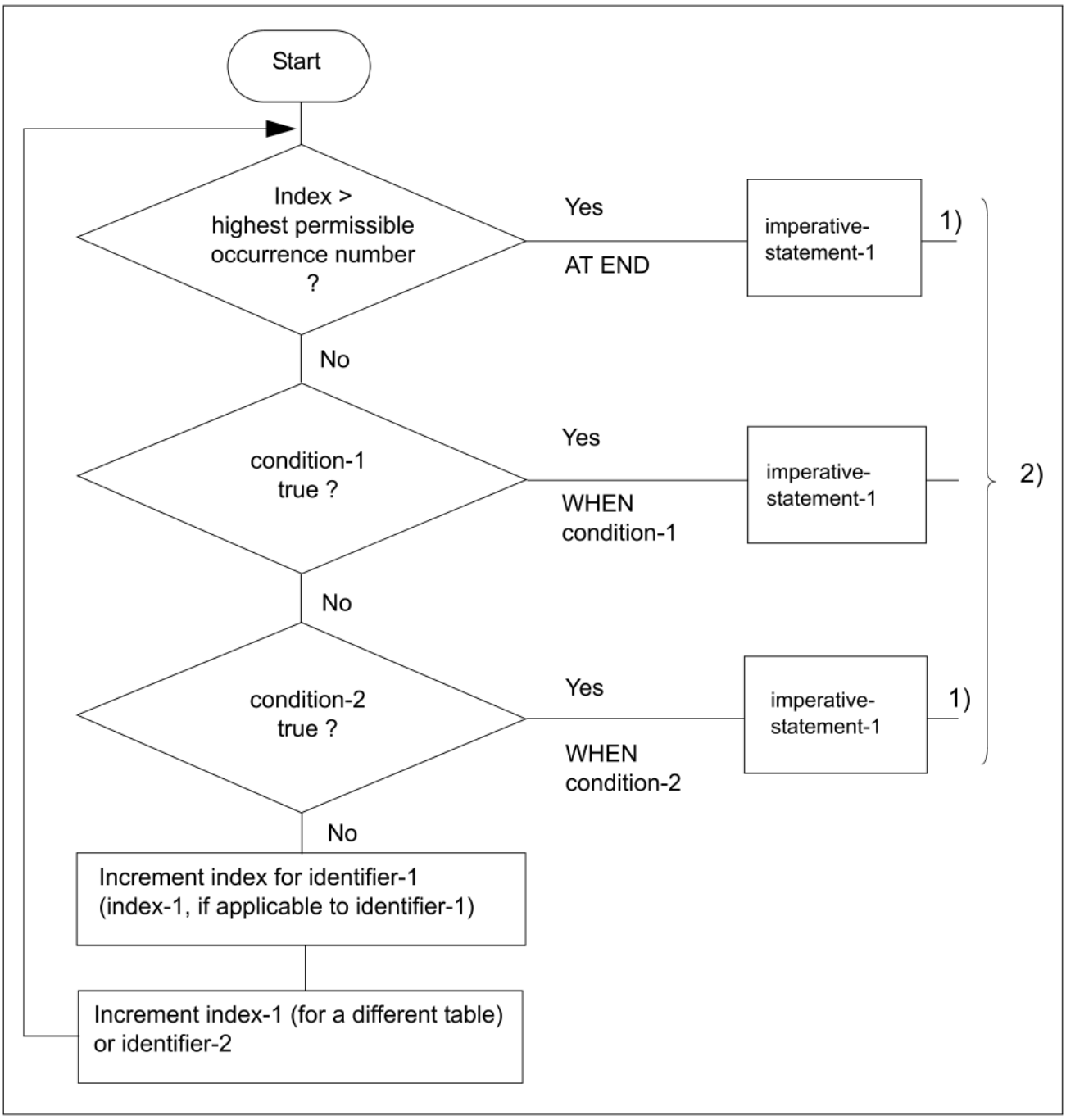
b. If none of the conditions is satisfied, the index-name for identifier-1 is incremented to refer to the next occurrence of a table element; and step a) is repeated, unless the new value of the index corresponds to an occurrence number outside the valid range, in which case step d) is performed.

c. If one of the WHEN conditions is satisfied, the search terminates immediately. The index points to the table element that satisfied the condition. The imperative statement associated with that condition is executed.

d. If the end of the table is reached without the WHEN condition being satisfied, the search terminates. If AT END is specified, imperative-statement-1 is executed. If this phrase is omitted, control passes to the next sentence.

14. When identifier-1 is a data item subordinate to a data item that contains an OCCURS clause, then multi-dimensional tables can be searched. In this case, an index-name must be associated with each dimension of the table through the INDEXED BY phrase of the OCCURS clause. Execution of a SEARCH statement modifies only the setting of the index associated with identifier-1 (and, if present, of index-1 or identifier-2). Therefore, in order to search a two- or three-dimensional table, a SEARCH statement must be executed for each possible value of the superordinate index.
15. Before the SEARCH statements are executed, the corresponding indices must be preset with the required values by means of SET statements or with PERFORM VARYING (see "Example 8-69").
16. If, in the AT END phrase and the WHEN conditions, none of the imperative-statements specified terminates with a GO TO statement, then control will pass to the next statement after the imperative statement is executed.
17. If the VARYING index-1 phrase is specified, the following takes place:
    - If index-1 is one of the indices for identifier-1, index-1 is used for the search. No other indices are incremented.
    - If index-1 is an index for another table entry, the first, or only, index associated with identifier-1 is used for the search. When the index associated with identifier-1 is incremented, index-1 is simultaneously incremented to represent the next element in its table.
18. If the VARYING identifier-2 phrase is specified, the following actions take place:
    - The first, or only, index associated with identifier-1 is used for the search.
    - When the index associated with identifier-1 is incremented, identifier-2 is simultaneously incremented.
    - If identifier-2 is a numeric item, it is incremented by 1.
    - If identifier-2 is an index data item, it is incremented by a value equal to that used to increment the index associated with identifier-1.

    If the VARYING phrase is not specified, the first, or only, index associated with identifier-1 (i.e. defined in the INDEXED BY phrase of the data description entry of identifier-1) will be used for the search.
19. If the END-SEARCH phrase is specified, the NEXT SENTENCE phrase must not be specified.
20. The scope of a SEARCH statement may be terminated by any of the following:
    - An END-SEARCH phrase at the same level of nesting.
    - A separator period.
    - An ELSE or END-IF phrase associated with a previous IF statement.



Format 1 SEARCH statement

1) These operations are included only when specified in the SEARCH statement.

2) Each of these control transfers is to the next sentence, unless the imperative-statement ends with a GO TO statement.

Example 8-67

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SEARCH1.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
```

```
    TERMINAL IS T.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  LETTER-TEST PIC X VALUE LOW-VALUE.
    88 LETTER-FOUND VALUE HIGH-VALUE.
01  INPUT-LETTER PICTURE A.
01  LETTER-WEIGHT-TABLE.
       03  LETTER-TABLE OCCURS 26 TIMES INDEXED BY PI
           VALUE FROM (1)                        "A01" "B03"
             "C03" "D02" "E01" "F04" "G02" "H04" "I01" "J08"
             "K05" "L01" "M03" "N01" "O01" "P03" "Q10" "R01"
             "S01" "T01" "U01" "V04" "W04" "X08" "Y04" "Z10".
           04  LETTER PICTURE A.
           04  VAL PICTURE 99.
PROCEDURE DIVISION.
MAIN SECTION.
LETTER-SEARCH.
    PERFORM UNTIL LETTER-FOUND
      ACCEPT INPUT-LETTER FROM T
      SET PI TO 1
      SEARCH LETTER-TABLE VARYING PI
         AT END DISPLAY "Letter is not alphabetic" UPON T
                EXIT TO TEST OF PERFORM
         WHEN LETTER (PI) = FUNCTION UPPER-CASE (INPUT-LETTER)
                SET LETTER-FOUND TO TRUE
      END-SEARCH
    END-PERFORM.
FOUND.
    DISPLAY LETTER (PI) " is assigned to " VAL (PI) UPON T.
    STOP RUN.
```

In this example, the table named LETTER-TABLE consists of 26 elements.

Each element contains a letter of the alphabet followed by a value associated with the letter.
The table is indexed by the index-names LI and PI.
The SEARCH statement searches the table for the element whose LETTER matches the current contents of the area called INPUT-LETTER. The associated index is PI.The search starts at the beginning of the table since PI points to the first table element.If the search is successful, the statement GO TO FOUND is executed. In this case, the index PI points to the element satisfying the condition. For example, if INPUT-LETTER contains B, the index points to the second table element.
If the search is unsuccessful, the statements following the AT END phase are executed.

Format 2

```
SEARCH ALL identifier-1 [AT END imperative-statement-1]

       WHEN condition {imperative-statement-2 | NEXT STATEMENT}

   [END-SEARCH]
```

Syntax rules

1. identifier-1 specifies the table to be searched.
2. The description of identifier-1 must contain an OCCURS clause with the INDEXED BY and ASCENDING /DESCENDING phrases.

3. identifier-1 must not be subscripted or indexed or subjected to reference modification.
4. The AT END phrase specifies the statement to be executed if condition-1 cannot be satisfied for any setting of the index within the valid range (see rule 10).
5. condition specifies the condition that must be satisfied during the execution of the SEARCH statement.
6. condition must be one of the following types of conditions (see also section "Conditions"):
    a. A relation condition incorporating EQUAL TO or the equal sign (=) as relational operator. Either the subject or the object (but not both) of the relation condition must consist solely of one of the data-names that appear in the ASCENDING/DESCENDING phrase of identifier-1. Each data-name must be indexed by the first index associated with identifier-1. It may be qualified, but not subjected to reference modification.
    b. A condition-name condition in which the VALUE clause describing the condition-name contains only a single literal. The conditional variable associated with the condition-name must be one of the data-names that appear in the ASCENDING/DESCENDING phrase of identifier-1.
    c. A combine condition formed from simple conditions of the types described above, with AND as the only connective.
7. Any data-name that appears in the ASCENDING/DESCENDING phrase of identifier-1 may be tested in condition. However, all data-names in the ASCENDING/DESCENDING phrase preceding the data-name to be tested, must also be tested in condition. No other tests can be made in condition.
8. imperative-statement-2 or NEXT SENTENCE specifies an action to be taken when condition is satisfied. Control passes to imperative-statement-2 or to the statement following the SEARCH statement.
9. The **first** index associated with identifier-1 is used for the search. This index does not have to be initialized with a SET statement since its initial value is ignored for the search.
10. The SEARCH ALL statement is executed as follows, whereby the table must be arranged in ascending or descending order of the key fields listed in the ASCENDING/DESCENDING phrase:
    a. During the search the value of the index associated with identifier-1 is varied.
    b. This setting is never less than the value corresponding to the first table element, and never greater than the value corresponding to the last table element.
    c. If condition cannot be satisfied for any setting of the index within this permitted range, control is passed to imperative-statement-1 if the AT END phrase is specified, or to the next statement if the AT END phrase is omitted.
11. If condition can be satisfied, the index points to the table element satisfying the condition. Control then passes to imperative-statement-2 or to the next sentence.
12. When identifier-1 is a data item subordinate to a data item that contains an OCCURS clause, then two- or three-dimensional tables can be searched. In this case, an index-name must be associated with each dimension of the table through the INDEXED BY phrase of the OCCURS clause. Execution of the SEARCH ALL statement modifies only the setting of the index-name associated with identifier-1. Therefore, in order to search a two- or three-dimensional table, a SEARCH ALL statement must be executed for each possible value of the superordinate index.
13. If, in the AT END phrase and the WHEN conditions, none of the statements specified terminates with a GO TO statement, then control will pass to the next statement after the imperative-statement is executed.
14. If the END-SEARCH phrase is specified, the NEXT SENTENCE phrase must not be specified.
15. The scope of a SEARCH statement may be terminated by any of the following:
    a. An END-SEARCH phrase at the same level of nesting.
    b. A period at the end of the SEARCH statement.
    c. An ELSE or END-IF phrase associated with a previous IF statement.

Example 8-68

(WHEN conditions)

Data Division entries:

```
       ...
77  A-VALUE PICTURE 9.
       ...
    02  TABLE-ITEM OCCURS 5 TIMES ASCENDING KEY IS A B C;
                INDEXED BY I.
        03  A  PICTURE 99.
        03  B  PICTURE 9.
            88 UNDER-30 VALUE 1.
            88 OVER-30 VALUE 2.
        03  C  PICTURE 9.
       ...
```

Valid WHEN phrases (in Procedure Division)

```
WHEN A(I) = 10
WHEN A(I) = 20 AND UNDER-30(I)
WHEN A(I) = 15 AND OVER-30(I) AND C(I) = A-VALUE
```

Example 8-69

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SRCHALL.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
    TERMINAL IS T
    SYSIPT IS INFILE.
DATA DIVISION.
WORKING-STORAGE SECTION.
77  IPD PIC 9(3).
77  INPUT-LN PIC 9(6).
01  EMPLOYEE-TABLE.
    02  PERSON OCCURS 100 TIMES INDEXED BY PI,
                    ASCENDING KEY IS DEPARTMENT LIFE-NUMBER.
       03  DEPARTMENT  PIC 9(3).
       03  LIFE-NUMBER PIC 9(6).
       03  NAME        PIC X(20).
PROCEDURE DIVISION.
MAIN SECTION.
P1.
    PERFORM VARYING PI FROM 1 BY 1 UNTIL PI > 100
      ACCEPT PERSON (PI) FROM INFILE
    END PERFORM
    SEARCH ALL PERSON
    AT END
```

```
      DISPLAY "Person is missing" UPON T
    WHEN DEPARTMENT (PI) = IPD AND LIFE-NUMBER (PI) = INPUT-LN
      DISPLAY DEPARTMENT LIFE-NUMBER NAME UPON T
    END-SEARCH
    STOP RUN.
```

In this example, the table named PERSON consists of 2000 elements.

Each element in the table consists of a 3-byte numeric item called DEPARTMENT, a 6-byte numeric item called LIFE-NUMBER and a 20-byte alphanumeric item called NAME.

The table is arranged in ascending order by DEPARTMENT and, within DEPARTMENT, in ascending order by LIFE-NUMBER.

The first portion of the table might have the following contents:

DEPARTMENT	LIFE-NUMBER	NAME
101	123456	ADAM, D.
101	234561	LANGEWIESCHE, W.
101	123456	ADAM, D.
101	523618	EBERLE, F.
183	200305	DAUTZENBERG, K.
183	328512	REINHARDT, M.
183	433333	GRUEN, L.
183	987245	RICHTER, L.
557	328835	SCHMIDT, S.
557	775247	ALBRECHT, N.

The SEARCH statement searches the table for an element whose DEPARTMENT matches the current contents of the area called IPD, and whose LIFE-NUMBER matches the current contents of the area called INPUT-LN. If the search is successful, the DISPLAY statement is executed. The index-name, PI, points to the element satisfying the condition.

For example, if IPD contains 183 and INPUT-LN contains 328512, the index-name points to the fifth element of the table.

If the search is unsuccessful, an appropriate message is issued.

8.10.40 SET statement

Function

The SET statement offers the following options with its various formats:

Format 1	sets an integer data item, index or index data item to a specified value.
Format 2	increments or decrements the value of an index-name to represent a new occurrence number.
Format 3	changes the status of external switches
Format 4	sets the value of conditional variables
Format 5	assigns object references
Format 6	assigns data pointer
Format 7	works with data pointers
Format 8	assigns program pointers
Format 9	reset the last exception status

Format 1

```
SET {index-1 | identifier-1}... TO {index-2 | identifier-2 | integer-1}
```

Syntax rules

In the following notes, all references to index-1 and identifier-1 apply equally to all recursions thereof.

1. Each index must be specified in an INDEXED BY phrase of an OCCURS clause.
2. Each identifier must reference either an index data item or a fixed-point numeric elementary item described as an integer.
3. The value of integer-1 or identifier-2 must be a valid occurrence number in the corresponding table. The compiler does, however, permit other integer values (e.g. 0 or negative numbers) within the permissible value range for index-1 (see section "Indexing").
4. Indices or identifiers preceding the TO specify the item whose value is to be set.
5. index-2, identifier-2 and integer-1 (which follow the TO) specify the value to which the receiving item (e.g. index-1) is to be set.
6. When the SET statement is executed, one of the following actions occurs:
   a. index-1 is set to a value causing it to refer to the table element that corresponds in occurrence number to the table element referenced by index-2, identifier-2, or integer-1.
   If identifier-2 references an index data item, or if index-2 is related to the same table as index-1, no conversion takes place.
   b. If identifier-1 is an index data item, it is set equal to either the contents of index-2 or identifier-2, where identifier-2 is also an index data item. No conversion takes place. integer-1 cannot be used in this case.
   c. If identifier-1 is not an index data item, it is set to an occurrence number that corresponds to the value of index-2. Neither identifier-2 nor integer-1 can be used in this case.

This process is repeated for each recurrence of index-1 or identifier-1. Each time, the value of index-2 or identifier-2 is used as it was at the beginning of the execution of the statement. Any subscripting or indexing associated with identifier-1 is evaluated before the value of the respective data item is changed.

7. The table below indicates the validity of various operand combinations in the SET statement. The letters following the slashes refer to the rules listed above under point 7; for example, valid/c indicates that a combination of items is valid according to rule c) above.

Sending item	**Receiving item**		
	Integer data item PIC9(n)	Index-name INDEXED BY	Index-name USAGE IS INDEX
Integer literal	no/c	valid/a	no/b
Integer data item	no/c	valid/a	no/b
Index-name	valid/c	valid/a	valid/b*
Index data item	no/c	valid/a*	valid/b*

Table 31: Valid uses of the SET statement

* = no conversion takes place.

8. If index-2 is used, its value prior to execution of the SET statement must correspond to an occurrence number of an element in the associated table

Example 8-70

Data Division entries:

```
    02 TABLE-A PICTURE XXX OCCURS 50,
               INDEXED BY IN-A1, IN-A2, IN-A3.
    02 TABLE-B PICTURE XX OCCURS 55,
               INDEXED BY IN-B.
77 ID-1 USAGE IS INDEX.
77 D-1 PICTURE IS S999, USAGE IS COMPUTATIONAL-3.
```

The Procedure Division statements are shown in the tabular presentation below:

Statement	**Operands used**	**Action taken** [1]
`SET IN-A2 TO IN-A1`	Index-name set to index-name	Simple move (displacement to displacement).
`SET IN-A1 TO D-1`	Index-name set to numeric data item	Multiply (numeric item minus 1) by item length to get displacement.
`SET IN-A1 TO`		Simple move (displacement to displacement).

`ID-1`	Index-name set to index data item	
`SET IN-A1 TO 4`	Index-name set to literal	Multiply (literal minus 1) [2] by item length to get displacement.
`SET ID-1 TO IN-A1`	Index data item set to index-name	Simple move (displacement to displacement)
`SET D-1 TO IN-A1`	Numeric data item set to index-name	Divide displacement by item length and add 1, to get occurrence number.
`SET IN-B TO IN-A1`	Index-name set to index-name (different tables)	Divide displacement (i.e., contents of IN-A1) by item length for TABLE-A and add 1, to get occurrence number. Then, multiply (occurrence number minus 1) by item length for TABLE-B, to get displacement.

[1] See section "Indexing" for the relationships between occurrence number and displacement.

[2] Calculated at compilation time; simple move during program run.

Example 8-71

```
02  TABLE-A OCCURS 50 TIMES
              INDEXED BY IND-1,IND-2,PIC 999.
 .
 .
 .
SET IND-1 TO 5.
SET IND-2 TO 7.
SET IND-1,TABLE-A (IND-1) TO IND-2.
```

The third SET statement is equivalent to the following two statements, which are executed in the order in which they appear:

Statement	Action
`SET IND-1 TO IND-2`	IND-1 is set to the occurrence number 7, the current value of IND-2.
`SET TABLE-A (IND-1) TO IND-2`	Since the first SET statement sets IND-1 to 7, TABLE-A (IND-1) = TABLE-A (7). Thus, this statement sets TABLE-A (7) to 7.

Format 2

```
SET {index-3}... {UP BY | DOWN BY} {identifier-3 | integer-2}
```

Syntax rules

1. Each index must be specified in an OCCURS clause with INDEXED BY phrase.
2. index-3... specifies the storage index whose value is to be changed.
3. identifier-3 must be a fixed-point numeric elementary item described as an integer.
4. integer-2 must be a non-zero integer and may contain a positive sign.
5. The UP BY or DOWN BY phrase specifies that the contents of the index specified is to be either incremented (UP BY) or decremented (DOWN BY) by a value which corresponds to the occurrence number representing the value of identifier-3 or integer-2.

Example 8-72

```
02  TABLE-A PICTURE X(20) OCCURS 5 INDEXED BY IND-1.
    .
    .
    .
SET IND-1 TO 4.
SET IND-1 UP BY 2.
SET IND-1 DOWN BY 3.
```

The first SET statement sets index IND-1 to occurrence number 4; the second, to 6 (i.e. 4 + 2); and the third, to 3 (i. e. 6 - 3).

Format 3

```
SET { {mnemonic-name-1}... TO {ON | OFF} }...
```

Syntax rule

1. Each mnemonic-name must be associated with an external switch whose status can be altered (see section "Condition-name condition").

Format 4

```
SET { {condition-name-1}... TO {TRUE | FALSE } }...
```

Syntax rules

1. condition-name-1 must be associated with a conditional variable (see section "Condition-name condition").
2. If FALSE is specified in the SET statement, it must also be specified in the VALUE clause of the condition-name.

General rules

1. The literal specified in the VALUE clause and associated with condition-name-1 is entered in the conditional variable according to the rules governing the VALUE clause (see section "VALUE clause").
   However, if a group is subordinate to the conditional variable, its length is determined in accordance with the rules of the OCCURS clause (see section "OCCURS clause").
   If more than one literal is specified in the VALUE clause, the conditional variable is set to the value of the first literal appearing in the VALUE clause.
2. If two or more condition-names are specified, they are treated as though a SET statement had been written for each individual condition-name.
3. For "SET...TO FALSE", the literal specified with FALSE in the VALUE clause which is associated with `condition-name` is entered in the conditional variable according to the rules governing the VALUE clause (see "SET condition-name TO TRUE").

Example 8-73

```
IDENTIFICATION DIVISION.
PROGRAM-ID. DAYSET.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
    TERMINAL IS T.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  EXAMPLE1.
    02  WORKDAY  PICTURE X.
        88 MONDAY  VALUE "1".
        88 FRIDAY VALUE "5".
01 EXAMPLE2.
   02   WEEKDAY PIC X.
        88 WORKDAYS VALUE "1" "2" "3" "4" "5".
PROCEDURE DIVISION.
P1 SECTION.
SETTING.
    SET FRIDAY TO TRUE.
    DISPLAY "Workday =" WORKDAY UPON T.
    SET WORKDAYS TO TRUE.
    DISPLAY "Weekday =" WEEKDAY UPON T.
FINISH-PAR.
    STOP RUN.
```

Once the first SET statement has been performed, the data item WORKDAY (conditional variable) will contain the literal assigned in the VALUE clause to the condition-name FRIDAY: "5".
Once the second SET statement has been performed, the data item WEEKDAY will contain the literal "1".

Format 5

```
SET {identifier-5}... TO identifier-6
```

Syntax rules

1. identifier-5 must be an object reference that is permitted as a receiving operand.
2. identifier-6 must be a class-name or object reference; the predefined object reference SUPER may not be specified.
3. If the data item referenced by identifier-5 is a universal object reference, all data of the class “object“ are permitted for identifier-6.
4. If the data item referenced by identifier-5 is described with an interface-name that identifies the interface int-1, the data item referenced by identifier-6 must be one of the following:
   a. an object reference that specifies an interface-name for an interface that conforms with int-1;
   b. an object reference described with a class-name, subject to the following rules:
      - if described with a FACTORY phrase, the interface of the factory object of the specified class must conform with int-1;
      - if described without a FACTORY phrase, the interface of the objects of the specified class must conform with int-1;
   c. an object reference described with an ACTIVE-CLASS phrase, subject to the following rules:
      - if described with a FACTORY phrase, the interface of the factory object of the specified class containing the data item referenced by identifier-6 must conform with int-1;
      - if described without a FACTORY phrase, the interface of the objects of the specified class containing the data item referenced by identifier-6 must conform with int-1;
   d. a class, where the interface of its factory object conforms with int-1;
   e. the predefined object reference SELF, where the interface of the source unit containing the SET statement conforms with int-1;
   f. the predefined object reference NULL.
5. If the data item referenced by identifier-5 is described with a class-name, the data item referenced by identifier-6 must be one of the following:
   a. an object reference described with a class-name, subject to the following rules:
      - If the data item referenced by identifier-5 is described with an ONLY phrase, the data ntifier-6 must also be described with the ONLY phrase. Furthermore, the class-name specified in the description of the data item referenced by identifier-6 must be the same as the class-name specified in the description of the data item referenced by identifier-5.
      - If the data item referenced by identifier-5 is described without an ONLY phrase, the class-name specified in the description of the data item referenced by identifier-6 must reference the same class or subclass specified in the description of the data item referenced by identifier-5.
      - The presence or absence of the FACTORY phrase must be the same as in the description of the data item referenced by identifier-5.
   b. an object reference described with an ACTIVE-CLASS phrase, subject to the following rules:
      - The data item referenced by identifier-5 must not be described with the ONLY phrase.
      - The class containing the data item referenced by identifier-6 must be the same class or a subclass of the class specified in the description of the data item referenced by identifier-5.
      - The presence or absence of the FACTORY phrase must be the same as in the description of the data item referenced by identifier-5.
   c. the predefined object reference SELF, subject to the following rules:

- The data item referenced by identifier-5 must not be described with the ONLY phrase.
- The class of an object containing the SET statement must be the same class or a subclass of the class specified in the description of the data item referenced by identifier-5.
- If the data item referenced by identifier-5 is described without a FACTORY phrase, the source unit containing the SET statement must be an object instance.
- If the data item referenced by identifier-5 is described with a FACTORY phrase, the source unit containing the SET statement must be a factory object.

d. a class-name, provided the following applies:

- The description of the data item referenced by identifier-5 contains a FACTORY phrase.
- If the data item to which identifier-5 refers is described with the ONLY phrase, the class name must be the same. Otherwise, it can be a subclass of the class by means of which the data item to which identifier-5 refers is described.

e. the predefined object reference NULL.

6. If the description of the data item referenced by identifier-5 is described with an ACTIVE-CLASS phrase, the data item referenced by identifier-6 must be one of the following:
   a. an object reference described with the ACTIVE-CLASS phrase, where the presence or absence of the FACTORY phrase is the same as in the data item referenced by identifier-5;
   b. the predefined object SELF, subject to the following rules:
      - if the data item referenced by identifier-5 is described without a FACTORY phrase, the object containing the SET statement must be an object instance;
      - if the data item referenced by identifier-5 is described with a FACTORY phrase, the object containing the SET statement must be a factory object.
   c. the predefined object reference NULL.

General rules

1. If identifier-6 is an object reference, a reference to the data item identified by identifier-5 is placed into each data item referenced by identifier-5 in the order specified.
2. If identifier-6 is a class-name, a reference to the factory object of the class identified by identifier-5 is placed into each data item referenced by identifier-5 in the order specified.

Conformance of SET object references

The table below provides a summary of the combinations of object references in the sending and receiving items.

Combination

always permitted	o.k.
permitted conditionally	condition
forbidden	----

Notation of conditions

A > B	interface A is conformant to interface B:
A >> B	class A is subclass of B:

A == B class A is the same class as class B:

Abbreviation

i5	Interface of receiving item
i6	Interface of sending item
c5	class of receiving item
c6	class of sending item
c	class corresponding to name
C	class containing SET statement[1]
FI(X)	interface of factory object of class X
OI(X)	interface of object object of class X

[1] This refers to the method containing the SET statement is defined and **not** a class which inherits this method.

		sending USAGE Object Reference							
receiving USAGE Object Reference		universal	interface i6	FACTORY classname c6	FACTORY classname ONLY c6	classname c6	classname ONLY c6	FACTORY ACTIVE-CLASS C	ACTIVE-CLASS C
universal		o.k.	o.k.	o.k.	o.k.	o.k.	o.k.	o.k.	o.k.
interface	i5	----	i6 > i5	FI(c6)>i5	FI(c6)>i5	OI(c6)>i5	OI(c6)>i5	FI(C)>i5	OI(C)>i5
FACTORY classname	c5	----	----	c6>>==c5	c6>>==c5	----	----	C>>==c5	----
FACTORY classname ONLY	c5	----	----	----	c6==c5	----	----	----	----
classname	c5	----	----	----	----	c6>>==c5	c6>>==c5	----	C>>==c5
classname ONLY	c5	----	----	----	----	----	c6==c5	----	----
FACTORY ACTIVE-CLASS	C	----	----	----	----	----	----	o.k.	----
ACTIVE-CLASS	C	----	----	----	----	----	----	----	o.k.

Table 32: combinations of object references in the sending and receiving items

		sending			
		classname	**SELF Object Reference**		**NULL**
receiving USAGE Object Reference		c	in factory-method C	in object-method C	
universal		o.k.	o.k.	o.k.	o.k.
interface	i5	FI(c)>i5	FI(C)>i5	OI(C)>i5	o.k.
FACTORY classname	c5	c>>==c5	C>>==c5	----	o.k.
FACTORY classname ONLY	c5	c==c5	----	----	o.k.
classname	c5	----	----	C>>==c5	o.k.
classname ONLY	c5	----	----	----	o.k.
FACTORY ACTIVE-CLASS	C	----	o.k.	----	o.k.
ACTIVE-CLASS	C	----	----	o.k.	o.k.

Table 33: combinations of object references in the sending and receiving items

Format 6

```
SET {ADDRESS OF data-name-1 | identifier-7 ...} TO identifier-8
```

Syntax rules

1. identifier-7 and identifier-8 must reference a data item of the category “data pointer”.
2. data-name-1 must be a data item specified with the BASED clause.
   However, see also the compiler option PERMIT-STANDARD-DEVIATION.
3. data-name-1 must not be defined with the ANY LENGTH clause.
4. If identifier-7 is a type-specific pointer, identifier-8 must either be the predefined address NULL or a type-specific pointer to the same type. If data-name-1 is strongly typed, identifier-8 must be a type-specific pointer to the same type.

   If identifier-8 is a type-specific pointer, either identifier-7 must be a type-specific pointer to the same type or data-name-1 must be based on the type to which identifier-8 is linked.

General rules

1. If identifier-7 is specified then the address specified by identifier-8 is stored in the data item referenced by identifier-7.
2. If data-name-1 is specified then the address specified by identifier-8 is assigned to the data item referenced by data-name-1.

Format 7

```
SET identifier-9 ... {UP | DOWN} BY {identifier-10 | integer-3}
```

Syntax rules

1. identifier-10 must be a numeric fixed-point item described as an integer.
2. integer-3 must be an integer or 0 and may be signed.
3. identifier-9 must belong to the class "data pointer”.
4. identifier-9 may not be a type-specific pointer for which STRONG applies in the associated type declaration.

General rule

1. The address present in the pointer item is incremented by the number of bytes defined in identifier-10 or integer-3 if UP is specified and decremented by the same value if DOWN is specified.

Format 8

```
SET {identifier-11}... TO identifier-12
```

Syntax rule of format 8

1. identifier-11 must be a data item of the category “program pointer”. identifier-12 must be of the catefory “program pointer”.

General rule of format 8

1. The address identified with identifier-12 is stored in the specified order in each identifier-11 data item.

Format 9

```
SET LAST EXCEPTION TO OFF
```

General rule of format 9

1. The last exception condition is reset, i.e. it indicates that no exception condition was triggered.

8.10.41 SORT statement

Function

Format 1 Sorting records.
The SORT statement is used to sort records either created in an input procedure or contained in a file according to a data items (set of specified keys).
The sorted records are released to an output procedure or entered in a file.

Format 2 Sorting tables.
The SORT statement causes table elements to be arranged according to a user-defined collating sequence.

Format 1 Sorting records

```
SORT sort-file-name

    {ON {DESCENDING | ASCENDING} {KEY | KEY-YY } {data-name-1}... }...

    [WITH DUPLICATES IN ORDER]

    [COLLATING SEQUENCE IS alphabet-name]

    {INPUT PROCEDURE IS paragraph-name-1 [{THRU | THROUGH} paragraph-name-2] | USING
    {file-name-1}...}

    {OUTPUT PROCEDURE IS paragraph-name-3 [{THRU | THROUGH} paragraph-name-4] | GIVING
    {file-name-2}...}
```

Syntax rules

1. The SORT statement may be specified anywhere in the Procedure Division except
   - in DECLARATIVES and
   - in input/output procedures which belong to a SORT statement.
2. sort-file-name must be described in a sort-file description (SD) entry in the Data Division.
3. sort-file-name must correspond to the sort-file-name defined in the SELECT clause (format 2).
4. data-name-1... are key data-names. A key is that part of a record which is used as a basis for sorting. Keys must be defined in a record description belonging to an SD entry. They are subject to the following rules:
   a. The data items must not be of variable length.
   b. Key data-names may be qualified (see section "Qualification").
   c. When two or more record descriptions are supplied, the keys need only be described in one of these descriptions. The byte positions addressed by the key in the record description are also regarded as a key in all other data descriptions of the file.
   d. A key must not be defined with an OCCURS clause and must not be subordinate to a data item defined with an OCCURS clause.
   e. If the sort file referenced by file-name-1 contains variable length records, all the data items identified by key data-names must be contained within the first n character positions of the record, where n equals the minimum record size specified for the sort file.
   f. A maximum of 64 keys may be specified for any file.

g. A key must begin within the first 4096 bytes of the record.

h. Keys for sorting purposes are always listed from left to right in order of significance, regardless of whether they are ascending or descending. Hence, the first specification of data-name-1 is the principal key and the second specification of data-name-1 the subsidiary key.

i. The maximum length of a key which can be processed by SORT depends on the format of the key. The maximum length is 16 bytes for the packed decimal format (PD), the length of the record for a character-string, and 256 bytes for all other formats.

j. The keys following the KEY-YY specification must be defined either with PIC 99 USAGE DISPLAY or USAGE PACKED-DECIMAL (see chapter "Report Writer").

5. file-name-1..., file-name-2... must be defined in a file description (FD) entry in the DATA DIVISION.
6. The record length of records from input files passed to the SORT operation are governed by the following rules: If the sort file has a fixed length record format, program execution is terminated if the record is too long. If the record is too short, the missing spaces will be filled with blanks. If the sort file has a variable length record format, the input records are accepted without any alteration of their length. This length must be within the range specified for the sort file in the RECORD clause (see section "RECORD clause").
7. At least one ASCENDING/DESCENDING phrase must be specified in a SORT statement.
8. No pair of file-names in the same SORT statement may be specified in the SAME SORT AREA or SAME SORT-MERGE AREA clause.
9. If file-name-2 references an indexed file, the first specification of data-name-1 must be associated with an ASCENDING phrase and the data item referenced by that data-name-1 must occupy the same character positions in its record as the data item associated with the prime record key for that file.

General rules

1. The collating sequence is set by means of the ASCENDING/DESCENDING option:
   a. When ASCENDING is specified, the sorted sequence will be from the lowest to the highest value of the key, i.e. in ascending order.
   b. When DESCENDING is specified, the sorted sequence will be from the highest to the lowest value of the key, i.e. in descending order.

   The collating sequence is governed by the same rules as apply to the comparison of operands in "Relation conditions".
2. If DUPLICATES is specified and several records have the same contents in all of their sort items, then the order of return of these records is as follows:
   a. If no input procedure is specified, the records are returned in the order in which the associated files were specified in the SORT statement. If records with identical sort item contents exist in one and the same file, the records are returned in the order in which they were entered.
   b. If an input procedure is specified, the records are returned in the order in which they left the input procedure.
3. If DUPLICATES is not specified and several records have the same contents in all of their items, then the order of return of these records is undefined.
4. When the program is executed, the collating sequence for the comparison of alphanumeric sort items is set as follows:
   a. If COLLATING SEQUENCE has been specified in the SORT statement, this entry is used as a sort criterion.
   b. If COLLATING SEQUENCE was not specified in the SORT statement, the program-specific collating sequence will be used (see section "OBJECT-COMPUTER paragraph").

The national (native) collating sequence is used for comparisons of national collating sequences.

5. INPUT PROCEDURE indicates that the Procedure Division contains an input procedure to process records prior to sorting. If INPUT PROCEDURE is specified, control passes to it when the input section of the SORT program is ready to accept the first record. During RELEASE statement processing the input procedure releases records to the sort-file (see section "RELEASE statement"). The compiler inserts a return mechanism at the end of the last section in the input procedure, i.e. once the last statement in the input procedure has been processed the input procedure is terminated and the released records will be sorted in the sort-file. The following rules apply to the input procedure, which is a self-contained section within the Procedure Division:
   a. It must consist of one or more sections that are written consecutively.
   b. It must contain at least one RELEASE statement so that records can be released to the sort-file (see section "RELEASE statement").
   c. It must not lead to the execution of a MERGE, RETURN, or SORT statement.
   d. It may include any procedures needed to select, modify or copy records.
   e. It is permitted to leave the input procedure if the programmer makes sure that a transfer from the input procedure is followed by a return to it, in order to effect a proper exit from this procedure (i.e. processing its last statement).
   f. It is permitted to branch from points outside an input procedure to procedure-names within that procedure if such a branch does not involve a RELEASE statement or the end of the input procedure.
6. An input procedure, when specified, is processed before the records in the sort-file are sorted.
7. If the USING phrase is specified, all the records in the input files (file-name-1...) are transferred to the sort file referenced by sort-file-name. The input files must not be in the open mode when execution of the SORT statement begins. The execution of the SORT statement for each of the named files consists of the following phases:
   a. The processing of the file is initiated. The initiation is performed as if an OPEN statement with the INPUT phrase had been executed.
   b. The logical records are obtained and released to the sort operation. Each record is obtained as if a READ statement with the NEXT RECORD and the AT END phrases had been executed. For a relative file, the content of the relative key data item is undefined after the execution of the SORT statement if file-name-1 is not additionally referenced in the GIVING phrase. A relative file must be defined in the FILE CONTROL paragraph with ACCESS MODE IS SEQUENTIAL.
   c. The processing of the file is terminated. The termination is performed as if a CLOSE statement without optional phrases had been executed. This termination is performed before the sort operation begins.

   These implicit functions are performed such that any associated USE AFTER EXCEPTION/ERROR procedures are executed; however, the execution of such a USE procedure must not cause the execution of any statement manipulating the input files or their record area declarations.
8. OUTPUT PROCEDURE means that the Procedure Division contains an output procedure in which records are processed after sorting. If OUTPUT PROCEDURE is specified, control is passed to the output procedure after the sort-file has been processed by the SORT command. During RETURN statement processing the output procedure accepts the records from the sort-file. The compiler inserts a return mechanism at the end of the last section in the output procedure, i.e. once the last statement in the output procedure has been executed the procedure is terminated and control passes to the statement that follows the SORT statement.
   The following rules apply to the output procedure, which is a self-contained section within the Procedure Division:
   a. It must consist of one or more sections that are written consecutively and that do not form part of an input procedure.

b. It must contain at least one RETURN statement to make the sorted records available for processing (see section "RETURN statement").

c. It must not lead to the execution of a MERGE, RELEASE, or SORT statement.

d. It may include any procedures needed to select, modify, or copy records before they are transferred.

e. It is permitted to leave the output procedure if the programmer makes sure that a transfer from the output procedure is followed by a return to it, in order to effect a proper exit from this procedure (i.e. processing its last statement).

f. It is permitted to branch from points outside an output procedure to procedurenames within that procedure if such a branch does not involve a RETURN statement or the end of the output procedure.

9. When the INPUT PROCEDURE or OUTPUT PROCEDURE phrase is used, a branch is performed in the program as if it were a format-1 PERFORM statement. This means that all sections that form the procedure are run once and procedure execution terminates once the last statement has been processed. Thus, either procedure (or both) may be terminated by an EXIT statement.

10. If GIVING file-name-2... is specified, all the sorted records are written on the output file (file-name-2). The output file must not be in the open mode when execution of the SORT statement begins. The SORT statement is executed for each of the referenced files in the following way:

    a. The processing of the file is initiated. The initiation is performed as if an OPEN statement with the OUTPUT phrase had been executed. This initiation is performed after the execution of any input procedure.

    b. The sorted logical records are returned and written onto the output file as if a WRITE statement without any optional phrases had been executed. The length of these records must be within the range defined for the output file (see section "RECORD clause").
    For a relative file, the relative key data item for the first record returned contains the value "1"; for the second record returned, the value "2", etc. After execution of the SORT statement, the content of the relative data item indicates the last record returned to the file. The file must be defined in the FILE-CONTROL paragraph with ACCESS MODE IS SEQUENTIAL.

    c. The processing of the file is terminated. The termination is performed as if a CLOSE statement without optional phrases had been executed.

    These implicit functions are performed such that any associated USE AFTER EXCEPTION/ERROR procedures are executed; however, the execution of such a USE procedure must not cause the execution of any statement manipulating the input files or their record area declarations. On the first attempt to write beyond the externally defined boundaries of the file, any USE AFTER STANDARD EXCEPTION/ERROR procedure specified for the file is executed; if control is returned from this USE procedure or if no such USE procedure was specified, the processing of the file is terminated, as described in c) above.

11. Since the SORT statement is not directed at individual records, it does not conform to the standard input/output statements (READ, WRITE, etc.). The READ statement, when executing, reads a single record; likewise, the WRITE statement writes an individual record. The SORT statement, on the other hand, does not treat an individual record but an entire file. Thus, this entire file must be placed at the disposal of SORT, either via the USING phrase or by repeated use of the RELEASE statement within an input procedure, before SORT can function. The SORT routine alters the sequence of the records within the file, and hence the first record returned by the SORT routine is, as a rule, not the first record released to the routine. SORT cannot provide any output before it has received the whole of the input.

Examples of file SORT

Example 8-74

Sort processing with one output file

```
IDENTIFICATION DIVISION.
PROGRAM-ID.
SORT1.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT MASTER-FILE ASSIGN TO "MASTER-FILE".
    SELECT OUTPUT-FILE ASSIGN TO "OUTPUT-FILE".
    SELECT SORT-FILE ASSIGN TO "SORTWK".
DATA DIVISION.
FILE SECTION.
FD  MASTER-FILE LABEL RECORD STANDARD.
01  MASTER-RECORD.
    02  E0            PIC X.
    02  E1            PIC 9(4).
    02  E2            PIC 9(4).
    02  E3            PIC 9(4).
FD  OUTPUT-FILE LABEL RECORD STANDARD.
01  OUTPUT-RECORD.
    02  A0            PIC X.
    02  A1            PIC 9(4).
    02  A2            PIC 9(4).
    02  A3            PIC 9(4).
SD  SORT-FILE LABEL RECORD STANDARD.
01  SORT-RECORD.
    02  S0             PIC X.
    02  S1             PIC 9(4).
    02  S2             PIC 9(4).
    02  S3             PIC 9(4).
PROCEDURE DIVISION.
P1 SECTION.
SORTING.
    SORT SORT-FILE ASCENDING S1 S2 S3                (1)
                                                      |
         USING MASTER-FILE GIVING OUTPUT-FILE.       (1)
    STOP RUN.
```

(1) The sort operation takes place in the following stages:

1. The records are released from the input file MASTER-FILE to the SORT-FILE.
2. The records are sorted in the sort-file according to ascending S1, or (in those records with identical S1) according to ascending S2, or (in records with identical S1 and S2) according to ascending S3.
3. The records are released from the sort-file to the OUTPUT-FILE.

Example 8-75

Sort processing with two output files

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SORT2.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT MASTER-FILE ASSIGN TO "MASTER-FILE".
```

```
    SELECT OUTPUT-FILE-1 ASSIGN TO "OUTPUT-FILE-1".
    SELECT OUTPUT-FILE-2 ASSIGN TO "OUTPUT-FILE-2".
    SELECT SORT-FILE ASSIGN TO "SORTWK".
DATA DIVISION.
FILE SECTION.
FD  MASTER-FILE LABEL RECORD STANDARD.
01  INPUT-RECORD.
    02  E0          PIC X.
    02  E1          PIC 9(4).
    02  E2          PIC 9(4).
    02  E3          PIC 9(4).
FD  OUTPUT-FILE-1 LABEL RECORD STANDARD.
01  OUTPUT-RECORD-1 PIC X(13).
FD  OUTPUT-FILE-2 LABEL RECORD STANDARD.
01  OUTPUT-RECORD-2 PIC X(13).
SD  SORT-FILE LABEL RECORD STANDARD.
01  SORT-RECORD.
    02  S0          PIC X.
    02  S1          PIC 9(4).
    02  S2          PIC 9(4).
    02  S3          PIC 9(4).
WORKING-STORAGE SECTION.
01  INPUT-STATUS PIC X VALUE LOW-VALUE.
    88  INPUT-END VALUE HIGH-VALUE.
01  SORT-STATUS PIC X VALUE LOW-VALUE.
    88  SORT-END VALUE HIGH-VALUE.
PROCEDURE DIVISION.
MAIN SECTION.
M1.
    OPEN INPUT MASTER-FILE OUTPUT OUTPUT-FILE-1 OUTPUT-FILE-2.
    SORT SORT-FILE ASCENDING S1 S2 S3             (1)
         INPUT PROCEDURE IPROC                     |
         OUTPUT PROCEDURE OPROC.                  (1)
    CLOSE MASTER-FILE OUTPUT-FILE-1 OUTPUT-FILE-2.
ME.
    STOP RUN.
IPROC SECTION.
IP0.
    PERFORM UNTIL INPUT-END                       (2)
      READ INPUT                                   |
      AT END                                       |
        SET INPUT-END TO TRUE                      |
      NOT AT END                                   |
        IF E0 NOT = "C"                            |
          THEN                                     |
          RELEASE SORT-RECORD FROM INPUT-RECORD    |
        END-IF                                     |
      END-READ                                     |
    END-PERFORM.                                  (2)
OPROC SECTION.
A0.
    PERFORM UNTIL SORT-END                        (3)
      RETURN SORT-FILE                             |
      AT END                                       |
        SET SORT-END TO TRUE                       |
      NOT AT END                                   |
        IF S0 = "A"                                |
        THEN                                       |
          WRITE OUTPUT-RECORD-1 FROM SORT-RECORD   |
```

```
        ELSE                                          |
          WRITE OUTPUT-RECORD-2 FROM SORT-RECORD      |
        END-IF                                        |
      END-RETURN                                      |
    END-PERFORM.                                     (3)
```

(1)

1. Control passes to the input procedure (IPROC SECTION).
2. The records in the sort-file are sorted in ascending order according to S1 S2 S3.
3. Control passes to the output procedure (OPROC SECTION).

(2) A record is read from the input file. Only those records without a "C" in their first character position are to be processed. If the INPUT-RECORD is valid, it is released to the sort-file as SORT-RECORD. This process continues until the end of the input-file is encountered. Control then returns to the SORT statement.

(3) A record is released from the sort-file. If its first character position contains an "A", it is written to the first output file (OUTPUT-FILE-1). All other records are written to the second output file (OUTPUT-FILE-2). When all the records in the sort-file have been processed, the statement following the SORT statement is executed.

Format 2 Sorting tables

```
SORT data-name-2 {ON {ASCENDING | DESCENDING} {KEY | KEY-YY} {data-name-1}... }...
    [WITH DUPLICATES IN ORDER]
    [COLLATING SEQUENCE IS alphabet-name]
    [USING data-name-3]
```

Syntax rules

1. The SORT statement may be used anywhere in the Procedure Division except in DECLARATIVES and input and output procedures that belong to a SORT statement.
2. data-name-2 and data-name-3, if USING is specified, specify the table to be sorted.data-name-1... denotes one or more data items to be used as sort keys (key fields).
3. The record specified by data-name-2 must be defined in a data description entry, and is subject to the following rules:
   a. data-name-2 may be qualified.
   b. The data description entry for data-name-2 must contain an OCCURS clause, i.e. be defined as a a table element.
   c. If the table specified by data-name-2 is subordinate to a table (multidimensional table), then data-name-2 must be defined as an indexable table, i.e. an index name must be specified in the data description entry for the superordinate table by means of the INDEXED-BY phrase. Before execution of the SORT statement, the indexname must be supplied with the desired element number (see section "Indexing").
4. The key fields specified by data-name-1 must be defined in the data description entry for data-name-2, where the following rules apply:
   a. data-name-1 is either the same as data-name-2 or the same as a data item subordinate to data-name-2.
   b. data-name-1 may be qualified.

c. If data-name-1 refers to a data item subordinate to data-name-2, then the description of this data item must neither contain an OCCURS clause itself nor be subordinate to a data item whose description contains an OCCURS clause.

d. data-name-1 must not refer to a data item whose description contains a SIGN clause.

e. If the data item specified by data-name-1 is defined as a signed numeric, it must not comprise more than 16 digits.

f. The keys following the KEY-YY specification must be defined either with PIC 99 USAGE DISPLAY or with USAGE PACKED-DECIMAL.

5. For data-name-3, the same rules apply as for data-name-2.
6. If the USING phrase is used, both, data-name-2 and data-name-3 must be described as national data items or as alphanumeric resp. alphabetic data items.

General rules

1. The key words ASCENDING and DESCENDING apply to all subsequent data-name-1 specifications up to the next key word ASCENDING or DESCENDING.
2. The data items specified by data-name-1 are the sort keys. The sort keys are used hierarchically from left to right for sort processing, irrespective of whether ASCENDING or DESCENDING is specified. The first instance of data-name-1 is thus the main sort key, the second instance of data-name-1 is the next most significant sort key etc.
3. If DUPLICATES is specified and two or more table elements are found to match in respect of all their key fields, then the relative sequence of these table elements will not be changed by sort processing.
4. If DUPLICATES is not specified and two or more table elements are found to match in respect of all their key fields, then the relative sequence of these table elements will be undefined after sort processing.
5. The collating sequence that is used in comparing the alphanumeric key fields is defined as follows on commencement of execution of the SORT statement:

   a. If COLLATING SEQUENCE is specified in the SORT statement, this specification serves as the criterion for the collating sequence,

   b. If COLLATING SEQUENCE is not specified in the SORT statement, the program-specific collating sequence is used (see section "OBJECT-COMPUTER paragraph").

   The national (native) collating sequence is used for comparisons of national collating sequences.
6. The table elements sorted in accordance with the ASCENDING/DESCENDING KEY phrases are stored in the table specified by data-name-2.
7. The table elements are sorted through comparison of the contents of the data items defined as sort keys, in accordance with the rules for relation conditions:

   a. If the contents of the compared key fields differ and the ASCENDING phrase is in effect, then the table element whose key field contains the lower value has the lower element number.

   b. If the contents of the compared key fields differ and the DESCENDING phrase is in effect, then the table element whose key field contains the higher value has the lower element number.

   c. If the contents of the compared key fields are the same, the next sort key specified is used for the comparison.
8. Sort keys specified in the SORT statement take priority over any sort keys that may be specified in the data description entry for data-name-2.
9. By specifying USING data-name-3, a second table can be used for sort processing.
   In this case, the elements of the table specified by data-name-3 are sorted as described above and then

transferred to the table specified by data-name-2 in accordance with the rules for the MOVE statement. Note the following with regard to the transfer of the individual table elements:
If the "sending" element is shorter than the "receiving" element, it is padded with blanks;
if it is longer, it is truncated.

10. Transfer of the sorted data-name-3 table elements to the data-name-2 table ends with the last sorted table element (data-name-3) or on reaching the number of table elements for data-name-2. This means that if the data-name-3 table contains more elements than the data-name-2 table, the excess elements are not transferred. If it contains fewer elements, the excess elements of the data-name-2 table are retained unchanged.
11. Sort processing does not change the contents of the data-name-3 table.

Example 8-76

Sorting a table

```
IDENTIFICATION DIVISION.
PROGRAM-ID. TABSORT.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
    TERMINAL IS T.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 NAMETAB.
   02 TAB-ELEM OCCURS 10 TIMES.
      03 FORENAME    PIC X(8).
      03 FILLER      PIC X(3) VALUE SPACES.
      03 NAME        PIC X(10).
PROCEDURE DIVISION.
SINGLE SECTION.
INITIALIZATION.
    MOVE "PETER" TO FORENAME (1) MOVE "KRAUS" TO NAME (1).
    MOVE "JANE" TO FORENAME (2) MOVE "FONDA" TO NAME (2).
    MOVE "PETER" TO FORENAME (3) MOVE "FONDA" TO NAME (3).
    MOVE "KARL" TO FORENAME (4) MOVE "KRAUS" TO NAME (4).
    MOVE "UWE" TO FORENAME (5) MOVE "SEELER" TO NAME (5).
    MOVE "WALT" TO FORENAME (6) MOVE "DISNEY" TO NAME (6).
    MOVE "CLARA" TO FORENAME (7) MOVE "WIECK" TO NAME (7).
    MOVE "LEONID" TO FORENAME (8) MOVE "KOGAN" TO NAME (8).
    MOVE "ERICH" TO FORENAME (9) MOVE "FROMM" TO NAME (9).
    MOVE "ELVIS" TO FORENAME (10) MOVE "PRESLEY" TO NAME (10).
    DISPLAY NAMETAB UPON T.
SORTING.
    SORT TAB-ELEM ON ASCENDING KEY NAME FORENAME.
END.
    STOP RUN.
```

The following table is output with `AID %DISPLAY TABELLE`

Sorted table:

```
Element number
(1)            |WALT    |    |DISNEY
(2)            |JANE    |    |FONDA
```

```
(3)             |PETER   |   |FONDA
(4)             |ERICH   |   |FROMM
(5)             |LEONID  |   |KOGAN
(6)             |KARL    |   |KRAUS
(7)             |PETER   |   |KRAUS
(8)             |ELVIS   |   |PRESLEY
(9)             |UWE     |   |SEELER
(10)            |CLARA   |   |WIECK
```

8.10.42 START statement

Function

The START statement defines the logical starting point within a file for subsequent sequential read operations.

Format

```
START  file-name  [WITH NO LOCK]  [ KEY  { IS EQUAL TO
                                         | IS =
                                         | IS GREATER THAN
                                         | IS >
                                         | IS NOT LESS THAN
                                         | IS NOT <
                                         | IS GREATER THAN OR EQUAL TO

                                         | IS >=
                                         | IS LESS THAN
                                         | IS <
                                         | IS LESS THAN OR EQUAL TO
                                         | IS <=
                                         | IS NOT GREATER THAN
                                         | IS NOT >
                                        } file-name]


   [INVALID KEY imperative-statement-1]

   [NOT INVALID KEY imperative-statement-2]

   [END-START]
```

The format extension WITH NO LOCK is effective during shared updating of files; it is described in the "COBOL2000 User Guide" [1].

Syntax rules

1. Relational operators are mandatory separators. To avoid possible misinterpretations, they have not been underlined in the above format.
2. file-name must refer to a file in sequential or dynamic access mode.
3. data-name may be qualified.
4. For files with **relative file organization**, data-name must be the name of the RELATIVE KEY data item declared for this file.
5. For **indexed** files, data-name must be the name of a RECORD KEY or ALTERNATE RECORD KEY data item declared for that file or an alphanumeric or national data item that is subordinate to the RECORD KEY or ALTERNATE RECORD KEY item. In this case, the first character of both items must be identical.
6. The INVALID KEY phrase must be present if no applicable USE procedure has been specified.

General rules

For **relative and indexed file organization**, the following also applies:

1. The file specified by file-name must be open in the INPUT or I-O mode at the time the START statement is executed (see section "OPEN statement").
2. If the KEY phrase is omitted from the START statement, the relational operator EQUAL is assumed.
3. After execution of a START statement, the contents of the FILE STATUS data item (if specified) of that file are updated (see "FILE STATUS clause").
4. The relational operators have the following effect:
   a. For the relational operators EQUAL, GREATER, GREATER OR EQUAL, NOT LESS and their equivalents, the file position indicator is set to the first record that satisfies the relation.
   b. For the relational operators LESS, LESS OR EQUAL, NOT GREATER and their equivalents, the file position indicator is set to the last record that satisfies the relation.
   c. If the relation condition is not satisfied for any record in the file, an invalid key condition occurs, the file position indicator indicates “invalid” and the START statement is terminated unsuccessfully.
5. Transfer of control following execution of the START statement depends on whether INVALID KEY or NOT INVALID KEY has been specified (see "Invalid key condition").

   For **relative file organization**, the following also applies:
6. Execution of the START statement alters neither the content of the record area of the file, nor the file’s RELATIVE KEY nor the content of any data item referenced in a DEPENDING ON phrase of the RECORD clause associated with the file.
7. The type of comparison is specified by the relational operator in the KEY phrase. The position of each record in the file specified by file-name is compared with the contents of the data item referenced by data-name (RELATIVE KEY).

   For **indexed file organization**, the following also applies:
8. Execution of the START statement alters neither the content of the record area of the file nor the content of any data item referenced in a DEPENDING ON phrase of the RECORD clause associated with the file.
9. If an alternate key is specified in the KEY phrase of the START statement, that key becomes the key of reference for subsequent READ statements (format 2).
10. The type of comparison is specified by the relational operator in the KEY phrase. The logical key of each record in the file specified by data-name is compared with the contents of the data item referenced by data-name (RECORD KEY or ALTERNATE RECORD KEY). If the length of data-name differs from that of the RECORD KEY or ALTERNATE KEY item, the comparison is performed as though the larger data item had been truncated from the right to the length of the smaller item. All key comparisons take place on the basis of the native collating sequence, i.e. as though no PROGRAM COLLATING SEQUENCE IS NATIVE or PROGRAM COLLATING SEQUENCE have been specified.

8.10.43 STOP statement

Function

The STOP statement causes a permanent or temporary suspension of the execution of the object program (run unit).

Format

```
STOP {RUN | literal}
```

Syntax rules

1. The literal may be numeric, alphanumeric or any figurative constant except ALL literal.
2. If the literal is numeric, it must be an unsigned integer.
3. If a STOP statement with RUN phrase appears in a sentence, then it must be the only statement in that sentence, or it must be the last statement in a sequence of imperative statements.
4. The STOP statement with the RUN phrase specified terminates execution of the program, and returns control to the operating system.
5. If STOP literal is used, the literal is communicated to the system operator. In this case, only the system operator can resume the program. Continuation of the program begins with the next executable statement.

General rules

1. If the number of characters of the alphanumeric literal exceeds the hardware capability of the master console or subconsole, then more than one physical output operation will be performed to output the literal.
2. During the execution of a STOP RUN statement, an implicit CLOSE statement without any optional phrases is executed for each file that is in the open mode in the run unit. Any USE procedures associated with any of these files are not executed.

8.10.44 STRING statement

Function

The STRING statement moves and juxtaposes the partial or complete contents of two or more data items into a single data item.

Format

```
STRING {{identifier-1 | literal-1}... [DELIMITED BY {identifier-2 | literal-2 | SIZE}
]}...

    INTO identifier-3 [WITH POINTER identifier-4]

    [ON OVERFLOW imperative-statement-1]

    [NOT ON OVERFLOW imperative-statement-2]

    [END-STRING]
```

Syntax rules

1. Each literal may be any figurative constant, but it may not begin with ALL.
2. All literals must be described as non-numeric literals, and all identifiers, except identifier-4, must be described implicitly or explicitly as USAGE DISPLAY or USAGE NATIONAL. If the class of literal-1, literal-2, identifier-1, identifier-2 or identifier-3 is national, the class of all must be national.
3. Where identifier-1... are elementary numeric data items, they must be described as integers without the symbol P in their PICTURE character-strings.
4. If DELIMITED BY is not explicitly specified, DELIMITED BY SIZE is applied.
5. identifier-1..., literal-1... represent the sending items; identifier-3 represents the receiving item.
6. identifier-3 must not represent an edited data item and must not be described with the JUSTIFIED clause. identifier-3 must not represent a strongly typed data item.
7. identifier-4 is a counter item and must be described as an elementary numeric integer data item of sufficient size to accommodate the size of the data item referenced by identifier-3 plus the value 1.
   The symbol P is prohibited in the PICTURE character-string of identifier-4.
8. identifier-3 must not be subjected to reference modification.

General rules

1. identifier-2, literal-2 represent delimiters, i.e. they mark a character string up to which the contents of a sending item should be moved. If the SIZE phrase is used, the contents of the complete data item defined by identifier-1, literal-1 are moved.
2. If a figurative constant is specified as literal-1 or literal-2, it is regarded as a data item with the length 1 with the same USAGE as identifier-3.
3. When the STRING statement is executed, the transfer of data is governed by the following rules:
   a. The sending items literal-1 or the contents of identifier-1 are transferred to the receiving item referenced by identifier-3. The rules for the MOVE statement from national to national apply here if identifier-3 is of the national class, otherwise from alphanumeric to alphanumeric. In both cases no space filling will be provided. If identifier-1 is a zero-length item, it will not be moved.

b. If the DELIMITED BY phrase is specified without the SIZE phrase, the content of identifier-1... or the value of literal-1 is transferred character-by-character to the receiving data item, beginning with the leftmost character and continuing from left to right until the end of the sending data item is reached or until the character(s) specified by the delimiter referenced by literal-2, or the contents of identifier-2, are encountered. The delimiter is not transferred.

c. If the DELIMITED BY phrase is specified with the SIZE phrase or identifier-2 is a zero-length item, the entire contents of the sending items referenced by literal-1 or identifier-1 are transferred to the receiving item referenced by identifier-3 until all data has been transferred or the end of identifier-3 has been reached. The transfer takes place in the sequence specified in the STRING statement.

4. If the POINTER phrase is specified, the counter item referenced by identifier-4 is explicitly available to the user, i.e. an initial value greater than 0, and of adequate size to contain a value of the length of identifier-3 plus one byte, must be set by the user.
5. The subscripts of an identifier in the POINTER phrase are evaluated before the STRING statement is executed.
6. If the POINTER phrase is not specified, the following general rules apply as if the user had specified identifier-4 referencing a data item with an initial value of 1.
7. When moved to the receiving item identifier-3, each character is transferred separately from the sending item to the character position of identifier-3 which is determined by the value of the counter item identifier-4. Each time a character is moved, identifier-4 is incremented by 1. This is the only manner in which the value of identifier-4 changes during execution of the STRING statement.
8. At the end of execution of the STRING statement, only the portion of the receiving item referenced by identifier-3 into which characters were moved is changed. The rest of identifier-3 remains the same.
9. If at any time during or after the initialization of the STRING statement, but before processing is completed, the value associated with the counter item referenced by identifier-4 is either less than one or exceeds the number of character positions in the receiving item referenced by identifier-3, no (further) data is transferred to identifier-3, and the imperative statement in the ON OVERFLOW phrase, if specified, is executed.
10. If the ON OVERFLOW phrase is not specified when one of the conditions described above is encountered, control is transferred to the end of the STRING statement.
11. The imperative statement in the NOT ON OVERFLOW phrase is executed if the STRING statement ends without one of the conditions described above having been encountered.

Example 8-77

```
IDENTIFICATION DIVISION.
PROGRAM-ID. STRNG.
ENVIRONMENT DIVISION.
     CONFIGURATION SECTION.
     SPECIAL-NAMES.
         TERMINAL IS T.
DATA DIVISION.
WORKING-STORAGE SECTION.
77  FIELD1   PIC X(16)  VALUE "ANFANGSBEDINGUNG".
77  FIELD2   PIC X(12)  VALUE "WERTEBEREICH".
77  FIELD3   PIC X(25)  VALUE SPACES.
77  FIELD4   PIC 99     VALUE 3.
PROCEDURE DIVISION.
PROC SECTION.
MAIN.
    DISPLAY "Before STRING: " UPON T.
    PERFORM DISPLAY-FIELDS.
    STRING FIELD1, FIELD2 DELIMITED BY "B",
```

```
           " INVALID" DELIMITED BY SIZE
      INTO FIELD3 WITH POINTER FIELD4
    ON OVERFLOW
      DISPLAY "Error" UPON T
    END-STRING
    DISPLAY "After STRING" UPON T.
    PERFORM DISPLAY-FIELDS.
    STOP RUN.
DISPLAY-FIELDS.
    DISPLAY "Field1 = *" FIELD1 "*" UPON T.
    DISPLAY "Field2 = *" FIELD2 "*" UPON T.
    DISPLAY "Field3 = *" FIELD3 "*" UPON T.
    DISPLAY "Field4 = *" FIELD4 "*" UPON T.
```

Result:

Before STRING	**After STRING**
FIELD1 = *ANFANGSBEDINGUNG*	FIELD1 = *ANFANGSBEDINGUNG*
FIELD2 = *WERTEBEREICH*	FIELD2 = *WERTEBEREICH*
FIELD3 = * *	FIELD3 = * ANFANGSWERTE SETZEN *
FIELD4 = *03*	FIELD4 = *22*

8.10.45 SUBTRACT statement

Function

The SUBTRACT statement is used to subtract the value of a numeric item, or the sum of two or more values of numeric data items, from one or more items.

Format 1 of the SUBTRACT statement stores the difference in one of the operands.
In a SUBTRACT statement, more than one subtraction may be specified by supplying more than one result item.

Format 2 of the SUBTRACT statement uses the GIVING phrase.

Format 3 of the SUBTRACT statement subtracts the items in one group item from the corresponding items in another group item.

Format 1

```
SUBSTRACT {identifier-1 | literal-1}...
        FROM {identifier-n [ROUNDED]}...
        [ON SIZE ERROR imperative-statement-1]
        [NOT ON SIZE ERROR imperative-statement-2]
        [END-SUBTRACT]
```

Syntax rules

1. Each identifier must refer to an elementary numeric data item.
2. The composite of operands determined by using all of the operands in a given statement, with the exception of the data item following the word GIVING, must not contain more than 31 digits
(see "Arithmetic statements").
3. All literals and identifiers preceding the word FROM are added together, and the sum is subtracted from the current value of identifier-n... . The results of the subtraction are stored as the new value of identifier-n.

Additional rules are given under "Phrases in statements", where the ROUNDED and (NOT) ON SIZE ERROR phrases are described.

Example 8-78

Statement	PICTURE of result item	Calculation
`SUBTRACT A, B FROM D`	999	D - (A + B) stored in D as nnn
`SUBTRACT A FROM B,C`	9(3) for B 9(5) for C	B - A stored in B as nnn C - A stored in C as nnnnn

Format 2

```
SUBSTRACT {identifier-1 | literal-1}... FROM {identifier-m | literal-m}
         GIVING {identifier-n [ROUNDED]}...
        [ON SIZE ERROR imperative-statement-1]
        [NOT ON SIZE ERROR imperative-statement-2]
        [END-SUBTRACT]
```

Syntax rules

1. Each identifier preceding the word GIVING must refer to an elementary numeric item.
2. identifier-n may refer either to an elementary numeric item or to an elementary numeric edited item.
3. The composite of operands determined by using all of the operands in a given statement, except for the data items which follow the word GIVING, must not contain more than 31 digits (see "Arithmetic statements").
4. All literals and identifiers preceding the word FROM are added together, and the sum is subtracted from literal-m or identifier-m. The results of the subtraction are stored as the new value of identifier-n.

Additional rules are given under "Options in arithmetic statements" (Phrases in statements et seq.), where the GIVING, ROUNDED, and (NOT) ON SIZE ERROR phrases are described.

Example 8-79

Statement	PICTURE of result item (C)	Calculation
`SUBTRACT A, B FROM 100 GIVING C.`	9(5)	100 - (A + B) stored in C as nnnnn

Format 3

```
SUBSTRACT { CORR | CORRESPONDING } identifier-1
         FROM identifier-2 [ROUNDED]
        [ON SIZE ERROR imperative-statement-1]
        [NOT ON SIZE ERROR imperative-statement-2]
        [END-SUBTRACT]
```

Syntax rules

1. Each identifier must refer to a group item.
2. The composite of operands, which is determined separately for each pair of corresponding data items, must not be greater than 31 digits (see "Arithmetic statements").
3. The elementary items within the first operand (identifier-1) are subtracted from the corresponding elementary items within the second operand (identifier-2). The results are stored in the items of the second operand.

Additional rules are given under "Options in arithmetic statements" (Phrases in statements et seq.), where the CORRESPONDING, ROUNDED, and (NOT) ON SIZE ERROR phrases are described.

Example 8-80

Refer to the description of the CORRESPONDING phrase for an example of the use of this phrase (CORRESPONDING phrase).

8.10.46 UNSTRING statement

Function

The UNSTRING statement causes contiguous data in a sending item to be separated and placed into multiple receiving items.

Format

```
UNSTRING identifier-1

    [DELIMITED BY [ALL] {identifier-1 | literal-1} [OR [ALL] {identifier-3 | literal-2
    }]...]

     INTO {identifier-4 [DELIMITER IN identifier-5] [COUNT IN identifier-6]}...

    [WITH POINTER identifier-7] [TALLYING IN identifier-8]

    [ON OVERFLOW imperative-statement-1]

    [NOT ON OVERFLOW imperative-statement-2]

    [END-UNSTRING]
```

Syntax rules

1. literal-1 and literal-2 must be literals of the category alphanumeric or national. However, they may not be figurative constants which begin with ALL.
2. identifier-1, identifier-2, identifier-3, identifier-5 must reference data items described as alphanumeric or national.
3. If the category of any literal-1, literal-2, identifier-1, identifier-2, ..., identifier-5 is national, the category of all must be national.
4. identifier-4 may be described as either
   - the category national
   - the category alphabetic, alphanumeric or numeric. It must be described, implicitly or explicitly, as USAGE DISPLAY. A numeric data item must not be contain the character "P" in the PICTURE character-string.
5. identifier-6, identifier-7, identifier-8 must reference integer data items.The character "P" must not be used in the PICTURE character-string.
6. identifier-1 represents the sending area.
7. identifier-4 represents the receiving area, identifier-5 the receiving area for delimiters.
8. identifier-1 must not be subjected to reference modification.
9. DELIMITER IN and COUNT IN can be used only in conjunction with DELIMITED BY.
10. No identifier may be defined with the level number 88.

General rules

1. All references to identifier-2 and literal-1apply to identifier-3 and literal-2 and analogously to all recursions thereof.
2. If identifier-1 is a zero-length item:
   - The content of the identifier-4 to identifier-8 remain unchanged.

- Neither imperative-statement-1 nor imperative-statement-2 is executed.
- The runtime control passes immediately to the end of the UNSTRING statement.

3. If a figurative constant is specified as literal-1, it stands for a 1-character long national data item if identifier-1 is a national data item, otherwise for a 1-character long alphanumeric data item.
4. When the ALL phrase is specified, contiguous occurrences of literal-1 or identifier-2 are treated as if they were only one occurrence, and one occurrence of literal-1 or identifier-2 is moved to the data item referenced by identifier-5.
5. When two contiguous delimiters are encountered, the current receiving area is spacefilled if it is described as alphabetic, alphanumeric or national, or zero-filled if it is described as numeric.
6. literal-1, identifier-2 represent delimiters. When a delimiter contains two or more characters, all of the characters must be present in contiguous positions of the sending item, and in the order given, to be recognized as a delimiter.
   If identifier-2 is a zero-length item, it is not recognized as a delimiter.
7. When two or more delimiters are specified in the DELIMITED BY phrase, an OR condition exists between them. Each delimiter is compared with the sending item. If a match occurs, the character(s) in the sending item is (are) considered to be a single delimiter. No character(s) in the sending item can be considered as part of more than one delimiter. Delimiters cannot overlap.
   Each delimiter is applied to the sending item in the sequence specified in the UNSTRING statement.
8. When the UNSTRING statement is initiated, identifier-4 represents the current receiving area. Data is transferred from identifier-1 to identifier-4 according to the following rules:
   a. If the POINTER phrase is specified, the string of characters referenced by identifier-1 is examined beginning with the relative character position indicated by identifier-7.
      If the POINTER phrase is not specified, the string of characters is examined beginning with the leftmost character position of identifier-1.
   b. If the DELIMITED BY phrase is specified, the examination proceeds left to right until either a delimiter specified by literal-1 or the value of the data item referenced by identifier-2 is encountered.
      If the DELIMITED BY phrase is not specified, the number of characters examined is equal to the size of the current receiving area. However, if the sign of the receiving item is defined as occupying a separate character position, the number of characters examined is one less than the size of the current receiving area. If the end of identifier-1 is reached before a delimiter is encountered, the examination terminates with the character examined last.
   c. The characters thus examined (excluding the delimiting character(s), if any) are treated as an elementary national data item if identifier-1 is ntional. In all the other cases this characters are treated as an elementary alphanumeric data item.The characters are moved into the current receiving area according to the rules for the MOVE statement (see section "MOVE statement"), but a decimal point is ignored.
   d. If the DELIMITER IN phrase is specified, the delimiting character(s) are treated as an elementary national data item if identifier-1 is ntional. In all the other cases this characters are treated as an elementary alphanumeric data item.
      The characters are moved into the current receiving area according to the rules for the MOVE statement (see section "MOVE statement").
      If the end of identifier-1 was reached without finding a delimiter, blanks are moved to identifier-5.
   e. If the COUNT IN phrase is specified, a value equal to the number of characters thus examined (excluding the delimiter character(s), if any) is moved into the area referenced by identifier-6 according to the rules for an elementary move.
   f. If the DELIMITED BY phrase is specified, the string of characters referenced by identifier-1 is further examined beginning with the first character to the right of the delimiter.

If the DELIMITED BY phrase is not specified, the examination will continue from the character immediately following the last character to be moved.

g. After the data is transferred to identifier-4, the current receiving area is represented by the next recurrence of identifier-4.
The behavior described above is repeated until either all the characters in identifier-1 are exhausted or until there are no more receiving areas.

9. The initialization of the contents of the data items associated with the POINTER or TALLYING phrase is the responsibility of the user.
10. The contents of the data item referenced by identifier-7 will be incremented by 1 for each character examined in the data item referenced by identifier-1.

    When the execution of an UNSTRING statement with a POINTER phrase is completed, identifier-7 will contain a value equal to the initial value plus the number of characters examined in the data item referenced by identifier-1.
11. When the execution of an UNSTRING statement with a TALLYING phrase is completed, identifier-8 contains a value equal to its initial value plus the number of receiving data items accessed.
12. Either of the following situations causes an overflow condition:
    a. If an UNSTRING is initiated, and the value in the data item referenced by identifier-7 is less than 1 or greater than the size of the data item referenced by identifier-1.
    b. If, during execution of an UNSTRING statement, all receiving areas have been acted upon, and the data item referenced by identifier-1 contains characters that have not been examined.
13. When an OVERFLOW condition exists, the UNSTRING operation is terminated. If an ON OVERFLOW phrase has been specified, the imperative statement given in this phrase is executed. If the ON OVERFLOW phrase is not specified, control is transferred to the next statement to be executed.
14. The imperative statement specified in the NOT ON OVERFLOW phrase is executed if the UNSTRING statement is terminated and none of the conditions described above has been encountered.
15. Subscripts and indexes for the identifiers are analyzed as follows:
    a. If the items identifier-1, identifier-7, identifier-8 are subscripted or indexed, the index value for these items is calculated once only, immediately before the UNSTRING statement is executed.
    b. Any subscripting of identifiers in the DELIMITED BY, INTO, DELIMITER IN, and COUNT phrases is evaluated immediately before the data is transferred to the respective data item.

Example 8-81

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. UNSTRING.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SPECIAL-NAMES.
     TERMINAL IS T.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  FIELD      PIC  X(12)  VALUE  "ABCDEFGHIJKL".
 01  AREA1.
     02  PART1   PIC X         VALUE SPACES.
     02  PART2   PIC XX        VALUE SPACES.
     02  PART3   PIC XXX       VALUE SPACES.
 01  NUMB        PIC 99        VALUE ZERO.
```

```
PROCEDURE DIVISION.
PROC SECTION.
MAIN.
    DISPLAY "Before UNSTRING" UPON T.
    PERFORM DISPLAY-FIELDS.
*
    UNSTRING FIELD DELIMITED BY "E" OR "H" OR "K" OR "L"
        INTO PART3, PART2, PART1
        TALLYING IN NUMB.
    END-UNSTRING
*
DISPLAY "After UNSTRING" UPON T.
    PERFORM DISPLAY-FIELDS.
    STOP RUN.
DISPLAY-FIELDS.
    DISPLAY "Field = *" FIELD "*" UPON T.
    DISPLAY "Part1 = *" PART1 "*" UPON T.
    DISPLAY "Part2 = *" PART2 "*" UPON T.
    DISPLAY "Part3 = *" PART3 "*" UPON T.
    DISPLAY "Numb  = *" NUMB  "*" UPON T.
```

Result:

Before UNSTRING	**After UNSTRING**
FIELD = *ABCDEFGHIJKL*	FIELD = *ABCDEFGHIJKL*
PART1 = * *	PART1 = *I*
PART2 = * *	PART2 = *FG*
PART3 = * *	PART3 = *ABC*
NUMB = *00*	NUMB = *03*

8.10.47 USE statement

Function

The USE statement introduces declarative procedures and defines the conditions for their execution. The USE statement itself, however, is not executed.

Format 1 declares label handling routines.

Format 2 declares procedures to be run if an input/output error occurs for a file.

Format 3 same as format 2 but with the GLOBAL statement, and applies to nested programs only.

Format 4 declares procedures to be run if an exception condition has been triggered.

Format 5 declares procedures to be executed by the Report Writer before the output of listings (see the chapter "Report Writer").

Format 1 for sequential file organization

```
USE {BEFORE | AFTER} STANDARD [ENDING | BEGINNING] [REEL | UNIT | FILE] LABEL
PROCEDURE ON {file-name-1}...
```

Syntax rules

1. file-name-1 must be defined in a file description (FD) entry of the program's Data Division.
2. file-name-1 must not be the name of a sort file.
3. file-name-1 refers to the file description entry for which the specified label handling procedures are to be performed.
4. The procedures specified are executed in conjunction with OPEN and CLOSE statements for the file.
5. Once a USE procedure has been executed, program execution resumes with the calling routine.
6. BEFORE or AFTER indicates the types of labels to be processed.

   BEFORE indicates that nonstandard labels are to be processed (such labels may be specified only for tape files).
   AFTER indicates that user labels follow standard file labels and that these user labels are to be processed.
7. BEGINNING or ENDING indicates that header or trailer labels, respectively, are to be processed.
   If the BEGINNING and ENDING phrases are omitted, the declared procedures will be run for both header and trailer labels.
8. The phrases REEL, UNIT or FILE indicate that the declared procedures are to be run if volume, reel, or file labels are present.

   The REEL phrase is not applicable to disk storage files. The UNIT phrase is not applicable to files in the random access mode, since only file labels are processed in that mode. The compiler treats the REEL and UNIT phrases as interchangeable.
9. If the above phrases are omitted, the declarative procedures are executed depending on the type of volume involved, either for reel labels and file labels, or for volume labels and file labels.
10. Format 1 of the USE statement is not permitted for line sequential files.

General rules

1. The labels to be processed for a file must be specified within the file description entry of that file as data-names in the LABEL RECORDS clause. These labels must be defined as level-01 data items within the file description entry or the LINKAGE SECTION.
2. The same file-name may appear in more than one variant of the format-1 USE statement. However, this must not cause two or more declarative procedures to be initiated at the same time.
3. If the file-name-1 specification is used, the file description for the file-name must not specify a LABEL RECORDS clause with the OMITTED phrase.
4. No user label routines can be declared for *external* files.
5. The standard system procedures are performed on all standard label records consisting of system and user labels.
   a. Labels on input or input/output files are checked in the following order:
      1. The I-O system checks standard header labels.
      2. The USE procedures (if any) check user header labels.
      3. The I-O system checks standard trailer labels.
      4. The USE procedures (if any) check user trailer labels.
   b. Labels on output files are created in the following order:
      1. The I-O system creates standard header labels.
      2. The USE procedures (if any) create user header labels.
      3. Before the user header labels are written, they are checked to see whether they begin with the string UHL.
      4. The I-O system creates standard trailer labels.
      5. The USE procedures (if any) create user trailer labels.
      6. Before the user trailer labels are written, they are checked to see whether they begin with the string UTL.
6. Within a USE procedure, there must be no reference to nondeclarative procedures except for the PERFORM statement.
   References to procedure names which are subordinate to a USE statement may be made from another procedure or from a nondeclarative procedure by using a PERFORM statement only.
7. The exit from a format-1 declarative procedure is generated by the compiler following the last statement in a given section. All logical paths within that section must lead to this exit point.
8. There is one exception to general rule 7:

   The GO TO statement with the MORE-LABELS phrase may be used as a special exit point. After this statement is executed, the runtime system will take one of the following actions:

   - If labels are currently being created, the system will write the current header or trailer labels; the program will then continue at the beginning of the declarative section in order to create more labels. The user must make sure that the last statement in the section is executed after all labels are created. At this point, it should be noted that after execution of the GO TO statement with the MORE-LABELS phrase, a label will be written in any case; if the user did not supply a new label, the runtime system will generate a dummy label record in analogy to general rule 7.
   - If labels are currently being checked, the system will read an additional header or trailer label; the program will then continue at the beginning of the declarative section in order to check more labels. However, when

processing user labels, the system will reenter the section only when there is another user label to check. Thus, in this case, the programmer does not have to provide a program path that flows through the last statement in the section. On the other hand, when processing nonstandard labels, the system does not know how many labels exist. The last statement in the section must therefore be executed in this case in order to terminate nonstandard label processing.

Example 8-82

In this example, one declarative section (ALPHA) handles header labels and another (BETA) handles trailer labels. The declarative procedures are executed both in input and output modes.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. USESEQ.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
    TERMINAL IS T.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT SAMPLE-FILE ASSIGN TO "TESTTAPE".
DATA DIVISION.
FILE SECTION.
FD SAMPLE-FILE,
   RECORD CONTAINS 100 CHARACTERS,
   LABEL RECORD IS SAMPLE-LABEL,
   DATA RECORD IS SAMPLE-RECORD.
01 SAMPLE-LABEL.
   02 LABEL-ID                 PICTURE X(4).
   02 LABEL-INFO               PICTURE X(76).
01 SAMPLE-RECORD               PICTURE X(100).
WORKING-STORAGE SECTION.
77  I-O-INDICATOR              PICTURE X.
    88  INPUT-MODE             VALUE "I".
    88  OUTPUT-MODE            VALUE "O".
77  LABEL-COUNTER              PICTURE 9.
PROCEDURE DIVISION.
DECLARATIVES.
ALPHA SECTION.
    USE AFTER STANDARD BEGINNING FILE
        LABEL PROCEDURE ON SAMPLE-FILE.
ALPHA-1.
    IF INPUT-MODE
    THEN
      DISPLAY "512010 THIS LABEL READ:-" SAMPLE-LABEL
              UPON T
      GO TO MORE-LABELS
    ELSE
      IF LABEL-COUNTER = 0
      THEN
        MOVE "UHL1" TO LABEL-ID
        MOVE "THIS WAS PRODUCED BY ALPHA SECTION."
             TO LABEL-INFO
        DISPLAY "511030 THIS LABEL CREATED:-" SAMPLE-LABEL
                UPON T
        MOVE 1 TO LABEL-COUNTER
        GO TO MORE-LABELS
      ELSE
```

```
        MOVE "UHL2" TO LABEL-ID
        MOVE "SECOND LABEL PRODUCED BY ALPHA SECTION" TO LABEL-INFO
        DISPLAY "511530 AND THIS LABEL TOO:-"
               SAMPLE-LABEL UPON T
        MOVE 2 TO LABEL-COUNTER
      END-IF
    END-IF.
ALPHA-END.
    EXIT.
BETA SECTION.
    USE AFTER STANDARD ENDING FILE
        LABEL PROCEDURE ON SAMPLE-FILE.
BETA-1.
    IF INPUT-MODE
    THEN
      DISPLAY "522010 THIS LABEL READ:-" SAMPLE-LABEL
               UPON T
      GO TO MORE-LABELS
    ELSE
      IF LABEL-COUNTER = 0
      THEN
        MOVE "UTL1" TO LABEL-ID
        MOVE "THIS WAS PRODUCED BY BETA SECTION" TO LABEL-INFO
        DISPLAY "521030 THIS LABEL CREATED:-" SAMPLE-LABEL
                 UPON T
        MOVE 1 TO LABEL-COUNTER
        GO TO MORE-LABELS
      ELSE
        MOVE "UTL2" TO LABEL-ID
        MOVE "SECOND LABEL PRODUCED BY BETA SECTION" TO LABEL-INFO
        DISPLAY "521530 AND THIS LABEL TOO: "
                 SAMPLE-LABEL UPON T
        MOVE 2 TO LABEL-COUNTER
      END-IF
    END-IF.
BETA-END.
    EXIT.
END DECLARATIVES.
GAMMA SECTION.
HERE-GOES.
    MOVE "O" TO I-O-INDICATOR.
    OPEN OUTPUT SAMPLE-FILE
    CLOSE SAMPLE-FILE
    STOP RUN.
```

Example 8-83

In this example, UHL-FIELD, LABEL-NO and USER-INFO are common label items. Therefore, the unqualified reference to UHL-FIELD in the paragraph named L-1 is legitimate.

Data Division entries:

```
FD  FILE-1.
    ...
    LABEL RECORD IS LABEL-1
    ...
01  LABEL-1.
```

```
02  UHL-FIELD          PIC X(3).
02  LABEL-NO           PIC 9.
02  USER-INFO          PIC X(76).
01  RECORD-1.
    ...
FD  FILE-2
    ...
    LABEL RECORD IS LABEL-2
    ...
01  LABEL-2.
02  UHL-FIELD          PIC X(3).
02  LABEL-NO           PIC 9.
02  USER-INFO          PIC X(76).
01  RECORD-2.
```

Procedure Division statements:

```
L SECTION.
    USE AFTER STANDARD LABEL PROCEDURE ON INPUT.
L-1.
    IF UHL-FIELD NOT = "UHL" THEN STOP RUN.
    ...
    GO TO MORE-LABELS.
L-9.
    EXIT.
M SECTION.
    ...
HOUSEKEEPING SECTION.
FILE-OPEN.
    OPEN INPUT FILE-1, FILE-2.
```

Format 2 for all types of file organization

```
USE AFTER STANDARD {ERROR | EXCEPTION} PROCEDURE ON { {FILE-NAME-1}... | INPUT |
OUTPUT | I-O | EXTEND}
```

Syntax rules

1. The ERROR and EXCEPTION phrases are equivalent and may be used interchangeably.
2. file-name-1 must not appear in more than one USE statement. The INPUT, OUTPUT,I-O, and EXTEND phrases may each be specified only once.
3. Files referenced either implicitly (INPUT, OUTPUT, I-O and EXTEND) or explicitly (file-name-1, file-name-2, ...) in the USE statement need not have the same organization and access mode.

General rules

1. The USE procedures are performed:
   a. when an at end condition occurs, provided that the input/output statement in which this condition appears does not contain an AT END phrase or an invalid key or when an invalid key condition occurs if the input /output statement which encountered this condition does not include an INVALID KEY.
   b. when a severe error occurs (FILE STATUS CODE 30).

   If no corresponding USE procedure exists for the file when a) or b) occurs, the program will abort.
2. When file-name-1 is specified, the error handling procedures are executed only for the named files. No other USE procedures are performed for these files.
3. Before execution of the user error routine, the standard system error routines for input/output error handling are executed.
4. After a USE procedure is executed, control passes to the calling routine.
5. INPUT indicates that the specified procedures are executed only for files opened in input mode (OPEN statement with the INPUT phrase.)
6. OUTPUT indicates that the specified procedures are executed only for files opened in output mode (OPEN statement with the OUTPUT phrase).
7. I-O indicates that the specified procedures are executed only for files opened in I-O mode (OPEN statement with the I-O phrase).
8. EXTEND indicates that the specified procedures are executed only for files opened in extend mode (OPEN statement with the EXTEND phrase).
9. Within a USE procedure, there must be no reference to nondeclarative procedures, except for the PERFORM and RESUME statement.
10. Reference to procedure names which are subordinate to a USE statement may be made from another procedure or from a nondeclarative procedure by using a PERFORM statement only.
11. The statements within a USE procedure may not result in a USE procedure which is still active being activated again.

Example 8-84

```
IDENTIFICATION DIVISION.
PROGRAM-ID. USESEQ2.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
    TERMINAL IS T.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT MASTER-FILE ASSIGN "MASTER-FILE"
      FILE STATUS INDICATOR.
FILE SECTION.
FD  MASTER-FILE.
01  REC                    PIC X(80).
WORKING-STORAGE SECTION.
01  INDICATOR              PIC XX.
01  CLOSE-INDICATOR        PIC X VALUE "0".
88  FILE-CLOSED            VALUE "0".
88  FILE-OPEN              VALUE "1".
PROCEDURE DIVISION.
DECLARATIVES.
INPUT-ERROR SECTION.
    USE AFTER ERROR PROCEDURE ON MASTER-FILE.
STATUS-QUERY.
    IF INDICATOR = "10"
       DISPLAY "End of MASTER-FILE encountered" UPON T
    ELSE
       DISPLAY "Unrecoverable error (" Indicator
               ") for MASTER-FILE" UPON T
       IF FILE-OPEN
          CLOSE MASTER-FILE
       END-IF
       DISPLAY "Program terminated abnormally" UPON T
       STOP RUN
    END-IF.
END DECLARATIVES.
BEGIN SECTION.
WORK.
    OPEN INPUT MASTER-FILE.
    SET FILE-OPEN TO TRUE.
    READ MASTER-FILE.
    CLOSE MASTER-FILE WITH LOCK.
    SET FILE-CLOSED TO TRUE.
    OPEN INPUT MASTER-FILE.
    CLOSE MASTER-FILE.
    STOP RUN.
```

Example 8-85

```
IDENTIFICATION DIVISION.
     PROGRAM-ID. USEREL.
     ENVIRONMENT DIVISION.
     CONFIGURATION SECTION.
     SPECIAL-NAMES.
         TERMINAL IS T.
```

```
    INPUT-OUTPUT SECTION.
    FILE-CONTROL.
        SELECT OPTIONAL WORKFILE ASSIGN TO "WORKFILE"
        ORGANIZATION IS RELATIVE
        ACCESS MODE IS DYNAMIC
        RELATIVE KEY IS RELKEY
        FILE STATUS INDICATOR.
    DATA DIVISION.
    FILE SECTION.
    FD WORKFILE.
    01 REC.
        03 INHOLD                PIC X(100).
    WORKING-STORAGE SECTION.
    01 INDICATOR                 PIC XX.
    01  RELKEY                   PIC 9(4).
    01  CLOSE-INDICATOR          PIC X VALUE "0".
        88 FILE-CLOSED           VALUE "0".
        88 FILE-OPEN             VALUE "1".
    PROCEDURE DIVISION.

    DECLARATIVES.
    INPUT-ERROR SECTION.
        USE AFTER ERROR PROCEDURE ON WORKFILE.
    STATUS-QUERY.
        EVALUATE INDICATOR
        WHEN    "10"
          DISPLAY "End of WORKFILE encountered" UPON T
        WHEN    "22"
          DISPLAY "Record with key" RELKEY "ALREADY EXISTS" UPON T
        WHEN    "23"
          DISPLAY "Record with key" RELKEY "NON-EXISTENT" UPON T
        WHEN OTHER
          DISPLAY "Unrecoverable error "(" INDICATOR")"
                  "FOR FILE-INPUT" UPON T
          IF FILE-OPEN
          THEN
            CLOSE WORKFILE
          END-IF
          DISPLAY "Program terminated abnormally" UPON T
          STOP RUN
        END-EVALUATE.
    END-DECLARATIVES.
    MAIN SECTION.
    OPEN-CLOSE.
        ...
        STOP RUN.
```

Example 8-86

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. USEIND.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SPECIAL-NAMES.
     TERMINAL IS T.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
```

```
    SELECT MASTER-FILE ASSIGN "IN-OUT"
           ORGANIZATION IS INDEXED
           RECORD KEY ISAMKEY
           ACCESS MODE IS DYNAMIC
           FILE STATUS INDICATOR.
DATA DIVISION.
FILE SECTION.
FD  FILE1.
01   RECORD-FORMAT.
     03 ISAMKEY              PIC 9(8).
     03 CONTENTS             PIC X(72).
WORKING-STORAGE SECTION.
01  INDICATOR                PIC XX.
01  CLOSE-INDICATOR          PIC X VALUE "0".
  88  FILE-CLOSED            VALUE "0".
  88  FILE-OPEN              VALUE "1".
PROCEDURE DIVISION.
DECLARATIVES.
FILE-ERROR SECTION.
    USE AFTER ERROR PROCEDURE ON FILE1.
STATUS-QUERY.
    EVALUATE INDICATOR
    WHEN    "10"
      DISPLAY "End of FILE1 encountered" UPON T
    WHEN    "21"
      DISPLAY "Record with key" ISAMKEY
              "ALREADY EXISTS OR NOT IN"
              "ASCENDING ORDER" UPON T
    WHEN    "22"
      DISPLAY "Record with key" ISAMKEY
              "ALREADY EXISTS" UPON T
    WHEN    "23"
      DISPLAY "Record with key" ISAMKEY
              "NON-EXISTENT" UPON T
    WHEN OTHER
      DISPLAY "Unrecoverable error"
              "(" INDICATOR") FOR MASTER FILE" UPON T
      IF FILE-OPEN
      THEN
        CLOSE FILE1
      END-IF
      DISPLAY "Program terminated abnormally" UPON T
      STOP RUN
    END-EVALUATE.
END DECLARATIVES.
BEGIN SECTION.
OPEN-CLOSE.
   . . .
     STOP RUN.
```

Format 3 USE statement with GLOBAL phrase

```
USE GLOBAL AFTER STANDARD {EXCEPTION | ERROR} PROCEDURE ON {{file-name-1}... | INPUT
| OUTPUT | I-O | EXTEND}
```

The USE statement with the GLOBAL phrase can be specified in nested programs in order to define procedure declarations or label routines as global.

For the declaration of user label routines cannot contain a GLOBAL clause.

Syntax rules

1. A USE procedure in format 2 requires no FD entry with a GLOBAL clause.
2. For further syntax rules relating to the USE statement, refer to the formats above.

General rules

1. The GLOBAL phrase for a USE procedure has the same effect as the GLOBAL clause in data and file descriptions.
2. If an I-O statement requires the use of a USE procedure, then the valid USE procedure is selected from the sum total of all the USE procedures declared in the nested program, in accordance with the following rules of precedence:
   a. The valid USE procedure (see format 2) is the one defined in the same program.
   b. If a) does not apply, the valid USE procedure is the one declared as global in the *directly* superordinate (next outer) program.
   c. If neither a) nor b) applies, the valid USE procedure is the one declared as global in the *indirectly* superordinate program.

      Rule c) applies until all indirectly superordinate programs have been searched for the valid global USE procedure.

   If no suitable USE procedure is found, then none is executed.
3. As an extension to ANS85, the compiler described here also permits PERFORM statements that relate to program segments outside of the DECLARATIVES. These program segments may also contain CALL, GO TO, GOBACK and EXIT PROGRAM statements. When a global USE procedure is used, this may result in recursive calling of the program that activated the USE procedure. A recursive call (always inadmissible) is not detected until program execution time, and results in program abortion.
   A global USE procedure should not therefore execute any EXIT PROGRAM statement.

Example 8-87

of the scope of validity of global USE procedures

```
PROGRAM-ID. A-PROG.
...
FD  FILE1  GLOBAL.
...
   PROGRAM-ID. B-PROG.
   ...
       USE GLOBAL ... ON FILE1.              (1) Global USE procedure
   ...
      PROGRAM-ID. C-PROG.
      ...
          USE GLOBAL ... ON INPUT.           (2) Global USE procedure
          USE        ... ON OUTPUT.          (3) Local USE procedure
      ...
          OPEN OUTPUT FILE1.                 USE procedure (3)
          OPEN EXTEND FILE1.                 USE procedure (1)
      ...
        PROGRAM-ID. D-PROG.
        ...
            OPEN INPUT FILE1.                USE procedure (2)
        ...
          PROGRAM-ID. E-PROG.
          ...
          FD  FILE1.
          ...
          OPEN INPUT FILE1.                  USE procedure (2)
          OPEN OUTPUT FILE1.                 USE procedure (1)
          ...
          END PROGRAM E-PROG.
        END PROGRAM D-PROG.
      END PROGRAM C-PROG.
   END PROGRAM B-PROG.

   PROGRAM-ID. F-PROG.
   ...
       OPEN I-O FILE1.                       Not a USE procedure
   END PROGRAM F-PROG.
END PROGRAM A-PROG.
```

Format 4 for exception conditions

```
USE AFTER {EXCEPTION CONDITION | EC } {exception-condition-name | EC-ALL} ...
```

Syntax rules

1. exception-condition-name must be one of the names listed in table 45 in section "Exception conditions and exception statuses".
2. No reference to procedures outside the declaratives may be contained in a USE statement with the exception of the PERFORM and RESUME statements.
3. Procedure names which are subordinate to a USE statement may only be referenced from another procedure or outside the declaratives with a PERFORM statement.
4. EC means the same as EXCEPTION CONDITION.

General rules

1. The statements within a USE procedure may not result in a USE procedure which is still active being activated again.
2. If an exception condition is triggered, the first USE procedure in which the associated exception condition name is specified is selected.
   If this exception condition name is not specified in any USE procedure, the first USE procedure in which EC-ALL is specified is selected.
   If EC-ALL is also not specified in any USE procedure, the program run is aborted depending on the category of the exception condition (*fatal/non-fatal*) or continued at the next executable statement.
3. If a USE procedure is activated by a exception condition, the program run is aborted if the USE procedure was not exited by executing an explicit statement (e.g. GOBACK, RESUME).

8.10.48 WRITE statement

Function

The WRITE statement causes a record to be released to an output file.
Format 1 can also be used to control form feeds in print files.

Format 1 for sequential file organization

```
WRITE record-name [FROM identifier-1]

    [{AFTER | BEFORE} ADVANCING{{identifier-2 | integer} [LINES | LINE] | {mnemonic-
    name | PAGE}}]

    [AT {END-OF-PAGE | EOP} imperative-statement-1]

    [NOT AT {END-OF-PAGE | EOP} imperative-statement-2]

    [END-WRITE]
```

Syntax rules

1. record-name and identifier-1 must not refer to the same storage area.
2. record-name must be associated with a file description (FD) entry in the Data Division of the program and may be qualified.
3. record-name must not be part of a sort file.
4. If identifier-2 is used in the ADVANCING phrase, it must be the name of an elementary integer data item.
5. The value of integer or the data item referenced by identifier-2 must be greater than or equal to zero, but not greater than 15.

   For files with the ASSIGN entry PRINTER literal-1, integer must be greater than 15 and less than 100.
6. If END-OF-PAGE or NOT END-OF-PAGE is used, a LINAGE clause must be present as part of the file description entry for the corresponding file (see section "LINAGE clause").
7. The EOP and END-OF-PAGE phrases are equivalent.
8. mnemonic-name, if specified, must have an entry associated with it in the SPECIAL-NAMES paragraph.
9. mnemonic-name must not be supplied in the ADVANCING phrase if a LINAGE clause was specified for the file referenced in the WRITE statement.
10. ADVANCING PAGE and END-OF-PAGE may not be specified together in a single WRITE statement.

General rules

1. The record supplied by a WRITE statement is no longer available in the record area, unless the file associated with the record was specified in a SAME RECORD AREA clause or the execution of the WRITE statement was unsuccessful. The record is also available to any other files which may have been referenced in any SAME RECORD AREA clause together with the specified file.
2. Execution of a WRITE statement with the FROM phrase is equivalent to the execution of the following statements:

   ```
   MOVE identifier-1 TO record-name
   the same WRITE statement without the FROM phrase
   ```

   Data is transferred according to the rules for a MOVE statement without the CORRESPONDING phrase.

The content of the record area before execution of the implicit MOVE statement has no effect on the execution of the WRITE statement.

After the WRITE statement is successfully executed, the information is still available in the area referenced by identifier-1; however, as pointed out in general rule 1, this is not necessarily true for the record area.

3. If the end-of-volume condition is encountered on a multi-volume output file for tape or disk storage and sequential access mode, the following steps will be performed during execution of the WRITE statement:
   a. The standard volume header label procedures and the user volume trailer label procedures specified via a USE procedure are performed. The order in which these procedures are executed depends on the BEFORE/AFTER phrases supplied in the USE procedure, if any.
   b. A volume swap is effected.
   c. The standard volume header label procedures and the user volume header label procedures specified via a USE procedure are performed. The order in which these procedures are executed depends on the BEFORE/AFTER phrases supplied in the USE procedure, if any.
4. After execution of a WRITE statement, the value of the data item of any FILE STATUS clause existing for that file is updated (see also "FILE STATUS clause").
5. If a file is open in extend mode, the WRITE statement causes records to be added to the end of the file as if the file were open in output mode. The first record written after execution of the OPEN EXTEND statement is the successor of the last record in the file.
6. If the VARYING phrase has been specified for the file associated with record-name, the number of character positions in the record referenced by record-name must not be larger than integer-3 or smaller than integer-2 (see section "RECORD clause"). If no RECORD clause is specified with the VARYING phrase, the number of character positions must not be longer than the largest record (as determined by the record description entry) of the associated file. If either of these cases occur, the WRITE statement is unsuccessful, and the WRITE operation does not take place. The content of the record area remains unchanged. The I-O status of the file associated with record-name is set to a value indicating the cause of the condition (see section "I-O status").
7. Both the ADVANCING and the END-OF-PAGE phrases allow control of the vertical positioning of each individual line on a representation of a printed page.
8. When the BEFORE/AFTER ADVANCING phrase is used, the corresponding record is printed before or after the page is advanced according to the rules described below.
9. If the end-of-page condition does not occur during the execution of a WRITE statement with the NOT END-OF-PAGE phrase, control is transferred to imperative-statement-2 if the WRITE statement is executed successfully. The transfer of control takes place after the record is written and after updating of the I/O status of the file from which the record originates.

   If the execution of the WRITE statement is unsuccessful, the I/O status of the file is updated, then the USE procedure specified for this file is executed, and finally control is transferred to the imperative-statement-2.
10. The ADVANCING phrase causes the representation of the printed page to be advanced according to the following rules:
    - If identifier-2, integer or mnemonic-name is present, the page is advanced in accordance with the rules specified in table 34.
    - If PAGE is present, the record is printed either before or after positioning to the beginning of the first line on a new page. The beginning of a new page is defined either by the physical properties of the printer (advance to channel 1) or, if a LINAGE clause is present, by the beginning of the next logical page (see section "LINAGE clause").
11. If the ADVANCING phrase is omitted, a WRITE...AFTER ADVANCING 1 LINE will be executed for files with the ASSIGN specification PRINTER or PRINTER literal-1 (see also table 35).

12. If the end of a logical page is encountered during execution of a WRITE statement with the END-OF-PAGE phrase, control is transferred to imperative-statement-1. The logical end of a page is defined by the LINAGE clause in the file description entry of the file.
13. An END-OF-PAGE condition occurs when the execution of a WRITE statement with the END-OF-PAGE phrase causes printing or spacing within the page footing area. This is the case whenever, during the execution of such a WRITE statement, the value of the special register LINAGE-COUNTER becomes equal to or greater than the value of integer-2 of the LINAGE clause data item referenced by data-name-2. In this case, the WRITE statement is executed, and control is then transferred to the imperative statement which follows the END-OF-PAGE phrase.
14. An automatic page overflow condition occurs when the execution of a given WRITE statement (with or without an END-OF-PAGE phrase) cannot be fully accommodated within the current page body.

    This is the case whenever, during the execution of such a WRITE statement, the value of the special register LINAGE-COUNTER becomes equal to or greater than the value of integer-1 or the LINAGE clause data item referenced by data-name-1. In this case, the record is printed either before or after positioning to the first line of the next logical page (depending on the BEFORE/AFTER phrase). This line is defined by specification of the LINAGE clause. The imperative statement specified in the END-OF-PAGE phrase is executed after the record has been written and the device repositioned.

    If the integer-2 or data-name-2 phrases of the LINAGE clause are not used, the END-OF-PAGE condition will occur whenever a WRITE statement is executed (with EOP phrase specified) and the value in the special LINAGE-COUNTER register is greater than or equal to the value of integer-1 or the LINAGE clause data item referenced by data-name-1.

    If integer-2 or data-name-2 is specified in the LINAGE clause but execution of a WRITE statement would result in the value of the LINAGE-COUNTER being both

    - equal to or greater than the value of data-name-2 or integer-2, and
    - equal to or greater than the value of data-name-1 or integer-1

    processing proceeds as if integer-2 or data-name-2 had not been specified.
15. When the ADVANCING phrase is used together with the LINAGE clause for a device which does not represent a physical printer, any blank lines at the top and bottom margins as well as within the page body are represented on the appropriate storage medium as records containing blank characters.
16. Execution of a WRITE statement which attempts to write outside the physical boundaries of a file results in an exception condition. The following steps are performed:
    - The value of the FILE STATUS data item of the file is set to indicate violation of the boundaries of the file (see section "FILE STATUS clause").
    - Any USE ERROR/EXCEPTION procedure that is explicitly or implicitly specified for this file will be executed.
    - If there is neither an explicit nor an implicit USE procedure for this file, the program will terminate abnormally.
17. When a WRITE statement is executed, the file must have been opened with an OPEN statement containing the OUTPUT or EXTEND phrase.
18. Table 36 gives precise detail of the use of the first character in the record, depending on the symbolic device names of the ASSIGN clause and the WRITE statement phrases.
19. When a WRITE statement is used with AFTER-ADVANCING phrase, the printer is positioned to the line requested and is advanced one additional line after printing.

    If PRINTER literal-1 is specified in the ASSIGN phrase, a WRITE AFTER phrase is performed as follows:

    - An empty record is printed with corresponding form-feed specification (WRITE BEFORE).
    - The record is written.

For all devices, form-feed suppression is only possible when the WRITE BEFORE phrase is specified.

Example 8-88

The file LOADFILE is read and written to the print file called AFILE.
Each current record is written before advancing 5 blank lines.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. WP1.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT AFILE ASSIGN TO PRINTER "PRINTOUT".
    SELECT LOADFILE ASSIGN TO "MASTER-FILE".
DATA DIVISION.
FILE SECTION.
FD   AFILE
     LINAGE IS 24 LINES
     LINES AT TOP 24
     LINES AT BOTTOM 24.
01  CUSTMRREC  PIC X(95).
FD   LOADFILE
     LABEL RECORD IS STANDARD.
01  CUSTREC.
    05 CNA PIC X(30).
    05 STR PIC X(20).
    05 LOC PIC X(20).
    05 ZIP PIC X(5).
    05 TYP PIC X(20).
WORKING-STORAGE SECTION.
01  FILE-SWITCH PIC X VALUE LOW-VALUE.
   88  EOF VALUE HIGH-VALUE.
PROCEDURE DIVISION.
MAIN SECTION.
FILE-OPEN.
    OPEN OUTPUT AFILE.
    OPEN INPUT LOADFILE.
READS.
    PERFORM UNTIL EOF
     READ LOADFILE INTO CUSTMRREC
     AT END
       CLOSE AFILE LOADFILE
       SET EOF TO TRUE
     NOT AT END
       WRITE CUSTMRREC BEFORE ADVANCING 5.
     END-READ
    END-PERFORM
    STOP RUN.
```

Indicator	Format	Action
identifier-2	Numeric integer with a value in the range from 01 to 15	n-line spacing
identifier-2	Alphanumeric data item, length 1 (PIC X). Following values are valid:	

	blank 0 - + 1-8 A B	Single-spacing Double-spacing Triple-spacing Suppress spacing Skip to channel 1-8 Skip to channel 10 Skip to channel 11	
integer	Numeric literal in the range 01 to 15	n-line spacing	
mnemonic-name	A name related to an implementor-name in a SPECIAL-NAMES paragraph.	implementor-name	Action
		C01 through C08 C10 OR C11	Skip to channel 1-8 Skip to channel 10-11

Table 34: Meaning and result of spacing indicators in a WRITE ADVANCING statement

Symbolic device name	**WRITE statement without ADVANCING phrase**	**WRITE statement with ADVANCING phrase**	**Comments**
PRINTER literal	Standard spacing when ADVANCING is omitted as if AFTER 1 LINE had been specified; the first character of the record is available for user data.	The first character of the record is available for user data.	The place for the carriage control character is reserved by the compiler and is not accessible to the user. This type of printer supports specification of the LINAGE clause in the file description entry. WRITE statements both with and without the ADVANCING phrase specified are allowed for a given file.
PRINTER PRINTER01-PRINTER99	As above.	As above.	The place for the carriage control character is reserved by the compiler and is not accessible to the user. The LINAGE clause is not permitted for this file. Use of WRITE statements with and without the ADVANCING phrase for the same file is not permitted. If this does occur, a WRITE AFTER ADVANCING is executed implicitly for the records without the ADVANCING phrase.
literal	Spacing is controlled by the first character in each logical record; the user must therefore supply the appropriate control	The user must reserve the first character of a logical record; the runtime system inserts the feed control character	Mixed use of WRITE statements both with and without specifications of the ADVANCING phrase is permitted. In either case, however, the user information of the printer record begins only with the second character of the record.

	character before every execution of such WRITE statement.	of the record program execution. Any user data there will be overwritten.	

Table 35: Use of symbolic device names for printers in connection with the WRITE statement

Format 2 for relative and indexed file organization

```
WRITE record-name [FROM identifier-1]
    [INVALID KEY imperative-statement-1]
    [NOT INVALID KEY imperative-statement-2]
    [END-WRITE]
```

Syntax rules

1. record-name and identifier-1 must not reference the same storage area.
2. record-name must be associated with a file description (FD) entry in the Data Division of the program and may be qualified.
3. The INVALID KEY phrase must be specified for a file unless an applicable USE procedure was declared.

General rules

For **relative and indexed file organization**, the following also applies:

1. The file whose record is referenced by the WRITE statement must be open in the OUTPUT, I-O or EXTEND mode.
2. The record released by a WRITE statement is no longer available in the record area, unless the file associated with the record was specified in a SAME RECORD AREA clause, or the execution of the WRITE statement was abnormally terminated as unsuccessful because of the occurrence of an invalid key condition. The record is also available to those files which were referenced in a SAME RECORD AREA clause together with the specified file.
3. Execution of a WRITE statement with the FROM phrase is equivalent to execution of the following statements:

   ```
   MOVE identifier-1 TO record-name
   the same WRITE statement without the FROM phrase
   ```

   Data is transferred according to the rules for a MOVE statement without the CORRESPONDING phrase.

   The contents of the record area before execution of the implicit MOVE statement has no effect on the execution of the WRITE statement.
4. Execution of a WRITE statement causes the contents of the data item that was specified in the FILE STATUS clause of the related file description entry to be updated (see also section "FILE STATUS clause").
5. If, during the execution of a WRITE statement with the NOT INVALID KEY phrase, the invalid key condition does not occur, control is transferred to imperative-statement-2 as follows:
   a. If the execution of the WRITE statement is successful: after the record is written and after updating the I/O status of the file from which the record originates.
   b. If the execution of the WRITE statement is unsuccessful: after the I/O status of the file has been updated and after executing the USE procedure that was specified for the file from which the record originates.
6. Occurrence of an invalid key condition indicates that the WRITE statement was unsuccessful; the contents of the record area are still available, and any existing FILE STATUS data item in that file is set to a value

indicating the cause of the invalid key condition. The program resumes execution in accordance with the rules for the invalid key condition.

7. The number of character positions in the record referenced by record-name must not be larger than the largest or smaller than the smallest number of character positions allowed by the associated RECORD IS VARYING clause. Otherwise, the WRITE statement is unsuccessful, the WRITE operation does not take place, the content of the record area is unaffected and the I-O status of the file associated with record-name is set to a value indicating a record length conflict (see section "I-O status").

For **relative file organization**, the following also applies:

8. When a WRITE statement is executed, the position within the file of the record to be output is located by means of the contents of the RECORD KEY data item. Before the WRITE statement is executed, the content of the associated key field must be set accordingly (see RELATIVE KEY in section "ACCESS MODE clause").
9. If a file is opened in output mode (OPEN statement with OUTPUT phrase), the following should be noted:
   a. In sequential access mode, the WRITE statement causes output of a record for creation of a new file. The first record is assigned a relative record number of 1 (one) while the subsequent records are numbered 2, 3, 4,... If RELATIVE KEY was specified, the File Control Processor will enter the relative record number in the RELATIVE KEY phrase during the execution of the write statement.
   b. In random or dynamic access modes (which are equivalent for OUTPUT), the value of the data item in the (here mandatory) RELATIVE KEY phrase must be set by the user to equal the relative record number which is to be assigned to the record contained in the record area. That record is then output as the nth record of the file, where "n" is the value of the relative record number.
10. If a file is in extend mode (OPEN statement with EXTEND phrase), a WRITE statement causes a record to be appended to the file. The first record released in this manner receives a relative record number which is 1 higher than the highest existing relative record number in the file. The relative record number of each subsequent record is incremented by 1 (with respect to the previous record). If the RELATIVE KEY phrase is specified, the relative record number of the MOVE statement is transferred to the record key field with each WRITE statement.
11. If a file is opened in update mode (OPEN statement with I-O phrase), and its access mode is either random or dynamic (which means the same in this case), the WRITE statement inserts records in the associated existing file. The value of the data item of the (here mandatory) RELATIVE KEY phrase must be set by the user to equal the relative record number which is to be assigned to the record contained in the record area. As the WRITE statement is executed, the contents of the record with the appropriate record number are transferred to the File Control Processor.
12. The invalid key condition is caused by the events listed in table 36.

ACCESS MODE clause	**OPEN mode and action taken**	**Reason for INVALID KEY condition**
SEQUENTIAL	OUTPUT or EXTEND A record is appended to an existing or newly created file.	There is no room in the file for the new record.
RANDOM or DYNAMIC	OUTPUT or I-O A record is appended to an existing file.	The content of the RELATIVE KEY field specifies a record which already exists in the file or there is no room in the file for the new record.

Table 36: WRITE statement - Causes of invalid key conditions

For **indexed file organization**, the following also applies:

13. When a WRITE statement is executed, the position within the file of the record to be output is located by means of the contents of the RECORD KEY data item. Before the WRITE statement is executed, the content of the associated key field must be set accordingly (see RELATIVE KEY in section "RECORD KEY clause").
14. If a file is opened in output mode (OPEN statement with OUTPUT phrase), the following should be noted:
    - In sequential access mode, the WRITE statement releases a record for the creation of a new file. In this context, the records must be transferred in ascending order of RECORD KEY values. Before execution of the WRITE statement, the RECORD KEY data item must be set to the required value.
    - In random or dynamic access modes (which are equivalent for OUTPUT), records may be released to the File Control Processor in any program-specified order.
15. The invalid key condition is caused by the events listed in table 37.

Access mode	**OPEN mode and action taken**	**Cause of invalid key condition**
SEQUENTIAL	OUTPUT / EXTEND A record is added to a file to be newly created. ("load mode").	Either the value of the primary key is not greater than that of the preceding record (the primary record keys must be sorted in ascending alphanumeric order), or no more space is available in the file for writing the record.
RANDOM or DYNAMIC	OUTPUT/ I-O / EXTEND A record is added to an existing file	The record has the same primary key value as a record already existing in the file, or no more space is available on the file for writing the record, or an alternate key value for which duplicates are not permitted already exists in a record in the file.

Table 37: WRITE statement - Causes of invalid key conditions

16. If the file was opened in extend mode, the first record passed to DMS must have a primary key whose value is higher than the highest existing primary key value in the file.
17. If ALTERNATE RECORD KEY WITH DUPLICATES is specified, the alternate key value of the record does not need to be unique.

9 Intrinsic functions

Intrinsic functions make it possible to reference a temporary data item that is made available by the COBOL program and whose value is calculated automatically when the function is used.

9.1 General

Function-name

Each intrinsic function has a name that the programmer uses to address it. A function-name is a key word from a special list of COBOL words and is a necessary part of the function-identifier. Outside the function-identifier context, a key word can also be used as a user-defined word.

Returned value of a function

Each function that executes successfully returns a function result, the *returned value*. To determine the returned value, the function processes the data values supplied by the arguments specified in the function-identifier.
The function result is defined

- by the length of the returned value in alphanumeric and national functions,
- by the sign of the returned value, or by the integer in the case of numeric and integer functions,
- or, in all remaining cases, by the returned value itself.

For the arguments particular rules apply: data type, number, length and value range of the arguments are determined by the definition of the function. The function returns a defined returned value only if these rules are observed.

Error default value

If a function is supplied with an invalid argument, the function result is undefined. A compiler option can be used in such cases to check the argument values and assign the function the *error default value* that indicates that the function has not executed successfully. A further compiler option can be used to ensure that the error is reported at run time (error messages COB9123 - COB9128). Refer to the "COBOL2000 User Guide" [1] for further information.

Date conversion

The Gregorian calendar is used in the date conversion functions.
The start date of Monday 1, January 1601 was selected to establish a simple relationship between the standard date and DAY-OF-WEEK; in other words, the integer date form 1 was a Monday, DAY-OF-WEEK 1.

Arguments

Arguments specify the values used when executing a function. The arguments are specified in the function-identifier (see section "Function-identifier") as identifiers, arithmetic expressions, or literals. The format description of each function indicates the number of arguments required (zero, one or more). For some functions the number of arguments can be variable.

Arguments belong to a particular data class or to a subset of a data class. There are the following argument types:

- Numeric. An arithmetic expression must be specified. The value of the arithmetic expression, including the sign, is used to determine the function value.
- Alphabetic. A data item of class alphabetic or an alphanumeric literal comprising exclusively alphabetic characters must be specified. The length of the argument can be used to determine the function value.
- Alphanumeric. A data item of class alphabetic or alphanumeric or an alphanumeric literal must be specified. The length of the argument can be used to determine the function value.
- National. A national data item or a national literal must be specified.The length of the argument can be used to determine the function value.

- Integer. An arithmetic expression that always produces an integer value must be specified. The value of the arithmetic expression, including sign, is used to determine the function value.
- Of class "object".
- Of class "pointer".
- Type declaration.
- Index. A data item with USAGE INDEX must be specified.

A table can be referenced if the format of a function permits argument repetition. Reference is made by specifying the data-name and any identifiers for the table, followed directly by subscription in which one or more subscripts is the word ALL.
If ALL is specified as a subscript, it has the same effect as specifying each table element at this subscript position.

Example

ALL subscripts in a three-dimensional table with ten items in each dimension.

```
FUNCTION MAX (TAB(ALL 2 ALL))
```

is the same as

```
FUNCTION MAX (TAB(1 2 1) TAB (1 2 2) ... TAB(1 2 10)
              TAB(2 2 1) TAB (2 2 2) ... TAB(2 2 10)
              .
              .
              TAB(10 2 1) TAB (10 2 2) ... TAB(10 2 10))
```

If the ALL subscript is linked with an OCCURS DEPENDING ON clause, the value range is determined by the object of the OCCURS DEPENDING ON clause. This object must be greater than zero at the time of evaluation of the arguments. If an argument subscripted with ALL is reference-modified, the reference-modifier refers to each implicitly referenced table item.

Function types

Functions are elementary data items. They return alphanumeric, national, numeric or index values, and cannot be receiving operands. There are the following types of function:

- Alphanumeric functions. They belong to class and category alphanumeric.
  Alphanumeric functions have implicit USAGE DISPLAY.
- National functions. They belong to class and category national.
  National functions have implicit USAGE NATIONAL.
- Numeric functions. They belong to class and category numeric.
  A numeric function is always treated as signed.
  A numeric function can be used only in an arithmetic expression.
  A numeric function must not be referenced if an integer operand is required, even if evaluation of the function results in an integer value.
- Integer functions. They belong to class and category "numeric".
  An integer function is always treated as signed.
  An integer function may be used only in an arithmetic expression.
- Index functions. They belong to class and category index.

9.2 Overview of intrinsic functions

The following table lists the available functions.

The "Arguments" column defines type and number of arguments as follows:

A	alphabetic
I	integer
Ind	index
Nat	national
Num	numeric
O	object reference
P	pointer
T	type declaration
X	alphanumeric[1]

[1] X in the "Arguments" column contains stongly typed group items. If the argument type deter-mines the function type, then the function is alphanumeric as soon as just one of its arguments is strongly typed. This also applies if all the function's arguments are of the same type. Strongly typed arguments are excluded for the ADDR, MAX, MIN, ORD-MAX and ORD-MIN functions.

The "Type" column defines function type as follows:

I	integer
Ind	index
Nat	national
Num	numeric
X	alphanumeric

Function-name	Arguments	Type	Returned value
ACOS	Num1	Num	Arccosine of Num1
ADDR	A1 or Ind1 Nat1 or Num1 or X1	I	Address of the argument
ANNUITY	Num1, I2	Num	Ratio of annuity paid for 12 periods at interest of Num1 to initial investment of one
ASIN	Num1	Num	Arcsine of Num1
ATAN	Num1	Num	Arctangent of Num1

BYTE-LENGTH	A1 or Ind1 or N1 or Nat1 or O1 or P1 or T1 or X1	I	Length of the arguments in bytes
CHAR	I1	X	Character in position I1 of the alphanumeric collating sequence
CHAR-NATIONAL	I1	Nat	Character in position I1 of the national collating sequence
COS	Num1	Num	Cosine of Num1
CURRENT-DATE	None	X	Current data and time
DATE-OF-INTEGER	I1		Standard date equivalent (YYYYMMDD) of integer date (number of days)
DATE-TO-YYYYMMDD	I1, I2	I	YYYYMMDD equivalent, depending on the value of I2, of I1 from a standard date with a 2-digit year (YYMMDD)
DAY-OF-INTEGER	I1	I	Julian date equivalent (YYYYDDD) of integer data
DAY-TO-YYYYDDD	I1, I2	I	YYYYDDD equivalent, depending on the value of I2, of I1 from a standard Julian date (YYDDD)
DISPLAY-OF	Nat1, X2	X	alphanumeric display the national character set Nat1
EXCEPTION-STATUS	none	X	Name of the last exception status
FACTORIAL	I1	I	Factorial of I1
INTEGER	Num1	I	Greatest integer not greater than Num1
INTEGER-OF-DATE	I1	I	Integer data equivalent of standard date (YYYYMMDD)
INTEGER-OF-DAY	I1	I	Integer date equivalent of Julian date (YYYYDDD)
INTEGER-PART	Num1	I	Integer part of Num1
LENGTH	A1 or Ind1 or Nat1 or	I	Length of argument in characters

	Num1 or X1 or T1		
LOG	Num1	Num	Natural logarithm of Num1
LOG10	Num1	Num	Logarithm to base 10 of Num1
LOWER-CASE	A1 or Nat1 or X1	Nat X	All letters in the argument are set to lowercase
MAX	A1... or I1... or Ind1or Nat1... or Num1... or X1...	X I Ind Nat Num X	Value of maximum argument
MEAN	Num1...	Num	Arithmetic mean of arguments
MEDIAN	Num1...	Num	Median of arguments
MIDRANGE	Num1...	Num	Mean of minimum and maximum arguments
MIN	A1... or I1... or Ind1 or Nat1... or Num1... or X1...	X I Ind Nat Num X	Value of minimum argument
MOD	I1, I2	I	I1 modulo I2
NATIONAL-OF	X1, Nat2	Nat	National display of the alphanumeric character set X1
NUMVAL	Nat1 or X1	Num	Numeric value of simple numeric string
NUMVAL-C	Nat1 or Nat2 or X1, X2	Num	Numeric value of numeric string with optional commas and currency sign
ORD	A1 or Nat1 or X1	I	Ordinal position of A1 / X1 in collating sequence
ORD-MAX	A1... or Ind1 or Nat1... or Num1... or X1...	I	Ordinal position of maximum argument
ORD-MIN		I	Ordinal position of minimum argument

	A1... or Ind1 or Nat1... or Num1... or X1...		
PRESENT-VALUE	Num1 Num2...	Num	Principal amount repaid by a series of deferred payments, Num2, at an interest rate of Num1
RANDOM	I1	Num	Random number
RANGE	I1... or Num1...	I Num	Value of maximum argument minus value of minimum argument
REM	Num1, Num2	Num	Remainder of Num1/Num2
REVERSE	A1 or Nat1 or X1	Nat X	Reverse order of the characters of the argument
SIN	Num1	Num	Sine of Num1
SQRT	Num1	Num	Square root of Num1
STANDARD-DEVIATION	Num1...	Num	Standard deviation of arguments
SUM	I1... or Num1...	I Num	Sum of arguments
TAN	Num1	Num	Tangent of Num1
UPPER-CASE	A1 or Nat1 or X1	Nat X	All letters in the argument are set to uppercase
VARIANCE	Num1...	Num	Variance of arguments
WHEN-COMPILED	None	X	Date and time program was compiled
YEAR-TO-YYYY	I1, I2	I	4-digit equivalent, depending on the value of I2, of the 2-digit year I1

9.2.1 ACOS - Arccosine

The ACOS function returns a numeric value in radians that approximates the arccosine of argument-1.
The type of this function is numeric.

Format

```
FUNCTION ACOS (argument-1)
```

Arguments

1. argument-1 must be class "numeric".
2. The value of argument-1 must be greater than or equal to -1 and less than or equal to +1.

Returned values

1. The returned value is the approximation of the arc cosine of argument-1 and is greater than or equal to zero and less than or equal to pi.
2. The error default value is -2.

See also: COS, SIN, ASIN, TAN, ATAN

Example 9-1

```
...
DATA DIVISION.
WORKING-STORAGE SECTION.
01  AS PIC S9V999 VALUE -0.25.
01  R PIC 9V99.
01  RES PIC +9.99.
PROCEDURE DIVISION.
P1 SECTION.
MAIN.
    COMPUTE R = FUNCTION ACOS (AS).
    MOVE R TO RES.
    DISPLAY RES UPON T.
    STOP RUN.
```

Result: +1.82

9.2.2 ADDR - Address of an identifier

The ADDR function returns an integer representing the address of argument-1.
The type of this function is integer.

Format

```
FUNCTION ADDR (argument-1)
```

Argument

1. argument-1 can be a literal or a data item of any class or category (except for the classes object, pointer and typed).

Returned value

1. The return value is an integer representing the address of argument-1 at run time.

9.2.3 ANNUITY - Annuity

The ANNUITY function returns a numeric value that approximates the ratio of an annuity paid at the end of each period for the number of periods specified by argument-2 to an initial investment of one. Interest is earned at the rate specified by argument-1 and is applied at the end of the period, before the payment.
The type of this function is numeric.

Format

```
FUNCTION ANNUITY (argument-1 argument-2)
```

Arguments

1. argument-1 must be class "numeric".
2. The value of argument-1 must be greater than or equal to zero.
3. argument-2 must be a positive integer.

Returned values

1. When the value of argument-1 is equal to zero, the value of the function is the approximation of:
   1 / argument-2
2. When the value of argument-1 is not equal to zero, the value of the function is the approximation of:
   argument-1 / (1 - (1 + argument-1)**(- argument-2))
3. The error default value is -2.

See also: PRESENT-VALUE

Example 9-2

The following program calculates the annual payments for a loan of 100000 at three different interest rates over a period of 1 to 10 years.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. INTEREST-TABLE.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
    TERMINAL IS WINDOW
    DECIMAL-POINT IS COMMA.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 CAPITAL            PIC 9(9).
01 PD                 PIC 99.
01 CALC-TABLE.
    02 INTEREST PIC V9(7) OCCURS 3 INDEXED BY R-IND-S.
 01 HEADER-LINE.
    02 PIC XX VALUE SPACE.
    02 OCCURS 3 INDEXED BY T-IND-S.
     10 INTR-ED PIC BBBZZ9,999999B.
     10 PIC X VALUE FROM (1) "%" REPEATED TO END.
 01 OUTPUT-TABLE.
    02 THIS-LINE OCCURS 10 INDEXED BY A-IND-Z.
     10 PERIOD  PIC Z9.
```

```
     10 RATE PIC BZZZBZZZBZZ9,99 OCCURS 3 INDEXED BY A-IND-S.
 PROCEDURE DIVISION.
 ONLY SECTION.
 PARA.
    MOVE 100000 TO CAPITAL
*** Interest 5,75 % ***
    MOVE 0,0575 TO INTEREST (1)
*** Interest 8,90 % ***
    MOVE 0,0890 TO INTEREST (2)
*** Interest 12,10 % ***
    MOVE 0,1210 TO INTEREST (3)
    PERFORM VARYING R-IND-S FROM 1 BY 1 UNTIL R-IND-S > 3
      SET T-IND-S TO R-IND-S
      MULTIPLY INTEREST (R-IND-S) BY 100 GIVING INTR-ED (T-IND-S)
    END-PERFORM
    PERFORM VARYING A-IND-Z FROM 1 BY 1 UNTIL A-IND-Z > 10
      PERFORM VARYING A-IND-S FROM 1 BY 1 UNTIL A-IND-S > 3
         SET R-IND-S TO A-IND-S
         SET PD TO A-IND-Z
         MOVE PD TO PERIOD (A-IND-Z)
         COMPUTE RATE (A-IND-Z  A-IND-S) = CAPITAL *
            FUNCTION ANNUITY (INTEREST (R-IND-S) PD)
      END-PERFORM
    END-PERFORM
    DISPLAY HEADER-LINE UPON WINDOW
    PERFORM VARYING A-IND-Z FROM 1 BY 1 UNTIL A-IND-Z > 10
      DISPLAY THIS-LINE (A-IND-Z) UPON WINDOW
    END-PERFORM
    STOP RUN.
```

Result:

```
    5,750000 %  8,900000 %  12,100000 %

  1  105 750,00  108 900,00   112 100,00

  2   54 352,67   56 769,79    59 247,57

  3   37 238,06   39 435,08   41 706,466

  4   28 694,12   30 799,14    32 992,81

  5   23 578,41   25 642,57    27 809,72

  6   20 176,80   22 225,55    24 391,49

  7   17 754,64   19 802,43    21 981,30

  8   15 944,62   18 000,34    20 200,66

  9   14 542,66   16 612,13    18 839,28

 10   13 426,32   15 513,49    17 770,92
```

9.2.4 ASIN - Arcsine

The ASIN function returns a numeric value in radians that approximates the arcsine of argument-1.
The type of this function is numeric.

Format

```
FUNCTION ASIN (argument-1)
```

Arguments

1. argument-1 must be class "numeric".
2. The value of argument-1 must be greater than or equal to -1 and less than or equal to +1.

Returned values

1. The returned value is the approximation of the arcsine of argument-1 and is greater than or equal to -pi/2 and less than or equal to +pi/2.
2. The error default value is -2.

See also: SIN, COS, ACOS, TAN, ATAN

Example 9-3

```
...
DATA DIVISION.
WORKING-STORAGE SECTION.
01  AN PIC S9V999 VALUE -0.45.
01  R PIC 9V99.
01  RES PIC -9.99.
PROCEDURE DIVISION.
P1 SECTION.
MAIN.
    COMPUTE R = FUNCTION ASIN (AN).
    MOVE R TO RES.
    DISPLAY RES UPON T.
    STOP RUN.
```

Result: 0.46

9.2.5 ATAN - Arctangent

The ATAN function returns a numeric value in radians that approximates the arctangent of argument-1.
The type of this function is numeric.

Format

```
FUNCTION ATAN (argument-1)
```

Argument

1. argument-1 must be class "numeric".

Returned value

1. The returned value is the approximation of the arctangent of argument-1 and is greater than -pi/2 and less than +pi/2.

See also: TAN, SIN, ASIN, COS, ACOS

Example 9-4

```
...
DATA DIVISION.
WORKING-STORAGE SECTION.
01  R PIC 9V9999.
01  RES PIC -9.9999.
PROCEDURE DIVISION.
P1 SECTION.
MAIN.
    COMPUTE R = FUNCTION ATAN (-0.45).
    MOVE R TO RES.
    DISPLAY RES UPON T.
    STOP RUN.
```

Result: 0.4228

9.2.6 BYTE-LENGTH - Number of bytes

The CHAR function returns an integer value that is the length in bytes of argument-1.
The type of this function is integer.

Format

```
FUNCTION BYTE-LENGTH (argument-1)
```

Arguments

1. argument-1 must be an alphanumeric or national literal or a data item of any class or category or type name.
2. If any data item subordinate to argument-1 is described with the DEPENDING phrase of the OCCURS clause, the content of the DEPENDING data item at the time the LENGTH function was analyzed is used.

Returned values

1. If argument-1 is a non-numeric literal, an elementary item or a group item which contains no variable-length data item, the return value is the length of argument-1 in bytes.

   > ¡ If argument-1 is an object reference then the length of the object reference and **not** the size of the object itself is returned.

2. If argument-1 is a group item to which a variable-length data item is subordinated, the return value is the length of argument-1 in bytes. It is calculated in accordance with the rules for a sending item with OCCURS clause.
3. The return value takes into account implicit FILLER characters which occur in argument-1.

See also: LENGTH

Example 9-5

```
...
DATA DIVISION.
WORKING-STORAGE SECTION.
01 RES PIC 9(3).
PROCEDURE DIVISION.
MAIN.
     COMPUTE RES = FUNCTION LENGTH (N"Rats live on no evil star").
     DISPLAY RES UPON T.
     STOP RUN.
```

Result: 050

9.2.7 CHAR - Character in the alphanumeric collating sequence

The CHAR function returns a one-character alphanumeric value that is a character in the alphanumeric collating sequence having the ordinal position equal to the value of argument-1.
The type of this function is alphanumeric.

Format

```
FUNCTION CHAR (argument-1)
```

Arguments

1. argument-1 must be an integer.
2. The value of argument-1 must be greater than zero and less than or equal to the number of positions in the collating sequence.

Returned values

1. If more than one character has the same position in the program collating sequence, the character returned as the function value is that of the first literal specified for that character position in the ALPHABET clause.
2. The error default value is a space.

See also: ORD

Example 9-6

```
...
DATA DIVISION.
WORKING-STORAGE SECTION.
01  LETTER PIC 999 VALUE 194.
01  R PIC X.
PROCEDURE DIVISION.
P1 SECTION.
MAIN.
    MOVE FUNCTION CHAR (LETTER) TO R.
    DISPLAY R UPON T.
    STOP RUN.
```

Result: A
The ordinal position of "A" in EBCDIC is 194.

9.2.8 CHAR-NATIONAL - Character in the national collating sequence

The CHAR-NATIONAL function returns the character whose ordinal position is determined within the national collating sequence using argument-1.
The type of this function is national.

Format

```
FUNCTION CHAR-NATIONAL (argument-1)
```

Arguments

1. argument-1 must be an integer.
2. The value of argument-1 must be greater than zero and less than or equal to 65536.

Returned values

1. The return value supplied is the character whose ordinal position is determined in UTF-16 with argument-1.
2. The error default value is a space.

See also: ORD

Example 9-7

```
...
DATA DIVISION.
WORKING-STORAGE SECTION.
01  LETTER PIC 999 VALUE 64.
01  R PIC N.
PROCEDURE DIVISION.
MAIN.
    MOVE FUNCTION CHAR-NATIONAL (LETTER) TO R.
    STOP RUN.
```

Result: R contains the character N“@" in UTF-16 representation.

9.2.9 COS - Cosine

The COS function returns the cosine of the angle or arc that is specified in radians by argument-1.
The type of this function is numeric.

Format

```
FUNCTION COS (argument-1)
```

Argument

1. argument-1 must be class "numeric".

Returned value

1. The returned value is the approximation of the cosine of argument-1 and is greater than or equal to -1 and less than or equal to +1.

See also: ACOS, SIN, ASIN, TAN, ATAN

Example 9-8

```
...
DATA DIVISION.
WORKING-STORAGE SECTION.
01  R PIC S9V9(10).
01  RES PIC -9.9(10).
PROCEDURE DIVISION.
P1 SECTION.
MAIN.
    COMPUTE R = FUNCTION COS (3.1425).
    MOVE R TO RES.
    DISPLAY RES UPON T.
    STOP RUN.
```

Result: -0.9999995883

9.2.10 CURRENT-DATE - Current date

The CURRENT-DATE function returns a 21-character alphanumeric value that represents the calendar date and time of day.
The type of this function is alphanumeric.

Format

```
FUNCTION CURRENT-DATE
```

Returned value

1. The character positions returned, numbered from left to right, are:

Character position	Contents
1-4	Four numeric digits of the year (Gregorian calendar).
5-6	Two numeric digits of the month of the year, in the range 01 through 12.
7-8	Two numeric digits of the day of the month, in the range 01 through 31.
9-10	Two numeric digits of the hours past midnight, in the range 00 through 23.
11-12	Two numeric digits of the minutes past the hour, in the range 00 through 59.
13-14	Two numeric digits of the seconds past the minute, in the range 00 through 59.
15-21	0000000

See also: DATE-OF-INTEGER, DAY-OF-INTEGER, INTEGER-OF-DATE, INTEGER-OF-DAY, WHEN-COMPILED

Example 9-9

```
...
DATA DIVISION.
WORKING-STORAGE SECTION.
01  A-DATE PIC XXXX/XX/XXBBXXBXXBXXBXXBBX(5).
PROCEDURE DIVISION.
P1 SECTION.
MAIN.
    MOVE FUNCTION CURRENT-DATE TO A-DATE.
    DISPLAY A-DATE UPON T.
```

Result: 1995/08/10 14 36 19 00 00000

9.2.11 DATE-OF-INTEGER - Date conversion

The DATE-OF-INTEGER function converts a date in the Gregorian calendar from integer date form to standard date form (YYYYMMDD).
The type of this function is integer.

Format

```
FUNCTION DATE-OF-INTEGER (argument-1)
```

Argument

1. argument-1 is a positive integer that represents a number of days succeeding December 31, 1600, in the Gregorian calendar.

Returned values

1. The returned value represents the ISO Standard date equivalent of the integer specified in argument-1.
2. YYYY represents a year in the Gregorian calendar, MM represents the month of that year, and DD represents the day of that month.
3. The error default value is 0.

See also: DAY-OF-INTEGER, INTEGER-OF-DATE, INTEGER-OF-DAY, CURRENT-DATE, WHEN-COMPILED

Example 9-10

```
...
DATA DIVISION.
WORKING-STORAGE SECTION.
01  A-DATE PIC 9(8).
PROCEDURE DIVISION.
P1 SECTION.
MAIN.
    COMPUTE A-DATE = FUNCTION DATE-OF-INTEGER (50000).
    DISPLAY A-DATE UPON T.
    STOP RUN.
```

Result: 17371123
The 50000th day as of 31.12.1600 was 23.11.1737.

9.2.12 DATE-TO-YYYYMMDD - year conversion

The DATE-TO-YYYYMMDD function converts a date specified with argument-1 from the standard date format with a 2-digit year into the standard date format with a 4-digit year. The end of the 100-year interval in which the year specified in argument-1 falls is determined by adding argument-2 to the current year (the year in which the function is executed) ("floating window").
The type of this function is integer.

Format

```
FUNCTION DATE-TO-YYYYMMD (argument-1 [argument-2])
```

Arguments

1. argument-1 must be a positive number less than 1000000.
2. argument-2, if specified, must be an integer.
3. If argument-2 is not specified, the value 50 is taken for the second argument.
4. The sum of the current year and argument-2 must be less than 10000 and greater than 1699.
5. No check is run to verify whether argument-1 is a valid date. This means that the values 0 and 999999 are also valid arguments for the DATE-TO-YYYYMMDD function if a check of the arguments is required by one of the options
   CHECK-FUNCTION-ARGUMENTS = YES or
   SET-FUNCTION-ERROR-DEFAULT = YES.

Returned values

1. The returned value is the date specified in argument-1 with a 4-digit year. For an argument of the form YYMMDD, the returned value is defined by:
   FUNCTION YEAR-TO-YYYY (YY, argument-2) * 10000 + MMDD
2. The error default value is 0.

See also: DAY-TO-YYYYDDD, YEAR-TO-YYYY

Example 9-11

```
...
DATA DIVISION.
WORKING-STORAGE SECTION.
01  A-DATE         PIC 9(8).
PROCEDURE DIVISION.
P1 SECTION.
MAIN.
    COMPUTE A-DATE = FUNCTION DATE-TO-YYYYMMDD (590123).
    DISPLAY A-DATE UPON T.                                    (1)
    COMPUTE A-DATE = FUNCTION DATE-TO-YYYYMMDD (470101 -50).
    DISPLAY A-DATE UPON T.                                    (2)
    STOP RUN.
```

A more detailed example is provided with the description of the YEAR-TO-YYYY function.

Result:

In the year 1996 the program returns the following results:
(1) 19590123
(2) 18470101

In the year 2009 the program returns the following results:
(1) 20590123
(2) 19470101

9.2.13 DAY-OF-INTEGER - Date conversion

The DAY-OF-INTEGER function converts a date in the Gregorian calendar from integer date form to Julian date form (YYYYDDD).
The type of this function is integer.

Format

```
FUNCTION DAY-OF-INTEGER (argument-1)
```

Argument

1. argument-1 is a positive integer that represents a number of days succeeding December 31, 1600, in the Gregorian calendar.

Returned values

1. The returned value represents the Julian equivalent of the integer specified in argument-1.
2. YYYY represents a year in the Gregorian calendar, and DDD represents the day of that year.
3. The error default value is 0.

See also: DATE-OF-INTEGER, INTEGER-OF-DAY, INTEGER-OF-DATE, CURRENT-DATE, WHEN-COMPILED

Example 9-12

```
...
DATA DIVISION.
WORKING-STORAGE SECTION.
01  DAYS PIC 9999 VALUE 5000.
01  A-DATE PIC X(7).
PROCEDURE DIVISION.
P1 SECTION.
MAIN.
    COMPUTE A-DATE = FUNCTION DAY-OF-INTEGER (DAYS)
    DISPLAY A-DATE UPON T.
    STOP RUN.
```

Result: 1614252
The 5000th day as of 31.12.1600 was the 252nd day of the year 1614.

9.2.14 DAY-TO-YYYYDDD - year conversion

The DAY-TO-YYYYDDD function converts a date specified with argument-1 from the Julian date format with a 2-digit year into the Julian date format with a 4-digit year. The end of the 100-year interval in which the year specified in argument-1 falls is determined by adding argument-2 to the current year (the year in which the function is executed) ("floating window").
The type of this function is integer.

Format

```
FUNCTION DAY-TO-YYYYDDD (argument-1 [argument-2])
```

Arguments

1. argument-1 must be a positive number less than 100000.
2. argument-2, if specified, must be an integer.
3. If argument-2 is not specified, the value 50 is taken for the second argument.
4. The sum of the current year and argument-2 must be less than 10000 and greater than 1699.
5. No check is run to verify whether argument-1 is a valid date. This means that the values 0 and 99999 are also valid arguments for the DAY-TO-YYYYDDD function if a check of the arguments is required by one of the options CHECK-FUNCTION-ARGUMENTS = YES or SET-FUNCTION-ERROR-DEFAULT = YES.

Returned values

1. The returned value is the date specified in argument-1 with a 4-digit year. For an argument of the form YYDDD, the returned value is defined by:
   FUNCTION YEAR-TO-YYYY (YY, argument-2) * 1000 + DDD
2. The error default value is 0.

See also: DATE-TO-YYYYMMDD, YEAR-TO-YYYY

Example 9-13

```
...
DATA DIVISION.
WORKING-STORAGE SECTION.
01 A-DATE PIC 9(7).
PROCEDURE DIVISION.
P1 SECTION.
MAIN.
    COMPUTE A-DATE = FUNCTION DAY-TO-YYYYDDD (59001).
    DISPLAY A-DATE UPON T.                                          (1)
    COMPUTE A-DATE = FUNCTION DAY-TO-YYYYDDD (47365 -50).
    DISPLAY A-DATE UPON T.                                          (2)
    STOP RUN.
```

A more detailed example is provided with the description of the YEAR-TO-YYYY function.

Result:

In the year 1996 the program returns the following results:
(1) 1959001
(2) 1847365

In the year 2009 the program returns the following results:
(1) 2059001
(2) 1947365

9.2.15 DISPLAY-OF - alphanumeric string

The DISPLAY-OF function returns a string which contains the characters of the argument converted into alphanumeric representation.
The type of this function is alphanumeric.

Format

```
FUNCTION DISPLAY-OF (argument-1 [argument-2])
```

Arguments

1. argument-1 must be of the national class and at least one character long.
2. argument-1 may not be defined with the ANY LENGTH clause.
3. argument-2 must be of the alphabetic class and precisely one character long.
   When conversion takes place in the result, argument-2 specifies a replacement character in place of the national characters for which there are no corresponding alphanumeric characters.

Returned values

1. As a return value a string is supplied in which each national character from argument-1 is converted into its corresponding character in alphanumeric representation.
2. If argument-2 is specified, each character from argument-1 for which no corresponding character exists in alphanumeric representation is converted into the replacement character specified by argument-2.
3. If argument-2 is not specified and argument-1 contains characters for which no corresponding character exists in alphanumeric representation, these are converted into a replacement character defined by XHCS (period '.'). The exceptional condition EC-DATA-CONVERSION occurs.
4. The return value contains exactly as many characters as argument-1.
5. The error return value is a blank.

> ¡ If FUNCTION DISPLAY-OF is contained as a sending field in a MOVE statement for which the check for the exceptional condition EC-DATA-CONVERSION is enabled and a USE procedure also exists, the receiving item remains unchanged if the exceptional condition occurs.

See also: NATIONAL-OF

Example 9-14

```
...
DATA DIVISION.
WORKING-STORAGE SECTION.
01 nat PIC XXX VALUE NX"E23A005A0040".
01 var PIC X VALUE "z".
01 R PIC xxx.
PROCEDURE DIVISION.
MAIN.
     DISPLAY FUNCTION DISPLAY-OF (nat, var).
     STOP RUN.
```

Result: zZ@

9.2.16 EXCEPTION STATUS

The EXCEPTION-STATUS function supplies the name of the last exception status.
Function type: alphanumeric.

Format

```
FUNCTION EXCEPTION-STATUS
```

Return value

1. The return value is a 31-character alphanumeric string which contains the name of the last exception status left-justified. If the last exception status indicates that no exception status has been triggered, the string contains 31 blanks.

9.2.17 FACTORIAL - Factorial

The FACTORIAL function returns an integer that is the factorial of argument-1.
The type of this function is integer.

Format

```
FUNCTION FACTORIAL (argument-1)
```

Argument

1. argument-1 must be an integer greater than or equal to zero and less than or equal to 29.

Returned values

1. If the value of argument-1 is zero, the value 1 is returned.
2. If the value of argument-1 is positive, its factorial is returned.
3. The error default value is -2.

See also: LOG, LOG10, SQRT

Example 9-15

```
...
DATA DIVISION.
WORKING-STORAGE SECTION.
01  RES PIC 9(8).
PROCEDURE DIVISION.
P1 SECTION.
MAIN.
    COMPUTE RES = FUNCTION FACTORIAL (07).
    DISPLAY RES UPON T.
    STOP RUN.
```

Result: 00005040

9.2.18 INTEGER - Next smaller integer

The INTEGER function returns the greatest integer that is less than or equal to the value of argument-1.
The type of this function is integer.

Format

```
FUNCTION INTEGER (argument-1)
```

Argument

1. argument-1 must be class "numeric" and must be greater than or equal to $-10^{31}+1$ and less than 10^{31}.

Returned values

1. The returned value is the greatest integer less than or equal to the value of argument-1. For example, if the value of argument-1 is -1.5, -2 is returned. If the value of argument-1 is +1.5, +1 is returned.
2. The error default value is -9’999’999’999’999’999’999’999’999’999’999.

See also: INTEGER-PART

Example 9-16

```
...
DATA DIVISION.
WORKING-STORAGE SECTION.
01  RES PIC 9(8).
PROCEDURE DIVISION.
P1 SECTION.
MAIN.
    COMPUTE RES = FUNCTION INTEGER (-3.3).
    DISPLAY RES UPON T.
    STOP RUN.
```

Result: 00000004

9.2.19 INTEGER-OF-DATE - Date conversion

The INTEGER-OF-DATE function converts a date in the Gregorian calendar from standard date form (YYYYMMDD) to integer date form.
This type of this function is integer.

Format

```
FUNCTION INTEGER-OF-DATE (argument-1)
```

Argument

1. argument-1 must be an integer of the form YYYYMMDD, whose value is obtained from the calculation: (YYYY * 10000) + (MM * 100) + DD
   - YYYY represents the year in the Gregorian calendar. It must be an integer greater than 1600.
   - MM represents a month and must be a positive integer less than 13.
   - DD represents a day and must be a positive integer less than 32 provided that it is valid for the specified month and year combination.

Returned values

1. The returned value is an integer that is the number of days the date represented by argument-1 succeeds December 31, 1600, (on sunday) in the Gregorian calendar.
2. The error default value is 0.

See also: INTEGER-OF-DAY, DATE-OF-INTEGER, DAY-OF-INTEGER, CURRENT-DATE, WHEN-COMPILED

Example 9-17

```
...
DATA DIVISION.
WORKING-STORAGE SECTION.
01  DAYS PIC 9(8).
PROCEDURE DIVISION.
P1 SECTION.
MAIN.
    COMPUTE DAYS = FUNCTION INTEGER-OF-DATE (19530410).
    DISPLAY DAYS UPON T.
    STOP RUN.
```

Result: 00128665
10.4.1953 was the 128665th day as of 31.12.1600.

9.2.20 INTEGER-OF-DAY - Date conversion

The INTEGER-OF-DAY function converts a date in the Gregorian calendar from Julian date form (YYYYDDD) to integer date form.
The type of this function is integer.

Format

```
FUNCTION INTEGER-OF-DAY (argument-1)
```

Argument

1. argument-1 must be an integer of the form YYYYDDD, whose value is obtained from the calculation: (YYYY * 1000) + DDD
   - YYYY represents the year in the Gregorian calendar. It must be an integer greater than 1600.
   - DDD represents the day of the year. It must be a positive integer less than 367 but 366 may only be specified for a leap year.

Returned values

1. The returned value is an integer that is the number of days the date represented by argument-1 succeeds December 31, 1600, in the Gregorian calendar.
2. The error default value is 0.

See also: INTEGER-OF-DATE, DAY-OF-INTEGER, DATE-OF-INTEGER, CURRENT-DATE, WHEN-COMPILED

Example 9-18

```
...
DATA DIVISION.
WORKING-STORAGE SECTION.
01  DAYS PIC 9(7).
PROCEDURE DIVISION.
P1 SECTION.
MAIN.
    COMPUTE DAYS = FUNCTION INTEGER-OF-DAY (1993299).
    DISPLAY DAYS UPON T.
    STOP RUN.
```

Result: 0143474
The 299th day of the year 1993 is the 143474th day as of 31.12.1600.

9.2.21 INTEGER-PART - Integer part of a floating-point value

The INTEGER-PART function returns an integer that is the integer portion of argument-1.
The type of this function is integer.

Format

```
FUNCTION INTEGER-PART (argument-1)
```

Argument

1. argument-1 must be of class "numeric" and the value v of argument-1 must lie in the range $-10^{31} < v < 10^{31}$

Returned values

1. If the value of argument-1 is zero, the returned value is zero.
2. If the value of argument-1 is positive, the returned value is the greatest integer less than or equal to the value of argument-1. For example, if the value of argument-1 is +1.5, +1 is returned.
3. If the value of argument-1 is negative, the returned value is the least integer greater than or equal to the value of argument-1. For example, if the value of argument-1 is -1.5, -1 is returned.
4. The error default value is - 9’999’999’999’999’999’999’999’999’999’999.

See also: INTEGER

Example 9-19

```
...
DATA DIVISION.
WORKING-STORAGE SECTION.
01  RES PIC 9(8).
PROCEDURE DIVISION.
P1 SECTION.
MAIN.
    COMPUTE RES = FUNCTION INTEGER-PART (-3.3).
    DISPLAY RES UPON T.
    STOP RUN.
```

Result: -00000003

9.2.22 LENGTH - Number of characters

The LENGTH function returns an integer equal to the length of argument-1 in character positions.
The type of this function is integer.

Format

```
FUNCTION LENGTH (argument-1)
```

Arguments

1. argument-1 may be a non-numeric literal or a data item of any class or category or a type name.
2. If any data item subordinate to argument-1 is described with the DEPENDING phrase of the OCCURS clause, the contents of the DEPENDING data item are used at the time the LENGTH function is evaluated.

Returned values

1. If argument-1 is a non-numeric literal or an elementary data item or a group data item that does not contain a variable occurrence data item, the value returned is an integer equal to the length of argument-1 in character positions.

   > If argument-1 is an object reference then the length of the object reference and **not** the size of the object itself is returned.

2. If argument-1 is a group item to which a variable-length data item is subordinated, the return value is the length of argument-1 in bytes. It is calculated in accordance with the rules for a sending item with OCCURS clause.
3. The return value takes into account any implicit FILLER characters which occur in argument-1.

Example 9-20

```
...
DATA DIVISION.
WORKING-STORAGE SECTION.
01  RES PIC 9(3).
PROCEDURE DIVISION.
MAIN.
    COMPUTE RES = FUNCTION LENGTH ("Rats live on no evil star").
    DISPLAY RES UPON T.
    STOP RUN.
```

Result: 025

9.2.23 LOG - Logarithm

The LOG function returns a numeric value that approximates the logarithm to the base e (natural log) of argument-1. The type of this function is numeric.

Format

```
FUNCTION LOG (argument-1)
```

Arguments

1. argument-1 must be class "numeric".
2. The value of argument-1 must be greater than zero.

Returned values

1. The returned value is the approximation of the logarithm to the base e of argument-1.
2. The error default value is - 9’999’999’999’999’999’999’999’999’999’999.

See also: LOG10, FACTORIAL, SQRT

Example 9-21

```
...
DATA DIVISION.
WORKING-STORAGE SECTION.
01  L PIC S99V999 VALUE 3.
01  R PIC S99V999999.
01  RES PIC 99.999999.
PROCEDURE DIVISION.
P1 SECTION.
MAIN.
    COMPUTE R = FUNCTION LOG (L).
    MOVE R TO RES.
    DISPLAY RES UPON T.
    STOP RUN.
```

Result: 01.098612

9.2.24 LOG10 - Logarithm of base 10

The LOG10 function returns a numeric value that approximates the logarithm to the base 10 of argument-1.
The type of this function is numeric.

Format

```
FUNCTION LOG10 (argument-1)
```

Arguments

1. argument-1 must be class "numeric".
2. The value of argument-1 must be greater than zero.

Returned values

1. The returned value is the approximation of the logarithm to the base 10 of argument-1.
2. The error default value is - 9’999’999’999’999’999’999’999’999’999’999.

See also: LOG, FACTORIAL, SQRT

Example 9-22

```
...
DATA DIVISION.
WORKING-STORAGE SECTION.
01  L PIC S99V999 VALUE 3.
01  R PIC S99V999999.
01  RES PIC 99.999999.
PROCEDURE DIVISION.
P1 SECTION.
MAIN.
    COMPUTE R = FUNCTION LOG10 (L).
    MOVE R TO RES.
    DISPLAY RES UPON T.
    STOP RUN.
```

Result: 00.477121

9.2.25 LOWER-CASE - Lowercase letters

The LOWER-CASE function returns a character string that is the same length as argument-1 with each uppercase letter replaced by a corresponding lowercase letter.
The type of this function depends upon the argument types as follows:

Argument type	Function type
alphabetic alphanumeric national	alphanumeric alphanumeric national

Format

```
FUNCTION LOWER-CASE (argument-1)
```

Arguments

1. argument-1 must be class alphabetic, alphanumeric or national and must be at least one character in length.
2. argument-1 must not be defined in the ANY LENGTH clause.

Returned values

1. The returned value is the same character string as argument-1, except that each uppercase letter is replaced by the corresponding lowercase letter.
2. The character string returned has the same length as argument-1.
3. Umlauts are not converted.
4. The error default value is a space.

See also: UPPER-CASE

Example 9-23

```
...
DATA DIVISION.
WORKING-STORAGE SECTION.
01  RES PIC X(20).
PROCEDURE DIVISION.
P1 SECTION.
MAIN.
    MOVE FUNCTION LOWER-CASE ("UPPER lower") TO RES.
    DISPLAY RES UPON T.
    STOP RUN.
```

Result: upper lower

9.2.26 MAX - Value of maximum argument

The MAX function returns the content of the argument that contains the maximum value.
The type of this function depends upon the argument types as follows:

Argument type	Function type
alphabetic	alphanumeric
alphanumeric	alphanumeric
all arguments integer	integer
index	index
national	national
numeric	numeric
numeric/integer	numeric

Format

```
FUNCTION MAX ({argument-1}...)
```

Arguments

1. If more than one argument is specified, all arguments must be of the same class.
2. The individual arguments must not be specified in the ANY LENGTH clause.

Returned values

1. The returned value is the content of the argument having the greatest value. The comparisons used to determine the greatest value are made according to the rules for simple conditions.
2. If more than one argument has the same greatest value, the content of the argument returned is the leftmost argument having that value.
3. If the type of the function is alphanumeric or national, the size of the returned value is the same as the size of the selected argument.
4. The error default value is 0.

See also: MIN, ORD-MAX, ORD-MIN, RANGE, MEAN, MEDIAN, MIDRANGE, SUM

Example 9-24

```
...
DATA DIVISION.
WORKING-STORAGE SECTION.
01  RES PIC 9(3).
01  RES1 PIC X(4).
PROCEDURE DIVISION.
P1 SECTION.
MAIN.
    COMPUTE RES = FUNCTION MAX (12 32 5 8 17 9).
    MOVE FUNCTION MAX ("HUGO" "EGON" "THEO" "OTTO") TO RES1.
    DISPLAY "Argument with greatest value: " RES UPON T.
    DISPLAY "Argument with greatest value: " RES1 UPON T.
    STOP RUN.
```

Result: Argument with greatest value RES: 032
Argument with greatest value RES1: THEO

9.2.27 MEAN - Arithmetic mean of arguments

The MEAN function returns a numeric value that is the arithmetic mean (average) of its arguments.
The type of this function is numeric.

Format

```
FUNCTION MEAN ({argument-1}...)
```

Argument

1. argument-1 must be class "numeric".

Returned values

1. The returned value is the arithmetic mean of the argument series.
2. The returned value is defined as the sum of the argument series divided by the number of arguments specified.
3. The error default value is 0.

See also: MIN, MAX, RANGE, MEDIAN, MIDRANGE, SUM

Example 9-25

```
...
DATA DIVISION.
WORKING-STORAGE SECTION.
01  R PIC 999V999.
01  RES PIC 999.999.
PROCEDURE DIVISION.
P1 SECTION.
MAIN.
    COMPUTE R = FUNCTION MEAN (12 32 5 8 17 9).
    MOVE R TO RES.
    DISPLAY RES UPON T.
    STOP RUN.
```

Result: 013.833

9.2.28 MEDIAN - Median of arguments

The MEDIAN function returns the contents of the argument whose value is the middle value in the list formed by arranging the arguments in sorted order.
The type of this function is numeric.

Format

```
FUNCTION MEDIAN ({argument-1}...)
```

Argument

1. argument-1 must be class "numeric".

Returned values

1. The returned value is the content of the argument having the middle value in the list formed by arranging all the argument values in sorted order.
2. If the number of occurrences referenced by argument is odd, the returned value is such that at least half of the occurrences referenced by argument are greater than or equal to the returned value and at least half are less than or equal. If the number of occurrences referenced by argument is even, the returned value is the arithmetic mean of the values referenced by the two middle occurrences.
3. The comparisons used to arrange the argument values in sorted order are made according to the rules for simple conditions (see section "Conditions").
4. The error default value is 0.

See also: MIN, MAX, RANGE, MEAN, MIDRANGE, SUM

Example 9-26

```
...
DATA DIVISION.
WORKING-STORAGE SECTION.
01  RES PIC 9(3).
PROCEDURE DIVISION.
P1 SECTION.
MAIN.
    COMPUTE RES = FUNCTION MEDIAN (2 32 8 128 16 64 256).
    DISPLAY RES UPON T.
    STOP RUN.
```

Result: 032

9.2.29 MIDRANGE - Mean of minimum and maximum arguments

The MIDRANGE function returns a numeric value that is the arithmetic mean (average) of the values of the minimum argument and the maximum argument.
The type of this function is numeric.

Format

```
FUNCTION MIDRANGE ({argument-1}...)
```

Argument

1. argument-1 must be class "numeric".

Returned values

1. The returned value is the arithmetic mean of the greatest argument value and the least argument value. The comparisons used to determine the greatest and least values are made according to the rules for simple conditions (see section "Conditions").
2. The error default value is 0.

See also: MIN, MAX, RANGE, MEAN, MEDIAN, SUM

Example 9-27

```
...
DATA DIVISION.
WORKING-STORAGE SECTION.
01  R PIC 999V999.
01  RES PIC 999.999.
PROCEDURE DIVISION.
P1 SECTION.
MAIN.
    COMPUTE R = FUNCTION MIDRANGE (12 32 5 8 17 9).
    MOVE R TO RES.
    DISPLAY RES UPON T.
    STOP RUN.
```

Result: 018.500

9.2.30 MIN - Value of minimum argument

The MIN function returns the content of the argument that contains the minimum value.The type of this function depends upon the argument types as follows:

Argument type	Function type
alphabetic	alphanumeric
alphanumeric	alphanumeric
all arguments integer	integer
index	index
national	national
numeric	numeric
numeric/integer	numeric

Format

```
FUNCTION MIN ({argument-1}...)
```

Arguments

1. If more than one argument is specified, all arguments must be of the same class.
2. The individual values must not be defined with the ANY LENGTH clause.

Returned values

1. The returned value is the content of the argument having the least value. The comparisons used to determine the least value are made according to the rules for simple conditions.
2. If more than one argument has the same least value, the content of the argument returned is the leftmost argument having that value.
3. If the type of the function is alphanumeric or national, the size of the returned value is the same as the size of the selected argument.
4. The error default value is 0.

See also: MAX, RANGE, MEAN, MEDIAN, MIDRANGE, SUM

Example 9-28

```
...
DATA DIVISION.
WORKING-STORAGE SECTION.
01  RES PIC 9(3).
PROCEDURE DIVISION.
P1 SECTION.
MAIN.
    COMPUTE RES = FUNCTION MIN (12 32 5 8 17 9).
    DISPLAY RES UPON T.
    STOP RUN.
```

Result: 005

9.2.31 MOD - Modulo

The MOD function returns an integer value that is argument-1 modulo argument-2.The type of this function is integer.

Format

```
FUNCTION MOD (argument-1 argument-2)
```

Arguments

1. argument-1 and argument-2 must be integers.
2. The value v of argument-1 must lie in the range $-10^{31} < v < 10^{31}$

   If the value of argument-1 is outside this range, the function value can be calculated but is not guaranteed to be correct.
3. The value of argument-2 must not be 0.

Returned values

1. The returned value is argument-1 modulo argument-2. The returned value is defined as:
   argument-1 - (argument-2 * FUNCTION INTEGER (argument-1 / argument-2))
2. The error default value is -9’999’999’999’999’999’999’999’999’999’999.

See also: REM

Example 9-29

argument-1	argument-2	Returned value
11	5	1
-11	5	4
11	-5	-4
-11	-5	-1

9.2.32 NATIONAL-OF - national character representation

The DISPLAY-OF function returns a string which contains the characters of the argument converted into alphanumeric representation.
The type of this function is national.

Format

```
FUNCTION NATIONAL-OF (argument-1 [argument-2])
```

Arguments

1. argument-1 must be of the class alphabetic or alphanumeric and at least one character long.
2. argument-1 may not be defined with the ANY LENGTH clause.
3. argument-2 must be of the class national and precisely one character long.
   When conversion takes place in the result, argument-2 specifies a replacement character in place of the alphanumeric characters for which there is no corresponding national character.

Returned values

1. As a return value a string is supplied in which each alphanumeric character from argument-1 is converted into its corresponding character in national representation.
2. If argument-2 is specified, each character from argument-1 for which no corresponding character exists in national representation is converted into the replacement character specified by argument-2.
3. If argument-2 is not specified and argument-1 contains characters for which no corresponding character exists in national representation, these are converted into a replacement character defined by XHCS (period '.'). The exceptional condition EC-DATA-CONVERSION occurs.
4. The return value contains exactly as many characters as argument-1.
5. The error return value is a blank.

> i If FUNCTION NATIONAL-OF is contained as a sending field in a MOVE statement for which the check for the exceptional condition EC-DATA-CONVERSION is enabled and a USE procedure also exists, the receiving item remains unchanged if the exceptional condition occurs. A MOVE statement with implicit conversion is the opposite of this.

See also: DISPLAY-OF

Example 9-30

```
...
DATA DIVISION.
WORKING-STORAGE SECTION.
01 alfa PIC XXX VALUE "ABC".
01 R PIC NNN.
PROCEDURE DIVISION.
MAIN.
    MOVE FUNCTION NATIONAL-OF (alfa) TO R.
    STOP RUN.
```

Result: R contains the characters ABC in UTF-16 representation.

9.2.33 NUMVAL - Numeric value of string

The NUMVAL function returns the numeric value represented by the character string specified by argument-1.
Leading and trailing spaces are ignored.
The type of this function is numeric.

Format

```
FUNCTION NUMVAL (argument-1)
```

Arguments

1. argument-1 must be from the category alphanumeric or national and must have one of the following two formats:

   ```
   ['BLANK'] [+ | -] ['BLANK'] {digit[.[digit]] | .digit} ['BLANK']
   or
   ['BLANK'] {digit[.[digit]] | .digit} ['BLANK'] [+ | - | CR | DB]
   ```

'BLANK'	String of one or more spaces
digit	One or more digits of the set {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

2. The letter string "CR" or "DB" in upper case must be specified for CR and DB.
3. The total number of digits in argument-1 must not exceed 31.
4. If the DECIMAL-POINT IS COMMA clause is specified in the SPECIAL-NAMES paragraph, a comma must be used in argument-1 rather than a decimal point.

Returned values

1. The returned value is the numeric value represented by argument-1.
2. If argument-1 has a variable length or a length with more than 16 characters (or more than 14 characters if the error return value is required), the result can deviate from the exact value, since a floating point representation is required for the return value.
3. If argument-1 has a fixed length of up to 14 characters then the error default value is -999’999’999’999’999’999. If argument-1 has a variable length or a length of more than 14 characters then the error default value is -9’999’999’999’999’999’999’999’999’999.

See also: NUMVAL-C

Example 9-31

```
...
DATA DIVISION.
WORKING-STORAGE SECTION.
01  V PIC X(8) VALUE "+ 15.00".
01  R PIC 99V99.
01  RES PIC 99.99.
PROCEDURE DIVISION.
P1 SECTION.
MAIN.
    COMPUTE R = FUNCTION NUMVAL (V).
```

```
    MOVE R TO RES.
    DISPLAY RES UPON T.
    STOP RUN.
```

Result: 15.00

9.2.34 NUMVAL-C - Numeric value of string with optional currency sign

The NUMVAL-C function returns the numeric value represented by the character string specified by argument-1. Any optional currency sign specified by argument-2 and any optional commas preceding the decimal point are ignored.
The type of this function is numeric.

Format

```
FUNCTION NUMVAL-C (argument-1 [argument-2])
```

Arguments

1. argument-1 must be from the category alphanumeric or national and must have one of the following two formats:

   ```
   ['BLANK'] [+ | -] ['BLANK'] [wz] ['BLANK'] {digit[,digit]...[.[digit]] | .digit}
   ['BLANK']
   or
   ['BLANK'] [wz] ['BLANK'] {digit[,digit]...[.[digit]] | .digit} ['BLANK'] [+ | - |
   CR | DB ] ['BLANK']
   ```

'BLANK'	String of one or more spaces
wz	Currency symbol: string of one or more characters (argument-2)
digit	One or more digits of the set {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

2. The letter string "CR" or "DB" in upper case must be specified for CR and DB.
3. If the DECIMAL-POINT IS COMMA clause is specified in the SPECIAL-NAMES paragraph, the functions of the comma and decimal point in argument-1 are reversed.
4. The total number of digits in argument-1 must not exceed 31.
5. argument-2, if specified, must be of the same class as argument-1.
6. If argument-2 is not specified, the character used for cs is the currency symbol specified for the program.

Returned values

1. The returned value is the numeric value represented by argument-1.
2. If argument-1 has a variable length or a length with more than 16 characters (or more than 14 characters if the error return value is required), the result can deviate from the exact value, since a floating point representation is required for the return value.
3. If argument-1 has a fixed length of up to 14 characters then the error default value is -999’999’999’999’999’999. If argument-1 has a variable length or a length of more than 14 characters then the error default value is -9’999’999’999’999’999’999’999’999’999’999.

See also: NUMVAL

Example 9-32

```
...
DATA DIVISION.
WORKING-STORAGE SECTION.
01  V PIC X(8) VALUE "- $15.00".
01  R PIC 99V99.
01  RES PIC 99.99.
PROCEDURE DIVISION.
P1 SECTION.
MAIN.
    COMPUTE RES = FUNCTION NUMVAL-C (V).
    DISPLAY RES UPON T.
    STOP RUN.
```

Result: 15.00

9.2.35 ORD - Ordinal position in collating sequence

The ORD function returns an integer value that is the ordinal position of argument-1 in the collating sequence for the program. The lowest ordinal position is 1.
The type of this function is integer.

Format

```
FUNCTION ORD (argument-1)
```

Argument

1. argument-1 must be one character in length and must be category alphabetic, alphanumeric or national.

Returned value

1. If argument-1 is class alphabetic or alpanumeric, the returned value is the ordinal position of argument-1 in the alphanumeric collating sequence.
2. If argument-1 is class national, the returned value is the ordinal position of argument-1 in the national collating sequence.

See also: CHAR

Example 9-33

```
...
DATA DIVISION.
WORKING-STORAGE SECTION.
01  L PIC X VALUE "Z".
01  R PIC X(3).
PROCEDURE DIVISION.
P1 SECTION.
MAIN.
    COMPUTE R = FUNCTION ORD (L).
    DISPLAY R UPON T.
    STOP RUN.
```

Result: 234
The ordinal position of the letter Z in EBCDIC is 234.

9.2.36 ORD-MAX - Ordinal position of maximum argument

The ORD-MAX function returns an integer value that indicates which of the specified arguments, seen from left to right, contains the maximum value.
The type of this function is integer.

Format

```
FUNCTION ORD-MAX ({argument-1}...)
```

Arguments

1. If more than one argument is specified, all arguments must be of the same class.
2. argument-1 may not be of the class "object“ or "pointer".

Returned values

1. The returned value is the ordinal number that indicates the position of the argument having the greatest value in the argument series.
2. The comparisons used to determine the greatest valued argument are made according to the rules for simple conditions (see section "Conditions").
3. If more than one argument has the same greatest value, the number returned corresponds to the position of the leftmost argument having that value.
4. The error default value is 0.

See also: ORD-MIN, MAX, MIN

Example 9-34

```
...
DATA DIVISION.
WORKING-STORAGE SECTION.
01  R PIC 9(3).
PROCEDURE DIVISION.
P1 SECTION.
MAIN.
    COMPUTE R = FUNCTION ORD-MAX (13 4 9 18 5 7).
    DISPLAY R UPON T.
    STOP RUN.
```

Result: 004
The fourth argument (18) has the greatest value.

9.2.37 ORD-MIN - Ordinal position of minimum argument

The ORD-MIN function returns an integer value that indicates which of the specified arguments, seen from left to right, contains the minimum value.
The type of this function is integer.

Format

```
FUNCTION ORD-MIN ({argument-1}...)
```

Arguments

1. If more than one argument is specified, all arguments must be of the same class.
2. argument-1 may not be of the class "object“ or "pointer".

Returned values

1. The returned value is the ordinal number that indicates the position of the argument having the last value in the argument series.
2. The comparisons used to determine the least valued argument are made according to the rules for simple conditions (see section "Conditions").
3. If more than one argument has the same least value, the number returned corresponds to the position of the leftmost argument having that value.
4. The error default value is 0.

See also: ORD-MAX, MAX, MIN

Example 9-35

```
...
DATA DIVISION.
WORKING-STORAGE SECTION.
01  R PIC 9(3).
PROCEDURE DIVISION.
P1 SECTION.
MAIN.
    COMPUTE R = FUNCTION ORD-MIN ("Z" "3" "B" "?" "a").
    DISPLAY R UPON T.
    STOP RUN.
```

Result: 004
The fourth argument ("?") has the least value.

9.2.38 PRESENT-VALUE - Present value (period-end amount)

The PRESENT-VALUE function returns the principal amount repaid by a series of deferred payments. The deferred payments are given in argument-2; the interest rate at which the payments are calculated is specified in argument-1. The type of this function is numeric.

Format

```
FUNCTION PRESENT-VALUE (argument-1 {argument-2}...)
```

Arguments

1. argument-1 and argument-2 must be class "numeric".
2. The value of argument-1 must be greater than -1.

Returned values

1. The returned value is the approximation of a summation of a series of calculations with each term in the following form:

   argument-2 / (1 + argument-1) ** n

   The exponent n is incremented from one by one for each term in the series.
2. The error default value is - 9'999'999'999'999'999'999'999'999'999'999'999.

See also: ANNUITY

Example 9-36

```
 ...
 DATA DIVISION.
 WORKING-STORAGE SECTION.
*** Interest rate 10% **********
 01  INTEREST PIC 9V99 VALUE 0.10.
*** Four payments *********
01  B-1 PIC 9(4) VALUE 1000.
01  B-2 PIC 9(4) VALUE 2000.
01  B-3 PIC 9(4) VALUE 1000.
01  B-4 PIC 9(4) VALUE 1000.
*** Principal amount repaid ***
 01  PAR PIC 9(6).
 PROCEDURE DIVISION.
 P1 SECTION.
 MAIN.
     COMPUTE PAR = FUNCTION PRESENT-VALUE (INTEREST B-1 B-2 B-3 B-4).
     DISPLAY "Amount repaid: " PAR UPON T.
     STOP RUN.
```

Result: Amount repaid: 003996
This is the principal amount repaid by 4 payments.

9.2.39 RANDOM - Random number

The RANDOM function returns a numeric value that is a pseudo-random number from a rectangular distribution. The type of this function is numeric.

Format

```
FUNCTION RANDOM [(argument-1)]
```

Arguments

1. If argument-1 is specified, it must be zero or a positive integer. It is used as the seed value to generate a sequence of pseudo-random numbers.
2. If a subsequent reference specifies argument-1, a new sequence of pseudo-random numbers is started.
3. If the first reference to this function in the run unit does not specify argument-1, the seed value is 0.
4. In each case, subsequent references without specifying argument-1 return the next number in the current sequence.

Returned values

1. The returned value is greater than or equal to zero and less than one.
2. For a given seed value, the sequence of pseudo-random numbers will always be the same.
3. The range of argument-1 values that will yield distinct sequences of pseudo-random numbers is 0 through $2^{31}-1$.
4. The error default value is `-1`.

Example 9-37

```
 IDENTIFICATION DIVISION.
PROGRAM-ID. LOTTO.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
    TERMINAL IS VIDEO.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 LIST.
 05 ELEM        OCCURS 6 INDEXED BY SI, LI.
  10 Z          PIC Z9 VALUE FROM (1) IS ZERO REPEATED TO END.
  10 T          PIC XX VALUE FROM (1) IS ", " REPEATED TO END.
01 Z-0         PIC Z9.
01 RAN-VAL       PIC V9(18) BINARY.
01 INIT-ARG       PIC 9(5) BINARY.
01 ID-1.
 02             PIC X(5).
 02 D1          PIC 9.
 02             PIC X.
 02 D2          PIC 9.
 02             PIC X.
 02 D3          PIC 9.
 02             PIC X.
 02 D4          PIC 9.
```

```
  02             PIC X.
  02 D5          PIC 9.
  02             PIC X(7).
 01 D6           PIC 9.
 PROCEDURE DIVISION.
 M SECTION.
 M1.
*
* Select a seed for the random function based on the
* time, date, and weekday
*
      MOVE FUNCTION CURRENT-DATE TO ID-1
      ACCEPT D6 FROM DAY-OF-WEEK
      COMPUTE INIT-ARG =
           (10 * D1 + 1000 * D2 + 100 * D3 + D4 + 10000 * D5) * D6
*
* Compute the first random number
*
      COMPUTE RAN-VAL = FUNCTION RANDOM (INIT-ARG)
*
* Traverse the loop until 6 elements have been entered
* into the list
*
      PERFORM VARYING LI FROM 1 BY 1 UNTIL LI > 6
*
* Traverse the loop until a unique number is found
* and entered into the current list element
*
        PERFORM UNTIL Z (LI) NOT ZERO
*
* Map the return value of the RANDOM function
* to an integer between 1 and 49
*
          COMPUTE Z-0 = FUNCTION INTEGER (49 * RAN-VAL) + 1
          SET SI TO 1
*
* Check the result for uniqueness
*
          SEARCH ELEM
*
* If number not found in list -> enter the number
*
          AT END MOVE Z-0 TO Z (LI)
*
* If number is already in the list -> new random number
*
            WHEN Z (SI) = Z-0 CONTINUE
          END-SEARCH
*
* Compute next random number
*
          COMPUTE RAN-VAL = FUNCTION RANDOM
        END-PERFORM
      END-PERFORM
      SORT ELEM ASCENDING Z
      MOVE "." TO T (6)
      DISPLAY LISTE UPON VIDEO
      STOP RUN.
```

Result: 6 numbers from 1 to 49

9.2.40 RANGE - Difference value

The RANGE function returns a value that is equal to the value of the maximum argument minus the value of the minimum argument.
The type of this function depends upon the argument types as follows:

Argument type	Function type
all arguments integer all arguments numeric mix of integer/numeric	integer numeric numeric

Format

```
FUNCTION RANGE ({argument-1}...)
```

Argument

1. argument-1 must be class "numeric".

Returned values

1. The returned value is equal to the greatest value of argument-1 minus the least value of argument-1. The comparisons used to determine the greatest and least values are made according to the rules for simple conditions.
2. The error default value is `-1`.

See also: MIN, MAX, MEAN, MEDIAN, MIDRANGE, ORD-MAX, ORD-MIN, SUM

Example 9-38

```
...
DATA DIVISION.
WORKING-STORAGE SECTION.
01  RES PIC 9(3).
PROCEDURE DIVISION.
P1 SECTION.
MAIN.
    COMPUTE RES = FUNCTION RANGE (12 32 5 8 17 9).
    DISPLAY RES UPON T.
    STOP RUN.
```

Result: 027

9.2.41 REM - Remainder

The REM function returns a numeric value that is the remainder of argument-1 divided by argument-2.
The type of this function is numeric.

Format

```
FUNCTION REM (argument-1 argument-2)
```

Arguments

1. argument-1 and argument-2 must be class "numeric".
2. The value of argument-2 must not be zero.
3. The value v of the quotient argument-1/argument-2 must lie in the range - 10^{31} < v < 10^{31}

Returned values

1. The returned value is the remainder of argument-1 / argument-2. It is defined as the expression: argument-1 - (argument-2 * FUNCTION INTEGER-PART (argument-1 / argument-2))
2. The error default value is - 9’999’999’999’999’999’999’999’999’999’999.

See also: MOD

Example 9-39

```
...
DATA DIVISION.
WORKING-STORAGE SECTION.
01  R PIC 999.
PROCEDURE DIVISION.
P1 SECTION.
MAIN.
    COMPUTE R = FUNCTION REM (928 14).
    DISPLAY R UPON T.
    STOP RUN.
```

Result: 004

9.2.42 REVERSE - Reverse order of string characters

The REVERSE function returns a character string of exactly the same length as argument-1 and whose characters are exactly the same as those of argument-1, except that they are in reverse order.
The type of this function depends upon the argument types as follows:

Argument type	Function type
alphabetic alphanumeric national	alphanumeric alphanumeric national

Format

```
FUNCTION REVERSE (argument-1)
```

Arguments

1. argument-1 must be class alphabetic, alphanumeric or national and must be at least one character in length.
2. argument-1 must not be defined in the ANY LENGTH clause.

Returned values

1. If argument-1 is a character string of length n, the returned value is a character string of length n such that for 1 <= j <= n, the character in position j of the returned value is the character from position n - j + 1 of argument-1.
2. The error default value is a space.

Example 9-40

```
...
DATA DIVISION.
WORKING-STORAGE SECTION.
01  REV PIC X(17) VALUE "never odd or even".
01  RES PIC X(17).
PROCEDURE DIVISION.
P1 SECTION.
MAIN.
    MOVE FUNCTION REVERSE (REV) TO RES.
    DISPLAY RES UPON T.
    STOP RUN.
```

Result: neve ro ddo reven

9.2.43 SIN - Sine

The SIN function returns the sine of the angle or arc that is specified in radians by argument-1.
The type of this function is numeric.

Format

```
FUNCTION SIN (argument-1)
```

Argument

1. argument-1 must be class "numeric".

Returned value

1. The returned value is the approximation of the sine of argument-1 and is greater than or equal to -1 and less than or equal to +1.

See also: ASIN, COS, ACOS, TAN, ATAN

Example 9-41

```
...
DATA DIVISION.
WORKING-STORAGE SECTION.
01  R PIC S9V9(10).
01  RES PIC -9.9(10).
PROCEDURE DIVISION.
P1 SECTION.
MAIN.
    COMPUTE R = FUNCTION SIN (3.1425).
    MOVE R TO RES.
    DISPLAY RES UPON T.
    STOP RUN.
```

Result: -0.0009073462

9.2.44 SQRT - Square root

The SQRT function returns a numeric value that approximates the square root of argument-1.
The type of this function is numeric.

Format

```
FUNCTION SQRT (argument-1)
```

Arguments

1. argument-1 must be class "numeric".
2. The value of argument-1 must be zero or positive.

Returned values

1. The returned value is the absolute value of the approximation of the square root of argument-1.
2. The error default value is -2.

See also: FACTORIAL, LOG, LOG10

Example 9-42

```
...
DATA DIVISION.
WORKING-STORAGE SECTION.
01  R PIC 99V999999.
01  RES PIC 99.999999.
PROCEDURE DIVISION.
P1 SECTION.
MAIN.
    COMPUTE R = FUNCTION SQRT (33).
    MOVE R TO RES.
    DISPLAY RES UPON T.
    STOP RUN.
```

Result: 05.744562

9.2.45 STANDARD-DEVIATION - Standard deviation of arguments

The STANDARD-DEVIATION function returns a numeric value that approximates the standard deviation of its arguments.
The type of this function is numeric.

Format

```
FUNCTION STANDARD-DEVIATION ({argument-1}...)
```

Argument

1. argument-1 must be class "numeric".

Returned values

1. The returned value is the approximation of the standard deviation of the argument series.
2. The returned value is calculated as follows:
   - The difference between each argument value and the arithmetic mean of the argument series is calculated and squared.
   - The values obtained are then added. This quantity is divided by the number of values in the argument series.
   - The square root of the quotient obtained is then calculated. The returned value is the absolute value of this square root.
3. If the argument series consists of only one value, or if the argument series consists of all variable occurrence data items and the total number of occurrences for all of them is one, the returned value is zero.
4. The error default value is -1.

See also: VARIANCE

Example 9-43

```
...
DATA DIVISION.
WORKING-STORAGE SECTION.
01  R PIC 99V9999.
01  RES PIC 99.9999.
PROCEDURE DIVISION.
P1 SECTION.
MAIN.
    COMPUTE R = FUNCTION STANDARD-DEVIATION (2 4 6).
    MOVE R TO RES.
    DISPLAY RES UPON T.
    STOP RUN.
```

Result: 01.6329

9.2.46 SUM - Sum of arguments

The SUM function returns a value that is the sum of all the arguments.

The type of this function depends upon the argument types as follows:

Argument type	Function type
all arguments integer all arguments numeric mix of integer/numeric	integer numeric numeric

Format

```
FUNCTION SUM ({argument-1}...)
```

Argument

1. argument-1 must be class "numeric".

Returned values

1. The returned value is the sum of all the arguments.
2. The error default value is 0.

See also: MAX, MIN, RANGE, MEAN, MEDIAN, MIDRANGE

Example 9-44

```
...
DATA DIVISION.
WORKING-STORAGE SECTION.
01  RES PIC 9(3).
PROCEDURE DIVISION.
P1 SECTION.
MAIN.
    COMPUTE RES = FUNCTION SUM (12 32 5 8 17 9).
    DISPLAY RES UPON T.
    STOP RUN.
```

Result: 00000083

9.2.47 TAN - Tangent

The TAN function returns the tangent of the angle or arc that is specified in radians by argument-1.
The type of this function is numeric.

Format

```
FUNCTION TAN (argument-1)
```

Argument

1. argument-1 must be class "numeric".

Returned values

1. The returned value is the approximation of the tangent of argument-1.
2. The error default value is - 9’999’999’999’999’999’999’999’999’999’999.

See also: ATAN, SIN, ASIN, COS, ACOS

Example 9-45

```
...
DATA DIVISION.
WORKING-STORAGE SECTION.
01  R PIC S9V9(10).
01  RES PIC -9.9(10).
PROCEDURE DIVISION.
P1 SECTION.
MAIN.
    COMPUTE R = FUNCTION TAN (3.1425).
    MOVE R TO RES.
    DISPLAY RES UPON T.
    STOP RUN.
```

Result: 0.0009073466

9.2.48 UPPER-CASE - Uppercase letters

The UPPER-CASE function returns a character string that is the same length as argument-1 with each lowercase letter replaced by the corresponding uppercase letter.The type of this function depends upon the argument types as follows:

Argument type	Function type
alphabetic alphanumeric national	alphanumeric alphanumeric national

Format

```
FUNCTION UPPER-CASE (argument-1)
```

Arguments

1. argument-1 must be class alphabetic, alphanumeric or national and must be at least one character in length.
2. argument-1 must not be defined in the ANY LENGTH clause.

Returned values

1. The same character string as argument-1 is returned, except that each lowercase letter is replaced by the corresponding uppercase letter.
2. The character string returned has the same length as argument-1.
3. Umlauts are not converted.
4. The error default value is a space.

See also: LOWER-CASE

Example 9-46

```
...
DATA DIVISION.
WORKING-STORAGE SECTION.
01  RES PIC X(20).
PROCEDURE DIVISION.
P1 SECTION.
MAIN.
    MOVE FUNCTION UPPER-CASE ("wonderwoman") TO RES.
    DISPLAY RES UPON T.
    STOP RUN.
```

Result: WONDERWOMAN

9.2.49 VARIANCE - Variance of arguments

The VARIANCE function returns a numeric value that approximates the variance of its arguments.
The type of this function is numeric.

Format

```
FUNCTION VARIANCE ({argument-1}...)
```

Argument

1. argument-1 must be class "numeric".

Returned values

1. The returned value is the approximation of the variance of the argument series.
2. The returned value is defined as the square of the standard deviation of the argument series (see STANDARD-DEVIATION function).
3. If the argument series consists of only one value, or if the argument series consists of all variable occurrence data items and the total number of occurrences for all of them is one, the returned value is zero.
4. The error default value is `-1`.

See also: STANDARD-DEVIATION

Example 9-47

```
...
DATA DIVISION.
WORKING-STORAGE SECTION.
01  R PIC 99V9999.
01  RES PIC 99.9999.
PROCEDURE DIVISION.
P1 SECTION.
MAIN.
    COMPUTE R = FUNCTION VARIANCE (2 4 6).
    MOVE R TO RES.
    DISPLAY RES UPON T.
    STOP RUN.
```

Result: 02.6666

9.2.50 WHEN-COMPILED - Date and time of compilation

The WHEN-COMPILED function returns the date and time the program was compiled.
The type of this function is alphanumeric.

Format

```
FUNCTION WHEN-COMPILED
```

Returned value

1. The character positions returned, numbered from left to right, are:

Character position	Contents
1-4	Four numeric digits of the year (Gregorian calendar).
5-6	Two numeric digits of the month of the year, in the range 01 through 12.
7-8	Two numeric digits of the day of the month, in the range 01 through 31.
9-10	Two numeric digits of the hours past midnight, in the range 00 through 23.
11-12	Two numeric digits of the minutes past the hour, in the range 00 through 59.
13-14	Two numeric digits of the seconds past the minute, in the range 00 through 59.
15-21	0000000

See also: CURRENT-DATE, DATE-OF-INTEGER, DAY-OF-INTEGER, INTEGER-OF-DATE, INTEGER-OF-DAY

Example 9-48

```
...
DATA DIVISION.
WORKING-STORAGE SECTION.
01  A-DATE PIC X(21).
PROCEDURE DIVISION.
P1 SECTION.
MAIN.
    MOVE FUNCTION WHEN-COMPILED TO A-DATE.
    DISPLAY A-DATE UPON T.
    STOP RUN.
```

Result: 199604211434270000000

9.2.51 YEAR-TO-YYYY year conversion

The YEAR-TO-YYYY function converts a 2-digit year into a 4-digit year. The end of the 100-year interval in which the year specified in argument-1 falls is determined by adding argument-2 to the current year (the year in which the function is executed) ("floating window").

The type of this function is integer.

Format

```
FUNCTION YEAR-TO-YYYY (argument-1 [argument-2])
```

Arguments

1. argument-1 must be a positive number less than 100.
2. argument-2, if specified, must be an integer.
3. If argument-2 is not specified, the value 50 is taken for the second argument.
4. The sum of the current year and argument-2 must be less than 10000 and greater than 1699.

Returned values

1. The returned value is the year specified in argument-1 together with the century.
   The returned value depends on the intermediate value

   $L_1L_2L_3L_4$ = current year +argument-2 (= last year of the 100-year interval).
   The century is calculated as follows:

   $100 * L_1L_2 + YY$ if $L_3L_4 >= YY$ (YY = argument-1)

   $100 * (L_1L_2 - 1) + YY$ if $L_3L_4 < YY$

2. The error default value is 0

See also: DATE-TO-YYYYMMDD, DAY-TO-YYYYDDD

Example 9-49

```
...
DATA DIVISION.
WORKING-STORAGE SECTION.
01 A-DATE PIC 9(7).
01 CURRENT-YEAR PIC 9(7).
01 YEAR PIC 9(7).
PROCEDURE DIVISION.
P1 SECTION.
MAIN.
*
* Calculation of the function with a floating window:
*
* The 100-year interval in which the calculated year falls is to
* include the years (current year -35) through (current year +64):
```

```
*
    COMPUTE A-DATE = FUNCTION YEAR-TO-YYYY (59 64).
    DISPLAY A-DATE UPON T.                                          (1)
*
* Without 2nd argument (or 2nd argument =50)
* The 100-year interval includes the years (current year -49)
* through (current year +50):
*
    COMPUTE A-DATE = FUNCTION YEAR-TO-YYYY (0).
    DISPLAY A-DATE UPON T.                                          (2)
*
* The 2nd argument can also be negative
* The 100-year interval includes the years (current year -109)
* through (current year -10):
*
    COMPUTE A-DATE = FUNCTION YEAR-TO-YYYY (96 -10).
    DISPLAY A-DATE UPON T.                                          (3)
*
* Calculation of the function with a fixed window
*
* The 100-year interval in which the calculated year falls is to
* include the years 1950 through 2049:
*
* Calculation of the last year of the 100-year interval
* relative to the current year
*
    MOVE FUNCTION CURRENT-DATE(1:4) TO CURRENT-YEAR.
    COMPUTE YEAR = 2049 - CURRENT-YEAR.
* Calculation of the function values
    COMPUTE A-DATE = FUNCTION YEAR-TO-YYYY (50 YEAR).
    DISPLAY A-DATE UPON T.                                          (4)
    COMPUTE A-DATE = FUNCTION YEAR-TO-YYYY (1 YEAR).
    DISPLAY A-DATE UPON T.                                          (5)

*
* The 100-year interval in which the calculated year falls is to
* include the years 1890 through 1989:
*
* Calculation of the last year of the 100-year interval
* relative to the current year
*
    MOVE FUNCTION CURRENT-DATE(1:4) TO CURRENT-YEAR.
    COMPUTE YEAR = 1989 - CURRENT-YEAR.
* Calculation of the function values
    COMPUTE A-DATE = FUNCTION YEAR-TO-YYYY (89 YEAR).
    DISPLAY A-DATE UPON T.                                          (6)
    COMPUTE A-DATE = FUNCTION YEAR-TO-YYYY (90 YEAR).
    DISPLAY A-DATE UPON T.                                          (7)
    STOP RUN.
```

Result:

In the year 1996 the program returns the following results:
(1) 2059
(2) 2000
(3) 1896
(4) 1950
(5) 2001
(6) 1989
(7) 1890

In the year 2050 the program returns the following results:
(1) 2059
(2) 2100
(3) 1996
(4) 1950
(5) 2001
(6) 1989
(7) 1890

10 Report Writer

10.1 General description

Special language elements are available for generating reports (report writer language features). These features allow the user to define the format and contents of a report in the Data Division in such detail that only a few Procedure Division statements are required to produce the report.

A printed report displays specific information in a structured format. The total number of all files and report description entries in a compilation unit must not exceed 254.

The Data Division contains report description entries that

- name the reports to be generated by the Report Writer
- assign the reports to a file (report file) to which they are to be written, and
- describe the content and format of reports to be generated.

The statements by which the reports are generated are specified in the Procedure Division.

The Report Writer produces the desired reports according to the defined formats by

- transferring program data into the reports in the required format,
- accumulating data sums, which are added up to subtotals and final totals,
- updating line and page counters, and
- editing and printing the generated reports.

The Report Writer described in this manual is functionally equivalent to the optional Report Writer module described in ANS85, except for some minor differences in syntax.

10.1.1 General description of the Data Division

The output file to which the report is to be written must be declared in the Data Division by means of a file description (FD) entry, which also contains the names of the reports (see section "REPORT clause").

The REPORT SECTION, which defines the format and content of each report, must be specified as the last section of the Data Division. This section contains two types of entries:

1. **The report description** (RD).
   It must specify the name of the report. A page format (PAGE LIMIT clause), a character (CODE clause) preceding each line of the report (not to be printed, however), and a hierarchy of control data items (CONTROL clause) may also be supplied here.
2. **The report group description entry**.
   It describes, for instance, the print fields of a report group; that is, data items with line and column positioning into which data will be entered either directly (VALUE) or indirectly (sending field, sums).

The report description entry is used to specify the format of a page by defining the number of lines per page as well as the line numbers as vertical boundaries of the region within which each type of report group may be printed. These entries are used to control the positioning of the various report groups at the time the report is produced.

Detail report groups are structured by specifying **control data items** in their hierarchical order.

Detail report groups, which are functionally related to the hierarchical items, are combined into several groups in the order in which they are created, so that all detail report groups within a given group are assigned the same value of the hierarchically lowest item. Depending on whether a control heading and/or control footing is defined for this level of hierarchy, each group is introduced by the control heading and is terminated by the control footing (which generally contains information applicable only to the current group, e.g. subtotals).

Similarly, when another hierarchical item occurs, the above series of detail report groups is, in turn, combined into groups, now governed by the next highest item in the hierarchy, etc.

This structure is generated by the Report Writer as follows:

before creating a detail report group, the Report Writer tests the control data items in their hierarchical order, from top to bottom. As soon as it detects a change in value, i.e. a **control break**, all existing control footings are created in hierarchical order from bottom to top, up to the level where the first change in value was encountered. This is followed by all existing control headings on the same level in reversed hierarchical order, and finally, by the detail report group itself.

The specification of control data items in conjunction with the report groups "control heading" and "control footing" thus enables the user to have the report produced in a structured format.

The **report group description entry**, formally similar to a record description, is used to describe the properties of all data items in the report. Beyond the ordinary framework of COBOL language elements for the description of data items, a report data item is assigned line and column, i.e. a page position; in other words, the data item is declared to be a print field. Each of these fields receives information.

Three types of information may be supplied:

SOURCE information (SOURCE clause information), which is available through a data item defined outside the REPORT SECTION.

SUM information (SUM clause information), as a result of adding up data representing, in turn, SOURCE information and/or SUM information.

VALUE information (VALUE clause information), as a predefined information value.

Since a given report may contain several types of report groups, the report group description specifies the type of the report group. The seven types that may be used are shown in table 38.

With its report and all associated report group description entries, the report is completely described as to its format and content (including the summation operations required); that is, all prerequisites to writing the report are met.

Type	Definition
REPORT HEADING	A report group that is written only once, as the first group of the report.
REPORT FOOTING	A report group that is written only once, as the last group of the report.
PAGE HEADING	A report group written at the top of each page. Exceptions: 1. The page heading is written after the report heading. 2. The page heading is suppressed when the page is explicitly reserved for the report heading or footing.
PAGE FOOTING	A report group written at the bottom of each page (exceptions: same as page heading).
DETAIL	This report group type is the only one which is not produced automatically but must be specified in a GENERATE statement. Each execution of this statement produces the specified detail report group at object time.
CONTROL HEADING	This report group is generated as a heading group for a series of detail report groups, due to a control break.
CONTROL FOOTING	This report group is generated as a footing group for a series of detail report groups, due to a control break. It usually sums up data concerning the preceding series of detail report groups.

Table 38: Report group types used in report group description

10.1.2 General description of the Procedure Division

In the Procedure Division, the programmer uses the INITIATE, GENERATE, and TERMINATE statements to instruct the Report Writer to produce the desired report according to its full description in the Data Division.

The INITIATE statement instructs the Report Writer to perform the initialization required for writing the specified report(s). This initialization enables the Report Writer to recognize, for example, whether a given GENERATE statement is the first GENERATE statement to be executed in the sequence of a program.

The GENERATE statement is responsible for creating the major part of the report. It causes a number of automatic functions to be performed in addition to generating a detail report group. For example, the control field test is carried out and a control break takes place, if necessary; that is, control footings and control headings are created. If the associated report group zone of the page does not provide space for a body group (detail report group, control footing, and control heading), then, automatically, the page footing is generated, a page change is implied, and the heading group is created. All of the functions involved in writing the individual report groups, such as incrementing or decrementing counters, supplying value, source, or sum information to the print fields, as well as positioning and writing the line(s) into which the print fields of a report group are structured, are performed automatically.

The TERMINATE statement instructs the Report Writer to complete the creation of the specified report(s). Thus, all control footings that belong to the report and, if present, the report footing as the last report group are written.

Within the range of the DECLARATIVES, the programmer may specify user procedures for report groups following a USE BEFORE REPORTING statement. As such procedures are executed immediately prior to the actual creation (editing, etc.) of the associated report group, this provides the programmer with a means of influencing the representation of the report group at object time.

The following four special registers are known to, and used by, the Report Writer:

LINE-COUNTER, PAGE-COUNTER, PRINT-SWITCH and CBL-CTR.

A report file is a sequentially organized file which is subject to the following restrictions:

Before entering an INITIATE statement, an OPEN OUTPUT or OPEN EXTEND statement must be performed. A TERMINATE statement must be followed by a CLOSE statement (without REEL or UNIT option). No further I-O statement may be issued for this file.

10.2 Language elements of the Data Division

10.2.1 REPORT clause

Function

The REPORT clause relates one or several reports described in the REPORT SECTION to a file in whose file description (FD) entry this clause is specified. Each of these reports is written line by line to the file assigned by the REPORT clause and described as an output file.

Format

```
{REPORT | REPORTS} {IS | ARE} {report-name-1}...
```

Syntax rules

1. Each report-name specified in the REPORT clause must be the subject of a report description entry defining that name as a report-name.
2. Within a class definition, the REPORT clause may only be specified in a method definition.

General rules

1. The name of each report to be produced by the Report Writer must appear in the REPORT clause of the file description for the sequential output file to which the report is to be written.
2. Specification of several report-names in one REPORT clause relates these reports to the file whose file description entry contains the REPORT clause. When created, these reports are written to the assigned file regardless of the order in which the report names are specified and regardless of their formats, their lengths or any similar details.
3. Each report-name defined by a report description entry must be specified in one, and only one, REPORT clause; that is, a given report may be assigned to one file only.
4. The record format (variable, fixed, or unspecified length) is determined by the entry in the RECORD CONTAINS clause.

10.2.2 REPORT SECTION

Function

The REPORT SECTION describes the format and contents of the reports to be generated.

Format

```
REPORT SECTION.

{report-description-entry {report-group-description-entry} ... } ...
```

General rules

1. The REPORT SECTION must be the last section of the Data Division, unless SUB-SCHEMA SECTION is also specified (see section "Structure of a Data Division").
2. The report group description entries must all follow the related report description entry in a body.
3. A report group description entry formally corresponds to a record description of the FILE SECTION or WORKING-STORAGE SECTION. The report group description entry comprises all of the data description entries for a single report group. The first entry in the report group description begins with level-number 01. All associated data description entries must begin with level-number 02 (see section "Report group description entry").

10.2.3 Report description entry

Function

The report description (RD) entry is used to name a report, define its page format, identify its lines, and for structuring the report. It may perform the following functions as required.

1. Specification of a character identifying the print lines of a report (CODE clause).
2. Declaration of a group hierarchy for structuring the report (CONTROL clause).
3. Definition of a page format by specifying vertical boundaries for report group specific regions (PAGE LIMIT clause).

Format

```
RD report-name

    [CODE clause]
    [CONTROL clause]
    [GLOBAL clause]
    [PAGE LIMIT clause]
```

Syntax rules

1. RD is the level indicator, which identifies the beginning of the report description and must immediately precede the report-name.
2. The report-name must be specified in a REPORT clause of the file description entry for the file on which the report is to be written.
3. If the GLOBAL clause is specified in the report description entry, the data items referenced in the clauses of the report description entry and in the clauses of the report group description entries may not be defined in a LOCAL-STORAGE SECTION.
4. No more than 31 data-names may be specified in a CONTROL clause within a report description.
5. The report-name identifies the report and, accordingly, must be unique.

The RD entry clauses are described on the pages that follow.

10.2.4 CODE clause

Function

The CODE clause specifies a character for identifying the print lines that belong to a specific report. This character is inserted in the first position of each print line of the report but must not be printed. It is used to separate the print lines of two or more reports that are interleaved or written consecutively on the report file.

Format

```
CODE literal
```

Syntax rules

1. The literal must be an alphanumeric literal. The COBOL2000 compiler uses only the first character of the literal for marking the report.
2. If a report description entry includes the CODE clause, all other report description entries assigned to the same output file (through the REPORT clause) must have a CODE clause also.
3. The character which is written at the beginning of each print line is associated with the literal.
4. The character for identifying the print lines of a report is inserted at the beginning of the line, following the carriage control character. This character must not appear in the print format.

General rule

1. The CODE clause must not be specified if the report is to be printed immediately ("online") or if it is assigned to the report file SYSLST. It must be ensured that reports do not overlap, i.e. that during the time lapse between the INITIATE and TERMINATE statement for one report no GENERATE statement is executed for another report.

10.2.5 CONTROL clause

Function

The CONTROL clause defines the data items that are used as control data items of the report to determine the group hierarchy and, thus, the hierarchical structuring of the report.

Format

```
{CONTROL | CONTROLS} {IS | ARE} {{data-name-1}... | FINAL [data-name-1]...}
```

Syntax rules

1. No more than 31 data-names may be entered in the CONTROL clause, including the FINAL phrase if specified.
2. data-name-1... may be qualified but must not be indexed.
3. No data item may be subordinated to a data-name if its length is defined in the OCCURS clause as variable.
4. data-name-1... must not be defined in the REPORT SECTION but rather in the FILE SECTION, LOCAL-STORAGE SECTION or WORKING-STORAGE SECTION. A data-name entered in the CONTROL clause may be defined in the LINKAGE SECTION of the called program, provided that the called program is continuously available in memory from the time the report is initiated until its production is terminated.
5. The data item used may be up to 256 characters in length.
6. Each data-name-1 must identify a different data item.
   Two separate recurrences of data-name-1 must not refer to data items which (by redefinition) refer to the same memory locations.
7. A control data item is a data item which is specified in the CONTROL clause. It is tested whenever a GENERATE statement is executed for the same report in order to ascertain whether its value has changed since the last GENERATE statement was executed for that report.
   If such a change in value is found to have taken place, a "control break" occurs, i.e. special action (described below) will be taken before the detail report group specified by the GENERATE statement is written. If changes in value have occurred in several control data items when a GENERATE statement is executed, it is always the hierarchically supreme change in value to which all control break concepts (such as control break and control break level) are related.
8. FINAL, data-name-1... define the control hierarchy of the report.
9. The data-names, in the order in which they are listed from left to right, specify the levels of the control hierarchy from major to minor. The last (i.e. rightmost) data-name is assigned the lowest level, the last but one is assigned the lowest level, the last but one is assigned the second lowest level, and so on.
10. FINAL defines the hierarchically highest control break. The associated control heading is generated when the first GENERATE statement is executed; the associated control footing when the TERMINATE statement is executed.

General rules

1. The action implied by a control break depends on whether a control heading and/or a control footing, or neither of the two are defined for each control hierarchy level. If a control break occurs, the Report Writer creates the following control headings and control footings (as present), in the order shown below:
   a. Control footing of the lowest level.
   b. Control footing of the next higher level.

.

.

.

c. Control footing of the level responsible for the control break.

d. Control heading of the level responsible for the control break.

e. Control heading of the next lower level.

.

.

.

f. Control heading of the lowest level.

Next, the Report Writer writes the detail report group initiated by the GENERATE statement.

For example, when the control break data-names for a report are specified as YEAR, MONTH, and DAY (listed in this order in the CONTROL clause), associating each of these data-names with one control heading as well as one control footing, then, if a control break occurs for YEAR (that is, the contents of the data item YEAR has changed between two chronologically successive GENERATE statements), the body groups are printed in the following order:

Control footing	for DAY
Control footing	for MONTH
Control footing	for YEAR
Control heading	for YEAR
Control heading	for MONTH
Control heading	for DAY

Detail report group produced by the GENERATE statement.

2. If the Report Writer starts creating a body group and detects that one of the conditions for a page break exists, it will first write the page footing (if specified), then skip to the next page, then write the page heading (if specified), and finally generate the body group.
3. The creation of a detail group must be requested by the programmer from the Report Writer by means of an appropriate GENERATE statement (to define the detail group) in the Procedure Division. All other report groups such as report heading, report footing, page heading, page footing, control headings, and control footings are written automatically by the Report Writer as soon the respective conditions are satisfied.
4. In order to determine a control break, an alphanumeric compare is performed, regardless of how the control data items are described. This means, for example, that the Report Writer will detect a change of value when the contents of a control data item described with COMPUTATIONAL-3 is changed from hexadecimal "7F" to hexadecimal "7C", although both variant representations print out the positive number 7.
5. When a CONTROL clause is not specified for a report, control headings and control footings must not, and cannot, be defined for the report.

Example 10-1

Extract from a report:

```
JANUARY 14  B10                   4  B    8.36
            B10                   1  C    9.00
PURCHASES & COST FOR 1-14         5     $17.36  $136.36
JANUARY 15  B10                   2  A   16.00
```

The relevant report description entry includes this CONTROL clause:

```
CONTROLS ARE FINAL MONTH WDAY
```

Except for the print line beginning with PURCHASES and constituting the control footing for WDAY, all other print lines from the above report extract are detail groups (same report description). The control footing of WDAY was written because the data changed from January 14 to January 15.

10.2.6 GLOBAL clause

Function

The GLOBAL clause can only be used within a nested program. It defines a report-name as global. A global name can be accessed by the program defining it and by any other program contained directly or indirectly in this program.

Format

```
IS GLOBAL
```

Formats for the file description entry can be found under the FILE SECTION.

Syntax rule

1. The GLOBAL clause may only be specified in report description entries

General rules

1. A report-name whose description contains a GLOBAL clause is a global name. All data-names that are subordinate to and associated with a global name a global name are global names.
2. Any program contained in the program that describes the global name may access a global name. The global name does not need to be described again in the program that references it. In the case of references to identical names, the local names have priority (see section "Local and global names").
3. The LINE-COUNTER and PAGE-COUNTER special registers are also global.

10.2.7 PAGE LIMIT clause

Function

The PAGE LIMIT clause is used to define the length of a page and the vertical subdivisions

within which report groups are presented.

Format

```
PAGE [{LIMIT | LIMITS} {IS | ARE}] integer-p [{LINE | LINES}]
     [HEADING integer-h]
     [FIRST DETAIL integer-d]
     [LAST DETAIL integer-e]
     [FOOTING integer-f]
```

Syntax rules

1. integer-p indicates the maximum possible number of print lines per page.
2. integer-p must not exceed 999.
3. The other integers of the PAGE LIMIT clause are line numbers. Since the numbering of the lines on a page starts with 1, integer-p may also be interpreted as a line number.
4. The integers of the PAGE LIMIT clause are subject to the following relationships:
   integer-h must be equal to or greater than 1.
   integer-d must be equal to or greater than integer-h.
   integer-e must be equal to or greater than integer-d.
   integer-f must be equal to or greater than integer-e.
   integer-p must be equal to or greater than integer-f.
5. integer-h of the HEADING phrase specifies the first line on the page on which anything may be written.
6. integer-h must not exceed 999.
7. integer-p of the LIMIT phrase specifies the last line on the page on which anything may be written.
8. integer-d of the FIRST DETAIL phrase specifies the lowest line number permitted for any body groups.
9. integer-d must not exceed 999.
10. integer-e of the LAST DETAIL phrase specifies the highest line number permitted for any detail groups and control headings.
11. integer-e must not exceed 999.
12. No more than 127 detail report groups (DETAIL) may be specified within a report description entry.
13. integer-f of the FOOTING phrase specifies the highest line number permitted for any control footings.
14. No more than 31 control footings may be specified within a report description entry.
15. integer-f must not exceed 999.
16. If the PAGE LIMIT clause is specified but one or more of the optional integers is omitted,the following values are internally assumed by default:

Omitted integer	Value assumed when nothing is specified
HEADING integer-h	1
FIRST DETAIL integer-d	Value of HEADING integer-h
LAST DETAIL integer-e	Value of FOOTING integer-f
FOOTING integer-f	Value of LIMIT integer-p

17. If the PAGE LIMIT clause is omitted entirely, the compiler assumes the following values for each integer of the PAGE LIMIT clause:

Integer	Assumed value
LIMIT integer-p	50
HEADING integer-h	1
FIRST DETAIL integer-d	1
LAST DETAIL integer-e	48
FOOTING integer-f	48

General rules

1. The Report Writer uses the phrases of the PAGE LIMIT clause for partitioning the page into regions. Only report groups of specific types may be printed in certain regions. The page regions for each type of report group are:
   a. If the report heading description includes the NEXT GROUP NEXT PAGE clause, the report heading prints from the line whose number is integer-h through the line whose number is integer-p. Otherwise, the report heading must not print beyond the line whose number is integer-d minus 1, which requires explicit entry of integer-d (see under b. below).
   b. The page heading must lie in the region from integer h through integer d minus 1.
   This region does not exist if the value of integer-d is established by the compiler. It is therefore necessary for integer-d to be supplied explicitly in the PAGE LIMIT clause when printing a page heading or report heading which does not appear on a page by itself.
   c. Detail report groups and control headings may be printed only in the region from integer-d through integer-e, these boundaries included.
   d. Printout of control footings is permitted in the region from integer-d through integer-f.
   e. A page footing may be printed only in the region from integer-f+1 through integer-p. This region does not exist if the PAGE LIMIT clause is supplied without the integer-f phrase. Therefore, in this case, printing a page footing requires that integer-f be specified explicitly.
   f. If a whole page is associated with the report footing by means of LINE NEXT PAGE, the report footing may print in the region from integer-h through integer-p.
   Otherwise, report footings are subject to rule e).

2. NEXT GROUP and LINE clauses of the report group description entries must not conflict with the PAGE LIMIT clause; that is, the lines of a report group must be within the assigned region.
3. Table 39 is a schematic view of the partitioning of a page into regions in those cases where each integer of the PAGE LIMIT clause has a different value. The partitioning is represented as follows:

   integer-h < integer-d < integer-e < integer-f < integer-p

   This is also the assumption for the rules below for the use of these regions:

integer	Page region and valid entries				
	Region 1 REPORT HEADING and REPORT FOOTING	Region2 PAGE HEADING and (if any) REPORT HEADING	Region 3 DETAIL and CONTROL	Region 4 CONTROL FOOTING	Region5 PAGE FOOTING and (if any) REPORT FOOTING
integer-h					
integer-h +1					
.					
.					
.					
integer-d −1					
integer-d					
integer-d +1					
.					
.					
.					
integer-e −1					
integer-e					
integer-e +1					
.					
.					
.					
integer-f −1					
integer-f					
integer-f +1					
.					
.					
.					
integer-p −1					
integer-p					

Table 39: Page partitioning into regions (schematic)

Region 1:

Scope: integer-h through integer-p.

Contents: report heading or report footing.

Rules: If the report heading description includes the NEXT GROUP clause with the NEXT PAGE phrase, a whole print page is reserved for the report heading; that is, no other report group is printed on this page. Whether the report head is printed using the whole or any part of the region, is determined by the LINE clause of the report heading description.

If the report footing description includes the first (or only) LINE clause with the NEXT PAGE phrase, a whole print page is reserved for the report footing; that is, the same rules apply as to the report heading.

Region 2:

Scope: integer-h through integer-d minus 1.

Contents: report heading and page heading, or page heading only.

Rules: The report heading will be printed in region 2 only if its report group description does not include the NEXT GROUP PAGE clause. Since a report heading is printed only once per report, the report heading appears in region 2 only on the first page of the report.

With the exception of those pages which are exclusively reserved for the report heading and report footing, a page heading, if defined, will appear in region 2 on all pages.

If the first page is provided with both the report heading and the page heading, the page heading must be printed after the report heading. LINE and NEXT GROUP clauses of the two report description entries must not violate this rule.

Region 3:

Scope: integer-d through integer-e.

Contents: body group.

Rules: As shown by the scope, region 4 generally extends beyond region 3 at the bottom.

Any body group may be written in region 3 or in the overlapped portion of region 4. The remaining portion of region 4 may receive control footings only.

Region 4:

Scope: integer-d through integer-f.

Contents: body group.

Rules: As shown by the scope, region 4 generally extends beyond region 3 at the bottom.

Any body group may be written in region 3 or in the overlapped portion of region 4. The remaining portion of region 4 may receive control footings only.

Irrespective of the body’s group type, region 4 applies to the NEXT GROUP clause of this group.

Region 5:

Scope: integer-f+1 through integer-p.

Contents: report footing and page footing.

Rules: If the first LINE clause in the report description entry for the report footing does not contain the NEXT PAGE phrase, the report footing will be printed in region 5 on the last page of the report.

The page footing of a report is always written in region 5.

If both the page footing and the report footing are to be printed in region 5 of the last page, then the page footing must precede the report footing. The LINE clauses of the two report groups must not be in conflict with this rule.

10.2.8 Report group description entry

Function

The report group description entry describes and defines the format, type, and properties of a report group as a series of elementary items and items associated with information. These items, organized in lines and columns, form the line(s) to be printed in a report group.

A prepositioning to the next report group to be written may be achieved by a NEXT GROUP phrase.

Using the type of report group, demands are placed both on the description of the report group and on the Report Writer when the report group is created. For example, the page positioning of a report group is also determined by the type of the report group. Much the same is true of the time the report group is produced. Summations, for instance, can be defined in control footings only. They are executed immediately preceding the creation of detail groups or the control footings concerned, depending on type of summation.

Format

```
01 [data-name-1]
    TYPE clause
   [LINE clause]
   [NEXT GROUP clause]

02 [data-name-2]
   [LINE clause]
   [COLUMN clause]
   [GROUP INDICATE clause]
   [ PICTURE clause]
   [BLANK WHEN ZERO clause]
   [JUSTIFIED RIGHT clause]
   [SIGN clause]
   [USAGE clause]

   {SOURCE clause | SUM clause | VALUE clause}
```

Syntax rules

1. The description of a report group must begin with a 01-level entry, followed by at least one 02-level entry.
2. With the exception of the data-name, which must immediately follow the level-number, the clauses may be written in any order.
3. A report consists of a series of report groups. Up to seven different report group types (see section "TYPE clause") may appear in the report.

   By definition, a report group consists of one or several elementary items, where each item with a COLUMN clause (printable item) must be assigned to one line. Report groups which do not contain printable items are non-printable. A printable report group is a unit comprising one or more lines.

   With the exception of the report heading and the report footing, any report group may be created several times in the course of creating report; in this case, there must be no change in the report group format or in the constant information contents.
4. No more than 127 report group description entries may be specified in a report description entry.
5. The data-name which immediately follows the level-number 01 is the name of the report group. It must be supplied if the report group is to be referenced directly in a GENERATE statement (detail group), a USE BEFORE REPORTING statement, an UPON phrase within a SUM clause, or is to be used for qualification.

6. The TYPE clause specifies the type of report group, enabling the Report Writer to determine the modalities (when, where) for producing that report group.
7. The LINE clause assigns the printable items specified within its range to a line of the current or next print page. The first LINE clause of a report group description entry therefore determines the positioning of that report group. The range extends until the next LINE clause or the end of the report group entry.
8. The NEXT GROUP clause causes (after a report group is printed) the next report group to be positioned to the line number supplied in the phrases of the NEXT GROUP clause.

   The COLUMN, GROUP INDICATE, BLANK, JUSTIFIED, and PICTURE clauses define the location and format of the printable data item for a particular print line of the report group.
9. SOURCE, SUM, and VALUE clauses associate the data items with the respective information. In addition, the SUM clause also defines an internal sum counter whose contents is automatically updated by the Report Writer.
10. Each printable item must be covered by a LINE clause. This LINE clause represents a print line.
11. The PICTURE clause can only be omitted for an elementary item if a VALUE clause is specified with an alphanumeric or hexadecimal alphanumeric literal. The compiler then assumes a PICTURE clause PICTURE X (length), where length is the number of characters displayed by the literal.

General rules

1. The name of a report group may be specified in the REPORT SECTION and in the Procedure Division. Only names of control footings and detail groups may be supplied in the REPORT SECTION. While the name of a detail group may be defined in the UPON phrase of a SUM clause, the name of a control footing may be used only for qualifying a sum counter. In the Procedure Division, the name of a report group may be specified only in two cases:
   - The name of a detail group may be specified in a GENERATE statement.
   - In a USE BEFORE REPORTING statement, any report group name (except names of detail groups) may be used.

   As a qualifier for a report group name, only the associated report name is allowable.
2. A data-name immediately following the level-number 02 of a report group entry should not be specified unless that data item description includes a SUM clause. In this case, the data-name will be interpreted as the name of the sum counter which is established as a result of the SUM clause (see section "SUM clause"). The name of a sum counter may again be specified in a SUM clause. If the data-name is supplied although no SUM clause is present, in other words, if the data-name is defined as the name of the item, its use is illegal.

   Sum counters may only be qualified by report group names and/or report names.

Report group description entry clause summary

Table 40 lists the report group entry clauses and briefly describes each.

The BLANK WHEN ZERO, JUSTIFIED, PICTURE, USAGE and VALUE clauses are also used in other sections of the Data Division; that is, they are assumed to be known here (refer to the pertinent descriptions).

All other report entry clauses are described on the following pages, in alphabetical order.

Clause	Brief description
BLANK WHEN ZERO	Indicates that a printable item is not to be printed when its contents are zero.
COLUMN	Defines an item as printable by specifying the starting position (column number) of that field on the print line.
GROUP INDICATE	Indicates that the printable item is to be printed only if the detail group with which it is associated is written for the first time in the report, after a page advance or a control break.
JUSTIFIED	Explicitly specifies the positioning of data in the associated item.
LINE	Specifies the line on the report page on which one or more printable items within its range are to be printed.
NEXT GROUP	Specifies prepositioning for the next report group to be printed.
PICTURE	Specifies the characteristics of a data item or of a data item and an internal sum counter.
SIGN	Specifies the position and the mode of representation of the operational sign for numeric data items.
SOURCE	Associates a data item with a sending field as its source of information.
SUM	Indicates how and which addends are to be accumulated in the internal sum counter. The sum counter is built automatically. Also, this clause associates the data item with the sum counter as its source of information.
TYPE	Specifies the particular type of report group being described.
USAGE	Specifies the data format used for storing an elementary item.
VALUE	Defines a constant value for a printable item.

Table 40: Report group entry clauses

10.2.9 COLUMN clause

Function

The COLUMN clause defines a data item as a printable field by specifying its column number (integer) to indicate the starting position of that item with respect to the print line.

Format

```
COLUMN integer
```

Syntax rules

1. integer must be an unsigned integer in the range of 1 <= integer <= 251.
2. The column number (integer) specifies the position in which the first (leftmost) character position of the printable item is to appear on the print line. The first printable character position of the print line is considered to be column 1. The highest column number depends on the type of printer being used.

General rules

1. The presence of the COLUMN clause in the description entry for a data item containing the mandatory PICTURE clause as well as either a SOURCE, SUM, or VALUE clause constitutes a printable item.
2. Every entry that defines a printable item must either contain, or be preceded by (in the same report group description), a LINE clause. The printable item will be printed on the print line specified by the LINE clause.
3. Within the scope of a LINE clause (that is, before the next LINE clause occurs or until the report the report entry ends), the printable items must be defined in ascending column number sequence. The printable items are printed on the line in the order in which they were defined.
4. Printable items on a print line must not be so defined that they overlap each other (concerning the COLUMN and PICTURE clauses).
5. Immediately before the printable items in a line are printed, i.e. before the print line is written to the report file, the information is automatically moved to these items by implicit MOVE statements. Depending on whether the description of the receiving item (=printable item) includes the SOURCE, VALUE, or SUM clause, the sending item will be the identifier from the SOURCE clause, the literal from the VALUE clause, or the internal sum counter which was established for the SUM clause.
6. The Report Writer supplies space characters for all positions of a print line that are not occupied by printable items.

Example 10-2

Assume the following description of an elementary item in a report group entry:

```
02 LINE 3 COLUMN 30 PIC A(12) VALUE IS "EXPENDITURES"
```

When this report group is produced, the character-string EXPENDITURES will be printed on line 3 of the current report page, starting at column 30.

10.2.10 GROUP INDICATE clause

Function

The GROUP INDICATE clause indicates to the Report Writer that the printable item with which it is associated is to be printed only if the detail report group is written for the first time on a page of the report or for the first time after a control break.

Format

```
GROUP INDICATE
```

Syntax rules

1. The GROUP INDICATE clause may be used only in a detail report group. The item description containing the GROUP INDICATE clause must also include a COLUMN clause.
2. The GROUP INDICATE clause causes the Report Writer to print the associated printable item according to the SOURCE or VALUE clause specified, only in the following cases:
   - On the first presentation of the detail report group in the report
   - On the first presentation of the detail report group after a page advance or
   - On the first presentation of the detail report group after a control break.

   In all other cases, space characters are substituted for the printable field, suppressing the associated data in the print layout.

Example 10-3

- Data Division entries:

```
         .
         .
         .
    CONTROLS ARE FINAL, MONTH, WDAY
         .
         .
         .
01  DETAIL-LINE PLUS 1 TYPE DETAIL.
    02  COLUMN 2 GROUP INDICATE PIC A(9)
        SOURCE MONTHNAME (MONTH).
    02  COLUMN 13 GROUP INDICATE PIC 99 SOURCE WDAY.
         .
         .
         .
```

- Excerpt from a report

```
        .
        .
        .
JANUARY 15 B11 16 A 20.00
           B11  4 B 13.45
           B11 20 D 35.40
```

The above extract from a report comprises three detail report groups of the same report group definition, created immediately following a control break (change in the current contents of DAY).The current values JANUARY and 15 of the sending items (SOURCE items) for the first two printable items of this 1-line detail report group appear, therefore, only in the first line (detail report group).

10.2.11 LINE clause

Function

The LINE clause specifies the line on the report page on which one or more printable items defined within its scope (see section "COLUMN clause") are to be printed. In special application, it is used simply for page advance.

Format

```
LINE NUMBER IS {integer-1 | PLUS integer-2 | NEXT PAGE}
```

Syntax rules

1. integer-1 and integer-2 may only be specified as unsigned integers. integer-1 must be greater than or equal to 1, integer-2 must be greater than or equal to 0. The value of integer-1 or integer-2 must not exceed 999.
2. A LINE clause with integer-1 phrase specified is called an absolute LINE clause. integer-1 is interpreted as a line number. Therefore, when creating the associated report group, integer-1 indicates the particular line on the report page on which the printable items which belong to the LINE clause and, thus, are organized as a line will ultimately be written. In this way, an absolute LINE clause always defines a print line.
3. A LINE clause with PLUS integer-2 phrase specified is called a relative LINE clause. The print line to receive the printable items associated with a relative LINE clause at the time the particular report group is generated, is determined relative to the last vertical positioning. For this purpose, integer-2 is added to the number of the line on which the report page is presently positioned (see section "NEXT GROUP clause" and section "LINE-COUNTER special register"). The sum then designates the number of the next line to be printed.

   Deviations from the above rules are only permissible for special types of report groups, when the relative LINE clause was used for the first line of a report group. These deviations will be discussed in due course.
4. A LINE clause with a NEXT PAGE phrase causes the associated report group to be printed on a free page (usually the next page), advancing the page before the report group is printed.

   Whether a LINE NEXT PAGE clause additionally specifies a print line can be seen from the rules described below.

General rules

The following programming rules are classified into two categories -"General notes" and "Report group syntax notes". Rules in the latter category describe the use of the LINE clause in the descriptions of the different types of report groups.

The abbreviations used in the report group notes have the following meaning.

Abbreviation	Meaning
A	one or more absolute LINE clauses
R	one or more relative LINE clauses
NP	a LINE clause with NEXT PAGE phrase

1. A printable report group item that is to be written on a line of the report page must either itself contain a LINE clause in its description or its description must be preceded, within the associated report group description entry, by a LINE clause. Conversely, all printable items in the associated report group description entry, which

follow the LINE clause and precede the next LINE clause or the end of that report group description entry, will be written on the line determined by the LINE clause. Subsequent printable items from the report group entry are printed combined into lines in the same manner.

2. The vertical spacing parameters (LINE and NEXT PAGE clauses) must be selected for the different report group types in such a way that these report groups can be printed within the page regions provided for them (see section "PAGE LIMIT clause" and section "TYPE clause"). It is not possible to continue a report group in the associated specific region on another page.

   If a body group cannot be printed in its specific region simply because part of that region is no longer available, then that body group as a whole will be written in the same region on the next page, after creating first the page footing on the old page and the page heading on the new page.

3. If a report group entry includes (one or more) absolute LINE clauses, they must all be specified:
   - preceding the first relative LINE clause if any, and
   - in the order of ascending integers.
4. The LINE clause with the NEXT PAGE phrase may only be specified in a report group description entry (01 level) to position to the next page.

Report group syntax notes

1. Report heading
   The following sequences of LINE clauses may be used in describing a report heading group:
   `{A | A R}`
2. Page heading
   Only the following sequences of LINES clauses may be used in describing a page heading group:
   `{A | A R | R}`

   When the first LINE clause from the description of the page heading is a relative clause, then, generally, the first line of the report group is printed relative to the region boundary integer-h of the PAGE LIMIT clause (see section "PAGE LIMIT clause"). Only when the report heading was printed on the same page will there be deviation in that the first line of the report group is printed relative to the vertical spacing resulting from the creation of the report group heading.
3. Body groups

   Detail groups, control headings, and control footings are referred to generically as body groups.

   a. The following sequences of LINE clauses may be used in describing a body group:
      `{NP | NP A | NP A R | NP P | A | A R | R}`
   b. When NP A or NP A R sequences are specified for a LINE NEXT PAGE clause, NP provides page changing instructions, and does not define a print line. Thus, the first absolute LINE clause must appear with, or prior to, the first entry defining a printable item.
   c. If a LINE clause with NEXT PAGE phrase is specified as the only LINE clause or with the sequence NP R, this implies not only page advance but also vertical
      spacing of the first line of that report group.
   d. A LINE NEXT PAGE clause in the description of a body group indicates to the Report Writer that the body group is to be printed on the next page which is not yet occupied by a body group. This may lead to a situation where no page advance is carried out. This will certainly be the case when the body group is to be written as the first body group on a particular report.
   e. If a body group defined with a LINE clause sequence NP, NP R, or R is the first body group to be printed on a page, then the first print line of that body group will be printed on the line whose number is integer-d

as specified in the PAGE clause. However, it may well be the case that the paper has already been advanced beyond region boundary integer-d, e.g. through the last body group printed because of a NEXT GROUP clause (see section "NEXT GROUP clause"). In this case, the first line of the body group will be written on the line immediately following the present positioning.

f. If a body group whose first LINE clause is relative is not to be printed as the first body group on a page, the present positioning (line number) is advanced by the number of lines specified by integer-2 in the first LINE clause, to the new position where the first line of this report group will be printed.

4. Page footing
   Only the following sequences of LINE clauses may be used in describing a page footing:
   `{A | A R}`
5. Report footing
   a. The following LINE clause sequence alternatives may be used in describing a report footing:
      `{NP A | NP A R | A | A R | R}`
   b. The LINE clause with the NEXT PAGE phrase specified is used solely for page advance.Therefore, the first absolute LINE clause must appear with, or prior to, the description of the first printable item, in order to permit positioning to the first line of the report group.

Example 10-4

```
01 DETAIL-GROUP TYPE DETAIL LINE NEXT PAGE.
   02  LINE 10 COLUMN 1  PIC X(10) VALUE "1ST ITEM".
   02           COLUMN 15 PIC X(4)  SOURCE CARD-FIELD-1.
   02  LINE 12 COLUMN 1  PIC X(10) VALUE "2ND ITEM".
   02           COLUMN 15 PIC 9(5)  SOURCE-WORK-FIELD-1.
```

The two-line detail group is printed on lines 10 and 12 of a page not yet used for other body groups, because the LINE clause sequence is NP A.

Example 10-5

```
01  DETAIL-LINE LINE PLUS 1 TYPE DETAIL.
    02  COLUMN 2 GROUP INDICATE PIC A(9) SOURCE FIELD-NO-1.
```

This example shows a report group whose description includes only one relative LINE clause. The report group will be printed on the line resulting from the present position being advanced by one line. If the report group is to be printed as the first body group on that page, and the present positioning has not gone beyond the integer-d region boundary (FIRST DETAIL phrase if the PAGE LIMIT clause), the report group is printed to the line whose number is integer-d.

10.2.12 NEXT GROUP clause

Function

The NEXT GROUP clause indicates to the Report Writer the particular line on a page to which the paper is to advance after printing the last line of the report group containing the NEXT GROUP clause, thereby causing the paper to be prepositioned for the next (chronologically speaking) report group to be printed (minimum distance from the next report group).

Format

```
NEXT GROUP IS {integer-1 | PLUS integer-2 | NEXT PAGE}
```

Syntax rules

1. integer-1 and integer-2 must be specified as unsigned integers equal to or greater than 1 . Their value must not exceed 999.
2. The NEXT GROUP clause must not be specified in a report group description entry unless a LINE clause has been specified.
3. A NEXT GROUP clause with integer-1 phrase is called an absolute NEXT GROUP clause. After printing the last line of the associated report group, the Report Writer advances the paper to the line whose number is specified by integer-1. This may also involve a page advance, printing the page footing on the previous page and the page heading on the new page (for body groups only).
4. A NEXT GROUP clause with PLUS integer-2 phrase specified is called a relative NEXT GROUP clause. Positioning is advanced by integer-2 lines from the last print line of the associated report group.
5. A NEXT GROUP clause with NEXT PAGE phrase specified indicates that the current page is considered to be full; that is, after the associated report group is printed, a page advance is executed (automatic generation of page footing and page heading for body groups).
6. If in a USE BEFORE REPORTING procedure of a control footing, the PRINT SWITCH special register is set to 1 by a MOVE statement, the function of the NEXT GROUP clause is suppressed by the Report Writer Control System.

General rules

1. The NEXT GROUP clause may appear only in a 01-level report group description entry.
2. Rules for the report heading
   a. If the report heading is to appear on a page by itself, its description must contain the NEXT GROUP clause with the NEXT PAGE phrase.
   b. If the report heading is not to appear on a page by itself, the following rules must be observed:
      - The NEXT GROUP clause must not include the NEXT PAGE phrase.
      - integer-1 of an absolute NEXT GROUP clause must indicate a line number greater than the one assigned to the last print line of the report heading.
      - An absolute or relative NEXT GROUP clause must be selected in such a way that the page heading as the next report group can still be contained in its specific region. A positioning to the line with the number integer-d (see "PAGE LIMIT clause") or even to a line which has a higher number owing to the NEXT GROUP clause is illegal.
3. Rule for the page heading

The NEXT GROUP clause must not be used in describing the page heading.

4. Rules for body groups
   a. integer-1 of an absolute NEXT GROUP clause must be greater than or equal to integer-d yet less than or equal to integer-f (see "PAGE LIMIT clause").

      If the line number integer-1 of the absolute NEXT GROUP clause is less than or equal to the number of the line that was printed as the last line of the body group whose description includes the NEXT GROUP clause, the Report Writer executes a page advance (automatic generation of page footing and page heading included) and positions on the line specified by integer-1.

   b. If a relative NEXT GROUP clause would advance the paper beyond the lower boundary specified by integer-f (see "PAGE LIMIT clause"), the Report Writer executes a page advance (generation of page footing and page heading included) and positions on the upper region boundary specified by integer-d (see "PAGE LIMIT clause").

   c. The NEXT GROUP clause with the NEXT PAGE phrase specified indicates that no further body group is to be written on the current page. The Report Writer positions on the upper region boundary integer-d of the next page (including the creation of page footing and page heading).

   d. If several control footings are created in immediate succession as a result of a control break, the Report Writer can be instructed to execute the function of the NEXT GROUP clause only for the particular control footing associated with the level of hierarchy at which the control break occurred (see section "CBL-CTR special register").

5. Rule for the page footing

   The NEXT GROUP clause must not be used in describing the page footing.

6. Rule for the report footing

   The NEXT GROUP clause must not be used in describing the report footing.

Example 10-6

```
01  LINE PLUS 2 NEXT GROUP PLUS 1.
    TYPE CONTROL FOOTING WDAY.
    02...
```

The NEXT GROUP clause causes the Report Writer to advance only one line after the above control footing is created.

10.2.13 PICTURE clause

Format

```
{PICTURE | PIC} IS character-string
```

Syntax rule

1. The PICTURE symbol N is not permitted.

For further syntax rules and general rules on PICTURE, see section "PICTURE clause".

10.2.14 SIGN clause

Function

The SIGN clause specifies the position and the mode of representation of the operational sign for numeric data items.

Format

```
[SIGN IS] {LEADING | TRAILING} [SEPARATE CHARACTER]
```

Syntax rules

1. The SIGN clause may be specified only for a numeric data description entry whose PICTURE contains the character S.
2. The data description entries to which the SIGN clause applies must be described, explicitly or implicitly, as USAGE IS DISPLAY.
3. If the SIGN clause is specified in a report group description entry, the SIGN clause must contain the SEPARATE CHARACTER phrase.
4. The SIGN clause specifies the position and the mode of representation of the operational sign. It applies only to numeric data description entries whose PICTURE contains the character S. The S indicates the presence, but not the mode of representation, of the operational sign.
5. A numeric data description entry whose PICTURE contains the character S, but to which no SIGN clause applies, has an operational sign, but neither the representation nor, necessarily, the position of the operational sign is specified by the character S. (For representation of the operational sign see section "USAGE clause").

General rules

1. The following rules apply with respect to the required SEPARATE CHARACTER phrase:
   a. The letter S in a PICTURE character-string is counted in determining the size of the item.
   b. The operational sign will be presumed to be the leading (or, respectively, trailing) character position of the elementary numeric data item; this character position is not a digit position.
   c. The operational signs for positive and negative are the standard data format characters + and -, respectively.
2. Every numeric data description entry whose PICTURE character-string contains the character S is a signed numeric data description entry. If a SIGN clause applies to such an entry and conversion is necessary for purposes of computation or comparisons, conversion takes place automatically.

10.2.15 SOURCE clause

Function

The SOURCE clause identifies the data item whose contents the Report Writer (by an implicit MOVE statement) will move to the printable item whose description includes the SOURCE clause, when this item is to be printed.

Format

```
SOURCE IS identifier
```

Syntax rule

1. Any identifier defined in any section of the Data Division may be specified in a SOURCE clause. In the case of the REPORT SECTION, however, a SOURCE clause may use only the PAGE-COUNTER special register, the LINE-COUNTER special register, or a sum counter if it is associated with the report in whose description the SOURCE clause appears.

General rules

1. The description of an elementary item containing a SOURCE clause must also contain a COLUMN clause, that is, the field must be printable.
2. The SOURCE clause in conjunction with the COLUMN clause generates an implicit MOVE statement. For this MOVE statement, the source (or sending) field is defined by the identifier from the SOURCE clause. The receiving field is the printable item whose description includes the SOURCE clause. The picture strings of the PICTURE clauses of the two fields must adhere to the rules for sending items in the MOVE statement (see section "MOVE statement").
3. As the SOURCE clause can never change the value of the data item it specifies, it may also refer to a control data item. Generally, the system prints the value of the sending field as it is when the MOVE statement is executed (creation of the associated report group). If the previous values of the control break data items are in control footings, Function 1 of the CBL-CTR special register (see section "CBL-CTR special register") should be used.

Example 10-7

```
FILE SECTION.
    ...
    02 DEPT  PIC XXX.
    ...
REPORT SECTION.
    ...
    02 COLUMN 19 PIC XXX SOURCE DEPT.
    ...
```

The SOURCE clause has the effect that the value in the data item DEPT is moved to the printable item concerned when the associated report group is printed.

10.2.16 SUM clause

Function

The SUM clause instructs the Report Writer to create and (immediately or subsequently) print arithmetic sums of data items selected from the detailed information that constitutes the report. The SUM clause indicates to the Report Writer the addends which are to be used for summation. In addition, this clause provides a numeric item, which is generated automatically, for the accumulation of the addends. This field, i.e. the sum counter, is also the sending field (source item) for the implicit MOVE statement used to edit the printable field whose description contains the SUM clause.

Format

```
SUM {identifier-1}... [UPON data-name-1]
    [RESET ON {data-name-2 | FINAL}]
```

Syntax rules

1. An identifier used as an addend in a SUM clause must be defined in the FILE SECTION, WORKING-STORAGE SECTION, LOCAL-STORAGE SECTION, LINKAGE SECTION or REPORT SECTION. From the REPORT SECTION, only the name of a sum counter may be referenced as an addend of a SUM clause.
2. Each identifier used as an addend in a SUM clause must represent a numeric data item.
3. data-name-1 is permitted only as the name of a detail report group that is defined in the current report description.
4. FINAL or data-name-2 must be specified in the CONTROL clause of the current report description.
5. identifier-1... specify the items to be accumulated in the sum counter.
6. The UPON phrase has the effect that the specified addends are added up only when a GENERATE statement is executed referencing the very detail group defined in the UPON phrase (see "Use of the UPON phrase").
7. The RESET phrase overrides the standard reset function of the sum counter to zero (see "Use of the RESET phrase").

General rules

1. The SUM clause can only be specified in control footing descriptors.

   Assume the following excerpt from a report:

   ```
   JANUARY 02 B10                  2 A   3.00
              B12                  1 A   4.00
              B12                  3 C  17.00
   PURCHASES & COST FOR 1-02       6   $24.00
   ```

   Printed here are the detailed cost figures and their sum for January 2.

   The detailed cost figures shown on the first three print lines were printed as a result of a GENERATE statement referring to the following detail group:

```
01 DETAIL-LINE TYPE DETAIL.
   02  ...
   .
   .
   .
   02  COLUMN 50 PICTURE ZZ9.99 SOURCE COST.
```

This GENERATE statement was executed three times without control break, causing the detail report group to be printed three times in succession. The data item by the name of COST, described in the FILE SECTION of the Data Division as:

`02 COST PICTURE 999V99.`

supplied the detailed cost figures.

When that GENERATE statement was executed once more, the value of the control data item DAY had been changed from 2 to a different value. The following control footing:

```
01 ... TYPE CONTROL FOOTING WDAY.
   02  ...
   .
   .
   .
   02  SUM-DAY COLUMN 49 PICTURE $$$9.99 SUM COST.
```

was generated as the fourth line of the above partial report, with the contents of sum counter SUM-DAY printed starting at column 49.

Summation took place as follows:

During compilation, the compiler created a sum counter (named SUM-DAY) in response to the SUM clause. At object time, when executing the appropriate INITIATE statement, the Report Writer resets the sum counter to zero. Then, each time the GENERATE DETAIL-LINE statement was executed, the current value of the data item COST was added to the contents of the SUM-DAY counter. Since the Report Writer performs this summation (see under 5. "Detail incrementing") immediately following the control break test and the action resulting from that test, the sum of the cost figures for January 2 is printed out with the control group footing for DAY. When the control footings are created, the sum counter is finally reset to zero, as the SUM clause contains no RESET phrase.

2. Use of the PICTURE clause

   If the description of a data item, within a given report group entry, includes a SUM clause, the associated PICTURE clause describes not only the data item but also the sum counter which the compiler will establish due to the SUM clause. The data item, if printable, is used for printing out the contents of the associated sum counter. The PICTURE clause must define the data item as a numeric or numeric-edited data item, where editing symbols for sum counters will be ignored.

3. Use of the sum counter

   If the data item entry that contains a SUM clause has a data-name immediately following the level-number, that data-name is the name of the internal sum counter, which can thus be accessed by the programmer (for example, for rounding its contents prior to printing). The sum counter is a compiler-generated data item whose USAGE is COMP-3 and whose numeric characteristics are described in the specified PICTURE clause.

4. Types of summation

The programmer can specify three types of summation:
detail-incrementing, rolling forward, and crossfooting.

5. Detail-incrementing

   The time at which the Report Writer adds up an addend (identifier-1... from the SUM clause) in the related sum counter depends on the addend itself.

   The addends used for detail-incrementing are those which are not themselves sum counters, in other words, are defined outside the REPORT SECTION.

   Detail-incrementing is the basis for the other two types of summation. The term "detailincrementing" derives from the fact that typically the addends involved in it are printed with the detail groups of the report.

   Detail-incrementing occurs each time that GENERATE statements are executed. Therefore, the programmer must ensure that the operands used for detail-incrementing contain the required values at the time that GENERATE statements are executed. If a SUM clause uses the UPON phrase, the addends in that SUM clause are added into their sum counter only when this detail-incrementing operation takes place in executing a GENERATE statement referring to the same detail group as the UPON phrase (there is, therefore, no point in using the UPON phrase for a summary report). However, if the SUM clause does not include the UPON phrase, then those addends which are not defined as sum counters are added to their related sum counters when any GENERATE statement for the report is executed (detail-incrementing).

   The Report Writer performs detail-incrementing only after taking certain actions as regards control break (test; creation of the control footings and headings if test is positive). This control break processing also includes resetting the sum counters to zero after creating the control footing whose description contains the corresponding SUM clauses (see "Use of the RESET phrase"). This ensures that the printed sum will contain only the values of the addends for the particular series of detail groups which is concluded by the associated control footing (for example, the sum of the cost figures for January 2).

6. Rolling-forward (hierarchical summation)

   The prerequisite for this kind of summation is that a SUM clause of a control footing must specify as an addend at least one sum counter which was defined as the result of a SUM clause of a hierarchically lower (that is, less inclusive) control footing. Therefore, rolling-forward cannot be designated unless a report description includes at least two control footings whose descriptions each contain at least one SUM clause.

   The contents of a sum counter which is specified as an addend in the SUM clause of another control footing will be added, at the time the associated (hierarchically lower) control footing is generated, to the contents of the sum counter in whose SUM clause it appears as an addend.

   Example 10-8 illustrates rolling forward:

Example 10-8

```
01 ... TYPE CONTROL FOOTING MONTH.
   02  ...
   .
   .
   02  SUM-MONTH COLUMN 46 PICTURE $$$9.99 SUM
       SUM-DAY.
```

In the above control footing description, the rolling-forward function is specified in conjunction with the following control footing description (see "SOURCE clause"):

```
01 ... TYPE CONTROL FOOTING WDAY.
   02  ...
   .
   .
   02  SUM-DAY COLUMN 49 PICTURE $$$9.99 SUM COST.
```

For each creation of the control footing with the control item DAY, the Report Writer adds the contents of the sum counter SUM-DAY to the contents of the sum counter SUM-MONTH, before resetting SUM-DAY to zero. If either a control break or execution of a TERMINATE statement causes the control footing to be generated with MONTH, the sum counter SUM-MONTH (before resetting to zero) will contain the sum of all daysums (values of SUM-DAY at summation times) of the current month.

7. Crossfooting (adding hierarchically equal sums)

   This type of summation takes place when a SUM clause contains, as addends, the names of sum counters defined by other SUM clauses in the same control footing. Normally, such addends are sum counters whose values are created through detail-incrementing.

Example 10-9

```
01 MINOR TYPE CONTROL FOOTING...
   02  SUM-1 SUM WORKING-ITEM-1...
   02  SUM-2 SUM WORKING-ITEM-2...
   02  SUM   SUM SUM-1 SUM-2...
```

WORKING-ITEM-1 and WORKING-ITEM-2 are data items defined in the WORKING-STORAGE SECTION of the Data Division. Sum counter SUM accumulates the values of SUM-1 and SUM-2, previously generated through detail-incrementing.
The Report Writer performs crossfooting just before printing the control footing concerned. If more than one SUM clause requires such addition, the order of execution is determined by the sequence of these SUM clauses. This order is essential to the result of the addition.

Obviously, this type of addition is carried out before rolling-forward, thereby ensuring that sums hierarchically equivalent in summation may also be rolled forward.

Example 10-10

```
    ...
    CONTROLS ARE STATE, CITY.
    ...
01  LINE PLUS 2 TYPE CONTROL FOOTING CITY.
    02  SUM-1 SUM MALES...
    02  SUM-2 SUM FEMALES...
    02  SUM-CITY SUM SUM-1, SUM-2...
01  LINE PLUS 1 TYPE CONTROL FOOTING STATE.
    02 SUM-STATE SUM SUM-CITY...

    ...
```

The values accumulated in sum counter SUM-CITY by crossfooting the values from the hierarchically equivalent sum counters SUM-1 and SUM-2 (detail-incrementing) are rolled forward in sum counter SUM-STATE (the control footing with STATE is higher in hierarchy than the control footing with CITY). This is possible only because the sum counter SUM-CITY contains the proper value before rolling-forward takes place in the sum counter SUM-STATE.

8. Mixing operands
   A SUM clause that does not contain an UPON phrase may include one or more of each of the following kinds of operands (= addends):
   - Operands defined in the FILE SECTION, WORKING-STORAGE SECTION, LOCAL-STORAGE SECTION or LINKAGE SECTION.
   - Operands defined as sum counters in a hierarchically inferior control footing.
   - Operands defined as sum counters in the same control footing (whose description contains the SUM clause). Summing for each of the above kinds of operands occurs at the times indicated in the preceding discussions.
9. Use of the UPON phrase
   - When an UPON phrase is used in a SUM clause, all addends of that clause must be defined outside the REPORT SECTION; that is, they may be defined only in the FILE SECTION, WORKING-STORAGE SECTION, LOCAL- STORAGE SECTION and LINKAGE SECTION.
   - The UPON phrase has the effect of preventing detail-incrementing of the addends from the present SUM clause, unless a GENERATE statement is executed specifying the detail report group indicating in the UPON phrase. A detail-incrementing caused by any other GENERATE statement will not, therefore, affect any of the addends in the SUM clause in question.

Example 10-11

```
DATA DIVISION:

FILE SECTION.
FD  INFILE...
...
    RECORDS ARE MUELLER, MEIER.
01  MUELLER PICTURE 999.
01  MEIER PICTURE 9999.
REPORT SECTION.
```

```
...
01  MUELLER-DETAIL TYPE DETAIL.
    02  LINE PLUS 1 COLUMN 1 PIC 999 SOURCE MUELLER.
01  MEIER-DETAIL TYPE DETAIL.
    02  LINE PLUS 1 COLUMN 1 PIC 9999 SOURCE MEIER.
...
01 MINOR TYPE CONTROL FOOTING...
   02  SUMME-1 SUM MUELLER UPON MUELLER-DETAIL...
   02  SUMME-2 SUM MEIER UPON MEIER-DETAIL...
   ...
PROCEDURE DIVISION.
...
    GENERATE MUELLER-DETAIL.
    ...
    GENERATE MEIER-DETAIL.
```

Because MUELLER and MEIER are the names of two different records on the same file, they cannot be available in memory concurrently. When a MUELLER record is read, the statement GENERATE MUELLER-DETAIL is executed; at that time, the current value of MUELLER is added to sum counter SUM-1. The present value of MEIER, on the other hand, is not added to the contents of sum counter SUM-2 at this time. When a MEIER record is read, the statement GENERATE MEIER-DETAIL is executed; at this time, the detail-incrementing occurs in sum counter SUM-2, and not in SUM-1.

10. Use of the RESET phrase
    - Only a data-name (FINAL included) supplied in the CONTROL clause of the same report may be used in a RESET phrase. Moreover, the control data item must be at a higher level in hierarchy than the control footing whose description contains the RESET phrase.
    - Normally, the Report Writer resets a sum counter to zero immediately after printing the control footing in whose description it is defined. A sum counter whose SUM clause contains the RESET phrase will be reset to zero only at a time when the (explicit or implicit) control footing for the control data item (or FINAL) that appears in the RESET phrase, is (or would be) created. Thus, the RESET phrase serves the purpose of creating a total for the specified hierarchical level.

Example 10-12

```
01 ... TYPE CONTROL FOOTING DAY.
   02  ...
   .
   .
   02  COLUMN 65 PIC $$$$9.99 SUM COST RESET ON FINAL.
01 ... TYPE CONTROL FOOTING FINAL.
   02  ...
   .
   .
   02  COLUMN 45 PIC $$$$9.99 SUM SUM-DAY.
```

Because the SUM clause in the description of the control footing with the control item DAY contains the phrase RESET ON FINAL, the current value of the associated sum counter is printed every time the control footing of DAY is generated, without ever resetting the sum counter to zero. Only when the control footing for FINAL is generated will the sum counter be reset to zero. Therefore, each printed control footing for DAY shows the running cost figures from the first detail report group of the report (1st day) through to the last detail report group written before the current control heading.

A control data item that appears in a RESET phrase does not have to be associated with a control footing. A sum counter will be reset, as mentioned earlier, even when no control footing exists for a control item specified in a RESET entry.

11. Actions taken by the Report Writer
When generating a control footing, the Report Writer executes the following steps (schematically speaking, because steps may be omitted):
    a. Adding hierarchically equivalent sums (crossfooting).
    b. Execution of the USE BEFORE REPORTING procedures for the control footing (see section "USE BEFORE REPORTING statement").
    c. PRINT SWITCH test.
    If the value of the PRINT SWITCH special register is 1, step d) is skipped, i.e. step e) immediately follows step c) after resetting the special register to zero. Otherwise, step d) comes next.
    d. Creation of the control footing (if printable).
    e. Hierarchical incrementing (rolling-forward).
    f. Any sum counters of the control footings whose SUM clauses do not contain RESET phrases are reset to zeros by implicit MOVE statements. The same applies to all those sum counters of the other control footings whose SUM clauses each contain one such RESET phrase which is referring to the control data item associated with the current (i.e. newly-created) control footing.

10.2.17 TYPE clause

Function

The TYPE clause indicates the type of the report group in whose description it appears; that is, it defines the functional characteristics of the report group, thereby also specifying the circumstances under which the Report Writer will generate that report group.

Format

```
TYPE IS {  {REPORT HEADING | RH}
         | {PAGE HEADING | PH}
         | {CONTROL HEADING | CH} {data-name-1 | FINAL}
         | {DETAIL | DE}
         | {CONTROL FOOTING | CF} { data-name-2 | FINAL }
         | {PAGE FOOTING | PF}
         | {REPORT FOOTING | RF}
        }
```

Syntax rules

1. RH is the abbreviation for REPORT HEADING.

   PH is the abbreviation for PAGE HEADING, etc.
2. FINAL, data-name-1 and data-name-2 must be defined in the CONTROL clause of the associated report description (RD) entry.
3. REPORT HEADING indicates the report group which is created only once per report as the first report group in that report. It is created automatically when the first GENERATE statement is executed.
4. PAGE HEADING indicates a report group which is created as the first group on every page. A page which is wholly reserved for the report heading or report footing is not assigned a page heading. If a report heading is present but does not appear on a page by itself, the page heading is generated as the second group on the first page of the report.
5. CONTROL HEADING indicates report groups which are written in series when executing the (chronologically) first GENERATE statement and upon every control break. Each control heading is associated (by FINAL or data-name-1) with one, and only one, hierarchical level, which determines the order in which the control headings are printed (see section "CONTROL clause").
6. DETAIL indicates those report groups which are written because they are supplied in a GENERATE statement. The name of a detail group must be unique throughout the report.
7. CONTROL FOOTING indicates those report groups which are written in series upon every control break and when the TERMINATE statement is executed. Each control footing is associated (by FINAL or data-name-2) with one, and only one, hierarchical level, which determines the order in which the control footings are printed (see section "CONTROL clause").
8. PAGE FOOTING indicates the report group which is generated as the last group of each report page. Any page that is wholly reserved for the report heading or report footing is not assigned a page footing. If the report footing does not appear on a page by itself, the page footing is generated as the last but one group on the last page of the report.
9. REPORT FOOTING indicates a report group that is produced only once, as the last group in the report. The report footing is the last group printed when the TERMINATE statement is executed.

General rules

1. Rules for the report heading
   a. Only one report heading may be defined for each report.
   b. The report heading may be printed on a whole page by itself. The NEXT GROUP clause with the NEXT PAGE phrase specified may be used to make sure that the same page will contain no further group besides the report heading (see section "NEXT GROUP clause").
2. Rules for page heading and page footing
   a. Only one page heading and one page footing may be defined for each report.
   b. Normally, the page heading appears as the first report group and the page footing appears as the last report group on every page of the report, with the following exceptions:
      - On the first page of the report, the page heading is preceded by the report heading in those cases where the report heading is not to appear on a page by itself.
      - The page footing of the last page of the report is followed by the report footing in those cases where the report footing is not to appear on a page by itself.
3. Rules for control headings and control footings
   a. Only one control heading and one control footing may be specified for each level of hierarchy.
   b. Control headings and control footings are printed at the following times:
      - When the first GENERATE statement for a report is executed, the Report Writer prints the entire hierarchy of control heading report groups (in the order highest level first, lowest level last) before printing the detail group as the immediate product of the GENERATE statement.
      - If the Report Writer detects a control break when executing one of the (chronologically) next GENERATE statements, it creates the appropriate series of control footings and control headings before the detail groups directly referenced by the current GENERATE statement (see section "GENERATE statement" and section "CONTROL clause").
      - When production of a report ends by execution of the TERMINATE statement, the Report Writer generates all control footings in increasing order of hierarchy levels.
   c. FINAL, data-name-1 or data-name-2 specified in the TYPE clause of a control heading or control footing must appear in the CONTROL clause of the associated report description entry.
   d. A control heading or control footing associated with FINAL may only be generated once for each report, as the first body group of the report (execution of the first GENERATE statement) or as the last (execution of the TERMINATE statement).
4. Rules for detail report groups

   The Report Writer generates a detail report group only when it is specified in a GENERATE statement and when this GENERATE statement is executed. At least one detail report group must be defined for each report. This is true, even if the detail report group is not explicitly used for generating the report; that is, even if it is mainly a summary report without going into details (that is, detail report groups) (see section "GENERATE statement").
5. Rules for the report footing
   a. Only one report footing may be defined for each report, and it is printed as the last report group in the report.
   b. The report footing may appear on a page by itself, by specifying the first LINE clause with the NEXT PAGE phrase in the description of the report footing (see section "LINE clause").
6. Sequence of report groups within a report

a. No report group of a report may be printed either before the report heading or following the report footing.

b. The (schematic) sequence of the heading and footing groups for a report is this:

   Report heading (may only appear once)
   Page heading
   ...
   Control heading
   Detail
   Control footing
   ...
   Page footing
   Report footing (may only appear once)

c. The control headings are always created in immediate succession in hierarchical order:

   Control heading with FINAL (highest level, may only appear once)
   Control heading at next highest level
   ...
   Control heading at lowest level

d. The control footings are always created in immediate succession in hierarchical order:

   Control heading at lowest level
   Control heading at next lowest level
   ...
   Control footing with FINAL (highest level of hierarchy, may only appear once).

10.2.18 USAGE clause

Format

```
[USAGE IS] DISPLAY
```

Syntax rule

1. The USAGE clause is only used in the REPORT SECTION to specify the data format of printable data elements.

For further syntax rules and general rules on USAGE IS DISPLAY, see section "USAGE clause".

10.2.19 VALUE clause

Format

```
[VALUE is literal]
```

Syntax rule

For syntax rules and general rules see format 1 in section "VALUE clause".

10.3 Language elements of the Procedure Division

10.3.1 GENERATE statement

Function

The GENERATE statement directs the Report Writer to produce a portion of the report in accordance with the report description specified in the REPORT SECTION of the Data Division.

Format

```
GENERATE {data-name | report-name}
```

Syntax rules

1. data-name must be defined in the REPORT SECTION of the Data Division, as the name of a detail report group (01-level entry).
2. The report-name must be defined as such (RD entry) in a report description entry of the REPORT SECTION in the Data Division.
3. As the result of a GENERATE statement referring to a detail report group, part of the report printed (see rule 5).
4. As the result of a GENERATE statement referring to a report-name, part of a report is printed (see rule 8).
5. As the result of a GENERATE statement referring to a detail report group, the Report Writer generates part of the report. The composition of this portion is indicated in syntax rules 6 and 7. In addition, various summations are generally executed (see section "SUM clause").
6. When executing the (chronologically) first GENERATE statement (relative to the execution of the related INITIATE statement), the following report groups (if defined) will be printed, provided this statement specifies a detail report group:
   a. report heading
   b. page heading
   c. all control headings from the highest to the lowest levels of hierarchy, and
   d. the detail report group specified in the GENERATE statement.
7. If a GENERATE statement specifying a detail report group is not executed as the (chronologically) first statement, the Report Writer will first check whether a control break occurred. If so, the following report groups are written in the order stated:
   a. All control footings from the lowest level up to, and including, the level at which the control break occurred, and all control footings from the control break level down to the lowest level of hierarchy.
   b. The detail report group specified by the GENERATE statement.

   Step a) is not applicable unless there was a control break.
8. As a result of a GENERATE statement referring to a report, the Report Writer takes the same actions, except for the creation of a detail report group, which is not applicable here. No additional actions beyond syntax rules 5, 6 and 7 will be taken.
9. When a page becomes full in the course of printing a report, the Report Writer automatically generates a page advance, preceded by the page footing (if present) of the previous page and followed by the page heading of the new page.

General rules

1. At the time of executing a GENERATE statement which specifies a detail report group, the following information must be available to the Report Writer:
   - All of the SOURCE clause information assigned to the detail report group and all other report groups to be created by the GENERATE statement.
   - The numeric data of those addends the Report Writer needs to accumulate the necessary sums.
2. Summary report printing is meaningful only for those reports for which control footings, too, are defined whose descriptions include SUM clauses.
3. Summary reporting should not be attempted for reports whose descriptions contain more than one detail report group because it does not allow relations to be established to the individual detail report groups.
4. The CBL-CTR special register (see section "CBL-CTR special register") is interrogated by the Report Writer at the time of the (chronologically) first GENERATE statement. By using the MOVE statement to supply one of several defined values to this special register any time between the execution of the INITIATE statement and the (chronologically) first GENERATE statement, the programmer can select one or both of the following Report Writer options:
   - Supply appropriate control data item values to the control footings, the page footing and the page heading.
   - Condition execution of the NEXT GROUP clauses in the control footings (see section "CBL-CTR special register").

10.3.2 INITIATE statement

Function

The INITIATE statement causes the Report Writer to begin the processing of one or more reports.

Format

```
INITIATE {report-name-1}...
```

Syntax rules

1. report-name-1... must be defined by a report description in the REPORT SECTION of the Data Division (RD entry).
2. The INITIATE statement indicates those reports (report-name-1...) which the Report Writer is to start generating.
3. The INITIATE statement instructs the Report Writer to perform the following initialization functions for each named report:
   - Set all sum counters to zero.
   - Set LINE-COUNTER special register to zero.
   - Set PAGE-COUNTER special register to one.

General rules

1. The INITIATE statement does not open the file the named report is associated with. This report file, which is defined as a sequential output file, must therefore be OPENed as OUTPUT before execution of the INITIATE statement.
2. If an INITIATE statement is followed by a second (different) INITIATE statement specifying the same report as the first one, an intervening TERMINATE statement must first be executed for this report.
3. If an INITIATE and a TERMINATE statement were executed for a given report without an intervening GENERATE statement, the TERMINATE statement cancels the INITIATE statement without printing the report (or any portion thereof).
4. Following execution of the INITIATE statement, and prior to execution of the (chronologically) first GENERATE statement, the programmer may select certain optional functions of the Report Writer by using the MOVE statement to supply the proper value to the CBT-CTR special register (see section "CBL-CTR special register").

10.3.3 TERMINATE statement

Function

The TERMINATE statement instructs the Report Writer to complete the processing for the specified reports.

Format

```
TERMINATE {report-name-1}...
```

Syntax rules

1. report-name-1 must be defined by a report description in the REPORT SECTION of the Data Division (RD entry).
2. The TERMINATE statement indicates those reports whose processing the Report Writer is to complete.
3. For each report specified in the TERMINATE statement, the Report Writer performs the following steps in the stated order:
   a. All control footings are created as if they were to be printed as the result of a control break at the highest level of hierarchy (obviously, this implies creating the page footing and the page heading if a page advance is required, and the execution of summations).
   b. The page footing is generated.
   c. The report footing is generated.

   However, the above actions are not taken if no GENERATE statement was executed for the report between the INITIATE and TERMINATE statements.

General rules

1. Each TERMINATE statement for report-name-1 must be chronologically preceded by an INITIATE statement for report-name-1.
2. Since the TERMINATE statement does not close the associated report file, it must be chronologically followed by a CLOSE statement. Each INITIATEd report in a particular file must be TERMINATEd before a CLOSE statement is executed for that report file.

10.3.4 USE BEFORE REPORTING statement

Function

The USE BEFORE REPORTING statement introduces Procedure Division statements that are to be executed just before the specified report group is printed by the Report Writer.

Format

```
USE [GLOBAL] BEFORE REPORTING report-group-name.
```

Syntax rules

1. The report-group-name must be defined as a name (data-name) in a 01-level entry of a REPORT SECTION report group entry in the Data Division. Any report group type except the detail report group may be specified in the USE BEFORE PRINTING statement.
2. The report-group-name identifies the report group for which the USE declaratives (USE BEFORE REPORTING procedures) following the USE BEFORE REPORTING statement are to be carried out.
3. The USE BEFORE REPORTING statement itself is never executed; it merely defines the conditions calling for the execution of the subsequently declared USE procedures.
4. The USE procedures declared for a report group are executed immediately before that report group is created.
5. No more than 39 USE BEFORE REPORTING statements may appear in the Procedure Division.
6. A USE procedure with a GLOBAL phrase is allowed only if the corresponding FD or RD entry contains the GLOBAL attribute and if the USE procedure is declared in the same program as the associated FD or RD entry. More details on using the GLOBAL clause can be found in "Format 3 USE statement with GLOBAL phrase" in section "USE statement".

General rules

1. The name of a given report group may be specified in one declarative section only.
2. The INITIATE, GENERATE, EXIT PROGRAM, GOBACK, CANCEL and TERMINATE statements must not appear in a paragraph within any declarative section.
3. A USE BEFORE REPORTING procedure must not alter the value of any control data item, nor any value of the subscripts used by the Report Writer to access the control data item.
4. The rules on references between a USE BEFORE REPORTING declarative and the remainder of the Procedure Division are the same as for other USE procedures.
5. The following are typical applications of USE BEFORE REPORTING procedures:
   a. Suppressing the printing of a report group:
   If the statement MOVE 1 TO PRINT-SWITCH is executed in a USE BEFORE REPORTING declarative, the Report Writer will not print the report group the USE procedure was associated with. Since the Report Writer always resets the PRINT-SWITCH special register (see section "PRINT-SWITCH special register") to zero immediately after suppressing the report group, this register must be set to 1 again each time that printing is to be suppressed; with report groups that are written automatically, this can only be done by means of USE BEFORE REPORTING declaratives.
   b. Modifying the contents of a data item specified in a SOURCE clause: In editing a line with a printable item whose description includes a SOURCE clause, the contents of the item defined by the SOURCE clause is transferred to the printable item by means of an implicit MOVE statement. A USE BEFORE REPORTING

procedure may be used to modify the contents of the sending field just prior to the execution of the implicit MOVE statement.

c. Rounding sum counters:
If the value of a sum counter is to be rounded before it is transferred to the associated print item for editing, by an implicit MOVE statement, this can only be achieved by means of a USE BEFORE REPORTING declarative for the control footing in which the sum counter was defined.

d. If special actions depending on the level within the control hierarchy at which a control break has occurred are to be taken before writing a page heading, control heading, control footing, or page footing, this can only be effected via USE BEFORE REPORTING declaratives for this report group, owing to their automatic generation (see section "SOURCE clause" and section "CBL-CTR special register").

10.4 Special registers of the Report Writer

10.4.1 LINE-COUNTER special register

LINE-COUNTER is a special register which is automatically defined for each report described in the REPORT SECTION of the Data Division. The internal COBOL notation of this special register is PICTURE S999 USAGE COMPUTATIONAL.

The Report Writer always uses the LINE-COUNTER register to control the vertical spacing of the groups to be printed in a report. For this reason, the LINE-COUNTER register must not be changed by any Procedure Division statement.

As regards the PAGE LIMIT, NEXT GROUP, and LINE clauses, the special register LINE-COUNTER is automatically interrogated and incremented (or set to zero for page advance) by the Report Writer. The INITIATE statement resets the LINE-COUNTER register to zero.

The LINE-COUNTER may be used both in the REPORT SECTION and in the Procedure Division. As far as the REPORT SECTION is concerned, LINE-COUNTER may only be specified in SOURCE clauses as indicated below:

```
SOURCE IS LINE-COUNTER [OF report-name]
```

As the Report Writer always sets the LINE-COUNTER register to the current line (print line or NEXT GROUP positioning), the line number will be displayed on which the printable item associated with the above SOURCE clause is to be written. The LINE-COUNTER special register may be interrogated at any time in the Procedure Division; at the time of interrogation, it will contain the value assigned to it by the Report Writer as a result of the NEXT GROUP clause of the last report group generated before the interrogation (last print line of the report group if no NEXT GROUP clause was specified).

If two or more reports are described in the REPORT SECTION, each explicit reference to a LINE-COUNTER must be qualified by the report-name.

10.4.2 PAGE-COUNTER special register

PAGE-COUNTER is a special register that is defined automatically for each report described in the REPORT SECTION of the Data Division. The internal COBOL notation of this special register is PICTURE S9(7) USAGE COMPUTATIONAL-3.

When the INITIATE statement is executed for a report, the Report Writer increments the PAGE-COUNTER special register to 1 by means of an internal MOVE statement. As soon as the Report Writer issues a page advance - after printing the page footing and before printing the page heading - it increments the current contents of the special register PAGE-COUNTER by 1.

The PAGE-COUNTER register is freely accessible. That is, its contents may be modified as well as interrogated. The most common use of PAGE-COUNTER is for printing page numbers in page headings or page footings. For this purpose, the PAGE-COUNTER register is assigned to a printable item of the associated report, by the following SOURCE clause:

```
SOURCE IS PAGE-COUNTER [OF report-name]
```

Program references to PAGE-COUNTER must be qualified by the report name if the REPORT SECTION describes more than one report.

10.4.3 PRINT-SWITCH special register

PRINT-SWITCH is a special register that is defined automatically for a program whose Data Division includes the REPORT SECTION. The internal COBOL notation of this register is PICTURE S9 USAGE COMPUTATIONAL-3. Only one such special register is defined for each compilation unit.

The purpose of PRINT-SWITCH is to enable the program to suppress the printing of a report group. This is achieved by including the statement MOVE 1 TO PRINT-SWITCH within a USE BEFORE REPORTING procedure for the appropriate report group. The Report Writer checks the contents of PRINT-SWITCH immediately after each execution of a USE BEFORE REPORTING declarative associated with a report group, and suppresses printing if PRINT-SWITCH contains a 1 . After evaluating PRINT-SWITCH, the Report Writer restores its value to 0, thereby preventing any inadvertent suppression of other report groups.

Report group suppression by the Report Writer has the following effects:

- No page spacing as specified by the LINE clauses of the report group.
- No editing of print lines.
- No page spacing as specified by any NEXT GROUP clause of the report group.

The suppression of a report group owing to the PRINT-SWITCH special register implies, in consequence of the above items a) and c), that the LINE-COUNTER special register is not changed.

PRINT-SWITCH should be set only within USE BEFORE REPORTING declaratives. If it is set anywhere else in the Procedure Division, it is generally impossible to predict which report group might be suppressed.

10.4.4 CBL-CTR special register

CBL-CTR (**C**ontrol **B**reak **L**evel **C**ounter) is a special register that is defined automatically for each report described in the REPORT SECTION of the Data Division. The internal COBOL notation of this special register is PICTURE S999 USAGE COMPUTATIONAL.

The Report Writer and the program use this special register to pass information to each other concerning control breaks and the action to be taken when control breaks occur.

CBL-CTR can be used in connection with two different functions. Either, neither, or both of these functions may be used for a given report. For the sake of simplicity, the two functions are known as "function 1" and "function 2". The user program notifies the Report Writer which function it requires for a report. By a MOVE statement, the CBL-CTR register is assigned a value that symbolizes the requested function. Value and function interact as shown below:

Function requested	**MOVE statement**
Function 1	MOVE 1 TO CBL-CTR.
Function 2	MOVE 2 TO CBL-CTR.
Function 1 and 2	MOVE 3 TO CBL-CTR.

The corresponding MOVE statement must be executed after the INITIATE statement for the report but before the (chronologically) first GENERATE statement. This is the only time that the programmer is permitted to change the value of the CBL-CTR special register.

If two or more reports are defined in the REPORT SECTION, each reference to CBL-CTR must be qualified by the report-name.

10.4.4.1 Function 1 of the CBL-CTR special register

This function is invoked by a MOVE statement which places 1 (or 3) in the CBL-CTR special register.
The function consists of two parts; the respective actions are covered in the following rules.

a. **Restoration of the previous values of the control data items in control footings**

Part 1 of function 1 makes previous values of control data items available to control footing SOURCE clauses or control footing USE BEFORE REPORTING declaratives.
SOURCE clauses (or USE BEFORE REPORTING declaratives) in control footings referring to control data items produce a problem. At the time the control footings are printed, some or all of the specified control data items have changed values. However, since the control footings printed because of a control break obviously belong to the precontrol break values, it is often desirable that the previous values (i.e. prior to the control break) should be printed. When function 1 of CBL-CTR is requested, these previous values of the control data items will be obtained by SOURCE clauses (or USE BEFORE REPORTING declaratives) of control footings.

For example, assume that the item MONTH-NAME is defined as a control data item for a report and that the control footing MONTH-FOOTING is defined as follows:

```
01 MONTH-FOOTING TYPE CONTROL FOOTING
                        MONTH-NAME LINE PLUS 1.
   02 COLUMN 10 PIC X(21) VALUE "***** END OF DATA FOR".
   02 COLUMN 33 PIC X(9) SOURCE MONTH-NAME.
```

In this case, the programmer wants the following control footing to be printed after all of the JANUARY data has been printed in the detail lines of the report:

```
***** END OF DATA FOR JANUARY.
```

Since the above control footing was printed because the item MONTH-NAME changed from JANUARY to FEBRUARY, FEBRUARY (the current contents) would be printed rather than JANUARY. By requesting function 1, the programmer can cause the prior value (JANUARY) to be printed instead of FEBRUARY.

b. **Indicating the control break level in CBL-CTR**

Part 2 of function 1 causes the Report Writer to place, in CBL-CTR, a value indicating the control break level in the hierarchy. This is done before control is passed to a USE BEFORE REPORTING procedure for execution. This kind of procedure begins with:

```
section-name SECTION. USE BEFORE REPORTING report-group-name.
```

Indicating the level of the current control break in the hierarchy is accomplished by numbering the control data items consecutively, starting from the highest level in the hierarchy, not including FINAL. Thus, the first (leftmost) data-name of the CONTROL clause is numbered 1, the next is numbered 2, and so on.

For instance, assume that a report description entry contains this CONTROL clause:

```
CONTROLS ARE FINAL STATE COUNTY CITY
```

In this case, STATE is assigned number 1, COUNTY number 2, and CITY number 3.

The meanings of the CBL-CTR values in the USE procedures declared for the page heading and control headings are listed below in table 41.

If, in the above example, the value of the control data item COUNTY changes, the value 2 will be placed in the CBL-CTR register by the Report Writer (provided that the value of CITY has not changed at the same time also).

Value	Meaning
0	Indicates that no GENERATE statement has been executed or that the very first GENERATE statement is being executed.
1-254	Indicates that the control data item with the corresponding number has just changed values, and that actions currently taking place were caused by this control break.
255	Indicates that no control break has occurred (cannot be encountered with control footing USE procedures).

Table 41: CBL-CTR values in USE procedures for page headings and control headings

The meanings of CBL-CTR values in USE declaratives for the page footing and control footings are listed below in table 42.

Value	Meaning
0	Indicates the final stage of processing, i.e. the TERMINATE statement is being executed.
1-254	Indicates that the control data item with the corresponding number has just changed values, and that actions currently taking place were caused by this control break.
255	Indicates that no control break has occurred (cannot be encountered with control footing USE procedures).

Table 42: CBL-CTR values in USE procedures for page footings and control listings

10.4.4.2 Function 2 of the CBL-CTR special register

Function 2 of CBL-CTR is invoked by moving a value 2 (or 3) to CBL- CTR. This causes conditional execution of the NEXT GROUP clauses in the control footings.

Normally, the NEXT GROUP clause is executed immediately after the creation of the report group in whose description it is supplied (vertical spacing). If a control break causes several control footings to be printed in succession, the NEXT GROUP clauses of these control footings lead to inadvertent blank lines. Such blank lines are really intended for spacing between a control footing and a control heading or a detail report group but not to another control footing.

When function 2 of CBL-CTR is requested, the Report Writer makes sure that only the last NEXT GROUP clause is executed, i.e. the NEXT GROUP clause associated with the last control footing generated in the series of control footings printed in succession as a result of a control break. Existing NEXT GROUP clauses of the other control footings of a closed sequence of control footings will be ignored. This is true even if the last control footing of the sequence has no NEXT GROUP clause.

11 XML

11.1 General description

COBOL2000 the reading and analysis of XML documents. This can be done in two different ways:

- On a structure-oriented basis, i.e. the XML document is transformed into a tree. This tree permits data to be positioned on nodes and to be transferred from (multiple) nodes to the COBOL program.
- On an event-oriented basis, i.e. the XML document is processed purely sequentially, and each syntax unit which is detected is made available in special registers for further processing.

The language elements which are available are specifically for the processing type selected. XML documents are actually analyzed and parsed by a separate open source program package called Parser. Further prerequisites must therefore be fulfilled for such programs to run, see the "COBOL2000 User Guide" [1].

Structure-oriented processing

This processing type regards an XML document as a file of the new file organization form 'XML' which can be made available both in a traditional file and in memory. In both cases it requires a file control entry and a file description entry. The record description entries of the FD do not provide the content of the file, but describe the tree structure of the XML document, either as a whole or only of the parts of it which are to be processed. New clauses in a data description entry are used to define the assignment of the data items to the nodes in the XML document’s tree presentation. The actual processing, i.e. the positioning and sequential or random reading of the data is then performed using the statements for file processing which have been extended for this purpose.

Event-oriented processing

This processing type uses no special language elements of its own to describe an XML document. The new statement XML PARSE analyzes and parses an XML document which is made available in memory sequentially. Each syntax unit of the document which is detected is made available to the user program in special registers for further processing. The user program can, if required, obtain a picture of the general structure of the document from this.

Assignment of the new language elements to the processing types

Language elements	Structure-oriented	Event-oriented
File control entry		
Extended ASSIGN format	x	
File organization XML	x	
Access method XML	x	
File description entry, record description entry		
IDENTIFIED clause	x	
COUNT clause	x	
Statements		
OPEN, CLOSE	x	

OPEN DOCUMENT, CLOSE DOCUMENT	x	
START, READ	x	
XML PARSE		x
Exceptional condition EC-XML-CODESET-CONVERSION	x	
Special registers		x

The description of the language elements is based on the presentation of the XML document as a tree and uses terms which are common in the graphical display of tree structures (see also section "Language elements for processing XML").

11.2 Language elements of the ENVIRONMENT DIVISION

In the case of structure-oriented processing, the ASSIGN clause defines whether the XML document is made available in memory or in a physical file. If the document is contained in a physical file, its BS2000/OSD access method is irrelevant. However, the file's record structure is transparent for the program in the form of end-of-line characters. But these characters can also be hidden using options (see the "COBOL2000 User Guide" [1] for details).

11.2.1 FILE-CONTROL paragraph

Format of files with organization XML

```
FILE-CONTROL.
  SELECT clause
  ASSIGN clause
  [ACCESS MODE clause]
  [FILE STATUS clause]
  ORGANIZATION clause .
```

Syntax rules

1. The SELECT clause must be the first clause in the file control entry. All other clauses can be specified in any order.
2. If the GLOBAL clause is specified in the file description entry, the data items referenced in the clauses of the FILE-CONTROL paragraph may not be defined in a LOCAL-STORAGE SECTION.

11.2.1.1 SELECT clause

Function

The SELECT clause is used to give a file a name.

Format

```
SELECT [ OPTIONAL ] file-name-1
```

Syntax rules

1. file-name-1 is the name with which the file is referenced in the compilation unit. Within a program, a factory, an object or a method, file-name-1 may only occur in a SELECT clause.
2. A file description entry (FD) must exist in the DATA DIVISION of the same program of the same program, the same factory, the same object or the same method for file-name-1.

General rule

1. If the OPTIONAL phrase references to an external file, OPTIONAL must be specified in all programs which describe this external file.

11.2.1.2 ASSIGN clause

Function

The ASSIGN clause provides the connection between the file control entry and a file. The file can be assigned to a physical file or be contained in memory.

Format

```
ASSIGN { TO {  literal-1
            | DATA data-name-1
            | data-name-2 LENGTH IS data-name-3 [FOR {ALPHANUMERIC | NATIONAL}]
           }

       | USING data-name-4

      }
```

Syntax rules

1. literal-1 must be an alphanumeric literal and may not be a figurative constant.
2. data-name-1 must be an alphanumeric or national data item.
3. data-name-2 must be an elementary item of the category data-pointer.
4. data-name-3 must be an integer, unsigned elementary item whose definition does not contain PICTURE symbol 'P'.
5. data-name-1, data-name-2 and data-name-3 may not be described in the record description entry of a file with organization XML.
6. data-name-4 must be an alphanumeric data item and may not be defined within the FILE SECTION.
7. The DATA or LENGTH phrase may not be specified if OPTIONAL is specified in the associated SELECT clause.
8. If neither ALPHANUMERIC nor NATIONAL is specified, ALPHANUMERIC is implied.

General rules

1. If literal-1 or data-name-4 is specified, the XML document is contained in a physical file, otherwise it is contained in memory.
2. literal-1 or the content of data-name-4 specifies the link name for the file. Only the first 8 characters are used for literal-1. If the eighth character is a hyphen (-), this is replaced by the hash character (#).
3. An XML document in memory is assigned
   - directly by the data item with data-name-1 which contains the XML document.
   - indirectly by a data pointer data-name-2 which points to a memory area containing the XML document. In this case data-name-3 specifies the length of the memory area in **characters**.
4. The FOR phrase defines whether the memory area containing the XML document is interpreted as an alphanumeric or national characters.
5. The address and length of the memory area containing the XML document are evaluated using the OPEN statement. The content of the memory area is only evaluated when the OPEN DOCUMENT statement executes. Modifications made subsequently have no influence on the evaluated address and length or on the content.

11.2.1.3 ACCESS MODE clause

Function

The ACCESS MODE clause determines the type of access to the XML document.

Format

```
ACCESS MODE IS XML
```

Syntax rule

1. XML may only be specified for a file with organization XML.

General rule

1. If the ACCESS MODE clause is missing and the file organization is XML, ACCESS XML is implied.

11.2.1.4 FILE STATUS clause

Function

The FILE STATUS clause specifies a data item that indicates the status of an XML file operation. In addition, when a further data item is specified, an error code is also made available.

Format

```
FILE STATUS IS data-name-1 [data-name-2]
```

Syntax rules

1. data-name-1 and data-name-2 must be defined in the WORKING-STORAGE SECTION, LOCAL-STORAGE SECTION or LINKAGE SECTION of the DATA DIVISION.
2. data-name-1 must be a two-character alphanumeric data item.
3. data-name-2 must be a six-byte group item with the following format:

```
01 data-name-2.
   02 data-name-2-1 PIC 9(2) COMP.
   02 data-name-2-2 PIC X(4).
```

General rules

1. If the FILE STATUS clause is specified, the I-O status is transferred to data-name-1 following an I/O statement for the file to whose file control entry this clause is subordinate.
2. If specified, data-name-2 is assigned as follows:
   - If data-name-1 contains the value 0, the contents of data-name-2 are undefined.
   - If data-name-1 contains a non-zero value, data-name-2 contains the additional error code. The value 96 in data-name-2-1 indicates that this code is the SIS code (POSIX). The value 231 indicates that this code is the CBX code (error code of the XML parser).
3. The command HELP SIS <content of data-name-2-2> or HELP CBX <content of data-name-2-2> supplies more detailed information on the particular error code.
4. The I-O status is transferred during execution of each OPEN, OPEN DOCUMENT, CLOSE, CLOSE DOCUMENT, READ or START statement that references the specified file, and prior to the execution of each corresponding USE procedure (see "I-O status for XML files" in section "Error handling").

11.2.1.5 ORGANIZATION clause

Function

The ORGANIZATION clause defines the logical structure of a file.

Format

```
[ORGANIZATION IS] XML
```

General rule

1. Specification of XML means that the file contains an XML document.

11.3 Language elements of the DATA DIVISION

COBOL data structure for an XML document

In structure-oriented processing, the hierarchical structure of an XML document is emulated by the level concept of the record description entries in COBOL. In this a data description entry with an IDENTIFIED clause corresponds to a node from the XML tree, i.e. an attribute or the (start) tag of an element in the XML document. End tags play no part here because they are derived implicitly from the hierarchy.

Optionally the following can be defined in the data structure for each node:

- the name of the node,
- the namespace of the node,
- the value of the node,
- if necessary, further subordinate elements in the XML document, also with an IDENTIFIED clause.

In the data structure all such additional data items must be directly subordinate to the data item which corresponds to the node. Only one of these additional data items may neither have an IDENTIFIED clause nor be referenced by an IDENTIFIED clause: the data item for the content. If this data item is the only data item which is subordinate to the node description entry, it can be omitted and its PICTURE clause can also be specified directly in the node description entry.

The data structures of the FD need only describe those nodes from the XML tree which the program wants to process. However, the parent of each described node must also be described.

Assignment and Element Position Vector

The basis for processing an XML document is provided by the **assignment** of nodes from the XML tree to the data description entries in the record description entries of the FD. This definition is stored in the logical information unit **Element Position Vector** (EPV). For each data item which specifies an IDENTIFIED clause, the EPV contains a reference regarding which node from the XML tree is assigned to the data item (i.e. has 'a valid position'), or the fact that no node is assigned (i.e. the 'position is invalid'). Which statement generated the valid position is also noted.

The IDENTIFIED clause in a data description entry determines which nodes from the XML tree can in principle be assigned to this data description entry.

The assignment of a node is possible if all the following conditions are satisfied (see "IDENTIFIED clause" for the various phrases):

- The type of node (element or attribute node) in the XML tree matches the phrase (ELEMENT or ATTRIBUTE) from the IDENTIFIED clause.
- The local name of the node in the XML tree matches the local name specified in the IDENTIFIED clause:
  - If the name is specified by means of BY, both names must be identical except for trailing blanks.
  - If the name is specified by means of USING, any names from the XML tree are regarded as matching.
- The namespace of the node in the XML tree matches the namespace (NAMESPACE) specified in the IDENTIFIED clause. This applies both for an explicit NAMESPACE phrase and for an implicit phrase, i.e. one which was inherited from subordinate data items:
  - If the namespace is specified by means of BY, both names must be identical except for trailing blanks.
  - If the namespace is specified by means of USING, any names from the XML tree are regarded as matching (in particular also the empty namespace).

- If the namespace is specified as NULL, the node must also have an empty namespace in the XML tree.

Assignment procedure

The new statements for processing an XML document execute assignments. The **current assignment procedure** executes in the following steps:

1. At most one node from the XML tree is assigned to precisely one data item in the FD. Which nodes and which data items need to be taken into consideration for this first step depends on the statement concerned.
2. For each data item to which it was possible to assign a node from the XML tree, all the data items which are made directly subordinate to it are examined by means of the IDENTIFIED clause, and an attempt is made to assign each of them one child of the assigned node, starting with the oldest unassigned child.
3. The 'invalid' position is noted for each data item to which it was not possible to assign a node in the previous steps and also for all data items in the EPV which are subordinate to such a data item.

> When assignment takes place, no data is transferred between the XML document and the data description entries in the program.

11.3.1 File description entry

Format File with organization XML

```
FD file-name-1
  [EXTERNAL clause]
  [GLOBAL clause] .
```

Syntax rules

1. file-name-1 must match the file name from a file control entry.
2. Clauses that follow the file-name may appear in any order.
3. The file description entry must be followed by one or more record description entries.
4. Record description entries of files with organization XML may only contain data items of the categories alphabetic, alphanumeric, national or numeric.

11.3.1.1 EXTERNAL clause

Function

The EXTERNAL clause enables a file with organization XML to be defined as external. External XML files can be accessed by any program in which they are described.

Format

```
IS EXTERNAL
```

Syntax rules

1. Names of external files can have a maximum length of 30 characters.
2. The EXTERNAL clause may not be specified for files with organization XML which have the DATA or LENGTH phrase specified in their SELECT clause.

General rules

1. If a file is defined as external, the records in that file are also implicitly external.
2. The following may not be used as names for external files:
   a. external record names from the WORKING-STORAGE SECTION of other compilation units in the run unit,
   b. PROGRAM-ID names of the run unit, except for program names of contained names of a nested program,
   c. names used as entry points in an ENTRY statement,
   d. names that identify an interface (e.g. runtime system names).
3. The effect of the FILE STATUS clause for external files is always local to the program, i.e. the file status is supplied only by I/O operations in the program that contains a corresponding phrase in the file description entry.
4. An external file with organization XML must be described largely identically in all compilation units that wish to access it.
   The table below shows how and to what extent the descriptions must match.

Clause /specification	**In all programs**
Name of the external file	Same to full length (30 characters)
ORGANIZATION clause	XML
ACCESS MODE clause	XML
Record description entries in the FD	Identical, also the order of the record description entries. However, the system only checks whether the hierarchical structure of the IDENTIFIED phrases and the COUNT phrases in the record description entries match, and whether the ATTRIBUTE and ELEMENT, BY and USING phrases in NAMESPACE match.

5. Data items specified in the IDENTIFIED clause under USING and BY or IS which are not contained in a record description entry of the FD are always effective for external files on a local program basis, i.e. they are only supplied with values or used in the program which executes the input operation.

> If a file is defined as external, the associated file name is not implicitly a global name.

11.3.1.2 GLOBAL clause

See section "GLOBAL clause" for the format and rules.

11.3.2 Data description entry

Function

A data description entry describes the attributes of a node from an XML tree or additional data which is connected to the node.

Format 1 for describing a node

```
level-number [data-name | FILLER]
             IDENTIFIER clause
             [COUNT clause]
             [GROUP-USAGE clause]
             [JUSTIFIED clause]
             [PICTURE clause]
             [SIGN clause]
             [SYNCHRONIZED clause]
             [USAGE clause]
             [VALUE clause].
```

Format 2 for describing additional data

```
level-number [data-name | FILLER]
             [GROUP-USAGE clause]
             [JUSTIFIED clause]
             [OCCURS clause]
             [PICTURE clause]
             [SIGN clause]
             [SYNCHRONIZED clause]
             [TYPE clause]
             [USAGE clause]
             [VALUE clause].
```

Syntax rules for both formats

1. The defined data items must be of the category alphabetic, alphanumeric, national or numeric.
2. The clauses may be written in any order, except for the FILLER or data-name phrase, which must immediately follow the level number.

Syntax rules for format 1

1. level-number must be a number from 01 through 49.

Syntax rules for format 2

1. level-number must be a number from 02 through 49 or 88. The rules for condition names apply for level-number 88 (see Data description entry formats).
2. Die DEPENDING ON phrase of the OCCURS clause is not permitted.

3. The OCCURS clause may not be specified for data items which are directly subordinate to a data item for which the IDENTIFIED clause is specified.

All clauses which are not described in the following are assumed to be known, see also the relevant descriptions starting on Data description entry .

11.3.2.1 COUNT clause

Function

For a data item described by means of the IDENTIFIED clause, the COUNT clause generates an additional elementary item which indicates whether a READ statement was able to assign a node from the XML document to the element or attribute which is described by the IDENTIFIED clause.

Format

```
COUNT IN data-name-1
```

Syntax rules

1. The COUNT clause may only be specified for data items which also have a correct IDENTIFIED clause.
2. data-name-1 may not be defined in the same source unit, and it must be possible to reference it uniquely without qualification.

General rules

1. For the data description entry in which it is specified the COUNT clause generates an internal elementary item with the name data-name-1.
2. data-name-1 has the implicit description PIC S9(9) USAGE COMP-5.
3. If the data description entry which the COUNT clause references is subordinate to a data description entry for which the EXTERNAL clause is specified, the COUNT elementary item is also an external data item.
4. If the data description entry which the COUNT clause references contains a GLOBAL clause or is subordinate to such a file or data description entry, data-name-1 is also a global name.
5. After a successful read statement, data-name-1 has the value 1 if a node from the XML document was assigned to the entry which the COUNT clause references. If no node was assigned, data-name-1 has the value 0.
6. An unsuccessful READ statement does not change the value of data-name-1.

> The COUNT elementary item does not indicate the number of repetitions of an element in the XML document.

11.3.2.2 IDENTIFIED clause

Function

The IDENTIFIED clause defines which element or attribute names from the XML document a COBOL data item can assign itself. The NAMESPACE phrase defines the namespaces which must be taken into account.

Format

```
IDENTIFIED {BY {data-name-1 | literal-1} | USING data-name-2} IS {ATTRIBUTE |
ELEMENT}
       [ NAMESPACE  {IS {data-name-3 | literal-2 | NULL} | USING data-name-4}]
```

Syntax rules

1. The IDENTIFIED clause may only be specified in record description entries of files with organization XML.
2. All the group items which are superordinate to the data description entry must also include an IDENTIFIED clause.
3. literal-1 and literal-2 must be alphanumeric or national literals, but not figurative constants.
4. literal-1 must be a valid XML name for an element or an attribute.
5. literal-2 must be a valid XML name for a namespace.
6. data-name-1, data-name-2, data-name-3 and data-name-4 may be qualified.
7. data-name-1, data-name-2, data-name-3 and data-name-4 must be alphanumeric or national data items. Their data description entry may not specify an IDENTIFIED clause.
8. If data-name-1, data-name-2, data-name-3 and data-name-4 are specified in a record description entry of a file with organization XML, they must be directly subordinate to the entry with the IDENTIFIED clause which references them.
9. No data description entries with an IDENTIFIED clause may be subordinate to a data description entry with an ATTRIBUTE phrase.
10. At most one data item which is neither referenced in the IDENTIFIED clause nor is itself an IDENTIFIED clause may be directly subordinate to a data description entry with an IDENTIFIED clause.
11. The table below specifies which combinations of phrases for the element or attribute name and specifications for the namespace are permitted within an IDENTIFIED clause.

IDENTIFIED	**NAMESPACE**		
	IS NULL	**IS data-name-3 / literal-2**	**USING data-name-4**
BY data-name-1 / literal-1	permitted	permitted	forbidden
USING data-name-2	permitted	forbidden	permitted

Table 43: Permitted combination of IDENTIFIED and NAMESPACE phrases in an IDENTIFIED clause

12. If multiple data description entries which have the ELEMENT phrase are directly subordinate to a group item, the possible phrase for IDENTIFIED and NAMESPACE are subject to further restrictions. The same applies for

data description entries which specify the ATTRIBUTE phrase. The table below shows which combinations are permitted in the case of two data description entries. The rows and columns highlighted in gray present combinations which are not permitted for a single data description entry.

First data description entry: IDENTIFIED / NAMESPACE Second data description entry: IDENTIFIED	NAMESPACE	BY NULL	USING NULL	BY IS	USING IS	BY USING	USING USING
BY	NULL	permitted	forbidden	permitted			forbidden
USING	NULL	forbidden	forbidden	forbidden			forbidden
BY	IS	permitted	forbidden	permitted			forbidden
USING	IS						
BY	USING						
USING	USING	forbidden	forbidden	forbidden			forbidden

Table 44: Permitted combination of two data description entries with IDENTIFIED and NAMESPACE phrases

13. If multiple data description entries which all specify the same explicit or implicit NAMESPACE phrase are directly subordinate to a group item, the following applies:
    - The values of literal-1 must be unique for all entries which specify or imply the ELEMENT phrase.
    - The values of literal-1 must be unique for all entries which specify the ATTRIBUTE phrase.

    Neither the representation as an alphanumeric or national literal nor trailing blanks play a role for the uniqueness here.
14. If ATTRIBUTE is not specified, ELEMENT implied.
15. The IDENTIFIED clause may not be used in a type declaration entry.

General rules

1. The IDENTIFIED clause describes which nodes from the XML document are to be assigned to the data item:
    - If ELEMENT is specified, only element nodes are assigned.
    - If ATTRIBUTE is specified, only attribute nodes are assigned.
2. If BY is specified, literal-1 or the content of data-name-1 defines the name which the assigned node is to have. Neither the representation as an alphanumeric or national literal or data item nor trailing blanks are relevant here.
3. If IS is specified, literal-2 or the content of data-name-3 defines the namespace which the assigned node is to have. Neither the representation as an alphanumeric or national literal or data item nor trailing blanks are relevant here.
4. If USING is specified, the assigned nodes may have any names and may be located in any namespaces. When reading is successful, the names of such nodes are transferred to data-name-2, or the associated namespace is transferred to data-name-4.
5. A node can be assigned to a data item only if a node is also assigned to all the data items which rank higher than it does in the structure.
6. The NAMESPACE specification defines the name pace for an entry and for all data items which are subordinate to it and have an IDENTIFIED clause with an explicit or implicit ELEMENT phrase.

7. If a subordinate data item also has a NAMESPACE phrase, this specification has priority over a NAMESPACE phrase in the superordinate group item.
8. NAMESPACE NULL means an empty string as namespace. If the NAMESPACE phrase is missing, and if no superordinate group item has a NAMESPACE phrase, NAMESPACE NULL is implied.
9. If an OPEN DOCUMENT, READ or START statement references a group item, the following conditions must be fulfilled for this group item and all group items which are subordinate to it:

   If a node from the XML tree can be assigned (see "Assigning nodes" in section "Statements for XML processing") to a data description entry which is directly subordinate to the group item, this assignment must be unique. In other words no other data description entry which is directly subordinate to the group item may exist to which the node from the tree could also be assigned.

   Neither the description of the nodes' names as alphanumeric or national literals or as data items is relevant with regard to uniqueness, nor are trailing blanks. If the uniqueness is not provided, the statement is not successful, and the I-O status of the file is set to 4C.

i
- The complete name of the namespace, the Uniform Resource Identifier (URI), must always be made available, or it is returned. The prefixes used in XML are **not** visible in COBOL.
- In an XML document namespace **declarations** have the form of an attribute, but they **not** supplied as attributes, even if they are described accordingly in an IDENTIFIED clause.

11.4 Language elements of the PROCEDURE DIVISION

Structure-oriented processing

Processing of a file with organization XML begins with an OPEN statement and ends with a CLOSE statement. In principle such a file contains multiple XML documents. Currently, however, only files which contain precisely one XML document are supported. Processing of this document, in particular generation of the tree presentation for the XML document, begins with an OPEN DOCUMENT statement and ends with a CLOSE DOCUMENT statement. The STARTstatement is used for the purpose of positioning in the XML tree. The READ statement transfers data from one or more nodes of the XML tree to the program. The OPEN DOCUMENT statement with the AT phrase (together with a CLOSE DOCUMENT phrase) enables a subtree to be handled like a separate document.

Event-oriented processing

The XML document to be edited must be made available in memory. The user program is responsible for determining whether a document should be read in from a file if required, whether an encoding should be converted, etc. The XML PARSE statement then edits and parses a complete XML document step by step. As soon as a syntax unit of the document is recognized, control is transferred temporarily to a routine – referred to as the processing procedure – which must be made available by the user and which handles the unit (comparable to the I /O procedures of a SORT statement). The syntax unit detected and additional data which is linked to it are made available in special registers in the processing program for further processing. When the processing program has been completed, the XML statement is resumed. This switching between the XML statement and the processing procedure continues until no further processing is possible as a result of errors, or the user program dispenses with parsing the XML document further.

Data transfer with structure-oriented processing

It is assumed that the XML document is presented in tree form in UTF-16, regardless how it is actually presented. Individual data items in the XML tree are referred to as XML values and must be regarded as elementary items of the national category. Transfer of the data from the XML tree to the receiving items in the COBOL program may therefore also involve conversion. The type of transfer is determined by the length of the XML value and the category of the receiving item.

- Length of the XML value = 0:
  The receiving item is initialized in accordance with the following statement:

  ```
  INITIALIZE receiving-item TO DEFAULT
  ```

- Length of the XML value > 0:
  - The category of the receiving item is national:
    The XML value is transferred in accordance with the rules for a MOVE statement.
  - The category of the receiving item is alphabetic or alphanumeric:
    The XML value is converted in accordance with the standard function DISPLAY-OF (without a second argument), and the result is transferred in accordance with the rules for a MOVE statement.
  - The category of the receiving item is numeric:
    A numeric value is formed from the XML value in accordance with the standard function NUMVAL-C (without a second argument), and this is transferred in accordance with the rules for a MOVE statement.

Exceptional condition in the case of structure-oriented processing

The XML document which is made available in a file or in memory is converted to national character representation in order to generate the tree presentation (in the case of the OPEN DOCUMENT statement without AT phrase). If characters which are not in national (UTF-16) code are found when this is done, the exceptional condition EC-XML-CODESET-CONVERSION occurs. This exceptional condition is NON-FATAL. The replacement character appears in the XML tree in place of the characters concerned.

After an XML value has been transferred to alphanumeric or alphabetic receiving items, the exceptional condition EC-DATA-CONVERSION may occur. The replacement character appears in this case in place of characters which could not be converted. If a USE procedure which is exited using RESUME AT NEXT STATEMENT exists for this exceptional condition, the program run is continued when the next XML value is transferred if multiple XML values are to be transferred using the READ statement and not all of these have been transferred yet.

Nesting statements for processing XML documents

The following restriction applies both for the statements for structure-oriented processing and for the statements for event-oriented processing of XML documents:

Execution of USE procedures or processing procedures **while** statements for processing an XML document – this can in particular involve the OPEN statement (see chapter "OPEN statement"), the READ statement (see chapter "READ statement") and the XML PARSE statement (see chapter "XML statement") – are being executed may not result in a further such statement being executed.

11.4.1 CLOSE statement

Function

The CLOSE statement terminates processing of a file with organization XML.

Format

```
CLOSE {file-name-1} ...
```

Syntax rules

1. In addition to names of files with organization XML, names of files which are organized differently can also be specified.

General rules

1. The rules for the CLOSE statement in chapter "CLOSE statement" apply for files which do not have organization XML.
2. If the file-name-1 file is already closed, the CLOSE statement is not successful and the I-O status of the file is set to 42.
3. If multiple file names are specified, the effect is the same as if a separate CLOSE statement had been issued for each file name.
4. If, for file-name-1, processing of an XML document has been started but has not yet been terminated by means of a CLOSE DOCUMENT statement, an implicit CLOSE DOCUMENT statement is also executed with the CLOSE statement.

11.4.2 CLOSE DOCUMENT statement

Function

The CLOSE DOCUMENT statement terminates processing of a complete XML document or of a subdocument and adjusts existing assignments of elements or attributes from the XML document to data items.

Format

```
CLOSE DOCUMENT file-name-1
```

Syntax rules

1. file-name-1 must be the name of a file with organization XML.

General rules

1. If the file-name-1 file is not open, the CLOSE DOCUMENT statement is not successful and the I-O status of the file is set to 4B.
2. If no XML document is open in the file-name-1 file, the CLOSE DOCUMENT statement is not successful and the I-O status of the file is set to 4D.
3. The behavior of the statement depends on the STACK phrase which was entered for the last OPEN DOCUMENT statement that was executed successfully for file-name-1:
   - STACK specified:
     The EPV is reset to the status it had before the OPEN DOCUMENT statement. The internal representation of the XML document is retained unchanged.
   - STACK not specified:
     All positions in the EPV are set to 'invalid' and the internal representation of the XML document is released.

> The CLOSE DOCUMENT statement only terminates processing of an XML document, but not processing of the file.

11.4.3 OPEN statement

Function

The OPEN statement opens a file with organization XML for processing.

Format

```
OPEN INPUT {file-name-1} ... [ END-OPEN ]
```

Syntax rules

1. In addition to names of files with organization XML, names of files which are organized differently can also be specified.

General rules

1. The rules for the OPEN statement (see section "OPEN statement") apply for files which do not have organization XML.
2. If the file-name-1 file is already opened, the OPEN statement is not successful and the I-O status of the file is set to 41.
3. If multiple file names are specified, the effect is the same as if a separate OPEN statement had been issued for each file name.
4. After a successful OPEN statement, the file specified by file-name-1 is available and is in open status. If the file is contained in memory, the OPEN statement sets up the connection to memory and evaluates any data items specified in the ASSIGN clause for pointers and length (see section "ASSIGN clause").
5. After a successful OPEN statement, the file position indicator points to the first XML document, provided the file is not empty. If the file is empty, or if OPTIONAL is specified in the file control entry, the file position indicator indicates 'end of file'.

11.4.4 OPEN DOCUMENT statement

Function

The OPEN DOCUMENT statement opens processing of an XML document. It generates the internal representation of the XML document and executes initial assignment of elements or attributes from the XML document to data items in the FD.

Format

```
OPEN DOCUMENT file-name-1 [ AT data-name-1 [STACK] ]
  [ RETURNING identifier-1 ]
  [ AT END imperative-statement-1 ]
  [ NOT AT END imperative-statement-2 ]
  [ END-OPEN ]
```

Syntax rules

1. file-name-1 must be the name of a file with organization XML.
2. data-name-1 may be qualified.
3. data-name-1 must be described with an IDENTIFIED clause with an explicit or implicit ELEMENT phrase and subordinate to the file-name-1 file.
4. identifier-1 must be an alphanumeric or national data item .
5. If data-name-1 is specified, the AT END and NOT AT END phrases may not be specified.

General rules

1. If, for file-name-1, processing of an XML document has been started but has not yet been terminated by means of a CLOSE DOCUMENT statement, an implicit CLOSE statement is executed before a CLOSE DOCUMENT statement without a data-name-1 statement is executed.
2. If the file-name-1 file is not open, the OPEN DOCUMENT statement is not successful. The I-O status is set to 4B.
3. If data-name-1 was specified but no XML document is open in the file-name-1 file, the OPEN DOCUMENT statement is not successful. The I-O status of the file is set to 4D.
4. If the XML document which is to be opened is not well-formed, the OPEN DOCUMENT statement is not successful. The I-O status is set to 3A.
5. If the encoding in which the XML document is represented could not be ascertained (see the "COBOL2000 User Guide" [1], section 10.4 “Encoding identification”), the OPEN DOCUMENT statement is not successful. The I-O status of the file is set to 3D.
6. If data-name-1 does not have a valid position, the OPEN DOCUMENT statement is not successful. The I-O status of the file is set to 46.
7. If the file position indicator points to 'end of file', the OPEN DOCUMENT statement is not successful. The I-O status of the file is set to 10.
8. If the statement is not successful, all positions in the EPV are set to 'invalid'.
9. If data-name-1 is not specified, the internal representation of the XML document is generated and the file position indicator is set to 'end of file'.

10. If data-name-1 is specified, the subsequent processing of the XML document is restricted to the subtree whose root is the node assigned to data-name-1.
    a. If STACK is specified, the status of the EPV is ensured by the OPEN DOCUMENT statement, which means that it can be restored again by the next OPEN DOCUMENT statement for file-name-1. However, no recourse can be made to the backed-up EPV status before such a CLOSE DOCUMENT statement is executed.
    b. If STACK is not specified, the status the EPV had before the OPEN DOCUMENT statement is lost.
11. The OPEN DOCUMENT statement performs assignment (see section "Language elements of the DATA DIVISION"). However, it transfers no data – except for the RETURNING phrase.
12. If the names in the IDENTIFIED clauses which are effective in the assignment are not unique (see section “IDENTIFIED clause”, general rule 9, on IDENTIFIED clause), the OPEN DOCUMENT statement is not successful. The I-O status of the file is set to 4C.
13. If individual attribute or element names from the IDENTIFIED clauses which are effective in the assignment cannot be represented in UTF-16 encoding, the OPEN DOCUMENT statement is not successful. The I-O status of the file is set to 4E.
14. When assignment takes place, all data items with level number 01 in the record description entries of file-name-1 must be taken into consideration in the first step on the COBOL side. Furthermore, the following node must be considered in the XML tree:
    a. if data-name-1 is not specified: the root node of the entire XML tree.
    b. if data-name-1 is specified: the node from the XML tree assigned to data-name-1.
15. If RETURNING is specified, the following is transferred to data item identifier-1:
    a. without AT specification, the name of the root element of the XML document.
    b. with AT specification, the name of the element in the XML document which is assigned to data-name-1.

    > **i** This transfer takes place with every successful OPEN DOCUMENT statement, even if the root element cannot be assigned to any data item in the COBOL description.

16. Whether execution of the OPEN DOCUMENT statement continues depends on whether AT END or NOT AT END is specified (see section "At end condition").

11.4.5 READ statement

Function

On the basis of an existing assignment, the READ statement positions to elements or attributes from the XML document, assigns them to the data items in the FD and transfers values associated with them.

Format

```
READ file-name-1 {ATTRIBUTE | [ONLY] ELEMENT} data-name-1
   [AT END imperative-statement-1]
   [NOT AT END imperative-statement-2]
   [END-READ]
```

Syntax rules

1. file-name-1 must be the name of a file with organization XML.
2. data-name-1 may be qualified.
3. data-name-1 must be described with an IDENTIFIED clause and subordinate to the filename-1 file.
4. If ATTRIBUTE is specified in the READ statement, the ATTRIBUTE phrase must be specified for the IDENTIFIED clause in the data description entry of data-name-1 .
5. If ELEMENT is specified in the READ statement, the ELEMENT phrase must be specified or implied for the IDENTIFIED clause in the data description entry of data-name-1.

General rules

1. If the file-name-1 file is not open, the READ statement is not successful. The I-O status of the file is set to 47.
2. If no XML document is open in the file-name-1 file, the READ statement is not successful. The I-O status of the file is set to 4D.
3. If data-name-1 does not have a valid position, the READ statement is not successful. The I-O status of the file is set to 46.
4. The READ statement first performs assignment (see section "Language elements of the DATA DIVISION"), and then transfers data if required. If no node from the XML tree can be assigned in the **first step** of this assignment procedure, the READ statement is not successful. The I-O status is set to 10, i.e. an at end condition exists.
5. If the names in the IDENTIFIED clauses which are effective in the assignment are not unique (see section "IDENTIFIED clause", general rule 9), the READ statement is not successful. The I-O status of the file is set to 4C.
6. If individual attributes or element names from the IDENTIFIED clauses which are effective in the assignment cannot be represented in UTF-16 encoding, the READ statement is not successful. The I-O status of the file is set to 4E.
7. When assignment takes place, only data item data-name-1 must be taken into consideration in the first step on the COBOL side, then the following nodes in the XML tree:
   a. If data-name-1 is an IDENTIFIED **BY** clause which has the ATTRIBUTE phrase: the node from the XML tree assigned to data-name-1 and **all** its siblings. As attributes of an element must be unique in XML documents, the order in which these nodes are to be considered is irrelevant here.
   b. Otherwise:

- If the node was assigned to data-name-1 using a READ statement: all **younger** siblings of this node in the order of the oldest of the younger siblings to the youngest of the younger siblings.
- If the node was assigned to data-name-1 using an OPEN DOCUMENT or START statement: this node itself and then all its younger siblings in the order of the oldest of the younger siblings to the youngest of the younger siblings.

8. If ONLY is specified, the assignments made for all data items which, explicitly or implicitly, have the ELEMENT phrase and which are subordinate to data-name-1 must be interpreted for all READ statements as if they had been specified by means of a START statement (and not using the READ statement).
9. If, in the subtree of the XML document whose root is the node which was assigned to data-name-1, nodes also exist which could not be assigned to any data item from the COBOL program, and if ONLY is not specified, the READ statement is not successful. The I-O status is set to 08.
10. As a result of the READ statement, data items in the record description entries of filename-1 e either remain unchanged, or are initialized, or the statement transfers data from the XML tree. In each individual case this depends on the assignment currently implemented and the position of the data items in the record descriptions relative to data-name-1:

Data items	**data-name-1 was assigned to a node**	**data-name-1 was not assigned to a node**
data-name-1	Transfer	Unchanged
Data item is subordinate to data-name-1; a node was assigned.	Transfer	–
Data item is subordinate to data-name-1; no node was assigned.	Initialization	Unchanged
Other data items	Unchanged	Unchanged

11. If ONLY is specified, transfer is restricted to data for data-name-1 and all the data items which are directly subordinate to it which have the ATTRIBUTE phrase in their IDENTIFIED clause.
12. 'Transfer' or 'Initialization' in conjunction with a data item means the assignment of values to the following data items (provided these are defined):

	Transfer	**Initialization**
Data item from the USING phrase of the IDENTIFIED clause	Name of the node from the XML tree	Blanks
Data item from the NAMESPACE USING phrase	Value of the namespace from the XML tree	Blanks
Data item for the value	Value of the node from the XML tree	INITIALIZE data-item TO DEFAULT
Data item from the COUNT clause	1	0

For the transfer, data from the XML tree is regarded as an XML value and handled accordingly (see "Data transfer with structure-oriented processing" in section "Language elements of the PROCEDURE DIVISION").

The data item for the value of a node corresponds to the data item with the IDENTIFIED clause, if this is elementary. Otherwise it must be directly subordinate to the data item with the IDENTIFIED clause and is the only one of the directly subordinate data items which may neither have an IDENTIFIED clause nor be referenced by an IDENTIFIED clause.

i
- Mixed content values are combined to form a single value. The content of CDATA sections is supplied as a component part of the 'normal values'.
- Values are also supplied for attributes which are not specified in the XML document if these are assigned a default value in the DTD.

13. Whether execution of the READ statement continues depends on whether AT END or NOT AT END is specified, see section "At end condition".

11.4.6 START statement

Function

The START statement positions to elements or attributes in the XML document and assigns them to data items in the FD.

Format

```
START file-name-1 {ATTRIBUTE | ELEMENT} data-name-1 [INDEX IS {identifier-1 |
integer-1}]
  [INVALID KEY imperative-statement-1]
  [NOT INVALID KEY imperative-statement-2]
  [END-START]
```

Syntax rules

1. file-name-1 must be the name of a file with organization XML.
2. data-name-1 may be qualified.
3. data-name-1 must be described with an IDENTIFIED clause and subordinate to the filename-1 file.
4. identifier-1 must be defined as an integer numeric elementary item.
5. If ATTRIBUTE is specified in the START statement, the ATTRIBUTE phrase must be specified in the IDENTIFIED clause in the data description entry of data-name-1.
6. If ELEMENT is specified in the START statement, the ELEMENT phrase must be specified or implied in the IDENTIFIED clause in the data description entry of data-name-1.
7. The INDEX specification is not permissible if the BY and ATTRIBUTE phrase is specified in the IDENTIFIED clause in the data description entry of data-name-1.

General rules

1. If the file-name-1 file is not open, the START statement is not successful. The I-O status of the file is set to 47.
2. If no XML document is open in the file-name-1 file, the START statement is not successful. The I-O status of the file is set to 4D.
3. If the data item directly superordinate to data-name-1 does not have a valid position, or if data-name-1 has level number 1 but no valid position, the START statement is not successful. The I-O status of the file is set to 25, i. e. an invalid key condition exists.
4. The START statement performs assignment (see section "Language elements of the DATA DIVISION"), but transfers no data. If no node from the XML tree can be assigned in the first step of this assignment procedure, the START statement is not successful. The I-O status is set to 23, i.e. an invalid key condition exists.
5. If the names in the IDENTIFIED clauses which are effective in the assignment are not unique (see section "IDENTIFIED clause", general rule 9), the START statement is not successful. The I-O status of the file is set to 4C.
6. If individual attribute or element names from the IDENTIFIED clauses which are effective in the assignment cannot be represented in UTF-16 encoding, the START statement is not successful. The I-O status of the file is set to 4E.
7. When assignment takes place, only data item data-name-1 must be taken into consideration in the first step on the COBOL side, then all children in the XML tree – in the order from the oldest to the youngest child – of the

node which is assigned to the data item that is directly superordinate to data-name-1. If data-name-1 has level number 1, only the node assigned to data-name-1 need be considered in the XML tree.

8. If INDEX is specified, integer-1 or the content of identifier-1 shows which node from the set of all nodes from the XML tree that can in principle be assigned in the first assignment step is actually to be assigned to data-name-1 by the statement. If the value specified is less than 1 or greater than 2^{31} - 1, the behavior is undefined.
9. Whether execution of the START statement continues depends on whether INVALID KEY or NOT INVALID KEY is specified (see section "Invalid key condition").

11.4.7 XML GENERATE statement

Function

The XML GENERATE statement converts data to an XML document.

Format

```
XML GENERATE data-name-1
        FROM data-name-2
      [COUNT [IN] data-name-3]
      [ON EXCEPTION imperative-statement-1]
      [NOT ON EXCEPTION imperative-statement-2]
[END-XML]
```

Syntax rules

1. The output operand *data-name-1* is designated for the generated XML document.
2. The group or elementary data item *data-name-2* is to be converted to the XML document.
3. The output operand *data-name-3* contains the count of generated XML characters.
4. The operands *data-name-1*, *data-name-2* and *data-name-3* must not overlap each other.

General rules

1. The output operand *data-name-1* must be large enough to contain the generated XML document and can reference one of the following:
   - elementary or group item of category alphanumeric;
   - elementary or group item of category national.

   If *data-name-1* is of category national, the generated XML document is encoded in UTF-16, otherwise in EBCDIC.
2. The sending operand *data-name-2* must not be described with the RENAMES clause.

   The following data items specified by *data-name-2* are ignored by the XML GENERATE statement:
   - Any unnamed or FILLER data item with its subordinates.
   - Any elementary data item that assumes a pointer, e.g. USAGE POINTER, OBJECT REFERENCE etc.
   - Any data item (with its subordinates) subordinate to *data-name-2* that is described with the REDEFINES clause.
   - Any group item (with its subordinates) subordinate to *data-name-2* that contains a non-unique name within the same level.
   - Any group item that does not contain at least one elementary data item of category alphabetic, alphanumeric, numeric, national or index.
3. The output operand *data-name-3* must be an integer data item and normally contains a length of the generated XML document in bytes. But if *data-name-1* is of category national, the count *data-name-3* is in national characters (UTF-16 encoding units).

4. After execution of the XML GENERATE statement, the special register XML-CODE contains either zero, which indicates successful completion, or a non-zero exception code.

CODE	Description
0	The receiver contains the successfully generated XML document. The COUNT [IN] data item contains the count of character positions in the generated XML document.
400	The receiver is too small to contain the generated XML document. The COUNT [IN] data item contains the count of character positions that were actually generated.
2999	COBOL2000 internal error. No data returned.

5. If an error (XML-CODE # 0) occurs during generation of the XML document, control is passed to the conditional statement *imperative-statement-1*. If ON EXCEPTION is not specified control is passed to the end of the XML GENERATE statement.
6. If no error (XML-CODE = 0) occurs during generation of the XML document, control is passed to the conditional statement *imperative-statement-2*. If NOT ON EXCEPTION is not specified control is passed to the end of the XML GENERATE statement.
7. While creating the XML document XML GENERATE observes the following rules:
   - The content of each elementary data item within *data-name-2* is converted to character format.Trailing spaces and leading zeroes are eliminated.
   - Any remaining instances of the five special characters & (ampersand), ’ (apostrophe), > (greater-than sign), < (less-than sign), and “ (quotation mark) are converted into the equivalent XML references ’&’, ’ '’, ’>’, ’<’, and ’"’, respectively.
   - The transformed content is then inserted as element character content in XML markup.
   - The XML element names are derived from the *data-name-2* source description. The names of group items are retained as XML parent names
   - If *data-name-1* is national, any non-national data is converted to national (UTF16) format and vice versa, if *data-name-1* is non-national, any national data is converted to alphanumeric (EBCDIC) format.
   - The XML declaration (header) is not generated. No extra white space, new line, etc. is inserted to make the generated XML more readable.

11.4.8 XML PARSE statement

Function

The XML PARSE statement parses an XML document sequentially into its syntax units. Data on known units is available for further processing in special registers in a processing procedure.

Format

```
XML PARSE identifier-1

  PROCESSING PROCEDURE IS procedure-name-1 [{THRU | THROUGH} procedure-name-2]

  [ON EXCEPTION imperative-statement-1]

  [NOT ON EXCEPTION imperative-statement-2]

  [END-XML]
```

Syntax rules

1. identifier-1 must be an alphanumeric or national data item.
2. identifier-1 may not be a function identifier.
3. If both procedure-name-1 and procedure-name-2 are specified and one of these is the name of a procedure within declarative sections, the other procedure name must be contained within the same declarative section.

General rules

1. While the XML PARSE statement is executed, identifier-1 contains the document to be parsed.
2. If the content of identifier-1 is modified while the XML PARSE statement is being executed, the further behavior is undefined.
3. When the XML PARSE statement is executed, control is transferred to the parser, which parses the document and makes the syntax units available to the program as 'events' in special registers.
4. The PROCESSING PROCEDURE specification defines the processing procedure to which the parser transfers control for every 'event' while the XML statement is being executed. The processing procedure is executed as if it were activated by the parser using PERFORM.
5. The processing procedure may not be exited using EXIT PROGRAM, EXIT METHOD or GOBACK. When the processing procedure’s last statement has been executed, control is automatically returned to the parser.
6. If the parser detects an error while it is parsing an XML document, it transfers control to the processing procedure with the 'EXCEPTION' event in the XML-EVENT special register. The XML-CODE special register then contains the value of the extended I-O status, see the "COBOL2000 User Guide" [1], section 10.6 "Extended I-O status for XML statements (CBX code).” With all other events the XML-CODE special register has the value 0.
7. The content of the XML-CODE special register XML-CODE when the processing procedure is **exited** controls how the XML PARSE statement is continued. The processing procedure can signal how it wants the statement to be continued by setting the appropriate values:

Values of XML CODE special register at the end of the processing procedure	Value of XML-EVENT special register at the start of the processing procedure	
	EXCEPTION	Other event
0	Attempt to continue the statement[1]	Continuation of the statement
-1	Immediate termination of the statement with 'error'	Immediate termination of the statement with 'error'
Unchanged error code	Immediate termination of the statement with 'error'	–
Other value	Further behavior undefined	Further behavior undefined

[1]Continuation is, however, only possible in the case of 'minor' errors and is then generally restricted to the search for further errors. When 'serious' errors occur, the XML statement is nevertheless always terminated with ’error’.

> i Values which are not equal to -1 will in future be assigned their own semantics. On account of this, do not set any values which are not equal to -1.

8. After control is returned from the parser, the XML-CODE special register contains the value which it had the last time it exited the processing procedure.
9. If parsing of the document is completed without an error, imperative-statement-2 (if it is specified) is executed when control is returned from the parser and the end of the XML PARSE statement is branched to.
10. If the XML PARSE statement was terminated with 'error', imperative-statement-1 (if it is specified) is executed when control is returned from the parser and the end of the XML PARSE statement is branched to.

11.5 Special registers for the XML PARSE statement

Event-oriented processing of an XML document makes data about the detected syntax units available in special registers at the start of the processing procedure.

Data description entries for the special registers may not be written by the programmer. Rather, the compiler generates these automatically when the option for detecting XML-specific keywords is enabled when the program is compiled (see the "COBOL2000 User Guide" [1], section 10.2 “Using XML language elements in programs”). The special registers exist only once per run unit and are also available in nested programs.

There are three groups of special registers:

- XML-EVENT, XML-CODE:

  The data supplied in these special registers always has a fixed length and describes an event.
- For XML documents which are made available in an alphanumeric data item:

  XML-TEXT, XML-NAMESPACE, XML-NAMESPACE-PREFIX.

  The data supplied in these special registers is of variable length and describes the data associated with an event.

  For the exception in the special case of the CONTENT-NATIONAL-CHARACTER event, see also sections "XML-TEXT" and "XML-NTEXT" .
- For XML documents which are made available in a national data item:

  XML-NTEXT, XML-NNAMESPACE, XML-NNAMESPACE-PREFIX.

  The data supplied in these special registers is of variable length and describes the data associated with an event.

The special registers – with the exception of XML-CODE – are modified only by the XML PARSE statement. If they occur in the program as a receiving item, this results in undefined behavior. The special registers – with the exception of XML-CODE – consequently only have meaningful contents while an XML statement is being executed. When required, the standard function FUNCTION LENGTH (special-register) supplies the current length of the variable-length data.

XML-EVENT

At the start of the processing procedure this special register contains the name of the event which has just been detected. The possible events are described in detail in section "Processing procedure".

This special register corresponds to an elementary item with the description

```
USAGE DISPLAY PIC X(30).
```

XML-CODE

At the start of the processing procedure this special register contains an error code if the event is 'EXCEPTION'. With all other events it contains the value 0. The possible values for the error code are provided in the "COBOL2000 User Guide" [1], section 10.6 "Extended I-O status for XML statements (CBX code).”

This is the only special register which may also be modified by the program in order to signal the required type of continuation when control is returned from a processing procedure to the XML PARSE statement (see section "XML statement").

This special register corresponds to a data item with the description

```
USAGE COMP-5 PICTURE S9(9).
```

XML-TEXT

If the XML document is contained in an alphanumeric data item, at the start of the processing procedure this special register contains part of the XML document which is described exactly by the event.

This special register corresponds to an elementary item with the description

    USAGE DISPLAY PICTURE X ANY LENGTH,

the current length being the same as the length of the document part supplied (in characters). If the XML document is contained in a national data item or if the event is END-OF-DOCUMENT, CONTENT-NATIONAL-CHARACTER or NAMESPACE-DECLARATION, the current length is 0.

XML-NTEXT

If the XML document is contained in a national data item, at the start of the processing procedure this special register contains part of the XML document which is described exactly by the event.

This special register corresponds to an elementary item with the description

    USAGE NATIONAL PICTURE N ANY LENGTH,

the current length being the same as the length of the document part supplied (in characters). If the XML document is contained in an alphanumeric data item – with the exception of the CONTENT-NATIONAL-CHARACTER event – or if the event is END-OF-DOCUMENT or NAMESPACE-DECLARATION, the current length is 0.

XML-NAMESPACE

If the XML document is contained in an alphanumeric data item, at the start of the processing procedure this special register contains the name of the namespace in the case of the ATTRIBUTE-NAME, DEFAULTED-ATTRIBUTE-NAME, END-OF-ELEMENT, NAMESPACE-DECLARATION and START-OF-ELEMENT events.

This special register corresponds to an elementary item with the description

    USAGE DISPLAY PICTURE X ANY LENGTH,

the current length being the same as the length of the namespace name (in characters). If the XML document is contained in a national data item or the namespace is empty or another event is involved, the current length is 0.

XML-NNAMESPACE

If the XML document is contained in a national data item, at the start of the processing procedure this special register contains the name of the namespace in the case of the ATTRIBUTE-NAME, DEFAULTED-ATTRIBUTE-NAME, END-OF-ELEMENT, NAMESPACE-DECLARATION and START-OF-ELEMENT events.

This special register corresponds to an elementary item with the description

    USAGE NATIONAL PICTURE N ANY LENGTH,

the current length being the same as the length of the namespace name (in characters). If the XML document is contained in an alphanumeric data item or the namespace is empty or another event is involved, the current length is 0.

XML-NAMESPACE-PREFIX

If the XML document is contained in an alphanumeric data item, at the start of the processing procedure this special register contains the prefix of the namespace in the case of the ATTRIBUTE-NAME, DEFAULTED-ATTRIBUTE-NAME, END-OF-ELEMENT, NAMESPACE-DECLARATION and START-OF-ELEMENT events.

This special register corresponds to an elementary item with the description

    USAGE DISPLAY PICTURE X ANY LENGTH,

the current length being the same as the length of the prefix (in characters). If the XML document is contained in a national data item or the namespace has no prefix or another event is involved, the current length is 0.

XML-NNAMESPACE-PREFIX

If the XML document is contained in a national data item, at the start of the processing procedure this special register contains the prefix of the namespace in the case of the ATTRIBUTE-NAME, DEFAULTED-ATTRIBUTE-NAME, END-OF-ELEMENT, NAMESPACE-DECLARATION and START-OF-ELEMENT events.

This special register corresponds to an elementary item with the description

    USAGE DISPLAY PICTURE N ANY LENGTH,

the current length being the same as the length of the prefix (in characters). If the XML document is contained in an alphanumeric data item or the namespace has no prefix or another event is involved, the current length is 0.

12 General concepts

12.1 File processing

A file is a collection of records that can be moved to, or read from, a volume. The user defines the organization of the file as well as the mode and order in which the records are processed.

The organization of a file describes its logical structure. There are sequential, indexed, and relative types of file organization. The file organization which is defined at the time a file is created cannot be changed later on.

A sequential file can only be processed sequentially, i.e. the records are either read or written in the order predetermined by the file. In the case of sequential files on disk storage devices, records may also be updated in place. This requires a READ statement followed immediately by a REWRITE.

12.1.1 Sequential file organization

12.1.1.1 Record sequential organization

When record sequential file organization is used, the logical records are placed on the file or read sequentially either forwards or backwards (REVERSED) in the order in which they were generated.

This type of file organization must be used for magnetic tape or unit record files and may be used for disk storage files. Sequentially organized files require no key for record processing.

12.1.1.2 Line sequential organization

The line sequential organization of COBOL files is one of the language elements supported by X/Open standards.It serves to generate and process text files that can be processed and generated by the text editors of the operating system.

12.1.1.3 I-O status

The I-O status is a value that can be used in a COBOL program to check the status of an input/output operation. In order to do this, the FILE STATUS clause must be specified in the FILE CONTROL paragraph of the ENVIRONMENT DIVISION.
The I-O status value is transferred to a two-character data item

- during the execution of a CLOSE, OPEN, READ, REWRITE, or WRITE statement,
- prior to the execution of any associated imperative statement, and
- prior to the execution of any corresponding USE AFTER STANDARD EXCEPTION procedure.

The table below shows the I-O status values and their meanings:

I-O status	Meaning
	Execution successful
00	The I-O statement terminated normally. No further information regarding the I-O operation is available.
04	Record length conflict: A READ statement terminated normally. However, the length of the record read lies outside the limits defined in the record description entry for this file.
05	Successful execution of an OPEN INPUT/I-O/EXTEND on a file; however, the referenced file indicated by the OPTIONAL phrase was not present at the time the OPEN statement was executed.
07	1. Successful OPEN statement with NO REWIND clause on a file that is on a UNIT-RECORD medium. 2. Successful CLOSE statement with NO REWIND, REEL/UNIT, or FOR REMOVAL clause on a file that is on a UNIT-RECORD medium.
	Execution unsuccessful: at end condition
10	An attempt was made to execute a READ statement. However, no next logical record existed, because the end-of-file was encountered. A sequential READ statement with the OPTIONAL phrase was attempted for the first time on a nonexistent file.
	Execution unsuccessful: unrecoverable error
30	1. No further information regarding the I-O operation is available (the DMS code provides further information). 2. In the case of line sequential processing: unsuccessful attempt to access PLAM item
34	An attempt was made to write outside the sequential file boundaries set by the system.
35	An OPEN statement with the INPUT/I-O/EXTEND phrase was attempted on a nonexistent file.
37	OPEN statement on a file that cannot be opened in any of the following ways:

	1. OPEN OUTPUT/I-O/EXTEND on a write-protected file (password, RETENTION-PERIOD, ACCESS=READ in catalog) 2. OPEN I-O on a tape file 3. OPEN INPUT on a read-protected file (password)
38	An attempt was made to execute an OPEN statement for a file previously closed with the LOCK phrase.
39	The OPEN statement was unsuccessful as a result of one of the following conditions: 1. One or more of the operands ACCESS-METHOD, RECORD-FORMAT or RECORD-SIZE were specified in the ADD-FILE-LINK command with values deviating from the corresponding explicit or implicit program specifications. 2. record length errors occurred for input files (catalog check if RECFORM=F), or 3. The record size is greater than the BLKSIZE entry in the catalog (in the case of input files). 4. The catalog entry of one of the FCBTYPE, RECFORM or RECSIZE (if RECFORM=F) operands for an input file is in conflict with the corresponding explicit or implicit program specifications or with the specifications in the ADD-FILE-LINK command.
	Execution unsuccessful: logical error
41	An attempt was made to execute an OPEN statement for a file which was already open.
42	An attempt was made to execute a CLOSE statement for a file which was not open.
43	While accessing a disk file opened with OPEN I-O, the most recent I-O statement executed prior to a REWRITE statement was not a successfully executed READ statement.
44	Boundary violation: 1. An attempt was made to execute a WRITE statement. However, the length of the record is outside the range allowed for this file. 2. An attempt was made to execute a REWRITE statement. However, the record to be rewritten did not have the same length as the record to be replaced.
46	An attempt was made to execute a READ statement for a file in INPUT or I-O mode. However, there is no valid next record since: 1. the preceding READ statement was unsuccessful without causing an at end condition 2. the preceding READ statement resulted in an at end condition.
47	An attempt was made to execute a READ statement for a file not in INPUT or I-O mode.
48	An attempt was made to execute a WRITE statement for a file not in OUTPUT or EXTEND mode.
49	An attempt was made to execute a REWRITE statement for a file not open in I-O mode.
	Other conditions with unsuccessful execution
90	System error; no further information available regarding the cause.

91	System error; a system call terminated abnormally; either an OPEN error or no free device; the actual cause is evident from the DMS code (see section "FILE STATUS clause")
95	The specifications in the BLOCK-CONTROL-INFO or BUFFER-LENGTH operands of the ADD-FILE-LINK command are not consistent with the file format, the block length, or the format of the volume being used.

12.1.2 Relative file organization

12.1.2.1 Relative organization

When using relative file organization, the location of each record in a relative file is determined by means of a relative record number, i.e. an integer value greater than zero, which specifies the position of that record within the logical sequence of the file. The record number is predefined by the user in a relative key field. The file may be seen as a serial sequence of areas, each of which contains one logical record. Each of these areas is identified by a relative record number. Storage and retrieval of the records is accomplished on the basis of that number. For example, the tenth record is referenced through the relative record number 10 and is located in the tenth record area, whether or not records have been written in any of the record areas 1 to 9. Relative file organization is permitted for disk storage files only.

12.1.2.2 Sequential access to records

Records in a relative file may be sequentially created, read, and updated.

A relative file may be created sequentially, in which case the RELATIVE KEY need not be specified, as the records are written to the output file in physically consecutive order.

In sequential reading of records from a relative file, the records are processed in the order of their relative record numbers, i.e. a READ statement will make available the next or preceding existing logical record of the file. The first record that is read is either the first record made available in the file, or a record whose position was specified in a START statement. In the latter case, RELATIVE KEY must be specified.

If the RELATIVE KEY is specified for a relative file, execution of a READ or WRITE statement causes the RELATIVE KEY item to be set to the relative record number of the record which is made available.

An already existing relative file may be updated with the aid of READ, REWRITE, or DELETE statements. The last record read may be updated and then rewritten to its original location in the file, or it may be logically deleted.

12.1.2.3 Random access to records

Records of a relative file may be randomly created, read, and updated. With this access method, the RELATIVE KEY phrase is always required. Before each execution of an input statement or output statement, the data item indicated in RELATIVE KEY phrase must be provided with the value of the required relative record number.

A relative file may be created randomly; in this case, the record placed on the file occupies the record area of the relative record number as specified in the RELATIVE KEY data item. When a relative file is read randomly, the retrieved record is the one whose relative record number is contained in the data item of the RELATIVE KEY phrase associated with that file.

An already existing relative file may be updated with the aid of REWRITE or DELETE statements. An explicit random READ statement may optionally be issued prior to the execution of a REWRITE or DELETE statement for the record to be updated, as these statements will modify the record that is denoted by the relative record number in the RELATIVE KEY data item.

12.1.2.4 Dynamic access to records

Combination of sequential and random access modes.

12.1.2.5 I-O status

The I-O status is a value that can be used in a COBOL program to check the status of an input/output operation. In order to do this, the FILE STATUS clause must be specified in the FILE CONTROL paragraph of the ENVIRONMENT DIVISION.
The I-O status value is transferred to a two-character data item

- during the execution of a CLOSE, OPEN, READ, REWRITE, or WRITE statement,
- prior to the execution of any associated imperative statement, and
- prior to the execution of any corresponding USE AFTER STANDARD EXCEPTION procedure.

The table below shows the I-O status values and their meanings:

I-O status	Meaning
	Execution successful
00	The I-O statement terminated normally. No further information regarding the I-O operation is available.
04	Record length conflict: A READ statement was executed successfully, but the length of the record which was read does not lie within the limits specified in the record description for the file. Successful OPEN INPUT/I-O/EXTEND for a file with the OPTIONAL phrase in the SELECT clause that was not present at the time of execution of the OPEN statement.
	Execution unsuccessful: at end condition
10	An attempt was made to execute a READ statement. However, no next logical record was available, since the end-of-file was encountered (sequential READ). A first attempt was made to execute a READ statement for a non-existent file which is specified as OPTIONAL.
14	An attempt was made to execute a READ statement. However, the data item described by RELATIVE KEY is too small to accommodate the relative record number. (sequential READ).
	Execution unsuccessful: invalid key condition
22	Duplicate key An attempt was made to execute a WRITE statement with a key for which there is already a record in the file.
23	Record not located or zero record key An attempt was made (using a READ, START, DELETE or REWRITE statement with a key) to access a record not contained in the file, or the access was effected with a zero record key.
24	Boundary values exceeded (see "COBOL2000 User Guide" [1]). An attempt was made to execute a WRITE statement beyond the system-defined boundaries of a

	relative file (insufficient secondary allocation in the FILE command), or a WRITE statement is attempted in sequential access mode with a relative record number so large that it does not fit in the data item defined with the RELATIVE KEY phrase.
	Execution unsuccessful: permanent error
30	No further information regarding the I-O operation is available.
35	An attempt was made to execute an OPEN INPUT/I-O statement for a nonexistent file.
37	An OPEN statement is attempted on a file that cannot be opened due to the following conditions: 1. OPEN OUTPUT/I-O/EXTEND on a write-protected file (password, RETPD in catalog, ACCESS=READ in catalog) 2. OPEN INPUT on a read-protected file (password)
38	An attempt was made to execute an OPEN statement for a file previously closed with the LOCK phrase.
39	The OPEN statement was unsuccessful as a result of one of the following conditions: 1. One or more of the operands ACCESS-METHOD, RECORD-FORMAT or RECORD-SIZE were specified in the ADD-FILE-LINK command with values which conflict with the corresponding explicit or implicit program specifications. 2. The catalog entry of the FCBTYPE operand for an input file does not match the corresponding explicit or implicit program specification or the specification in the ADD-FILE-LINK command. 3. Variable record length has been defined for a file that is to be processed using the UPAM access method of the DMS.
	Execution unsuccessful: logical error
41	An attempt was made to execute an OPEN statement for a file which was already open.
42	An attempt was made to execute a CLOSE statement for a file which was not open.
43	For ACCESS MODE IS SEQUENTIAL: The most recent I-O statement executed prior to a DELETE or REWRITE statement was not a successfully executed READ statement.
44	Record length limits exceeded: An attempt was made to execute a WRITE or REWRITE statement, but the length of the record does not lie within the limits defined for the file.
46	An attempt was made to execute a sequential READ statement for a file in INPUT or I-O mode. However, there is no valid next record since: 1. the preceding START statement terminated abnormally, or 2. the preceding READ statement terminated abnormally without causing an at end condition. 3. the preceding READ statement caused an AT END condition.
47	An attempt was made to execute a READ or START statement for a file not in INPUT or I-O mode.

48	An attempt was made to execute a WRITE statement for a file • not in OUTPUT or EXTEND mode (for sequential access) • not in OUTPUT or I-O mode (for random or dynamic access)
49	An attempt was made to execute a DELETE or REWRITE statement for a file not in I-O mode.
	Other conditions with unsuccessful execution
90	System error; no further information regarding the cause is available.
91	System error; OPEN error
93	For simultaneous processing only (see "Shared updating of files" in the "COBOL2000 User Guide" [1]): The I-O statement could not terminate normally because a different task is accessing the same file, and the access operations are incompatible.
94	For shared update processing only (see "Shared updating of files" in the "COBOL2000 User Guide" [1]): deviation from call sequence READ - REWRITE/DELETE.
95	The specifications in the BLOCK-CONTROL-INFO or BUFFER-LENGTH operands of the ADD-FILE-LINK command are not consistent with the file format, the block length, or the format of the volume being used.
96	READ PREVIOUS is not supported for a module compiled with COBRUN ENABLE-UFS-ACCESS=YES or the file has to be processed with the DMS UPAM access method.

12.1.3 Indexed file organization

12.1.3.1 Indexed organization

When indexed file organization is used, the position of each logical record in the file is determined by indices which are generated with the file and are maintained by the system. These indices are based on keys which must be supplied by the user in the records. Indexed files must be assigned to disk storage devices.

The RECORD KEY clause must be specified when creating an indexed file. This clause defines which data item within the record is to be used as the primary key.

The ALTERNATE RECORD KEY clause can be used to define one or more alternate keys (secondary keys) in addition to the primary key.

The START statement can be used to define a starting point, within an indexed file, for a series of subsequent sequential access operations.

12.1.3.2 Sequential access to records

Records in an indexed file may be sequentially created, read and updated.

The records are read in ascending or descending key order.

12.1.3.3 Random access to records

Records of indexed files may be randomly created, read, and updated.

The data item defined as the key is the data-name specified in the RECORD KEY clause.

When a new record is created, the value of the RECORD KEY should not be identical to the value of a key field that is already in existence.

12.1.3.4 Dynamic access to records

In dynamic access mode, the user may, by using the appropriate I/O statements, switch as required between sequential and random access.

12.1.3.5 I-O status

The I-O status is a value that can be used in a COBOL program to check the status of an input/output operation. In order to do this, the FILE STATUS clause must be specified in the FILE CONTROL paragraph of the ENVIRONMENT DIVISION.
The I-O status value is transferred to a two-character data item

- during the execution of a CLOSE, OPEN, READ, REWRITE, or WRITE statement,
- prior to the execution of any associated imperative statement, and
- prior to the execution of any corresponding USE AFTER STANDARD EXCEPTION procedure.

The table below shows I-O status values and their meanings:

I-O status	Meaning
	Execution successful
00	The I-O statement terminated normally. No further information regarding the I-O operation is available.
02	A record was read with ALTERNATE KEY and subsequent sequential reading with the same key has found at least one record with an identical key. A record was written with ALTERNATE KEY WITH DUPLICATES and there is already a record with an identical key value for at least one alternate key.
04	Record length conflict: A READ statement terminated normally. However, the length of the record read lies outside the limits defined in the record description entry for the given file.
05	An OPEN statement was executed for an OPTIONAL file which does not exist.
	Execution unsuccessful: at end condition
10	An attempt was made to execute a sequential READ operation. However, no next logical record was available, as the end-of-file was encountered.
	Execution unsuccessful: invalid key condition
21	File sequence error in conjunction with ACCESS MODE IS SEQUENTIAL: 1. The record key value was changed between the successful execution of a READ statement and the execution of the next REWRITE statement for a file, or 2. the ascending sequence of record keys was violated in successive WRITE statements.
22	Duplicate key An attempt was made to execute a WRITE statement with a primary key for which there is already a record in the indexed file. An attempt was made to create a record with ALTERNATE KEY, but without WITH DUPLICATES, and there is already an alternate key with the same value in the file.

23	Record not located An attempt was made (using a READ, START, DELETE or REWRITE statement with key) to access a record not containing in the file.
24	Boundary values exceeded An attempt was made to execute a WRITE statement beyond the system-defined boundaries of an indexed file (see "COBOL2000 User Guide" [1]).
	Execution unsuccessful: unrecoverable error
30	No further information regarding the I-O operation is available (the DMS code provides further information).
35	An OPEN statement with the INPUT, I-O or EXTEND phrase was issued for a non-optional file which does not exist.
37	OPEN statement on a file that cannot be opened due to the following violations: 1. OPEN OUTPUT/I-O/EXTEND on a write-protected file(password, RETPD in catalog, ACCESS=READ in catalog) 2. OPEN INPUT on a read-protected file (password)
38	An attempt was made to execute an OPEN statement for a file previously closed with the LOCK phrase.
39	The OPEN statement was unsuccessful as a result of one of the following conditions: 1. One or more of the operands ACCESS-METHOD, RECORD-FORMAT, RECORD-SIZE or KEY-LENGTH were specified in the ADD-FILE-LINK command with values that conflict with the corresponding explicit or implicit program specifications. 2. Record length error occurred for an input file (catalog check, if RECFORM=F). 3. The record size is greater than the BLKSIZE entry in the catalog of an input file. 4. The catalog entry of one of the FCBTYPE, RECFORM, RECSIZE (if RECFORM=F), KEYPOS, or KEYLEN operands for an input file is in conflict with the corresponding explicit or implicit program specifications or with the corresponding specifications in the ADD-FILE-LINK command. 5. An attempt was made to open a file whose alternate key does not match the key values specified in the ALTERNATE RECORD KEY clause in the program.
	Execution unsuccessful: logical error
41	An attempt was made to execute an OPEN statement for a file which was already open.
42	An attempt was made to execute a CLOSE statement for a file which was not open.
43	For ACCESS MODE IS SEQUENTIAL: The most recent I-O statement executed prior to a DELETE or REWRITE statement was not a successfully executed READ statement.

44	Record length limits exceeded: An attempt was made to execute a WRITE or REWRITE statement. However, the length of the record is outside the range allowed for this file.
46	An attempt was made to execute a sequential READ statement for a file in INPUT or I-O mode. However, no valid next record is available since: 1. the preceding START statement was unsuccessful, or 2. the preceding READ statement was unsuccessful without leading to an at end condition, or 3. an attempt was made to execute a READ statement after the at end condition was encountered.
47	An attempt was made to execute a READ or START statement for a file that is not open in INPUT or I-O mode.
48	An attempt was made to execute a WRITE statement for a file that is not in OUTPUT, I-O or EXTEND mode.
49	An attempt was made to execute a DELETE or REWRITE statement for a file that is not in I-O mode.
	Other conditions with unsuccessful execution
90	System error; no further information regarding the cause is available.
91	OPEN error: the actual cause is evident from the DMS code (see "section FILE STATUS clause" specifying data-name-2).
93	For shared update processing only (see "COBOL2000 User Guide" [1], "Shared updating of files"): The I-O statement could not terminate normally because a different task is accessing the same file, and the access operations are incompatible:
94	For shared update processing only (see "COBOL2000 User Guide" [1], "Shared updating of files"): 1. deviation from call sequence READ - REWRITE/DELETE. 2. The record size is greater than the block size.
95	The specifications in the BLOCK-CONTROL-INFO or BUFFER-LENGTH operands of the ADD-FILE-LINK command are not consistent with the file format, the block length, or the format of the volume being used.
96	READ PREVIOUS is not supported for a module compiled with COBRUN ENABLE-UFS-ACCESS=YES.

12.1.4 Input/output statements

In COBOL, input and output are record-oriented. Thus the READ, WRITE, DELETE and REWRITE statements process records. The COBOL user is therefore only concerned with the processing of individual records. The following operations are performed automatically:

- moving data into input/output areas (buffers) and/or internal storage
- validity checking
- correcting errors (where feasible)
- blocking/unblocking
- switching volumes (only for **sequential file organization**).

Note

The description of input/output statements for **sequentially organized files** uses the expressions "volume" and "reel". Volume may be used for all input/output devices; reel is applicable to magnetic tape devices only. The handling of disk storage files with sequential access is logically equivalent to the handling of magnetic tape files.

Overview

Statement	Function
CLOSE	Terminates processing of a file
DELETE	Deletes a record
OPEN	Opens a file for processing
READ	Reads a record
REWRITE	Replaces a record
START	Positions within a file
USE	In addition to input/output statements, USE statements may be supplied to specify error handling procedures (see section "DECLARATIVES")
WRITE	Writes a record

DELETE and START are only applicable to **relative and indexed file organization**.

12.1.5 Invalid key condition

The invalid key condition may occur after the execution of a DELETE, READ, REWRITE, START or WRITE statement.

How execution continues after the input/output statement has been executed depends on the I-O status displayed. The table below shows how/where the program run is continued in each case. '-' stands for phrases which are not relevant for selecting the type of continuation.

I-O status after the statement indicates	INVALID KEY imperative-statement-1		NOT INVALID KEY imperative-statement-1		USE procedure	
	Specified	Not specified	Specified	Not specified	Declared	Not declared
successful	-	-	imperative-statement-2	End of the statement	-	-
invalid key condition	imperative-statement-1	Continuation as in 'USE procedure' column for other error condition	-	-	-	-
other error condition	-	-	-	-	use-procedure	Program abortion

imperative-statement-1, imperative-statement-2 and use-procedure are executed in accordance with the rules for the statements specified there. Depending on the type of statement, the program run is either continued in another part of the program or at the end of the input/output statement.

12.1.6 At end condition

The at end condition can occur after an OPEN DOCUMENT or READ statement has been executed.

How execution continues after the input/output statement has been executed depends on the I-O status displayed. The table below shows how/where the program run is continued in each case. ’-’ stands for specifications which are not relevant for selecting the type of continuation.

I-O status after the statement indicates	AT END imperative-statement-1		NOT AT END imperative-statement-2		USE procedure	
	Specified	Not specified	Specified	Not specified	Declared	Not declared
successful	-	-	imperative-statement-2	End of the statement	-	-
at end condition	imperative-statement-1	Continuation as in 'USE procedure' column for other error condition	-	-	-	-
other error condition	-	-	-	-	use-procedure	Program abortion

imperative-statement-1, imperative-statement-2 and use-procedure are executed in accordance with the rules for the statements specified there. Depending on the type of statement, the program run is either continued in another part of the program or at the end of the input/output statement.

12.2 Exception conditions and exception statuses

An exception condition can occur at runtime in the form of a deviation from the normal execution of a COBOL statement.
To permit a targeted reaction to an exception condition in a program or method, it is possible, during file processing, not only to specify explicit clauses (e.g. SIZE ERROR phrase) and inquire input and output statuses, but also to trigger exception statuses for some exception conditions and to process these in USE procedures. These exception conditions are assigned exception condition names which enable them to be referenced. A list of these names is provided in table 45.
In order to trigger the associated exception status for an exception condition which has occurred, checking of the condition must be activated using the >>TURN directive.

If the check is activated for an exception condition but a clause (without NOT) which can be used to handle the exception condition is also specified (e.g. INVOKE ... ON EXCEPTION for EC-OO-NULL), the clause has precedence, i.e. the exception condition is checked (because of the clause), but the associated exception status is **not** triggered.

Furthermore, exception statuses can be triggered directly by the RAISE statement.

If an exception status is triggered, the last exception status is updated. This applies for the entire run unit until a new exception status is triggered or the SET LAST EXCEPTION TO OFF statement is executed. The associated name can be inquired via the EXCEPTION-STATUS function.
The further procedure depends on the severity (category) of the exception condition. Exception conditions are either FATAL or NON-FATAL.

Triggering a FATAL exception condition

1. If a USE procedure is specified for the exception condition, it is activated.
2. If the end of the USE procedure is reached without the USE procedure being exited explicitly beforehand, the program run is aborted.
3. If no USE procedure is specified, the program run is aborted.

Triggering a NON-FATAL exception condition

1. If a USE procedure is specified for the exception condition, it is activated.
2. When the end of the USE procedure is reached without being exited explicitly beforehand, program execution is aborted.
3. If no USE procedure is specified, the program run is continued as if the check of the exception condition is deactivated.

Exception condition names

The table below contains a list of the exception condition names which are supported by the current compiler version.

Meaning of the columns

Name:	Names of the exception conditions.
Cat:	Category of the exception condition: Fatal, Non-Fatal (NF).
Brief description:	Case in which the exception condition occurs

Clauses: The exception condition can be handled by the specified clauses.

Name	Cat	Brief description	Clauses
EC-DATA-CONVERSION	NF	Replacement character while conversion	-
EC-OO-CONFORMANCE	Fatal	Error in object view	-
EC-OO-METHOD	Fatal	Method not found	ON EXCEPTION
EC-OO-NULL	Fatal	Method called with a NULL object reference	ON EXCEPTION
EC-OO-RESOURCE	Fatal	Not enough storage space available to generate an object	-
EC-OO-UNIVERSAL	Fatal	Conformity rules not met for method call via universal object reference	ON EXCEPTION
EC-STORAGE-NOT-ALLOC	NF	Pointer in the FREE does not point to a storage area allocated by ALLOCATE	-
EC-STORAGE-NOT-AVAIL	NF	The storage specified in an ALLOCATE statement is not available	-
EC-XML-CODESET-CONVERSION	NF	Replacement character while converting an XML document	-

Table 45: Exception condition names

12.3 Initial and "last-used" states

The **“initial state”** means the following for:

- *indices:* undefined
- *object references/pointers:* initialized to NULL
- *other data*: value from the VALUE clause, if specified; otherwise, undefined
- *files*: file connector set to be not in any open mode.

The **last-used” state** means the following for:

- *data*: state following the last preceding update
- *files*: state following the last preceding update

The states of data or files on calling a method or program depend on

- the section in which they are defined and other clauses of the definition
- what occurred earlier on executing the run unit

Section of the definition	Preceding execution sequence					
	Method		Program			
	First call	Further calls	with INITIAL[1]	without INITIAL		
			First / further calls	Very first call	First call after Cancel[2]	Other calls
Working Storage with EXTERNAL	initial last-used	last-used last-used	initial last-used	initial last-used	initial last-used	last-used last-used
Local Storage	initial	initial	initial	initial	initial	initial
Linkage [3]	last-used	last-used	last-used	last-used	last-used	last-used

[1] The program contains the clause itself or is itself contained in such a program

[2] A Cancel on the program itself or any other program in which it is contained

[3] Refers to the currently passed parameters

12.4 Inter-program communication

Inter-program communication with COBOL2000 comprises the following:

- Transfer of control from one program to another, with the facility to pass between the individual programs parameters which allow data from the calling program to be made available to a called program.
- The use of common data and files by the different programs of a run unit.

12.4.1 Concepts

Separately compiled program

A complete COBOL program that was compiled in a separate compiler run is referred to as a separately compiled program. It can be either an individual program or the outermost containing program of a nested program.

Nested program

A COBOL program that comprises a number of complete programs nested within one another is referred to as a "nested program".
A program that contains further programs is referred to as a "containing program".
A program that is contained in another program is referred to as a "contained program".

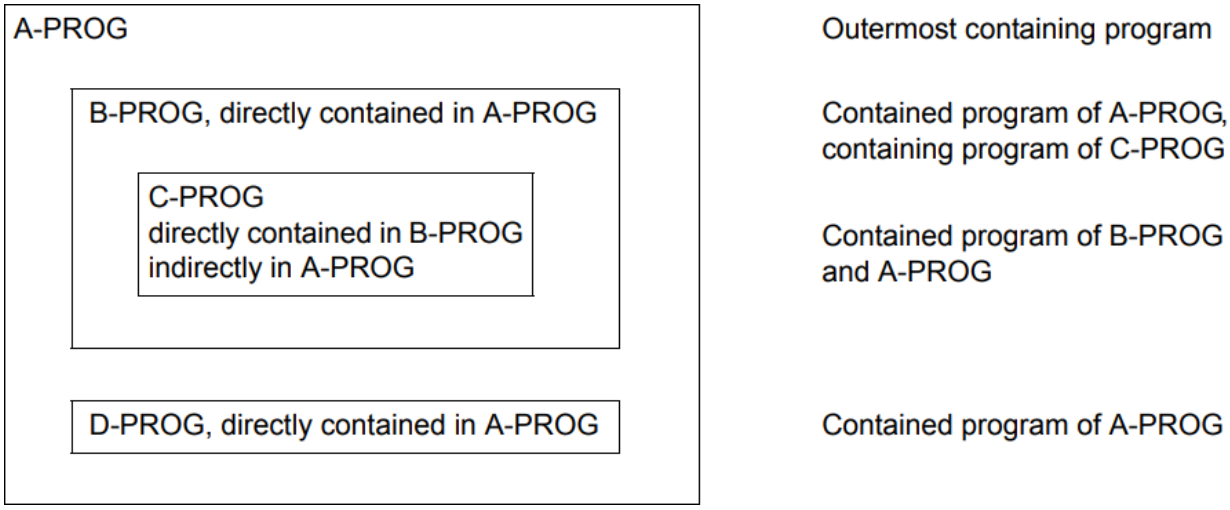
The programs of a nested program can be both containing programs and contained programs, apart from the "outermost" containing program that is treated in exactly the same manner as a separately compiled program within the run unit.

A contained program can be contained directly or indirectly in another program.

With respect to the immediately superordinate nesting level, a contained program is *directly* contained; with respect to other superordinate nesting levels it is *indirectly* contained.

Example 12-1

of the structure of a nested program



"Sibling program"

The programs comprising a nested program that are contained on the same level of nesting in a program are referred to as "sibling programs".

"Descendant"

Each program contained directly or indirectly in a "sibling program" is referred to as a "descendant" of this "sibling program".

Run unit

A run unit is a particular number of executable programs that act as a logical unit at run time.

A run unit may consist of

- one or more separately compiled programs,
- one or more nested programs,
- a combination of separately compiled programs and nested programs.

The program started on the system level is referred to as "main program" and all other programs of the run unit are known as "subprograms".

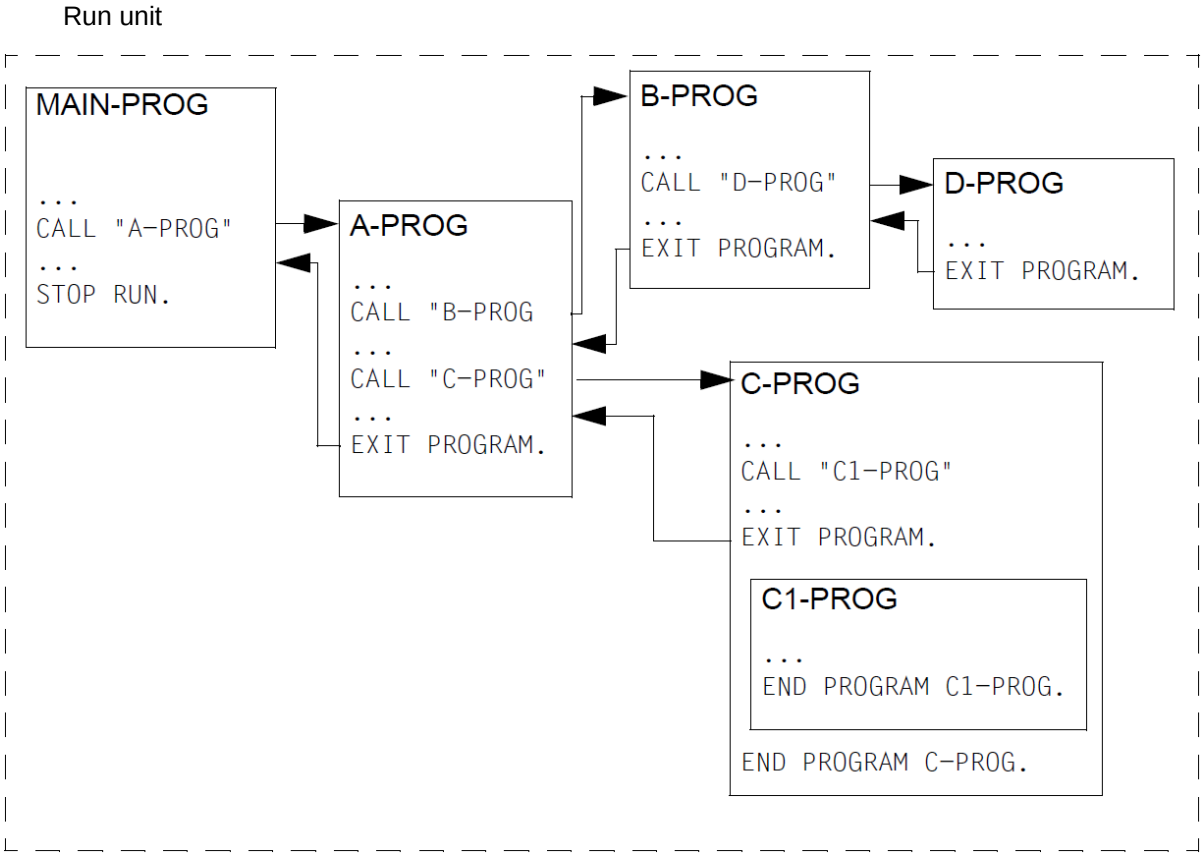
12.4.2 Control of inter-program communication

12.4.2.1 Runtime control

Control of a run unit begins with the program that is called on the system level. Each further program of the run unit is called by means of the CALL statement.
The CALL statement transfers program control to the called program. From there, control is subsequently returned to the calling program by means of the EXIT PROGRAM statement. The program is continued at the statement following the CALL statement.

The following example illustrates the logical structure of a run unit comprising five separately compiled programs, including one nested program:

Example 12-2



12.4.3 Rules for program names

A program is called via the name of the program declared in the PROGRAM-ID paragraph of the IDENTIFICATION DIVISION.

The following basic rules apply to program names:

1. A program name may be referenced only by the CALL statement, the CANCEL statement and the end program header.
2. All separately compiled programs of a run unit must have different program names.
3. All programs of a nested program must have different names.
4. The contained programs of a nested program are "invisible" to all the separately compiled programs of a run unit; i.e. a separately compiled program cannot call any contained program of another separately compiled program.
5. The contained programs of a nested program may have the same name as the separately compiled programs of the run unit. The procedure provided for determining the valid program name in this instance is discussed in the following section ("Selecting the valid program name").
6. Within a nested program, one program can normally only call another program that is contained directly in it.
7. The call facilities in a nested program can be expanded with respect to the normal case when the COMMON attribute is applied to a contained program by means of the COM-MON clause in the PROGRAM-ID paragraph. A program provided with the COMMON attribute can be called not only by the directly superordinate program but also by any "sibling program" and its "descendants" (see "Common clause" on PROGRAM-ID paragraph).

Selecting the valid program name

When a program is called in a nested program, the relevant valid program name is then selected from the sum total of all the program names present in the run unit in accordance with the following rules of precedence:

1. The call applies to the name of a program that is directly contained in the calling program.
2. If 1) does not apply, the call applies to a COMMON program, i.e. to a program that can be called by its "sibling programs" and their "descendants".
3. If neither 1) nor 2) applies, the call applies to a separately compiled program of the run unit.

Example 12-3

```
PROGRAM-ID. A-PROG.
...
    CALL "B-PROG".                  B-PROG in A-PROG              (rule 1)
    CALL "C-PROG".                  Separately compiled C-PROG    (rule 3)
    CALL "D-PROG".                  D-PROG in A-PROG              (rule 1)
   PROGRAM-ID. B-PROG COMMON.
   ...
       CALL "D-PROG".               Separately compiled D-PROG    (rule 3)
       CALL "C-PROG".               C-PROG in B-PROG              (rule 1)
      PROGRAM-ID. C-PROG.
      ...
          CALL "B-PROG".            Separately compiled B-PROG    (rule 3)
      END PROGRAM C-PROG.
   END PROGRAM B-PROG.

   PROGRAM-ID. D-PROG.
   ...
       CALL "B-PROG".               B-PROG in A-PROG              (rule 2)
       CALL "C-PROG".               Separately compiled C-PROG    (rule 3)
   END PROGRAM D-PROG.
END PROGRAM A-PROG.
```

This example illustrates selection of the relevant valid program name.

A CALL statement for a program "A-PROG" within this nested program would be inadmissible since this call would apply to a further separately compiled program called "A-PROG" that may not be present in the run unit.

12.4.4 Initial state for inter-program communication

The INITIAL attribute is created by specifying the INITIAL clause in the PROGRAM-ID paragraph of the program (see "Initial clause").

Initial state means:

- The program's internal data defined in the WORKING-STORAGE SECTION or LOCAL-STORAGE SECTION is initialized. If the VALUE clause is specified for a data item, the item is set to the defined value. If no VALUE clause is specified, the initial value of the data item is undefined.
- Files (contained files) belonging to the program are closed.
- The control mechanisms for all the PERFORM statements contained in the program are set to their initial state.
- A GO TO statement which refers to an ALTER statement in the same program is set to its initial state.

A program is in the initial state:

1. when it is called for the first time in a run unit,
2. when it is called for the first time since either it or a program in which it is directly or indirectly contained was the object of a CANCEL statement,
3. whenever it is called if it has the INITIAL attribute,
4. when it is contained directly or indirectly in a program that has the INITIAL attribute and when it called for the first time after this program has been called.

Example 12-4

```
PROGRAM-ID. A-PROG.
...
    CALL "B-PROG".                    Statement 1
    CALL "B-PROG".                    Statement 2
...
  PROGRAM-ID. B-PROG INITIAL.
  ...
      CALL "C-PROG".                  Statement 3
      CALL "D-PROG".                  Statement 4
  ...
    PROGRAM-ID. C-PROG COMMON.
    ...
    END PROGRAM C-PROG.

    PROGRAM-ID. D-PROG.
    ...
        CALL "C-PROG".                Statement 5
        CANCEL "C-PROG".
        CALL "C-PROG".                Statement 6
    ...
    END PROGRAM D-PROG.
  END PROGRAM B-PROG.
END PROGRAM A-PROG.
```

Statement 1: Program B-PROG is in the initial state (rules 1 and 3).

Statement 2: Program B-PROG is in the initial state (rule 3).

Statement 3: Program C-PROG is in the initial state (rules 1 and 4).

Statement 4: Program D-PROG is in the initial state (rules 1 and 4)

Statement 5: Program C-PROG is not in the initial state.

Statement 6: Program C-PROG is in the initial state (rule 2).

12.4.5 Using common data

12.4.5.1 External and internal data

Data items and files defined in a program may be internal or external data.

Internal data can only be accessed within the program in which this data is defined.
External data can be accessed by any program in the run unit.

Internal data occupies a storage area associated exclusively with the particular program in which the data was defined. External data, on the other hand, occupies a storage area associated with the run unit. Data defined as external in the programs of a run unit can be accessed by all programs of the run unit.

External files and data items are created through specification of the EXTERNAL clause in the FILE SECTION or WORKING-STORAGE SECTION (see section "EXTERNAL clause").

Example 12-5

A run unit comprises three programs in which certain data is defined as external. The resulting storage occupancy is illustrated by the following diagram:

Storage area Program A
Storage area Program B
Storage area Program C
Storage area External data

All internal data, i.e. data not defined as being external, is contained only in the storage area of the program in which it is defined. All data defined as external, on the other hand, is contained only in the storage area for external data.

12.4.5.2 Local and global names

The names used in *nested* programs may be classified as "local" and "global" names.

Local names

Local names are valid only within the program in which the associated data is described. Most of the names that are local as standard can be declared as global. The following names are always local:

- paragraph-names
- section-names

Global names

Global names are valid not only in the program in which they are defined but also in that program's contained programs.
The following name types are always global:

- Names defined in the CONFIGURATION SECTION:

  computer-name
  alphabet-names
  class-names
  condition-names (SPECIAL NAMES paragraph)
  mnemonic-names
  symbolic characters
  CURRENCY SIGN character
  DECIMAL-POINT
- COBOL special registers:

  TALLY
  PRINT-SWITCH
  SORT registers
  RETURN-CODE

The following name types are local as standard and can be explicitly defined as global in nested programs:

- condition-names
- data-names
- file-names
- index-names
- record-names
- report-names
- type-names

Definition of a name as global is performed using the GLOBAL clause (see section "GLOBAL clause" of chapter "File description" and section "GLOBAL clause" of chapter "Data description entry").

In the case of type names the GLOBAL phrase refers to the definition of the type and not to the data description generated with the aid of the type definition.

Determining the valid name

When a name is referenced, the valid name is selected from the sum total of all the names defined in the nested program, in accordance with the following rules of precedence:

1. The referenced name is defined in the same program.
2. If 1) does not apply, the referenced name is defined as a *global* name in the *directly* superordinate (next outer) program.
3. If neither 1) nor 2) applies, the referenced name is defined as a *global* name in the *indirectly* superordinate program.

   Condition 3) is checked until the data description entry of the referenced name is found in one of the other indirectly superordinate programs, possibly in the outermost containing program.

Example 12-6

```
PROGRAM-ID. A-PROG.
...
01  A1   PIC X  GLOBAL.
01  B1   PIC X  GLOBAL.
01  C1   PIC X  GLOBAL.
...
  PROGRAM-ID. B-PROG.
  ...
  01  A1   PIC X  GLOBAL.
  01  B1   PIC X.
  ...
      MOVE  "A"  TO  A1.          A1 in program B-PROG (rule 1)
      MOVE  "B"  TO  B1.          B1 in program B-PROG (rule 1)
      MOVE  "C"  TO  C1.          C1 in program A-PROG (rule 2)
  ...
    PROGRAM-ID. C-PROG.
    ...
        MOVE  "X"  TO  A1.        A1 in program B-PROG (rule 2)
        MOVE  "Y"  TO  B1.        B1 in program A-PROG (rule 3)
        MOVE  "Z"  TO  C1.        C1 in program A-PROG (rule 3)
    ...
    END PROGRAM C-PROG.
  END PROGRAM B-PROG.
END PROGRAM A-PROG.
```

Example 12-7

```
PROGRAM-ID. A-PROG.
...
01 T1 TYPEDEF STRONG GLOBAL.
   02 NAME PIC X(30).
   02 STREET PIC X(80).
...
01 A1 TYPE T1.                1)
01 B1 TYPE T1 GLOBAL.
...
PROGRAM-ID. B-PROG.
...
01 T1 TYPEDEF STRONG.         2)
   02 SUBSTRUKTUR.
     05 NAME-3 PIC X(30).
     05 STREET-3 PIC X(80).
...
01 C1 TYPE T1.                3)
...
MOVE B1 TO C1.                4)
...
END PROGRAM B-PROG.
END PROGRAM A-PROG.
```

1. Uses type definition T1 from A-PROG;
   Definition A1 is local in A-Prog, definition B1 is global
2. Type T1 is local in B-PROG
3. Uses type definition T1 from B-PROG
4. Allocation is permissible as T1 in A-PROG is equivalent to T1 in B-PROG

12.4.6 Language elements for inter-program communication

12.4.6.1 Overview

The language elements relevant to inter-program communication are summarized in the following table:

Language element	Function
END PROGRAM entry	Denotes the end of the named compilation unit
INITIAL clause	The program is set to its initial state each time it is called
COMMON clause	The program can also be called by its sibling programs and their descendants
LINKAGE SECTION	In the called program: to define data passed from the calling program
EXTERNAL clause	Declaration of files and data items as external
GLOBAL clause	Declaration of names as global
PROCEDURE DIVISION USING phrase	In the called program: definition of the standard entry point List of data-names; indicates that data is passed to the called program
CALL statement USING phrase	In the calling program: call for a subprogram or a contained program List of data-names; indicates that data is transferred to the called program
CANCEL statement	The program is set to its initial state the next time it is called
ENTRY statement USING phrase	Only in the subprogram of a run unit: definition of a non-standard entry point List of data-names (defined in the LINKAGE SECTION); indicates that data is transferred from the calling program
EXIT PROGRAM statement	Return of control to the calling program
GOBACK statement	Identifies the logical end of a program
USE statement with GLOBAL attribute	In nested programs: definition of global USE procedures

12.5 Sorting records

Records can be sorted in two different ways:

- sorting of records contained in a file (file sorting)
- sorting of records contained as elements in a table (table sorting).

Both sorting methods use the BS2000 utility SORT for sort processing.

File sorting is described below, table sorting is described in section "SORT statement".

12.5.1 Sorting and merging files

12.5.1.1 Sort processing

The user can sort a file on the basis of a series of keys. These sort keys are specified by the user and are present in each record of the file. The records may be sorted so that all of their keys are in ascending or in descending order, or so that some of their keys are ascending and others are descending.

Records are sorted on a file called a sort-file. This file is defined in the Data Division in a sort-file description (SD) entry, and in the record description associated with the SD. The SORT statement in the Procedure Division initiates the sort operation.

Sort processing consists of:

1. Releasing all records to the sort-file.
2. Sorting the records on the sort-file.
3. Returning all records from the sort-file.

The user may either provide an input procedure to process records and transfer them to the sort-file, or he may specify an input file containing the records to be sorted and allow the SORT statement to transfer these records to the sort-file. Accordingly, he may either provide an output procedure to retrieve records from the sort-file and process them further, or he may specify an output file and allow the SORT statement to output the sorted records to this file.

Multiple sort operations may be specified in a single program.

12.5.1.2 Merge processing

The user has the capability of merging from two to 16 sorted input files with the same record format and transferring them to an output file.

Merging takes place in a file called a sort-file. This file is defined in the Data Division within a sort-file description (SD) entry, and in the record description associated with the SD. Merge processing is initiated by means of the MERGE statement in the Procedure Division. The merge operation is performed in three stages:

1. Release the records of all input files to the sort-file.
2. Merge the records in the sort-file.
3. Return all records from the sort-file.

The MERGE statement releases the records of all specified input files to the sort-file. The user can specify an output file into which the MERGE statement returns the merged records. In connection with returning, it is also possible to specify an output procedure. This procedure accepts records from the sort-file and continues processing them.Several merge operations may be specified in a single program.

12.5.1.3 Sort and merge without input/output procedures

If no input or output procedures are specified, the compiler will generate procedures to take charge of record input /output. In this case the user must describe the following files:

1. One or more input files containing the records to be sorted or merged,
2. One sort-file and one or more output files to which the records will be returned.

When referenced, a SORT or MERGE statement performs the following functions:

1. Opens the input and sort-files.
2. Releases all the records in the input file to the sort-file.
3. Closes input files.
4. Sorts or merges the records on the sort-file.
5. Opens output files.
6. Returns the records from the sort-file to the output file.
7. Closes the sort and output files.

12.5.1.4 Sort with input/output procedures

If input and output procedures are specified, the following actions are taken:

1. When a SORT statement is executed, it transfers control to the input procedure.
2. The input procedure performs the following functions:
    a. Processes a record (for example, reads a record from a file or creates a new record).
    b. Releases the record to the sort-file.
    c. Repeats steps a) and b) until all records have been released.
3. The SORT statement sorts the records on the sort-file.
4. Control is then passed to the output procedure.
5. The output procedure performs the following functions:
    a. Accepts a record from the sort-file.
    b. Processes the record (for example, writes the record to an output file).
    c. Repeats steps a) and b) until all records have been returned and processed.
6. Control is returned to the statement following the SORT statement.

If options are mixed (e.g. if only an input procedure is specified), appropriate variants of the above procedures are performed.

12.5.1.5 Overview of language elements

To use the sort feature, the user must provide additional information in the Environment, Data, and Procedure Divisions of the compilation unit. This information is summarized in the following table and is described in detail in the remainder of this section.

<table>
<tr><th>compilation unit division</th><th>Contents and meaning</th></tr>
<tr><td rowspan="3">Environment division</td><td>A SELECT clause in the FILE-CONTROL paragraph for the sort-file (sort-file-name).</td></tr>
<tr><td>SELECT clauses in the FILE-CONTROL paragraph for all files used as input and output procedures for the sort-file (section-name-1, section-name-2, section-name-3, section-name-4 for SORT; section-name-1, section-name-2 for MERGE) and for files appearing in the USING or GIVING phrase of a SORT or MERGE statement (file-name-1,...).</td></tr>
<tr><td>To permit restart of programs containing the SORT statement, the RERUN clause may be used in the I-O-CONTROL paragraph.</td></tr>
<tr><td></td><td>The SAME SORT AREA or the SAME SORT-MERGE AREA clause in the I-O-CONTROL paragraph is intended to optimize the allocation of internal storage to a sort-file.</td></tr>
<tr><td rowspan="3">Data Division</td><td>A sort-file description (SD) entry and associated record description entries for the sort-file; record descriptions must include sort-key fields.</td></tr>
<tr><td>File description (FD) entries and associated record description entries for all files used in input /output procedures for the sort-file section-name-1, section-name-2, section-name-3, section-name-4 for SORT; section-name-1, section-name-2 for MERGE) and for files appearing in the USING or GIVING phrase of a SORT or MERGE statement (file-name-1,...).</td></tr>
<tr><td>Data description entries for the above files.</td></tr>
<tr><td rowspan="6">Procedure Division</td><td colspan="2">A SORT or MERGE statement for the sort-file specifying the following information:
Sort-key names
Whether the sort is to be in ascending or descending order of key.
Input/output information specified by the following options:</td></tr>
<tr><td>Specification</td><td>Meaning</td></tr>
<tr><td>INPUT PROCEDURE</td><td>Records will be released to the sort-file by an input procedure.</td></tr>
<tr><td>USING file-name-1,...</td><td>Specifies files from which the SORT statement will accept records to be released to the sort-file.</td></tr>
<tr><td>OUTPUT PROCEDURE</td><td>Records will be returned from the sort-file by an output procedure.</td></tr>
<tr><td>GIVING file-name-3,...</td><td>Specifies a file to which the SORT and MERGE statements will return sorted and merged records, respectively.</td></tr>
</table>

	If input and/or output procedures are specified in the SORT or MERGE statement, these procedures must be included in the Procedure Division. An input procedure must include a RELEASE statement, to transfer records to the sort-file. An output procedure must include a RETURN statement, to accept records from the sort-file.
	Special sort-registers can be referenced in the Procedure Division (see section "Special registers for files: SORT").

12.5.2 Special registers for files: SORT

In the compiler, four special registers are provided for communication between the programmer and the SORT control routine. Each of these registers is a 4-byte binary data item with a fixed name; the description of each register is PICTURE S9(8) USAGE IS COMPUTATIONAL. The data description entries for the registers are generated automatically by the compiler and cannot be declared by the programmer.

SORT-FILE-SIZE register

The programmer may set the contents of the SORT-FILE-SIZE register to the approximate number of records to be processed in the next sort. This must be done before the SORT statement is executed. From this information, the sort routine calculates how much space it will need on internal working files. SORT-FILE-SIZE is initially set to zero by the compiler; and, if the value of the register is still zero at the time a sort begins, the SORT control routine will make a default space allocation. Thus, the programmer does not have to enter any estimate of file size into this special register, although supplying such information aids the efficiency of the sort. The value entered in the special register by the user (e.g. MOVE 25 TO SORT-FILE-SIZE) is released to the control routine.

SORT-CORE-SIZE register

The programmer may set the contents of the SORT-CORE-SIZE register to the number of bytes of memory that are to be available for use by the SORT routine. This must be done before the SORT statement is executed.

SORT-MODE-SIZE register

SORT-MODE-SIZE may be set if variable-length records are to be sorted.

The programmer may set this register to the most commonly used record length in the input file. The value may not be less than the lowest value and not greater than the highest value in the sort-file description. The appropriate value must be moved into SORT-MODE-SIZE before the execution of the SORT statement. The compiler initializes the contents of this register to zero; and, if the programmer has not supplied any other value before the SORT statement is executed, the maximum record length will be assumed to be the most common record length.

SORT-RETURN register

After a SORT statement or RELEASE/RETURN statement has been executed, the special register SORT-RETURN contains a value to indicate whether the sort has been successful. A value of zero indicates that the sort was successful. A non-zero value indicates that the sort terminated abnormally.

Summary

None of the special registers (except SORT-RETURN) has its value reset or set to zero by the execution of the SORT statement. If a program contains several sorts, the programmer must move appropriate values into the SORT-FILE-SIZE, SORT-CORE-SIZE and SORT-MODE-SIZE registers before each sort is executed.

It should be noted that the special registers are binary items and, therefore, cannot be used as immediate operands of ACCEPT statements.

Example 12-8

```
ACCEPT MODE-SIZE FROM MASTER-FILE.
MOVE MODE-SIZE TO SORT-MODE-SIZE.
```

Since the ACCEPT statement transfers input data from the terminal without conversion, the MOVE statement is used to achieve conversion to COMPUTATIONAL.

12.5.3 Sorting two-digit year numbers with a century window

General description

This description applies both to file sorting and to table sorting. The century window is defined in the same way as in the COBOL function YEAR-TO-YYYY. The last year belonging to the window is specified relative to the current year. The specified value determines the number of years in the future which belong to the century window. For example, the value 50 when run in 1998 represents the period from 1949 through 2048.

A SORT special register SORT-EOW (SORT-END-OF-WINDOW) is defined to determine the century window. It is made available in COBOL programs with the SORT or MERGE statement, and implicitly described by the compiler with PIC 9(7) PACKED-DECIMAL. The value saved in SORT-EOW must be between 0 and 99. The default value is 50.

The ASCENDING/DESCENDING phrases in the SORT and MERGE statements have been extended to enable two-digit year numbers to be used as a SORT key depending on a century window.

In the MERGE statement the position of the century window is determined at the start of processing (evaluation of SORT-EOW and current year). If, for example, files are first sorted with the SORT statement, the same century window must be selected.

Example 12-9

SORT with century window selection

```
IDENTIFICATION DIVISION.
PROGRAM-ID.SORTIERY.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
   SELECT MASTER-FILE ASSIGN TO "MASTER-FILE".
   SELECT OUTPUT-FILE ASSIGN TO "OUTPUT-FILE".
   SELECT SORT-FILE ASSIGN TO "SORTWK".
DATA DIVISION.
FILE SECTION.
FD MASTER-FILE.
01 MASTER-RECORD.
   02 E0      PIC X.
   02 EY1     PIC 99.
   02 EY2     PIC 99 USAGE PACKED-DECIMAL.
   02 E3      PIC X(10).
FD OUTPUT-FILE.
01 OUTPUT-RECORD.
   02 A0      PIC X.
   02 AY1     PIC 99.
   02 AY2     PIC 99 USAGE PACKED-DECIMAL.
   02 A3      PIC X(10).

SD SORT-FILE.
01 SORT-RECORD.
   02 S0      PIC X.
   02 SY1     PIC 99.
   02 SY2     PIC 99 USAGE PACKED-DECIMAL.
   02 S3      PIC X(10).
PROCEDURE DIVISION.
P1 SECTION.
SORT.
```

```
* Specification of the century window: starting from the
* year 1998, 2008 is the final year of the century window

      MOVE 10 TO SORT-EOW

* The keys SY1 and SY2 are handled as two-digit year
* numbers within the century window of 1909-2008:
* 06 is greater than 75

     SORT SORT-FILE ASCENDING KEY-YY SY1 SY2
                 DESCENDING KEY S3
     USING MASTER-FILE GIVING OUTPUT-FILE.
SORTEND.
     STOP RUN.
```

12.5.4 Sorting with extended character sets (XHCS)

The following description applies to both file and table sorting.

To specify the character set of alphanumeric sort keys, a SORT special register SORT-CCSN ("Coded Character Set Name“) is defined. It is made available in COBOL programs that contain a SORT statement and described implicitly by the compiler with PIC X(8). This SORT special register is supplied with the name of the selected character set before the call of the SORT statement (e.g. MOVE "EDF041" TO SORT-CCSN).

If this option is not to apply to a subsequent SORT statement, the value of the new SORT special register SORT-CCSN must be filled with spaces before this SORT statement is executed (e.g. MOVE SPACES TO SORT-CCSN).

The specifications for the sort sequence apply in the following order:

- specification of the SORT COLLATING SEQUENCE in the SORT statement or PROGRAM COLLATING SEQUENCE in the program
- value in the SORT special register SORT-CCSN
- COMOPT SORT-EBCDIC-DIN or SDF option SORTING-ORDER

Example 12-10

SORT with extended character sets (XHCS)

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SORTX.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
SELECT INPUT ASSIGN TO "INPUT".
SELECT OUTPUT ASSIGN TO "OUTPUT".
SELECT SORT ASSIGN TO "SORTWK".
DATA DIVISION.
FILE SECTION.
FD INPUT LABEL RECORD STANDARD.
01 INPUT-RECORD.
02 I0 PIC X.
02 NAME     PIC X(30).
02 FORENAME PIC X(30).
02 PLACE PIC X(30).
FD OUTPUT- LABEL RECORD STANDARD.
01 OUTPUT-RECORD
02 O0 PIC X.
02 NAME     PIC X(30).
02 FORENAME PIC X(30).
02 PLACE PIC X(30).
SD SORT LABEL RECORD STANDARD.
01 S-RECORD.
02 S0 PIC X.
02 NAME     PIC X(30).
02 FORENAME  PIC X(30).
02 PLACE PIC X(30).
PROCEDURE DIVISION.
P1 SECTION.
SORT.
MOVE "EDF03IRV" TO SORT-CCSN.              (1)
```

```
SORT SORT ASCENDING NAME FORENAME PLACE
USING INPUT GIVING OUTPUT.
STOP RUN.
```

(1) Supplies the SORT special register SORT-CCSN with the name of the Extended Character Code Set; this code is used to specify the sort sequence.

12.6 Character representation by UTF-16

Characters which are processed by a COBOL program can be represented by different character sets. For this purpose the language defines the two data classes alphanumeric and national.

COBOL2000 represents alphanumeric characters in the EBCDIC character set and national characters in UTF-16. Data of the national class is largely defined and used in the same way as the data of the alphanumeric class.

12.6.1 National data

A national character is described in the PICTURE clause by the symbol 'N'. This specifies the number of characters, i.e. not the number of occupied bytes. A national character occupies 2 bytes and, in data structures, is aligned on the byte boundary.

Reference modification is also permitted for national data. In this case the characters also count for the start position and the length.

Example 12-11

```
01 nat    PIC N(30).
```

defines a data item for 30 characters in UTF-16 representation, i.e. 60 bytes

```
nat(3:6)
```

selects the 3rd to 6th characters of nat, i.e. bytes 5 through 12

12.6.2 Data structures, clauses

The national data format can also be specified explicitly as NATIONAL for the PICTURE clause in the USAGE clause.

National data may also be grouped to form data structures. Here it must be taken into account that group items in COBOL always have the class alphanumeric. As a result a group item is treated like an alphanumeric data item even if it only contains elementary national items, e.g. when filling with blanks. The GROUP-USAGE NATIONAL clause causes a group item to be treated like a national data item.

Example 12-12

```
01 alfanum-structure.
  02 nat1 PIC N(10).
  02 nat2 PIC N(10).
01 national-structure GROUP-USAGE NATIONAL.
  02 nat1 PIC N(10).
  02 nat2 PIC N(10).
```

The first data structure is always alphanumeric. If a shorter sending item is moved there, alphanumeric blanks (X'40') are appended, which are not meaningful in national data items. The second data structure, on the other hand, is national because of the additional clause. National blanks (X'0020') would be appended.

12.6.3 National literals

A national literal may describe up to 90 characters. In the case of a national literal, the opening literal delimiter consisting of the letter N followed by a quotation mark is provided to distinguish alphanumeric literals. As a COBOL source text is available in EBCDIC, only EBCDIC characters can be written as the literal content. The compiler automatically replaces these by the corresponding UTF-16 representation.

In addition, there is also the hexadecimal variant, introduced by the two letters NX, followed by a quotation mark. Here every character must be written directly in its UTF-16 representation in the form of 4 hexadecimal digits. This also enables national literals to be specified which have no equivalent in the EBCDIC character set.

Example 12-13

```
N'AB '
NX"004100420020"
```

both describe the same string in UTF-16 representation.

The figurative constants are a particular form of literal.
There are also national forms of these:

- The keyword ALL followed by a national literal.
- SPACE, QUOTE, ZERO, HIGH-VALUE and LOW-VALUE which are defined as keywords. Depending on the context in which they are used, they stand for the corresponding alphanumeric or national literals.

Example 12-14

```
01 alfa PIC X.
01 nat PIC N.
MOVE SPACE to alfa, nat.
```

supplies the alfa item with an alphanumeric blank (X'40') and the nat item with a national blank (X'0020').

12.6.4 Moving national data items

National data items are moved using the MOVE statement. In the case of operands of different classes, the content of an alphanumeric item can be converted implicitly to the corresponding national representation. However, this is not possible in the other direction. If any national receiving item is longer, it is then filled with national blanks (X'0020') when the move takes place.

Example 12-15

```
01 a  PIC XXX VALUE "123".
01 s.
  02 nat1  PIC N(2).
  02 nat2  PIC N(2).
01 n  PIC N(4) JUSTIFIED RIGHT.
MOVE a TO s, n.
```

after the MOVE, a, s and n have the following content (in hexadecimal notation):

a: F1F2F3

s: F1F2F34040404040

n: 0020003100320033'

12.6.5 National data items in conditions

In a relation condition a national data item may be compared with operands of various classes. In this case, data item contents can also be converted implicitly to the corresponding national representation.

For this comparison, non-national operands are treated as if they had been moved to a national data item of the same size using MOVE. The national data is then compared. To do this, the shorter operand is filled on the right with national blanks up to the length of the longer operand.

Example 12-16

```
01 num     PIC 9(4) VALUE 1860.
01 alfa      VALUE "TSV MÜNCHEN".
01 nat.
  02 nat1  PIC NNN VALUE N"TSV".
  02 nat2  PIC NNNN VALUE N"1860".
IF num = nat2 THEN ...                (1)
IF alfa = nat1 THEN ...               (2)
```

(1) The numeric field num is moved to a 4-character long national relational item ein (defined as NNNN) and compared with nat2. The comparison is true.

(2) The alphanumeric item alfa is converted to a national relational item (defined as N(11)). This is compared with a relational item (also defined as N(11), whose content corresponds to nat1 supplemented by 8 national blanks). The comparison is false.

Individual national characters are always compared on the basis of the binary representation of these characters. This also applies for the SORT and MERGE statements. The extended options in the case of a comparison in accordance with Unicode are not supported (culturally sensitive, normalization).
It must also be borne in mind that a relation operation which exists between 2 alphanumeric characters need not exist in precisely the same form between the two corresponding national characters (e.g. "A" < "1", but N"A" > N"1").

12.6.6 Conversions between EBCDIC and UTF-16 representation

In addition to the implicit conversions in the case of moves and relation conditions, conversions can be performed explicitly with the aid of FUNCTIONs:

- NATIONAL-OF returns the national representation (UTF-16) of an alphanumeric argument
- DISPLAY-OF returns the alphanumeric representation (EBCDIC) of a national argument

As an optional second argument, both functions permit a replacement character to be specified which is used in place of characters which have no equivalent in the other character set.

Example 12-17

```
01 n  PIC N(3) VALUE NX"E23A00410040".
01 a  PIC X VALUE "?".
FUNCTION DISPLAY-OF (n, a)
```

The function call returns an alphanumeric data item containing "?A@" because the first national character has no EBCDIC equivalent. The replacement character "?" is used for this in the result.

12.6.7 Error handling in the event of conversion

If a character cannot be represented in the other character set when implicit or explicit conversion takes place (by calling the FUNCTIONs without a second parameter) because there is no equivalent, the exceptional condition EC-DATA-CONVERSION occurs. In this case the replacement character "period" (".") which is defined in the system is used.

If the check for the exception condition is enabled by the associated TURN directive, the exceptional condition is triggered and can, if required, be treated in a corresponding USE procedure.

Example 12-18

```
01 n  PIC N VALUE NX"E23A".
...
IF  FUNCTION DISPLAY-OF(n) = "a"    (1)
THEN ...
ELSE ...                            (2)
END-IF                              (3)
```

(1) The exceptional condition EC-DATA-CONVERSION occurs (NX'E23A' has no EBCDIC equivalent):

- If the check for the exceptional condition is enabled with >>TURN EC-DATA-CONVERSION CHECKING ON and a USE procedure exists which is exited with RESUME AT NEXT STATEMENT, the procedure continues after (3).
- If the check is not enabled or no corresponding USE procedure exists, the procedure continues with (2) because the replacement character used (period, ".") is not equal to "a".

12.7 Object-oriented concepts

12.7.1 Fundamentals of object-oriented programming

Object-oriented software development is based on some key techniques that make it possible to design software systems that are easy to understood, modify and reuse.
The most important among them are:

- defining objects that exchange messages
- building classes primarily intended for general use
- use of inheritance as a structuring mechanism and a means of avoiding multiple implementations
- use of so-called virtual methods such as polymorphism and “late binding” for concrete specialization.

A classically programmed application system essentially consists of an algorithm and a data structure. In an object-oriented application system, by contrast, the data can only be indirectly modified by functions.
Some of the basic concepts of object-oriented programming are explained below.

Objects

An object is defined by a class and consists of a combination of data and functions. These functions are called methods of the object. Objects communicate with one another by exchanging messages, which in turn are sent via calls to methods.
Whenever an object is created, it is always assigned an object reference value that uniquely identifies it for the lifetime of the object.

Classes

Classes are defined by class definitions and designate a set of objects with their attributes and methods.

Object references

An object reference is an implicitly or explicitly defined data item. The contents of the object reference uniquely references an object and its associated information.
Implicitly defined object references are the predefined object references and object references returned from an object view. Explicitly defined object references are data items defined by a data description entry specifying a USAGE OBJECT REFERENCE clause.

Predefined object references

A predefined object reference is an implicitly generated data item referenced by one of the identifiers NULL, SELF or SUPER.

Object instances

Whenever a new object is to be generated, the system creates a new copy of the attributes of the class and thus creates an object instance. The methods of the class then apply to this new object as well.

Factory objects

In OO-COBOL, every class contains one and only one special object, the so-called factory object, which is described by the FACTORY definition of the class. The factory object is responsible for creating object instances (objects) of a class.

Methods

Methods are operations, functions and statements that can be executed by an object.

Inheritance

The basic idea behind inheritance is to use a class hierarchy, which is structured from the “general” to the “special”, so that classes which are “lower” in the hierarchy can inherit the methods and attributes from the classes above them. The concept of inheritance helps prevent the redundant duplication of code for similar applications.
A class may inherit from more than one other class; this is called **multiple inheritance**.

Polymorphism

Polymorphism means that the same message sent to different objects can trigger different actions depending on the type of object, i.e. can call different methods, provided their interface is identical.

Interfaces

Every object has an interface consisting of the name and parameter specifications for every method of an object, including the inherited methods.
Every class has two interfaces: one for the factory object and the other for the remaining objects.

Interface

OO-COBOL provides the language element Interface for defining an interface - independently of a concrete object or class. Here it is not necessary to record the complete object interface in an Interface. Rather, with the aid of an Interface which only defines the common part of the interfaces of unrelated classes, it is possible to process objects of these different classes in a uniform manner.

Conformance

Conformance is a unidirectional relation from one interface to another interface and from an object to an interface. Conformance is one of the underlying principles that enables fundamental features such as inheritance and interface definitions, for example.
If there is conformance between two classes, all interfaces of one class can also be used in the other class.
Conformance is bound to specific rules, which are generally checked at compile time. Note, however, that if an object view is used or a method for a universal object reference is called, this check occurs only at runtime.

Schematic examples

Example 12-19

for polymorphism, late binding

```
Class A                          {M(A)} Set of available methods
  Method M
    Display ’AAA’

Class B inherits A               {M(B)} Set of available methods
  Method M override
    Display ’BBB’

User program
01 aref usage object reference A     a)
```

```
invoke B 'NEW' returning aref      b)
invoke aref 'M'                    c)

           Output --> BBB

invoke A 'NEW' returning aref      d)
invoke aref 'M'

           Output --> AAA
```

Notes

a) An object reference that can point to objects of class A and all classes that inherit from it.

b) aref points to a class B object.

c) The method actually called need not be one of those available in the class specified on defining the object reference (which would be class A and thus the method $M_{(A)}$), but will depend on the content of the object reference at the time of invoking the method: the method is one that is available in the class of the current object and could be either defined in the class itself or inherited from an object class.
In this case, this is the method $M_{(B)}$.

d) Here, aref now points to a class A object, and an “externally” identical “invoke” as in c) produces a different output, since the current object is a different object.

Useful tips

- This example illustrates the concept of a “late binding”, since the assignment of the current method to the method call can only occur at runtime, i.e., when the current contents of the object reference (and thus the class in which the method is to be found) is determined.
- This effect occurs only when a method of a super class is overwritten in an inheriting class. Otherwise, the called method is the one from the super class.
- The important point to remember is that besides the identical method names, even the interface of the corresponding methods must be the same (so that a subclass object can be addressed like an object of its super class even on the part of the interface). This is what effectively produces the conformance and a form of polymorphism.

Example 12-20

for SELF

```
Class A                 {M(A),N(A)} Set of available methods
  Method M
    Display 'AAA'
  Method N
    invoke SELF 'M'                    c)

Class B inherits A      {M(B),N(A)} Set of available methods
  Method M override
    Display 'BBB'
```

```
User program
01 aref usage object reference A

  invoke B 'NEW' returning aref         a)
  invoke aref 'N'                       b)

         Output --> BBB
```

Notes

a) The current object is a class B object.

b) Method $N_{(A)}$ is called here, since no local N was defined in the inheriting class B (which is the source of the current object).

c) Executing N leads to a further method call. In this case, SELF stands for the object with which the method was called, i.e. an object of class B.
This again results in the situation depicted in the prior example:
The method that applies to an object of class B is the one available in class B, i.e., from the set {$M_{(B)}$, $N_{(A)}$ }, and does not have to be a method that is available in the class containing the invoke with SELF (which would be from the set {$M_{(A)}$, $N_{(A)}$ }).

Useful tips

- This also represents a form of “late binding”.
- If 'no-one' inherits from class A, then the method called with SELF is exactly the same as the one also available in class A.
- By using SELF, methods can invoke other methods for their “current object“, which may not exist at compile time. This feature can be very useful, since it is not always possible to predict at compile time “who” will inherit the corresponding class in the future and possibly replace those methods by methods of its own.

Example 12-21

for SUPER

```
Class A                 {M(A)} Set of available methods
  Method M
    Display 'AAA'

Class B inherits A      {M(B),N(B} Set of available methods
  Method M override
    Display 'BBB'
  Method N
    invoke SUPER 'M'              c)

Class C inherits B      {M(C),N(B} Set of available methods
  Method M override
    Display 'CCC'
```

```
User program
01 aref usage object reference A

  invoke C 'NEW' returning aref     a)
  invoke aref as C 'N'              b)

        Output --> AAA
```

Notes

a) The current object is a C object.

b) The method must be found in the set of methods available for C objects, i.e., in {$M_{(C)}$,$N_{(B)}$ }. Consequently, the methods defined in class B are invoked.

c) This leads to an internal method call for the same object for which method $N_{(B)}$ was invoked (i.e., still the class C object). SUPER indicates that the method is now to be found among the methods available in the super-class (for multiple inherited classes, the appropriate class should be defined with a class name qualification!). This super-class is not relative to the current object, but statically refers to the class containing the method call with SUPER, i.e. class A.
Method M in 3) is thus to be found among the methods available in class A, i.e. the set {$M_{(A)}$ }.

Useful tips

- SUPER enables you to control the method selection by restricting the search to the statically known inheritance hierarchy instead of the methods available for the current object.
- By overwriting a method and using SUPER, it is possible to execute additional operations with a “new” method, while retaining the same interface. Since the methods available in each class must have unique names, and the “new” method makes thereplaced method invisible, a facility to use this overwritten method of the super-class is required. The language element SUPER is used for this purpose.

12.7.2 Parameterized classes and interfaces

A parameterized class is a class with one or more formal parameters. All that is known of a formal parameter is whether a class or an interface is concerned. Concrete characteristics of the formal parameters that go beyond this remain open.

A parameterized class is used by specifying a class K as an expansion of the parameterized class P with current parameters in the REPOSITORY paragraph of the user program: the formal parameters must then be replaced by the names of concrete classes or interfaces (the current parameters), and the name of the parameterized class by that of the concrete expansion. As a result, the new concrete class K, which behaves like a non-parameterized class and is also handled as such, is created at this time from P.

The actual compilation of this new class K with the replaced class and interface names takes place automatically after the user has been compiled (see the "COBOL2000 User Guide" [1]).

The following definitions are required in this context:

- The concrete class K expands the parameterized class P.
- The parameterized class P is expanded as concrete class K.

The process of parameter replacement, the compilation and the resulting class are referred to as “expansion of the parameterized class P”.

Every such expansion of a parameterized class generates a separate concrete class. It has its own factory object and is independent of other expansions in the same parameterized class.
However, the names of all referenced classes must be unique within a run unit: if a parameterized class P is expanded in multiple programs of the run unit with the same name K, the current parameters should also be the same in these explansions.

The advantage of parameterized classes over those which provide the same performance using universal object references instead of parameterization is that the compiler can perform tests statically and once only at compilation time; with the other approach these tests would have to be repeated dynamically at runtime.

The descriptions of parameterized classes provided above apply analogously for parameterized interfaces.

Example 12-22

The following example shows a “first-in-first-out” list for including object references. The class of the references is not to be specified more closely and is thus defined as a parameter. Such a list can incorporate any number of elements, the first element entered in the list being the one that is returned first when the list is read, at the same time being removed from the list.

This list is implemented with the two parameterized classes FIFO-LISTE and LIST-ELEMENT1. The FIFO-LISTE class describes the management structure for the entire list. As a parameter it has the class of the objects to be added and also the class of the corresponding concrete expansion of LIST-ELEMENT1.

The parameterized class LIST-ELEMENT1 describes a single list element. As a parameter it has the class of the objects whose reference is stored in such a list element. LIST-ELEMENT1 functions as an “auxiliary class” which does not appear separately on the outside. It is only used within the FIFO-LISTE and should be generated there automatically using the appropriate parameters.

Nesting of parameterized classes is not permitted by the COBOL Standard 2002, however. Users must therefore program the required expansions themselves.

```
       >>IMP Compiler-Action UPDATE-REPOSITORY ON                        a)
       IDENTIFICATION DIVISION.
>>>    CLASS-ID.               LIST-ELEMENT1 INHERITS BASE USING PAR.   b)
       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.
       SPECIAL NAMES.
           TERMINAL IS TTT.
       REPOSITORY.
           CLASS               PAR,                                     b)
           CLASS               BASE.

FFF    FACTORY.
       PROCEDURE DIVISION.
+++    METHOD-ID.              MAKE-NEW.                                c)
       DATA DIVISION.
       LINKAGE SECTION.
       01 P USAGE OBJECT REFERENCE PAR.
       01 R USAGE OBJECT REFERENCE ACTIVE-CLASS.
       PROCEDURE DIVISION USING BY VALUE P RETURNING R.
       1.
           INVOKE SELF "NEW" RETURNING R.
           INVOKE R "SET-CONTENT" USING P.
           EXIT METHOD.
       END METHOD              MAKE-NEW.
       END FACTORY.

OOO    OBJECT.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01 NXT  USAGE OBJECT REFERENCE LIST-ELEMENT1.                     d)
       01 CONT USAGE OBJECT REFERENCE PAR.                               e)
       PROCEDURE DIVISION.

+++    METHOD-ID.              SET-NEXT.                                f)
       DATA DIVISION.
       LINKAGE SECTION.
       01 P USAGE OBJECT REFERENCE LIST-ELEMENT1.
       PROCEDURE DIVISION USING P.
       1.
           SET NXT TO P.
           EXIT METHOD.
       END METHOD              SET-NEXT.

+++    METHOD-ID.              GET-NEXT.
       DATA DIVISION. LINKAGE SECTION.
       01 R USAGE OBJECT REFERENCE LIST-ELEMENT1.
       PROCEDURE DIVISION RETURNING R.
       1.
           SET R TO NXT.
           EXIT METHOD.
       END METHOD              GET-NEXT.

+++    METHOD-ID.              GET-CONTENT.
       DATA DIVISION.
       LINKAGE SECTION.
       01 P USAGE OBJECT REFERENCE PAR.
```

```
       PROCEDURE DIVISION RETURNING P.
       1.
           SET P TO CONT.
           EXIT METHOD.
       END METHOD               GET-CONTENT.

+++    METHOD-ID.               SET-CONTENT.
       DATA DIVISION.
       LINKAGE SECTION.
       01 P USAGE OBJECT REFERENCE PAR.
       PROCEDURE DIVISION USING BY VALUE P.
       1.
           SET CONT TO P.
           EXIT METHOD.
       END METHOD               SET-CONTENT.
       END OBJECT.
       END CLASS                LIST-ELEMENT1.
```

Notes

a) The expansions of this class are entered as parameters in the expansions of the parameterized class FIFO-LIST. The repository data of the relevant expansion of LIST-ELEMENT1 must therefore be available. This is ensured by specifying the directive (however, the user must ensure that a repository is assigned correctly, see the "COBOL2000 User Guide" [1]).

b) Parameter PAR is the name of the class whose object references are to be stored in the list.

c) The MAKE-NEW method generates a new list element and also supplies the object reference that is to be stored there with values. The list element object is not yet linked to the list, however - that is the task of the list class FIFO-LIST.

d) The list elements are simply linked “forward” to the next element in the list via an object reference.

e) The user data of a list element consists of an object reference for objects of the class which is later specified as the current parameter or classes that inherit from this.

f) The other 4 methods represent the elementary accesses for writing and reading the two data parts of a list element.

The actual list accesses, i.e. the methods APPEND-ENTRY and REMOVE-ENTRY, are made available by FIFO-LIST. The LIST-LENGTH method acts as an additional information service.

```
        IDENTIFICATION DIVISION.
>>>>    CLASS-ID.                FIFO-LIST INHERITS BASE
                                        USING PAR-OBJ, PAR-ELEM.        a)
        ENVIRONMENT DIVISION.
        CONFIGURATION SECTION.
        SPECIAL NAMES.
            TERMINAL IS TTT.
        REPOSITORY.
            CLASS               PAR-OBJ,
            CLASS               PAR-ELEM,
            CLASS               BASE.

OOO     OBJECT.
        DATA DIVISION. WORKING-STORAGE SECTION.
        01 ELEM-COUNT PIC S9(8) COMP-5 VALUE 0.                          b)
        01 FST        USAGE OBJECT REFERENCE PAR-ELEM.                   c)
        01 LST        USAGE OBJECT REFERENCE PAR-ELEM.                   c)
        PROCEDURE DIVISION.

+++     METHOD-ID.               APPEND-ENTRY.                          d)
        DATA DIVISION.
        LOCAL-STORAGE SECTION.
        01 W USAGE OBJECT REFERENCE PAR-ELEM.
        LINKAGE SECTION.
        01 P USAGE OBJECT REFERENCE PAR-OBJ.
        PROCEDURE DIVISION USING P.
        1.
            INVOKE PAR-ELEM "MAKE-NEW" USING P RETURNING W.
            IF LST = NULL
            THEN                     *> MAKE VERY FIRST ENTRY IN LIST
              SET FST, LST TO W
              MOVE 1 TO ELEM-COUNT
            ELSE                     *> APPEND ENTRY TO THE END OF LIST
              INVOKE LST "SET-NEXT" USING W
              SET LST TO W
              ADD 1 TO ELEM-COUNT
            END IF.
            EXIT METHOD.
        END METHOD               APPEND-ENTRY.

+++     METHOD-ID.               REMOVE-ENTRY.                          e)
        DATA DIVISION.
        LOCAL-STORAGE SECTION.
        01 W USAGE OBJECT REFERENCE PAR-ELEM.
        LINKAGE SECTION.
        01 R USAGE OBJECT REFERENCE PAR-OBJ.
        PROCEDURE DIVISION RETURNING R.
        1.
            IF FST = NULL
            THEN                     *> LIST IS EMPTY
              SET R TO NULL
            ELSE                     *> DELETE 1st LIST-ELEM,MAKE SURE
                                     *> THAT IT WILL BE GARBAGE COLLECTED
              INVOKE FST "GET-CONTENT" RETURNING R
              SUBTRACT 1 FROM ELEM-COUNT
              SET W TO FST
              INVOKE W "GET-NEXT" RETURNING FST
```

```
              INVOKE W "SET-NEXT" USING NULL
            END IF.
            EXIT METHOD.
        END METHOD                REMOVE-ENTRY.

+++     METHOD-ID.                LIST-LENGTH.
        DATA DIVISION.
        LINKAGE SECTION.
        01 R PIC S9(8) COMP-5.
        PROCEDURE DIVISION RETURNING R.
        1.
            MOVE ELEM-COUNT TO R.
            EXIT METHOD.
        END METHOD                LIST-LENGTH.
        END OBJECT.
        END CLASS                 FIFO-LIST.
```

Notes

a) Parameter PAR-OBJ is the name of the class whose object references (also those of subclasses) are to be stored in the list.
Parameter PAR-ELEM is the name of the associated expanded auxiliary class LIST-ELEMENT1.

b) This counter is used only for the additional information service LIST-LENGTH. It is not needed for list accesses and the services.

c) To manage the FIFO list it is sufficient if the first (FST) and last (LST) elements in the list can be found.

d) The APPEND-ENTRY method itself generates a new list element with the object reference transferred as a parameter as its content. This new element is the last that is linked to the list and the management data is adjusted accordingly.

e) The REMOVE-ENTRY method removes the first element from the list and returns the object reference stored there. Here it would be sufficient just to adapt the management data - however, to facilitate garbage collection, the link to the list in the list element removed is set to NULL.

The program extract below provides an example of a possible application of these two parameterized classes:

Classes A and B are any required concrete classes. FIFOB is a concrete class for a “first-in-first-out” list of object references for objects of class B (and its subclasses), FIFOA1 and FIFOA2 are concrete classes for “first-in-first-out” lists of object references of class A (and its subclasses).

```
        IDENTIFICATION DIVISION.
>>>>    PROGRAM-ID.               N.
        ...
        REPOSITORY.
             CLASS              A,
             CLASS              B,
             CLASS              FIFOA1 EXPANDS FIFO-LIST USING A ELEMA   a)
             CLASS              FIFOA2 EXPANDS FIFO-LIST USING A ELEMA   a)
             CLASS              FIFOB EXPANDS FIFO-LIST USING B ELEMB    b)
             CLASS              ELEMA EXPANDS LIST-ELEMENT1 USING A      c)
             CLASS              ELEMB EXPANDS LIST-ELEMENT1 USING B   b) c)
             CLASS              LIST-ELEMENT1
             CLASS              FIFO-LIST
        ...
        WORKING-STORAGE SECTION.
        01 FLA1       USAGE OBJECT REFERENCE FIFOA1.
        01 FLA2       USAGE OBJECT REFERENCE FIFOA2.
        01 FLA3       USAGE OBJECT REFERENCE FIFOA2.
        01 FLB        USAGE OBJECT REFERENCE FIFOB.
        01 OB         USAGE OBJECT REFERENCE B.
        01 W          PIC X(10).
        ...
            INVOKE FIFOA1 "NEW" RETURNING FLA1.
            INVOKE FIFOA2 "NEW" RETURNING FLA2.                       d)
            INVOKE FIFOA2 "NEW" RETURNING FLA3.                       d)
            INVOKE FIFOB  "NEW" RETURNING FLAB.
        ...
            INVOKE FLB "APPEND-ENTRY" USING OB. ...
            INVOKE FLB "REMOVE-ENTRY" RETURNING OB.
            IF OB = NULL                                              e)
              DISPLAY " ---> LISTE LEER"
            ELSE
              ...
```

Notes

a) Although the parameterized class and the current parameters are identical in FIFOA1 and FIFOA2, two different concrete classes are produced because of the different names in the two expansions.

b) It is permissible to use the expansion of a parameterized class (here ELEMB) as the current parameter for another expansion (here FIFOB) provided this does not result in any cyclical dependencies. However, it is not permissible to use a parameterized class itself, e.g. LIST-ELEMENT1, as the current parameter.
The compiler does not necessarily perform subsequent expansions in the order in which they are written in the program, but in such a way that data required for current parameters is available before the expansion takes place.

c) Owing to the restrictions of the COBOL standard mentioned above, the expansion of the two auxiliary classes ELEMA and ELEMB must be specified by the user. Use of these classes, the generation of objects from them, etc. should be reserved solely for the expansions of FIFO-LIST lists!

d) Several concrete objects can be generated from the expansion of a parameterized class. The object references FLA2 and FLA3 make 2 FIFO lists for objects of class A available to the program here. This procedure provides the same function as that described in note 1) - using 2 different classes which offer the same performance. However, it is simpler and should thus always be preferred.

e) If the list is empty, the REMOVE-ENTRY method returns NULL as the object reference.

12.7.3 Files in objects

You can define a file in an object by describing it in a FILE-CONTROL paragraph and a FILE SECTION within a FACTORY or OBJECT area and then specifying appropriate statements to process it, such as OPEN, CLOSE, READ, WRITE, in one or more methods.

If an application has created multiple object instances of a class in which a file is defined then this application is itself responsible for ensuring that no conflicts occur when the file is accessed. This can be achieved as follows:

1. The file has the attribute EXTERNAL. This means that there is only a single copy of the file and that all objects work with the same file. Thanks to the EXTERNAL attribute, other programs can also simultaneously access the file.
2. The file's ASSIGN clause uses the variant "ASSIGN to data-name". If necessary each object can select a separate copy of such a file on OPEN at runtime by supplying an object-specific value for data-name. In this way it is possible to avoid object conflicts with files in the other object instances of the same class.
3. If variant 2 is not used then different object instances work with the same object. The application coordinates the file accesses in such a way that, for example, only one object (out of multiple objects of the same class) can open and process the file at any given time. Another possibility is to employ simultaneous file processing (see the "COBOL2000 User Guide" [1]). In this case it is generally advisable to generate only a single object of the class.

Inheritance applies not only to data in the WORKING-STORAGE SECTION, but also to data defined in OBJECT or FACTORY. It should therefore be noted that the inheritance of a class which contains a file definition can also lead to the above-mentioned conflicts, even if each class instantiates only a single object.

Unlike files in objects as described above, files *defined in methods* behave in the same way as file definitions in normal programs: no further copies of the file definition are created by the instantiation of new objects or through class derivation (inheritance).

Example 12-23

The following class permits access to sequential files consisting of records of 80 characters each.

```
>>>>   CLASS-ID.                 cseqf80 INHERITS BASE.
       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.
       REPOSITORY.
           CLASS                 BASE.
       OBJECT.
       ENVIRONMENT DIVISION.
       INPUT-OUTPUT SECTION.
       FILE-CONTROL.
           SELECT seq ASSIGN TO link-name
                      FILE STATUS file-status.
       DATA DIVISION.
       FILE SECTION.
       FD seq                    RECORD CONTAINS 80.
       01 recs                   PIC X(80).
       WORKING-STORAGE SECTION.
       01 file-status PIC XX.
       01 link-namePIC X(8).
       PROCEDURE DIVISION.
```

```
+++    METHOD-ID.              F-OPEN.
       DATA DIVISION.
       LINKAGE SECTION.
       01 open-mode            PIC X.
         88 open-input         VALUE "I".
         88 open-output        VALUE "O".
       01 file-link          PIC X(8).
       PROCEDURE DIVISION USING open-mode, file-link.
       DECLARATIVES.
       first SECTION.
           USE AFTER ERROR PROCEDURE ON seq.
       open-error.
           IF file-status = ....
       END DECLARATIVES.
       1 SECTION.
       A.
           MOVE file-link TO link-name
           IF open-output
             OPEN OUTPUT seq
           ELSE
             OPEN INPUT seq
           END-IF
           EXIT METHOD.
       END METHOD             F-OPEN.

+++    METHOD-ID.             F-CLOSE.
       ....                   *> poss. DECLARATIVES
       1 SECTION.
       A.
           CLOSE seq.
           EXIT METHOD.
       END METHOD             F-CLOSE.

+++    METHOD-ID.             F-READ.
       DATA DIVISION.
       LINKAGE SECTION.
       01 rec                  PIC X(80).
       01 eof-ind              PIC X.
         88 eof                VALUE "Y" WHEN FALSE "N".
       PROCEDURE DIVISION USING rec RETURNING eof-ind.
                              ....              *> poss. DECLARATIVES
       1 SECTION.
       A.
         SET eof TO FALSE
           READ seq INTO rec
             AT END SET eof TO TRUE
           END-READ
           EXIT METHOD.
       END METHOD          F-READ.

+++    METHOD-ID.             F-WRITE.
       DATA DIVISION.
       LINKAGE SECTION.
       01 rec                  PIC X(80).
       PROCEDURE DIVISION USING rec.
            ....             *> poss. DECLARATIVES
       1 SECTION.
       A.
           WRITE recs FROM rec.
```

```
           EXIT METHOD.
       END METHOD              F-WRITE.
       END OBJECT.
       END CLASS               cseqf80.
```

This class can be used, for example, to process such a file:

```
     ....
01 obj1 USAGE OBJECT REFERENCE cseqf80.
01 obj2 USAGE OBJECT REFERENCE cseqf80.
01 xrec PIC X(80).
01 xind PIX X VALUE "N".
  88 xeof VALUE "Y".
     ....
INVOKE cseqf80 "NEW" RETURNING obj1
INVOKE cseqf80 "NEW" RETURNING obj2
INVOKE obj1 "F-OPEN" USING "I", "EIN-DAT"
INVOKE obj2 "F-OPEN" USING "O", "AUS-DAT"
INVOKE obj1 "F-READ" USING xrec RETURNING xind
PERFORM UNTIL xeof
  ....              *> Process record
INVOKE obj2 "F-WRITE" USING xrec
INVOKE obj1 "F-READ" USING xrec RETURNING xind
END-PERFORM
INVOKE obj1 "F-CLOSE"
INVOKE obj2 "F-CLOSE"
  ....
```

12.7.4 Conformance

Conformance is bound to specific rules, which are explained in more detail below.

The term “conformance” is used in two contexts:

1. Conformance between interfaces that are either defined explicitly with the Interface language element or given implicitly via the interface of the factory or object section of a class definition.
   As an extension to the above context, it is also possible to speak of a “conformance of classes” if their factory and object interfaces conform to one another.
2. Conformance between the current parameters and the formal parameters of a call to a subprogram or method.

12.7.4.1 Conformance between interfaces

An interface interface-1 conforms to an interface interface-2 if and only if:

1. For every method in interface-2 there is a method in interface-1 with the same name taking the same number of parameters, with consistent BY REFERENCE, BY VALUE and OPTIONAL specifications.
2. If the formal parameter of a given method in interface-2 is an object reference, the corresponding parameter in interface-1 must be an object reference following these rules:
   a. If the parameter in interface-2 is a universal object reference, the corresponding parameter in interface-1 must also be a universal object reference.
   b. If the parameter in interface-2 is described with an interface-name, the corresponding parameter in interface-1 must have the same interface-name.
   c. If the parameter in interface-2 is described with a class-name, the corresponding parameter in interface-1 must have the same class-name, and the presence or absence of the FACTORY and ONLY phrases must be the same in both interfaces.
   d. If the parameter in interface-2 is described with ACTIVE-CLASS, the corresponding parameter in interface-1 must also be described with ACTIVE-CLASS. The FACTORY phrase must be either present or not present at both interfaces.
3. If the formal parameter of a given method in interface-2 is not an object reference, the corresponding formal parameter in interface-1 must have the same ANY LENGTH, PICTURE, USAGE, SIGN, JUSTIFIED, BLANK WHEN ZERO and SYNCHRONIZED clauses. Furthermore, the following additional condition applies:
   If a decimal point appears in the Picture clause, then the same DECIMAL-POINT IS COMMA clause must be in effect for both interface-1 and interface-2.
4. The presence or absence of the RETURNING phrase in the Procedure Division must be the same in the corresponding methods.
5. If the returning item in a given method of interface-2 is an object reference, the corresponding returning item in interface-1 must also be an object reference following these rules:
   a. If the returning item in interface-2 is a universal object reference, the corresponding returning item in interface-1 must also be an object reference.
   b. If the returning item in interface-2 is described with an interface-name that identifies the interface int-r, the corresponding returning item in interface-1 must be either of the following:
      - an object reference for an interface that conforms to int-r
      - an object reference for a class, in accordance with the following rules:
        1. If a FACTORY phrase exists, the factory interface of the specified class must be in conformance with int-r.
        2. If no FACTORY phrase exists, the object interface of the specified class must be in conformance with int-r.
   c. If the returning item in interface-2 is described with a class-name, the corresponding returning item in interface-1 must be an object reference, subject to the following rules:
      - If the returning item in interface-2 is described with the ONLY phrase, then the returning item in interface-1 must also be described with the ONLY phrase and the same class-name.
      - If the returning item in interface-2 is described without the ONLY phrase, then the returning item in interface-1 must also be described with the same class-name or a subclass of that class-name.
      - The presence or absence of the FACTORY phrase must be the same.

   d. If the returning item in interface-2 is described with the ACTIVE-CLASS phrase, the corresponding returning item in interface-1 must also be described with the ACTIVE-CLASS phrase, and the presence or absence of the FACTORY phrase must be the same.

   If the description of the returning item of a method in interface-1 directly or indirectly references interface-2, the description of the returning item of the corresponding method in interface-2 must not directly or indirectly reference interface-1.

6. If the returning item in a given method of interface-2 is not an object reference, the corresponding returning item must have the same ANYLENGTH, PICTURE, USAGE, SIGN, JUSTIFIED, BLANK WHEN ZERO and SYNCHRONIZED clauses. Furthermore, the following additional condition applies:
   If a decimal point appears in the Picture clause, then the same DECIMAL-POINT IS COMMA clause must be in effect for both interface-1 and interface-2.

Example 12-24

```
Interface i1
    Method X    no parameters
    Method Y    using a with 01 a Pic 999 usage display.

Interface i2
    Method Y    using b with 01 b Pic 9(3).

Interface i3
    Method Y    using c with 01 c Pic 9(4)
    Method Z    returning d with 01 d usage object reference i1

Interface i4
    Method Z    returning e with 01 e usage object reference i2
```

Notes

1. Interface i1 conforms with Interface i2, since method Y from i2 also exists in i1 and its parameter definitions are the same (9(3) is equivalent to 999 and the usage display is standard).
2. Interface i2 does not conform with Interface i1, since method X does not exist in i2.
3. Interface i3 does not conform with Interface i2 although methods from i2 are also present in i3 because the parameters are defined differently here (9(4) instead of 9(3)).
4. Interface i3 conforms with i4, since method Z exists in both, and the returning items, though not identical, still match: Interface i1 (from the method definition of Z in i3) conforms with i2 (from the method definition of Z in Z in i4).

12.7.4.2 Conformance for parameters and returning items

Parameters

The number of arguments in the calling source unit must be equal to the number of formal parameters in the called unit, with the exception of trailing formal parameters that are specified with an OPTIONAL phrase in the Procedure Division header of the called unit and omitted from the list of arguments of the calling unit.
If either the formal parameter or the associated argument is a strongly typed group item, the formal parameter and the argument must be of the same type (see the section "Types").
A national group is always treated as an elementary data item.

Group items

If either the formal parameter or the argument is a group item, then that argument or the formal parameter corresponding to it must be a group item or an alphanumeric elementary item. In this case, the formal parameter may be shorter than the current parameter.

Elementary items

The conformance rules for elementary items depend on whether the argument is passed BY REFERENCE, BY CONTENT or BY VALUE (phrases in the CALL or INVOKE statement).

Elementary items passed BY REFERENCE

If either the formal parameter or the corresponding argument is an object reference, the corresponding argument or formal parameter must be an object reference following these rules:

1. If either the argument or the formal parameter is a universal object reference, the corresponding formal parameter or argument must also be a universal object reference.
2. If either the argument or the formal parameter is described with an interface-name, the corresponding formal parameter or argument must have the same interface-name.
3. If the formal parameter is specified with a class name, the corresponding argument must be specified with the same class name. In addition, the FACTORY and ONLY phrases must be the same.
4. If the formal parameter is specified with ACTIVE-CLASS, one of the following conditions must apply:
    a. The argument must be specified with ACTIVE-CLASS, and the method must be called with one of the predefined object references SELF or SUPER or with an object reference specified with ACTIVE-CLASS. In addition, the FACTORY phrase must be the same for the formal parameter and the argument.
    b. The argument must be specified with a class name and the ONLY phrase, and the method must be called either with this class name or with an object reference specified with this class name and the ONLY phrase. In addition, the FACTORY phrase must be the same for the for the formal parameter and the argument.

If neither the formal parameter nor the argument is of class object, the following rules apply:

1. If the called source unit is a program for which there is no entry of a program prototype name (i.e. a program-specifier) in the REPOSITORY paragraph of the calling unit and there is no NESTED phrase specified on the CALL statement, the formal parameter must have the same number of character positions as the corresponding argument.
2. If the called unit is one of the following:
    - a program for which there is a program prototype name in the REPOSITORY paragraph of the calling unit,
    - a program, and the NESTED phrase is specified on the CALL statement,
    - a method,

then the definition of the formal parameter and that of the argument must have the same PICTURE, USAGE, SIGN, JUSTIFIED, and BLANK WHEN ZERO clauses, with the following additional conditions:

- If a decimal point appears in the Picture clause, then the same DECIMAL-POINT IS COMMA clause must be in effect for both the calling and the called source unit
- If the formal parameter has the SYNCHRONIZED phrase, the current parameter (argument) must also have the same phrase or be defined on level 01.
- If the formal parameter is defined with the ANY LENGTH clause, then its length is taken over as the length of the corresponding argument.
- If the argument is defined with the ANY LENGTH clause then the associated formal parameter must also be defined with the ANY LENGTH clause.
- If the argument or the formal parameter is a type-specific pointer, both must be typespecific pointers and of the same type.

Elementary items passed BY CONTENT or BY VALUE

If the formal parameter is an object reference specified wtih ACTIVE-CLASS, one of the following conditions must apply:

1. The method must be called with one of the predefined object references SELF or SUPER or with an object reference specified with ACTIVE-CLASS. A SET statement with, as the sending operand, the argument and, as the receiving operand, an object reference specified with ACTIVE-CLASS with the same FACTORY phrase as the formal parameter must be valid in the calling runtime unit.
2. The method must be called with a class name or with an object reference that is specified with a class name and the ONLY phrase. A SET statement with, as the sending operand, the argument and, as the receiving operand, an object reference specified with this class name and the ONLY phrase with the same FACTORY phrase as the formal parameter must be valid in the calling runtime unit.

If the formal parameter is an object reference that is not specified with ACTIVE-CLASS, the conformance rules are the same as when a SET statement is executed in the calling runtime unit with the argument as the sending operand and the corresponding formal parameter as the receiving operand.

If the formal parameter is not of the “object” class, the following conformance rules apply:

1. If the called source unit is a program for which there is no entry of a program prototype name (i.e. a program-specifier) in the REPOSITORY paragraph of the calling unit and there is no NESTED phrase specified on the CALL statement, the formal parameter must have the same number of character positions as the corresponding argument.
2. If the called unit is one of the following:
   - a program for which there is a program prototype name in the REPOSITORY paragraph of the calling unit,
   - a program, and the NESTED phrase is specified on the CALL statement,
   - a method,

   then the following rules apply, depending on the type of the formal parameter:
   a. If the formal parameter is numeric, the conformance rules are the same as for a COMPUTE statement with the argument as the sending operand and the corresponding formal parameter as the receiving operand.
   b. If the formal parameter is an index elementary item or an elementary item of the class “pointer”, a SET statement with the argument as the sending operand and the corresponding formal parameter as the receiving operand must be permissible.

c. Otherwise, a MOVE statement with the argument as the sending operand and the corresponding formal parameter as the receiving operand is permissible.

d. If the formal parameter is defined with the ANY LENGTH clause, then:

- The associated argument must be of the class “alphabetic”, “alphanumeric” or national and be defined with the same USAGE DISPLAY or USAGE NATIONAL.
- The associated argument must **not** be defined with the ANY LENGTH clause.
- The length of the formal parameter corresponds to the length of the argument.

Returning items

A returning item must be specified in a calling statement if and only if a returning item is defined in the Procedure Division header of the called source unit.

If one of the operands is a strongly typed group item, both must be of the same type.

A national group is always treated as an elementary item.

Group items

If one of the operands is a group item, then the corresponding item must also be a group item or an alphanumeric elementary item and of the same length.

Elementary items

If either of the operands is an object reference, the corresponding item must also be an object reference, and the following rules apply:

1. If no ACTIVE-CLASS phrase is specified in the returning item of the called source unit, the conformance rules are the same as if a SET statement were performed in the called runtime unit with the returning item in the called source unit as the sending operand and the corresponding returning item in the calling source unit as the receiving operand.
2. If an ACTIVE-CLASS phrase exists in the returning item of the called source unit, the conformance rules are the same as if a SET statement were performed in the calling runtime unit with the returning item in the called source unit as the receiving operand, and a sending operand described with USAGE OBJECT REFERENCE as determined by the following rules:
   a. If a method is invoked with a class-name, the sending operand must be described with the same class-name and an ONLY phrase.
   b. If a method is invoked with the predefined object references SELF or SUPER, the sending operand must be described with an ACTIVE-CLASS phrase.
   c. If a method is invoked with an object reference described with an interface-name, the sending operand must be a universal object reference.
   d. If a method is invoked with any other object reference, this identifier is used as the sending operand, including the ONLY phrase if specified.
   e. If the sending operand selected by applying the above rules is described with a class-name or an ACTIVE-CLASS phrase, the presence or absence of the FACTORY phrase must be the same as in the returning item of the called source unit.
   f. If the operands are not object references, then rule b) given under ’Elementary items passed BY REFERENCE’ applies.

   If neither the formal nor the current return value is of the class object, then both of the operands must have the same PICTURE, USAGE, SIGN, JUSTIFIED, and BLANK WHEN ZERO clauses, with the following additional conditions:

   - If a decimal point appears in the Picture clause, then the same DECIMAL-POINT IS COMMA clause must be in effect for both the calling and the called source unit
   - If the formal returning item has the SYNCHRONIZED phrase, the current returning item must also have the same phrase or be defined on level 01.
   - If the formal returning item is defined with the ANY LENGTH clause, then its length is taken over as the length of the current returning item.

- If the current returning item is defined with the ANY LENGTH clause then the associated formal returning item must also be defined with the ANYLENGTH clause.
- If one of the operands is a type-specific pointer, both must be type-specific pointers and of the same type.

12.7.5 The system class BASE

BASE serves as a basis for creating objects. If this class is not used, it is only possible to work with factory objects.

The interface definition BaseFactoryInterface describes the interface of the factory object of the class BASE. The interface definition BaseInterface describes the interface of objects of the class BASE. The class BASE is defined as if the IMPLEMENTS phrases were made for the interfaces BaseFactoryInterface and BaseInterface in the FACTORY and OBJECT paragraphs.

Both the class BASE and the interface definitions BaseFactoryInterface and BaseInterface are made available by the compiler or runtime system.

The BASE class includes two methods: ’NEW’ and ’FACTORYOBJECT’.

12.7.5.1 New method

The New method is a factory method that provides a standard mechanism for creating object instances of a class. Described with COBOL semantics, the interface for the New method is as follows:

```
Interface-id. BaseFactoryInterface.
Procedure division.
   Method-id. New.
   Data division.
   Linkage section.
   01 outObject usage object reference active-class.
   Procedure division returning outObject.
   End-method New.
End Interface BaseFactoryInterface.
```

General rules

1. The New method allocates storage for an object and initializes its data. An object is in the initial state immediately after it is created.
2. If there is not enough storage space to create a new object, the exception condition EC-OO-RESOURCE occurs. The procedure continues in accordance with rule 6 f) of the INVOKE statement (see the section "INVOKE statement").

Example 12-25

Override New method

The New method can be overridden in order to perform additional object-specific initialization operations which go beyond the initial values specified in the VALUE clause. In this case, the new object must first be generated using the “original” New method.

```
FACTORY.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 ident pic 9(9)   COMP VALUE 0.
...
METHOD-ID.          NEW OVERRIDE.
DATA DIVISION.
LINKAGE SECTION.
01 new-object     USAGE OBJECT REFERENCE ACTIVE-CLASS.
PROCEDURE DIVISION RETURNING new-object.
1.
   INVOKE SUPER "NEW" RETURNING new-object.
   COMPUTE ident = ident + ...
   INVOKE new-object "SPECIAL-INITIALIZATIONS" USING ident...
   ...
2.
   EXIT METHOD.
END METHOD          NEW.
```

12.7.5.2 FactoryObject method

FactoryObject is an object method that provides a standard mechanism for acquiring access to the factory object of a class. Described with COBOL semantics, the interface for the FactoryObject method is as follows:

```
Interface-id. BaseInterface.
Procedure division.
   Method-id. FactoryObject.
   Data division.
   Linkage section.
   01 outFactory usage object reference factory of active-class.
   Procedure division returning outFactory.
   End method FactoryObject.
End Interface BaseInterface.
```

General rules

If the FactoryObject method is called for an object, it determines the class of the object and references the factory object of this class.

Example 12-26

The FactoryObject method is helpful when if the class of an object is not known. If the class is known, you can access the methods of a factory object as follows:

```
INVOKE CLASSNAME "XYFACTORYMETHODNAME"
```

Even if the class of an object is unknown, one of its factory methods can still be called as follows:

```
INVOKE OBJECTA "FACTORYOBJECT" RETURNING FACTORYOBJECTN
INVOKE FACTORYOBJECTN "XYFACTORYMETHODNAME"
```

12.7.6 Automatic release of memory space (garbage collection)

In COBOL2000, programmers do not have to worry about releasing the storage space that is occupied by program objects that are no longer needed. This task is performed for COBOL2000 programs by a “Garbage Collector”. However, the Garbage Collector does not request the space occupied by these objects at the point when they are no longer required by the program. Instead, this mechanism requests the memory space when the total amount of available memory space becomes low. When this occurs, the program may temporarily appear to stop.

12.8 Data types

A type is a pattern containing all the characteristics of the data description unit and its subordinate data items. A type is defined by specifying the TYPEDEF clause. The principal characteristics of a type which is identified by its type name are the relative positions and lengths of its elementary items which are defined in the type declaration, plus the BLANK WHEN ZERO clause, JUSTIFIED clause, PICTURE clause, SIGN clause, SYNCHRO-NIZED clause and USAGE clause which are specified for these elementary data items (for details on the clauses see the section "Data description entry").

The following data types exist:

- weakly typed elementary items
- weakly typed group items
- strongly typed group items

Note: Elementary items cannot be strongly typed.

A type - whether weak or strong - defines a particular data structure which has a precisely specified name which is used precisely with this structure. This prevents the structure being affected, for example, by a different alignment or a change in the data representation through the USAGE phrase on higher-ranking group levels.

12.8.1 Weakly typed data descriptions

Weakly typed data descriptions relate to a type declaration which does not contain the STRONG phrase, and they are not subordinate to such type declarations. Weakly typed data descriptions can be either group items or elementary items.

Their data structure is not always protected in the manner described as weakly typed data descriptions can be used like practically any untyped data description. The type declaration could be regarded as an abbreviation for a number of data descriptions.

The following example illustrates how the TYPEDEF clause is used to define a type, and the TYPE clause to use a type:

Example 12-27

```
01 Employee TYPEDEF.                    *> defines type name Employee
   02 name PIC X(30).                   *> with this description
   02 personnel-number PIC 9(8).

      01 Department.
   02 Dept-Name PIC X(16).
   02 Dept-Head TYPE Employee.          *> uses type name Employee

   02 Dept-Deputy TYPE Employee.
   02 Dept-Office TYPE Employee.
```

This results in a record with the description:

```
01 Department.
   02 Dept-Name PIC X(16).
   02 Dept-Head.
      03 name PIC X(30).
      03 personnel-number PIC 9(8).
   02 Dept-Deputy.
      03 name PIC X(30).
      03 personnel-number PIC 9(8).
   02 Dept-Offic.
      03 name PIC X(30).
      03 personnel-number PIC 9(8).
```

The expansion of a type can create hierarchies with a depth of more than 49 levels which can otherwise at most be directly written in a data description.

12.8.2 Strongly typed data descriptions

Strongly typed data descriptions are defined with a type declaration which contains the STRONG phrase or are subordinate to a type declaration with the STRONG phrase. Only group items can be strongly typed as the concept of strong type definition for elementary items would entail a large number of serious restrictions.

In addition to the protection of the data structure which is defined by the type, strong type definition is designed above all to protect the integrity of the data content. The correct use of strongly typed data descriptions should therefore never lead to data which is incompatible with its data description.

An important consequence is therefore to forbid any implicit or explicit redefinitions.

A group item can be regarded as a new definition of the elementary items which are subordinate to it. Consequently accesses to strongly typed group items are restricted to those which do not affect the integrity of the data which is subordinate to the group item in question. If a strongly typed group item is a receiving item, the sending item must be of the “same type”. The “same type” is defined as a type description with the same name and the same relevant characteristics.

There is no need to require the same restrictions for elementary items since accesses to elementary items do not generally corrupt the content of the receiving item.

Further restrictions exist for strongly typed group items and elementary items which are subordinate to strongly typed group items. To guarantee data integrity they cannot be renamed (RENAMES clause). Reference modification is only permitted for alphanumeric or national data items.

Restrictions for the addresses of strongly typed group items and pointers containing such addresses are described in the section "Type-specific pointers".

You should be aware that protecting the integrity of data is only meaningful if the data is above all “correct”.

12.9 Addresses and pointers

Addresses and pointers reference storage locations in the computer and addresses which make it possible to identify these storage locations. Usually, little use is made of these possibilities in applications.

In order to ensure the universal applicability of programming languages and in the light of the need to interact with systems such as POSIX or programming languages such as C and C++, COBOL also provides address and pointer handling capabilities. However, address and pointer handling should be used carefully since such techniques may impair the clarity and ease of maintenance of programs.

12.9.1 Data addresses and data pointers

The term **data address** refers to the storage location of an elementary data item. It is addressed via a data address specified. A data address specifier may not be used as a receiving item. Here you should note that the ADDRESS OF phrase for the receiving item in the SET statement does not represent a data address identifier but is instead part of the SET syntax.

A **data pointer** (also known as a pointer) is an elementary data item in which a data address can be stored.

The data address of an elementary data item defined with a BASED clause can be set either by means of a data address identifier or using a data pointer. Data addresses and data pointers can be passed to other source units and returned by them.

12.9.2 Using data pointers

Pointers can be used as follows:

1. as formal parameters in the PROCEDURE DIVISION USING phrase
2. as current parameters in the CALL or INVOKE statement
3. in the SET statement: assignment and updating of addresses (UP/DOWN)
4. in relational conditions: comparisons of equality or inequality with the predefined NULL address, with a data address identifier and other pointers
5. in the LENGTH function: this function returns the value 4 for FUNCTION LENGTH (LENGTH OF ...) and FUNCTION LENGTH (pointer).
   FUNCTION LENGTH (ADDRESS OF ...) is not currently supported.

Example 12-28

Let us assume that the subprogram GNR returns a pointer to a record in accordance with the following program prototype:

```
Program-id. Get-Next-Record is prototype. *> returns the address of a
                                             record
Data Division.
Linkage Section.
01 ptrl usage pointer.
Procedure Division returning ptrl.
End-program Get-Next-Record.
```

and that a user program contains the following declarations

```
program get-next-record  *> in the Repository Paragraph

01 p usage pointer.  *> in the Working-Storage Section

01 my-wreck based.   *> in the Linkage Section
   02 name pic x(30).
   02 addr pic x(30).
```

The following statement in the Procedure Division calls the program described by the prototype:

```
Call "GNR" as Get-Next-Record returning p
```

The data can be accessed with my-wreck since the pointer p contains the address of a record:

```
Set address of my-wreck to p
Move "SAM JONES" to name in my-wreck
```

Example 12-29

```
01 p2 usage pointer.
01 data-record.     *> the full record layout is described
   02 ...
```

If the Program Process Record “PR” is to be passed to a pointer then coding can be as follows:

```
Set p2 to address of data-record.
Call "PR" using p2.
```

Alternatively, the address of `data-record` could be passed as follows *:

```
Call "PR" using address of data-record
```

* in this case, however, any change to the passed pointer by the called program remains ineffective for the calling program.

12.9.3 Program addresses and program pointers

The term program address describes the address of an entry point of the program. It is addressed by means of a program address identifier. A program address identifier must not be used as a receiving operand.

A program pointer is an elementary item in which a program address can be stored.

A program pointer can be used to call a program. Program address identifiers and program pointers can be passed to other source units, and program pointers can also be returned from other source units.

12.9.4 Using program pointers

Program pointers can be used as follows:

1. as the program to be called in the CALL statement
2. as formal parameters in the PROCEDURE DIVISION USING phrase
3. as current parameters in the CALL or INVOKE statement
4. in the SET statement: to assign program pointers or program address identifiers
5. in the INITIALIZE statement to occupy program pointer data items in advance
6. in relation conditions: to carry out comparisons with the predefined address NULL, with a program address identifier and with other program pointers and ascertain whether they are the same or different
7. in the LENGTH function: this function supplies the value 4 for FUNCTION LENGTH (program pointer). FUNCTION LENGTH (ADDRESS OF PROGRAM ...) is currently not supported.

Example 12-30

Calling a program using a program pointer:

```
01 pgmptr USAGE PROGRAM-POINTER.
SET pgmptr TO ADDRESS OF PROGRAM "UPROGP".
CALL pgmptr.
```

Example 12-31

Calling a subprogram using a program pointer as a parameter:

```
01 pgmptr USAGE PROGRAM-POINTER.
SET pgmptr TO ADDRESS OF PROGRAM "UPROGP".
CALL "UNDER" USING pgmptr.
```

12.9.5 Type-specific pointers

A type-specific data pointer may only contain the address of a data description of the same type. It may only be defined within a type description. The use of type-specific data pointers is important in supporting the security of types. It prevents data of one type being handled like data of another type.

This integrity is especially important for strongly typed data items. The address of a strongly typed data item is regarded as a type-specific pointer. Consequently it can only be assigned to a pointer which refers to the same type.

Furthermore, a type-specific pointer can only be used to address an elementary item which is specified with the BASED clause with the same type. By the same token, the address of a strongly types elementary item which is specified with the BASED clause can only be set to the address of an elementary item of the same type. As a result, existing restrictions which force the integrity of strongly typed data items cannot be bypassed by using addresses, pointers and data items with the BASED clause.

12.10 Language elements for processing XML

COBOL2000 offers language elements for reading XML documents. Such documents can be made available in files or in memory. A separate open source program package, the **parser**, analyzes and parses the XML document. This program package is not supplied with the COBOL compiler or CRTE, but can be downloaded from the Internet. You must make the parser available before COBOL programs which use XML language elements are executed, and ensure that it conforms with them.

The language extensions of COBOL2000 enable an XML document to be processed in two different ways:

- structure-oriented, based on a tree presentation of the document
- event-oriented, based on a purely sequential view of the document

	Structure-oriented	**Event-oriented**
Language elements	As in the Technical Report "Native COBOL Syntax for XML Support"	Based on the language extension of other compiler providers
Document	In memory or file	Only in memory
Access to parts of the XML document	Any Possible to repeat access to same parts.	Sequential Each part can only be acces sed once.
Data supplied	Actual data of the document, normalized	Almost everything in the document
Additional processing effort of the parser	The document must be transferred to the tree presentation.	The document can be processed directly.
Additional memory requirement of the parser	High (for the entire tree)	Low (for administrative data)
Parser interface	DOM (Document Object Model)	SAX (Simple API for XML)

Table 46: Comparison of structure- and event-oriented processing

To permit the new language elements to be used, the relevant option must be set when the program is compiled.

12.10.1 Structure-oriented processing

When processing starts, the parser transforms the entire XML document into a tree presentation in working memory. In the COBOL program the familiar data structures correspond to the hierarchical structures of the XML document. The structure of the XML document can thus be described with the aid of two additional new clauses. However, you do not need to describe the entire document; it is sufficient if those parts which the program wishes to process are described. The enhanced statements for file processing then permit such COBOL data structures to be assigned to specific positions in the tree, data selected from the tree in this way to be transferred, and access to this data in the COBOL program.

12.10.1.1 XML document as a tree

To understand structure-oriented processing it is helpful to perceive the hierarchical arrangement of elements and attributes in an XML document as a tree structure. In much simplified form, such a tree can be defined as follows:

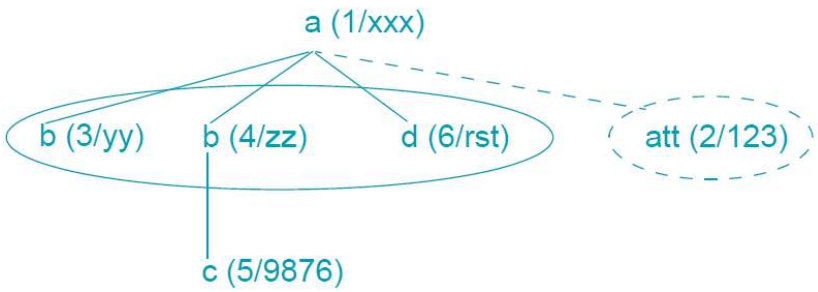
- Each element or attribute from the XML document corresponds to a **node** in the tree, which is assigned the name and the value of the element or attribute. Element names do not have to be unique here; attribute names need only be unique with respect to their element names.
- Each **element node** has two **ordered** sequences of nodes which are **directly subordinate** to it, its **children**: a sequence of element nodes and a sequence of attribute nodes. Each of these sequences can be empty, or contain one or more nodes.
- The first node in one of the ordered sequences is the **oldest**, and the last the **youngest child**, and consequently all the nodes which follow a node in such a sequence are referred to as **younger siblings**, all those in front of it as **older siblings**.
- The outermost element of the XML document, which encompasses everything, is the **root** of the entire tree. In the same way, every other element node is at the same time also the root of a **subtree**. A node defines a subtree; the term subtree also applies for a single node which has no children.
- With the exception of the root of the entire tree, each node in the tree has precisely one directly superordinate node, its **parent**.
- Nodes which have no children are also called **leaves**.
- Each element or attribute in the tree has a unique **position**, which is used to establish the connection between the description in the COBOL program and the elements or attributes currently assigned.

Example 12-32 XML document as a tree

XML document:

```
<a
   att="123"
>xxxx
   <b>yy</b>
   <b>zz
      <c>9876<
/c>
  </b>
  <d>rst</d>
</a>
```

In tree presentation:



The position and the value are noted in parentheses for each node. Element nodes are represented by solid lines, attribute nodes by broken lines.

Comments:

- 'a' is the root of the entire tree.
- Its children are the nodes at positions 2, 3, 4 and 6.
- The sequence of element nodes consists of the two 'b's' and 'd'.
- The sequence of attribute nodes consists of a single node, 'att'.
- 'b' at position 4 has 'd' as its younger sibling and 'b' at position 3 as its older sibling.

? Unbekannter Anhang

? Unbekannter Anhang

- ’a’ is the parent of the two 'b's’, of 'd' and of the 'att' node
- Node 'a' has both element and attribute nodes as children. The node at position 4 has only one element node as a child, but no attribute nodes. The nodes at positions 2, 3, 5 and 6 are leaves; they have no children, in other words neither subordinate element nor subordinate attribute nodes.

12.10.1.2 COBOL language elements for describing an XML document

The level concept for structuring records which already exists in COBOL is used to represent the hierarchical structure of an XML document.

For the sake of clarity, the new COBOL language elements will be presented in the first step without taking namespaces into account. Please refer to section "Namespace" for information on the special aspects of describing and processing namespaces.

A data item in the COBOL data structure corresponds to a node from the tree. the COBOL structure can also contain further data items to which no node in the tree corresponds.

- The data item corresponding to a root has the level number 01.
- The data items for **children of a node** all have the **same level number**, which is greater than that of their parent.
- In the COBOL structure the new IDENTIFIED clause identifies the data items to which a node in the tree corresponds and specifies the **type** of node: element or attribute.
- the IDENTIFIED clause also serves to specify the **name** of an element or attribute as contained in the tags in the XML document. This name is case-sensitive.
- In the structure the data item which is to contain the **value** of an element or attribute is **directly subordinate** to the data item with the IDENTIFIED clause for the element or attribute. If this data item is the only data item which is directly subordinate to an IDENTIFIED clause, it can be omitted, and the PICTURE clause for the value can be specified directly together with the IDENTIFIED clause.
- If an IDENTIFIED clause is specified for a data item, all superordinate group items in the structure must also have an IDENTIFIED clause. this means that the hierarchical structure of the tree must be reflected without a gap with the COBOL structure. Additional intermediate levels on the COBOL side – even if they are only intended to enhance the structuring – are not permitted.

Example 12-33 COBOL description of the entire XML document

XML document

```
<a
   att="123">
   xxxx
   <B>zz

      <c>9876</c>

   </B>
</a>
```

COBOL data structure

```
01 root                IDENTIFIED BY "a" ELEMENT.
   02 root-att         PIC 999.
      03 att-value     IX X(10).
   02 root-value       IDENTIFIED BY "B".
   02 child            PIC X.
      03 child-value   IDENTIFIED BY "c" ELEMENT
      03 grandchild    PIC 9(8) BINARY.
```

Comments:

- The data item grandchild has no further substructure and can therefore be used at the same time as the specification of the element name (’c’) to accommodate the value.
- The values from the XML document are made available in accordance with the COBOL description:
  - numeric values aligned on the decimal point, if necessary converted (e.g. data item grandchild)
  - alphanumeric values, truncated if necessary (e.g. data item child-value), or filled with blanks (e.g. data item root-value)
- It is not necessary that a data item should also be defined in the COBOL data structure for the value of a node, e. g. if the node never has a value in the tree or the program does not wish to process the value.
- The end tags from the XML document do not appear in the COBOL representation. They are implied in the hierarchical structure.
- ELEMENT is the default value for the node type and can also be omitted in the IDENTIFIED clause (e.g. for the group item child).
- The COBOL description also contains data items which do not correspond to any node in the tree (e.g. att-value, root-value and child-value).
- Do not confuse the literal specified in the IDENTIFIED clause with the initial value of the data item. The latter is only specified by a VALUE clause.

Specifying an element or attribute name in the IDENTIFIED clause

Several options are available for specifying the name of an element or attribute in the IDENTIFIED clause; see the list below.

For the sake of clarity, the special features of namespaces which are also a component part of the name are described summarily in section "Namespace".

- You have **predefined** the name by means of IDENTIFIED **BY**
  - constantly, as a literal
  - variably, as the name of a data item which contains the current element or attribute name
- The name is **not predefined**; instead, the name found in the document is returned by means of IDENTIFIED **USING**.
  This can also be regarded as a particular form of 'predefined' in which not exactly one name is predefined, but all possible names, in other words a type of wildcard notation.

- The data items specified for the element and attribute names in an IDENTIFIED clause may be defined 'almost everywhere' in the program. However, if their data description entry is contained in a data structure for an XML document, this data description entry must be directly subordinate to the IDENTIFIED clause which references it.
- The use of BY or USING in data structures is subject to the following restrictions:
  - If **multiple** data items with an ELEMENT phrase in the IDENTIFIED clause are directly subordinate to a data item, these must all specify the BY phrase.
  - If **multiple** data items with an ATTRIBUTE phrase in the IDENTIFIED clause are directly subordinate to a data item, these must all specify the BY phrase.
  - If only **one** data item with an ELEMENT phrase in the IDENTIFIED clause is directly subordinate to a data item, it may specify either the BY or the USING phrase.
  - If only **one** data item with an ATTRIBUTE phrase in the IDENTIFIED clause is directly subordinate to a data item, it may specify either the BY or the USING phrase.

Example 12-34 Same description for different documents

XML document 1

```
<a
   att="123">
   xxxx
   <b>zz
      <c>9876</c>
   </b>
</a>
```

XML document 2

```
<a_2
  att="345">
   abc
   <b>@#!??*
      <d>100</d>
   </b>
</a_2>
```

COBOL data structure

```
01 root                       IDENTIFIED BY root-name.
   02 root-name               PIC XXXX VALUE "a".
   02 root-att                IDENTIFIED BY "att" ATTRIBUTE.
      03 att-value            PIC 999.
   02 root-value              PIC X(10).
   02 child                   IDENTIFIED BY "b".
      03 child-value          PIC X.
      03 grandchild           IDENTIFIED USING grandchild-name.
         04 grandchild-value  PIC 9(8) BINARY.
         04 grandchild-name   PIC X.
```

Comments:

- The indirection in the IDENTIFIED clause for the root group item that instead of the literal "a" a data item which has the value "a" should be specified enables the same COBOL data structure to be used to process XML documents with the same structure in which the name of the root element differs. For XML document 2, for example, the elementary item root-name would need to be supplied with the value "a_2".
- The IDENTIFIED USING phrase in the grandchild data item means there is no default for the element name, i.e. any names are suitable (e.g. the ’c’ from XML document 1 or the ’d’ from XML document 2) and are returned in the elementary item grandchild-name.

- The order of the data items for name and value in the structure is arbitrary (e.g. root-value before root-name or grandchild-name before grandchild-value would be possible).

Scope of the COBOL description

> The XML document may and need not be reflected 100% in COBOL

It is sufficient if the COBOL program describes only those parts of the XML document which it also wishes to process. However, one requirement must be satisfied here: the parent of each node from the XML document which is described in the COBOL document must also be described in the COBOL program. the **root** of the XML document must therefore **always** be described in the COBOL program.

The COBOL data structure may describe either more or fewer nodes than are currently present in the XML document.

Nodes from the XML document can be omitted from the COBOL description for the following two reasons:

- **Subordinate nodes** need not be described in the COBOL structure if the application does not wish to process a node – and consequently also all nodes in the subtree whose root the node represents.
- **Repetitions** of nodes with the same name in a sequence of element nodes may not be described in the COBOL structure. this cannot occur with attributes as they must have unique names with respect to their element.

  Analogy to files which do not have organization XML: in their file declaration entry (FD) not every single record from the file has its own description; only the different record structures are described in the FD. Similarly repetitions of elements with the same name in the XML document are irrelevant for the description in COBOL – a single description of the element is sufficient. Like the READ statement, which supplies further records, access always taking place via the same record description entry, the XML-specific statements also supply the repetitions of elements, and access also always takes place only via the one description of the element in the COBOL structure.

Example 12-35 Partial description of an XML document

XML document

```
<a>xxx
   <b>yyy</b>
   <d>123</d>
   <b>zz
      <c>9876</c>
   </b>
 </a>
```

COBOL data structure 1

```
01 a              IDENTIFIED BY "a".
   02 a-w         PIC X(10).
   02 b           IDENTIFIED BY "b".
      03          b-w PIC X(10).
      03 c        IDENTIFIED BY "c".
         04 c-w   PIC X(10).
```

COBOL data structure 2

```
01 a           IDENTIFIED BY "a".
   02 a-w      PIC X(10).
   02 b        IDENTIFIED BY "b".
      03       b-w PIC X(10).
   02 d        IDENTIFIED BY "d".
      03 d-w   PIC X(10).
```

Comments:

- COBOL data structure 1 permits the root of the XML document to be processed, and below this only those subtrees whose root has the name ’b’ – but not the subtree with the root ’d’.
- When processing the XML document with COBOL data structure 1, the first b node in the document has no children, but a child is described for it in the COBOL structure. What this means in detail is explained in the example "Principle of assigning nodes" in section "Statements for XML processing".
- COBOL data structure 2 permits the root of the XML document and its children (’b’ and ’d’) to be processed, but no further nodes which are subordinate to these (’c’).
- When processing the XML document with COBOL data structure 2, the second b node in the document has children, but no child is described in the COBOL structure for these. What this means in detail is explained in the example "Sequential reading" in section "READ".

COUNT clause

If nodes which currently do not currently exist in the tree are described in the COBOL structure, it can be important for further processing to know which data items of the structure are concerned.

The COUNT clause is used for this purpose: it defines an integer numeric element which can contain only the value 0 or 1.

- The value 1 indicates that when the XML document was read it was possible to assign a node from the tree to the description in the COBOL structure, see also "Assignment procedure" in section "Statements for XML processing".
- The value 0 indicates that no corresponding node exists in the tree.

Example 12-36 COUNT phrase

```
...
 08 x-node      IDENTIFIED BY "x" COUNT IN x-number.
   09 x-value   PIC 9(8).
...
```

Comments:

- The name 'x-number' of the elementary item is freely selectable. This name must, however, be unique throughout the entire program **without qualification**. However, you may **not define** the data item with this name **yourself**.
- The definition is implied by specifying the data name in the COUNT clause.
- The COUNT clause may only be specified for data items which also have an IDENTIFIED clause.
- The COUNT elementary item does **not** specify the number of repetitions of nodes with the name 'x' in the tree.
- The COUNT elementary item has nothing to do with the value of a node.

12.10.1.3 Definition of an XML document in a COBOL program

The XML document which is to be processed by a COBOL program can be made available both in a file and in memory (in alphanumeric or national representation) – e.g. in a data item of the LINKAGE-SECTION. In **both cases** it is handled like a file with the new file organization XML. In the following the term XML file therefore always covers these two options. As for files which already existed, entries in the ENVIRONMENT DIVISION and in the DATA DIVISION are consequently also required for an XML document.

ENVIRONMENT DIVISION

The XML document requires a file control entry in the FILE-CONTROL paragraph:

- In addition to the SELECT clause, the ORGANIZATION clause **must** be specified with the (new) organization XML.
- The connection to the medium which contains the XML document is established in the ASSIGN clause.
  - If the XML document is contained in a file, the same ASSIGN clause is used as usual.
  - If the XML document is contained in a memory area, you can specify the corresponding data item directly by means of **DATA data-name-1**.
    Alternatively you can use **data-name-2 LENGTH data-name-3** to specify a data pointer which points to the memory area containing the XML document in the length (in characters) specified by data-name-3.
- The ACCESS MODE clause can be omitted. However, if it is specified, the only permissible access type is **XML**.
- All other clauses are forbidden for an XML document, with the exception of the FILE STATUS clause, whose extended form is permitted. As a result, a return code of the parser is available as a supplement.

> The file control entry does not describe any attributes of a specific file for an XML document.

Consequently if the XML document is contained in a file, the entry does not define a file format. Rather, sequential and index-sequential files, POSIX files and also library members are possible. In the case of the index-sequential files the, ISAM key at the beginning (8 bytes) is removed.

Example 12-37 File control entry for an XML document

```
...
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT xml-doc1 ASSIGN TO "LINK-NAM"
                    ORGANIZATION IS XML
                    ACCESS XML.
    SELECT xml-doc2 ASSIGN TO DATA w-s-doc
                    ORGANIZATION IS XML.
    SELECT xml-doc3 ASSIGN TO doc-ptr LENGTH doc-lg FOR NATIONAL
                    ORGANIZATION IS XML.
...
WORKING-STORAGE SECTION.
01 w-s-doc        VALUE "<a att="123">xxxx<b>zz<c>9876</c></b></a>".
01 doc-ptr        USAGE POINTER.
01 doc-lg         PIC 9(8) BINARY.
```

Comments:

- The XML document processed with xml-doc1 is contained in a file with the link name LINK-NAM.
- The XML document processed with xml-doc2 is contained in memory in data item w-sdoc. It has the same length as data item w-s-doc.
- The XML document processed with xml-doc3 is contained in a memory area in UTF-16 representation (national). Data item doc-ptr points to this memory area, and the length of the document (in characters) is contained in data item doc-lg. You must supply both data items with appropriate values before processing begins.

DATA DIVISION

The XML document requires a data description entry in the FILE SECTION:

- All clauses except the EXTERNAL and GLOBAL clauses are forbidden for an XML document.
- The 01 record description entries in the FD describe parts of the XML document or the entire document. At least one of the record description entries must describe the root of the document. Multiple record description entries make sense if XML documents with different structures are to be processed with one data description entry or individual record description entries each only describe parts of the document, see also "OPEN DOCUMENT". You may omit individual or all children in the description for each node – regardless of whether elements or attributes are concerned. If no data item for a node in the tree is contained in the record description entry, no node from the corresponding subtree can occur in the record description entry.
- Nodes from the document must be described with the new IDENTIFIED clause (see also "Specifying an element or attribute name in the IDENTIFIED clause" in section "COBOL language elements for describing an XML document"). This clause is permitted only in record descriptions of files with organization XML.

> The data description entry does not describe attributes of a specific file, but only the logical structure of the XML document.

The 01 record description entries have nothing to do with the presentation of the file in the file administration system or anything similar. For this reason the record description entries in the FD of an XML file do **not** redefine themselves, as is implicitly the case with sequential, relative and indexed file organization.

If the XML document is contained in a file, any attributes are possible for the records (record format, record length, blocking, character code).

Example 12-38 File description entry for an XML document

```
...
FILE SECTION.
FD xml-doc2.
01 a                 IDENTIFIED BY "a".
   02 a-value       PIC X(10).
   02 att           IDENTIFIED BY "att"
                    ATTRIBUTE PIC 999.
   02 b1            IDENTIFIED BY "b"
                    PIC X(10).
01 b2                IDENTIFIED BY "b".
   02 b-value       PIC X(10).
   02 c             IDENTIFIED BY "c"
                    PIC X(10).
...
```

Comments:

- The file description entry refers to the corresponding SELECT clause in example 12-37.
- Record description entry a describes the section of the XML document consisting of its root ’a’, its attribute ’att’ and its children ’b’. Record description entry b2 describes the section of the XML document consisting of node ’b’ and this node’s children ’c’.
- The overlap of the two 01 structures in node ’b’ means that it is also possible to access grandchildren of root node ’a’ although each of the 01 structures describes only parents and children.

12.10.1.4 Statements for XML processing

In accordance with the description of XML documents which is based on file processing, XML documents are also processed using slightly extended statements for file processing:

- OPEN DOCUMENT:
  Starts processing of an XML (sub)document and assignment of a record description entry from the FD of the XML document to a document in the tree presentation.
- START ATTRIBUTE or START ELEMENT:
  Positions in the tree on the basis of a record description entry which has already been assigned.
- READ ATTRIBUTE or READ ELEMENT:
  Transfers data from the tree to corresponding data items of the assigned record description entry.
- CLOSE DOCUMENT:
  Ends processing of an XML (sub)document.

As with file processing, error handling is possible using statement-specific phrases (AT END, INVALID KEY) or by means of USE statements. The data item of the FILE STATUS clause describes the result of the statement; here new XML-specific values are possible in addition to the values already defined.

When processing an XML document, the following is important for understanding the statements:

- Nodes from the tree are assigned to the data items of a record description entry which have an IDENTIFIED clause.
- The current assignment is stored.
- The current assignment has an influence on subsequent statements.
- The current assignment can be modified by statements.

Element Position Vector (EPV)

The assignment of nodes in the tree to their description in the COBOL program is recorded using the Element Position Vector. The name corresponds to the 'Element Position Vector' used in the Technical Report "Native COBOL Syntax for XML Support" – however, it is used for assigning **all** nodes, element and attribute nodes. An internal management structure of the COBOL system for processing an XML document is concerned here: each node described in the program (i.e. only data items with an IDENTIFIED clause) has an entry in the EPV. The position of a currently assigned node from the tree is stored in this entry – this status is also referred to below with '**the data item has a position**'.

If no node has yet been assigned to a data item, or if the assignment has been made invalid by a statement, this special status is also recorded in the EPV. The interpretation of the stored positions by subsequent statements also depends on whether they resulted from a READ, OPEN or START statement.

Furthermore, a data item can only have a valid position if all the data items which are superordinate to it in the structure also have a valid position.

Assigning nodes

> An assignment only modifies the EPV, but does not transfer data.

Requirement for assigning a node from the tree to a data item

- The nodes in the tree and in the IDENTIFIED clause of the data item must be of the same type, i.e. both must be element nodes or both must be attribute nodes.
- The 'matching' of the name specified in the IDENTIFIED clause with the name of the node in the tree:
  - In the case of a BY phrase the names must be identical, except for trailing blanks.
  - In the case of a USING phrase any arbitrary name from the tree is regarded as matching.

Assignment procedure

1. **At most one** node from the tree is assigned to precisely one data item with an IDENTIFIED clause. Here the rules for the language elements ensure that there can be at most one such data item.
   The number of nodes and data items which needs to be taken into account depends on the statement.
2. For each data item to which a node from the tree could be assigned an attempt is made to assign precisely one of the children of the associated node (starting with the oldest child not yet assigned) to **each** of the data items with an IDENTIFIED clause which are directly subordinate to this data item.
3. Each data item examined in the steps above to which no node could be assigned is given the position 'invalid', as are all the data items which are subordinate to it.

Example 12-39 Principle of assigning nodes

XML document	Node position	COBOL data structure	EPV
<a>	1	FD xml-fil	
<d>ddd</d>	2	01 x IDENTIFIED BY "b".	inv
1	3	02 y IDENTIFIED BY "c".	inv
22	4	03 ...	
<c>level3</c>	5	01 z IDENTIFIED BY "a".	1
		02 u IDENTIFIED BY "b".	3
<c>level2</c>	6	03 v IDENTIFIED USING w-name.	inv
>		04 ...	
		02 w IDENTIFIED BY "d".	2
		03 ...	

Comments:

- Data items for values are irrelevant for assignment. They are therefore omitted in the COBOL description.
- Assumption for the first step: on the COBOL side the data items at level 01 (x and z) are examined, on the XML side only node 1 – this corresponds to the behavior of an OPEN DOCUMENT statement, see also "OPEN DOCUMENT". To assign the XML document to the COBOL description a data item must be found which specifies the name of the root node 'a' in its IDENTIFIED clause, namely data item z.
- In the following steps assignment is then continued on the tree side with the children of node 1 which has just been assigned - starting with the oldest, i.e. with nodes 2, 3, 4 and 6. On the side of the COBOL description only

the data items with an IDENTIFIED clause which are directly subordinate to the data item to which the parent node has just been assigned (i.e. data items u and w which are subordinate to data item z) are taken into consideration here.

- Node 3 with name 'b' can be assigned to data item u, and node 2 with name 'd' to data item w.
- Node 4, also with name 'b', cannot be assigned: in principle data item u would come into consideration here, but u is no longer available because it has already been assigned node 3. As node 4 cannot be assigned, none of its children, their children, etc. – here only node 5 – can be considered for further assignments, either.
- Node 6 also cannot be assigned as neither of the data items (u, w) which are directly subordinate to data item z specifies the name 'c' in its IDENTIFIED clause. Data item y does specify the appropriate name, but is not one of the data items taken into consideration because it is not a child of data item z.
- Since node 3 was assigned to data item u, the data items (v) which are directly subordinate to data item u also come into consideration for assignments. However, as the assigned node 3 has no children, no further assignments are possible.
- In the last step all the data items which were taken into consideration during the procedure to which no nodes could be assigned obtain invalid positions, these being, from the first step, data item x with its subordinate data item y, and, from the following steps, data item v.

The procedure for the assignment of nodes means:

- The order of assigned sibling nodes in the tree need not be the same as the order in which they are described in the COBOL data structure.
- The hierarchy of the data items in the COBOL description corresponds directly to the hierarchy of the assigned nodes in the tree. There are no gaps. For the nodes this means that if a node from the tree is assigned to a data item, its parent is also assigned to a data item of the COBOL structure.
- If more than one 01 structure exists in an FD, there can only ever be valid positions in one single 01 structure.
- The aim is not to create as many assignments as possible, but always to execute the assignments in the same, fixed order.

Statement sequences

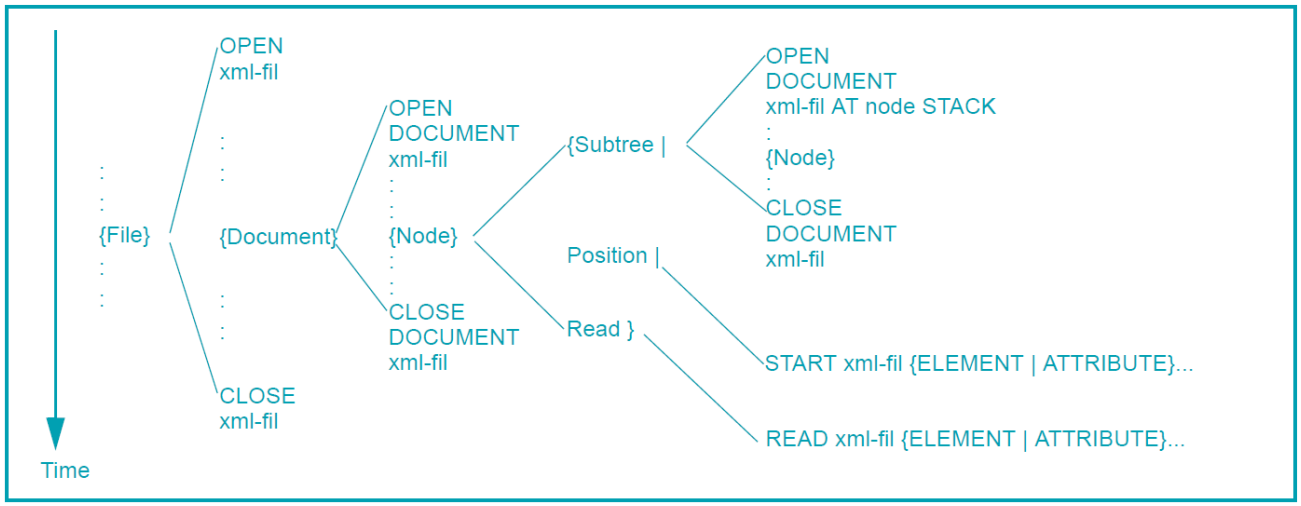
In the sections below the principal effects of the statements will be explained briefly, but without any details of the syntax and any rules.

Initially a minimum amount of information which shows how the statements function in principle is offered for each topic group. This is then illustrated by means of examples with comments. Finally, important information is combined for each topic group without laying any claim to completeness. To obtain all the information on a topic, please read the relevant sections in the chapter "XML".

The statements for processing the XML document or individual parts thereof must be executed in the order shown in the figure below. Further COBOL statements for actually processing the XML data have been omitted for the sake of clarity.

{} Indicates repeatable individual steps

| Separates possible alternatives



For the sake of clarity the presentation of the new statements largely dispenses with possible error situations and how these are handled. An overview of error situations is provided below in section "Error handling".

Important information on the presentation of examples 12-40 in section "OPEN DOCUMENT" through 12-56 in section "Namespace" below:

The following is displayed from left to right in the columns of the example tables:

- XML document:

  The XML documents have been reduced to what is important for the examples. They should not be misunderstood as well-formed XML.

  To make them easier to read, the XML documents include indents/blanks and line feeds. These editing measures are not intended as part of the values of a node.
- Positions which correspond to the XML nodes
- COBOL description
- Then, in each column, come the statements which are executed one after the other. Only the status **after** execution is recorded here: a node assigned in the EPV (i.e. its position) is noted in the IDENTIFIED clause for this purpose. Which statement created the position is added in parentheses:

  o OPEN DOCUMENT, START

  r READ

  All positions and contents which are modified as a result of the statement are displayed in bold print in the associated column, all unmodified values in italics.

  Statements and definitions which are not important for the particular case are omitted.

12.10.1.5 OPEN, CLOSE

Processing of an XML document must **always** begin and end with the OPEN and CLOSE statements normally used in file processing, irrespective of whether the document is located in a file or memory.

OPEN only permits the INPUT open mode for XML files. CLOSE permits none of the additional phrases.

These statements have the same effect for documents in a file as for non-XML files. For documents contained in memory, the connection to memory is established or a data point and the length from the ASSIGN clause are evaluated when the OPEN statement is executed.

The values which were possible for the I-O status in the previous method of file processing are to be expected in both types of document.

12.10.1.6 OPEN DOCUMENT

The second format of the OPEN statement (exclusively for XML files) with the DOCUMENT phrase introduces the actual processing of the XML document. It has the following three tasks:

- Procuring working memory which accommodates the document’s tree structure.
- Generating the tree presentation corresponding to the XML document in this memory.
- Executing an initial assignment of the tree to one of the 01 record description entries from the FD.

> The assignment procedure begins in the first step on the COBOL side with all data items at level 01 in the FD. On the XML side it begins with the root of the XML document tree. Further assignments are then implemented, as described under "Assignment procedure" in section "Statements for XML processing".

Positioning (START) and reading (READ) are then possible via successfully positioned data items.

OPEN DOCUMENT also permits **AT data-name-1 [STACK]** to be specified (here data-name-1 must be defined with an IDENTIFIED clause for element nodes). In contrast to the statement without the AT phrase, which makes a complete XML document available for processing, the AT phrase restricts the (further) processing to a subtree of this document. However, the root of the subtree must have become reachable (by means of preceding statements), i.e. the specified data-name-1 must have a valid position.

> The assignment procedure begins in the first step on the COBOL side with all data items at level 01 in the FD, and on the XML side with the root of a subtree. The node assigned to data-name-1 serves as the root. Further assignments are then implemented, as described under "Assignment procedure" in section "Statements for XML processing".

The STACK phrase causes the status of the EPV to be saved before the OPEN statement is executed and ensures that it can be restored later by means of a CLOSE DOCUMENT statement. Without the STACK phrase the 'old' EPV status is lost.

Example 12-40 OPEN DOCUMENT

XML document	Node position	COBOL data structure	**OPEN DOCUMENT xml-fil**	**OPEN DOCUMENT xml-fil AT y**
<a> <d>ddd</d> 22 <c>level3 </c> <c>level2</c> 	1 2 3 4 6	**FD xml-fil** **01 x IDENTIFIED BY "a".** **02 y IDENTIFIED BY "b".** **03 ...** **01 z IDENTIFIED BY "b".** **02 u IDENTIFIED BY "c".** **03 ...**	 **1(o)** **3(o)** **inv** **inv**	 **inv** **inv** **3(o)** **4(o)**

Comments:

- Information on the presentation of this example is provided on Statements for XML processing.
- After the first OPEN DOCUMENT statement, data item y has a valid position and can therefore be used in the AT phrase in the second OPEN DOCUMENT statement.
- The assignment procedure in the second OPEN DOCUMENT statement includes all 01 structures, but only regards the subtree with node 3 as a root. This results in invalid positions for data items x and y.
- As no STACK phrase was specified, only the valid positions in the EPV are available for further accesses, i.e. the root of the XML document (node 1) can no longer be reached by the subsequent statement unless the document and the file are closed and opened again.

Summary

In summary, the following must be observed in the case of the OPEN DOCUMENT statement:

- Access to an XML document always begins at its root. Consequently the OPEN DOCUMENT statement **without** AT phrase must always be executed first. Even the specification of the root as an identifier in the AT phrase does not work, because this requires the specified data item to have a valid position.
- OPEN DOCUMENT with an AT phrase but without a STACK phrase corresponds to a movement in the tree from the root toward the leaves without the option of reversing it.
- The COBOL record structures are actually only templates which describe sections of the tree. Starting from the root of the tree they can be made to cover any parts of the tree provided they match at runtime in at least one node, the data item from the AT phrase.
- One CLOSE DOCUMENT statement alone is not sufficient to open the same document a second time. For this purpose the file must also be closed and opened again.
- **OPEN DOCUMENT modifies the EPV only, but leaves data unchanged** .

12.10.1.7 CLOSE DOCUMENT

The mode of operation depends on the file's last OPEN DOCUMENT statement for which no CLOSE DOCUMENT statement has yet been provided:

- If the last OPEN DOCUMENT statement contained no STACK phrase, all entries in the EPV are set to invalid, the values are removed from the document's internal tree structure, and the working memory used for this is released.
- If the last OPEN DOCUMENT statement contained a STACK phrase, the EPV is reset to the status which it had before this OPEN statement was executed.
- **CLOSE DOCUMENT modifies the EPV only, but leaves data unchanged.**

A CLOSE DOCUMENT statement need not be specified: if it is missing, it is implied by a CLOSE statement or an OPEN DOCUMENT statement without a STACK phrase (for the same file).

Example 12-41 CLOSE DOCUMENT and OPEN DOCUMENT with STACK

XML document	Pos	COBOL data structure	**OPEN DOCUMENT xml-fil**	**OPEN DOCUMENT xml-fil AT y STACK**	**CLOSE DOCUMENT**	**CLOSE DOCUMENT**
<a>	1	FD xml-fil				
<d>ddd</d>	2	01 x IDENTIFIED BY "a".	**1(o)**	**inv**	**1(o)**	**inv**
22	3	02 y IDENTIFIED BY "b".	**3(o)**	**inv**	**3(o)**	**inv**
<c>level3</c>	4	03 ...				
		01 z IDENTIFIED BY "b".	**inv**	**3(o)**	**inv**	**inv**
<c>level2</c>	6	02 u IDENTIFIED BY "c".	**inv**	**4(o)**	**inv**	**inv**
		03 ...				

Comments:

- Information on the presentation of this example is provided on Statements for XML processing .
- The STACK phrase in the OPEN DOCUMENT statement does not immediately have an effect on the EPV's current contents after this statement (cf. the example 12-40 in section "OPEN DOCUMENT").
- The first CLOSE DOCUMENT statement restores the EPV with the status before the associated (second) OPEN DOCUMENT statement because the STACK phrase was specified there.
- The (first) OPEN DOCUMENT statement associated with the second CLOSE DOCUMENT statement contained no STACK phrase. Consequently all EPV entries are set to 'invalid'.

12.10.1.8 READ

The new, third format of the READ statement just for XML files is used for reading one or more nodes from the XML document tree. Reading an XML document differs from how a file was read to date in several respects:

1. The fixed, known data set 'record' which an old READ statement transfers has no equivalent in XML documents. Rather, the new READ statement transfers data on a set of nodes in the tree (element and/or attribute nodes) which may vary from statement to statement.
2. The fixed, implicit range 'next record' in the case of a old, sequential READ statement or 'whole file' in the case of a new, optional READ statement has no equivalent in XML documents. Rather, the **range** of the new READ statement – one or more subtrees of the document – must be specified in the statement.
3. The 'sequential' and 'random' access types which are possible with an old READ statement have no equivalents in XML documents. Rather, in the new READ statement the access type is defined statically for each node in the 01 structures of the FD.
4. The one key (primary or alternate) for random access in an old READ statement has no equivalent in XML documents. Rather, the 01 structures define the keys, and the individual new READ statement specifies which subset of these keys is to be used.
5. The new READ statement handles element nodes and attribute nodes differently when reading. This is not the case with the old READ statement.

The differences described above are reflected in the extended 01 data structures in the FD and the syntax and semantics of the new READ statement which differ from the old READ statement:

- As a rule only some of the data described in a 01 data structure is supplied with data – there is no INTO phrase (see also 1.). Data items can exist in the record description entry which is never modified by READ statements, e. g. the data item specified in IDENTIFIED BY.
- The range (see also 2.) is specified in the form of a data item in the new READ statement. A node in the tree must be assigned to this data item. This node determines the range:
  - If assignment took place using an OPEN or START statement, it covers the subtree defined by this node and the subtrees defined by its younger siblings.
  - If assignment took place using a READ statement, it covers only the subtrees defined by the younger siblings.
- The IDENTIFIED clauses in the record description entries of the FD define the access type (see also 3.):
  - BY: random access
  - USING: sequential access without the option of reading backward; there is no PREVIOUS phrase.
- It is not possible to switch from random to sequential reading (see also 3.) – there is no NEXT phrase. Further element nodes with the same name (i.e. the repetitions mentioned above) can also be read randomly because a node which has just been read no longer belongs to the range of a subsequent READ statement with an unchanged key.
- The data item which must be specified in the new READ statement also defines the subset of the keys to be used for random access (see also 4.). There is no KEY phrase. The following applies for the data item specified and all the data items which are subordinate to it:
  - The specifications made with IDENTIFIED BY describe the keys for random access.
  - The specifications made with IDENTIFIED USING describe the data items in which the name of the element or attribute node read sequentially is made available.

- The (context-sensitive) keywords ATTRIBUTE and ELEMENT specify the type of node to be processed by the READ statement (see also 5.).

A new READ statement looks like this:

```
READ xml-file ELEMENT data-name-1
```

or

```
READ xml-file ATTRIBUTE data-name-1
```

where data-name-1 is the name of the data item with an IDENTIFIED clause, in other words with the description of a node in the COBOL structure (not to be confused with the data item which contains the name of the node and which may have been specified in the IDENTIFIED clause). The ELEMENT or ATTRIBUTE phrase specified there must match the phrase in the READ statement.

The specified data item data-name-1 is decisive for the new READ statement. Consequently 'reading the file' is no longer spoken of in this context, but ’reading via (data item) dataname-1.’

A prerequisite for successful execution of the READ statement is that the assignment of a node from the tree to the specified data-name-1 already exists. When the READ statement is executed,

- the FD node is assigned to (a few) data items in a 01 structure,
- for individual nodes data is transferred from the tree to the COBOL data structure.

Assignment

The assignment procedure begins in the first step on the COBOL side with the data item which was specified in the READ statement.

- If the data item specified in the READ statement contains an IDENTIFIED BY clause with an ATTRIBUTE phrase, the attribute node assigned and **all** its siblings are checked for a possible assignment.
- If the data item specified in the READ statement contains an IDENTIFIED USING clause with an ATTRIBUTE phrase or an IDENTIFIED USING/BY clause with an ELEMENT phrase, on the XML side the root nodes of the subtrees from the range (as defined above) are checked for a possible assignment in the order from the older nodes to the younger ones.

Further assignments are then implemented, as described under "Assignment procedure" in section "Statements for XML processing" .

- Data items in the structure which are above the specified data item, next to it on the same level or in other 01 structures of the FD are not taken into consideration,and their assignment in the EPV remains unchanged.
- Existing positions for data items which are subordinate to the data item specified in the READ statement are irrelevant for assignment.
- It actually seems superfluous to include data item data-name-1 (for which an assigned node was already required as a prerequisite) in the assignment procedure again. However, this assignment need not necessarily still apply if, for example, in the case of random access the value of a key has changed in the meantime.
- As attribute names must be unique, the order of the attribute nodes is irrelevant. There can at most be one suitable name. In this case all attribute nodes are therefore always checked for a possible assignment.

Data transfer

The data-name-1 specified in the READ statement splits the data items defined in the FD into three sets: data-name-1 itself, the data items subordinate to it, and all other data items from the FD.

Depending on whether a node is successfully assigned to data-name-1, the affiliation to one of the three sets and the assignment of nodes to the subordinate data items, data items remain unchanged, are initialized, or are supplied with data from the tree. This is illustrated in the table below:

	A node has been assigned to data-name-1	No node has been assigned to data-name-1
data-name-1	Data transfer	Unchanged
Subordinate data item: node assigned	Data transfer	
Subordinate data item: no node assigned	Initialization	Unchanged
All other data items	Unchanged	Unchanged

The sending items in the XML document tree with USAGE NATIONAL (i.e. UTF-16) must be implied for data transfer.

Data transfer or initialization takes place in accordance with the COBOL rules (for this purpose conversion according to FUNCTION DISPLAY-OF takes place when transferring values from the tree to alphanumeric receiving items, and according to FUNCTION NUMVAL-C when transfer is to numeric receiving items, without a second parameter). For a node described in the COBOL structure (i.e. a data item with an IDENTIFIED clause), data transfer or initialization here concerns any data items for accommodating the name, the value and the display which may be connected to the node:

	Data transfer	**Initialization**
Data item from USING phrase in IDENTIFIED clause	Name of the node from the tree	Blank
Data item for the value	Value of the node from the tree	Initialized by means of INITIALIZE...TO DEFAULT
Data item from COUNT clause	1	0

Example 12-42 Sequential reading

XML document	Pos	COBOL data structure	**OPEN DOCUMENT xml-fil**	**READ xml-fil ELEMENT y**	**READ xml-fil ELEMENT y**	**READ xml-fil ELEMENT y**
<doc>000 <a>111 222 <c>333</c> </doc>	1 2 3 4	**FD xml-fil** **01 x IDENTIFIED BY** "doc". **02 y IDENTIFIED USING y-name.** **03 y-name PIC X.** **03 y-value PIC 999.**	**1(o)** **2(o)**	*1(o)* **2(r)** **a** **111**	*1(o)* **3(r)** **b** **222**	*1(o)* **at-end** **b** **222**
		FILE STATUS	**00**	**00**	**08**	**10**

Comments:

- Information on the presentation of this example is provided in section "Statements for XML processing".
- Successful assignment of a node (node 2) using the OPEN DOCUMENT statement is sufficient for reading via y. It is not necessary that superordinate data items (x) should already have been read in the 01 structure.
- Reading via y takes place sequentially as IDENTIFIED USING is specified.
- The position assigned to y before the first READ statement was set by an OPEN DOCUMENT statement. Consequently the subtrees with roots nodes 2 and 3 (in this order) can be taken into consideration as the range. Node 2 is assigned, and the name and value of the node read are transferred.
- Only one **single** data item which is **not** referenced in the IDENTIFIED clause may be subordinate to a data item with an IDENTIFIED clause: the value of the assigned node is transferred to this field (y value).
- If the value of a node is of no significance for the program, the data item for it can also be completely omitted, as with data item x.
- After the first READ statement, y is still assigned to node 2. However, the position indicates that it was created by READ. Only younger siblings, i.e. node 3, can be taken into consideration in the following READ statement.
- In the second READ statement the assigned subtree contains a node (c) which is not described in the corresponding COBOL structure y. I-O status 08 displays this.
- In the third READ statement the assigned node 3 has no younger siblings, and consequently the at end condition is created. The data items for name and value remain unchanged.

Example 12-43 Random reading and repetitions

XML document	Pos	COBOL data structure	**OPEN DOCUMENT xml-fil**	**READ xml-fil ELEMENT y**	**READ xml-fil ELEMENT y**	**READ xml-fil ELEMENT y**
<doc>000 <c>111</c> <a>222 <a>333 <c>444</c> </doc>	1 2 3 4 5	**FD xml-fil** **01 x IDENTIFIED BY** "doc". **02 y IDENTIFIED USING y-name.** **03 y-name PIC X VALUE "a".** **03 y-value PIC 999.**	**1(o)** **3(o)** *a*	*1(o)* **3(r)** *a* **222**	*1(o)* **4(r)** *a* **333**	*1(o)* **at-end** *a* *333*
		FILE STATUS	**00**	**00**	**00**	**10**

Comments:

- Information on the presentation of this example is provided in section "Statements for XML processing".
- Reading via y takes place randomly as IDENTIFIED BY is specified. The current value of the key is 'a'.
- The position assigned to y in the first READ statement was set by an OPEN DOCUMENT statement. Consequently the subtrees with roots nodes 3, 4 and 5 are taken into consideration as the range. Node 3 is the first of these which can be assigned.
- After the first READ statement, node3 is still assigned to y. However, the position now indicates that it was created by READ.
- The range of the second READ statement includes the subtrees with roots nodes4 and 5. Of these, node 4 can be assigned. As node 3 is no longer part of the range, this READ statement reads the repetition, i.e. the second example of the node with the name 'a'.
- In the third READ statement the assigned node 4 does have a younger sibling (node 5), but no assignment is possible because the name does not match.
- In the event of random reading, an **at end** condition may also occur.

Example 12-44 Random reading with changing key value

XML document	Pos	COBOL data structure	**OPEN DOCUMENT xml-fil**	**READ xml-fil ELEMENT y**		**READ xml-fil ELEMENT y**
<doc>000 <c>111</c> <a>222 <a>333 <c>444</c> </doc>	1 2 3 4 5	**FD xml-fil** **01 x IDENTIFIED BY** "doc". **02 y IDENTIFIED BY y-name.** **03 y-name PIC X VALUE "a".** **03 y-value PIC 999.**	**1(o)** **3(o)** *a*	*1(o)* **3(r)** **a** **222**	**MOVE "c" TO y-name**	*1(o)* **5(r)** *c* ***444***

Comments:

- Information on the presentation of this example is provided in section "Statements for XML processing".
- Up to and including the first READ statement this example is no different to the previous one.
- As in the previous example, the range of the second READ statement includes the subtrees with the roots nodes 4 and 5. Changing the value of the y-name key to 'c' now causes node 5 to be assigned.
- As the OPEN DOCUMENT statement was executed with the key value 'a', node 2 can never again be reached by reading via y despite the subsequently modified and actually suitable key value 'c'. The open function assigned node 3 to y. Consequently only this node and its younger siblings - but not the older sibling node 2 - come into consideration as the range for all further READ statements.

Example 12-45 Effect of changing key values

XML document	Pos	COBOL data structure	**OPEN DOCUMENT xml-fil**		**READ xml-fil ELEMENT z**	**READ xml-fil ELEMENT y**
<doc>111 <a>222 <c>333</c> <d>444</d> 555 <c>666</c> <d>777</d> </doc>	1 2 3 4 5 6 7	**FD xml-fil** **01 x IDENTIFIED BY** "doc". **02 x-value PIC 999.** **02 y IDENTIFIED BY y-name.** **03 y-name PIC X VALUE "a".** **03 y-value PIC 999.** **03 z IDENTIFIED BY z-name.** **04 z-name PIC X VALUE "c".** **04 z-value PIC 999.**	**1(o)** **2(o)** *a* **3(o)** *c*	**MOVE "b" TO y-name** **MOVE "d" TO z-name**	*1(o)* *2(o)* *b* ***4(r)*** *d* ***444***	*1(o)* **5(r)** *b* **555** **7(r)** *d* **777**

Comments:

- Information on the presentation of this example is provided in section "Statements for XML processing".

- Modifications to key values (y-name and z-name) after successful assignment following the OPEN DOCUMENT statement do not always affect subsequent assignments.
- The two MOVE statements after OPEN DOCUMENT suggest that the first READ statement should access the subtree with root node 5 (name = 'b') randomly and in this access node 7 (name = 'd'). However, reading via data item z is not suitable for this purpose.
- Values of keys which in the COBOL structure are **superordinate** (y with y-name) to the key used while reading (z with z-name) are **not taken into consideration** during assignment. Changes to such key values have no effect. In the example access continues to take place in the assigned subtree with root 2, and in this to node 4.
- In addition to changing the key values, reading must also take place via a data item which is not subordinate to any of the modified keys, such as data item y in the second READ statement.

Example 12-46 Transferred data

XML document	Node position	COBOL data structure	**OPEN DOCUMENT xml-fil**	**READ xml-fil ELEMENT x**
<doc>000	1	**FD xml-fil**		
<a>22	2	**01 x IDENTIFIED BY "doc".**	**1(o)**	**1(r)**
pqrst	3	**02 x-value PIX X(5).**		**000'BLANK''BLANK'**
<c>99</c>	4	**02 y IDENTIFIED BY "b".**	**3(o)**	**3(r)**
		03 y-value PIC XX.		**pq**
		02 z IDENTIFIED BY "a"	**2(o)**	**2(r)**
		COUNT z-count.		**1**
		03 z-value PIC 999V99.		**02200**
		02 u IDENTIFIED BY "C"	**inv**	**inv**
		COUNT u-count.		**0**
		03 u-value PIC XXX.		**'BLANK''BLANK''BLANK'**

Comments:

- Information on the presentation of this example is provided in section "Statements for XML processing".
- In the case of random reading, the order of the data items in the COBOL structure – first y for nodes with the name 'b', then z for nodes with the name 'a' – and that of the assigned nodes in the tree – first that with the name 'a', then that with the name 'b' – is irrelevant.
- Non-numeric values from the tree which are too short are filled with blanks (x-value), Values which are too long are truncated (y-value).
- Numeric values are transferred on a decimal-point-oriented basis (z-value), possibly also converted to another USAGE.
- The COUNT elementary items indicate whether nodes could be **assigned** to a data item (z-count) or not (u-count). Only the READ statement sets this display.
- The keywords are case-sensitive. Node 4 with the name 'c' is consequently not assigned to data item u with the key value 'C'.
- Data items which are subordinate to the data item via which reading took place (x) and to which no node is assigned (u) have invalid items in the EPV and are initialized.

Example 12-47 Elements and attributes

XML document	Pos	COBOL data structure	OPEN DOCUMENT xml-fil	READ xml-fil ELEMENT x	READ xml-fil ATTRIBUTE y-att	READ xml-fil ELEMENT y	READ xml-fil ELEMENT y
<doc	1	**FD xml-fil**					
d1="g"	2	**01 x IDENTIFIED**	**1(o)**	**1(r)**	*1(r)*	*1(r)*	*1(r)*
d2="f">	3	**BY "doc".**					
<a	4	**02 x-att1**	**3(o)**	**3(r)**	*3(r)*	*3(r)*	*3(r)*
a1="1"	5	**IDENTIFIED**					
a2="2"	6	**BY "d2"**					
a3="3">	7	**ATTRIBUTE.**					
ppp		**03 x-att1-value**		**f**	*f*	*f*	*f*
		PIC X.					
<a	8	**02 x-att2**	**inv**	**inv**	*inv*	*inv*	*inv*
a1="9"	9	**IDENTIFIED**					
a4="8">	10	**BY "d3"**					
qqq		**ATTRIBUTE**					
		COUNT					
</doc>		**x-att2-count.**		**0**	*0*	*0*	*0*
		03 x-att2-value		**'BLANK'**	*'BLANK'*	**'BLANK'**	**'BLANK'**
		PIC X.					
		02 y IDENTIFIED	**4(o)**	**4(r)**	*4(r)*	**8(r)**	**at-end**
		USING y-name.					
		03 y-name		**a'BLANK'**	*a'BLANK'*	**a'BLANK'**	*a'BLANK'*
		PIC XXX.		**'BLANK'**	*'BLANK'*	**'BLANK'**	*'BLANK'*
		03 y-att	**5(o)**	**5(r)**	**6(r)**	**9(r)**	**inv**
		IDENTIFIED					
		USING					
		y-att-name					
		ATTRIBUTE.					
		04 y-att-name		**a1**	**a2**	**a1**	*a1*
		PIC XX.					
		04 y-att-value		**001**	**002**	**009**	*009*
		PIC 999.					
		03 y-value		**ppp**	*ppp*	**qqq**	*qqq*
		PIC XXX.					
		FILE STATUS	**00**	**08**	**00**	**00**	**10**

Comments:

- Information on the presentation of this example is provided in section "Statements for XML processing".
- The OPEN DOCUMENT statement assigns both element nodes and attribute nodes simultaneously.

- In the case of predefined keys, node 3 is assigned to attribute x-att1 (key='d2'). No node can be assigned to attribute x-att2 (key='d3'). If no keys are defined, the first attribute node (node 5) is assigned to attribute y-att.
- The COUNT clause can also be used for attributes (x-att2).
- Attribute d2, which is not described in the COBOL structure, results in I-O status 08 when reading takes place via x.
- The rule with the range also applies for sequential reading of attributes, even if the subtrees here only consist of one node. The range of the second READ statement includes nodes 6 and 7. As every name fits in the case of sequential access, node 6 is therefore assigned.
- It is not necessary to read sequentially to the end via a data item, e.g. also 'a3' via y-att. Another elementary item may be switched to beforehand (y).
- The third READ statement (via y) assigns node 8 to y and the first attribute 'a1' of this node (node 9) to the subordinate attribute (y-att).
- When an end is reached in sequential reading, this is noted in the EPV (for data item y). In this case the EPV entries for all subordinate data items are set to invalid (y-att). However, the values of the data items remain unchanged (y-name, y-value, y-att-name, y-att-value).
- BY (x-att1) and USING (y) may be specified for data items with the same immediate superordinate group item (x) because they describe different types of nodes (attribute and element respectively).

Example 12-48 Random reading of attributes

XML document	Pos	COBOL data structure	**OPEN DOCU-MENT xml-fil**	**READ xml-fil ATTRI-BUTE y-att**	**READ xml-fil ATTRI-BUTE y-att**		**READ xml-fil ATTRI-BUTE y-att**
<doc	1	**FD xml-fil**					
<a	2	**01 x IDENTIFIED BY** "doc"**.**	**1(o)**	*1(o)*	*1(o)*		*1(o)*
a1="1"	3	**02 y IDENTIFIED BY "a".**	**2(o)**	*2(o)*	*2(o)*		*2(o)*
a2="2"	4	**03 y-att IDENTIFIED**	**4(o)**	**4(r)**	**4(r)**		**3(r)**
a3="3">	5	**BY y-att-name**					
ppp		**ATTRIBUTE.**				**MOVE**	
		04 att-name PIC XX	*a2*	*a2*	*a2*	**"a1" TO**	*a1*
</doc>		**VALUE "a2".**		**002**	**002**	**att-name**	**001**
		04 att-value PIC 999.					

Comments:

- Information on the presentation of this example is provided in section "Statements for XML processing".
- The first READ statement assigns the attribute node 4 with the predefined key 'a2' to y-att.
- The second READ statement for random reading of an attribute therefore examines – because of the unchanged key – nodes 3, 4 and 5 for a possible assignment, i.e. the node already assigned and all its siblings: it assigns node 4 once again.
- Since all nodes are examined when attributes are read randomly (in contrast to the ELEMENT phrase, cf. the example "Random reading and repetitions"), the at end condition can never occur in the case of an unchanged key (which also exists in the tree).

- Since **all** sibling nodes are examined when attribute nodes are read randomly, changing the key value (to 'a1') with the third READ statement also permits older siblings (node 3) of the node assigned to data item y-att (node 4) to be read.

Example 12-49 Multiple reading of the same node

XML document	Pos	COBOL data structure	**OPEN DOCUMENT xml-fil**	**READ xml-fil ELEMENT z**	**READ xml-fil ELEMENT z**	**READ xml-fil ELEMENT y**
<doc>111 <a>222 <c>333</c> <d>444</d> 555 </doc>	1 2 3 4 5	**FD xml-fil** **01 x IDENTIFIED BY** "doc"**.** **02 x-value PIC 999.** **02 y IDENTIFIED USING y-name.** **03 y-name PIC X.** **03 y-value PIC 999.** **03 z IDENTIFIED USING z-name.** **04 z-name PIC X.** **04 z-value PIC 999.**	*1(o)* *2(o)* **3(o)**	*1(o)* *2(o)* **3(r)** **c** **333**	*1(o)* *2(o)* **4(r)** **d** **444**	*1(o)* **2(r)** **a** **222** **3(r)** **c** **333**

Comments:

- Information on the presentation of this example is provided in section "Statements for XML processing".
- Data on node 3 is supplied through reading via z. The second READ statement sets the position of z to node 4.
- Only the position which y assigns itself (node 2) is relevant for subsequent reading via the superordinate data item. The positions of subordinate data items (z) are irrelevant.
- The third READ statement then assigns node 3 to z again and also supplies its data (not the data of node 5).

Restricted data transfer using ONLY

As an option, READ permits **ONLY** to be specified before the keyword ELEMENT. In this case data is only transferred in conjunction with the data item specified in the READ statement, if necessary including any existing attributes. No data transfer takes place for any other subordinate data items. However, node assignment also takes place for the subordinate data items. As no data has yet been transferred for the subordinate data items, in the case of subsequent read accesses its EPV entry must be interpreted as if it had been created by an OPEN DOCUMENT statement.

ONLY is forbidden for reading attributes and also makes no sense, because attribute nodes have no children.

Example 12-50 Restricted data transfer using ONLY

XML document	Pos	COBOL data structure	**OPEN DOCUMENT xml-fil**	**READ xml-fil ELEMENT y**	**READ xml-fil ELEMENT z**	**READ xml-fil ONLY ELEMENT y**	**READ xml-fil ELEMENT z**
<doc>1 	1 2 3	**FD xml-fil** **01 x IDENTIFIED** **BY "doc".**	**1(o)**	*1(o)*	*1(o)*	*1(o)*	*1(o)*

2		02 x-value PIC 9.					
4	4	02 y IDENTIFIED	2(o)	2(r)	*2(r)*	6(r)	*6(r)*
<c>5</c>	5	USING y-name.					
		03 y-name PIC X.		a	*a*	a	*a*
<a	6	03 y-value PIC 9.		2	*2*	6	*6*
q="B">	7	03 z IDENTIFIED	4(o)	4(r)	5(r)	8(o)	8(r)
6		USING z-name.					
8	8	04 z-name PIC X.		b	c	*c*	b
<c>9</c>	9	04 z-value PIC 9.		4	5	*5*	8
		03 y-att IDENTIFIED	3(o)	3(r)	*3(r)*	7(r)	*7(r)*
</doc>		BY "q"					
		ATTRIBUTE.		A	*A*	B	*B*
		04 y-att-value PIC					
		X.					

Comments:

- Information on the presentation of this example is provided in section "Statements for XML processing".
- The first READ statement reads node 2 with its attribute node 3, as well as the subordinate node 4. The following READ statement via the subordinate data item z reads the next node 5 sequentially.
- By way of comparison, this is how reading takes place in a subtree which has a similar structure and root 6, but has an ONLY phrase, i.e. with no transfer of the subordinate elements: the name and value of the element ('a' and '6') are transferred for data item y from the READ statement. Data transfer ('B') also takes place for the data items which are subordinate to y and describe attribute nodes (y-att).
- Nodes (node 8) are assigned for the data items which are subordinate to y and describe element nodes (here only z), but no data is transferred. Data contained in the COBOL structure remains unchanged. The assignment in the EPV records that it was **not** created by READ.
- The subsequent fourth READ statement via the subordinate data item (z) now first transfers the name and value of the assigned node 8, not the name and value of the following node like the second READ statement.

Summary

In summary, the following must be observed in the case of the READ statement:

- Random read access to elements only ever permits **access to children and younger siblings**, but **never to parents or older siblings**.
- Random read access without an intermediate OPEN or START statement or modification of the key supplies the repetitions in the case of an element node and the at end condition after the last copy. In the case of attribute nodes, the same (sole) copy is supplied repeatedly.
- **Data in mixed content** resulting from inserted, subordinate elements is **combined** to form one piece.
- A READ statement without an ONLY phrase supplies the value 08 for the I-O status if the subtree used for assignment contains at least one node below the root which can never be assigned to a data item in the COBOL structure.
- **The fundamental option of being able to assign the node to a data item is involved here. This has nothing to do with whether this assignment has been implemented in the current READ statement and data has been transferred.** In particular this means: repetitions of element names in the tree of which a copy could be assigned do not result in value 08 for the I-O status.

- In the case of a READ statement containing the ONLY phrase, I-O status 08 can only exist for non-assignable attributes of the element which was read, but not for other nodes of the subtree examined.
- Current positions or assigned nodes for data items which are subordinate in the COBOL structure to the data item specified in the READ statement are irrelevant.
- The data item specified in the READ statement determines simultaneously **where** the search/reading is to take place and **what** is to be searched for/read.
- A READ statement is generally a mixture of sequential and random access, all element children and attribute children of a node being accessed in the same way, irrespective of how the node itself is accessed.
- In the EPV a READ statement only changes the position of the data item via which reading takes place and of the data items which are subordinate to this data item. The positions of all other data items remain unchanged.

12.10.1.9 START

The new second format of the START statement solely for XML files is used for positioning on one or more nodes (elements and/or attributes) from the XML document tree. A comparison with the old START statement for files reveals only the following minor differences for the simplest form:

- A START statement can position on multiple keys simultaneously: on the data item specified in the statement and all data items which are subordinate to it.
- The KEY phrase is omitted. As an XML document tree is generally a two-dimensional structure, it does not make sense to transfer the positioning 'before' and 'after' a key which is possible with one-dimensional files to XML documents (with more than one key simultaneously).
- The type of data item which is used for positioning must be specified analogously to the XML format of the READ statement.

A new READ statement looks like this:

```
START xml-file ELEMENT data-name-1
```

or

```
START xml-file ATTRIBUTE data-name-1
```

The difference between the START statement and the READ statement which also positions on nodes in the tree is that the range starting from the data item specified in the statement is defined.

The data item specified in the START statement plays no role itself here. In contrast to the READ statement, no node need be assigned to this data item. It is decisive that a node from the tree must be assigned to the **data item which is directly superordinate** in the COBOL structure. The range of the START statement then includes the subtree which has this node as its root.

The assignment procedure begins in the first step on the COBOL side with the data item which was specified in the READ statement. On the XML side **all the children** of the root node which are in the range (as defined above) are checked for a possible assignment in the order from the older ones to the younger ones (in other words, in contrast to the READ statement, if data-name-1 has a valid position, its older siblings are also checked). Further assignments are then implemented, as described under "Assignment procedure" in section "Statements for XML processing".

START modifies the EPV only, but leaves data unchanged.

Example 12-51 Positioning using START

XML document	Pos	COBOL data structure		OPEN DOCUMENT xml-fil	READ xml-fil ELEMENT x		START xml-fil ELEMENT y	READ xml-fil ELEMENT y
<doc>1	1	**FD xml-fil**						
<a>2	2	**01 x IDENTIFIED**		**1(o)**	**1(r)**		*1(r)*	*1(r)*
3	3	**BY "doc".**						
<c>4</c>	4	**02 x-value PIC 9.**			**1**		*1*	*1*
		02 y IDENTIFIED	MOVE	**inv**	**inv**	MOVE	**2(o)**	**2(r)**
<a>5	5	**BY y-name.**	"f" TO			"a" TO		
6	6	**03 y-name PIC X.**	y-	*f*	*f*	y-	*a*	*a*
<c>7</c>	7	**03 y-value PIC 9.**	name			name		**2**
		03 z IDENTIFIED		**inv**	**inv**		**3(o)**	**3(r)**
</doc>		**BY z-name.**	MOVE			MOVE		
		04 z-name PIC X.	"g" TO z-	*g*	*g*	"b" TO z-	*b*	*b* **3**
		04 z-value PIC 9.	name			name		

Comments:

- Information on the presentation of this example is provided in section "Statements for XML processing".
- The data item (y) specified in the START statement need not have a valid position. The directly superordinate data item (x with node 1) must have a valid position.
- The range of the START statement is the subtree with the root node 1. The children of this node are checked (first node 2, then node 5) for assignment to data item y. Node 2 matches first, and assignment is continued with this subtree.
- As the START statement only provides a position but transfers no data, the following START statement supplies the data to the positioned node 2 and the subtree determined by node 2.

Example 12-52 Repeated positioning on the same node (continuation of Example 12-51)

XML document	Pos	COBOL data structure	...	READ xml-fil ELEMENT y	START xml-fil ELEMENT y
<doc>1	1	**FD xml-fil**			
<a>2	2	**01 x IDENTIFIED**	*1(r)*	*1(r)*	*1(r)*
3	3	**BY "doc".**			
<c>4</c>	4	**02 x-value PIC 9.**		*1*	*1*
		02 y IDENTIFIED	*1*	**5(r)**	***2(o)***
<a>5	5	**BY y-name.**	*2(r)*		
6	6	**03 y-name PIC X.**		*a*	*a*
<c>7</c>	7	**03 y-value PIC 9.**		**5**	*5*
		03 z IDENTIFIED	*a*	**6(r)**	**3(o)**
</doc>		**BY z-name.**	*2*		
		04 z-name PIC X.	*3(r)*	*b*	*b*
		04 z-value PIC 9.		**6**	*6*
			b		
			3		

Comments:

- Information on the presentation of this example is provided in section "Statements for XML processing".
- Reading via data item y supplies the next copy of the node with the name 'a'.
- The positions of data items y and z are irrelevant for the following START statement via y; only the position of data item x, which is superordinate to y, is relevant.
- As the position of data item x has not changed since the first START statement (see Example 12-51), the range is still the same. Since the values of keys y-name and zname are also unchanged, the assignment is the same as with the first START statement.
- The START statement permits positioning once more on one of the older siblings which have already been read (node 2). This is never possible with the READ statement; there only younger siblings would be contained in the range.
- The START statement transfers no data. Consequently the contents of the values be longing to y and z (y-value and z-value) are unchanged, but do not correspond to those of the nodes assigned by the new positioning (nodes 2 and 3).

INDEX phrase with START

START also permits the following to be specified after data-name-1:

```
INDEX {identifier-1 | integer-1}
```

The integer positive value specified determines which node from a number with the same element name is to be positioned on.

The INDEX phrase is also permitted for attribute nodes, although attribute names must be unique. If the nth attribute is to be read regardless of the name (i.e. IDENTIFIED USING), the INDEX phrase still makes sense for attribute names.

Example 12-53 INDEX phrase with START

XML document	Pos	COBOL data structure		**OPEN DOCU-MENT xml-fil**	**START xml-fil ELE-MENT z INDEX 2**	**START xml-fil ELE-MENT u INDEX 3**	**START xml-fil ELE-MENT y**
<doc>1	1	**FD xml-fil**					
<a>2	2	**01 x IDENTIFIED BY "doc".**	MOVE "a" TO y-name	**1(o)**	*1(o)*	*1(o)*	*1(o)*
3	3	**02 x-value PIC 99.**					
<c>4	4	**02 y IDENTIFIED BY y-name.**		**2(o)** *a*	*2(o)* *a*	*2(o)* *a*	**2(o)** *a*
<d>5</d>	5	**03 y-name PIC X.**					
<d>6</d>	6	**03 y-value PIC 99.**					
</c>		**03 z IDENTIFIED BY z-name.**	MOVE "c" TO z-name	**4(o)** *c*	**7(o)** *c*	*7(o)* *c*	**4(o)** *c*
<c>7	7	**04 z-name PIC X.**					
<e>8</e>	8	**04 z-value PIC 99.**					
<e>9</e>	9	**04 u IDENTIFIED BY u-name.**		**inv** *e*	**8(o)** *e*	**inv** *e*	**inv** *e*
</c>		**05 u-name PIC X.**					
		05 u-value PIC 99.	MOVE "e" TO u-name				
<f>10</f>	10						
</doc>							
		FILE STATUS		**00**	**00**	**23**	**00**

Comments:

- Information on the presentation of this example is provided in section "Statements for XML processing".
- The range of the first START statement via z is the subtree with the root node 2 (as the node which is assigned to the superordinate data item y).
- In the first step the children of node 2, i.e. nodes 3, 4 and 7, are checked for a possible assignment. Since 2 is specified as the index, not the first matching node (node 4) is assigned, but the second matching node (node 7).
- The index refers only to the data item specified in the START statement, not to the as signment of subordinate data items (u). Of the 2 copies with the name 'e', the first is assigned. If an INDEX phrase is also to be effective for subordinate data items, a corresponding START statement is required for each of these.
- The assignment procedure for the second start STATEMENT takes nodes 8 und 9 into consideration (as children of node 7, which is assigned to z), and therefore does not find as many nodes as requested (index 3, but only 2 matching nodes with the name 'e'). Consequently no node can be assigned. The position of the data item (u) is set to invalid and an invalid key condition exists.
- The third START statement’s range is the entire document. Positioning which has alrea dy taken place for subordinate data items (y, z, u) is irrelevant.

- In the case of the third START statement, no node can be assigned to the data item (u).
  As u is **subordinate** to data item y which is specified in the statement, this situation does **not** lead to an invalid key condition.

12.10.1.10 Error handling

The language elements used for handling errors in current file input/output are also available for processing XML documents using the new statement formats:

- OPEN DOCUMENT and READ: **AT END** and **NOT AT END** phrases
- START: **INVALID KEY** and **NOT INVALID KEY** phrases
- For all new formats of the statements: **USE procedures** and **additional new XML-specific values for the I-O status** (FILE STATUS).
- OPEN DOCUMENT: new exceptional condition EC-XML-CODESET-CONVERSION

The [NOT] END and [NOT] KEY phrases correspond to the phrases in the old input/output statements, both in their syntax and also in their function.

OPEN DOCUMENT

Basically multiple XML documents are permitted in a file one after the other. However, at present only files with a single document are supported. Each OPEN DOCUMENT statement without an AT phrase closes any document which may still be open and opens the next document in the file. If there is no next document or the file is empty, the at end condition occurs and the I-O status is set to 10.

If the XML document contains characters which cannot be represented as national characters (UTF-16), the exceptional condition EC-XML-CODESET-CONVERSION occurs. This exceptional condition does not lead to termination. The characters concerned are replaced in the XML document tree by the replacement character '.'. EC-XML-CODESET-CONVERSION occurs only when the external representation of the XML document is converted to a tree – at most once.

READ

The AT END phrase refers only to an at end condition for the one data item which was specified in the READ statement, not for subordinate data items. The **at end condition** occurs during both sequential **and random reading** if there is no node in the XML document tree which can be assigned to the data item (from the statement). The corresponding value for the I-O status is 10.

For transfers from the XML document tree, national (UTF-16) must be assumed as the en-coding for representing the data. As a result, the exceptional condition EC-DATA-CONVER-SION can occur during each transfer in the context of a READ statement.

In contrast to EC-XML-CODESET-CONVERSION from the OPEN DOCUMENT statement, EC-DATA-CONVERSION can occur when data is transferred from the tree to the program – possibly even several times during a single READ statement.

START

The INVALID KEY phrase refers only to an invalid key condition for the one data item which was specified in the START statement, not to subordinate data items The invalid key condition occurs when there is no node in the XML document tree which can be assigned to this data item. The corresponding value for the I-O status is 23.

All statements

When the order of the statements is not observed (see also "Statement sequences" in section "Statements for XML processing"), the prerequisites for executing a single statement are not satisfied. This results in unsuccessful execution on account of logical errors and a corresponding I-O status.

I-O status for XML files

I-O status	Meaning
	Successful execution
00	The statement was executed successfully. No further information is available.
05	OPEN statement for a non-existent optional file.
08	A READ statement was unable to assign some elements and/or attributes from the XML document to any COBOL description entry.
10	**Unsuccessful execution: at end condition** 1. A READ statement was used to attempt to read an attribute or element which does not exist. 2. An attempt was made to execute an OPEN DOCUMENT statement for an empty or a non-existent optional file. 3. An attempt was made to open a second document in an XML file by means of an OPEN DOCUMENT statement.
23	**Unsuccessful execution: invalid key condition** 1. A START statement was used to attempt to position on an attribute or element which does not exist. 2. The index phrase in the START statement was not positive.
25	An attempt was made to execute a START phrase without a valid position.
	Unsuccessful execution: permanent error
30	No further information is available.
35	OPEN for non-existent file.
39	An attempt was made to open a file with unsuitable file attributes.
3A	An attempt was made to execute an OPEN DOCUMENT statement for an XML document which is not well-formed. This also includes file contents which are not XML documents.
3D	An attempt was made to execute an OPEN DOCUMENT statement, but the encoding in which the XML document is represented could not be determined.
	Unsuccessful execution: logical error
41	An attempt was made to execute an OPEN statement for a file which is already open.
42	An attempt was made to execute a CLOSE statement for a file which has not been opened.
46	

	1. An attempt was made to execute an OPEN DOCUMENT statement (with AT phrase) or a READ statement without a valid position. 2. An attempt was made to execute an OPEN DOCUMENT statement (without AT phrase) after the preceding OPEN DOCUMENT statement had caused an at end condition.
47	An attempt was made to execute a READ or START statement for a file which has not been opened.
4B	An attempt was made to execute an OPEN DOCUMENT or CLOSE DOCU-MENT statement for a file which has not been opened.
4C	An attempt was made to execute an OPEN DOCUMENT, START or READ statement in which names in IDENTIFIED clauses for sibling nodes were not unique.
4D	An attempt was made to execute a CLOSE DOCUMENT, OPEN DOCUMENT (with AT phrase), READ or START statement for a document which has not been opened.
4E	An attempt was made to execute an OPEN DOCUMENT, READ or START statement in which element and/or attribute names specified in the COBOL program cannot be represented in UTF-16.
	Other cases of unsuccessful execution
90	System error - no further information is available about the cause.
91	OPEN error. Actual cause may be apparent from the SIS code
97	The memory required for an OPEN DOCUMENT statement is not available.

Table 47: I-O statuses for XML files

These I-O statuses refer only to the primary XML file which is to be processed. If errors occur during access (which may also be required) to external entities, these generally lead to I-O status 3A.

Example 12-54 Ambiguous names

XML document	Pos	COBOL data structure		**OPEN DOCUMENT xml-fil**		**READ xml-fil ELEMENT x**
<doc>1 <a>2 3 <a>4 </doc>	1 2 3 4	**FD xml-fil** **01 x IDENTIFIED BY "doc".** **02 y IDENTIFIED BY y-name.** **03 y-name PIC X.** **02 z IDENTIFIED BY z-name.** **03 z-name PIC X.**	**MOVE "a" TO y-name** **MOVE "b" TO z-name**	**1(o)** **2(o)** *a* **3(o)** *b*	**MOVE "a" TO z-name**	**inv** **inv** *a* **inv** *a*
		FILE STATUS		**00**		**4C**

Comments:

- Information on the presentation of this example is provided in section "Statements for XML processing".

- The current keywords of data items y and z are the same in the READ statement: This is forbidden because no unique assignment is defined for this case (is node 2 to be assigned to z and node4 to y or vice versa?). The READ statement consequently returns the I-O status 4C.
- In this case the data item **via which reading took place** – although the problem did not occur in this data item at all but in subordinate ones – and all the data items subordinate to it contain an invalid position.

12.10.1.11 Namespace

Qualified names in XML documents consist of a local name and a namespace within which the local name must be unique. The namespace is specified like an attribute in the case of an element:

- It can define a standard namespace: `xmlns="..."`
  This standard namespace applies for this element and all subordinate elements provided no other namespace is specified there in an element name, but not for attributes.
- It can be completely disabled: `xmlns=""`
  This empty namespace also applies for this element and all those subordinate to it, provided nothing else is specified there.
- It can define an abbreviation (prefix) for a (non-standard) namespace: `xmlns:prefix="..."`
  This is also be permitted more than once using different prefixes to make more than one namespace available. Such a prefix may precede the local name in this element, all the subordinate elements **and also** in the attributes of these elements in order to form a unique, qualified name with the format `prefix:local-name`.

The element and attribute names listed in the **IDENTIFIED clause always** describe **the local name**. The optional NAMESPACE phrase in the IDENTIFIED clause is used for processing namespaces from the XML document. For the element or attribute described, it permits:

- a required namespace to be predefined:
  NAMESPACE IS {data-name-3 | literal-2}
- arbitrary namespaces to be accepted: **NAMESPACE USING data-name-4**
  The effect is the same as for the USING phrase in IDENTIFIED: the namespace which is read is returned in data-name-4.
- the empty namespace to be defined: **NAMESPACE IS NULL**

The implicit effect of namespaces for subordinate elements of an XML document is reflected in COBOL:

- The NAMESPACE phrase of an IDENTIFIED clause applies for the specified data item **and** is inherited by all subordinate data items with an IDENTIFIED clause and ELEMENT phrase, but **not** by data items with the ATTRIBUTE phrase.
- When a subordinate data item specifies a NAMESPACE phrase itself, this has priority over any that may have been inherited.
- If the NAMESPACE phrase is missing at level 01 in an IDENTIFIED clause, this is equivalent to the NAMESPACE NULL phrase.

The prefixes which can be used as abbreviations in XML notation have nothing to do with the NAMESPACE in COBOL. They are not visible in COBOL.

> The COBOL statements always use the 'complete' value from the xmlns specification in the XML document as the value for a namespace, never the prefix.

The distinction on the XML side between standard namespaces and namespaces specified with prefixes is irrelevant on the COBOL side because no prefixes exist there.

The possible combinations of the BY and USING phrases in the case of local names and namespaces are subject to particular restrictions. Here it is irrelevant whether the NAME-SPACE phrase is specified or implied. On the one hand the restrictions affect the phrases within a single IDENTIFIED clause. On the other hand they affect the combination of two or more data description entries with IDENTIFIED clauses of the same type (ELEMENT or ATTRIBUTE), provided they are directly subordinate to the same data item in the COBOL structure.

The table below provides an overview of the permitted combinations of BY and USING phrases in an IDENTIFIED clause with any specification of the type (ELEMENT or ATTRIBUTE):

Local name	NAMESPACE		
	NULL	IS	USING
BY	X	X	–
USING	X	–	X

The table below provides an overview of the permitted combinations of BY and USING phrases in two data description entries with IDENTIFIED clauses of the same type. In other words the prohibitions only apply within the set of IDENTIFIED clauses which all specify the ELEMENT phrase or all specify the ATTRIBUTE phrase. There are no restrictions for mixtures. It is therefore permissible to subordinate two data items with IDENTIFIED **BY** ... ATTRIBUTE and IDENTIFIED **USING**...ELEMENTE directly to one data item.

(1) IDENTIFIED clause is already invalid

(2) Invalid because of combination of BY and USING

(3) More than one IDENTIFIED clause with USING invalid

NAMESPACE Local name	NAMESPACE Local name	NULL BY	NULL USING	IS BY	IS USING	USING BY	USING USING
NULL	BY	X	– (2)	X	– (1)	– (1)	– (2)
NULL	USING	– (2)	– (3)	– (2)	– (1)	– (1)	– (3)
IS	BY	X	– (2)	X	– (1)	– (1)	– (2)
IS	USING	– (1)	– (1)	– (1)	– (1)	– (1)	– (1)
USING	BY	– (1)	– (1)	– (1)	– (1)	– (1)	– (1)
USING	USING	– (2)	– (3)	– (2)	– (1)	– (1)	– (3)

The **procedure** for assignment described above is not affected by namespaces. However, the namespaces must be taken into consideration **as prerequisites for assigning a single node**.

The following prerequisites therefore apply for assigning a node from the tree to a data item:

- The same type of node in the tree and in the IDENTIFIED clause of the data item (i.e. both are element nodes or attribute nodes)
- The **local name** specified in the IDENTIFIED clause must be the same as the local name of the node in the tree:
  - In the case of a BY phrase the names must be identical, except for trailing blanks.
  - In the case of a USING phrase any name from the tree is regarded as matching.

- The **namespace** specified in the IDENTIFIED clause must match the namespace of the node in the tree (here it does not matter whether the NAMESPACE phrase was specified for the data item or was inherited; the same applies for the namespaces in the XML tree):
  - In the case of an IS phrase the namespaces must be identical, except for trailing blanks.
  - In the case of a USING phrase any namespace from the tree is regarded as matching, also the empty namespace.
  - If NULL is specified, the node in the tree must have an empty namespace.

Example 12-55 Predefined NAMESPACE

XML document	Pos	COBOL data structure	**OPEN DOCUMENT xml-fil**	**READ xml-fil ELEMENT x**
<doc xmlns="http://www.abc" xmlns:r="http://www.efg"> <a>2 3 <r:c>4</r:c> </doc>	1 2 3 4	**FD xml-fil** **01 x IDENTIFIED BY "doc"** **NAMESPACE "http://www.abc".** **02 y IDENTIFIED BY "a"** **NAMESPACE "http://www.abc".** **03 y-value PIC 9.** **02 z IDENTIFIED BY "c".** **03 z-value PIC 9.** **02 u IDENTIFIED BY "b"** **NAMESPACE NULL.** **03 u-value PIC 9.**	**1(o)** **2(o)** **inv** **inv**	**1(r)** **2(r)** **2** **inv** **inv**
		FILE STATUS	**00**	**08**

Comments:

- Information on the presentation of this example is provided in section "Statements for XML processing".
- Node 1 and data item x specify the same namespace. As the element names are also identical, node 1 can be assigned.
- The NAMESPACE phrase for data item y is actually superfluous because it is identical to that of the superordinate data item x. If it did not have its own phrase, y would inherit the namespace of x.
- Node 2 inherits the standard namespace of node 1 and consequently has the same namespace as data item y. As the element names are also identical, node 2 can be assigned.
- Node 3 and data item u do have the same local name (b), but node 3 inherits the namespace http://www.abc, while data item u requests the empty namespace. Node 3 can therefore not be assigned.
- Node 4 and data item z also have the same local name (c). However, node 4 specifies the namespace http://www.efg, while data item z, as an element without its own NAMESPACE phrase, inherits the namespace http://abc of the superordinate data item x. As the namespaces are different, node 4 cannot be assigned.
- When reading via x, nodes 3 and 4 can never be assigned and cause I-O status 08.

Example 12-56 Arbitrary NAMESPACE

XML document	Pos	COBOL data structure	**OPEN DOCUMENT xml-fil**	**READ xml-fil ELEMENT x**	**READ xml-fil ELEMENT y**

<doc xmlns:r= "http://www.efg"> <a>2 <r:c>4</r:c> </doc>	1 2 3	**FD xml-fil** **01 x IDENTIFIED** **USING x-name** **NAMESPACE** **USING x-nsp.** **02 x-name PIC XXX.** **02 x-nsp PIC X(14).** **02 y IDENTIFIED** **USING y-name** **NAMESPACE** **USING y-nsp.** **03 y-name PIC XXX.** **03 y-nsp PIC X(14).** **03 y-value PIC 9.**	**1(o)** **2(o)**	**1(r)** **doc** 'BLANK' (x14) [1] **2(r)** **a**'BLANK''BLANK' 'BLANK' (x14) [1] **2**	*1(r)* *doc* 'BLANK' (x14) [1] **3(r)** **c**'BLANK''BLANK' **http://www.efg** **4**
		FILE STATUS	**00**	**00**	**00**

[1] The symbol is repeated 14 times.

Comments:

- Information on the presentation of this example is provided in section "Statements for XML processing".
- Neither the element name nor the namespace stipulate requirements. Consequently all nodes can in principle be assigned. For this reason the first READ statement (unlike in Example 12-55) supplies I-O status 00.
- Node 1 has no (i.e. an empty) namespace: for the transfer to data item x-nsp it acts like a sending item with the size zero. Hence the blanks in x-nsp. The same applies for node 2 and y-nsp.
- The second READ statement reads node 3: only the local name 'c' is supplied in y-name as the element name. The complete namespace is supplied from the associated xmlns phrase (in the case of node 1) as namespace. Prefix 'r' which is used as the abbreviation is not visible in COBOL.

12.10.2 Event-oriented processing

The new **XML PARSE** statement parses an XML document, the document being processed sequentially as a data stream. When the parser detects syntax units in the document, such as tags for the start and end of an element, attributes, values, etc., it initiates an event specifically for this purpose. The COBOL program provides a routine for handling such events. The data which characterizes the event is available there in special registers. Here it is important that control is transferred to the COBOL program in the handling routine **multiple times** – whenever one of the events has occurred – while the XML statement executes, but control must also be returned to the parser. Only when the entire document has been processed in this way or the user has decided not to process it further is the XML statement terminated.

12.10.2.1 XML statement

The new **XML** statement is used to read XML documents. The following two phrases must be entered to permit this; the third is optional:

1. What is to be done? (mandatory)

   PARSE identifier-1

   The XML document contained in data item identifier-1 is to be parsed.

2. Who should process the results of parsing? (mandatory)

   PROCESSING PROCEDURE IS procedure-name-1 [{ THRU | THROUGH } procedure-name-2]

   Each time an 'event' is detected by the parser while the XML statement is being executed, control is transferred to processing procedure procedure-name-1. If the last statement has been executed by procedure-name-1 or procedure-name-2, control automatically returns to the XML statement which is still active.

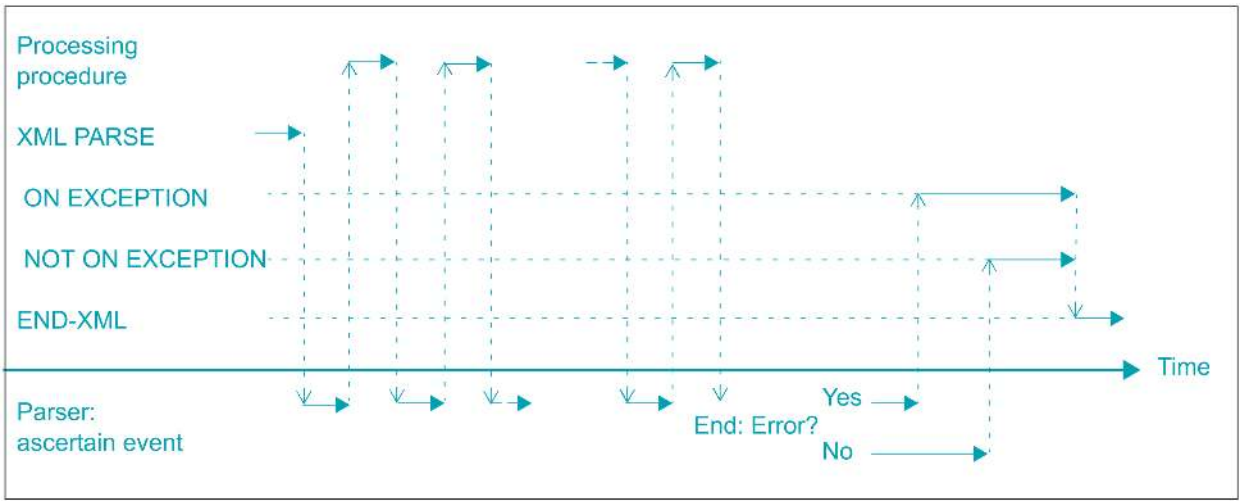
3. How should the statement be terminated? (optional)

   [ON EXCEPTION imperative-statement-1]

   [NOT ON EXCEPTION imperative-statement-2]

   identifies additional statements which are possibly to be executed if parsing ends abnormally or normally.

The activities of XML statement, processing procedure for events and XML parse are restricted in time as follows:



In summary, the following must be taken into consideration in the case of the XML statement:

- The XML document – precisely one – is contained in the specified data item.
- The repeated switch between XML statement and processing procedure continues until one of the following situations occurs:
  - The entire XML document has been processed.
  - The processing procedure indicates that processing is to be terminated immediately.
  - An error is reported.
- For documents which are **not well-formed** the sequential processing method means that the parser might possibly have initiated events before it finds the faulty place in the document.
- The precise description of the event and the data associated with it are available to the processing procedure in special registers.
- The complete document is parsed and processed by executing a **single** statement.

- Existing [NOT] EXCEPTION phrases are not executed after each event, but just once after the entire XML statement has ended.

12.10.2.2 Special registers

The table below provides an overview of the names, contents and attributes of the special registers:

Special register	Content	USAGE	Length
XML-EVENT	Name of the event	DISPLAY	Fixed; 30 characters
XML-CODE	Error code	COMP-5	Fixed; 9 digits with sign
XML-TEXT	Text connected to the event	DISPLAY	Variable; at most the length of the data item specified in the XML statement
XML-NTEXT		NATIONAL	
XML-NAMESPACE	Namespace of an element, attribute or declaration	DISPLAY	
XML-NNAMESPACE		NATIONAL	
XML-NAMESPACE-PREFIX	Prefix for abbreviation of a namespace	DISPLAY	
XML-NNAMESPACE-PREFIX		NATIONAL	

XML-CODE is used for communication between the application and parser and may consequently also be modified by the application. The other special registers are supplied with values by the parser. The application should therefore only read them; writing leads to undefined behavior. The current length of the data provided in variable-length special registers must, if necessary, be determined using FUNCTION LENGTH.

When an event occurs, the parser supplies only some of the special registers with values depending on the class of the data item containing the XML document. Special registers which are not supplied with values always have the current length 0.

	identifier-1	
	DISPLAY	NATIONAL
XML-EVENT	X	X
XML-CODE	X	X
XML-TEXT	X	–
XML-NTEXT	–[1]	X
XML-NAMESPACE	X	–
XML-NNAMESPACE	–	X
XML-NAMESPACE-PREFIX	X	–

XML-NNAMESPACE-PREFIX	–	X

[1] For the exception see the CONTENT-NATIONAL-CHARACTER event in section Processing procedure

The special registers are available in every program which was compiled using the relevant option. However, plausible values are only to be expected in the variable-length registers (i.e. lengths > 0) while an XML PARSE statement is active. The last value is still available in XML-CODE and XML-EVENT even after the statement has terminated.

12.10.2.3 Processing procedure

During execution a processing procedure behaves as if it were activated by the XML statement via PERFORM, in a similar manner to an INPUT/OUTPUT PROCEDURE with SORT, see also section "SORT statement". The same rules and restrictions therefore apply for specification in the XML statement as for a PERFORM statement.

Events

Within the processing procedure the application can perform further processing operations which are of interest. Almost all events make specific data available in special register XML-TEXT or XML-NTEXT for this purpose. The nsp column in the table below indicates events for which the namespace special registers are also supplied with values. These registers are XML-NAMESPACE and XML-NAMESPACE-PREFIX or XML-NNAMESPACE and XML-NNAMESPACE-PREFIX. If no namespace or the empty namespace is specified, the current length of the special registers is 0; in the case of a standard namespace the current length of the prefix special register is 0.

The single quotes enclosing various strings in the table below are included to highlight the strings. They are not part of the strings.

Name of the event (in XML-EVENT)	**Content of XML-TEXT or XML-NTEXT**	**nsp**
ATTRIBUTE-CHARACTERS	Value of the attribute **without** the opening and closing literal identifiers.	–
ATTRIBUTE-NAME	Local name of the attribute.	X
COMMENT	Content of the comment **without** the '<!--' string at the start and the '-->' string at the end.	–
CONTENT-CHARACTER	A single character which corresponds to a predefined entity reference.	–
CONTENT-CHARACTERS	Substring between an element's start and end tags. This is only the entire value if it contains no other elements or entity references.	–
CONTENT-NATIONAL-CHARACTER	Only in XML-NTEXT for documents which are made available in an alphanumeric data item: a single national character which corresponds to a numeric entity that cannot be represented as an alphanumeric character. With this event XML-TEXT always has 0 as its current length.	–
DEFAULTED-ATTRIBUTE-NAME	Local name of an attribute which is not contained in the XML document but is declared with a default value in the DTD (Document Type Definition) used.	X
DOCUMENT-TYPE-DECLARATION	The entire document type definition, **inclu ding** the '<!DOCTYPE' string at the start and '>' at the end.	–
ENCODING-DECLARATION	Encoding name (as expected by XHCS) which is implied for interpreting the document. This is generally not the name specified in the encoding declaration of the XML	–

	document. Rather, it can be a different encoding if, for example, the XML document was transferred and, in the process, converted by means of File Transfer without at the same time adjusting the specification in the encoding declaration.	
END-OF-CDATA-SECTION	The ']]>' string.	–
END-OF-DOCUMENT	Empty, i.e. current length = 0	–
END-OF-ELEMENT	Local name of the element	X
EXCEPTION	Start of the document, up to and including the error location.	–
NAMESPACE-DECLARATION	Empty, i.e. current length = 0.	X
PROCESSING-INSTRUCTION-DATA	Content of the processing instruction after the name **without** the '?>' string at the end. Leading white spaces are not part of this, but trailing ones are.	–
PROCESSING-INSTRUCTION-TARGET	Name of the processing instruction (**without** the '?>' string at the start).	–
STANDALONE-DECLARATION	The 'yes' or 'no' string from the text declaration.	–
START-OF-CDATA-SECTION	The '<![CDATA[' string.	–
START-OF-DOCUMENT	The complete document.	–
START-OF-ELEMENT	Local name of the element.	X
VERSION-INFORMATION	Value from the version information of the text declaration without the opening and closing literal identifiers.	–

Error handling

The parser uses the 'EXCEPTION' event to notify the processing procedure of errors it detects. The special register XML-CODE then has a value not equal to 0 which describes the error more exactly. For all other events this special register has the value 0.

The content of the special register XML-CODE when the processing procedure is exited controls how the XML statement is continued. The processing procedure signals how the statement is to continue by setting corresponding values:

-1	Terminates the XML statement with 'error'. This is possible with every event, even if the EXCEPTION event did not occur beforehand.
0	Continuation of the XML statement. If an error has already been reported, continuation is only possible in the event of 'minor' errors. However, it is restricted to searching for further errors. The XML statement is nevertheless always aborted in the case of 'serious' errors.
Other values	The further behavior is undefined.

Depending on how the XML statement was terminated, existing EXCEPTION and NOT EXCEPTION clauses are executed after it:

NOT EXCEPTION	In the case of normal termination.
EXCEPTION	I the case of abnormal termination because of errors. This includes active termination by setting the special register XML-CODE in the processing procedure.

Example 12-57 Events and content of the special registers

```
DATA DIVISION.
WORKING-STORAGE SECTION.
...
01 document.
  02     VALUE "<?xml version='1.1'?>".
  02     VALUE "<doc xmlns:r='http://www.efg'>".
  02     VALUE "111".
  02     VALUE "<a att='XX'/>".
  02     VALUE "<![CDATA[tse-da-da segdschn]]".
  02     VALUE "222".
  02     VALUE "<r:b>".
  02     VALUE "yyyyy ".
  02     VALUE "</r:b>".
  02     VALUE "</doc>".
```

Parsing of the document by the XML statement results in the following series of events and contents of the special registers

XML-EVENT	XML-TEXT	XML-NAME-SPACE	XML-NAME-SPACE-PREFIX
VERSION-INFORMATION	1.1		
START-OF-DOCUMENT	<?xml version='1.1'?> <doc xmlns:r= 'http://www.efg'>111 <a att='XX'/><![CDATA [tse-da-da segdschn]]> 222		

	<r:b>yyyyy'BLANK''BLANK''BLANK' /r:b> </doc>		
START-OF-ELEMENT	doc		
NAMESPACE-DECLARATION		http://www.efg	r
CONTENT-CHARACTERS	111		
START-OF-ELEMENT	a		
ATTRIBUTE-NAME	att		
ATTRIBUTE-CHARACTERS	XX		
END-OF-ELEMENT	a		
START-OF-CDATA-SECTION	<![CDATA[		
CONTENT-CHARACTERS	tse-da-da segdschn		
END-OF-CDATA-SECTION	]]>		
CONTENT-CHARACTERS	222		
START-OF-ELEMENT	b	http://www.efg	r
CONTENT-CHARACTERS	yyyyy'BLANK''BLANK''BLANK'		
END-OF-ELEMENT	b	http://www.efg	r
END-OF-ELEMENT	doc		
END-OF-DOCUMENT			

Comments:

- The value (111222) of the element 'doc' is split into two parts by the inserted element 'a'. Because the XML document is processed sequentially, the parser supplies the segments separately – interrupted by the events for element 'a' – each with the CONTENT-CHARACTERS event.
- An empty entry indicates that the special register’s current length in the event is 0.

Example 12-58 XML statement and processing procedure

```
PROCEDURE DIVISION.
...
  XML PARSE document
    PROCESSING PROCEDURE IS event
    ON EXCEPTION ...
    NOT ON EXCEPTION ...
  END-XML.
...
```

```
event.
  EVALUATE XML-EVENT
   WHEN "VERSION-INFORMATION"
     IF XML-TEXT NOT = "1.0"
        DISPLAY "XML Version not 1.0"
        MOVE -1 TO XML-CODE
     END-IF
   WHEN "CONTENT-CHARACTERS"
     ...
   WHEN "START-OF-ELEMENT"
     IF FUNCTION LENGTH(XML-TEXT) > 31
        DISPLAY "Element name too long"
        MOVE -1 to XML-CODE
     ELSE
        MOVE XML-TEXT TO ...
     END-IF
   WHEN "END-OF-ELEMENT"
   ...
   WHEN OTHER
     CONTINUE
END-EVALUATE.
```

Comments:

- The program wants to process only XML documents of Version 1.0 and terminate processing for all other versions: if the special register XML-CODE is supplied with the value -1 in the processing procedure, this signals the wish for the XML statement to be aborted immediately.
- -1 in the special register XML-CODE when the XML statement is returned to means abnormal termination and continuation with the statements in the ON EXCEPTION phrase.
- The program only processes elements and their values. The processing procedure 'swallows' other events without doing anything to achieve this (WHEN OTHER).
- The program wants to accept only element names with a maximum of 31 characters. If longer element names occur in the document, the XML statement is terminated immediately. The length of an element name made available in the special register XML-TEXT corresponds to the current length of the special register: it is supplied by FUNCTION LENGTH.
- Even if individual special registers cannot be defined with conventional COBOL language elements, they may be used in COBOL statement. They behave like alphanumeric or national groups of variable length, e.g. as a sending item in the MOVE statement.

12.10.3 XML Common Syntactic Constructs

Some of the constructs available in an XML document are no longer visible at the interface in COBOL, and in some cases the visibility depends on the type of processing.

Construct / Property	Structure-oriented	Event-oriented
Numeric character reference	Replaced by the corresponding character	
	If no representation is available in the target encoding, the replacement character is supplied.	If no representation is available in the target encoding, the national represen-tation is supplied, i. e. the CONTENT-NATIONAL-CHARACTER event.
Predefined entity	Replaced by the corresponding character	
General entity	Replaced according to the available definition	
Internal Document Type Definition (DTD)	Used to define entities	
	Not used to validate the document	
	Not supplied[1]	Supplied in one piece
External DTD	Used to define entities	
	Not used to validate the document	
	Not supplied	
Defaulted attribute from DTD	Supplied; cannot be distinguished from the attributes which are specified directly in the XML document	Supplied; can be distinguished from the attributes which are specified directly in the XML (event = DEFAULTED-ATTRIBUTE-NAME)
Schema	Not supported	
Text declaration	Interpreted	
	Not supplied[2]	Supplied in three parts: first VERSION-INFORMATION, then ENCODING-DECLARATION, finally STANDALONE-DECLARATION
Processing instructions	Not supplied[1]	Supplied in two parts: first PROCESSING-INSTRUCTION-TARGET, then PROCESSING-INSTRUCTION-DATA
Comments	Not supplied[1]	Supplied in one piece
CDATA section	Component part of the values supplied; no longer recognizable	

		Supplied in three parts: first START-OF-CDATA-SECTION, then CONTENT-CHARACTERS, finally END-OF-CDATA-SECTION
Namespace declaration	Not supplied; however, interpreted when used in element or attribute names	Supplied in one piece
Data im mixed content	Combined to form a single value	Supplied in parts; fragmentation can also be caused by entity references.
Handling end of line, file record structure, etc.	Suppressed or forwarded in standard manner according to options (concerns external entities and documents in files)	Suppressed or forwarded in standard manner according to options (concerns external entities only; the document itself is already contained in memory)
Well-formed structure of the document	Irregularities determined ahead of processing	Irregularities detected only in the course of processing
Determining the document's encoding	From the definition of the memory area; If the memory area is alphanumeric, the exact EBCDIC variant is determined from the content, i. e. any encoding declaration which exists is determined.	
	From file attribute and file content	

[1] No language elements provided to describe it in COBOL
[2] Language elements provided in order to supply the encoding name from the encoding declaration, but currently not supported

12.11 Debugging

As debugging aids, the compiler provides the user with debugging lines and a compile time switch for debugging lines.

Debugging lines

A debugging line is any line with a ’D’ in column 7 (the indicator area) of the line.Any debugging line that consists solely of spaces from margin A to margin R is considered the same as a blank line.
The content of a debugging line must be such that a syntactically correct program is formed with or without the debugging lines being considered as comment lines.
After all COPY statements have been processed, a debugging line will be treated as program text if it is preceded by a WITH DEBUGGING MODE clause. Otherwise, it is treated as a comment line. Successive debugging lines are allowed.

> ¡ Debugging lines are only permitted in fixed format.

Compile time switch

The WITH DEBUGGING MODE clause specified in the SOURCE-COMPUTER paragraph serves as a compile time switch for the debugging lines.
If the WITH DEBUGGING MODE clause is specified, all following debugging lines are compiled as specified in the rules described above. If it is not specified, the debugging lines are treated as comment lines by the compiler.

13 Segmentation

13.1 General description

The Segmentation feature allows the programmer, at compile time, to specify program overlay requirements. Only the Procedure Division may be segmented. The Procedure and Environment Divisions are therefore provided to define the segmentation requirements for the program.

Note

Segmentation is superfluous when generating shareable code. Shareable code can be generated simply by means of a control statement to the compiler (see "COBOL2000 User Guide" [1]).

13.1.1 Organization

Although it is not mandatory, the Procedure Division for a compilation unit is usually written as several consecutive sections, each of which is composed of a series of statements which, taken together, perform a particular function. When segmentation is used, the entire Procedure Division must be in sections. In addition, each section must be classified as belonging either to the fixed portion of the program or to one of the independent segments of the program. On the other hand, segmentation in no way affects the need for qualification of procedure-names to ensure uniqueness.

13.1.2 Fixed portion of the program

The fixed portion of the program is defined as that portion that is logically treated as if it were always resident in memory. This portion of the program is composed of two types of computer storage segments: permanent segments and fixed overlayable segments.

a. A permanent segment is a segment in the fixed portion that cannot be overlaid by another part of the program.
b. A fixed overlayable segment is a segment in the fixed portion which can overlay, or be overlaid by, either another fixed overlayable segment or an independent segment. From the execution point of view, however, it is considered to be resident in internal storage. When a fixed overlayable segment is called, it is always made available in the state in which it was last used; GO TO statement branch addresses modified by ALTER are not returned to their initial states.

The number of the permanent segments in the fixed portion can be varied through the use of the SEGMENT-LIMIT clause.

13.1.3 Independent segments

An independent segment is defined as that part of the program that can overlay other segments and can be overlaid by an overlayable fixed segment or by another independent segment. If procedures contained within an independent segment are referenced by a PERFORM or GO TO statement from outside that independent segment, the program is always provided with an independent segment which is in its initial state. GO TO statement branch addresses that have been modified by ALTER are returned to their initial state.

13.2 General rules for segmentation

1. If the sections that belong to a given segment, i.e. sections which have the same segment-number, are scattered through the compilation unit, they must be reordered by the compiler. However, the compiler will maintain the logic flow of the compilation unit. The compiler will also insert statements necessary to load and/or initialize a segment as necessary. Control may be transferred within a compilation unit to any paragraph in a section; that is, it is not mandatory to transfer control to the beginning of a section.
2. Only the fixed segments remain in internal storage throughout program execution. Both the overlayable fixed segments and the independent segments may overlay one another.
3. The following criteria should be noted when classifying segments:

   Logical requirements:

   Sections that must be available for reference at all times or which are referred to frequently should be classified as one of the permanent segments; sections that are less frequently used should be classified as one of the overlayable fixed segments or one of the independent segments, in accordance with the logic requirements of the program flow.

   Relationship to other sections:

   Sections that frequently communicate with one another should be given equal segment-numbers. All sections with the same segment-number each constitute a program segment.
4. When segmentation is used, the following rules apply to the ALTER, PERFORM, MERGE and SORT statements as well as the called programs:

   ALTER statement:

   a. A GO TO statement in a section whose segment-number is greater than or equal to 50 must not be referenced by an ALTER statement in a section with a different segment-number.

   b. A GO TO statement in a section whose segment-number is less than 50 may be referenced by an ALTER statement in any section, even if the GO TO statement referenced by the ALTER statement belongs to a program segment which has not yet been called for execution.

   PERFORM statement:

   a. A PERFORM statement that appears in a section whose segment-number is less than the segment-number supplied in the SEGMENT-LIMIT clause can have within its range only the following:

      - Sections each having a segment-number less than 50.
      - Sections wholly contained in a single segment whose segment number is greater than 49.

      However, the compiler permits the PERFORM statement to reference, within its range, sections having any section number.

   b. A PERFORM statement appearing in a section whose segment-number is equal to or greater than the segment number specified in the SEGMENT-LIMIT clause can have within its range only one of the following:

      - Sections each having the same segment-number as that of the section containing the PERFORM statement.
      - Sections each having a segment-number less than that specified in the SEGMENT-LIMIT clause.

   However, the compiler permits the PERFORM statement to reference, within its range, sections having any section number.

SORT/MERGE statement:

a. When using a SORT or MERGE statement within a section which is not an independent segment, the input procedures or output procedures referenced by that SORT or MERGE statement:
   - must be contained wholly within independent segments,
   - or must be contained wholly within a single independent segment.

b. When using a SORT or MERGE statement within an independent segment, the input procedures or output procedures referenced by that SORT or MERGE statement:
   - must be wholly contained within independent segments, or
   - must be wholly contained within the same independent segment as the SORT or MERGE statement.

These restrictions do not apply to the compiler discussed in this manual.

Called programs

A program that was called by a CALL statement may have its entry points only in the permanent segment.

13.3 Language elements

There are two COBOL language elements which are directly related to segmentation. Both language elements are optional and should only be specified when segmentation is used:

SEGMENT-LIMIT clause	specified in the OBJECT-COMPUTER paragraph of the Environment Division.
Segment number	specified in the appropriate section header in the Procedure Division.

13.3.1 Language elements of the Environment Division

13.3.1.1 SEGMENT-LIMIT clause

Function

The SEGMENT-LIMIT clause enables the user to vary the number of permanent and overlayable fixed segments in his program, while still retaining the logical properties of fixed portion segments (segment-number 0 through 49) and keeping constant the total number of segments in the fixed portion.

Format

```
SEGMENT-LIMIT IS segment-number
```

Syntax rules

1. The SEGMENT-LIMIT clause is supplied in the OBJECT-COMPUTER paragraph. This is an optional clause.
2. segment-number must be an unsigned integer that ranges in value from 1 through 50.
3. When the SEGMENT-LIMIT clause is specified, only those segments having segment-numbers up to, but not including, the segment-number designated in the SEGMENT-LIMIT clause are considered as permanent.
4. Those segments having segment-numbers from the segment limit through 49 are considered to be overlayable fixed segments.
5. When the SEGMENT-LIMIT clause is omitted, all segments having segment numbers from 0 through 49 are considered as permanent segments of the program.

General rule

1. Ideally, all program segments having segment-numbers ranging from 0 through 49 are treated as permanent segments. However, when insufficient storage is available to contain all permanent segments plus the largest overlayable segment, the number of permanent segments must be reduced. This may be done by subsequently including a SEGMENT-LIMIT clause with appropriate entries without otherwise changing the program.

Example 13-1

```
SEGMENT-LIMIT IS 40
```

This clause indicates that all segments having segment-numbers from 0 through 39 are considered to be permanent segments of the program. Segments having segment-numbers from 40 through 49 are overlayable fixed segments.

13.3.2 Language elements of the Procedure Division

13.3.2.1 Segment number

Function

Section classification is achieved by a system of segment-numbers. The segment-number is included in the section heading.

Format

```
section-name SECTION [segment-number].
```

Syntax rules

1. Section-name identifies the section.
2. Segment-number indicates whether the chapter belongs to a permanent, overlayable fixed, or independent segment.
3. segment-number must be an unsigned integer ranging in value from 0 through 99.
4. All sections that have the same segment-number constitute a program segment with that number.
5. Segments with segment-numbers 0 through 49 belong to the fixed portion of the object program. The SEGMENT-LIMIT clause indicates which of these segments are treated as permanent and which as overlayable fixed segments.
6. Segments with segment-numbers 50 through 99 are independent segments.
7. If the segment-number is omitted from the section heading the segment number is assumed to be 0.
8. Sections in the DECLARATIVES subdivision of the Procedure Division must not contain segment-numbers in their section headings. They are treated as permanent segments whose segment-number is 0.

General rule

1. When a procedure-name in an independent segment is referred to by a PERFORM or GO TO statement contained in a segment with a different segment-number, the segment referred to is made available in its initial state for each execution of the PERFORM or GO TO statement. If the PERFORM statement is repeated, the initial state is restored only for the first execution of the PERFORM statement.

14 Summary of obsolete elements

This chapter contains a summary of the language elements which should not be used in new programs because they will not be supported by future COBOL standards. It is advisable to remove these obsolete elements from old programs.

- MEMORY SIZE clause:

  Only used for documentation (see the section "OBJECT-COMPUTER paragraph").
- MULTIPLE FILE TYPE clause:

  Required when more than one file shares a tape reel (see the section "MULTIPLE FILE TAPE clause").
- RERUN clause:

  Indicates when and where the restart points are to be issued (see the section "RERUN clause").
- DATA RECORDS clause:

  Only used for documentation and specifies names of the records in a file (see the section "DATA RECORDS clause").
- LABEL RECORDS clause:

  Used to specify the names and values of the labels contained in a file (see the section "LABEL RECORDS clause").
- VALUE OF clause:

  Defines the description of data items in a label record (see the section "VALUE OF clause").
- ALTER statement:

  Used to modify one or more GO TO statements (see the section "ALTER statement").
- REVERSED phrase:

  Used to read in the records of a file in reversed order (see the section "OPEN statement").
- STOP literal phrase:

  Outputs the literal on the associated main console or subconsole so that only the system operator can resume the program (see the section "STOP statement").
- Segmentation:

  Segmentation in the PROCEDURE DIVISION to permit program overlay requirements to be specified at compile time (see the chapter "Segmentation").
  - SEGMENT-LIMIT clause:

    Permits the number of permanent and overlayable fixed segments in the program to be varied.
  - Segment Number:

    Permits segment classification by a system of segment numbers.

15 Related publications

The manuals are available as online manuals, see http://manuals.ts.fujitsu.com, or in printed form which must be paid and ordered separately at http://manualshop.ts.fujitsu.com.

[1] **COBOL2000** (BS2000/OSD))
COBOL Compiler
User's Guide

[2] **CRTE (BS2000/OSD)**
Common Runtime Environment
User Guide

[3] **AID** (BS2000)
Advanced Interactive Debugger
Core Manual
User Guide

[4] **AID** (BS2000)
Advanced Interactive Debugger
Debugging of COBOL Programs
User Guide

[5] **AID** (BS2000/OSD)
Debugging on Machine Code Level
User Guide

[6] **UDS/SQL** (BS2000/OSD)
Application Programming
User Guide

[7] **openUTM** (BS2000/OSD, UNIX, Windows)
Programming Applications with KDCS for COBOL, C and C++
User Guide

[8] **SORT** (BS2000/OSD)
User Guide

[9] **BS2000/OSD-BC**
Introductory Guide to DMS
User Guide

[10] **EDT** (BS2000/OSD)
Statements
User Guide

[11] **COBOL85** (BS2000)
COBOL Compiler
Ready Reference

[12] **BS2000/OSD**
Softbooks English
CD-ROM