5256

Burroughs

B 7000/B 6000 Series

PL/I

REFERENCE MANUAL

PRICED ITEM

Printed in U.S. America

May 1977

5001520

Burroughs

Hans Vlems

B 7000/B 6000 Series

PL/I

REFERENCE MANUAL

Copyright © 1977, Burroughs Corporation, Detroit, Michigan 48232

PRICED ITEM

Printed in U.S. America

May 1977

5001530

Burroughs believes that the information described in this manual is accurate and reliable, and much care has been taken in its preparation. However, no responsibility, financial or otherwise, is accepted for any consequences arising out of the use of this material. The information contained herein is subject to change. Revisions may be issued to advise of such changes and/or additions.

Correspondence regarding this document should be addressed directly to Burroughs Corporation, P.O. Box 4040, El Monte, California 91734, Attn: Publications Department, TIO–West.

Rdok 10591

Burroughs

PUBLICATION CHANGE NOTICE

PCN No.: 5001530-001 Date: March, 1981

Publication Title: B 6000/B 7000 PL/I Language Reference Manual

Other Affected Publications: None

Supersedes: N/A

Description:

Change Pages		Add Pages
TOC-i	7-45	7-21B
TOC-iii	7-53	7-46A
TOC-v	8-1	7-46C
TOC-vii	8-23	
TOC-ix	A1-11	
TOC-xi	A2-1	
7-21	A2-9	
7-33	A7-29	
7-35	A7-33	
7-37		
7-39		
7-41		
7-43		

Retain this PCN as a record of changes made to the basic publication.

The above pages covering
PCN 5001530-001

COPYRIGHT © 1981
BURROUGHS CORPORATION
Detroit, Michigan 48232

Burroughs believes that the information described in this manual is accurate and reliable, and much care has been taken in its preparation. However, no responsibility, financial or otherwise, is accepted for any consequences arising out of the use of this material. The information contained herein is subject to change. Revisions may be issued to advise of such changes and/or additions.

Correspondence regarding this document should be addressed directly to Burroughs Corporation, P.O. Box 4040, El Monte, California 91734, Attn: Publications Department, TIO–West.

Rdck 12367

Burroughs

PUBLICATION CHANGE NOTICE

PCN No.: 5001530-002 Date: September 1981

Publication Title: B 5000/B 6000/B 7000 PL/I Language Reference Manual

Other Affected Publications: None

Supersedes:

Description:

Add Pages

Index-1 thru Index-13

The above pages covering
PCN 5001530-002

COPYRIGHT © 1981
BURROUGHS CORPORATION
Detroit, Michigan 48232

"The names used in this publication are not of individuals living or otherwise. Any similarity or likeness of the names used in this publication with the names or any individual, living or otherwise, is purely coincidential and not intentional."

Burroughs believes that the information described in this document is accurate and reliable, and much care has been taken in its preparation. However, no responsibility, financial or otherwise, is accepted for any consequences arising out of use of this information..

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

Correspondence regarding this publication should be forwarded using the Remarks form at the back of the manual, or may be addressed directly to TIO West Documentation, Burroughs Corporation, 1300 John Reed Court, City of Industry, California 91745, U.S.A.

Rdok 12368

Burroughs

PUBLICATION CHANGE NOTICE

PCN No.: 5001530-003 Date: March 1, 1982

Publication Title: B 5000/B 6000/B 7000 Series PL/I Language Reference Manual (May 1977)

Other Affected Publications:

Supersedes: N/A

Description:

Incorporates the Mark 3.3 System Software release.

Replace these pages	Add pages
TOC-1	4-36-A
TOC-3	7-14-A
TOC-5	7-24-A
TOC-7	
TOC-9	
TOC-11	
4-35	
7-13	
7-23	
7-53	
7-55	
9-11	
Index-1	
Index-3	
Index-5	
Index-7	
Index-9	
Index-11	
Index-13	

Copyright © 1982, Burroughs Corporation, Detroit, Michigan 48232

TABLE OF CONTENTS

SECTION 1. INTRODUCTION 1- 1

1.1 PURPOSE 1- 1

SECTION 2. SYNTAX NOTATION. 2- 1

2.1 NOTATION VARIABLE 2- 1

2.2 NOTATION CONSTANT 2- 1

2.3 SYNTACTICAL UNIT. 2- 1

2.4 SYNTAX-LANGUAGE SYMBOLS 2- 2

2.4.1 Braces. 2- 2

2.4.2 The vertical stroke 2- 2

2.4.3 Brackets. 2- 2

2.4.4 The ellipsis. 2- 2

2.4.5 Syntax rule 2- 2

2.4.6 Blanks. 2- 3

SECTION 3. PROGRAM ELEMENTS 3- 1

3.1 60-CHARACTER SET. 3- 1

3.2 EXTRALINGUAL CHARACTER SET. 3- 2

3.3 DELIMITERS. 3- 2

3.4 OPERATORS 3- 2

3.4.1 Arithmetic Operators. 3- 2

3.4.2 Comparison Operators. 3- 2

3.4.3 Bit String Operators. 3- 2

3.4.4 The string operator is: 3- 3

3.5 PARENTHESES 3- 3

3.6 SEPARATORS AND OTHER DELIMITERS 3- 3

3.7 IDENTIFIERS 3- 3

3.8 KEYWORDS. 3- 4

3.8.1 Statement Identifiers 3- 4

3.8.2 Attributes. 3- 4

3.8.3 Separating Keywords 3- 4

3.8.4 Built-in Function Names 3- 4

3.8.5 Option Keywords 3- 5
3.8.6 Condition 3- 5
3.9 DATA ELEMENTS 3- 5
3.9.1 Problem Data. 3- 5
3.9.1.1 Arithmetic Data 3- 5
3.9.1.2 String Data 3- 6
3.9.2 Program-Control Data. 3- 6
3.9.2.1 Label Data. 3- 6
3.9.2.2 Entry-Label Data. 3- 6
3.9.2.3 Task Data 3- 6
3.9.2.4 Locator Data. 3- 6
3.9.2.5 Area Data 3- 7
3.10 CONSTANTS 3- 7
3.10.1 Real Arithmetic Constants 3- 7
3.10.1.1 Decimal-Fixed-Point Constants 3- 7
3.10.1.2 Binary Fixed-Point Constants. 3- 7
3.10.1.3 Decimal Floating-Point Constants. 3- 7
3.10.1.4 Binary Floating-Point Constants 3- 8
3.10.1.5 Precision 3- 8
3.11 STRING CONSTANTS. 3- 8
3.11.1 Character String Constants. 3- 8
3.11.2 Bit String Constants. 3- 9
3.12 NAMED CONSTANTS 3- 9
3.12.1 Statement Label Constants 3- 9
3.12.2 Entry Constants 3- 10
3.12.3 File Constants. 3- 10
3.12.4 Format Label Constants. 3- 10
3.13 VARIABLES 3- 11
3.13.1 Arithmetic Variables. 3- 11
3.13.2 Character String Variables. 3- 11
3.13.3 Bit String Variables. 3- 11
3.13.4 Statement Label Variables 3- 11
3.13.5 Entry Variables 3- 12

3.13.6 File Variables 3- 12
3.13.7 Format Label Variables 3- 13
3.14 DATA ORGANIZATION 3- 13
3.14.1 Scalar Items 3- 13
3.14.1.1 Constants 3- 13
3.14.1.2 Scalar Variables 3- 14
3.14.2 Data Aggregates 3- 14
3.14.2.1 Arrays 3- 14
3.14.2.2 Structures 3- 14
3.14.2.3 Structure Arrays 3- 15
3.14.3 Naming 3- 17
3.14.3.1 Simple Names 3- 17
3.14.3.2 Subscripted Names 3- 17
3.14.3.3 Qualified Names 3- 19
3.14.3.4 Subscripted Qualified Names 3- 21
3.15 Comments 3- 22
SECTION 4. DATA DESCRIPTION 4- 1
4.1 DECLARATIONS 4- 1
4.1.1 Explicit Declarations 4- 1
4.1.1.1 Label Prefixes 4- 1
4.1.1.2 Parameters 4- 2
4.1.2 Contextual Declarations 4- 2
4.1.3 Implicit Declarations 4- 3
4.1.4 Establishment of Declarations 4- 3
4.1.5 Scope of Declarations 4- 3
4.1.5.1 Scope of External Names 4- 4
4.1.5.2 Basic Rule of Use of Names 4- 6
4.2 CLASSIFICATION OF ATTRIBUTES 4- 7
4.3 ARITHMETIC DATA DESCRIPTIONS 4- 7
4.3.1 Mode Attribute 4- 7
4.3.2 Base Attributes 4- 8
4.3.3 Scale Attributes 4- 8
4.3.4 Precision Attribute 4- 9

4.4 STRING DATA DESCRIPTIONS 4- 10
4.4.1 String Type Attribute 4- 10
4.4.2 Length Attribute. 4- 10
4.4.3 Varying Attribute 4- 11
4.5 PICTURE DATA DESCRIPTIONS 4- 12
4.5.1 Character Picture Data Description. 4- 13
4.5.2 Numeric Picture Data Description. 4- 14
4.5.2.1 Decimal Specifiers. 4- 16
4.5.2.2 Packed Picture Classification 4- 16
4.5.2.3 Zero Suppression Characters 4- 17
4.5.2.4 Insertion Characters. 4- 18
4.5.2.5 Signs and Currency Characters 4- 19
4.5.2.6 Credit, Debit and Overpunch Characters. 4- 21
4.5.2.7 Exponent Characters 4- 22
4.5.2.8 Scale Factor. 4- 22
4.6 PROGRAM CONTROL DATA DESCRIPTION. 4- 23
4.6.1 Label Attribute 4- 23
4.6.2 Format Attribute. 4- 25
4.6.3 Locator Attributes. 4- 26
4.6.3.1 Locator Qualification 4- 27
4.6.4 In Attribute. 4- 28
4.6.5 Area Attribute. 4- 29
4.7 ENTRY DATA DESCRIPTION. 4- 30
4.7.1 Entry Attribute 4- 30
4.7.2 Generic Entry Name. 4- 32
4.7.3 Builtin Attribute 4- 34
4.7.4 Returns Attribute 4- 35
4.7.5 Parameter Attribute 4- 35
4.7.6 Library Attribute 4- 36
4.7.7 Options Attribute 4- 36
4.8 FILE DATA DESCRIPTION . . . . . 4-36- A
4.8.1 File Attribute. . . . . 4-36- B
4.8.2 Function Attribute. 4- 37

4.8.3 Transmission Attribute. 4- 37
4.8.4 Print Attribute 4- 38
4.8.5 Keyed Attribute 4- 39
4.8.6 Access Attribute. 4- 39
4.8.7 Environment Attribute 4- 40
4.9 ARRAY DATA DESCRIPTION. 4- 43
4.9.1 Dimension Attribute 4- 43
4.10 STRUCTURE DATA DESCRIPTION. 4- 44
4.10.1 Structure Attribute 4- 44
4.10.2 Member Attribute. 4- 45
4.10.3 Like Attribute. 4- 45
4.11 STORAGE CLASS 4- 46
4.12 SCOPE ATTRIBUTE 4- 49
4.13 DATA ATTRIBUTES 4- 50
4.13.1 Alignment Attribute 4- 50
4.13.2 Initial Attribute 4- 51
4.13.2.1 Initial List. 4- 51
4.13.2.2 Initial Call. 4- 53
4.13.3 Variable Attribute. 4- 54
4.13.4 Defined Attribute 4- 54
4.13.4.1 Simple Defining 4- 56
4.13.4.2 iSUB Definings. 4- 57
4.13.4.3 String overlay Defining 4- 58
4.14 DEFAULT RULES 4- 59
4.14.1 Rejection Rules 4- 60
4.14.2 Standard System Default Rules 4- 60
SECTION 5. DATA MANIPULATION. 5- 1
5.1 EXPRESSIONS 5- 1
5.1.1 Scalar Expressions. 5- 1
5.1.2 Aggregate Expressions 5- 1
5.1.2.1 Built-In Functions With Aggregate Arguments 5- 1
5.1.2.2 Value of an Aggregate Expression. 5- 2
5.2 OPERATIONS ON EXPRESSIONS 5- 2

5.2.1 Prefix Operations 5- 2
5.2.2 Arithmetic Operations 5- 2
5.2.2.1 Mixed Characteristics 5- 3
5.2.2.2 Results of Arithmetic Operations. 5- 3
5.2.2.3 Infix Operators and Aggregate Operands. 5- 5
5.2.2.4 Integer Conversion. 5- 5
5.2.2.5 Arithmetic Base and Scale Conversion. 5- 5
5.2.2.6 Floating-Point to Fixed-Point Conversion. 5- 5
5.2.3 Comparison Operations 5- 6
5.2.4 Bit String Operations 5- 7
5.2.5 Concatenation Operations. 5- 8
5.2.6 Type Conversion 5- 8
5.2.6.1 Bit String to Character String. 5- 8
5.2.6.2 Character String to Bit String. 5- 8
5.2.6.3 Character String to Arithmetic. 5- 9
5.2.6.4 Bit String to Arithmetic. 5- 9
5.2.6.5 Arithmetic to Character String. 5- 9
5.2.6.6 Arithmetic to Bit String. 5- 9
5.3 EVALUATION OF EXPRESSIONS 5- 9
5.3.1 Order of Evaluation of Scalar Expressions 5- 9
5.3.2 Order of the Evaluation of Aggregate Expressions. 5- 10
SECTION 6. PROGRAM STRUCTURE. 6- 1
6.1 Statements. 6- 1
6.1.1 Simple Statements 6- 1
6.1.2 Compound Statements 6- 1
6.1.3 Label Prefixes. 6- 2
6.2 GROUPS. 6- 2
6.3 BLOCKS. 6- 3
6.4 CONDITION PREFIXES. 6- 6
6.5 PROGRAMS. 6- 7
SECTION 7. STATEMENTS 7- 1
7.1 CLASSIFICATION OF STATEMENTS. 7- 1
7.1.1 Assignment Statement. 7- 1

7.1.2 Control Statements 7- 1
7.1.3 Program Structure Statements 7- 1
7.1.4 Data Declaration Statement 7- 1
7.1.5 Error Control and Debug Statements 7- 1
7.1.6 Input/Output Statements 7- 2
7.1.6.1 File Preparation Statements 7- 2
7.1.6.2 Record Status Statements 7- 2
7.1.6.3 Data Specification Statements 7- 2
7.1.6.4 Data Transmission Statements 7- 2
7.1.7 Storage Allocation Statements 7- 2
7.1.8 System Attribute Statements 7- 2
7.1.9 Null Statement 7- 3
7.2 SEQUENCE OF CONTROL 7- 3
7.3 LIST OF STATEMENTS BY CLASSIFICATION 7- 4
7.3.1 The Assignment Statement 7- 4
7.3.1.1 Scalar Assignments 7- 5
7.3.1.2 Aggregate Assignments 7- 6
7.3.2 Control Statements 7- 9
7.3.2.1 The CALL Statement 7- 9
7.3.2.2 The DELAY Statement 7- 10
7.3.2.3 The DO Statement 7- 10
7.3.2.4 The EXIT Statement 7- 14
7.3.2.4A The FREEZE Statement 7- 14
7.3.2.5 The GO TO Statement 7- 14
7.3.2.6 The IF Statement 7- 16
7.3.2.7 The RETURN Statement 7- 17
7.3.2.8 The SORT Statement 7- 18
7.3.2.9 The STOP Statement 7- 20
7.3.2.10 The WAIT Statement 7- 21
7.3.3 Program Structure Statements 7- 21
7.3.3.1 The BEGIN Statement 7- 21
7.3.3.2 The END Statement 7- 22
7.3.3.3 The ENTRY Statement 7- 22

7.3.3.4 The PROCEDURE Statement 7- 23
7.3.4 Data Declaration Statements . . . . . 7-24- A
7.3.4.1 The DECLARE Statement 7- 25
7.3.4.1.1 Declaration of Structures 7- 26
7.3.4.1.2 Factoring in DECLARE Statements 7- 26
7.3.4.1.3 Multiple Declarations 7- 27
7.3.4.2 DEFAULT Statement 7- 27
7.3.5 Error Control and Debug Statements. 7- 30
7.3.5.1 The DUMP Statement. 7- 30
7.3.5.2 The ON Statement. 7- 30
7.3.5.3 The REVERT Statement. 7- 31
7.3.5.4 The SIGNAL Statement. 7- 33
7.3.6 Input/Output Statements 7- 34
7.3.6.1 The CLOSE Statement 7- 34
7.3.6.2 The DELETE Statement. 7- 35
7.3.6.3 The DISPLAY Statement 7- 36
7.3.6.4 The FORMAT Statement. 7- 36
7.3.6.5 The GET Statement 7- 37
7.3.6.6 The LOCATE Statement. 7- 39
7.3.6.7 The OPEN Statement. 7- 40
7.3.6.8 The PUT Statement 7- 43
7.3.6.9 The READ Statement. 7- 44
7.3.6.10 The REWRITE Statement . . . . . 7-46- A
7.3.6.11 The WRITE Statement . . . . . 7-46- A
7.3.7 Storage Allocation Statements 7- 47
7.3.7.1 The ALLOCATE Statement. 7- 47
7.3.7.2 The FREE Statement. 7- 51
7.3.8 System Attribute Statements 7- 53
7.3.8.1 The System File Attribute Assignment Statement. 7- 53
7.3.8.2 The System File Attribute Reference Statement 7- 53
7.3.8.3 The System Task Attribute Assignment Statement. 7- 54
7.3.8.4 The System Task Attribute Reference Statement 7- 54
7.3.8.5 The System Library Attribute Assignment Statement 7- 54

7.3.8.6 The System Library Attribute Reference Statement. 7- 55
7.3.9 The Null-Statement. 7- 55
SECTION 8. INPUT/OUTPUT 8- 1
8.1 FILE ATTRIBUTES 8- 1
8.1.1 Merging of Attributes 8- 2
8.1.2 Valid Combinations for File Attributes. 8- 3
8.2 Opening a File. 8- 3
8.2.1 Explicit Opening. 8- 4
8.2.2 Implicit Opening. 8- 4
8.3 Closing a File. 8- 4
8.4 Stream Transmission 8- 4
8.4.1 Statements. 8- 4
8.4.2 Modes of Stream Transmission. 8- 6
8.4.2.1 List-Directed Transmission. 8- 6
8.4.2.2 Data-Directed Transmission. 8- 6
8.4.2.3 Edit-Directed Transmsissions. 8- 6
8.4.3 Data Lists. 8- 7
8.4.3.1 Repetitive Specification. 8- 7
8.4.3.2 Transmission of Data List Elements. 8- 9
8.4.4 List-Directed Data Specification. 8- 10
8.4.4.1 List-Directed Input Format. 8- 10
8.4.4.2 List-Directed Output Format 8- 11
8.4.4.2.1 Coded Arithmetic Data 8- 12
8.4.4.2.2 Numeric Character Data. 8- 14
8.4.4.2.3 Character String Data 8- 14
8.4.4.2.4 Bit String Data 8- 14
8.4.5 Data-Directed Data Specification. 8- 14
8.4.5.1 Data-Directed Data in the Stream. 8- 15
8.4.5.1.1 Data Directed Input 8- 15
8.4.5.1.2 Data Directed Output. 8- 17
8.4.5.2 Length of Data Fields in Data-Directed Output 8- 18
8.4.6 Edit-Directed Data Specification. 8- 19
8.4.6.1 Format Lists. 8- 20

8.4.6.2 Data Format Items 8- 20
8.4.6.2.1 Fixed-Point Format Items. 8- 21
8.4.6.2.2 Floating-Point Format Items 8- 22
8.4.6.2.3 Numeric Picture Format Item 8- 23
8.4.6.2.4 Bit String Format Items 8- 24
8.4.6.2.5 Character String Format Items 8- 24
8.4.6.3 Control Format Items. 8- 25
8.4.6.3.1 Spacing Format Item 8- 25
8.4.6.3.2 Positioning Format Items. 8- 25
8.4.6.3.3 Printing Format Items 8- 26
8.4.6.3.4 Remote Format Item. 8- 26
8.5 RECORD TRANSMISSION 8- 27
8.5.1 Record Transmission Operations. 8- 28
SECTION 9. PROCEDURES 9- 1
9.1 PARAMETERS. 9- 1
9.2 REFERENCES. 9- 2
9.2.1 Procedure References. 9- 2
9.2.1.1 Function References 9- 2
9.2.1.2 Subroutine References 9- 3
9.3 Procedure Reference Arguments 9- 3
9.3.1 Evaluation of Argument Subscripts 9- 4
9.3.2 Use of Dummy Arguments. 9- 4
9.3.3 Entry Names as Arguments. 9- 5
9.4 USE OF THE ENTRY ATTRIBUTE. 9- 7
9.5 CORRESPONDENCE OF PARAMETERS AND ARGUMENTS. 9- 8
9.6 ALLOCATION OF PARAMETERS. 9- 10
9.7 BUILT-IN FUNCTIONS. 9- 10
9.8 LIBRARIES 9- 11
9.8.1 Introduction. 9- 11
9.8.2 Creating Libraries. 9- 11
9.8.3 Referencing Libraries 9- 11
9.8.4 Changing Library Attributes 9- 11
9.8.5 Entry Points and their Parameters 9- 11

SECTION 10. DYNAMIC PROGRAM STRUCTURE 10- 1
10.1 PROLOGUES 10- 1
10.2 ACTIVATION AND TERMINATION OF BLOCKS. 10- 2
10.2.1 Dynamic Descent 10- 2
10.2.2 Dynamic Encompassing. 10- 3
10.2.3 The Environment of a Block Activation 10- 3
10.2.4 The Environment of a Label Constant 10- 4
10.3 GENERATION OF A VARIABLE. 10- 4
10.4 ALLOCATION OF DATA AND STORAGE CLASSES. 10- 5
10.4.1 Definitions and Rules 10- 5
10.4.2 Storage Classes 10- 5
10.4.2.1 The Static Storage Class. 10- 6
10.4.2.2 The Automatic Storage Class 10- 6
10.4.2.3 The Controlled Storage Class. 10- 6
10.4.2.4 The Based Storage Class 10- 8
10.5 INTERRUPT OPERATIONS. 10- 8
10.5.1 Purpose of the Condition Prefix 10- 9
10.5.2 Scope of the Condition Prefix 10- 9
10.5.3 Use of The ON Statement 10- 10
10.5.4 System Interrupt Action 10- 11
10.5.5 Use of the Revert Statement 10- 13
10.5.6 Programmer-Named Conditions 10- 14
10.5.7 Condition Built-In Functions and Pseudo Variables 10- 14
SECTION 11. COMPILE-TIME FACILITIES 11- 1
11.1 PREPROCESSOR INPUT. 11- 1
11.1.1 Effects of Compile-Time Statements. 11- 1
11.2 PREPROCESSOR OUTPUT 11- 2
11.2.1 Rescanning And Replacement of Compile-Time Identifiers. . 11- 2
11.3 COMPILE-TIME VARIABLES. 11- 3
11.4 COMPILE-TIME EXPRESSIONS. 11- 3
11.5 COMPILE-TIME PROCEDURES 11- 4
11.5.1 Declaring Compile-Time Procedures 11- 5
11.5.2 Execution of Compile-Time Procedures. 11- 5

11.6 COMPILE-TIME BUILTIN-FUNCTIONS 11- 6
11.7 COMPILE-TIME STATEMENTS 11- 6
11.7.1 Activate and Deactivate Statements 11- 6
11.7.2 Assignment Statement 11- 7
11.7.3 Declare Statement 11- 8
11.7.4 Do-Group 11- 8
11.7.5 GO TO Statement 11- 9
11.7.6 If Statement 11- 9
11.7.7 Include Statement 11- 10
11.7.8 Null Statement 11- 11
11.7.9 Put Statement 11- 11
APPENDIX 1. BUILTIN FUNCTIONS A1- 1
APPENDIX 2. CONDITIONS A2- 1
APPENDIX 3. KEYWORD ABBREVIATIONS A3- 1
APPENDIX 4. 48-CHARACTER SET A4- 1
APPENDIX 5. ERROR MESSAGES A5- 1
APPENDIX 6. CONVERSION FROM IBM A6- 1
APPENDIX 7. COMPILER CONTROL IMAGES A7- 1

SECTION 1. INTRODUCTION

1.1 PURPOSE

The purpose of this document is to provide a reference manual for the Burroughs PL/I compiler. It is intended as neither a primer nor as a tutorial document.

This reference manual is divided into the following 11 sections and 7 appendices:

Section 1, INTRODUCTION: this section explains the layout of the reference manual.

Section 2, SYNTAX NOTATION: this section explains the method of syntax representation used.

Section 3, PROGRAM ELEMENTS: this section discusses the PROGRAM, and it's structure.

Section 4, DATA DESCRIPTION: this section describes the method of data definition.

Section 5, DATA MANIPULATION: this section describes how to use and manipulate the data.

Section 6, PROGRAM STRUCTURE: this section describes how to construct a basic program from program elements.

Section 7, STATEMENTS: this section describes each statement in the PL/I language.

Section 8, INPUT/OUTPUT: this section describes data transmission statements and associated files.

Section 9, PROCEDURES, FUNCTIONS AND SUBROUTINES: this section describes the interaction between segments of a PL/I program.

Section 10, DYNAMIC PROGRAM STRUCTURE: this section discusses program control.

Section 11, COMPILE-TIME FACILITIES: this section discusses the PL/I preprocessor.

Appendix 1, BUILT-IN FUNCTIONS AND PSUEDO VARIABLES.

Appendix 2, CONDITIONS.

Appendix 3, PERMISSABLE KEYWORD ABBREVIATIONS.

Appendix 4, THE 48-CHARACTER SET.

Appendix 5, ERROR MESSAGES.

Appendix 6, IBM VS B 7000/B 6000 DATA MAPPING AND CONVERSION.

Appendix 7, COMPILER CONTROL IMAGES.

SECTION 2. SYNTAX NOTATION

Throughout this manual, whenever a PL/I statement, or some other combination of elements is discussed, the manner of writing that statement or phrase is illustrated with a uniform system of notation that may be used to describe the syntax, or construction , of any programming language. The notation provides a brief but precise explanation of the general patterns that the language permits. It does not describe the meaning of the language elements, merely their structure. It indicates the order in which the elements may (or must) appear, punctuation that is required, and options that are allowed.

The following rules explain the use of this notation for any programming language. The examples apply specifically to PL/I.

2.1 NOTATION VARIABLE

A notation variable is the name of a general class of elements in the programming language. A notation variable consists of letters and hyphens enclosed in "less than" (<) and "greater than" (>) symbols and begins with a letter. If necessary to properly indicate the beginning and end of notation variables, they are enclosed in braces.

Examples

a. <digit>. This denotes the occurrence of a digit, which may be 0 through 9 inclusive.

b. <file-name>. This denotes the occurrence of the notation variable named <file-name>. An explanation of <file-name> is given elsewhere in the standard.

c. <do-statement>. This denotes the occurrence of a DO statement.

2.2 NOTATION CONSTANT

A notation constant denotes the literal occurrence of the characters represented. A notation constant consists of any characters included in the PL/I language character set. Throughout this standard, notation constants will be represented in capital letters.

Example

DECLARE

This denotes the literal occurrence of the word DECLARE.

If necessary to properly indicate the beginning and end of notation constants, notation constants are enclosed in braces.

2.3 SYNTACTICAL UNIT

The term "syntactical unit" is defined as one of the following:

a. A single variable or constant.

b. Any collection of notation variables, notation constants, and syntax-language symbols enclosed by braces or brackets.

2.4 SYNTAX-LANGUAGE SYMBOLS

2.4.1 Braces

Braces { } are used to denote grouping.

2.4.2 The vertical stroke

The vertical stroke, | , indicates that a choice is to be made.

Example

<identifier> {FIXED | FLOAT}

2.4.3 Brackets

Brackets [ ] denote options. Anything enclosed in brackets may appear one time or may not appear at all.

Example

[VARYING]

This denotes the optional literal occurrence of the keyword VARYING.

2.4.4 The ellipsis

The ellipsis (...) denotes the occurrence of the immediately preceding syntactical unit one or more times in succession.

Example

[<digit>] ...

The notation variable, digit, may or may not occur since it is surrounded by brackets. If it does occur, it may be repeated one or more times.

2.4.5 Syntax rule

A syntax rule is the sequence of the name being defined, the definition symbol (either ::= or :¯), and the definition. Whenever the definition symbol ::= is used, blanks and comments are not explicitly shown in the syntax rule.

The notation variables that are used in this standard but not defined by a syntax rule are defined in the general text.

Example

Syntax

<scope-attribute> ::= EXTERNAL | INTERNAL

It is assumed that symbols of the syntax language can be recognized by the reader either by the unique shape (braces, vertical stroke, square brackets) or by their position within a syntax rule (definition symbol).

2.4.6 Blanks

Blanks appearing in syntax formats do not represent the character blank of the language being described. Blanks are used as delimiters of the syntax notation and to improve the readability of the formats. Any two adjacent notation variables and notation constants are separated by a blank. Blanks are optional preceding and following any of the syntax-language symbols (::=, :¯, |, [,], {,}, ...).

The character blank of the language being described is represented by the symbol <b> in the formats. However, the character blank is shown explicitly only in those formats that are indicated by the special definition symbol :¯. (See Section 2.4.5, Syntax Rules)

SECTION 3. PROGRAM ELEMENTS

The PL/I language allows the programmer to write the statements of his program in a free-field format. A statement, which is a string of characters, is always terminated by the special character semicolon. A program which is, in turn, a sequence of statements, can thus be regarded simply as a string of single characters, with no special internal grouping.

3.1 60-CHARACTER SET

The 60-character set is composed of digits, special characters, and English language alphabetic characters. There are 26 alphabetic characters, letters A through Z.

There are ten digits. Decimal digits are the digits 0 through 9. A binary digit (bit) is either a 0 or a 1.

An alphanumeric character is either an alphabetic character or a digit.

There are 24 special characters included in the PL/I language set. The names and graphics by which these special characters are represented follows:

NAME	GRAPHIC
Blank	
Equal or assignment symbol	=
Plus	+
Minus	-
Asterisk or multiply symbol	*
Slash or divide symbol	/
Left parenthesis	(
Right parenthesis	)
Comma	,
Decimal point or period	.
Single quote mark (apostrophe)	'
Percent symbol	%
Semicolon	;
Colon	:
Not symbol	¬
And symbol	&
Or symbol	|
Greater than symbol	>
Less than symbol	<
Question mark	?
Dollar	$
At sign	@
Double quote mark	"
Break character (underline)	_

Note that the quotation mark used in PL/I is the single quotation mark (also known as an apostrophe or prime).

Two consecutive special characters are sometimes used as operators (e.g., >=, denoting "greater than or equal to"; ||, denoting concatenation).

3.2 EXTRALINGUAL CHARACTER SET

Although the language character set is a fixed set defined for the language, the data character set has not been limited. Data may be represented by characters from the language set plus any other EBCDIC characters permitted by the machine configuration.

3.3 DELIMITERS

Certain single characters and certain combinations of characters are used as delimiters and fall into three classes:

Operators
Parentheses
Separators and other delimiters

3.4 OPERATORS

Operators used by the language are divided into four types:

Arithmetic operators
Comparison operators
Bit string operators
String operators

3.4.1 Arithmetic Operators

The arithmetic operators are:

+	Denoting addition or prefix plus
-	Denoting subtraction or prefix minus
*	Denoting multiplication
/	Denoting division
**	Denoting exponentiation

3.4.2 Comparison Operators

The comparison operators are:

>	Denoting greater than
¬>	Denoting not greater than
>=	Denoting greater than or equal to
=	Denoting equal to
¬=	Denoting not equal to
<=	Denoting less than or equal to
<	Denoting less than
¬<	Denoting not less than

3.4.3 Bit String Operators

¬ Denoting not
& Denoting and
| Denoting or

3.4.4 The string operator is:

|| Denoting concatenation

3.5 PARENTHESES

Parentheses are used in expressions, for enclosing lists, and for specifying information associated with various keywords.

(Left parenthesis
) Right parenthesis

3.6 SEPARATORS AND OTHER DELIMITERS

NAME	GRAPHIC	USE
Comma	,	Separates elements of a list
Semicolon	;	Terminates statements
Assignment Symbol	=	Used in assignment statement and DO statement
Colon	:	Following label and condition prefixes. Also used with dimension specifications and range specification of statement
Blank		Used as a separator
Period	.	Separates items in qualified names and used as a decimal or binary point in constants
Arrow	->	Qualifies a reference to a based variable
Percent Symbol	%	Precedes compile-time statement
Quotation Mark	'	Encloses string constants and picture specifications

Identifiers, constants (except character string constants), picture specifications, and composite operators (e.g., >=) may not contain blanks.

Identifiers, constants, or picture data descriptions may not be immediately adjacent. They must be separated by a delimiter or comment. Moreover, additional intervening blanks or comments are always permitted. Blanks are optional between keywords of the statement identifier GO TO.

Examples

```
CALLA           Is not equivalent to CALL A
A TO B BY C     Is not equivalent to ATOBBYC
AB+BC           Is equivalent to AB + BC
```

3.7 IDENTIFIERS

An identifier is a string of alphanumeric and break characters that begins with an alphabetic character, is not contained in a comment or string constant and is preceded and followed by a delimiter. The maximum length of an identifier is 63 characters.

3.8 KEYWORDS

A keyword is an identifier which is part of the language. Keywords are not reserved words. They may be classified as follows:

statement identifiers
attributes
separating keywords
built-in function names
options
conditions

3.8.1 Statement Identifiers

A statement identifier is a sequence of one or more keywords used to define the function of a statement. For additional information concerning statement identifiers, refer to Section 6.1.1, Simple Statements.

Examples

```
GO TO
DECLARE
READ
```

3.8.2 Attributes

Attributes are keywords that specify the characteristics of data, procedures, and other elements of the language.

Examples

```
FLOAT
BACKWARDS
SEQUENTIAL
```

3.8.3 Separating Keywords

There are five separating keywords, which are used to separate parts of the IF statements and DO statements. They are THEN, ELSE, BY, TO, WHILE.

3.8.4 Built-in Function Names

A built-in function name is a keyword that is the name of an algorithm provided by the language and accessible to the programmer.

Examples

DATE
EXP

3.8.5 Option Keywords

An option is a specification that may be used by the programmer to influence the execution of a statement.

Example

BY NAME

3.8.6 Condition

A condition is a keyword used in the ON, SIGNAL, and REVERT statements, and as a prefix to other statements. The programmer may specify special action on the occurrence of the condition.

Examples

OVERFLOW
ZERODIVIDE

3.9 DATA ELEMENTS

Information that is operated on in a PL/I object program during execution is called data. Each data item has a definite type and representation.

The aim of this section is to present a discussion of the various organizations that data may have, the methods by which data can be referred to, and the types of data allowed.

The types of data allowed by PL/I can be categorized as problem data and program-control data.

3.9.1 Problem Data

Problem data is any data that can be classified as type arithmetic or type string.

3.9.1.1 Arithmetic Data

An arithmetic data item is one that has a numeric value with characteristics of base, scale, mode, and precision. The characteristics of an arithmetic data item are specified by attributes declared for the item, or attributes applied by default.

Base (DECIMAL or BINARY), scale (fixed-point or floating-point), and precision have reference to internal representation of the data described and to the internal arithmetic that is used.

BASE: Arithmetic data may be specified as having either decimal or binary base.

SCALE: Arithmetic data may be specified as having either fixed-point (FIXED) or floating-point (FLOAT) scale. Fixed-point data items are rational numbers for which the decimal or binary points are specified in the user program. The decimal or binary points specification is called the scale factor.

Floating-point data items are rational numbers with the decimal or binary point specification as part of the data item. Floating-point data items consist of a fractional part and an exponent part. The exponent part specifies the decimal or binary point location.

MODE: Arithmetic data may be operated on in real mode.

PRECISION: The precision of fixed-point data (P,Q) is specified by giving the total number of binary or decimal digits, P, to be maintained and a scale factor, Q. The precision of floating-point data is specified by giving the effective number, P, of binary or decimal digits to be maintained in the fractional part.

3.9.1.2 String Data

String data can be classified as character-string or bit-string. The length of a string data item is equivalent to the number of characters (for a character-string) or the number of binary digits (for a bit-string) in the item. A string data item of length zero is known as a null string.

3.9.2 Program-Control Data

Program-control data is any data that has one of the attributes FORMAT, LABEL, ENTRY, TASK, EVENT, AREA, POINTER, OFFSET, or FILE.

3.9.2.1 Label Data

Statement-label data is used only in connection with statement labels. Statement-label data may be constants or variables, and the variables may be elements of structures or arrays.

3.9.2.2 Entry-Label Data

Entry-label data is used in conjunction with procedure invocation. Entry-label data consists of constants and variables. The variables may be elements of structures or arrays.

3.9.2.3 Task Data

A task variable is a name of a task. A task variable may be an element of an array or a member of a structure.

3.9.2.4 Locator Data

Locator data consists of POINTER variables and OFFSET variables. A pointer variable has a value that is used to identify the location of a single generation of a variable. An offset variable has a value that is used to identify the location of a based variable relative to the beginning of an area. (See Section 4.6.3, Locator Attributes.)

3.9.2.5 Area Data

An area variable represents an area of storage in which specified variables may be allocated and freed.

3.10 CONSTANTS

A constant is a data item that denotes itself (i.e., its representation is both its name and its value).

3.10.1 Real Arithmetic Constants

Syntax

```
<real-arithmetic-constant> ::=

      <decimal-fixed-point-constant>|
      <binary-fixed-point-constant> |
      <decimal-floating-point-constant> |
      <binary-floating-point-constant>
```

3.10.1.1 Decimal-Fixed-Point Constants

A decimal fixed-point constant is represented by one or more decimal digits with an optional decimal point. If a decimal point is not specified, the constant is an integer constant.

Examples

```
72.192
.308
255.
158
```

3.10.1.2 Binary Fixed-Point Constants

A binary fixed-point constant is represented by one or more binary digits with an optional binary point, followed by the letter B.

Examples

```
10.B
11011B
11.110B
.001B
```

3.10.1.3 Decimal Floating-Point Constants

A decimal floating-point constant is represented by one or more decimal digits with an optional decimal point, followed by the letter E, followed by an optionally signed exponent. The exponent is one or more decimal digits specifying an integral power of ten.

Examples

```
12.E23
317.5E-16
0.1E+3
.42E+73
32E-5
```

3.10.1.4 Binary Floating-Point Constants

A binary floating-point constant is represented by one or more binary digits with an optional binary point, followed by the letter E, followed by an optionally signed exponent, followed by the letter B. The exponent is one or more decimal points specifying an integral power of two.

Examples

```
1.1011E3B
.11011E-27B
```

3.10.1.5 Precision

For purposes of expression evaluation, an apparent precision is defined for real arithmetic constants.

Real fixed-point constants have a precision (P,Q), where P is the total number of digits in the constant and Q is the number of digits specified to the right of the decimal point. The maximum precisions allowed are: binary single precision 39; binary double precision 78; decimal single precision 11; decimal double precision 23.

The precision of a floating-point constant is (P), where P is the number of digits of the constant to the left of the E. System default and maximum precisions for decimal-floating-point and binary floating point constants are the same as for fixed point constants.

Examples

```
3.14 has precision (3,2)
0.012E5 has precision (4)
0000001B has precision (7,0)
```

3.11 STRING CONSTANTS

3.11.1 Character String Constants

A character string constant is zero or more characters in the data character set enclosed in single quotation marks ('). If it is desired to represent a quotation mark, it must appear as two immediately adjacent quotation marks. A simple character string may optionally be preceded by an integer constant in parentheses to specify repetition. If the constant specifying repetition is zero, the result is a null character string.

In a string repetition factor, blanks may optionally surround the integer constant, or they may separate the right parenthesis and leading quote.

A character string constant may contain a string of characters which syntactically constitute a comment. These characters are treated as part of the string value rather than as a comment.

Examples

```
'$ 123.45'
'JOHN JONES'
'IT S'
(3) 'TOM'
''
```

The fourth example is exactly equivalent to:

```
'TOMTOMTOM'
```

The last example, which is two single quotation marks with no intervening blank, specifies the null character string.

3.11.2 Bit String Constants

A bit string constant is zero or more binary digits enclosed in quotation marks ('), followed by the letter B. The constant may optionally be preceded by an integer constant in parentheses to specify repetition. If the constant specifying repetition is zero, the result is the null bit string.

Examples

```
'0100'B
(10)'1'B
''B
```

The second example is exactly equivalent to:

```
'1111111111'B
```

The last example specifies the null bit string.

3.12 NAMED CONSTANTS

3.12.1 Statement Label Constants

A statement label constant is an identifier that appears in the program as a statement label. It permits references to be made to statements.

Example

```
                .
                .
                .
ROUTINE1:       IF X > 5 THEN GO TO EXIT;
                .
                .
                .
     GO TO ROUTINE1;
                .
                .
                .
     EXIT: RETURN;
```

ROUTINE1 and EXIT are statement label constants.

3.12.2 Entry Constants

An entry constant is an identifier that specifies a point at which a procedure may be entered.

Example

```
        .
        .
        .
P:  PROC;
        .
        .
        .
P1: ENTRY;
        .
        .
        .
END P;
```

Both P and P1 are entry constants.

3.12.3 File Constants

A file constant is an identifier that specifies a file.

Example

```
        .
        .
        .
DCL F FILE INPUT;
        .
        .
        .
GET FILE (F) DATA;
```

F is file constant.

3.12.4 Format Label Constants

A format label constant is an identifier that specifies the representation of edit-directed data. Format label constants may only be used in I/O statements.

Example

```
COMMON: FORMAT(A(5), F(5,2), X(3), F(10,0));
```

COMMON is a format label constant.

3.13 VARIABLES

A variable is an identifier that names a data item.

3.13.1 Arithmetic Variables

Arithmetic variables are names of arithmetic data items. These names are given the characteristics (i.e., attributes) of base, scale, mode, and precision.

3.13.2 Character String Variables

Character string variables are names of character data items. Character string data consists of strings of zero or more characters in the data character set. A string may be fixed or varying in length. The actual number of characters must be specified in the declaration if it is of fixed length, or the maximum length if it is of varying length.

3.13.3 Bit String Variables

Bit string variables are names of character data items. Bit string data consist of strings of zero or more binary digits (0 and 1). A string may be fixed or varying in length. The actual length of the field must be specified in the declaration if it is of fixed length, or the maximum length if it is of varying length.

3.13.4 Statement Label Variables

A statement label variable is a variable that has as values statement label constants. These variables can be grouped into arrays, or they may be elements of structures.

Example

```
     DECLARE X LABEL;
     X = POSROUTINE;
           .
POSROUTINE:.
           .
     X = NEGROUTINE;
     GO TO X;
           .
           .
NEGROUTINE:.
```

The label variable X may have the value of either POSROUTINE or NEGROUTINE, both of which are labels in the procedure. In the above example, GO TO X transfers control to NEGROUTINE.

A statement label constant or a statement label variable is called a statement label designator.

3.13.5 Entry Variables

An entry variable is a variable that has as values entry constants.

Example

```
          .
          .
          .
     DCL P ENTRY VARIABLE;
          .
          .
          .
     PI: PROC;
          .
          .
          .
     END PI;
     P = PI;
     CALL P;
          .
          .
          .
```

P is an entry variable which is assigned the value of the entry constant PI.

3.13.6 File Variables

A file variable is a variable that has as values file constants.

Example

```
        .
        .
        .
DCL F FILE VARIABLE;
DCL FI FILE INPUT;
F = FI;
OPEN FILE (F);
        .
        .
        .
```

F is a file variable which is assigned the value of the file constant FI.

3.13.7 Format Label Variables

A format label variable is a variable that has as values format label constants.

Example

```
        .
        .
        .
DCL FV FORMAT VARIABLE;
FC  :FORMAT (A(5), X(10));
FV= FC;
        .
        .
        .
FV is a format label variable
which is assigned the value of
the format constant FC.
```

3.14 DATA ORGANIZATION

Data may be organized as scalar items (i.e., single data items) or aggregates of data items (i.e., arrays and structures). File names, entry names, and programmer defined condition names are not considered to be data.

3.14.1 Scalar Items

A data item may be either a constant or the value of a scalar variable. Constants and scalar variables are called scalar items. Scalar variables and scalar data items may also be called element variables and element data items, respectively.

3.14.1.1 Constants

A constant is a data item that denotes itself (i.e., its representation is both its name and its value); thus, it cannot change during the execution of a program. Each constant has a type, as described in this section. A signed constant is an arithmetic constant preceded by one of the prefix operators + or -. Wherever the word "constant" appears alone, and refers to an arithmetic constant, it is assumed to be an unsigned constant.

3.14.1.2 Scalar Variables

A scalar variable, like a constant, denotes a data item. This data item is called the value of the scalar variable. Unlike a constant, however, a variable may take on more than one value consecutively during the execution of a program. The set of values that a variable may take on is the range of the variable. The range of a variable is always restricted to one data type (and, if the type is arithmetic, to one base, scale, mode, and precision). If there are no further restrictions declared for the range, the variable may assume values over the entire set of data of that type.

Reference is made to a scalar variable by a name, which may be a simple name, a qualified name, or a subscripted qualified name. (See Section 3.14.3, Naming)

3.14.2 Data Aggregates

In PL/I, all classes of variable data items may be grouped into arrays or structures. Rules for this grouping are given below. (For the method of referring to an array or structure or a particular item of an array or structure, see Section 3.14.3, Naming.)

3.14.2.1 Arrays

An array is an N-dimensional, ordered collection of elements, all of which have identical data declarations. If arithmetic, all of the elements of the array must have the same base, scale, mode, and precision or the same picture. If character-string or bit-string, all of the elements must have the same actual length (if fixed length) or the same maximum length (if varying length). The number of dimensions of an array, and the upper and lower bounds of each dimension, are specified by the use of the dimension attribute.

Example

```
DECLARE A (3,4);
```

This statement defines A as an array with 2 dimensions: 3 rows and 4 columns. The matrix given below illustrates the array A.

```
A(1,1) A(1,2) A(1,3) A(1,4)
A(2,1) A(2,2) A(2,3) A(2,4)
A(3,1) A(3,2) A(3,3) A(3,4)
```

The elements of an array may be structures. (See Section 3.13.2.3, Structure Arrays.)

3.14.2.2 Structures

A structure is a hierarchial collection of scalar variables, arrays, and structures. These need not be of the same data type nor have the same attributes.

The outermost structure is a major structure, and all contained structures are minor structures. Scalar variables or arrays forming the elements of major or minor structures are known as base elements.

A structure is specified by declaring the major structure name and following it with the names of all contained minor structures and base elements. Each name is preceded by a level number, which is an unsigned non-zero integer constant. A major structure is always at level one and all minor structures and base elements contained in a structure (at level N) have a level number that is numerically greater than N, but they need not be at level N + 1, nor need they all have the same level number.

A minor structure at level N contains all following items declared with level numbers greater than N up to but not including the next item with a level number less than or equal to N. A major structure description is terminated by the declaration of another item at level one, by the declaration of an item having no level number, or by the end of a DECLARE statement.

Examples

1.
```
DECLARE 1 PAYROLL,
        2 NAME,
        2 HOURS,
          3 REGULAR,
          3 OVERTIME,
        2 RATE;
```

In the above example, PAYROLL is defined as the major structure containing the scalar variables NAME and RATE and the structure HOURS. The structure HOURS contains the scalar variables REGULAR and OVERTIME.

2.
```
DECLARE 1 A,
        2 B,
        2 C,
          3 D (2),
          3 E,
        2 F;
```

The decimal integers before the identifiers specify the levels. The decimal integer in parentheses specifies the bounds of the one-dimensional array. A is defined as the major structure and contains the minor structure C and the scalar variables B and F. C contains D, a one-dimensional array with two scalar variables, and the scalar variable E.

```
3.  DECLARE 1 A,
          3 B,
          2 C;
```

B and C are at the same level although their level numbers differ.

3.14.2.3 Structure Arrays

A structure array is formed by giving the dimension attribute to a structure.

Examples

```
1.  DECLARE 1 CARDIN(2),
            2 NAME,
            2 WAGES,
              3 NORMAL,
              3 OVERTIME;
```

The decimal integers before the identifiers specify the level. The name, CARDIN, represents an array of structures. Because CARDIN has a dimension specified, NAME, NORMAL and OVERTIME are arrays, and their elements are referred to by subscripted names. Note that WAGES is also a structure array.

The form of the data is as follows:

```
CARDIN (1) [NAME (1)
           [WAGES (1)  [NORMAL (1)
                       [OVERTIME (1)

CARDIN (2) [NAME (2)
           [WAGES (2)  [NORMAL (2)
                       [OVERTIME (2)
```

```
2.  DECLARE 1 X,
            2 Y,
            2 Z (2),
              3 P (2,2),
              3 Q,
            2 R;
```

X is an undimensioned major structure containing scalar variables, arrays and a structure.

Y is a scalar variable.

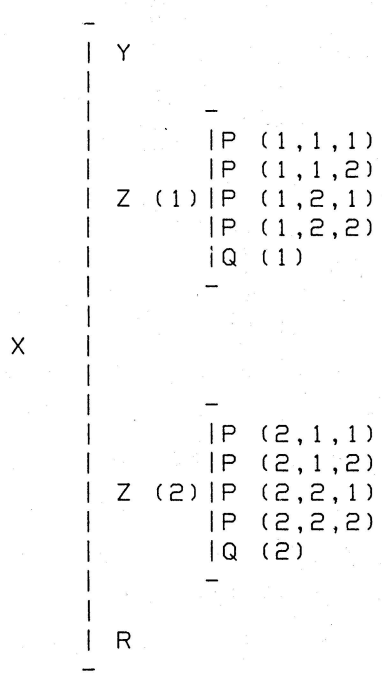
Z is a structure array.

P is a three-dimensional array.

Q is a one-dimensional array.

R is a scalar variable.

The form of the data is as follows:



3.14.3 Naming

This section describes the rules for referring to a particular data item, groups of items, arrays, and structures. The permitted types of data names are simple, qualified, subscripted, and subscripted qualified.

3.14.3.1 Simple Names

A simple name is an identifier that refers to a scalar, an array, or a structure. (See Section 3.7, Identifiers.)

3.14.3.2 Subscripted Names

Syntax

```
<subscripted-name> ::=
  <identifier> (<subscript> [,<subscript>]...)

<subscript> ::= <scalar-expression>|*
```

Semantics

A subscripted name is used to refer to an element or a cross section of an array. It is a simple name that has been declared to be the name of an array followed by a list of subscripts. The subscripts are separated by commas and are enclosed in parentheses. A subscript is an asterisk or a scalar expression that is evaluated and converted to an integer before use. (See Section 5.3, Evaluation of Expressions.) The number of subscripts must be equal to the number of dimensions of the array, and the value of a specified subscript must fall within the bounds declared for that dimension of the array.

The concept of cross sections of arrays is a logical extension of the subscripting notation. A cross section of an array is referred to by the array name, followed by a list of subscripts, at least one of which is an asterisk (*). The subscripts are separated by commas, and the entire list is enclosed in parentheses. The number of items in the list must be equal to the entire list enclosed in parentheses. The number of items in the list must be equal to the number of dimensions of the array. If the array is of dimensionality N, an asterisk may appear in K <= N positions. If the Jth list position is occupied by an asterisk, the cross section of the array includes elements covered by varying the Jth subscript between its bounds. The dimensionality of the cross section is equal to the number of asterisks, K, in the subscript list. If all subscript positions are occupied by asterisks, then this reference to the cross section is equivalent to a reference to the entire array.

A cross section may be used anywhere that the name of an array of dimensionality K is required. Subsequent references to the word array in this document should therefore be taken to include cross section of arrays.

Examples

If A is the array:

```
 _         _
| 1  2  3 |
|         |
| 4  5  6 |
|         |
| 7  8  9 |
 -         -
```

1. A(3,*) denotes the third row of the array, i.e.,

   [7 8 9]

2. A(*,2) denotes the second column of the array, i.e.,

```
 _   _
| 2 |
|   |
| 5 |
|   |
| 8 |
 -   -
```

3. If R*S = 2 and T = 3, then A(R*S,T) denotes the element [6] of array A.

4. A(3,3) denotes the element [9] of array A.

3.14.3.3 Qualified Names

Syntax

<qualified name> ::= <identifier>(.<identifier>)...

Semantics

A simple name usually refers uniquely to a scalar variable, an array, or a structure. However, it is possible for a name to refer to more than one variable, array, or structure if the identically named items are themselves parts of different structures. In order to avoid any ambiguity in referring to these duplicately/ named items, it is necessary to create a unique name; this is done by forming a qualified name. This means that the name common to more than one item is preceded by the name of the structure in which it is contained. This, in turn, can be preceded by the name of its containing structure, and so on, until the qualified name refers uniquely to the required item.

Thus, the qualified name is a sequence of names specified left to right in the order of increasing level numbers. The names are separated by periods, and the blanks may be placed as desired around the periods. The sequence of names need not include all of the containing structures, but it must include sufficient names to resolve any ambiguity. Any of the names may be subscripted.

If the sequence of names includes the names of all the structures containing the member with the rightmost name, then that name is said to be completely qualified.

If the sequence of names includes only some of the names of the structures containing members with the rightmost name, then that name is said to be partially qualified.

A completely or partially qualified name is said to be applicable to the declarations of structures that include the same hierarchy of structure names used in the qualified name.

A qualified name is ambiguous if it refers to more than one structure member. A reference to a structure member by means of an unqualified name is ambiguous if any other structure name internal to the same block has the same identifier. An ambiguous reference is an error.

Where, however, a level-1 name and structure member name internal to the same block have the same identifier, then the unqualified use of that identifier is taken to refer to the level-1 name. Reference to the structure member can, in this case, be achieved only by means of a suitably qualified name.

The qualified name, once composed, is itself a name. Subsequently, in this document, when the terms scalar variable name, array name, or structure name are used they should also be taken to include qualified names.

Examples

```
1. DECLARE 1 A, 2 C, 2 D, 3 E;
   BEGIN;
   DECLARE 1 A, 2 B, 3 C, 3 E;
   A.C = D.E;
   END;
```

A.C refers to C in the inner block.
D.E refers to E in the outer block.

```
2. DECLARE 1 A, 2 B, 2 C, 3 D, 2 D;
```

A.D refers to the second D, since A.D is a complete qualification of only the second D; the first D would have to be referred to as A.C.D or C.D.

```
3. DECLARE 1 A, 2 B, 3 C, 2 D, 3 C;
```

A.C is ambiguous because neither C is completely qualified by this reference. A.B.C and A.D.C are unambiguous completely qualified names of both C's in the declaration.

```
4. DECLARE 1 A, 2 A, 3 A;
```

A refers to the first A.
A.A refers to the second A.
A.A.A refers to the third A.

```
5. DECLARE X; DECLARE 1 Y, 2 X, 3 Z, 3 A, 2 Y, 3 Z, 3 A;
```

X refers to the first DECLARE.
Y.Z is ambiguous.
Y.Y.Z refers to the second Z.
Y.X.Z refers to the first Z.

6. A program may contain the structures:

```
DECLARE 1 CARDIN, 2 PARTNO, 2 DESCRIPTION,
  2 PRICE;
DECLARE 1 CARDOUT, 2 PARTNO, 2 DESCRIPTION,
  2 PRICE;
```

Elements are then referred to as:

```
CARDIN.PARTNO
CARDOUT.PARTNO
CARDIN.PRICE
```

7. A program may contain the structure:

```
DECLARE 1 MARRIAGE, 2 MAN, 3 NAME,
  3 DATE, 2 WOMAN, 3 NAME, 3 DATE;
```

Elements are then referred to as:

```
MAN.NAME
      or MARRIAGE.MAN.NAME

WOMAN.NAME
```

```
      or MARRIAGE.WOMAN.NAME
```

8. If the same program also contains the structure:

```
DECLARE 1 BIRTH, 2 WOMAN, 3 NAME,
      3 DATE, 2 ADDRESS:
```

Elements are then referred to as:

```
MAN.NAME
      or MARRIAGE.MAN.NAME
MARRIAGE.WOMAN.NAME
BIRTH.NAME
      or BIRTH.WOMAN.NAME
ADDRESS
```

And the minor structures are referred to as:

```
MARRIAGE . WOMAN
BIRTH . WOMAN
```

3.14.3.4 Subscripted Qualified Names

Syntax

```
<subscripted-qualified-name>::=
     <identifier> [(<subscript> [,<subscript>]...)]
     [. <identifier> [(<subscript> [,<subscript>]...)]]...
```

If any subscripts are given in reference to a qualified name, all those subscripts which apply to the dimensions of containing structures must be given.

Examples

1. A is a structure array with the following description:

```
DECLARE 1 A (10,12) 2 B (5), 3 C (7), 3 D;
```

The following subscripted qualified names refer to the same element, which is the seventh element of C contained in the fifth element of B contained in the tenth row and twelfth column of A:

```
(1)     A (10,12) .B (5) . C (7)
(2)     A (10) . B (12,5) . C (7)
(3)     A (10) . B (12) . C (5,7)
(4)     A . B (10,12,5) . C(7)
(5)     A . B (10,12) . C (5,7)
(6)     A . B (10) . C (12,5,7)
(7)     A . B. C (10,12,5,7)
(8)     A (10,12) . B. C (5,7)
(9)     A (10) . B . C (12,5,7)
(10)    A (10,12,5,7) . B . C
```

If structure B, but not structure A is necessary for unique identification of this use of C, any of the forms (4), (5), (6), or (7) may be used without including the A.

If structure A, but not structure B, is necessary for identification of C, forms (7), (8), (9), or (10) may be used without including the B.

2. If FIELD is the array of structures:

```
DECLARE 1 FIELD (3),
        2 STATUS,
        2 VALUE;
```

Then FIELD (*). STATUS represents the array:

```
FIELD(1).STATUS
FIELD(2).STATUS
FIELD(3).STATUS
```

3.15 Comments

Syntax

<comment> :¯ /* [<comment-string>] */
<comment-string> :¯ <data-character>...

Semantics

Comments are normally used for documentation and do not participate in the execution of a program.

A <comment-string> must not contain the character combination */ in that sequence. Example

```
LABEL: /* THE BLOCK OF CODING BETWEEN
       BEGIN-END IS USED FOR PAYROLL
       CALCULATIONS */

       BEGIN;
       .
       .
       .
       END;
```

SECTION 4. DATA DESCRIPTION

A data description in PL/I attaches an identifier to a particular variable. Data descriptions are specified by data description attributes. This section includes a description of each data specification attribute of the PL/I language. An identifier appearing in a PL/I program may refer to one of many classes of objects. The recognition of an identifier as a particular name is established through declaration of the name. Those properties that characterize the object represented by the name, and the scope of the name itself, together make up the set of attributes that are to be associated with the name.

4.1 DECLARATIONS

A given identifier is established as a name, which holds throughout a certain scope in the program, and a set of attributes may be associated with the name by means of a declaration.

If a declaration is internal to a certain block, then the name is said to be declared in that block.

In a program, a given identifier may be established in different parts of the program as different names. For example, an identifier may represent an arithmetic variable in one part of a program and an entry name in another part. The two parts cannot overlap.

Each different use of the identifier is established by a different declaration. References to different uses are distinguished by the rules of scope.

Declarations may be explicit, contextual, or implicit.

4.1.1 Explicit Declarations

Explicit declarations are made through use of the DECLARE statement, label prefixes, and specification in a parameter list. In an explicit declaration an identifier is established as a name and may be given a certain set of attributes.

Only one DECLARE statement can be used to establish an internal name. However, complementary sets of explicit declarations are permitted.

One or more explicit declarations in parameter lists may be combined with an explicit declaration in a DECLARE statement.

All declarations of a complementary set must be internal to the same block.

4.1.1.1 Label Prefixes

A label acting as a prefix to a PROCEDURE statement or an ENTRY statement explicitly declares the identifier as having the ENTRY attribute. If the PROCEDURE statement or ENTRY statement applies to an external procedure, it is given the EXTERNAL attribute, and this declaration is considered to be internal to an imaginary block containing the external procedure. In all other cases, the label is

given the INTERNAL attribute, and the declaration is internal to the block containing the procedure.

A label acting as a prefix to any other statement is an explicit declaration of the identifier as a statement label constant. The declaration is internal to the block containing the statement.

4.1.1.2 Parameters

The appearance of an identifier in a parameter list of a PROCEDURE statement or an ENTRY statement explicitly declares the PARAMETER attribute for that name, as well as explicitly declaring the name itself.

4.1.2 Contextual Declarations

The syntax of PL/I allows unqualified identifiers appearing in certain contexts to be recognized without an explicit declaration. Such contextual declarations will not, however, override any explicit declaration of the same identifier whose scope includes the block containing a statement that might otherwise cause contextual declaration.

Contextual declarations can occur as follows:

1. POINTER. An undeclared identifier can be contextually declared as a pointer variable if it appears:

   a. In parentheses following the keyword BASED, in a BASED attribute specification of a DECLARE statement.

   b. In parentheses following the keyword SET, in the SET option of an ALLOCATE, LOCATE, or READ statement.

   c. As a locator qualifier.

2. AREA. An undeclared identifier can be contextually declared as an area variable if it appears in parentheses following the keyword IN, in the IN clause of an ALLOCATE or FREE statement, or if it appears in parentheses following the keyword OFFSET in an OFFSET declaration, or by its appearance in an IN attribute specification.

3. TASK. An undeclared identifier can be contextually declared as a task variable if it appears in parentheses following the keyword TASK in the TASK option of a CALL statement.

4. BUILT-IN. An undeclared identifier can be contextually declared with the BUILTIN attribute if it appears followed by an argument list.

5. FILE. An undeclared identifier can be contextually declared as a file name if it appears:

   a. In the FILE option of a data transmission statement.

   b. In parentheses following one of the input/output condition names.

6. CONDITION-NAME. An undeclared identifier can be contextually

declared as a condition name if it appears in parentheses following the keyword CONDITION in an ON, SIGNAL, or REVERT statement.

A contextual declaration is treated as if it had been made in the external procedure, even if the reference is made in an internal block. The scope of a contextually declared name is the entire external procedure, except for any internal blocks in which the same identifier is explicitly declared.

4.1.3 Implicit Declarations

An identifier which is neither explicitly nor contextually declared is implicitly declared in the outermost block and is given the VARIABLE attribute.

Each "descriptor" appearing in an ENTRY or RETURNS attribute specification is implicitly declared in the block in which the entry declaration has been established.

Each literal constant is implicitly declared in the block to which it is internal and is given the CONSTANT attribute. Declarations of bit string constants are given the BIT and LENGTH attributes. Declarations of character string constants are given the CHARACTER and LENGTH attributes. Declarations of arithmetic constants are given the FIXED attribute unless they contain "E", in which case they are given the FLOAT attribute.

4.1.4 Establishment of Declarations

The establishment of declarations of names is based on a system of priority, with explicit declarations having the highest priority. It follows a three-step process:

1. Explicit declarations are established, with the scope of each name determined by the block in which the declaration is made.

2. Undeclared identifiers are scanned to determine if their meaning can be recognized contextually (See Section 4.1.2, Contextual Declarations.) Note that no contextual declaration of an identifier can be made if the identifier lies within the scope of an already established explicit declaration. If any undeclared identifier is recognized contextually, a declaration is generated, with scope established as if the declaration had been made in the external procedure.

3. Following contextual declarations, implicit declarations are established for all remaining undeclared identifiers, with scope established as if the declaration were made in the external procedure.

4.1.5 Scope of Declarations

When a declaration of an identifier is made in a block, there is a certain well-defined region of the program over which this declaration is applicable. This region is called the scope of the declaration or the scope of the name established by the declaration.

The scope of a declaration of an identifier is defined as that block B

to which the declaration is internal, but excluding from block B all contained blocks to which another declaration of the same identifier is internal.

4.1.5.1 Scope of External Names

In general, distinct declarations of the same identifier imply distinct names with distinct non-overlapping scopes. It is possible, however, to establish the same name for distinct declarations of the same identifier by means of the EXTERNAL attribute. The EXTERNAL attribute is defined as follows:

> A declaration of an identifier that declares the identifier as EXTERNAL is called an external declaration for the identifier. All external declarations for the same identifier in a program will be linked and considered as establishing the same name. The scope of this name will be the union of the scopes of all the external declarations for this identifier.

In all of the external declarations for the same identifier, the attributes declared must be consistent, since the declarations all involve a single name.

Example

> It would be an error if the identifier "ID" were used as an external file name in a READ statement in a program, and in the same program declared "ID" EXTERNAL ENTRY.

The EXTERNAL attribute can be used to communicate between different external procedures or to obtain non-continuous scopes for a name within an external procedure.

An external name is a name that has the EXTERNAL attribute. If a name is not external, it is said to be internal and has the attribute INTERNAL.

The following examples illustrate scope of declarations. The numbers on the left are for reference only, and are not part of the procedure. See the table below for an explanation of the scope and use of each name.

Examples

1.

```
1 A: PROCEDURE;
2    DECLARE (X,Z) FLOAT;
                      .
                      .
                      .
3  B: PROCEDURE (Y);
4     DECLARE Y BIT (6);
5     C: BEGIN;
6        DECLARE (A,X) FIXED;
                      .
                      .
                      .
7     Y: RETURN;
      END C;
   END B;
8  D: PROCEDURE;
9     DECLARE X FILE;
10    Y=Z;
                      .
                      .
                      .
   END D;
  END A;
```

Since entry names of external procedures and file names have the EXTERNAL attribute, the scope of the entry name A and the file name X may include parts of other external procedures of the program.

Table of Scope and Use of Names in Example 1.

REFERENCE LINE	NAME	USE	SCOPE (BY BLOCK NAME)
1	A	External Entry Name	All of A except C
2	X	Floating-Point Variable	All of A except C and D
2	Z	Floating-Point Variable	All of A
3	B	Internal Entry Name	All of A
4	Y	Bit String	All of B except C
5	C	Statement Label	All of B
6	A	Fixed-Point Variable	All of C
6	X	Fixed-Point Variable	All of C
7	Y	Statement Label	All of C
8	D	Internal Entry Name	All of A
9	X	File Name	All of D
10	Y	Floating-Point Variable	All of A except B

2.

```
  A: PROCEDURE;
1    DECLARE X EXTERNAL;
           .
           .
           .
     B: PROCEDURE;
2       DECLARE X FIXED;
              .
              .
              .
        C: BEGIN;
3          DECLARE X EXTERNAL;
                 .
                 .
                 .
        END C;
     END B;
  END A;
  D: PROCEDURE;
4    DECLARE X FIXED;
           .
           .
           .
     E: PROCEDURE;
5       DECLARE X EXTERNAL;
              .
              .
              .
    END E;
  END D;
```

In example 2 there are five declarations for the identifier X.

Declaration 2 declares X as a fixed-point variable name. Its scope is all of block B except block C.

Declaration 4 declares X as another fixed-point variable name, distinct from that of declaration 2. Its scope is all of block D except block E.

Declarations 1, 3, 5 all establish X as a single name. Its scope is all the program except the scope of declarations 2 and 4.

4.1.5.2 Basic Rule of Use of Names

A name is said to be known only within its scope. This definition suggests a basic rule on the use of names:

> All appearances of an identifier which are intended to represent a given name in a program must lie within the scope of that name.

There are many implications to the above rule. One of the most important is the limitation of transfer of control by the statement "GO TO A", where A is a statement label.

The statement "GO TO A", internal to a block B, can cause a transfer of control to another statement internal to block B or to a statement in a block containing B, and to no other statement. In particular, it

cannot transfer control to any point within a block contained in B.

4.2 CLASSIFICATION OF ATTRIBUTES

Attributes may be classified into the following logical groups: arithmetic, string, picture, program-control, entry, file, array, structure, storage, scope, and data.

Syntax

```
<data-description-attributes> ::=

    <arithmetic> |

    <string> |

    <picture> |

    <program-control> |

    <entry> |

    <file> |

    <array> |

    <structure> |

    <storage> |

    <scope> |

    <data>
```

4.3 ARITHMETIC DATA DESCRIPTIONS

Arithmetic attributes specify the representation of numeric data.

Syntax

```
<arithmetic> ::=

    <mode-attribute> |

    <base-attribute> |

    <scale-attribute> |

    <precision-attribute>
```

4.3.1 Mode Attribute

The <mode-attribute> is used to specify the mode of an arithmetic variable.

Syntax

<mode-attribute> ::= REAL

Semantics

REAL specifies that the mode of an arithmetic variable is to be a real number.

Assumptions:

REAL is implied for all arithmetic variables.

4.3.2 Base Attributes

The <base-attribute> specifies the base of the data item represented by an arithmetic variable.

Syntax

<base attribute> ::= BINARY | DECIMAL

Semantics

BINARY specifies that the data representation is binary. DECIMAL specifies that the data representation is decimal.

Restrictions:

The <base-attribute> cannot be specified with the PICTURE attribute.

Assumptions:

Any identifier starting with any of the letters I thru N defaults to BINARY. All other identifiers default to DECIMAL.

4.3.3 Scale Attributes

The <scale-attribute> specifies the scale of the arithmetic variable being declared.

Syntax

<scale-attribute> ::= FIXED | FLOAT

Semantics

FIXED specifies that the variable is to represent fixed-point data items. FLOAT specifies that the variable is to represent floating-point data items.

Restrictions:

The <scale-attribute> may not be specified with the PICTURE attribute.

Assumptions:

See Section 4.14.2, Standard System Default Rules for the default scale attribute specifications.

4.3.4 Precision Attribute

The <precision-attribute> is used to specify the minimum number of significant digits and the scale factor of a variable.

Syntax

```
<precision-attribute> ::=
     [PRECISION] (<number-of-digits> [,<scale-factor>])

<scale-factor> ::=  [+|-] <integer-constant>

<number-of-digits> ::= <integer-constant>
```

Semantics

The <number-of-digits> is an unsigned integer constant and specifies the minimum number of significant digits to be maintained for the value of the data item. The <scale-factor> is an optionally-signed integer constant and specifies the assumed position of the binary or decimal point. No point is actually present. Its location is assumed. The precision attribute specification is often represented, for brevity, as (P,Q), where P represents the <number-of-digits> and Q represents the <scale-factor>.

The <scale-factor> can be negative, and it can be larger than the number of digits. A negative scale factor always specifies integers, with the point assumed to be located Q places to the right of the rightmost actual digit. A positive scale factor (Q) that is larger than the number of digits always specifies a fraction, with the point assumed to be located Q places to the left of the rightmost actual digit.

A scale factor of 0 specifies integers with the decimal point assumed to be to the right of the rightmost digit.

Restrictions:

If the keyword PRECISION is omitted, the precision attribute, if it appears, must immediately follow the scale (FIXED or FLOAT), base (DECIMAL or BINARY), or mode (REAL) attribute at the same factoring level.

The scale factor can be specified for fixed-point variables only. The number of digits can be specified for both fixed-point and floating-point variables.

The <precision-attribute> cannot be specified in combination with the PICTURE attribute.

Assumptions:

If no scale factor is specified it is assumed to be 0.

Default values for precision are:

(5,0) for fixed decimal

(15,0) for fixed binary

(6) for float decimal

(21) for float binary

4.4 STRING DATA DESCRIPTIONS

String attributes specify the representation of string variables.

Syntax

<string> ::=

<string-type-attribute> |

<length-attribute> |

<varying-attribute>

4.4.1 String Type Attribute

The <string-type-attribute> specifies the internal representation of the string variables.

Syntax

<string-type-attribute> ::= BIT | CHARACTER

Semantics

BIT specifies a bit string. CHARACTER specifies a character string.

The PICTURE attribute can be used instead of CHARACTER to declare a fixed-length string variable.

Restrictions:

The <string-type-attribute> cannot be specified with the PICTURE attribute.

4.4.2 Length Attribute

The <length-attribute> is used to specify the length of a fixed length string or the maximum length of a varying length string.

Syntax

<length-attribute> ::= [LENGTH]

((<expression> [<refer-option>]) |

<asterisk>)

<refer-option> ::=

REFER (<scalar-name>)

<asterisk> ::= *

Semantics

If the length expression contains a <refer-option>, it is converted to an integer when storage is allocated for the variable. See section 4.11, Based Attribute for a complete description of the <refer-option>.

The length of a string parameter may be specified as an <integer> or as an <asterisk>. An <asterisk> indicates that the actual length of the string parameter is the length of the associated string.

055

Restrictions:

An <asterisk> may only be used to specify the string length of a data item that is a parameter.

If the keyword LENGTH is omitted, the <length-attribute>, if it appears, must immediately follow the <string-type-attribute>.

If the <length-attribute> appears with the RETURN attribute or the STATIC data attribute it may only be an unsigned integer constant.

The <refer-option> may only be used with variables declared with the BASED attribute.

Assumptions:

The default length for strings is (1).

4.4.3 Varying Attribute

The <varying-attribute> specifies whether or not a string may vary in length.

Syntax

<varying-attribute> ::= VARYING | NONVARYING

Semantics

The length of a NONVARYING string and the maximum length of a VARYING string are specified by the <length-attribute>.

Restrictions:

The <varying-attribute> may not be specified with the PICTURE attribute.

Assumptions:

If the <varying-attribute> is not specified, the default is NONVARYING.

4.5 PICTURE DATA DESCRIPTIONS

The picture data attribute is used to define the internal and external formats of character string and numeric character data and to specify the editing of data. Picture characters are used to describe the attributes of the associated data item, whether it is the value of a variable or a data item to be transmitted between the program and external storage.

Syntax

<picture-attribute> ::= PICTURE '<picture-type>'

<picture-type> ::=

<character-picture-specification> |

<numeric-picture-specification>

Semantics

A picture specification always describes a character representation that is either a character string data item or a numeric character data item. A pictured numeric character item is one in which the data itself can consist only of decimal digits, a decimal point, and optionally, a plus or minus sign. Other characters generally associated with arithmetic data, such as currency symbols, can also be specified, but they are not part of the arithmetic value of the numeric character variable. These characters are stored with the digits and are considered to be part of the character string value of the variable.

A picture specification, either character or numeric, is composed of a string of picture characters enclosed in single quotation marks ('). Picture characters are considered to be grouped into fields, some of which contain subfields.

Restrictions:

The PICTURE attribute cannot be specified in combination with the <base-attribute>, the <scale-attribute>, or the <precision-attribute>.

4.5.1 Character Picture Data Description

The <character-picture-specification> is used to describe a character string data item.

Syntax

<character-picture-specification> ::=

<character-picture-item>...

<character-picture-item> ::=

[(<repetition-factor>)] <character-picture-element>

<repetition-factor> ::= <integer-constant>

<character-picture-element> ::= A | X | 9

Semantics

An individual picture character may be preceded by a <repetition-factor>, which indicates that the picture character is to be repeated that number of times. If the repeat part is zero, the picture character is ignored.

X Specifies that the associated position can contain any character whose internal bit configuration can be recognized by the computer.

A Specifies that the associated position can contain any alphabetic character or a blank character.

9 Specifies that the associated position can contain any decimal digit or a blank character.

Restrictions:

The <repetition-factor> must be an unsigned integer constant.

Data assigned to a variable declared with a character string picture specification, or data to be written with a character string picture item, should conform, or be convertible, character by character, to the picture specification. If it does not the CONVERSION condition is raised.

Editing, such as zero suppression and the insertion of other characters, cannot be specified for picture character string data.

Assumptions:

If no <repetition-factor> is specified, one is assumed.

4.5.2 Numeric Picture Data Description

The <numeric-picture-specification> is used to represent numeric values.

Syntax

```
<numeric-picture-specification> ::=
     <fixed-point-picture>[<picture-scale-factor>] |
     <floating-point-picture>
<fixed-point-picture> ::= <numeric-picture-item>...
<floating-point-picture> ::=
     <picture-mantissa> <picture-exponent>
<picture-mantissa> ::= <numeric-picture-item>...
<picture-exponent> ::= <numeric-picture-item>...
<numeric-picture-item> ::=
     [(<repetition-factor>)] <numeric-picture-element>
<numeric-picture-element> ::=
     <decimal-specifier> |
     <packed-specifier> |
     <zero-suppression-character> |
     <insertion-character> |
     <sign-currency-character> |
     <credit> | <debit> |
     <overpunch-character> |
     <exponent-character> |
     <picture-scale-factor>
```

Semantics

The <repetition-factor> for numeric pictures is the same as for character pictures.

A numeric picture variable can be considered to have two different kinds of values, depending upon its use: (1) its arithmetic value and (2) its character string value.

The arithmetic value is the value expressed by the decimal digits of the data item, the assumed location of a decimal point and possibly a sign. The arithmetic value of a numeric character variable is used whenever the variable appears in an expression that results in a coded arithmetic value or whenever the variable is assigned to a coded arithmetic variable. In such cases, the arithmetic value of the numeric character variable is converted to internal coded arithmetic representation.

The character string value is the value expressed by the decimal digits of the data item, as well as all of the editing and insertion characters appearing in the picture specification. The character string value does not, however, include the assumed location of a decimal point, as specified by picture character V, or an assumed exponent, as specified by the picture character K. The character string value of a numeric character variable is used whenever the variable appears in a character string variable or whenever the data is printed using the PUT statement with the DATA, LIST, or STRING option that is defined on the numeric character variable. In such cases no data conversion occurs.

Numeric picture data must be decimal, except for picture '1' specifications, which are binary. See Section 4.5.2.2, Packed Picture Classification. The arithmetic value is stored internally in character form. Numeric picture data is converted to coded arithmetic before arithmetic operations can be performed.

A numeric picture data item can have a decimal base. Its scale and precision are specified by the picture characters.

The picture characters for numeric character data can specify detailed editing of the data.

The picture characters in these groups may be used in various combinations. Consequently, a numeric character specification can consist of two or more parts such as a sign specification, an integer subfield, a fractional subfield and, for floating-point, an exponent field.

Restrictions:

Data assigned to a variable declared with a numeric picture specification, or data to be written with a numeric picture format item, must be either internal coded arithmetic data or data that can be converted to coded arithmetic.

The picture specification of a numeric picture may not contain X or A since numeric character data must represent numeric values.

A major requirement of the picture specification for numeric character data is that each field must contain at least one picture character that specifies a digit position. This digit character, however, need not be the picture character 9. Other picture characters, such as drifting characters, or zero suppression characters also specify digit positions. At least one of these characters must be used to define a numeric picture specification.

All the picture characters, except the point specifier, V, act as checking characters when used in a P format item for edit-directed input.

4.5.2.1 Decimal Specifiers

The <decimal-specifier> picture characters are used in the simplest form of decimal numeric character specifications that represent fixed-point decimal values.

Syntax

<decimal-specifier> ::= 9 | V

Semantics

9 Specifies that the associated position in the data item is to contain a decimal digit.

V Specifies that a decimal point is assumed at this position in the associated data item. However, it does not specify that an actual decimal point is to be inserted.

The integer and fractional parts of the assigned value are aligned on the V character, actual or assumed. Because of this alignment, an assigned value may be truncated or extended with zero digits at either end.

Character V is considered to be a subfield delimiter in the picture specification, that is, the portion preceding V and the portion following it (if any) are each a subfield of the specification.

Restrictions:

The V character cannot appear more than once in a picture specification, and cannot appear in the second field of a picture specification of a floating-point decimal value.

If, when alignment occurs, significant digits are truncated on the left, the result is undefined and the SIZE condition will be raised.

Assumptions:

If the V character does not appear in the picture specification of a fixed-point decimal value, or in the first field of a picture specification of a floating-point decimal value, a V is assumed at the right end of the field specification.

4.5.2.2 Packed Picture Classification

The <packed-specifier> picture characters allow the use of 4-bit packed decimal digits, binary digits, and signs in pictures.

Syntax

<packed-specifier> ::= H | 1 | S | V

Semantics

H specifies that the associated position will contain a 4-bit packed decimal digit.

1 specifies that the associated position will contain a binary digit, i.e., either 1B or 0B.

S specifies that the associated position in a packed decimal picture will contain a 4-bit sign (either 1101 or 1100).

V specifies the implied packed decimal point, or binary point.

The PICTURE 'H' specification should be used instead of the decimal fixed attributes and the PICTURE '1' for binary fixed attributes to ensure the correct internal representation for non-Burroughs PL/I programs.

Example

```
P1 PICTURE 'HHHVHHS'
```

The precision of P1 is (5,2) and P1 will occupy 6 4-bit bytes internally.

4.5.2.3 Zero Suppression Characters

The <zero-suppression> picture characters specify the replacement of zeros by blanks or asterisks.

Syntax

<zero-suppression-character> ::= Z | * | Y

Semantics

These picture characters specify conditional digit positions in the character string value and may cause leading zeros to be replaced by asterisks or blanks and non-leading zeros to be replaced by blanks. Leading zeros are those that occur in the leftmost digit positions of fixed-point numbers, or in the leftmost digit positions of the two parts of floating-point numbers, and are to the left of the assumed position of a decimal point and are not preceded by any of the digits 1 through 9. The leftmost non-zero digit in a number and all digits, zeros or not, to the right of it represent significant digits. A floating-point number can also have a leading zero in the exponent field.

Z specifies a conditional digit position and causes a leading zero in the associated data position to be replaced by a blank character.

* specifies a conditional digit position and is used the way the picture character Z is used, except that leading zeros are replaced by asterisks.

Y specifies a conditional digit position and causes a zero digit, leading or non-leading, in the associated position to be replaced by a blank character.

Where picture characters * or Z appear in all fractional digits positions in the specification, fractional zeros of the value are suppressed only if all positions in the fractional part contain zeros and all integer positions have been suppressed. The entire character string value of the data item will then consist of blanks or asterisks.

Restrictions:

If the picture character Z or * appears to the right of the picture character V, then all fractional digit positions in the specification, as well as all integer digit positions, must employ the Z or * picture character.

No digits in the fractional part are replaced by blanks or asterisks if the fractional part contains any significant digit.

The picture characters Z and * cannot appear in the same subfield, or to the right of a drifting picture character, or to the right of any of the picture characters 9, T, I, or R in a field.

4.5.2.4 Insertion Characters

The <insertion-character> picture characters cause the specified character to be inserted in the data.

Syntax

<insertion-character> ::=

, | . | / | B

Semantics

The specified insertion character is inserted into the associated position of the numeric character data. They do not indicate digit positions, but are inserted between digits. Each does, however, actually represent a character position in the character string value, whether or not the character is suppressed.

Insertion characters are applicable only to the character string value. They specify nothing about the arithmetic value of the data item.

, Causes a comma to be inserted into the associated position of the numeric character data when no zero suppression occurs. If zero suppression does occur, the comma is inserted only when an unsuppressed digit appears to the left of the comma position, or when a V appears immediately to the left of it and the fractional part contains any significant digits. In all other cases where zero suppression occurs, one of three possible characters is inserted in place of the comma. The choice of the character to replace the comma depends upon the first picture character that both precedes the comma position and specifies a digit position:

1. If this character position is an asterisk, the comma position is assigned an asterisk.

2. If this character position is a drifting sign or a

drifting currency symbol (see section 4.5.2.5, Signs and Currency Characters), the drifting string is assumed to include the comma position, which is assigned the drifting character.

3. If this character position is not an asterisk or a drifting character, the comma position is assigned a blank character.

. Is used the same way the comma picture character is used, except that a point (.) is assigned to the associated position. This character never causes point alignment in the picture specifications of a fixed-point decimal number and is not a part of the arithmetic value of the data item. That function is served by picture character V.

/ is used the same way the comma picture character is used, except that a (/) is inserted in the associated position.

B specifies that a blank character will always be inserted into the associated position of the character string value of the numeric character data.

The point, the comma, or the slash can be used in conjunction with the V to cause insertion of the point, comma, or slash in the position that delimits the end of the integer portion and the beginning of the fractional portion of a fixed-point, or floating-point number. The picture character V in a picture specification does not cause a decimal point to be printed. The point must immediately precede or immediately follow the V. If the point precedes the V, it will be inserted only if a significant digit appears to the left of the V, even if all fractional digits are significant. If the point immediately follows the V, it will be suppressed if all digits to the right of the V are suppressed, but it will appear if there are any fractional digits.

Assumptions:

Unless the V actually appears, it is assumed to be to the right of the rightmost digit position in the field, and point alignment is handled accordingly, even if the point insertion character appears elsewhere.

4.5.2.5 Signs and Currency Characters

The <sign-character> and <currency-character> picture characters specify signs and currency symbols in numeric character data.

Syntax

<sign-character> ::= + | - | S

<currency-character> ::= $

Semantics

The picture character $ specifies a currency symbol in the character string value of numeric character data. The picture characters +, -, and S specify signs in numeric character data.

The sign and currency picture characters may be used in either a static or a drifting manner. If the picture character appears more than once in a picture field, it is a drifting character, otherwise it is a static character.

$ Specifies the dollar sign character ($).

S Specifies the plus sign character (+) if the data value is > or = 0. If the data value is < 0, it specifies the minus character (-).

+ Specifies the plus sign character (+) if the data value is > or = 0. If the data value is < 0, a blank is specified.

- Specifies the minus sign character (-) if the data value is < 0. If the data value is > or = 0, a blank is specified.

A static character specifies that a sign, a currency symbol or a blank always appears in the associated position. An S, +, or -, used as a static character can appear to the left of all digits in mantissa and exponent fields of a floating-point specification and either to the right or left of all digit positions of a fixed-point specification. A $ used as a static character must appear either to the left of all digit positions in a field of a specification or to the right of all digit positions in a specification.

A drifting character is similar to a zero suppression character in that it can cause zero suppression. However, the character is always inserted in the position immediately to the left of the first significant digit, or in the rightmost position associated with the character. The drifting character must be specified in each digit position through which it may drift, and in one more position to the left of the digit positions. The position associated with the leftmost drifting character can contain only the drifting character or a blank, never a digit. Drifting characters must appear in strings. A string is a sequence of the same drifting character, optionally containing a V and one of the insertion characters (comma, point, slash, or B). Any insertion characters following the last drifting symbol of the string is considered part of the drifting string.

The position in the data associated with the characters slash, comma, point, and B appearing in a string of drifting characters will contain one of the following:

1. The slash, comma, point, or blank if a significant digit has appeared to the left.

2. The drifting symbol, if the next position to the right

contains the leftmost significant digit of the field.

3. Blank, if the leftmost significant digit of the field is more than one position to the right.

If a drifting string contains a V within it, the V delimits the preceding portion as a subfield, and all digit positions of the subfield following the V must also be part of the drifting string that commences the second subfield. In this case, suppression in the second subfield will occur only if all of the integer and fractional digits are zero. The resulting edited data item will then be all blanks, except for the rightmost digit position, which will contain the drifting character. If there are any significant fractional digits, the entire fractional portion will appear unsuppressed.

Restrictions:

A field of a picture specification can contain only one drifting string. If a drifting string is specified for a field, the other potentially drifting characters can appear only once in the field (i.e., the other character represents a static sign or currency symbol).

Only one type of sign character may appear in a picture field.

A drifting string cannot be preceded by a digit position.

Picture characters * and Z cannot appear to the right of a drifting string in a field.

4.5.2.6 Credit, Debit and Overpunch Characters

The <debit> and <credit> character pairs specify signs of real numeric character data items. The <overpunch-character> picture characters specify an overpunch sign in the associated digit position of numeric character data.

Syntax

<credit> ::= CR

<debit> ::= DB

<overpunch-character> ::= T | I | R

Semantics

CR Specifies that the associated positions will contain the letters CR if the value of the data item is less than zero. Otherwise the positions will contain two blanks.

DB is used the same way that CR is used except the the letters DB appear in the associated positions.

T Specifies that the associated position, on input, will contain a digit overpunched with the sign of the data. It also specifies that an overpunch is to be indicated in the character string value.

I Specifies that the associated position, on input will contain

the digit overpunched if the value is > or = 0. Otherwise, it will contain the digit with no overpunching. It also specifies that an overpunch is to be indicated in the character string value if the data value is > or = 0.

R Specifies that the associated position, on input, will contain a digit overpunched if the value is < or = 0. Otherwise, it will contain the digit with no overpunching. It also specifies that an overpunch is to be indicated in the character string value if the data value is < 0.

When an overpunch character occurs in a P format item for edit-directed input, the corresponding character in the input stream may contain an overpunched sign.

Restrictions:

CR and DB may appear only to the right of all digit positions in a picture field.

Only one overpunched sign can appear in a specification for a fixed-point number. A floating-point specification can contain two: one in the mantissa field and one in the exponent field. The overpunch character can, however, be specified for any digit position within a field. The overpunched number then will appear in the specified digit position.

Picture characters CR, DB, T, I, and R cannot be used with any other sign characters in the same field.

4.5.2.7 Exponent Characters

The <exponent-character> picture characters delimit the exponent field of a numeric character specification that describes a floating-point number.

Syntax

<exponent-character> ::= E | K

Semantics

K Specifies that the exponent field appears to the right of the associated position. It does not specify a character in the numeric character data item.

E Specifies that the associated position contains the letter E, which indicates the start of the exponent field, The value of the exponent is adjusted in the character string value so that the first significant digit of the first field (the mantissa) appears in the position associated with the first digit specifier of the specification (even if it is a zero suppression character).

Restrictions:

The exponent field is always the last field of a numeric character floating-point picture specification.

Picture characters K and E cannot appear in the same picture specification.

4.5.2.8 Scale Factor

The <picture-scale-factor> specifies a scaling factor for fixed-point decimal numbers.

Syntax

<picture-scale-factor> ::=

F([+|-] <integer-constant>)

Semantics

F Specifies that the optionally signed integer constant enclosed in parentheses is the scaling factor.

The scaling factor specifies that the decimal point in the arithmetic value of the variable is that number of places to the right (if the scaling factor is positive) or to the left (if it is negative) of its assumed position in the character string value.

Restrictions:

The scaling factor is always the last field of a numeric character fixed-point picture specification.

4.6 PROGRAM CONTROL DATA DESCRIPTION

Syntax

<program-control> ::=

<label-attribute> |

<format-attribute> |

<locator-attribute> |

<in-attribute> |

<area-attribute>

4.6.1 Label Attribute

The <label-attribute> specifies that the identifier being declared is a label variable and is to have statement label constants as values.

Syntax

```
<label-attribute> ::= LABEL

    [(<statement-label-constant>

    [,<statement-label-constant>]...)]
```

Semantics

If a list of <statement-label-constant>s is given, the variable can have as values only members of the list. If multiple labels are prefixed to a statement, all of the labels have the same value. The label constants in the list must be known in the block containing the declaration. These label constants may include elements of label constant arrays. Label constant arrays are declared in a block via the appearance of such arrays as statement prefixes.

The parenthesized list of <statement-label-constant>s can be used in a LABEL attribute specification for a label array. The list applies to each element of the array.

If the variable is a parameter, its value can be any statement label variable or constant passed as an argument. If the argument is a label variable, the value of the label parameter can be any value permitted for the label variable that is passed.

Restrictions:

An entry name cannot be a value of a label variable (See Section 4.7, Entry Name Attributes).

The INITIAL attribute cannot be specified for label variables with the STATIC attribute.

Label variables cannot be operands of any operators other than the comparison operators equal(=) and not equal(¬=).

A label variable may have only data of type label assigned to it. Data of type label may not be converted to any other type.

Example

```
DECLARE LBL LABEL(X(2), Y);

          .
          .
          .
X(1) : IF G = Y THEN GO TO EXIT;
       IF G < Y THEN LBL = X(2);
                ELSE LBL = Y;
       GO TO LBL;
          .
          .
          .
X(2) : A = A + B + C + D;
          .
          .
          .
Y : X(3) : A= A+ 10;
          .
          .
          .
GO TO X(I);
          .
          .
          .
EXIT : RETURN;
```

X is a label constant array.

Y and EXIT are label constants.

Y and X(3) have the same value.

LBL is a label variable that can take on the values of the label constants X(2) and Y.

4.6.2 Format Attribute

The <format-attribute> specifies that the identifier will have format constants as values.

Syntax

<format-designator> ::= FORMAT

Semantics

The format variable may have other format variables or format constants assigned to it. When the assignment is made, the environment of the source format is assigned to the target.

A format constant is the <statement-label-constant> of a FORMAT statement. See Section 7.3.6.4, The Format Statement.

Restrictions:

Format variables cannot be operands of any operators other than the comparison operators equal (=) and not equal (¬=).

A format variable may only have data of type format assigned to it. Data of type format may not be converted to any other type.

Example

```
DECLARE (FV, FVA(10)) FORMAT VARIABLE, X FIXED;
        FC: FORMAT (F(10));
        FCA(1) : FORMAT(X(2), F(10,2));
        FCA(2) : FORMAT(X(5), F(10,5));
FV = FC;
FVA(2) = FCA(1);
        .
        .
        .
PUT EDIT(X) (R(FVA(2)));
PUT EDIT(X,X) (X(1), R(FC), R(FCA(1)));

FV is a format variable
FVA is a format variable array
FC is a format constant
FCA is a format constant array
```

4.6.3 Locator Attributes

The <locator-attribute> describes locator variables.

Syntax

<locator-attribute> ::=

<pointer-attribute> | <offset-attribute>

<pointer-attribute> ::= POINTER

<offset-attribute> ::= OFFSET [(<area-reference>)]

Semantics

A locator variable can be used as a based variable reference to identify a particular generation of the based variable (See Section 10.4.2.4, The Based Storage Class).

Offset variables identify a location relative to the start of an area.

Pointer variables identify any location, including those within an area.

The <area-reference> in the <offset-attribute> specification is optional and indicates the area in which the generation of the based variable will be allocated. (See section 4.6.4, In Attribute.)

The value of an offset variable or function identifies the position of a generation within an area relative to the area. This value

may be converted to a pointer to the generation by supplying the area and the offset value as arguments to the POINTER built-in function. A value of offset type may be obtained from the NULLO built-in function and OFFSET built-in function.

The value of a locator variable can be set in any of the following ways:

1. With the <set-option> of a READ statement.
2. By a LOCATE statement.
3. By an ALLOCATE statement.
4. By assignment of the value of a locator variable or function.

When an offset value is assigned to an offset variable, the area variables named in the IN attribute specifications are ignored.

Restrictions:

An offset variable must be explicitly declared.

Locator variables cannot be operands of any operators other than the comparison operators equal (=) and not equal (¬=).

Locator data cannot be converted to any other data type, but pointer and offset variables may be converted to each other.

A locator value can be assigned only to a locator variable.

Locator data cannot be transmitted using STREAM data transmission.

Assumptions:

An undeclared identifier appearing in the <based-attribute> specification, in a <set-option>, or as a locator qualifier, is contextually declared to be a pointer variable.

An undeclared identifier appearing in the <offset-attribute> specification is contextually declared to be an area variable.

A variable appearing in the <area-reference> is given the <area-attribute>.

4.6.3.1 Locator Qualification

Locator qualification is used to associate a pointer or offset value with a based variable to identify a particular generation of data.

Syntax

<locator-qualifier> ::= <scalar-locator-expression> ->

[<based-locator-variable> ->]...<based-variable>

<scalar-locator-expression> ::=

<pointer-expression> |

<offset-expression>

Semantics

Locator qualification is used to indicate the generation of a based variable to which the associated reference applies.

If an offset expression or an offset variable is used as a locator qualifier, its value is implicitly converted to a pointer value.

More than one locator qualifier can be specified in a reference.

If more than one qualifier is used, they are read from left to right.

Restrictions:

If more than one locator qualifier is specified in a reference, only the first can be a function reference. All other locator qualifiers must themselves be based variables.

Assumptions:

If a based variable is referenced without a locator qualifier, the reference is the same as a reference qualified by the locator variable declared with the based variable in the BASED attribute specification.

Examples

1. A = P->B;

2. A = P->Q->B;

3. A = ADDR(X)->B;

Example 1 causes the value of B in the generation pointed to by P, to be assigned to A. Example 2 specifies that the value of P is to be used to locate the generation of Q, which locates the specific generation of B to be assigned to A. In Example 3, the generation of B is derived from the location of the variable X and assigned to A.

4.6.4 In Attribute

The <in-attribute> is used to specify the area in which the generation of a based variable will be allocated.

Syntax

<in-attribute> ::= [IN] (<area-reference>)

Restrictions:

If the keyword IN is omitted, the <in-attribute> must immediately follow the keyword OFFSET.

4.6.5 Area Attribute

The <area-attribute> defines storage that, following allocation, is to be reserved for the allocation of based variables.

Syntax

<area-attribute> ::= AREA [<size-attribute>]

<size-attribute> ::= [SIZE]

([<expression> [<refer-option>]) |

<asterisk>)

Semantics

The <size-attribute> specifies the number of words that an area variable will require.

The size of an area may be specified as an integer constant, an * or an expression with an optional refer specification.

The area size for areas that are not of static storage class may be an expression which is converted to an integer when the area is allocated. It is used to indicate the number of words of storage to be reserved.

An <asterisk> may be used to specify the size of an area parameter. It indicates that the actual size of the area is the size of the associated area argument.

Areas can be allocated into and freed from based variables by naming the area variable in the <in-option> of the ALLOCATE statement and FREE statement. Storage that has been freed can be subsequently reallocated to a based variable.

An <area-expression> is either a reference to a variable with the AREA attribute or a reference to a function returning data of type area.

Area data may be transmitted using RECORD data transmission and maintain its validity.

During execution, the state of the storage allocated for an area depends only on the allocations made and freed in the area. It does not depend on the size of the area. This state is represented by the significant allocations made in this area. When an area is allocated, it contains no significant allocations. Its value is identical to the EMPTY built-in function. An allocation "A" made in an area is significant at some given time if it has not been freed by that time. If it has been freed by that time, it is significant only if a subsequent allocation was made before "A" was freed.

Restrictions:

If the keyword SIZE is omitted the <size-attribute>, if it appears, must immediately follow the keyword AREA.

The size for areas of static storage class must be specified as an integer constant.

Data of type area cannot be converted to any other type. An area can be assigned only to an area variable.

No operators may be applied to area variables.

Only the <initial-call> form of the INITIAL attribute is allowed with area variables.

An area may have the DEFINED attribute. Only simple and iSUB defining are allowed. The base must have the same size as the defined area.

Area data cannot be transmitted by STREAM data transmission.

The AREA condition is raised if an attempt is made to allocate a based variable in an area that does not contain sufficient free storage for that allocation.

Assumptions:

If the <size-attribute> is omitted, default is 256 words.

An area variable can be contextually declared by its appearance in an <in-attribute> specification or in an <in-option>.

4.7 ENTRY DATA DESCRIPTION

Syntax

```
<entry> ::=

    <entry-attribute> |

    <generic-attribute> |

    <returns-attribute> |

    <builtin-attribute> |

    <parameter-attribute>
```

4.7.1 Entry Attribute

The ENTRY attribute specifies that the identifier being declared is an entry-label variable. If an entry-label constant is being declared, the attributes of the parameters of the entry point (if any) must be declared.

Syntax

```
<entry-attribute> ::=

    ENTRY [(<parameter-attribute-list>

    [, <parameter-attribute-list>]...)]
```

Semantics

The <entry-attribute> with the associated <parameter-attribute-list> must be declared for any entry variable name that is invoked within the block.

The term entry name is applied to names that are explicitly declared with the ENTRY attribute and to names that receive the ENTRY attribute contextually or by implication.

The ENTRY attribute should be declared for internal entry constants.

The following apply to <parameter-attribute-list>s:

1. Each <parameter-attribute-list> describes the attributes of a single parameter. The parameter name is not listed.

2. The <parameter-attribute-list>s must appear in the same order as the associated parameters. If the attribute of any parameter need not be described, the absence of the corresponding <parameter-attribute-list> must be indicated by a comma.

3. If the parameter is a structure, the level number must precede the attributes for each level.

4. Attributes may appear in any order, except for an array parameter, where the dimension attribute must be the first attribute specified for that parameter.

Restrictions:

The ENTRY attribute must appear for each entry name used to invoke an external function.

The ENTRY attribute must be specified or implied for an entry name that is a parameter.

The ENTRY attribute cannot be specified with the BUILTIN attribute.

Lengths, sizes and dimensions in <parameter-attribute-list>s may only be specified by unsigned integer constants or asterisks.

Factoring of attributes is not permitted within <parameter-attribute-list>s of an <entry-attribute> specification.

Assumptions:

The ENTRY attribute can be assumed either contextually or by implication. The appearance of a name as a label prefix of either a PROCEDURE statement or an ENTRY statement constitutes an explicit declaration of that identifier as an entry name.

If there is no <parameter-attribute-list>, the ENTRY attribute is implied by the RETURNS attribute.

No defaults are applied for <parameter-attribute-list>s unless attributes and/or level numbers are specified. If only a level number and/or the DIMENSION attribute is specified for a <parameter-attribute-list>, FLOAT, DECIMAL and REAL are assumed.

4.7.2 Generic Entry Name

The <generic-attribute> specifies that the identifier has been declared as a generic entry name. A reference to the identifier refers to the entry expression whose argument list matches the parameter list in the generic reference.

Syntax

```
<generic-attributes> ::=GENERIC (<generic-element>
                                 [,<generic-element>]...)

<generic-element> ::= <entry-expressions> WHEN
                      (<generic-attribute-list>)

<generic-attribute-list> ::= ALIGNED | AREA | BIT |
                             BINARY [<precision>] |
                             CHARACTER | DECIMAL |
                             ENTRY [<description-list>] |
                             FILE | FORMAT | EMPTY
                             FIXED [<precision>] |
                             FLOAT [<precision>] |
                             LABEL | OFFSET |
                             PICTURE <picture-specification> |
                             POINTER | UNALIGNED |
                             PRECISION [<precision>] |
                             REAL [<precision>] |
                             RETURNS [<attribute-list>]
```

```
<description-list> ::= <parameter-attribute-list>
                       [,<parameter-attribute-list>]...
```

Semantics

The only attribute which may appear with GENERIC is INTERNAL.

The selected <entry-expression> for a given generic reference is the reference part of one of the generic elements given in the declaration of the generic name.

The actual selection depends on the number of arguments, their attributes, and the order in which the generic elements are written.

For a generic reference with no arguments, the first generic element with an empty generic attribute list will be selected.

Attributes not specified in the generic attribute list will be supplied according to the default rules.

Selection of an entry expression according to the argument attributes requires that all the attributes specified for a generic element be contained in the attributes of the arguments.

4.7.3 Builtin Attribute

The <builtin-attribute> specifies that any reference to the associated name within the scope of the declaration is to be interpreted as a reference to the built-in function or pseudo-variable of the same name.

Syntax

```
<builtin-attribute> ::= BUILTIN
```

Semantics

The BUILTIN attribute is used to refer to a built-in function or pseudo-variable in a block that is contained in another block in which the same identifier has been declared to have another meaning.

The BUILTIN attribute must be specified to identify a builtin function that has no <parameter-attribute-list>.

Restrictions:

The BUILTIN attribute cannot be declared for parameters.

If the BUILTIN attribute is declared for an entry name, the entry name can have no other attributes except INTERNAL.

Assumption:

The default <scope-attribute> is INTERNAL for any item with the BUILTIN attribute.

4.7.4 Returns Attribute

The <returns-attribute> may be specified in a DECLARE statement for an entry name that is used as a function reference within the scope of the declaration.

Syntax

<returns-attribute> ::= RETURNS (<attribute-list>);

Semantics

The <returns-attribute> specifies the attributes of the function value returned when the entry name is invoked as a function.

The attributes in the <attribute-list> must agree with the attributes specified explicitly or by default in the PROCEDURE statement or ENTRY statement to which the entry name is prefixed.

The <length-attribute> specifications are evaluated on entry to the block containing the RETURNS attribute specification. Such evaluated <returns-attribute>s form part of the environment of blocks contained within the block declaring the attribute and dynamically descendant from the block.

Restrictions:

For an internal function, the <returns-attribute> can be specified only in a DECLARE statement that is internal to the same block as the function procedure.

Only arithmetic, string, locator, picture, aligned and unaligned attributes can be specified in the attribute list.

Assumptions:

If an entry name has no <returns-attribute> specification, a RETURNS attribute is assumed.

If the entry name begins with any of the letters I thru N, the defaults are REAL, FIXED, BINARY with default precision. If the entry name begins with any other letter, the defaults are REAL, FLOAT, DECIMAL with default precision.

4.7.5 Parameter Attribute

The <parameter-attribute> denotes a data item that is a parameter to an entry point of a procedure.

Syntax

<parameter-attribute> ::= [PARAMETER]

Semantics

This attribute is explicitly specified for a variable by the appearance of the name of that variable in a <parameter-list> and does not need to be specified by the programmer.

4.7.6 Library Attribute

A library attribute specifies that the identifier being declared is a <library-identifier>.

Syntax

```
<library-attribute> ::=

     LIBRARY [(<system-library-attribute-list>)]

<system-library-attribute-list> ::=

     <system-library-attribute>
     [,<system-library-attribute>]...

<system-library-attribute> ::=

     INTNAME = <char-string-constant>
     TITLE = <char-string-constant>
     LIBPARAMETER = <char-string-constant>
```

Semantics

The TITLE, INTNAME and LIBPARAMETER attributes are EBCDIC string-type attributes. The INTNAME is the internal identifier for the library. The TITLE is the code file name of the library. The LIBPARAMETER transmits information from the user program to the selection procedures of library programs that provide dynamic entry point linkage.

The character strings specified for system library attributes must be constants. Specification of these attributes is optional; by default, the identifier being declared is used for the TITLE and the INTNAME. If the INTNAME is specified but the TITLE is not, the string specified for the INTNAME is used for the TITLE.

4.7.7 Options Attribute

The OPTIONS attribute defines an entry as an external library entry point.

Syntax

```
<options-attribute> ::=

     ENVIRONMENT/OPTIONS (LIBRARY=<library-identifier>

     [,ACTUALNAME=<char-string-constant>])
```

Semantics

Use of an OPTIONS list that includes LIBRARY or ACTUALNAME implies the ENTRY and CONSTANT attributes for the identifier being declared. LIBRARY and ACTUALNAME are mutually exclusive with file attributes; this use implies that a file is being declared. The library must be named; the declared <library identifier> is used by default. The specification of an ACTUALNAME is optional.

Example

In this example, calling the entry DO-THIS from the PL/I program causes the MCP to link the program to a library called YOUR-LIBRARY with an entry point called DO-THAT.

```
DECLARE LIBI LIBRARY
     (TITLE = "MATH/LIBRARY");

DECLARE MY LIBRARY LIBRARY
     (INTNAME = "YOUR-LIBRARY");

DECLARE DO-THIS ENTRY (BINARY FIXED, CHAR(*))
     RETURNS (CHARACTER(100) VARYING)
     OPTIONS (LIBRARY = MY-LIBRARY,
              ACTUALNAME = "DO-THAT");
```

4.8 FILE DATA DESCRIPTION

Syntax

```
<file> ::=

     <attribute> |

     <function-attribute> |

     <transmission-attribute> |

     <print-attribute> |

     <key-attribute> |

     <access-attribute> |

     <environment-attribute>
```

4.8.1 File Attribute

The <file-attribute> specifies that the identifier being declared is a file name.

Syntax

<file-attribute> ::= FILE

Assumptions:

The <file-attribute> can be implied by any of the other file description attributes.

An identifier may be contextually declared with the <file-attribute> through its appearance in the <file-option> of any data transmission statement or in an ON statement for any data transmission condition.

The <scope-attribute> EXTERNAL is assumed for files that are not parameters.

4.8.2 Function Attribute

The <function-attribute> indicates the function of a file.

Syntax

<function-attribute> ::= INPUT | OUTPUT | UPDATE

Semantics

INPUT specifies that data is to be transmitted from external storage to the program.

OUTPUT specifies that a new data set is to be created to which data is to be transmitted from the program to external storage.

UPDATE specifies that the data can be transmitted in either direction. The file is both an input and an output file.

A file with the UPDATE attribute and the SEQUENTIAL attribute indicates an update-in-place mode. To access such a file the sequence of statements must be READ and REWRITE.

Restrictions:

A file with the UPDATE attribute cannot have the STREAM attribute, or the PRINT attribute.

A file with the INPUT attribute cannot have the PRINT attribute.

Assumptions:

The default <function-attribute> is INPUT.

The PRINT attribute implies the <function-attribute> OUTPUT.

If a file is opened implicitly by a PUT, LOCATE or WRITE statement, the <function-attribute> OUTPUT is assumed.

If a file is opened implicitly by a GET or READ statement, the <function-attribute> INPUT is assumed.

If a file is opened implicitly by a DELETE or REWRITE statement, the <function-attribute> UPDATE is assumed.

4.8.3 Transmission Attribute

The <transmission-attribute> specifies the kind of data transmission to be used for the file.

Syntax

<transmission-attribute> ::= RECORD | STREAM

Semantics

STREAM indicates that the data of the file is considered to be a continuous stream of data items, in character form, to be assigned from the stream to variables, or from expressions into the stream.

RECORD indicates that the file consists of a collection of physically separate records, each of which consists of one or more data items in any form. Each record is transmitted as an entity to or from a variable or buffer.

A file with the RECORD attribute can be specified only in the OPEN, CLOSE, READ, WRITE, REWRITE, LOCATE and DELETE data transmission statements.

A file with the STREAM attribute can be specified in the OPEN, CLOSE, GET, and PUT data transmission statements.

Restrictions:

A file with the STREAM attribute cannot have any of the following attributes: UPDATE, DIRECT, SEQUENTIAL and KEYED, any of which implies RECORD.

A file with the RECORD attribute cannot have the PRINT attribute.

Assumptions:

If a file is implicitly opened by a READ, WRITE, REWRITE, LOCATE, or DELETE statement, the default <transmission-attribute> is RECORD.

4.8.4 Print Attribute

The <print-attribute> specifies that the data of the file is ultimately to be printed.

Syntax

<print-attribute> ::= PRINT

Semantics

The PAGE and LINE options of the PUT statement and the PAGESIZE option of the OPEN statement can be used only with files having the PRINT attribute.

Restrictions:

The PRINT attribute conflicts with the RECORD attribute.

Assumptions:

The PRINT attribute implies the OUTPUT and STREAM attributes.

4.8.5 Keyed Attribute

The <key-attribute> specifies that the options KEY, KEYTO and KEYFROM may be used to access records in the file. These options indicate that keys are involved in accessing the records in the file.

Syntax

<key-attribute> ::= KEYED

Semantics

The KEYED attribute must be specified for every file which uses any of the options KEY, KEYTO or KEYFROM.

See Section 4.8.7, Environment Attribute for additional information concerning keyed files.

The KEYED attribute can be specified only for files with the RECORD attribute.

Restrictions:

A file with the KEYED attribute cannot have the STREAM or PRINT attributes.

4.8.6 Access Attribute

The <access-attribute> specifies the manner in which the records of a file with the RECORD attribute are to be accessed.

Syntax:

<access-attribute> ::= SEQUENTIAL | DIRECT

Semantics:

SEQUENTIAL specifies that the records are to be accessed according to their logical sequence in the data set.

DIRECT specifies that the records of the file are to be accessed by use of a key. Each record of a direct file must, therefore, have a key associated with it. The RECORD attribute is implied by either the DIRECT or KEYED attribute.

Files with the DIRECT attribute must also have the KEYED attribute which is implied by the DIRECT attribute.

Restrictions:

The DIRECT and SEQUENTIAL attributes cannot be specified with the STREAM attribute.

Assumptions:

Default is SEQUENTIAL for files with the RECORD attribute.

If a file is implicitly opened by a LOCATE statement, SEQUENTIAL is assumed.

4.8.7 Environment Attribute

The <environment-attribute> specifies the characteristics of a file on the system.

Syntax

<environment-attribute> ::=

{ ENVIRONMENT | OPTIONS }

(<system-file-attribute> = <constant-expression>

[,<system-file-attribute> = <constant-expression>]...)

For a complete discussion of <system-file-attribute>, see the FILE MANAGEMENT MANUAL.

The keyed, update, and indexed sequential (ISAM) input-output features of PL/I allow the following:

1. Direct keyed reads.
2. Adding of keyed records.
3. Reading of next keyed record in sequence (ISAM read).
4. Rewriting keyed record just read.
5. Overwriting a keyed record.
6. Deleting keyed records.

Keyed files may be ordered in ascending or descending sequence. Records may be physically or logically deleted. Duplicate keys may be optionally allowed. A variety of key types are allowed. A keyed file is declared by:

1. Declaring the file to have the "KEYED" and/or "DIRECT" attribute.
2. Specifying the file attributes AREAS and AREASIZE, which will define prime data area.
3. Specifying the desired following keyed options:

KEYLENGTH Length of key in 8-bit bytes.

KEYORDER — Sequence of keys: ascending or descending; default is ascending.

KEYSTART — Starting byte number (1-relative) of the key embedded in the record; if 0, the key is not part of the record.

FILEOVERFLOW — The number of areas allocated for the records for area allocated for overflow space; default is zero.

KEYSPERENTRY — The "fineness" of the fine table, i.e., the number of records represented by one fine table entry; default is one.

NODUPLICATES — If specified, duplicate keyed records are errors, otherwise they are allowed on a "first-in-first-out" basis

PHYSICALUPDATE — If specified, causes all modified tables and buffers to be rewritten to disk following each ISAM operation.

WAITUPDATEIO — If specified, causes the intrinsics to immediatly wait for the completion of the write operations of the PHYSICALUPDATE. Otherwise, the wait is performed at the beginning of the next ISAM operation.

SAVEDELETIONS — If specified, deleted records are "logically deleted", but are available on sequential reads. The first byte of record will be flagged with a 4"FF", indicating deletion. The table entry is removed and the record is "seen" only on sequential reads.

KEYTYPE — Mode of key:

0 = BINARY
1 = 8-bit characters
2 = 8-bit unsigned numeric
3 = 8-bit MSD signed numeric (sign in zone of most significant byte)
4 = 8-bit LSD signed numeric
5 = 4-bit characters
6 = 4-bit unsigned numeric
7 = 4-bit MSD signed numeric
8 = 4-bit LSD signed numeric
Default is 8-bit characters (KEYTYPE = 1).

A user may "rewind" the keyed file and read the first record by performing a read-key, with a key of all zeros. The LOW built-in function may be used for this special key.

Keyed options may not appear as OPEN statement options, file attribute assignments or be label equated.

KEYLENGTH and KEYSTART are the only required keyed options.

Examples

```
DECLARE K FILE KEYED ENVIRONMENT(KEYLENGTH=4, KEYSTART=1,
     AREAS = 2, AREASIZE=10);

DECLARE KY FILE DIRECT KEYED
ENVIRONMENT(AREAS=10, AREASIZE=15, KEYORDER='DESCENDING',
     KEYSPERENTRY=4, FILEOVERFLOW=4, AREAOVERFLOW=2,
     NODUPLICATES, SAVEDELETIONS, KEYLENGTH=10,
     KEYSTART=0);
```

The following I/O statements can be used to manipulate a keyed file:

READ FILE (<file-exp>) INTO (<variable>);

READ FILE (<file-exp>) INTO (<variable>) KEY (<scalar-exp>);

READ FILE (<file-exp>) INTO (<variable>) KEYTO (<scalar-ref>);

READ FILE (<file-exp>) SET (<scalar-ptr-var>);

READ FILE (<file-exp>) SET (<scalar-ptr-variable>) KEY (<scalar-exp>);

READ FILE (<file-exp>) SET (<scalar-ptr-variable>) KEYTO (<scalar-exp>);

WRITE FILE (<file-exp>) FROM (<variable>) KEYFROM (<scalar-exp>);

DELETE FILE (<file-exp>) [KEY (<scalar-exp>);

REWRITE FILE (<file-exp>) [KEY <scalar-exp>] FROM (<variable>);

LOCATE FILE (<file-exp>) [SET(<scalar-ptr-variable>)] KEYFROM (<scalar-exp>);

By using the WRITE-KEYFROM statement, a keyed file may be created, in ascending or descending sequence.

The file may be read directly (READ-KEY statement) or sequentially (READ-INTO, READ-SET).

By opening the file update, records may be added (WRITE-KEYFROM), deleted (DELETE statement) or overwritten (REWRITE-KEY statement). Additionally, LOCATE-MODE I/O may be performed on a keyed file (LOCATE-KEYFROM, READ-SET).

All keys are passed to the keyed intrinsics as B6700 hardware pointers with an associated 8-bit character length. Hence, a binary key is passed with a length of 6 (one-word), a pic 'HHHHHS' is passed with a length of three and character (50) is passed with a length of 50.

If abnormal errors arise during file creation or access, the key condition will be raised, and the built-in ONCODE will provide the error code. See Appendix 2 for the list of ONCODES. Variable length records are not allowed for keyed files.

Inside a file option or environment declaration, a system file attribute has one of five specifications:

1. NUMERIC Requires a decimal integer-constant greater than zero specification, e.g., MAXRECSIZE = 10, AREAS =

2....

2. MNEMONIC — Requires a character-string mnemonic enclosed in single quotes ,e.g., KIND = 'DISK', MYUSE = 'OUT'....

3. STRING — Requires a character string enclosed in single quotes ,e.g., FORMMESSAGE = 'USE FORM 1040'....

4. TITLE — Is a string valued attribute that has some syntax restrictions. subtitles with slashes must be enclosed in double quotes and double quotes may only appear in pairs enclosing an entire subtitle, e.g., TITLE = 'A/"B/C"/D'

5. NULL — Is valid only for attributes with the mnemonics of 'TRUE' and 'FALSE'. The attribute will be set to TRUE. e.g. , OPTIONAL.... is the same as OPTIONAL = 'TRUE'....

4.9 ARRAY DATA DESCRIPTION

Syntax

<array> ::=

<dimension-attribute>

4.9.1 Dimension Attribute

The <dimension-attribute> specifies the number of dimensions of an array and the bounds of each dimension.

Syntax

<dimension-attribute> ::=

[DIMENSION] (<bound> [, <bound>]...)

<bound> ::=

{[<expression>[<refer-option>]:]

{<expression>[<refer-option>]}) | <asterisk>

Semantics

The <dimension-attribute> either specifies the bounds (only the upper bound or both the upper and the lower bounds) or indicates, by the use of an asterisk, that the actual bounds of the array parameter are to be the bounds of its associated array arguments.

The number of bounds specifications indicates the number of dimensions in the array unless the variable being declared is contained in a structure array, in which case it inherits dimensions from the containing structure.

Bounds that are expressions are known as adjustable bounds and are evaluated and converted to integer data when storage is allocated for the array.

An <asterisk> specifies that the actual bounds of the array parameter are to be the bounds of its associated array argument.

The <refer-option> can be used to specify a bound of a variable with the BASED attribute.

Restrictions:

If the keyword DIMENSION is missing, the <dimension-attribute> must be the first attribute to follow the array name (or the parenthesized list of names, if it is being factored) in the declaration where it appears.

On allocation of storage, the lower bound must be less than or equal to the upper bound.

The bounds of arrays with the STATIC attribute must be optionally signed integer constants.

For parameters, bounds can be only optionally signed integer constants or asterisks.

The <refer-option> may only be used to specify the bound of an array with the <storage-attribute> BASED.

Assumptions:

If only the upper bound is given, the lower bound is assumed to be one (1).

4.10 STRUCTURE DATA DESCRIPTION

Syntax

```
<structure> ::=

        <structure-attribute> |

        <member-attribute> |

        <like-attribute>
```

4.10.1 Structure Attribute

The <structure-attribute> is given by the compiler to any item which contains items with the MEMBER attribute.

Syntax

<structure-attribute> ::= [STRUCTURE]

Assumptions:

The <structure-attribute> is an attribute supplied by the compiler and need not be specified by the programmer.

4.10.2 Member Attribute

The <member-attribute> is given to all elements of a structure, except the structure name itself.

Syntax

<member-attribute> ::= [MEMBER]

Assumptions:

The <member-attribute> is an attribute supplied by the compiler and need not be specified by the programmer.

4.10.3 Like Attribute

The <like-attribute> specifies that the name being declared is a structure variable with the same structuring as that for the name following the attribute keyword LIKE. Substructure names, elementary names and attributes for substructure names and elementary names are to be identical.

Syntax

<like-attribute> ::= LIKE <structure-variable>

Semantics

The <structure-variable> can be a major structure name or a minor structure name. It can be a qualified name, but it cannot be subscripted.

The <structure-variable> must be known in the block containing the <like-attribute> specification. The structure names in all <like-attribute>s are associated with declared structures before any <like-attribute>s are expanded.

Attributes of the <structure-variable> itself do not carry over to the created structure. For example, storage class attributes do not carry over. If the <structure-variable> represents a structure array, its <dimension-attribute> specification is not carried over. The only <aligned-attribute>s carried over are those explicitly specified for substructures and elements of the structure variable. The <like-attribute> specification is expanded before the <aligned-attribute>s are applied to the contained elements of the <structure-variable>. The other attributes of the substructure names and elementary names are carried over. If the attributes that are carried over contain names, these names are interpreted in the block containing the LIKE attribute.

If a direct application of the description to the structure declared LIKE would cause an incorrect continuity of level numbers (for example, if a minor structure at level 3 were declared LIKE a major structure at level 1), the level numbers are modified by a constant before application.

Restrictions:

Neither additional substructures nor elementary names can be added to the created structure. Any level number that immediately follows the <structure-variable> in the <like-attribute> in a DECLARE statement must be algebraically equal to or less than the level number of the name declared with the <like-attribute>.

Neither the <structure-variable> nor any of its substructures can be declared with the LIKE attribute, nor may the <structure-variable> have been completed by a <like-attribute> specification.

Example

```
DECLARE 1 A, 2 C, 3 E, 3 F,
        1 D, 2 C, 3 G, 3 H;
      .
      .
      .
BEGIN
  DECLARE 1 A LIKE D, 1 B LIKE A.C;
      .
      .
      .
END;
```

These declarations result in the following:

```
1 A LIKE D is expanded to give:
     1 A, 2 C, 3 G, 3 H

1 B LIKE A.C is expanded to give:
     1 B, 3 E, 3 F
```

4.11 STORAGE CLASS

The <storage-class> attributes are used to specify the type of storage allocation to be used for data variables.

Syntax

```
<storage-class> ::=

     STATIC |

     AUTOMATIC |

     CONTROLLED |

     BASED [(<scalar-locator-expression>)]
```

Semantics

AUTOMATIC specifies that storage is to be allocated upon each entry to the block to which the storage declaration is internal. The storage is released upon exit from the block. If the block is a procedure that is invoked recursively, the previously allocated storage is "pushed down" upon entry. The latest allocation of storage is "popped up" upon termination of each generation of the recursive procedure.

STATIC specifies that storage is to be allocated when the program is loaded and is not to be released until program execution has been completed.

CONTROLLED specifies that full control will be maintained by the programmer over the allocation and freeing of storage by means of the ALLOCATE and FREE statements. Multiple allocations of the same controlled variable without intervening freeing, will cause stacking of generations of the variable.

BASED, like CONTROLLED, specifies that full control will be maintained by the programmer. However, the separate generations are not stacked. Each may be accessed by a pointer value that identifies the generation and is used as a <locator-qualifier> applied to the based variable. A based variable can be used to identify data of any storage class by associating the based variable name with a locator qualifier that points to that data. Based variables can be allocated and freed by use of the ALLOCATE and FREE statements. Such allocations are not stacked. Any generation is available as long as it remains in an allocated state.

If the INITIAL attribute is specified for a based variable, the values are used only upon explicit allocation of the based variable with an ALLOCATE statement or LOCATE statement. If the <refer-option> appears in a structure for which any element has the INITIAL attribute, initialization specified by the INITIAL attribute is done after contained variables named in all <refer-option>s have been assigned their proper values.

AUTOMATIC and BASED variables may only have the <scope-attribute> INTERNAL. STATIC and CONTROLLED variables may have INTERNAL or EXTERNAL scope.

The following rules govern the use of based variables:

1. If no locator expression is specified in the BASED attribute specification, any reference to the based variable must have an explicit <locator-qualifier>. This does not apply to a based variable that is the object of a <refer-option> or that is to be allocated through the use of an ALLOCATE statement or LOCATE statement.

2. A reference to a based variable without an explicit <locator-qualifier> is impiicitly qualified by the locator expression in the BASED attribute specification in the DECLARE statement for that based variable. Identifiers in this implicit qualification are those of the names in the declaring block. Expressions occurring in the implicit qualifier are evaluated in the current environment of the declaring block, with enabling of

conditions as exists at block entry.

Example

```
DECLARE B BASED (P(I)),
        P(3) POINTER;
      .
      .
      .
BEGIN;
DECLARE P POINTER, I;
      .
      .
      .
L : B = X;
```

The statement "B = X" has the same effect as P(I) -> B=X. P and I are the names known in the outer block, not those declared in the begin block. Conditions enabled at L are used when P(I) is evaluated.

3. When a reference is made to a based variable, the data attributes assumed are those of the based variable, while the associated locator variable identifies the generation of data. If the reference is to a component of a based structure, a second, temporary variable is created to determine the location of the component in relation to the beginning of the structure.

4. Array bounds, area sizes, and string lengths declared with the based variable are evaluated dynamically with each reference to the based variable. Therefore the <asterisk> notation for dimensions, area sizes, and lengths is not permitted. A reference to a component of a based structure causes evaluation of sufficient elements of the structure to determine the position of the component.

5. When a based variable is allocated using the ALLOCATE statement or LOCATE statement, expressions for bounds, lengths, and sizes are evaluated at the time of allocation.

6. A based variable cannot appear in a data-directed data list.

7. Whenever a based variable containing arrays, strings, or areas is passed as an argument, dimensions, lengths, and sizes are determined at the time the argument is passed and remain fixed throughout execution of the invoked block.

8. The <refer-option> can be used to create structures that contain self-defining data. It may be used in a DECLARE statement to specify a bound of an array, the length of a string, or the size of an area.

9. The <scalar-name> is a reference, possibly qualified, but not subscripted or locator qualified. The reference must be to a scalar item preceding the <refer-option> in the structure.

10. Upon allocation of a structure containing one or more <refer-option>s, all expressions for bounds, string lengths, and area sizes are evaluated (in any order). A new generation of the structure is then allocated, and the relevant locator variable is assigned a value to identify this generation. If any <refer-option>s exist, initialization is then done (in any order) for the new generation of variables that are objects of the <refer-option>s, using the value obtained for each from the expression which precedes the option.

11. In a reference specifying some generation of a based variable, some of whose bounds, lengths, and sizes are specified by <refer-option>s, the values are taken from the variables in the generation which are objects of the <refer-option>s.

12. The <scalar-name> that is the object of the <refer-option> differs from other based variables in that when a reference is made to it, the implied pointer from the based variable is not used, but the reference is always to that generation of the structure that is currently being accessed or allocated.

Restrictions:

A <storage-class> may not be specified for entry names, file names, members of structures, or defined data items.

A <storage-class> may not be specified for parameters.

Variables declared with adjustable lengths and dimensions may not have the STATIC attribute.

Only level one items may be allocated, freed and given a <storage-class>. In particular, storage is always allocated for a complete major structure, and the contained elements may not be independently allocated and freed.

The EXTERNAL attribute may not be specified for a based variable.

Assumptions:

If no <storage-class> is specified and the scope is internal, AUTOMATIC is assumed.

If no <storage-class> is specified and the scope is external, STATIC is assumed.

If neither the <storage-class> nor the <scope-attribute> is specified, AUTOMATIC is assumed.

An undeclared identifier appearing in parentheses following the keyword BASED in the <based-attribute> specification is contextually declared with the POINTER and AUTOMATIC attributes.

4.12 SCOPE ATTRIBUTE

The <scope-attribute> specifies the scope of a name.

Syntax

<scope-attribute> ::= EXTERNAL | INTERNAL

Semantics

INTERNAL specifies that the name can be known only in the declaring block and its contained blocks.

EXTERNAL specifies that the name may be known in other blocks containing an external declaration of the same name.

Assumptions:

INTERNAL is assumed for entry names of internal procedures and for variables with any storage class.

EXTERNAL is assumed for file names and entry names of external procedures.

Programmer defined condition names are assumed to be EXTERNAL.

4.13 DATA ATTRIBUTES

Syntax

<data-attribute> |
<initial-attribute> |
<variable-attribute> |
<defined-attribute>

4.13.1 Alignment Attribute

The <alignment-attribute> specifies the arrangement of data elements in storage to provide speed of access or storage economy.

Syntax

<alignment-attribute> ::= ALIGNED | UNALIGNED

Semantics

ALIGNED specifies that data elements should be arranged to provide speed of access.

UNALIGNED specifies that data elements should be arranged to provide storage economy.

The <alignment-attribute> is an element data attribute, but, syntactically, it may also be applied to any aggregate. This is semantically equivalent to the application of the attribute to all contained elements of the aggregate which are not explicitly declared with an <alignment-attribute>.

Application of the <alignment-attribute> to an aggregate affects the contained members, unless the member is specifically declared otherwise.

If the <alignment-attribute> of an argument in a procedure invocation does not match the attributes of the corresponding parameter, a dummy argument is created with the attributes specified in the parameter declaration, and the argument value is assigned to it.

Restrictions

For string overlay defining, all the base elements of the defined item must have the UNALIGNED attribute, as must those of the base item covered by the range of defining (i.e., from its beginning for a length equal to the length of the item plus the value of the starting position minus 1).

If a variable with the BASED attribute is used to access a generation of another variable, the <alignment-attribute> of the accessed variable and the based variable must agree.

For simple and iSUB defining, the <alignment-attribute> must agree between corresponding elements of the defined item and the base.

Assumptions

The <alignment-attribute> is applied by default at the base element level. The default for bit class and character class data is UNALIGNED. The default for all other types of data is ALIGNED.

For all operators and built-in functions, the default for the <alignment-attribute> is applicable to the elements of the result.

4.13.2 Initial Attribute

The <initial-attribute> specifies an initial value for a data item.

Syntax

<initial-attribute> ::= <initial-list> | <initial-call>

Semantics

The <initial-list> specifies an initial value to be assigned to a data item when storage is allocated to it.

The <initial-call> specifies that a procedure is to be invoked to perform initialization at allocation time.

Restrictions

The INITIAL attribute cannot be specified for entry names, file names, defined variables or structures. It can be given for an element of a structure.

4.13.2.1 Initial List

Syntax

<initial-list> ::= INITIAL (<item> [,<item>]...)

<item> ::= <value-item> | <iteration-specification>

<value-item> ::= <constant> | <scalar-variable>

<iteration-specification> ::= (<iteration-factor>)

(<value-item> | (<item> [,<item>]...))

<iteration-factor> ::= <scalar-expression>

Semantics

In this discussion the term "value" denotes one of the following:

[+ | -] arithmetic constant
arithmetic scalar variable
character string constant
character string variable
bit string constant
bit string variable

Only one value can be specified for an element variable. More than one value can be specified for an array variable. A structure variable can be initialized only by separate initialization of its elementary names, whether they are element or array variables.

Values specified for an array are assigned to successive elements of the array in row-major order (final subscript varying most rapidly).

If too many values are specified for an array, excess values are ignored. If not enough values are specified, the remainder of the array is not initialized.

Each item in the list can be a constant, a scalar variable, or an asterisk denoting no initialization for a particular element, or an

iteration specification.

The <iteration-factor> specifies the number of times the constant, scalar variable, item list, or asterisk is to be repeated in the initialization of elements of an array. If a <value-item> follows the <iteration-factor>, then the specified number of elements are to be initialized with that value. If an item-list follows the <iteration-factor>, then the list is to be repeated the specified number of times, with each item initializing an element of the array. If an asterisk follows the <iteration-factor>, then the specified number of elements are to be skipped in the initialization operation.

A negative or zero <iteration-factor> causes no initialization to occur.

For initialization of a string array, if only one constant parenthesized element expression precedes the constant string initial value, the expression is interpreted to be a string repetition factor for the string. It is interpreted as a part of the specification of the value for a single element of the array. This applies to constant string initial values only. Consequently, to cause initialization of more than one element of a string array, both the string repetition factor and the iteration factor must be explicitly stated, even if the string repetition factor is (1).

Example

```
((2)'A') is equivalent to ('AA') (for a single element)
((2) (1) 'A') is equivalent to ('A(, 'A') (for two elements)
```

Restrictions

Label constants given as initial values for label variables must be known within the block in which the label variable declarations occur. Label variables with the scope attribute STATIC cannot have the INITIAL attribute.

Locator or area variables may not be initialized with an <initial-list>.

Examples

```
DECLARE SWITCH BIT (1) INITIAL ('1'B);
DECLARE MAXVALUE INITIAL(99), MINVALUE INITIAL (-99);
DECLARE A(100,10) INITIAL ((920)0,(20)((3)5,9));
```

This example results in the following: each of the first 920 elements of A is set to 0; the next 80 elements consist of 20 repetitions of the sequence 5,5,5,9.

4.13.2.2 Initial Call

Syntax

<initial-call> ::= INITIAL CALL <entry-name> [(<argument-list>)]

Semantics

The <initial-call> form of the INITIAL attribute can be used to initialize data with the STATIC attribute.

The <initial-call> can be used to initialize locator and area variables .

Restrictions

The <entry-name> and <argument-list> passed must satisfy the condition stated for prologues in Section 10.1, Prologues.

Examples

```
DECLARE TABLE (20,20) INITIAL CALL INITIALIZE (X,Y);
```

INITIALIZE is the name of the procedure that sets the initial values of elements in TABLE. X and Y are arguments passed to intialize.

4.13.3 Variable Attribute

The <variable-attribute> is used with the ENTRY, FILE, and LABEL attributes to establish the name as a variable.

Syntax

<variable-attribute> ::= VARIABLE

4.13.4 Defined Attribute

The <defined-attribute> specifies that the level 1 scalar, array, or structure data is to occupy some or all of the storage assigned to the base item specified in the attribute.

Syntax

<defined-attribute> ::= DEFINED <base-item> [<position-attribute>]

<position-attribute> ::= POSITION (<integer-constant>)

Semantics

The defined-item:

1. Must be a level 1 identifier.

2. May be a scalar, array or structure.

3. Must not be declared with any of the attributes:

   INITIAL
   AUTOMATIC
   STATIC
   CONTROLLED
   BASED
   EXTERNAL

4. Is always internal.

5. Cannot be a parameter.

6. May not contain strings with the VARYING attribute.

The <base-item>:

1. May be declared within a structure at any level as a scalar (possibly subscripted), as an array, or as a structure.

2. May not have either the BASED or DEFINED attributes.

3. May not contain strings with the VARYING attribute.

The <position-attribute> may appear anywhere within the declaration of the level-one name of the defined item. If it is omitted, POSITION(1) is assumed. The number of bits or characters in the defined item, plus N-1, where N is the integer constant in the <position-attribute>, must not be greater than the number of bits or characters in the <base-item>.

Restrictions

The <position-attribute> may appear only with the DEFINED attribute in string overlay defining.

Restrictions applying to base-item and defined-item may be summarized in the following table:

	Defined-item	Base-item
Prohibited Attributes	INITIAL AUTOMATIC STATIC CONTROLLED BASED EXTERNAL	BASED DEFINED
Other Prohibitions	to be a parameter	
	level other than 1 in a structure	contain VARYING strings
	to contain VARYING strings	

In references to defined data, the bounds and string lengths of the defined data are used to determine whether the STRINGRANGE and SUBSCRIPTRANGE conditions occur.

There are three types of defining:

SIMPLE DEFINING
iSUB DEFINING
STRING OVERLAY DEFINING

The type of defining can be determined from the syntax of the program text according to the following rules:

1. If iSUB is specified, and POSITION is not specified then the type of defining is iSUB DEFINING.

2. If POSITION is specified and iSUB is not specified, then the type of defining is STRING OVERLAY DEFINING.

3. If POSITION is not specified and iSUB is not specified and the base-item "matches" the defined-item, the type of defining is SIMPLE DEFINING. If the base-item does not match the defined-item, the type of defining is STRING OVERLAY DEFINING.

Base item and defined item are considered matching if the base item , when passed as an argument, would match a parameter which had the attributes of the defined item (except for the DEFINED attribute itself). For this purpose, the parameter is assumed to have all bounds, string lengths and area sizes specified by asterisks.

A picture attribute can be matched only by a picture attribute that is identical except for the repetition factor, and presence or absence of a plus sign(+) in a scaling factor.

4.13.4.1 Simple Defining

Simple defining allows a (possibly subscripted) scalar, array, or structure item to be accessed by a different name. The attributes ALIGNED and UNALIGNED must agree between corresponding elements of the defined item and base item. Array bounds and string lengths associated with the defined item may differ from those of the base item, although they are subject to certain constraints given below.

1. Corresponding to any simple defined reference, there is an equivalent reference to the based item given in the DEFINED attribute of the defined item. The qualified name in this equivalent reference is the name of the base item. If the defined reference was qualified, the equivalent reference is further qualified by those identifiers in the declaration of the base item which correspond to the qualifying identifiers in the defined reference. If the base item names an array, the equivalent reference contains a subscript corresponding to each dimension in the array. The I-th subscript in the equivalent reference is the I-th subscript specified in the base item, unless an asterisk is specified for the base item in the DEFINED attribute specification. Wherever an asterisk appears it indicates that the subscript to be used in the equivalent reference is the corresponding subscript of the reference to the defined item.

2. The range specified by a bound pair of a defined array must equal or be contained within the range specified by the corresponding bound pair of the base array.

3. The length of a simple defined string must not be greater than the length of the corresponding base string.

4. The size of a simple defined area must be equal to the size of the corresponding base area.

Example

```
DECLARE A(10),
        1 X(M,N),
          2 Y,
          2 Z,
        C DEFINED A(3),
        1 E(M/2) DEFINED X(*,I),
          2 F,
          2 G;
```

Independent of DEFAULT statements:

```
C refers to A(3)
E.F(3) refers to X.Y(3,I)
```

4.13.4.2 iSUB Definings

The use of iSUB defining allows a transformation to be applied to the subscripts of a defined reference to designate a chosen element of the base array. If the defined reference does not specify some subscript expression, the transformation is applied to the subscripts generated during the evaluation of the aggregate expression or aggregate assignment which contains the reference. The defined item and base items may be structure arrays.

The subscripts in the base item in the DEFINED attribute make one or more references to the dummy iSUB variables. i is an integer constant in the range 1 to N, where N is the number of dimensions in the defined array. The number of subscripts in the base item must be equal to the number of declared dimensions of the base array. Subscript positions must not be specified by asterisks.

Corresponding to a subscripted iSUB defined reference is an equivalent subscripted reference to an element of the base array. The qualified name part is derived in the way used for simple defined references. However, the subscript list is evaluated differently. The I-th subscript in the equivalent reference is the I-th subscript in the base item, after each iSUB variable has been replaced by the integer value of the I-th subscript in the defined reference.

The attributes of the elements of the base array and attributes of the elements of the defined array must match according to the rules for valid simple defining.

An array reference to an iSUB array must not be passed as an argument, unless a dummy is created. Scalar references to iSUB defined arrays may be passed without the creation of a dummy.

Within the expressions in a base item, iSUB variables are treated as fixed binary variables with the precision given by the conversion rules.

Examples

1.
```
DECLARE X(10,10),
        Y(5) DEFINED X(2*1SUB, 2*1SUB);
```

The array Y refers to the even elements of the diagonal of X. Thus y(1) refers to X(2,2), Y(2) X(4,4), etc.

2.
```
DECLARE V(25),
        W(5,5) DEFINED V(5*(1SUB-1)+2SUB);
```

The rectangular array W refers to the elements of the linear array V. Thus W(1,1) refers to V(1),W(1,2) to V(2),... W(2,1) to V(6), W(2,2) to V(7), etc.

4.13.4.3 String overlay Defining

String overlay defining is applicable only to string and pictured data. It enables some or all of the storage associated with a variable to be accessed using any suitable string, pictured scalar, or aggregate of string and pictured data.

The <position-attribute> can be used to specify the bit or character within the base item at which the defined item is to begin.

The defined item and the base item must both be of bit class or character class and must have the UNALIGNED attribute. Any bit strings or character strings must be fixed length.

The bit class consists of bit strings.

The composition of the character class is:

1. Character strings
2. Pictured decimal numeric data
3. Pictured character string data
4. Aggregates consisting of any of the items in 1, 2, and 3.

All the elements of the base item covered by the range of defining and all the elements of the defined item must have the UNALIGNED attribute.

The base item cannot be an aggregate parameter, nor can it be an interleaved array. An interleaved array is an array whose associated storage contains gaps occupied by other fields. An array is interleaved if, when written in cross-section notations, it has an asterisk to the right of any subscript expression or has no asterisk corresponding to an array of structures which contains the array.

Example

```
DECLARE A CHARACTER (10),
        B(10) CHARACTER (1) DEFINED A;
B = '0';
```

The assignment to B sets each character in A to '0'. Order of evaluation proceeds as follows:

1. The array bounds, string lengths, and area sizes of a defined item are evaluated upon entry to the block in which the item is declared.
2. A defined reference is treated as a reference to some or all of that generation of its base item that is available at the point of reference. When a defined item is passed as an argument without creation of a dummy, the corresponding parameter refers to the relevant part of that generation of the base item that is available when the argument is passed. Reallocation of the base within the called procedure will not affect the meaning of the parameter.
3. In a reference to a defined item, all subscripts in the reference

are evaluated and converted to integer before any of the subscripts in the base item are evaluated. Expressions in the base item are then evaluated in the current environment of the block containing the declaration of the defined item. Names used in the base item are interpreted in the block containing the declaration of the defined item.

4.14 DEFAULT RULES

Unless a declaration was produced by a DECLARE statement that explicitly provided all attributes, the declaration is likely to have an incomplete set of attributes. The attribute set of each declaration is completed by performing the following rules in the indicated order:

1. If the declaration contains a precision attribute of the form (P,Q) the FIXED attribute is given to the declaration.

2. If the declared item is a member of a structure and does not have an alignment attribute, then the alignment attribute of its immediately containing structure is given to the declaration, if one was specified.

3. Beginning in the block in which the declaration occurred, all statements are evaluated in the order in which they appear. When all default statements in that block have been evaluated, then the DEFAULT statements in the next outer-most block are evaluated. This process continues outward until the DEFAULT statements in the outer-most block have been evaluated.

4. The default rules for the system are applied by evaluating the DEFAULT statements in Section 4.14.2, Standard System Default Rules, as if they had appeared in the block containing the external procedure.

5. Each entry declaration produced by a label prefix is given a set of parameter descriptors derived from the declaration of each parameter of the entry. This ensures that the attributes of the descriptor will be consistent with those of the parameter.

6. The declaration of each arithmetic literal constant that does not have a base attribute is given the DECIMAL attribute unless it contains a "B", in which case it is given the BINARY attribute.

If the declaration of an arithmetic literal constant does not contain a precision, then the precision obtained by converting the source precision to the base and scale of the attribute set developed so far is used. The source precision is (P,Q), where P is the number of digits in the "decimal-number" or "binary-number", and for constants with scale "F" (i.e., FIXED constants) Q is the number of fractional digits in the "decimal-number" or "binary-number". The source scale for this conversion is the same as the target scale.

The value of the decimal constant is N * 10 ** K, where N is the value of the "decimal-number", and K is the value of the exponent. If the exponent does not occur, K is taken to be zero.

4.14.1 Rejection Rules

A declaration receives an attribute-set of a DEFAULT statement only if the result of combining the attribute-set with the attributes already acquired by the declaration produces an attribute-set described by these rejection rules. Before the consistency test is made, the attributes are re-ordered to conform to the order implied by the rejection rules.

```
<attribute-set> ::=
     {<stored-data> | BUILTIN | CONDITION}
     [INTERNAL | EXTERNAL]

<stored-data> ::= [<data-type>] [<storage-class>] [<storage>]

<data-type> ::=
     <arithmetic-type> | <string-type> | <picture-type> |
     <entry-type> | <file-type> | POINTER | OFFSET [IN] |
     AREA [SIZE] | LABEL [LOCAL] | TASK | STRUCTURE

<arithmetic-type> ::=
     [BINARY | DECIMAL] [REAL]
     {[FIXED] [PRECISION (P[,Q])] |
     [FLOAT] [PRECISION] (P)}

<string-type> ::= [BIT | CHARACTER] [LENGTH] [VARYING | NONVARYING]

<picture-type> ::= PICTURE [REAL]

<entry-type> ::= [ENTRY] [RETURNS]

<stream-type> ::= [STREAM] [[INPUT | PRINT | OUTPUT [PRINT]]

<record-type> ::= [SEQUENTIAL | DIRECT]
     [INPUT | OUTPUT | UPDATE]
     [KEYED] [RECORD]

<storage-class> ::= AUTOMATIC | BASED | STATIC | CONTROLLED |
     DEFINED | PARAMETER | MEMBER

<storage> ::= [CONSTANT | VARIABLE] [ALIGNED | UNALIGNED]
     [DIMENSION] [INITIAL]
```

4.14.2 Standard System Default Rules

```
Entry Defaults:
     DEFAULT (RETURNS) ENTRY:
     DEFAULT (ENTRY & RANGE (I:N))
          RETURNS(FIXED | BINARY | REAL):
     DEFAULT (ENTRY) RETURNS(FLOAT DECIMAL REAL):

File Default:
     DEFAULT (INPUT | OUTPUT | UPDATE | STREAM | RECORD |
          PRINT | KEYED | DIRECT | SEQUENTIAL | ENVIRONMENT ) FILE

Arithmetic Defaults:
     DEFAULT (RANGE(I:N)
          &¬ (FIXED | FLOAT | BINARY | DECIMAL | REAL))
          FIXED BINARY:
     DEFAULT (¬CONSTANT) FLOAT, DECIMAL, REAL:
     DEFAULT (FIXED & BINARY &¬ CONSTANT) PRECISION (15,0);
     DEFAULT (FIXED & DECIMAL & ¬ CONSTANT) PRECISION (5,0);
     DEFAULT (FLOAT & BINARY &¬ CONSTANT) PRECISION (21);
     DEFAULT (FLOAT & DECIMAL &¬ CONSTANT) PRECISION (6);

String and Area Defaults:
     DEFAULT (CHARACTER | BIT) NONVARYING;
     DEFAULT (AREA) SIZE (256);

Scope and Storage Class Defaults:
     DEFAULT ((ENTRY | FILE) & (AUTOMATIC
          | BASED | STATIC | PARAMETER | DEFINED | CONTROLLED |
          ALIGNED | UNALIGNED | MEMBER | ARRAY | INITIAL)) VARIABLE;
     DEFAULT (PARAMETER) INTERNAL VARIABLE;
     DEFAULT (MEMBER) INTERNAL;
     DEFAULT (ENTRY | FILE) CONSTANT;
     DEFAULT ¬CONSTANT) VARIABLE;
     DEFAULT (CONDITION | ((FILE | ENTRY) & CONSTANT)) EXTERNAL;
     DEFAULT (¬CONSTANT) INTERNAL;
     DEFAULT (VARIABLE & EXTERNAL) STATIC;
     DEFAULT (VARIABLE) AUTOMATIC;

Storage Mapping Defaults:
     DEFAULT (CHARACTER | BIT | PICTURE) UNALIGNED:
     DEFAULT (¬CONSTANT) ALIGNED;
```

SECTION 5. DATA MANIPULATION

5.1 EXPRESSIONS

An expression is an algorithm used for computing a value. Expressions are of three types: scalar, array, and structure, depending upon the type of result. An array or structure expression is an aggregate expression. An array (or structure) expression is simply an array (or structure) evaluated by expansion of the expression into a collection of scalar expressions. Syntactically, a scalar expression consists of a constant, a scalar variable, a scalar expression enclosed in parentheses, a scalar expression preceded by a prefix operator, two scalar expressions connected by an infix operator, or a function reference that returns a scalar value. Operands in a scalar expression need not have the same data attributes. If they differ, conversion will be performed before the operation.

5.1.1 Scalar Expressions

A scalar expression returns a scalar value. The class of the expression is dependent upon the operators involved: arithmetic, comparison, bit-string or concatenation. For program-control data , only the comparison operation may be performed; only the comparisons = and ¬= are allowed.

If A and B are expressions, then the operators + and - used in expressions of the form +A or -A, are called prefix operators. When these operators are used in expressions of the form A+B or A-B they are called infix operators.

5.1.2 Aggregate Expressions

An aggregate expression is an expression involving one or more aggregate operands (i.e., array or structure operands).

Aggregate expressions can be divided into three types: simple array, simple structure, and structure array expressions. A simple array expression is an expression which has at least one array operand but no structure or structure array operands. A simple structure expression is an expression which has at least one structure operand but no array or structure array operands. A structure array expression is an expression which has an array operand and a structure operand, or has at least one structure array operand.

5.1.2.1 Built-In Functions With Aggregate Arguments

The computational built-in functions may be given aggregate expressions in argument positions other than those which must be integer constants. The aggregate type of the result is determined by evaluating the arguments as if they were operands in an expression using the rules specified above.

For example, if A is a structure, B is a simple array and C is a scalar.

SIN(A) is a structure expression

```
MAX(B,C)  is an array expression
MIN(A,B)  is a structure array expression
```

5.1.2.2 Value of an Aggregate Expression

Aggregate expressions can be used only on the right-hand side of an assignment statement, as arguments, and in a data list of a PUT statement.

In an assignment statement the values designated by an aggregate expression are assigned to one or more aggregate target variables. Such an assignment is carried out as a sequence of scalar assignments. The definition has two major consequences:

1. Array expressions may not yield the results of conventional matrix algebra.

2. When a variable, or part thereof, is specified both as an operand and as a target, the values of a variable when used in the expression may be those assigned earlier in the sequence of scalar assignments.

When an aggregate expression is passed as an argument a dummy variable (the dummy argument) is constructed. The aggregate type of the dummy argument is that specified in the corresponding parameter position of an entry attribute. The values transmitted to the parameter are determined by assignment of the expression to the dummy argument.

The values transmitted by an aggregate expression in an output data list are those which would be assigned to a target variable having the aggregate type of the expression.

5.2 OPERATIONS ON EXPRESSIONS

5.2.1 Prefix Operations

The prefix operations of plus and minus yield a result having the base, scale, mode, and precision of the operand.

A prefix operator applied to an aggregate yields a result whose aggregate type is the same as the operand. Thus, if A is an array and B is a structure, -A is an array expression and -B is a structure expression. The bounds and number of dimensions of an array expression are those of the operand.

5.2.2 Arithmetic Operations

An arithmetic expression of any complexity is composed of a set of elementary arithmetic operators.

Syntax

<elementary-arithmetic-operation> ::=

([+|-] <operand>)|
(<operand> (+|-|*|/|**) <operand>)

The syntax specifies the prefix operations of plus and minus and the infix operations of addition, subtraction, multiplication, division, and exponentiation. Operations are performed only with coded arithmetic data. If necessary, the data will be converted to coded arithmetic type before the operation is performed.

5.2.2.1 Mixed Characteristics

The two operands of an arithmetic operation may differ in form, base, scale, mode, and precision. When they differ (except in some cases of exponentiation), conversion takes place according to the following rules:

FORM: Pictured numeric operands of arithmetic operations will be converted to coded form. The result of an arithmetic operation is always in coded form.

BASE: If bases differ, the decimal operand is converted to binary.

SCALE: If the scales of the operands differ, the fixed-point operand will be converted to floating-point, except in the case of exponentiation in which the first operand is floating-point and the second is fixed-point with precision (P,Q). In the latter case, the second operand is not converted, and the result has the base, scale, mode, and precision of the first operand.

MODE: Mode is always real so no conversion is necessary.

PRECISION: If precisions differ, no conversion is done.

5.2.2.2 Results of Arithmetic Operations

After the conversions specified above have taken place, the arithmetic operation is performed. Any necessary truncations will be made towards zero, regardless of the base or scale of the operands.

The base, scale, mode, and precision of the result depend on the operands and the operator in the following ways:

1. Infix Operations Floating Point: If the operands of an infix operation are floating-point and the base and mode of the result are the common base and mode of the operands, the precision of the result is the greater of the precisions of the two operands.

2. Infix Operation Fixed-Point: If the operands of an infix operation are fixed, and if the operation is not exponentiation, the result is fixed and the base and mode of the result are the common base and mode of the operands. If the operation is exponentiation, the second operand is converted to floating-point if its scale factor is not zero. The first operand is converted to floating-point unless the

second operand is an unsigned integer constant meeting the conditions of item 4 below. In these cases, the rules for floating-point apply.

The precision of a fixed-point result depends on the operation and the precisions of the operands, according to the rules given below. The following symbols are used:

MP = The maximum precision allowed by the machine configuration for the base of the result.

M = The total number of positions in the result.

N = The scale factor of the result.

P = The total number of positions in operand one.

Q = The scale factor of operand one.

R = The total number of positions in operand two.

S = The scale factor of operand two.

Y = The value of operand two if it is an unsigned integer constant.

1. Addition and Subtraction:

```
M=MIN(MP,MAX(P-Q,R-S)+MAX(Q,S)+1)
N=MAX(Q,S)
```

2. Multiplication:

```
M=MIN(MP,P+R+1)
N=Q+S
```

3. Division:

```
M=MP
N=MP-P+Q-S
```

4. Exponentiation:

If the second operand is an unsigned non-zero real fixed-point constant of precision (R,0),

```
M=(P+1) *Y - 1
N=Q *Y
```

If M>MP or Y is not an unsigned non-zero real fixed-point constant of precision (R,0), the first operand is converted to floating-point and rules for floating-point exponentiation apply.

Some special cases of exponentiation, X1**X2, are defined as follows:

a. IF X1 = 0 and X2 > 0, the result is 0.

b. If X1 = 0 and (X2 <= 0), the ERROR condition is raised.

c. If X1 = 0 and X2 = 0, the result is 1.

d. If X1 < 0 and X2 is not fixed-point with precision (P,0), the ERROR condition is raised.

5.2.2.3 Infix Operators and Aggregate Operands

An infix operator applied to two aggregate operands, or to an aggregate operand and a scalar, yields a result whose aggregate type is determined by the operands. The following table gives the aggregate type of the result of an infix operation in the terms of the aggregate type of the operands:

OPERAND 1	OPERAND 2			
	SCALAR	SIMPLE ARRAY	STRUCTURE	STUCTURE ARRAY
SCALAR	SCALAR	SIMPLE ARRAY	STRUCTURE	STRUCTURE ARRAY
SIMPLE ARRAY	SIMPLE ARRAY	SIMPLE ARRAY	STRUCTURE ARRAY	STRUCTURE ARRAY
STRUCTURE	STRUCTURE	STRUCTURE ARRAY	STRUCTURE	STRUCTURE ARRAY
STRUCTURE ARRAY	STRUCTURE ARRAY	STRUCTURE ARRAY	STRUCTURE ARRAY	STRUCTURE ARRAY

If both operands are arrays they must have the same bounds and number of dimensions. The result has these common bounds and number of dimensions. If only one operand is an array the result has the bounds and the number of dimensions of this array. When structures are involved, they must all have the same structuring.

5.2.2.4 Integer Conversion

If conversion to integer is specified, as in the evaluation of subscript expressions, the conversion will be to fixed-point binary. The scale factor is zero. Truncation, if necessary, will be toward zero.

5.2.2.5 Arithmetic Base and Scale Conversion

The table below defines the precision resulting from base and scale conversion. CEIL refers to the ceiling of the expression, (The "ceiling" of a number is the smallest integer equal to or greater than the number.)

5.2.2.6 Floating-Point to Fixed-Point Conversion

Conversion from floating-point scale to fixed-point scale will occur only when a destination precision is known, as in an assignment to a fixed-point variable. If the destination precision is incapable of holding the floating-point value, the result is undefined, and the SIZE condition will be raised if enabled.

Arithmetic Base and Scale Conversion

After \ Before Conversion	Binary Fixed (p,q)	Decimal Fixed(p,q)	Binary Float (p)	Decimal Float(p)
Binary Fixed	(p,q)	(MIN(CEIL (p*3.32)+1,N1), CEIL(ABS(q) *3.32*SIGN(q)		
Decimal Fixed	(MIN(CEIL(p/3.32) +1,N2)CEIL(ABS(q) /3.32)*SIGN (q))/	(p,q)		
Binary Float	(MIN(p,N3))	(MIN(CEIL (p*3.32),N3))	(p)	(MIN(CEIL (p*3.32),N3))
Decimal Float	MIN (CEIL(p/3.32),N4)	(MIN(p,N4))	(MIN(CEIL (p/3.32),N4))	(p)

N1 is the maximum number of digits allowed for binary fixed-point data (38 by default or 78 when OPTIONS (DOUBLE) is specified.)

N2 is the maximum number of digits allowed for decimal fixed-point data (11 by default or 23 when OPTIONS (DOUBLE) is specified.)

N3 is the maximum number of digits allowed for binary floating-point (38 by default or 78 when OPTIONS (DOUBLE) is specified.)

N4 is the maximum number of digits allowed for decimal floating-point data (11 by default or 23 when OPTIONS (DOUBLE) is specified.)

5.2.3 Comparison Operations

Syntax

<comparison-operation> ::=

<operand>(<|¬<|<=|=|¬=|>|¬>|>=)<operand>

Arithmetic comparison involves the comparison of signed numeric values in coded arithmetic form. Conversion of pictured numeric operands will be performed.

Character comparison involves left-to-right, pair-by-pair comparisons of characters. If the operands are of different lengths, the shorter is extended to the right with blanks.

Bit comparison involves the left-to-right comparison of binary digits. If the strings are of different lengths, the shorter is extended on the right with zeros.

The result of a comparison is a bit string of length one; the value is '1' B if the relationship is true and '0' B if it is false.

Comparison operations always take place between values in common representation. If the operands of a comparison are of different

types, the operand of the lower type is converted to conform with the representation of the operand of the higher type. The priority of types is (1) arithmetic (highest), (2) character string, (3) bit string. If one or both of the operands is arithmetic, the operands are converted to the same attributes as those defined for arithmetic operations.

As a result of conversion, both operands will then be arithmetic or character string, and arithmetic or character comparison will be performed.

Only the operators = and ¬= may be used with locator variables, and both operands must be locator variables or a function that returns a locator value.

5.2.4 Bit String Operations

Syntax

<bit-string-operation> ::=

{[¬]<operand>}|{<operand>{& | |}<operand>}

Semantics

The prefix operation NOT (denoted by ¬) and the infix operations AND and OR (denoted by & and | respectively) are specified above. The operands will be converted to bit string type before the operation is performed. The result will be of bit string type. If the operands are of different lengths after conversion, the shorter will be extended on the right with zeros to the length of the longer. The length of the result will be of this extended length. The result is of varying length if either operand has the VARYING attribute.

The operations are performed on a bit-by-bit basis. As a result of the operations, each bit position has the value defined in the following table:

A	B	not A	not B	A and B	A or B
1	1	0	0	1	1
1	0	0	1	0	1
0	1	1	0	0	1
0	0	1	1	0	0

Examples

If field A is 010111 B, Field B is 111111 B, and Field C is 101000 B, then

```
      ¬A yields 101000 B
  C & B yields 101000 B
  A |¬C yields 010111 B
(¬C|¬B) yields 010111 B
```

For a discussion of how these expressions are evaluated, see Section 5.3, Evaluation of Expressions.

5.2.5 Concatenation Operations

Syntax

<concatenation-operation> ::= <operand> || <operand>

Semantics

If both operands are of bit string type, no conversion is performed, and the result is of bit string type. In all other cases, the operands are converted where necessary to character string type before the concatenation is performed, and the result is of character string type. The length of the result is the sum of the lengths of the two operands. The result is a varying string if either of the operands has the VARYING attribute.

Examples

```
If A is '010111' B, B is '101' B, C is
'XY,Z' and D is 'AA/BB', then
A||B Yields "010111101' B
A||A||B yields '010111010111101' B
C||D yields 'XY,ZAA/BB'
D||C yields 'AA/BBXY,Z'
```

5.2.6 Type Conversion

5.2.6.1 Bit String to Character String

The bit 1 becomes the character 1, and the bit 0, the character 0. The length is unchanged. The null bit string becomes the null character string.

5.2.6.2 Character String to Bit String

The characters 1 and 0 become the bits 1 and 0. The CONVERSION condition will be raised if the character string contains characters other than 0 and 1 in the portion of the string to be converted. The null character string becomes the null bit string.

5.2.6.3 Character String to Arithmetic

The string for conversion must contain:

[+|-] arithmetic constant

The optionally signed constant may be surrounded by an arbitrary number of blanks. However, blanks may not appear between the optional sign and the constant.

The arithmetic value of the constant is converted to the base, scale, mode, and precision that a real fixed decimal value of the maximum single precision (11 by default, 23 if OPTIONS(DOUBLE) is specified) would have been converted to if this had appeared in place of the character string value. A null string gives the value 0.

5.2.6.4 Bit String to Arithmetic

The bit string is interpreted as an unsigned binary integer, and is converted to the base, scale, mode, and precision that a real fixed binary value of the maximum single precision (38 by default, 78 if OPTIONS(DOUBLE) has been specified) would have been converted to had it appeared. A null string gives the value 0.

5.2.6.5 Arithmetic to Character String

See Section 8.4.4.2, List Directed Output Format.

5.2.6.6 Arithmetic to Bit String

The absolute arithmetic value is converted to real, then to fixed-point binary, precision (P,0), where P is related to the precision before conversion as follows (with ceilings of expressions used):

```
BINARY FIXED (R,S)   P = MIN(N1,MAX(R-S,0))

DECIMAL FIXED (R,S)  P= MIN(N1,MAX(CEIL((R-S)*3.32),0))

BINARY FLOAT (R)     P = MIN(N1,R)

DECIMAL FLOAT (R)    P = MIN(N1,CEIL(R*3.32))
```

The resulting binary fixed-point value is interpreted as a bit string length P.

5.3 EVALUATION OF EXPRESSIONS

5.3.1 Order of Evaluation of Scalar Expressions

In the evaluation of an expression, the priority of operations is as follows:

Highest:	¬,**,PREFIX +, PREFIX -
	*,/
	INFIX +, INFIX -
	||
	>=, >, ¬>, ¬=, <, ¬<, <=, =
	&
Lowest:	|

Operations within an expression are performed in the order of decreasing priority. For example, in the expression A+B**3, exponentiation is performed before addition. If an expression involves operations of the same priority, the operations not, exponentiation, prefix +, and prefix - are performed from right to left. All other operations are performed from left to right.

If an expression is enclosed in parentheses, it is treated as a single operand. Each parenthesized expression is evaluated and then their associated operations performed. Thus, parentheses modify the normal rules of priority.

The operators + and * are commutative, but not associative, as low order rounding errors will depend on the order of evaluation of an expression, thus, A+B+C is not necessarily equal to A+(B+C).

5.3.2 Order of the Evaluation of Aggregate Expressions

Array expressions are evaluated by performing in turn a complete scalar evaluation of the expression for each position of the array. The evaluations proceed in row-major order (final subscript varying most rapidly). The result of an evaluation for an earlier position can alter the values of scalar elements for the evaluation of a later position.

Structure expressions are evaluated by performing a complete scalar evaluation of the expression for each eligible field, in the order in which the fields in the structures are declared. The results of an evaluation for an earlier position can alter the result for the evaluation of a later position.

SECTION 6. PROGRAM STRUCTURE

6.1 Statements

A PL/I program is constructed from basic program elements called statements. Statements are grouped into larger program-elements, the group and the block. There are two types of statements: simple and compound.

6.1.1 Simple Statements

Syntax

```
<simple-statement> ::=

     [[<statement-identifier>] <statement-body>];
```

Semantics

The <statement-identifier>, if it appears, is a keyword, characterizing the kind of statement. If it does not appear, and the statement body does appear, then the statement is an assignment statement. If only the semicolon appears, the statement is called a null statement.

Examples

```
DO I = J TO 10; (DO is the <statement-identifier>)
A = B + C;      (assignment statement)
      ;         (null statement)
```

6.1.2 Compound Statements

A compound statement is a statement that contains other elements.

Syntax

```
<compound statement>::=

     <conditional compound statement>|
     <ON compound statement>
```

Semantics

The final contained statement of a compound statement is a simple statement and thus has a terminal semicolon. Hence, the compound statement will automatically be terminated by this semicolon.

Examples

```
IF A=B THEN GO TO S1; ELSE A=C;
(conditional compound statement)
ON OVERFLOW GO TO OVFIX;
(ON compound statement)
```

6.1.3 Label Prefixes

Statements may be labeled to permit reference to them.

Syntax

```
<labeled-statement> ::=
    <identifier> : [<identifier>:] ...<statement>
```

Semantics

The one or more <identifiers> are called labels and may be used interchangeably to refer to that statement. Labels appearing before procedure statements and entry statements are special cases and are known as entry names. Labels appearing before format statements are known as format names. All other labels are called statement labels.

A label appearing before a statement is said to be declared, by virtue of its appearance as a label. Statement labels appearing before DECLARE statements are ignored.

Labels may also be used which are followed by constant subscripts in parentheses. Such labels are elements of label constant arrays. A label constant array is declared by virtue of the appearance of its elements as statement prefixes. The following is an example of the use of a label constant array:

```
                .
                .
(1)   L(1):  A=B;
                .
                .
                .
        L(2):  A=C;
                .
        J = 1;
                .
                .
(2)     GO TO  L(J);
                .
                .
                .
```

Statement (2) transfers control to statement (1).

6.2 GROUPS

A group is a collection of one or more statements and is used for control purposes.

Syntax

```
<group> ::= <do-group>|<single-statement>
<do-group>::= [<label>:] ... <do-statement>
                                [program-element-1
                                 program-element-2
                                        .
                                        .
                                        .]
                                END [<label>];

<single-statement> ::= [<label:>] ... <statement>
```

Semantics

The <label> following END is one of the labels of the DO statement.

The DO statement is called the heading statement of the <do-group>, and may specify iteration. Each program-element represents one or more statements.

The <statement> is any statement except DO, END, PROCEDURE, BEGIN, DECLARE, FORMAT, ENTRY, or DEFAULT.

Example

```
ALPHA:DO;
    A=B*C;
    IF A < 0 THEN
    DO;
      B = 1;
      C = 0;
    END;

END ALPHA;
```

In the example above, any of the <single statements>s (except the DO statement and END statement) is an example of the <single-statement> form of a group.

6.3 BLOCKS

A block is a collection of statements that defines the program region, or scope, throughout which an identifier is established as a name. It also is used for control purposes. There are two kinds of blocks: begin blocks and procedure blocks.

Syntax

```
<begin-block> ::= [<label>:]...<begin-statement>
                                  [program-element-1]
                                   program-element-2
                                           .
                                           .
                                           .]
                                   END [<label>];

<procedure-block> ::=
     <label>: [<label:>]...<procedure-statement>
                               [program-element-1
                                program-element-2
                                        .
                                        .
                                        .]
                                END [<label>];
```

Semantics

The label following END in the <begin-block> is one of the labels of the BEGIN statement.

The label following END in the <procedure-block> is one of the labels of the PROCEDURE statement.

The BEGIN statement and the PROCEDURE statement in the above forms are called heading statements. While the labels of the BEGIN statement are optional, the PROCEDURE statement must have at least one label.

Although the begin block and the procedure block have a physical resemblance and play the same role in delimiting scope of names and defining allocation and freeing of storage, they differ in an important functional sense. A begin block, like a single statement, is activated by a normal sequential flow (except when used as an on-unit), and it can appear whenever a single statement can appear. A procedure can only be activated remotely by a CALL statement or by a function reference. When a program containing a procedure is executed, control passes around the procedure from the statement before the PROCEDURE statement to the statement after the END statement of the procedure.

Since a procedure can be activated only by a reference to it, every procedure must have a name. The label required for the heading statement of a procedure serves as the procedure name.

The procedure name gives a means of activating the procedure at its primary entry point. Secondary entry points can also be defined for the procedure by use of the ENTRY statement. The labels preceding all ENTRY statements in a given procedure and the heading statement of the procedure are collectively called entry names for the procedure.

As the above definition of block implies, any block A can include another block B, but partial overlap is not possible, block B must be completely included in block A.

A procedure that is not included in any other block is called an external procedure. A procedure included in some other block is

called an internal procedure.

The only external blocks are external procedures. Every begin block must be included in some other block. The entry names of external procedures are not internal to any procedure and are called external names.

All of the text of a begin block, except the labels preceding the heading statement of the block, is said to be contained in the block.

All of the text of a procedure, except the entry names of the procedure, is said to be contained in the procedure.

That part of the text of a block B that is contained in block B, but not contained in any other block contained in block B, is said to be internal to block B.

The notion of internal to is vital in the definition of scope.

Example

```
A:  PROCEDURE;
          .
          .
          .
    statement 1
    B:  BEGIN;
              .
              .
              .
        statement 2
        statement 3
        END B;
    statement 4
    C:  PROCEDURE;
              .
              .
              .
        statement 5
    X:  ENTRY;
              .
              .
              .
        D:  BEGIN;
              .
              .
              .
            statement 6
            statement 7
            END D;
              .
              .
              .
        statement 8
        END C;
          .
          .
          .
    statement 9
    END A;
```

In this example, statements 1 through 9 are labeled or unlabeled simple statements. Block A contains block B and block C, and block C contains block D.

Block C is an internal procedure.

Block A is an external procedure. The procedure name is A, which is an external name, and the only entry name for the procedure.

X is an entry name corresponding to a secondary entry point for procedure C.

Blocks B and D are begin blocks.

The text internal to block A consists of:

```
PROCEDURE;
statement 1
B
statement 4
C:
X:
statement 9
END A;
```

The text internal to block B consists of:

```
BEGIN;
statement 2
statement 3
END B;
```

The text internal to block C consists of:

```
PROCEDURE;
statement 5
ENTRY;
D:
statement 8
END C;
```

The text internal to block D consists of:

```
BEGIN;
statement 6
statement 7
END D;
```

6.4 CONDITION PREFIXES

A condition prefix specifies whether or not a program interrupt will result upon the occurrence of the specified condition.

One or more condition prefixes may be attached to a statement. Each condition prefix is followed by a colon to separate it from the rest of the statement or from other prefixes. Condition prefixes precede the entire statement, including any possible label prefixes for the statement.

A condition prefix is a list of condition names, separated by commas

and enclosed in parentheses.

Syntax

```
<fully-prefixed-statement> ::=
      ([<condition-prefix>:]...
       [<label>:]...)...<statement>

<condition-prefix> ::= (<condition-name>
                        [,<condition-name>]...)
```

Semantics

The condition names are chosen from the following fixed set:

UNDERFLOW
OVERFLOW
ZERODIVIDE
FIXEDOVERFLOW
CONVERSION
SIZE
STRINGRANGE
SUBSCRIPTRANGE

Any of these conditions names may be preceded by the word NO. If NO is used, there can be no intervening blank between NO and the condition. For example, NOCONVERSION can be specified in the prefix list. NO<condition-name> disables the condition referred to by the <condition-name>.

6.5 PROGRAMS

A program is composed of one or more external procedures.

SECTION 7. STATEMENTS

This section includes a description of each statement in the PL/I language.

To show the relationships among these statements, they are classified into logical groups.

7.1 CLASSIFICATION OF STATEMENTS

Statements may be classified into the following logical groups: assignment, control, declaration, error control and debug, input/output, program structure, storage allocation, system attribute, and null.

7.1.1 Assignment Statement

The assignment statement is used to evaluate expressions and to assign values to scalars, arrays, and structures.

7.1.2 Control Statements

The control statements affect the normal sequential flow of control through a program. The control statements are: CALL, DELAY, DO, EXIT, GO TO, IF, RETURN, SORT, STOP, and WAIT.

7.1.3 Program Structure Statements

The program structure statements are: PROCEDURE, BEGIN, END, DO, and ENTRY. The first three statements delimit the scope of declarations within a program. The ENTRY statement provides a secondary entry point for a procedure.

7.1.4 Data Declaration Statement

The data declaration statements DECLARE and DEFAULT specify attributes for identifiers.

7.1.5 Error Control and Debug Statements

The error control and debug statements include: DUMP, ON, REVERT, and SIGNAL. When an interrupt occurs during program execution, standard operating system action is taken. However, the language provides the facility to override system action on these interrupts. By using the ON statement a programmer may specify the action to be taken when an interrupt occurs and can record the status of the program at the point of execution where the interrupt occurs. By using the SIGNAL statement, the programmer may initiate programmed interrupts and may simulate machine interrupts to facilitate debugging.

7.1.6 Input/Output Statements

The input/output statements may be classified as follows: file preparation, record status, data specification, and data transmission.

7.1.6.1 File Preparation Statements

The OPEN statement associates a file name with a file and completes the specification of the attributes of the file, in preparation for input/output on a file. The CLOSE statement dissociates the file name from the file and thereby releases the file name for use in connection with any other file.

7.1.6.2 Record Status Statements

The DELETE statement deletes a record from a file with the UPDATE attribute.

7.1.6.3 Data Specification Statements

The format of data fields to be transmitted may be specified by the FORMAT statement or in the GET or PUT data transmission statements.

7.1.6.4 Data Transmission Statements

The GET and PUT statements cause values to be transmitted between a file and specified variables in the program.

The READ and WRITE statements cause a single record to be transmitted between a file and variables in the program.

The REWRITE statement specifies the updating of an existing record of the file.

The LOCATE statement permits a record to be created in the buffer storage and subsequently written.

The DISPLAY statement causes messages to be transmitted between the program and the machine operator.

7.1.7 Storage Allocation Statements

The storage allocation statements are ALLOCATE and FREE. These statements allocate and free storage for variables.

7.1.8 System Attribute Statements

The system attribute statements consist of the file attribute assignment statement, the file attribute reference statement, the task attribute assignment statement, and the task attribute reference statement. These statements allow system file and task attributes to be assigned to and/or referenced.

7.1.9 Null Statement

The null statement is an empty statement and is treated as a non-operational statement.

7.2 SEQUENCE OF CONTROL

Within a block, control normally passes sequentially from one statement to the next. If a DECLARE, FORMAT, or ENTRY statement is encountered, control passes to the next statement. If an internal PROCEDURE statement is encountered, control passes to the statement following the end of the procedure. Control passes to the statement following an IF statement when control reaches the end of the then-unit. Sequential operation is also modified by the following statements: CALL, DO, END, EXIT, GO TO, PROCEDURE, RETURN, SIGNAL, and STOP.

A CALL statement passes control to the specified entry point.

A DO statement defines a group that is treated as a single statement and can cause repeated execution of a group.

An END statement which logically terminates a procedure acts as a RETURN statement, causing control to return to the invoking procedure.

The EXIT statement causes control to leave a task. The STOP statement causes control to leave a program.

A GO TO statement causes control to transfer to the specified statement label.

A PROCEDURE statement heads a procedure. Procedures may be considered as independent blocks and are placed anywhere within an external procedure, consistent with desired identifier scopes. However, a procedure may be invoked only by a CALL statement or a function reference. Thus, control passes around a nested procedure, from the statement before a PROCEDURE statement to the statement after the appropriate END statement for the procedure.

A RETURN statement returns control from a procedure to the invoking procedure.

A SIGNAL statement specifying an enabled condition causes control to pass to the on-unit of the associated ON statement. If there is no associated ON statement, control is passed to the appropriate system routine.

The following conditions may cause sequential operation to be modified:

1. A function reference in any expression causes control to pass to the specified procedure.

2. The occurrence of an enabled condition specified in an ON statement causes control to pass to the associated on-unit. If there is no ON statement, standard system action is taken.

3. The flow of control through the IF and ON statements and through a do group may or may not be sequential.

The following example illustrates sequence of control:

```
A:  PROCEDURE;
      B:  X=Y+Z;
      C:  CALL D;
      E:  W=P*G;

      D:  PROCEDURE;
            G:  S=T/P;
            H:  RETURN;
      I:  END;
      J:  U=V**W;
      K:  GO TO N;
          .
          .
          .
N:  END;
```

Control flows in the following order: A, B, C, D, G, H, I,E,J,K,N.

7.3 LIST OF STATEMENTS BY CLASSIFICATION

7.3.1 The Assignment Statement

The <assignment-statement> is used to evaluate an expression and to assign its value to one or more target variables. The target variables may be scalar, array, structure, or pseudo variables.

Syntax

```
<assignment-statement> ::=
    <scalar-assignment>|<array-assignment>|
     <structure-assignment>;

<scalar-assignment> ::=
    {<scalar-variable>|<pseudo-variable>}
    [,{<scalar-variable>|<pseudo-variable>}]...)
  = <scalar-expression>

<array-assignment> ::=
   {<array-variable>|<pseudo-variable>}
    [,{<array-variable>|<pseudo-variable>}]...)
  = {structure-expression>[,BY NAME]}|
    {array-expression>[,BY NAME]}|
     <scalar-expression>

<structure-assignment> ::=
    {<structure-variable>|<pseudo-variable>}
    [,{<structure-variable>|<pseudo-variable>}]...)
  = {<structure-expression>[,BY NAME]}|
     <scalar-expression>
```

Semantics

A structure array is treated as an array.

Aggregate assignments (i.e., <array-assignment> and <structure-assignment>) are expanded into a series of scalar assignments.

7.3.1.1 Scalar Assignments

A <scalar-assignment> is performed as follows:

1. Subscripts of the target variables, arguments of SUBSTR pseudo-variables, and the second and third references, are evaluated from left to right.

2. The <scalar-expression> is then evaluated.

3. For each target variable (in left-to-right order), the expression is converted to the characteristics of the target variable according to the rules in Section 5.2.6, Type Conversions (except that whenever a conversion of arithmetic base is involved, the value is converted directly to the precision of the target variable). The converted value is then assigned to the target variable.

The following rules apply to string scalar assignment:

1. If the target variable is a fixed-length string, the expression value is truncated on the right if it is too long or padded on the right (with blanks for character strings, zeros for bit strings) if the value is too short. A string pseudo-variable is considered to be a fixed-length string. The resulting value is assigned to the target.

2. If the target is a VARYING string and the value of the expression is longer than the maximum length declared for the variable, the value is truncated on the right. The target string obtains a current length equal to its maximum length.

3. If the target is a VARYING string and the value of the expression is not greater than the maximum length, the value is assigned. The target string obtains a current length equal to the length of the value.

The following rules apply to scalar assignments other than string:

1. If the target is an area variable, the expression must be an area variable or function reference. All unfreed allocations in the target area are freed. A sequence of allocations and freeings is then performed in the target area for generations corresponding to the significant allocations in the source area. These operations are performed in precisely the same order as the significant allocations were allocated and, where appropriate, freed. The AREA condition is raised by the assignment if any such allocation in the target area overflows the area. Finally, the value of each allocation (which has not been freed) in the source area is assigned to the corresponding allocation in the target area.

2. If the target is a locator variable, the expression can be a locator variable or function reference. If the expression is of OFFSET type, its value is converted to POINTER by an implicit reference to the POINTER built-in function if the IN attribute was given. If the expression is of POINTER type, its value is converted to OFFSET by an implicit reference to the OFFSET built-in function if the IN attribute was given.

3. If the target is a label variable, the expression can only be a label name. Environmental information is always assigned to the label variable.

4. If the target is an event variable, the expression can only be an event variable or function reference. The assignment is uninterruptable, and it involves both the completion and status values; i.e., no other operations will take place (for example, in other tasks) while the assignment is being performed. An event variable does not become active when it has an active event variable assigned to it. It is an error to assign an event variable or function reference to an active event variable.

5. If the target is the STATUS pseudo-variable, a value can be assigned whether or not the event variable is active. It is an error to assign to a COMPLETION pseudo-variable if the named event variable is active.

7.3.1.2 Aggregate Assignments

The first target variable in an aggregate assignment is known as the master variable. If the master variable is an array, then an array expansion is performed. Otherwise, a structure expansion is performed.

In an <array-assignment>, all array operands must have the same number of dimensions and identical bounds. The array assignment is expanded into a loop of the form:

```
LABEL:  DO J1 = LBOUND (MASTER-VARIABLE,1) TO
                HBOUND (MASTER-VARIABLE,1);
        DO J2 = LBOUND (MASTER-VARIABLE,2) TO
                HBOUND (MASTER-VARIABLE,2);
            .
            .
            .
        DO JN = LBOUND (MASTER-VARIABLE,N) TO
                HBOUND (MASTER-VARIABLE,N);
        Generated Assignment Statement
        END LABEL;
```

In this expansion, N is the number of dimensions of the master variable that are to participate in the assignment. In the generated assignment statement, all array operands are fully subscripted, using (from left to right) the dummy integer variables J1 to JN. If an array operand appears with no subscripts, it will only have the subscripts J1 to JN. If cross-section notation is used, the asterisks are replaced by J1 to JN. If the original assignment statement has a condition prefix, the generated assignment statement is given this condition prefix. If the original assignment statement has a BY NAME option, the generated assignment statement is given a BY NAME option. If the generated

assignment statement is a structure assignment, it is expanded as given below.

In a <structure-assignment>, where the BY NAME option is not specified, the following rules apply:

1. The operands must be structures.

2. All of the structure operands must have the same number, K, of immediately contained items.

3. The assignment statement is replaced by K generated assignment statements. The I-th generated assignment statement is derived from the original assignment statement by replacing each structure operand by its I-th contained item. Such generated assignment statements may require further expansion. All generated assignments statements are given the condition prefix of the original statement.

In a <structure-assignment>, where the BY NAME option is given, the structure assignment is expanded according to the steps below. The operands must be structures.

1. The name of the first item immediately contained in the master variable is considered.

2. If each structure operand and target variable has an immediately contained item with the same identifier, an assignment statement is generated as follows: the statement is derived by replacing each structure operand and target variable with its immediately contained item that has this identifier. If any structure contains no such identifier no statement is generated. If the generated assignment is a structure or structure array assignment, BY NAME is appended. All generated assignment statements are given the condition prefix of the original assignment statement.

4. Steps 1 through 3 may generate further array and structure assignments. These are expanded in the same manner.

Examples

1. Suppose that the following three structures have been declared:

```
1 ONE,
  2 PART1,
    3 RED,
    3 WHITE,
    3 BLUE,
  2 PART2,
    3 GREEN,
    3 YELLOW,
    3 ORANGE(3),
  2 PART3,
    3 BLACK,
    3 WHITE;
```

```
1 TWO,
  2 PART1,
    3 RED,
    3 GREEN,
    3 WHITE,
  2 PART2,
    3 BLUE,
    3 YELLOW,
    3 ORANGE (3);
```

```
1 THREE,
  3 PART1,
    5 BLACK,
    5 WHITE,
    5 RED,
  3 PART2,
    5 YELLOW,
    5 WHITE,
    5 ORANGE (3),
    5 PURPLE;
```

Consider the following assignment:

```
ONE = TWO - 2 * THREE, BY NAME;
```

this generates:

```
ONE.PART1 = TWO.PART1 - 2 * THREE.PART1, BY NAME;
ONE.PART2 = TWO.PART2 - 2 * THREE.PART2, BY NAME;
```

these statements are replaced by:

```
ONE.PART1.RED = TWO.PART1.RED - 2
                * THREE.PART1.RED;
ONE.PART1.WHITE = TWO.PART1.WHITE - 2
                * THREE.PART1.WHITE;
ONE.PART2.YELLOW = TWO.PART2.YELLOW - 2
                * THREE.PART2.YELLOW;
ONE.PART2.ORANGE = TWO.PART2.ORANGE - 2
                * THREE.PART2.ORANGE;
```

The final assignment is expanded as an <array-assignment>.

2. The following example illustrates <array-assignment>:

Given the array A:

```
2    4
3    6
1    7
4    8
```

and the array B:

```
1    5
7    8
3    4
6    3
```

Consider the assignment statement:

```
A = (A+B)**2-A(1,1);
```

After execution, A has the value

```
7    74
93   189
9    114
93   114
```

Note that the new value for A (1,1), which is 7, is used in evaluating the expression for all other elements.

3. The following example illustrates string assignment:

Given:
A is a fixed-length string whose value is 'XZ/BQ'.
B is a varying-length string of maximum length 8 whose value is 'MAFY'.
C is a fixed-length string of length 3.
D is a varying-length string of maximum length 5.

Then in the statement:
C = A, the value of C is 'XZ/'.
C = 'X', the value of C is 'X '.
D = B, the value of D is 'MAFY'.

```
D=SUBSTR(A,2,3) ||SUBSTR(A,2,3), the value of D is 'Z/BZ/'.
SUBSTR(A,2,4)=B, the value of A is 'XMAFY'.
SUBSTR(B,2,2)='R', the value of B is 'MR Y'.
SUBSTR (B,2)='R', the value of B is  'MR  '.
```

7.3.2 Control Statements

7.3.2.1 The CALL Statement

The CALL statement invokes a procedure and causes control to be transferred to a specified entry point of the procedure.

Syntax

```
<call-statement> ::=

     CALL <entry-name>
          [(<argument> [,<argument>] . . .)]
           [TASK (<scalar-task-name>)];
```

Semantics

The <entry-name> specifies the entry point of the procedure invoked, and it may be GENERIC.

An argument is an expression, a constant, a variable or a computational built-in function name.

The TASK option specifies that the invoked and invoking procedures are to be executed independently.

When the TASK option is used, the task name, if given, is associated with the task created by the call.

Expressions in these options, as well as any argument expressions, are evaluated in the task in which the call is executed. This includes execution of any on-units entered as the result of the evaluations. The environment of the invoked procedure is established after evaluation of the expressions and before the procedure is invoked.

Examples

1.
```
CALL CRITICAL_PATH (A,B*C,D);
                    .
                    .
                    .
CRITICAL_PATH: PROCEDURE(ALPHA,BETA,GAMMA);
                    .
                    .
                    .
                   END;
```

2.
```
CALL PAYROLL (NAME, DATA, HRRATE);
```

7.3.2.2 The DELAY Statement

The DELAY statement causes execution to be suspended for a specified period of time.

Syntax

```
<delay-statement> ::= DELAY (<scalar-expression>);
```

Semantics

Execution of the DELAY statement causes the scalar expression to be evaluated and converted to an integer N and the task to be suspended for N seconds.

Execution resumes after N seconds only if the controlling task is of sufficiently high priority to be selected in preference to all other ready tasks.

Example

```
DELAY (10);
```

The controlling task is suspended for ten seconds.

7.3.2.3 The DO Statement

The DO statement delimits the start of a do-group and may specify repetitive execution of the statements within the group.

Syntax

```
<do-statement> ::=
    DO [<while-expression>|<iteration-expression>];
<while-expression> ::=
    WHILE (<scalar-expression>)
<iteration-expression> ::=
    {<variable>|<pseudo-variable>) =
     <iteration-specification>
     [,<iteration-specification>]...

<iteration-specification> ::=
    <expression 1>[REPEAT(<expression 4>)|
    {TO <expression 2> BY <expression 3>)|
    {BY <expression 3> TO <expression 2>)]
    [<while-expression>]
```

Restrictions:

The <variable> in the <iteration-expression> is a subscripted or unsubscripted scalar variable. It cannot be a task, event, or area variable.

Each expression in the <iteration-specification> list is a scalar expression.

If the BY-clause is omitted from the <iteration-specification> and the TO-clause appears, <expression3> is assumed to be one (1).

No TO-clause or BY-clause is allowed if <expression1> is a string constant.

If the TO-clause is omitted from the specification and the BY-clause appears, the iteration is performed until terminated by the <while-expression>, if present, or by some other statement within the group.

If both the TO-clause and BY-clause are omitted this form of the specification implies a single execution of the do-group with the control variable having the value of <expression1>. It implies no execution if the <while-expression> is false.

If the <variable> in the <iteration-specification> is not a string variable or a real arithmetic variable, the TO and BY-clauses cannot be used.

135

Semantics

If no <while-expression> or <iteration-expression> exists, the DO statement delimits the start of a do-group.

If a <while-expression> by itself exists, the DO statement delimits the start of a do-group and specifies repetitive execution defined by the following:

```
LABEL:  DO WHILE (<expression>);
        statement 1
        .
        .
        .
        statement n
        END;
NEXT:   statement m
```

The above is exactly equivalent to the following expansion:

```
LABEL:  IF (<expression>) THEN; ELSE GO TO NEXT;
        statement 1
        .
        .
        .
        .
        statement n
        GO TO LABEL;
NEXT:   statement m
```

If an <iteration-expression> exists, the DO statement delimits the start of a do-group and specifies controlled repetitive execution defined by the following:

```
LABEL:  DO VARIABLE (A1,...,AN)=
        <expression1>
         TO<expression2>
           BY <expression3>
         WHILE (<expression5>);
         statement 1
         .
         .
         .
LABEL1:  END;
NEXT:  statement 2
```

This is exactly equivalent to the following expansion:

```
LABEL:     TEMP1=A1;
           .
           .
           .
           TEMPN=AN;
           E1=<expression1>;
           E2=<expression2>;
           E3=<expression3>;
           V=E1;
LABEL2:    IF (E3 >= 0)&(V > E2) |
                (E3<0)&(V<E2)
                THEN GO TO NEXT;
           IF (<expression5>) THEN;
           ELSE GO TO NEXT;
           statement 1
           .
           .
           .
           statement m
LABEL1:  V=V+E3;
           GO TO LABEL2;
NEXT:      statement n
```

In the above expansion, A1,...,AN are expressions that may appear as subscripts of the control variable. TEMP1,...,TEMPN are compiler-created integer variables to which the expression values are assigned. V is equivalent to "VARIABLE" with the associated TEMP subscripts. E1, E2, and E3 are compiler-created variables having the attributes of <expression1>, <expression2>, and <expression3>, respectively. In the simplest cases, there are no subscripts (i.e., n = 0) and the first statement in the expansion is therefore E1 = <expression1>.

Additional rules for the above expansion follow:

1. The above expansion only shows the result of one <iteration-specification>. If the DO statement contains more than one <iteration-specification>, the statement labeled NEXT is the first statement in the expansion for the next <iteration-specification>. The second expansion is analogous to the first expansion in every respect. Thus, if a second <iteration-specification> appeared in the DO statement with <expression1> through <expression4>

represented by <expression5> through <expression8>, the second expansion would look like this:

```
NEXT:   E5 = <expression5>;
           .
           .
           .
          V=E5;
LABEL3:  IF ... THEN GO TO NEXT1;
          IF (<expression8> THEN;
            ELSE GO TO NEXT1;
          statement 1
           .
           .
           .
          statement m
LABEL4:   V=V+E7;
          GO TO LABEL3;
NEXT1:  statement n
```

2. If the <while-expression> is omitted, the IF statement immediately preceding statement 1 in the expansion is omitted.

3. If TO <expression2> is omitted, the statement E2 = <expression2> and the IF statement identified by LABEL2 are omitted.

4. If both TO <expression2> and BY <expression3> are omitted, all statements involving E2 and E3, and the statement GO TO LABEL2 are omitted.

5. Although the above expansions show a specific order in which the BY and TO-clauses are evaluated, no specific ordering is defined by the language.

The <while-expression> specifies that before each associated execution of the do-group, the expression is evaluated and, if necessary, converted to give a bit-string value. If any bit in the resulting string has the value '1', the iteration continues. If all bits have the value '0', the iterations associated with the current specification are terminated.

In the <iteration-specification> list, <expression1> represents the starting value of the control variable, <expression3> represents the increment to be added to the control variable after each iteration of the statements in the do-group. <expression2> represents the terminating value of the control variable. Iteration terminates as soon as the value of the control variable passes its terminating value. When the last specification is completed, control passes to the statement following the do-group.

Control may, under any circumstances, be transferred into a do-group from outside the do-group if iteration is not specified, i.e., no <while-expression> or <iteration-expression> exists. If the do-group is iterative and a GO TO statement transfers control to a statement inside the group, the results are undefined unless the GO TO is an abnormal return from a block that has been activated from within the do-group.

The effect of allocating or freeing the control variable is undefined.

Examples

```
1   DO INDEX = Z WHILE (A>B), 5 TO 10 WHILE (A=B), 100;
2.  DO I = 1 TO 9,11 TO 20;
3.  DO WHILE (P);
4.  DO;
5.  DO WHILE (TAX-DEDCT < ESTTAX * 4);
6.  DO I = 1 TO 100 BY 3;
7.  DO INDEX = V REPEAT(Z) WHILE(X<100);
```

7.3.2.4 The EXIT Statement

Syntax

```
<exit-statement> ::= EXIT;
```

Semantics

The EXIT statement causes immediate termination of the task that contains the statement and all tasks attached by this task. The execution of the EXIT statement in a major task is equivalent to a STOP statement.

7.3.2.4A The FREEZE Statement

Syntax

```
<freeze-statement> ::=FREEZE OPTIONS(<freeze-option>);
<freeze-option>::= PERMANENT/TEMPORARY
```

Semantics

A program must be "frozen" at run time using the FREEZE statement in order to become a library. The FREEZE statement must occur in the outer block of the program.

7.3.2.5 The GO TO Statement

The GO TO statement causes control of a program to be transferred to the specified statement within the program.

Syntax

```
<go-to-statement> ::=
    {{GO TO}|GOTO} <statement-label-name>;
```

Semantics

If a label variable is specified, the GO TO statement has the effect of a multi-way switch. The value of the label variable is the label of the statement to which control is transferred when the GO TO statement is executed.

Since the label variable may have different values at each execution of the GO TO statement, control may not always pass to the same statement. (Example 2 illustrates a GO TO statement used as a multi-way switch.)

A GO TO statement cannot pass control to an inactive block or to another task.

If control is transferred into a do-group that specifies iteration, the results are undefined unless the GO TO specifies an abnormal return from a block that has been activated from within the do-group.

THIS PAGE IS INTENTIONALLY LEFT BLANK FOR FORMATTING PURPOSES

A GO TO statement that transfers control from one block (D) to a dynamically encompassing block (A) has the effect of terminating block D, as well as all other blocks that are dynamically descendant from block A. On-units are reestablished, and automatic variables are freed. When a GO TO statement transfers control out of a procedure invoked as a function, the evaluation of the expression that contained the corresponding function reference is discontinued, and control is transferred to the specified statement.

A GO TO cannot terminate any block activated during a prologue or during execution of an ALLOCATE statement.

Examples

1.
```
GO TO A234;
          .
          .
          .
A234: ...
```

2. The following example illustrates a GO TO statement that effectively is a multi-way switch:

```
          .
          .
          .
     DECLARE  L LABEL (L1, L2) INITIAL (L2);
     GO TO L;
L1:  X = Y - 1;
     L = L2;
     GO TO MEET;
L2:  Y = X -1;
     L = L1;
MEET: CALL FUDGE (X, Y, Z);
     IF Z = LIMIT THEN GO TO L;
          .
          .
          .
```

3. The following procedure illustrates use of the GO TO statement with a subscripted label variable to effect a multi-way switch:

```
CALC: PROCEDURE (N1, N2);
       DECLARE SWITCH(3) LABEL INITIAL (CALC1,CALC2,CALC3);
       I=MOD(N1+N2,3)+1;
       GO TO SWITCH (I);
CALC1:...
        .
        .
        .
       RETURN;
CALC2:...
        .
        .
        .
       RETURN;
CALC3:...
        .
        .
```

```
END CALC;
```

7.3.2.6 The IF Statement

The IF statement specifies evaluation of an expression and conversion of the expression to a bit string, and a consequent flow of control dependent upon the value of the bit string.

Syntax

```
<if-statement> ::=
    IF <scalar-expression> THEN <unit-1> [ELSE <unit-2>]
```

Semantics

Each unit is a do-group, a begin block, or any statement other than DECLARE, END, ENTRY, FORMAT, or PROCEDURE. The unit may have its own labels and condition prefixes.

The IF statement is not itself terminated by a semicolon.

When the ELSE-clause (ELSE, and its following unit) is not specified, the <scalar-expression> is evaluated and, if necessary, converted to a bit string. If any bit in the resulting string has the value 1, <unit-1> is executed, and control passes to the statement following the IF statement. If all bits have the value 0, <unit-1>is not executed, and control passes to the next statement. When the ELSE-clause is specified, the expression is similarly evaluated. If any bit is 1, <unit-1> is executed, and control passes to the statement following the IF statement. If all bits have the value 0, <unit-2> is executed, and control passes to the next statement. The units may contain statements that specify transfer of control and so override these normal sequencing rules. A null bit string is regarded as having no bits with the value 1.

IF statements may be nested, that is, either <unit-1> or <unit-2>, or both, may themselves be IF statements. Each ELSE-clause is always associated with the innermost unmatched IF in the same block or do-group. Consequently, an ELSE or a THEN with a null statement may be required to specify a desired sequence of control.

A condition prefix to an IF statement enables (or disables) the condition only during evaluation of the scalar expression of the IF-clause. It is not applicable to either of the THEN or ELSE-clauses, which may have their own condition prefixes.

Examples

```
1.  IF A = Z THEN CALL X(0);
              ELSE CALL X(A);

2.  IF X > Y
    THEN IF Z = W
         THEN L: Y = 1;
         ELSE;
    ELSE (SIZE): Y = A;

3.  IF  A THEN GO TO M;
     GO TO N;
```

7.3.2.7 The RETURN Statement

The RETURN statement terminates execution of the procedure that contains the RETURN statement. If the procedure has not been invoked as a task, the RETURN statement returns control to the invoking procedure. The RETURN statement may also return a value.

Syntax

<return-statement> ::=

RETURN[(<scalar-expression>)];

Semantics

A RETURN statement without a <scalar-expression> can be used to terminate procedures not invoked as functions. Control is returned to the point logically following the invocation.

A RETURN statement without a <scalar-expression> represents the only form of the RETURN statement that can be used to terminate a procedure initiated as a task. If the RETURN statement terminates the major task, the FINISH condition is raised prior to the execution of any termination processes. If the RETURN statement terminates any other task, the completion value of the associated event variable (if any) is set to '1'B, and the status value is left unchanged.

A RETURN statement with a <scalar-expression> is used to terminate a procedure invoked as a function. Control is returned to the point of invocation and the value returned to the function reference is the value of the specified expression converted to conform to the attributes declared for the invoked entry point. These attributes may be explicitly specified at the entry point.

If control reaches an END statement corresponding to the end of a procedure, this END statement is treated as a RETURN statement with no <scalar-expression> for the procedure.

Example

```
A:  PROCEDURE (X,Y) FIXED;
        DECLARE (X,Y) FLOAT;
        .
        .
        .
        RETURN (A**2+Y**2);
        END;
B:  PROCEDURE;
        DECLARE A ENTRY RETURNS (FIXED);
        .
        .
        .
        R = A(P,Q);
        .
        .
        .
        END;
```

In the assignment statement (R = A(P,Q);), procedure B invokes procedure A as a function. Procedure B specifies that the scalar expression in the RETURN statement is to be evaluated. Since X and Y are floating-point variables and the PROCEDURE statement specifies that the value returned is to be fixed-point, the value of the expression is converted to fixed-point, and this value is returned to B.

7.3.2.8 The SORT Statement

The SORT statement causes a set of records to be sorted according to the options specified in the SORT statement.

Syntax

```
<sort-statement> ::= SORT <sort identifier> [ON]
                       <sort-option>;

<sort option> ::= <key option> [<input option>]
                    [<output option>] [<memory option>]

<key-option> ::= ((ASCENDING | DESCENDING) [KEY]
                   (<identifier>...))...

<input-option> ::= USING FILE (<file expression>) |
                     INPUT (<entry constant>)

<output-option> ::= GIVING FILE (<file expression>) |
                      OUTPUT (<entry constant>)

<memory-option> ::= ENVIRONMENT (TAPES=
                      <constant expression>,
                      CORESIZE=<constant expression>,
                     (DISKSIZE=<constant expression>) |
                      PACKSIZE=<constant expression>))
```

Restrictions:

The <sort-option>s may appear in any order.

The <sort-identifier> must be a data variable which describes the individual records to be sorted and may not be controlled or based.

Semantics

The <key-option> must appear. This option specifies the order in which the records are to be sorted and the keys to be used in the sort. The order of precedence of the keys is determined by the order of appearance of the key in the <key-option>. The sort keys are subject to the following rules:

a. Each key must be in or defined on the <sort-identifier>.

b. No variable length keys are allowed.

c. The records must all be of a fixed length, and the keys must be in the same location in each record.

The <input-option> may be either a file designation or an input procedure designation. If the <input-option> is not explicitly

stated, the <input-option> USING FILE (SYSIN) is assumed. If an explicit file designation is used, the file should be declared as an input file. The records of the file are then used as input to the sort. If an input procedure is used, however, the procedure must have SORTINPUT declared in the <option-list> of the procedure declaration. The input procedure is subject to the following rules:

a. The input procedure must have one and only one parameter. This parameter must be declared CHAR (*).

b. The input procedure must return a BIT (1) value.

c. A false value ('0'B) must be returned by the input procedure until the end of the input data is encountered. Then, a true value ('1'B) must be returned.

d. As long as a false value is being returned, the input procedure should insert the next record to be sorted into its parameter.

The <output-option> may be either a file designation or an output procedure designation. If the <output option> is not explicitly stated, the <output option> GIVING FILE (SYSPRINT) is assumed. If a file designation is used, the file should be declared as an output file. The sort will then write the sorted output to this file. If an output procedure is used, however, the procedure must have SORTOUTPUT declared in the <option-list> of the procedure declaration. The output procedure is subject to the following rules

a. The output procedure must have two parameters. The first parameter must be declared as BIT (1), and the second parameter must be declared CHAR (*).

b. The first parameter will contain a false value ('0'B) as long as the second parameter contains a sorted record. When all records have been returned to the output procedure by sort, the first parameter will contain a true value ('1'B), and the second parameter must not be accessed.

The <memory-option> specifies the number of tapes to be used by the sort as well as the core size and the disk size to be allocated for the sort. The options may appear in any order, and any of the options may be deleted. The default values for any unspecified memory options are: TAPES = 0, CORESIZE = 12000, DISKSIZE = 600000. If TAPES > 0, the value must be greater than or equal to 3 or less than or equal to 8.

Examples

Consider the following declarations:

```
DCL F FILE INPUT RECORD,
    G FILE OUTPUT RECORD;
DCL AR(10);
DCL CH CHAR(10)
    KEYS CHAR(5) DEFINED CH POSITION (6);
SORTIN:  PROC(A) RETURNS(BIT1))
    OPTIONS (SORTINPUT);
    DCL A CHAR(*);
        .
        .
        .
    END SORTIN;
DCL  1 AS,
       2 B CHAR(1),
       2 C CHAR(10);
SORTOUT:  PROC (B,A)  OPTIONS (SORTOUTPUT);
       DCL B  BIT(1),
           A  CHAR(*);
          .
          .
          .
       END SORTOUT;
```

The following <sort-statement>s would be legal using the above declarations:

```
1.  SORT  AS ON
        DESCENDING KEY (AS.B, AS.C)
        USING FILE (F) OUTPUT (SORTOUT)
        ENV (TAPES=4, CORESIZE=30000, DISKSIZE=100000);

2.  SORT  AR
        ASCENDING KEY (AR(1)) DESCENDING (AR(6))
        INPUT (SORTIN)
        ENV (CORESIZE=25000, TAPES=5);

3.  SORT  CH ON ASCENDING (KEYS) GIVING FILE(G);

4.  SORT CH ON ASCENDING (KEYS);
```

7.3.2.9 The STOP Statement

The STOP statement causes immediate termination of the major task and all subtasks.

Syntax

```
<stop-statement> ::= STOP;
```

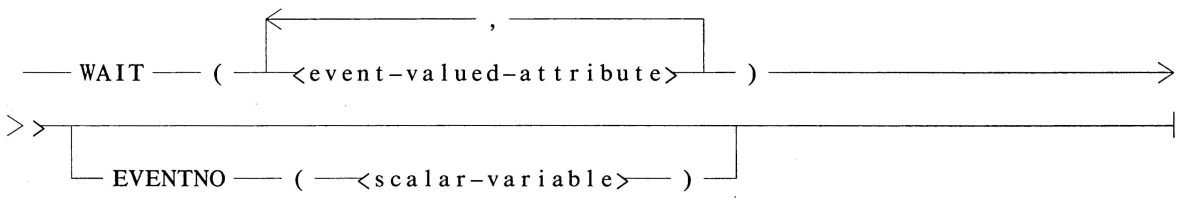
Semantics

Prior to any termination activity the FINISH condition is raised in the task in which the STOP is executed. On normal return from the finish on-unit, all tasks in the program are terminated.

7.3.2.10 The WAIT Statement

The WAIT statement causes suspension of a task until one of the listed <event-valued-attribute>s is caused.

Syntax



Semantics

If the EVENTNO clause is used, the ordinal position of the <event-valued-attribute> that caused activation of the task is stored in <scalar-variable>. <scalar-variable> may be a pseudovariable.

7.3.3 Program Structure Statements

7.3.3.1 The BEGIN Statement

The BEGIN statement is the heading statement of a begin block.

Syntax

```
<begin-statement> ::=BEGIN [OPTIONS
                          (<begin-option-list>)];

<begin-option-list> ::= DOUBLE
```

Semantics

A BEGIN statement is used in conjunction with an END statement.

See Section 6.3, Blocks for a discussion of blocks.

Examples

```
1.  ON OVERFLOW BEGIN;
                  .
                  .
                  .
                 END;
2.  (SIZE):  Q: PROCEDURE;
                  .
                  .
                  .
      (NOSIZE): A: BEGIN
                  .
                  .
                  .
                END;
                  .
                  .
                  .
              END;
```

The SIZE condition is enabled with the prefix to the PROCEDURE statement. This enabling is negated throughout the begin block with the prefix NOSIZE. On exit from the begin block, SIZE errors are again enabled because statements again are in the scope of the SIZE prefix.

THIS PAGE IS INTENTIONALLY LEFT BLANK FOR FORMATTING PURPOSES.

7.3.3.2 The END Statement

The END statement terminates blocks and do-groups.

Syntax

```
<end-statement> ::= END [<label>];
```

Semantics

If a label follows the END, the END statement terminates the block or do-group that is headed by the nearest preceding heading statement having that label. It also terminates all unclosed blocks and do-groups that are physically within that block or group.

If a label does not follow the END, the END statement terminates that group or block headed by the nearest preceding DO, BEGIN, or PROCEDURE statement for which there is no other corresponding END statement.

If control reaches an END statement terminating a procedure, it is treated as a RETURN statement.

If control reaches an END statement which terminates a begin block that is an on-unit, control is returned to the point specified for that particular interrupt.

If a label follows the END, that label may not be an element of a label array.

7.3.3.3 The ENTRY Statement

The ENTRY statement specifies a secondary entry point to a procedure.

Syntax

```
<entry-statement> ::=<entry-name>: ... ENTRY
    [(<parameter>[,<parameter>]...)]

     [RETURNS(<data-specification-attributes>)];
```

Semantics

The parameters are names that specify the parameters of the entry point.

The <data-specification-attributes> permitted for the RETURNS attribute of an ENTRY statement are the ARITHMETIC, STRING, AREA, OFFSET, ENTRY, and POINTER attributes. These attributes specify the characteristics of the value returned by the procedure when invoked as a function by any of the entry names. The value specified in the RETURN statement of the invoked entry is converted, if necessary, to conform to the specified attributes.

If an ENTRY statement has more than one label, each label is interpreted as if it were a single entry name for a separate ENTRY statement having the same parameter list. If <data-specification-attributes> are specified, they apply to all

entry names. If no <data-specification-attributes> are specified, arithmetic defaults are applied separately to each name.

Consider the Statement:

```
A:I:  ENTRY;
```

This statement is equivalent to:

```
A:    ENTRY;
I:    ENTRY;
```

An ENTRY statement cannot be internal to a begin block or to a do-group that specifies iteration.

A condition prefix cannot be prefixed to an ENTRY statement.

7.3.3.4 The PROCEDURE Statement

Syntax

```
<procedure-statement> ::=
    <entry-name>: ...PROCEDURE
        [(<parameter> [,<parameter>]...)]
        [OPTIONS (<option-list>)]
        [RECURSIVE]
        [RETURNS (<data-specification-attributes>)];

<option-list> ::= <option> [,<option>]...

<option>::=MAIN / DOUBLE / SORTINPUT / SORTOUTPUT /

    EXPORT (<entry-name>[AS<char-string-constant>]

    [,<entry-name>[AS<char-string-constant>]]...)
```

Semantics

The OPTIONS, RECURSIVE, and RETURNS options may appear in any order.

The PROCEDURE statement has the following functions:

1. Identifies a portion of program text as a procedure.
2. Defines the primary entry point to a procedure.
3. Specifies the parameters for the primary entry point.
4. Defines any special attributes of the procedure.
5. Specifies the attributes of the value that is returned if the procedure is invoked as a function at the primary entry point.

The <parameter>s are names that specify the parameters of the entry point. When the procedure is invoked, a relationship is established between the arguments of the invocation and the parameters of the invoked entry point. (See Section 9.5, Correspondence of Parameter Arguments).

OPTIONS specifies a list of options. OPTIONS may be specified for any procedure. DOUBLE allows the use of double precision for

arithmetic calculations, and SORTINPUT and SORTOUTPUT are discussed in Section 7.3.2.8, The Sort Statement.

The EXPORT option may only be used in the main (that is, outer block) <procedure-statement>. This option specifies which entry points can be used by other programs that call the program as a library. The <entry-names> used in the EXPORT option must be declared in the outer block of the program.

RECURSIVE specifies that the procedure may be invoked recursively (i.e., it may invoke itself). RECURSIVE does not have to be specified as all procedures are recursive by default.

The <data-specification-attributes> permitted for the RETURNS attribute of a PROCEDURE statement are the arithmetic, string, AREA, OFFSET, and POINTER attributes. These attributes specify the characteristics of the value returned by the procedure when invoked as a function at the primary entry point. (This rule applies to each entry name prefixed to the PROCEDURE statement.) The value specified in the RETURN statement of the invoked procedure is converted to the specified <data-specification-attributes>.

If a PROCEDURE statement has more than one entry name, the first name is interpreted as the only label on the statement. Each subsequent entry name is interpreted as a separate ENTRY statement having an identical parameter list and the same <data-specification-attributes> as written in the PROCEDURE. This equivalence is true only after multiple closure has been resolved. Defaults for the <data-specification-attributes> are applied separately for each such ENTRY statement and for the resulting PROCEDURE statement.

Examples

1.
```
A:I:  PROCEDURE;
```

Is effectively the same as:

```
A:  PROCEDURE;
I:  ENTRY;
```

The equivalence applies only after multiple closure has been resolved.

2.
```
B:  PROCEDURE;
    DCL A ENTRY RETURNS(FIXED);
        .
        .
        .
    D=A(X,Y);
    END B;
A:  PROCEDURE (B,C) RETURNS(FIXED);
    .
    .
    .
    RETURN (B*C + SIN (P));
    END A;
```

If procedure A is invoked as a function as it is in procedure B, then when control is returned to B, the expression (B*C + SIN (P)); is evaluated, converted to fixed point, and the value assigned to D in procedure B.

3. PROGRAM: PROCEDURE OPTIONS(MAIN, EXPORT(E1, E2 AS "E"));

In the following example, E1 and E2 are library entry points.

```
E1:  PROCEDURE
          :
     END E1;

 A:  PROCEDURE
          :
E2:  ENTRY
          :
     END A;
```

7.3.4 Data Declaration Statements

THIS PAGE IS INTENTIONALLY LEFT BLANK FOR FORMATTING PURPOSES

7.3.4.1 The DECLARE Statement

The DECLARE statement is a non-executable statement used for the specification of attributes of simple names.

Syntax

```
<declare-statement> ::=
      DECLARE [<level>] <identifier> [<attribute>] ...
      [,[<level>] <identifier> [<attribute>] ...] ...;
<level> ::= <integer-constant>
```

Restrictions:

Any number of identifiers may be declared as names in one DECLARE statement.

Attributes must follow the names to which they refer. The above format does not show factoring of attributes, which is allowable as explained later.

<level> is a non-zero unsigned integer constant. If it is not specified, level 1 is assumed.

A DECLARE statement may have a label prefix, but such use does not cause a declaration of the identifier as a label constant.

A DECLARE statement cannot have a condition prefix.

Semantics

All of the attributes given explicitly for a particular name must be declared together in one DECLARE statement. For variables with the FILE attribute, certain attributes may be specified in an OPEN statement. See Section 8.2, Opening a File.

The following attributes may not be specified more than once for the same name:

```
BASED (<scalar-locator-expression>)
CHARACTER (<length>)
DEFINED
DIMENSION
ENTRY
IN
INITIAL
LENGTH
LIKE
OFFSET (<area>)
PICTURE
POSITION
PRECISION
RETURNS
SIZE
```

The BASED attribute without the <scalar-locator-expression> may be used more than once for the same name. The BASED attribute without the <scalar-locator-expression> must not be specified with the BASED attribute with the <scalar-locator-expression> of the same name.

Attributes of EXTERNAL names, declared in separate blocks and compilations, must not conflict or supply explicit information that was not explicit or implicit in other declarations.

7.3.4.1.1 Declaration of Structures

The outermost structure is the major structure, and all contained structures are minor structures.

A structure is specified by declaring the major structure name and following it with the names of all contained elements. Each name is preceded by a level number, which is a non-zero unsigned integer constant. A major structure is always at level one and all elements contained in a structure (at level N) have a level number that is numerically greater than N, but they need not necessarily be at level N+1, nor need they all have the same level number.

A minor structure at level N contains all following items declared with level numbers greater than N up to but not including the next item with a level number less than or equal to N. A major structure description is terminated by the declaration of another item at level one, by the declaration of an item having no level number, or by the end of a DECLARE statement.

7.3.4.1.2 Factoring in DECLARE Statements

Attributes common to several name declarations can be factored to eliminate repeated specification of the same attribute for many identifiers. This factoring is achieved by enclosing the name declarations in parentheses, and following this by the set of attributes which are to apply. Level numbers also may be factored, but in such cases, the level number precedes the parenthesized list of name declarations. Factoring of attributes is permitted only in the DECLARE statement, but not within an ENTRY attribute declaration.

Syntax

```
<declare-statement> ::=
      DECLARE <declaration-list>;

<declaration-list> ::=
      <declaration> [, <declaration>] ...

<declaration> ::=
      [<level>] (<declared identifiers> |
      (<declaration-list>))[<attribute-set>]
```

Examples

1. DECLARE ((A FIXED, B FLOAT) STATIC, C CONTROLLED) EXTERNAL;

   This declaration is equivalent to the following:

   DECLARE A FIXED STATIC EXTERNAL, B FLOAT STATIC EXTERNAL, C CONTROLLED EXTERNAL;

2. DECLARE 1 A AUTOMATIC, 2 (B FIXED, C FLOAT, D CHAR (10));

   This declaration is equivalent to the following:

   DECLARE 1 A AUTOMATIC, 2 B FIXED, 2 C FLOAT, 2 D CHAR (10);

7.3.4.1.3 Multiple Declarations

Two or more declarations of the same identifier, internal to the same block, constitute a multiple declaration of that identifier only if they have identical qualification, including the case of two or more declarations of an identifier at level 1 (i.e., scalars of major structures).

Examples

1. DECLARE 1 A, 2 B, 2 B, 2 C, 3 D, 2 D;

   B has a multiple declaration.

151

7.3.4.2 DEFAULT Statement

The DEFAULT statement enables the programmer to specify default attributes for identifiers which are explicitly, contextually or implicitly declared.

Syntax

```
<default statement> ::= DEFAULT (SYSTEM | NONE |
      <default-set>)

<default-set> ::= (<selector-function>) (ERROR |
      <attribute-set> [,<attribute-set>...])

<selector-function> ::= <attribute-expression1> |
    <selector-function><or-symbol><attribute-expression1>

<attribute-expression1> ::= <attribute-expression2> |
      <attribute-exression1> & <attribute-expression>

<attribute-expression2> ::= <attribute-expression3> |
      ¬<attribute-expression3>

<attribute-expression3> ::= (<selector-function>) |
      <attribute-keyword> | <range-specification>

<range-specification> ::=
      RANGE(*) | RANGE (<identifier>) |
      RANGE (<letter> : <letter>)
```

<attribute-set> ::= <attribute>...

<or-symbol> ::= |

Restrictions:

The following attributes cannot be specified in a DEFAULT statement:

LIKE
ENTRY
RETURNS

If the label attribute is used in a DEFAULT statement, it cannot have a label list.

If the range specification RANGE (<letter>:<letter>) is used, then only single alphabetic letters (i.e., A through Z) may be specified. Furthermore, the second letter of the pair must be later in the alphabetic sequence than the first.

If a DEFAULT statement has a label prefix, the label will not be explicitly declared as a label constant.

A DEFAULT statement cannot have a condition prefix.

Semantics

By using the default statement, the programmer may specify some or all of the default attributes which are to be associated with all names declared within the scope of the DEFAULT statement which satisfy the <selector-function>. A DEFAULT statement can also be used to specify when standard system default rules will be used. The programmer may also use the DEFAULT statement to stop adding attributes to an identifier or to print out a warning message when any attributes are added to an identifier. Attributes which are supplied by a DEFAULT statement are considered to be explicitly declared attributes. All attributes which are not declared for an item are controlled by standard system default rules. See Section 4.14.2, Standard System Default Rules.

The <selector-function> indicates which identifiers within the DEFAULT statement's scope will be influenced by the DEFAULT statement.

If the <selector-function> is an expression consisting of attribute keywords (as in examples 2, 3, and 5), then those identifiers which satisfy the attribute conditions specified by the selector-function will be affected by the DEFAULT statement. For instance, if the DEFAULT statement in example 2 appeared in a PL/I program, then all identifiers within the scope of that DEFAULT statement which had either the attributes BINARY and FLOAT or DECIMAL would now be given the attribute PRECISION (9).

If the <selector-function> is a <range-specification>, it may have one of three forms, RANGE (<identifier>) specifies that all identifiers beginning with the sequence of characters which make up the identifier in the range specification will be influenced by the DEFAULT statement. For example, if RANGE (MON) were specified, then the identifiers MON, MON_KEY, MONOMOLECULAR and MONUMENTAL_IDENTIFIER would all be affected by the DEFAULT statement.

RANGE (<letter>:<letter>) indicates that all identifiers whose initial letter lies alphabetically between the two letters, inclusive, will be affected by the DEFAULT statement. For example, RANGE(A:C) includes any identifier whose initial letter is A, B or C.

RANGE(*) is equivalent to:

A through Z, $, #, @
and thus indicates all declared identifiers.

The <attribute-set> of the DEFAULT statement specifies which attributes will be added to an identifier as defaults. These attributes will be added if they do not conflict with attributes which are already declared for the identifier.

The <attribute-set> members are added to an identifier, satisfying the <selector-function>, according to their order of occurrence in the list.

The keyword SYSTEM specifies that the standard system default rules are to be used to add attributes to the identifier satisfying the <selector-function>.

The keyword ERROR means that if an identifier satisfies the <selector-function>, a warning message is to be printed.

The keyword NONE means that if an identifier satisfies the <selector-function>, no more attributes are to be added. Syntax errors will flag missing attributes.

The range of a DEFAULT statement is all identifiers which satisfy the <selector-function> of the DEFAULT statement.

The scope of a DEFAULT statement is defined as that block in which the DEFAULT statement occurs, as well as all blocks contained by that block.

If two DEFAULT statements have overlapping scopes, and an identifier satisfying the <selector-function> of both DEFAULT statements is declared within their overlapping scope, then the DEFAULT statement with the contained scope will be applied before the DEFAULT statement with the containing scope.

If two DEFAULT statements with the same range occur within the same block, and an identifier in this range is declared in that block, then both DEFAULT statements will be used. The order in which the DEFAULT statements occur will be the order in which they are applied to the identifier.

A DEFAULT statement is inconsistent if it contains:

a. Conflicting alternative attributes.

b. More than one occurrence of an attribute that may only appear once. See Section 7.3.4.1, The Declare Statement.

A PROCEDURE or ENTRY statement with more than one label prefix is replaced by the equivalent number of PROCEDURE and/or ENTRY statements, each with one of the label prefixes and each with identical options, before the DEFAULT statement is applied.

The system default rules are treated as a set of DEFAULT statements whose scope is the entire PL/I program. Hence, system default attributes are applied after all programmer DEFAULT statements have been applied, except where SYSTEM is explicitly specified by a DEFAULT statement. See Section 4.14.2, Standard System Default Rules.

Examples

1. DEFAULT (RANGE (A:F)) FIXED BINARY, PRECISION (9,0);

2. DEFAULT((BINARY & FLOAT) | (DECIMAL & FLOAT)) PRECISION (5);

3. DEFAULT (OFFSET) IN(TREE);

4. DEFAULT (RANGE (LAB)) LABEL, SYSTEM;

5. DEFAULT (DECIMAL & FLOAT) ERROR;

7.3.5 Error Control and Debug Statements

7.3.5.1 The DUMP Statement

The DUMP statement activates a system procedure which produces a program dump. When the program dump has been completed, control passes to the next executable statement.

Syntax

```
<dump-statement> ::= DUMP [(<dump-option>
                          [,<dump-option>]...)];
<dump-option> ::= ALL | BASE | CODE | FILE | ARRAY | DBS
```

If no dump options are specified in the DUMP statement, the default options will apply.

7.3.5.2 The ON Statement

The ON statement specifies the action to be taken when an interrupt occurs for the named condition.

Syntax

```
<on-statement> ::=
     ON <condition> [SNAP](<on-unit> | SYSTEM;)
```

Restrictions:

The <condition> may be any of those described in Appendix 2, Conditions.

The <on-unit> is an action specification, and it is either an unlabeled single simple statement (other than BEGIN, DO, END, RETURN, ENTRY, FORMAT, PROCEDURE, or DECLARE) or an unlabeled begin block. It may have a condition prefix. Since the on-unit itself requires a semicolon, no semicolon appears after the <on-unit> in the syntax notation above.

The <on-unit> may not be a RETURN statement, nor may a RETURN statement be internal to the begin block.

Semantics

An ON statement must be executed before its effect can be established.

The standard action to be taken for all ON conditions is defined by the language. When an interrupt takes place before an ON statement for that condition has been executed, standard system action is taken. This standard system action is described in Appendix 2, Conditions. An ON statement with SYSTEM stated specifies that standard system action is to be taken when an interrupt results from the occurrence of the specified condition.

An ON statement with an <on-unit> is a means for the programmer to specify action (other than standard system action), that is, execution of the <on-unit>, is to take place when an interrupt occurs for the specified condition. The <on-unit> is treated as a procedure internal to the block in which it appears.

If SNAP is specified, when the given condition occurs, a system program dump is produced.

Control can reach an <on-unit> only when an interrupt occurs for the condition associated with this <on-unit> in an ON statement or when a SIGNAL statement for the condition is executed.

If an action specification is established by execution of an ON statement, it remains in effect until it is overridden by another ON statement or REVERT statement specifying the same condition, or until termination of the block in which the ON statement is executed.

A single statement <on-unit> cannot contain a remote format item.

7.3.5.3 The REVERT Statement

A REVERT statement specifying a given condition is used to reestablish the action specification for the named condition as it was in the immediate, dynamically encompassing block. In the case of an initial external procedure, standard system action is established.

Syntax

<revert-statement> ::= REVERT <condition-list>;

The <condition-list> is a list of one or more conditions separated by commas. See Appendix 2, Conditions.

Examples

```
A:  PROCEDURE;
      .
      .
      .
ON1:   ON CONVERSION GO TO ERRSPEC;
         .
         .
         .
       CALL B;
          .
          .
          .
         B:  PROCEDURE:
               .
               .
               .
         ON2:ON CONVERSION;
               .
               .
               .
               REVERT CONVERSION;
               .
               .
               .
               END B;
      .
      .
      .
Conditions.
      .
      .
      .
      END A;
```

Unless it is stated otherwise, the condition CONVERSION always is enabled. If a conversion error occurs prior to execution of statement ON1, an interrupt with standard system action takes place.

If a conversion error occurs after execution of ON1 and prior to execution of statement ON2, an interrupt takes place and control transfers to the statement GO TO ERRSPEC.

If a conversion error occurs after execution of ON2 and prior to the REVERT statement, an interrupt takes place resulting in the execution of no program error code since no <on-unit> is specified.

When the REVERT statement is executed, the effect of the statement ON2 is nullified, and statement ON1 again becomes effective.

7.3.5.4 The SIGNAL Statement

The SIGNAL statement simulates the occurrence of the named condition and causes an interrupt if the condition is enabled. It may be used to test the action specification of the current ON statement. The result is that the <on-unit> will be executed.

Syntax

```
<signal-statement> ::= SIGNAL <condition>;
```

Examples

```
1.      X:  PROCEDURE;
              .
              .
              .
          ON1:  ON ENDFILE (DATIN) Y,Z = 0;
              .
              .
              .
           S1:  SIGNAL ENDFILE (DATIN);
              .
              .
              .
          ON2:  ON ENDFILE (DATIN) SYSTEM;
              .
              .
              .
           S2:  SIGNAL ENDFILE (DATIN);
              .
              .
              .
              END X;
```

Statement S1 causes an interrupt in the same way as if an attempt to read past a file delimiter had actually occurred. Control is transferred to the statement Y,Z = 0; in the ON1 statement.

When statement S2 causes an interrupt, control is transferred to the ON2 statement, and standard system action is taken.

```
2.    ON CONDITION (TAX) TAXCT = TAXCT+1;
        .
        .
        .
      SIGNAL CONDITION (TAX);
```

The ON statement establishes an action for the programmer-specified condition TAX. This condition can occur only when a SIGNAL statement causes the condition to occur.

7.3.6 Input/Output Statements

7.3.6.1 The CLOSE Statement

The CLOSE statement makes the named file inaccessible in the current task.

Syntax

```
<close-statement> ::=
    CLOSE<options-group> [,<options-group>]...;

<options-group> ::=
    FILE(<file-name>)
   [SUBFILE(<scalar-expression>)]
    [VOLUME | {{ENVIRONMENT | OPTIONS} (<close-option>)}]

<close-option> ::= PURGE | LOCK | CRUNCH | NOREWIND
```

Semantics

The options may appear in any order within an <options-group>.

The FILE option specifies the file to be closed. Several files may be closed by one CLOSE statement via multiple <options-group>s forming a multiple CLOSE statement.

Closing an unopened file, or a closed file, has no effect.

The SUBFILE clause has no effect on a CLOSE statement for non-PORT files. For PORT files, the value of <scalar-expression> in the SUBFILE clause determines which subfile is to be closed. If this value is zero, all open subfiles are closed. If this value is greater than zero but less than or equal to the value of MAXSUBFILES, only the specified subfile is closed. A bad subfile index is ignored.

Because only one meaning exists for CLOSE on a PORT file, any <close-option>s specified in the CLOSE statement are ignored.

All I/O event variables associated with operations on the file that have not been completed before the file is closed are set complete, with a status value of 1 if not already non-zero.

When no VOLUME, ENVIRONMENT, or OPTIONS option appears, an end-of-file mark, record, or label is placed in the file and the file is closed. A temporary disk file is removed from the directory, and a tape file is rewound.

When the VOLUME option appears, the file must be a tape file. On multi-file tape reels, the tape is positioned after file closure at the next file on the reel.

When the ENVIRONMENT(PURGE) or OPTIONS (PURGE) option appears, the file is closed, purged, and released to the system. A disk file is removed from the directory.

When the ENVIRONMENT(LOCK) or OPTIONS (LOCK) option appears, the file is closed, locked, and saved by the system. A tape file is rewound.

When the ENVIRONMENT (CRUNCH) or OPTIONS (CRUNCH) option appears , the file must be a disk file. The file is closed, locked, and the unused portion of the last row of disk space (beyond the end-of-file indication) is returned to the system. For KEYED disk files CRUNCH implies LOCK.

When the ENVIRONMENT (NOREWIND) or OPTIONS (NOREWIND) option appears, the file must be a tape file. The file is closed and the tape remains positioned at the beginning of the next file. This option is identical to the VOLUME option.

Examples

```
CLOSE FILE(MASTER);
CLOSE FILE(TABLEA)OPTIONS(LOCK),
      FILE(TABLEB)OPTIONS(PURGE);
CLOSE FILE(GN)VOLUME;
```

7.3.6.2 The DELETE Statement

The DELETE statement deletes a record from an UPDATE file.

Syntax

```
<delete-statement> ::= DELETE <option-list>;
<option-list> ::=
    FILE (<file-name>) [KEY (<scalar-expression>)]
```

Semantics

The options may appear in any order.

The FILE (<file-name>) option specifies an UPDATE file and must be specified.

The KEY option must be specified if the file is a DIRECT UPDATE file. It cannot be specified otherwise. The expression is converted to a character string and determines which record is to be deleted.

If the file is a SEQUENTIAL UPDATE file, the record to be deleted is the last record that was read.

The DELETE statement unlocks a record only if that record had been locked in the same task in which the DELETE appears.

The DELETE statement can cause implicit opening of a file. The system file attribute MYUSE is set equal to 'UPDATE' for an implicitly opened file.

Example

```
DELETE FILE(ALPHA) KEY (DKEY);
```

This statement causes the record identified by DKEY to be deleted from the data set associated with the file ALPHA. If the record was previously locked in the same task, it is unlocked.

7.3.6.3 The DISPLAY Statement

The display statement causes a message to be displayed at the supervisory console. A response may be requested.

Syntax

```
<display-statement> ::=

     DISPLAY (<scalar-expression>)
     [REPLY (<scalar-character-variable>)]
```

Semantics

In the following, the format of the DISPLAY statement without the option is referred to as option 1, and with the REPLY option as option 2.

Execution of the DISPLAY statement causes the <scalar-expression> to be evaluated and, where necessary, converted to a varying character string. This character string is the message to be displayed.

In option 2, the character variable receives a string that is a message to be supplied by the operator.

Example

```
DISPLAY ('END OF JOB');
```

This statement causes the message, "END OF JOB" to be displayed at the supervisory console.

7.3.6.4 The FORMAT Statement

The FORMAT statement specifies a format list for use with data transmitted under edit direction.

Syntax

```
<format-statement> ::=
    <label>: [<label>:] ... FORMAT (<format-list>);
```

Semantics

The <format-list> is as described for use with an EDIT-directed data specification. See Section 8.4.6.1, Format Lists.

At least one <label> is required.

A GET or PUT statement may include a remote format specification. See Section 8.4.6.3.4, Remote Format Item.

The remote format item and the FORMAT statement must be internal to the same block.

A FORMAT statement encountered in sequential flow of control is treated as a no-operation.

It is an error to attempt to transfer control to a FORMAT statement by means of a GO TO statement.

7.3.6.5 The GET Statement

The GET statement normally causes values from a data set to be assigned to variables specified in a data list. Alternatively, the values may come from a character-string variable.

Syntax

```
<get-statement> ::=GET <option-list>;

<option-list> ::=

        {[FILE(<file-name>) [COPY] [SKIP
        [(<scalar-expression>)]]] |

       STRING(<scalar-character-string-variable>)}
            [<data-specification>];
```

Semantics

If neither the FILE(<file-name>) option nor the STRING (<scalar-character-string-variable>) option appears, the file option FILE(SYSIN) is assumed. (Default <file-name> is SYSIN.)

The <data-specification> must appear unless the SKIP option is specified.

The options may appear in any order.

The <file-name> refers to the file which is to provide the values. It must be a STREAM INPUT file.

The <scalar-character-string-variable> refers to the character string that is to provide the data to be assigned to the data list. This name may be a reference to a character string built-in function. Each GET operation using this option always begins at the beginning of the specified string. If the number of characters in this string is less than the total number of characters implied by the data specification, the ERROR condition is raised.

When the STRING option is used under data-directed transmission, the ERROR condition is raised if an identifier within the string does not have a match within the data specification.

For the rules concerning data specification see Section 8.4.3, Data Lists.

If the FILE<file-name> option refers to a file that is not open, the file is implicitly opened for the stream input transmission; the system file attribute MYUSE is set equal to 'INPUT' for the file.

The COPY option, which may only be used with the FILE option, specifies that the source data, as read, is to be written, without alteration, on the standard installation print file.

The SKIP option, which may only be used with the FILE option, causes a new current line to be defined for the data set. The expression, if present, is converted to an integer W, which must be greater than zero. The data set is positioned at the start of the W-th line relative to the current line. If the expression is omitted, SKIP(1) is assumed. The SKIP option always is executed before any data is transmitted. If the first action on a stream input file is a GET SKIP statement, the data set is positioned at the start of the first line.

Examples

1. GET LIST (A,B,C);

Specifies the list directed transmission of the values to be assigned to A, B, and C from the file named SYSIN.

2. GET FILE (BETA) EDIT (X,Y,Z) (A(5), F(5,2), A(10));

Specifies the edit-directed transmission of the values assigned to X, Y, and Z from file BETA.

3.
```
C:PROC;
      ON ENDFILE(SYSIN) GOTO X;
      L:GET COPY SKIP;
      GOTO L;
X:END C;
```

Reads all cards in the default input file and prints them on the SYSPRINT output file.

7.3.6.6 The LOCATE Statement

The LOCATE statement, which applies to output files, causes allocation of the specified BASED variable in a buffer. It may also cause transmission of a BASED variable previously allocated in a buffer.

Syntax

```
<locate-statement> ::= LOCATE <variable> <option-list>;

<option-list> ::=

     FILE (<file-name>)
     [SET(<scalar-pointer-variable>)]
     [KEYFROM(<scalar-expression>)]
```

Restrictions:

The <variable> must be an unsubscripted level 1 BASED variable.

Semantics

The options in the <option-list> may appear in any order.

The FILE(<file-name>) clause specifies the file involved. This clause must appear.

Execution of the LOCATE statement causes the specified based variable to be allocated in the buffer. Components of the based variable that have been given the INITIAL attribute, or components specified in REFER options, are initialized. A pointer value is assigned to the pointer variable named in the SET option or, if the SET option is omitted, to the pointer variable specified in the declaration of the BASED variable. The pointer value identifies the record in the buffer. If the pointer variable is an OFFSET variable, the pointer value is implicitly converted. After execution of the LOCATE statement, values may be assigned to the based variable for subsequent transmission to the file, which will occur immediately before the next LOCATE, WRITE, or CLOSE operation on the file, at which time the record is freed.

IF the KEYFROM (<scalar-expression>) option appears, the value of the <scalar-expression> is converted to a character string and is used as the key of the record when it is subsequently written.

If the FILE(<file-name>) clause refers to a file that is not open, the file is implicitly opened; the system file attribute MYUSE is set equal to 'OUTPUT' for the file.

Example

```
LOCATE ALPHA SET (REC-POINT) FILE (BETA);
```

The BASED variable ALPHA is allocated in a buffer and the <scalar-pointer-variable> REC-POINT is set to identify ALPHA in the buffer. Values may subsequently be assigned to ALPHA and the record will be written in the data set associated with the file BETA when a subsequent LOCATE or WRITE statement is executed for file BETA or if BETA is closed, either explicitly or implicitly.

7.3.6.7 The OPEN Statement

The OPEN statement completes the specification of attributes for a file and explicitly opens a file if it has not been previously opened. If the file has already been opened, the statement is ignored. Attribute specifications in the OPEN statement take precedence over attribute specifications in the file declaration.

Syntax

```
<open-statement> ::= OPEN <options-group>
                     [,<options-group>]...;

<options-group> ::=
    FILE(<file designator>)
    [TITLE (<file-title>)]
    [{ENVIRONMENT | OPTIONS}
    (<system-file-attribute-specification-list>)]
    [PRINT]
    [INPUT | OUTPUT | UPDATE ]
    [STREAM | RECORD]
    [LINESIZE(<scalar-expression>)]
    [PAGESIZE(<scalar-expression>)]
    [SUBFILE(<scalar-expression>)]
    [SEQUENTIAL | DIRECT]
    [TAB(<scalar-expression list>)]
    [WAIT | OFFER | AVAILABLE]

<file-title> ::= <character-string>

<system-file-attribute-specification-list>::=
    <system-file-attribute-specification>
    [,<system-file-attribute-specification>]...

<system-file-attribute-specification> ::=
    <system-file-attribute> = <scalar-expression> |
    <Boolean-valued-system-file-attribute>
```

Semantics

The INPUT, OUTPUT, UPDATE, STREAM, RECORD, DIRECT, SEQUENTIAL, KEYED, PRINT, and ENVIRONMENT (or OPTIONS) options specify attributes which may augment or override the attributes specified in the file declaration.

The options may appear in any order within a group.

The FILE (<file-name>) clause specifies which file is to be opened. The clause must appear once in each options group. An OPEN statement can be used to open several files, by multiple use of the options-group. This is known as a multiple statement, and is equivalent to a list of single OPEN statements, where the options groups in the list are taken in left-to-right order from the multiple OPEN statement.

If a file has been opened in a task and not subsequently closed, then reopening this file in the same task or a descendant task has no effect on the file. All options (including TITLE) are evaluated whether or not they conflict with the options of the previous open, but they are not used. If a file has been opened and subsequently closed, it may be reopened.

The TITLE option may be used to specify the name by which the system is to identify the file. The default TITLE is the <file-name> in the FILE option. In the case of a parameter, the corresponding argument identifier is the default title.

The ENVIRONMENT or OPTIONS option is identical in form to the ENVIRONMENT attribute as discussed in Section 4.8.7, Environment Attribute.

The LINESIZE option can be specified only for a STREAM OUTPUT file. The expression is evaluated, converted to an integer, and used as the length of a line during subsequent operations on the file. New lines may be started by use of the printing and control format items or by options in a GET or PUT statement. If an attempt is made to position a file past the end of a line before explicit action to start a new line is taken, a new line is automatically started, and the file is positioned to the start of this new line.

The LINESIZE option cannot be specified for an INPUT file. The linesize taken into consideration whenever a SKIP option appears in a GET statement is the linesize that was used to create the file.

The PAGESIZE option can be specified only for a STREAM PRINT file. The expression is converted to an integer and used as the number of lines on a page. During subsequent output to the file, new pages may be started by use of the PAGE format item or PUT PAGE statement option. If a page overflows before action to start a new page is given, the ENDPAGE condition is raised.

The SUBFILE option has no effect on OPEN statements for non-PORT files. For PORT files, the value of <scalar-expression> in the SUBFILE clause specifies which subfile is to be opened. If this value is zero, the OPEN statement opens all subfiles.

Attempting to set conflicting language file attributes will generate a syntax error of level three. For a list of compatible language file attributes, refer to Section 8.1, File Attributes.

Attempting to set a system file attribute inside the environment or options clause that conflicts with a language attribute will be ignored.

Syntax for setting system file attributes inside the OPTIONS or ENVIRONMENT clause in an OPEN statement is the same as for a file declaration except that the specifications need not be constant. In the case of system file attributes that require mnemonics, they must be used. This can be done by specifying a string constant or a string variable which has been assigned the correct value. If a constant is used, a compile time check is made to see if it is a valid mnemonic for the specified file attribute. If not, an error of level three is given. If a variable is used, it is checked at run time, and if it is an invalid mnemonic, a non-fatal run time error is given.

Attempting to set a read-only attribute inside the ENVIRONMENT or OPTIONS clause will cause a syntax error error of level three to be generated and the specification will be ignored.

When a TAB (<scalar-expression-list>) clause appears the <scalar-expression> list defines the tab columns to be used for list and data directed PUT statements or for tab format items. The expressions must be ascending, positive and less than the file line size. All erroneous expressions will be ignored.

The WAIT, AVAILABLE, and OFFER options apply only to PORT files. WAIT is the default value; the subfile or subfiles are offered for matching, and the program is suspended, until a matching subfile is found for all offered subfiles. OFFER causes the subfile or subfiles to be offered for matching; the program is resumed without waiting for the subfiles to be matched. AVAILABLE causes the subfiles to be matched only to complementary subfiles that are already offered. If a match is found for a particular subfile, that subfile is opened; if no match is found, a NOFILEFOUND result is returned for that subfile, and the subfile is not left offered for subsequent matching.

Examples

1. OPEN FILE (ALPHA), FILE (BETA) TITLE ('WORKFILE');

   The files ALPHA and BETA are opened. The data set associated with BETA is identified through use of the external name WORKFILE, whereas ALPHA is identified with a data set through use of the external name ALPHA.

2. OPEN FILE (MASTER) UPDATE;

   The file MASTER is opened as an UPDATE file. MASTER is by default the name used to associate a data set with the file.

3. OPEN FILE (SYSPRINT) LINESIZE (132) TAB (20,40,105);

   If the TAB settings are not supplied, the following default TABS will be used (1, 25, 49, 73, 47, 121, ...).

7.3.6.8 The PUT Statement

The PUT statement causes the transmission of data and/or the execution of control options. Data items transmitted are the character string representations of values of expressions that are assigned to a data set or to a designated character string variable.

Syntax

```
<put-statement> ::= PUT <option-list>;

<option-list> ::=
   {[FILE(<file-name>)] | STRING
   (<scalar-character-string-variable>)}
         [<data-specification>] [PAGE]
         [SKIP [(<scalar-expression>)]]
         [LINE (<scalar-expression>)]
```

Semantics

The PAGE, SKIP, and LINE options cannot be used with the STRING option.

If neither the FILE(<file-name>) option nor the STRING (<scalar-character-string-variable>) option appears, the file option FILE(SYSPRINT) is assumed.

The <file-name> refers to a file that is to receive the values. It must be a STREAM OUTPUT file.

The <scalar-character-string-variable> refers to the character string variable or pseudo-variable that is to receive the values. After appropriate conversion, the data specified by the <data-specification> is assigned to the string starting at the leftmost character (leftmost specified character in the case of a SUBSTR pseudo-variable). Any subsequent PUT statement will cause assignment to begin at the same place. If the string is not long enough to accommodate the data, the ERROR condition is raised.

The options may appear in any order. The PAGE and LINE options can be specified for PRINT files only. All of the options take effect before transmission of any values defined by the <data-specification>, if given. Of the three options PAGE, SKIP, and LINE, only PAGE and LINE may appear in the same PUT statement, in which case, the PAGE option is applied first.

The PAGE option causes a new current page to be defined within the file. If a data specification is present, the transmission of values occurs after the definition of the new page. The page remains current until the execution of a PUT statement with the PAGE option, until a PAGE format item is encountered, or until an ENDPAGE interrupt results in the definition of a new page. A new current page implies line one.

The SKIP option causes a new current line to be defined for the file. The expression, if present, is converted to an integer W, which for non-print files must be greater than zero. The file is positioned at the start of the W-th line relative to the current line. If the expression is omitted, SKIP(1) is assumed.

For PRINT files, W may be less than or equal to zero. In this case, the effect is that of a carriage return with the same current line. If less than W lines remain on the current page when a SKIP(W) is issued, ENDPAGE is raised.

The LINE option causes a current line to be defined for the file. The expression is converted to an integer W. If W specifies the current line of the most recent PUT statement, no new current line is established. If W is greater than the current line, blank lines are inserted so that the next line will be the W-th line of the current page. If more than W lines have already been written on the current page or if W exceeds PAGESIZE of the file, the ENDPAGE condition is raised. If W is less than or equal to zero, it is assumed to be one.

If the FILE(<file-name>) option refers to a file that is not open, the file is opened implicitly for stream output; the system file attribute MYUSE is set equal to 'OUTPUT' for the file.

Examples

1. PUT DATA (A,B,C);

   Specifies the data-directed transmission of the values A, B, and C to the file SYSPRINT.

2. PUT FILE (LIST) EDIT (X,Y,Z) (A(10)) PAGE;

   Specifies that a new page is to be defined for the print file list. The values of X, Y, and Z are placed starting in the first print position of the new page. Each of the values will use the A(10) format item.

7.3.6.9 The READ Statement

The READ statement causes a record to be transmitted from a RECORD INPUT or RECORD UPDATE file to a variable or buffer.

Syntax

```
<read-statement> ::= READ <option-list>;

<option-list> ::=
     FILE (<file-name>)
     { [INTO (<variable>)]|
       [SET (<scalar-pointer-variable>)]|
       [IGNORE (<scalar-expression>)]}

       [INDEX(<scalar-expression>)]

     { [KEY (<scalar-expression>)]|
       [KEYTO (<scalar-character-string-variable)] }

       [SUBFILE (<scalar-expression>)]


       [DONTWAIT (<scalar-expression>)]
```

Semantics

The options may appear in any order.

The FILE (<file-name>) clause specifies the file from which the record is to be read. This clause must appear. If the file specified is not open, it is implicitly opened; if the system file attribute MYUSE is not already 'UPDATE', it is changed to 'INPUT'.

The INTO(<variable>) option specifies an unsubscripted level 1 variable into which the record is to be read. It cannot be a parameter, nor can it have the DEFINED attribute.

The INDEX option may be used to specify a particular record of a tape or disk file to be read.

The KEY and KEYTO options can be specified for KEYED files only.

The KEY(<scalar-expression>) option must appear if the file is DIRECT. The expression is converted to a character string that determines which record is to be read.

The KEYTO(<scalar-character-string-variable>) option may be given only if the file is SEQUENTIAL KEYED. It specifies that the key of the record is to be copied into the string variable, which may be a pseudo-variable. This copying follows the rules for character string assignment, and if proper assignment cannot be made the KEY condition is raised. The key is the same key that was specified in the KEYFROM option when the record was written. KEYTO and KEY may not appear in the same READ statement.

The SET option specifies that the record is to be read into a buffer and that a pointer value is to be assigned to the named pointer variable. The pointer value identifies the record in the buffer.

The IGNORE option may be specified for SEQUENTIAL INPUT and SEQUENTIAL UPDATE files. The expression in the IGNORE option is evaluated and converted to an integer value. If this value, N, is greater than zero, N records are ignored. A subsequent READ statement for the file will access the (N+1)th record. If N <=0, no records are ignored. A READ statement without an INTO, SET, or IGNORE option is equivalent to a READ with an "IGNORE(1)".

A file with the KEYED attribute being accessed sequentially may be positioned by issuing a READ statement with the KEY option. The specified key will be used to identify the record required. Thereafter, records may be read sequentially from that point by use of READ statements without the KEY option. This applies to INPUT and UPDATE files.

For SEQUENTIAL files, two positioning statements can be used, with the following formats:

```
READ FILE (<file-name>)
    INTO (<variable>) KEY (<expression>);

and

READ FILE ( file name )
SET ( pointer-variable ) KEY ( expression)
```

The SUBFILE and NOWAIT clauses may be specified only for PORT files.

The value of <scalar-expression> in the SUBFILE clause determines which subfile in a PORT file is to be read. If this value is zero, a READ statement may read from any subfile. If <scalar-expression> is a variable or pseudovariable, after execution of the READ statement, the subfile number of the origin of the record that was read is stored in the variable or pseudovariable.

The value of <scalar-expression> in the DONTWAIT clause is converted to type BIT(1). If the resulting value is '0'B (or if the DONTWAIT clause is not specified), program execution waits until the read is finished. If the resulting value is '1'B, a read is done only if data are ready to be read. If <scalar-expression> is a variable or pseudovariable:

1. '1'B is stored in the variable or pseudovariable if the read was not done because no data were ready to be read.

2. '0'B is stored in the variable or pseudovariable if the read was done.

Execution of a READ statement with a SUBFILE clause or a DONTWAIT clause for non-PORT files results in the UNDEFINEDFILE condition.

For SEQUENTIAL files, only the first form shown immediately above can by used.

Examples

1. READ FILE (ALPHA) SET (REC-IDENT);

   The next record from the file ALPHA is made available and the pointer variable REC-IDENT is set to identify the record in the buffer.

2. READ FILE (BETA) KEY (VALUE) INTO (WORK);

   The record identified by the key value is transmitted from the file BETA into the variable WORK.

7.3.6.10 The REWRITE Statement

The REWRITE statement causes replacement of an existing record in a data set referred to be an UPDATE file.

Syntax

```
<rewrite-statement> ::= REWRITE <option-list>;
<option-list> ::=
     FILE (<file-name>) [KEY (<scalar-expression>)]
        [FROM (<variable>)]
```

Semantics

The options may appear in any order.

The FILE(<file-name>) option specifies the file involved. If it refers to a file that is not open, the file is opened implicitly.

The KEY(<scalar-expression>) option must appear if the file is a DIRECT UPDATE file, and it cannot appear otherwise. The expression is converted to a character string and determines which record is written.

The FROM(<variable>) option may be given to specify an unsubscripted level 1 variable which is to be used as the source for the record. The FROM(<variable>) option must be specified for a DIRECT UPDATE or SEQUENTIAL UNBUFFERED UPDATE file. The FROM option can be omitted for SEQUENTIAL BUFFERED UPDATE files only, in which case, the file is updated from the buffer associated with the file.

To access such a file, the sequence of statements should normally be READ followed by REWRITE.

For SEQUENTIAL UPDATE files, consecutive rewrites without reads increment the record pointer.

Example

```
REWRITE FILE (ALPHA);
```

The last record read from the data set associated with file ALPHA is rewritten from the buffer.

7.3.6.11 The WRITE Statement

The WRITE statement transfers the contents of a variable in internal storage to a record in a RECORD file.

Syntax

```
<write-statement> ::=WRITE <option-list>;
<option-list> ::=
     [FILE(<file-name>)] [FROM (<variable>)]
     [PAGE] [SKIP[(<scalar-expression>)]]
     [INDEX(<scalar-expression>)]
     [KEYFROM(<scalar-expression>)]
     [SUBFILE(<scalar-expression>)]
     [DONTWAIT(<scalar-expression>)]
```

Semantics

The file referenced must be either a RECORD OUTPUT or a DIRECT RECORD UPDATE file.

The options may appear in any order.

The FILE (<file-name>) option specifies the file in which the record is to be written. If this option is omitted, SYSPRINT is assumed. When this statement is executed the specified file is implicitly opened if it was not open at that time. If the system file attribute MYUSE is not already 'UPDATE', it is changed to 'OUTPUT'.

The FROM (<variable>) option specifies the unsubscripted level 1 variable from which the record is to be written.

The PAGE option can be specified for printer files only. The effect of this option is to advance the paper to the top of the following page before writing the record.

The SKIP option causes a new current line to be defined within the file. The expression, if present, is converted to an integer, W, which for non-print files must be greater than zero. The file is positioned at the start of the W-th line (i.e., record) relative to the current line. If the expression is omitted, SKIP(1) is assumed.

For printer files, W may be less than or equal to zero. In this case, the effect is that of a carriage return with the same current line. If less than W lines remain on the current page when a SKIP(W) is issued, the ENDPAGE condition is raised.

The INDEX option causes a record to be written at a specific location in a tape or disk file. The expression is converted to an integer, W, which must be greater than or equal to zero. The file is positioned at the start of the W-th record in the file before the write is performed, i.e., file is zero relative.

The expression in the KEYFROM option is converted to a character string and associated with the record as its key.

A record written to a line printer file may employ special carriage control characters in the first character position of the record. This character will control paper positioning before or after the record is written without use of the PAGE or SKIP option. The details for this facility will be specified after system file attributes are specified.

The SUBFILE and DONTWAIT options may be specified only for PORT files.

The value of <scalar-expression> in a SUBFILE clause specifies the subfile in a PORT file to which the WRITE statement writes. If this value is zero, a WRITE statement may write to all subfiles.

The <scalar-expression> in the DONTWAIT clause is converted to type BIT(1). If the resulting value is '0'B (or if the DONTWAIT clause is absent), program execution waits until the write is finished. If the resulting value is '1'B, a write is done only if a buffer is available. If <scalar-expression> is a variable or pseudovariable:

1. '1'B is stored in the variable or pseudovariable if the write was not done because no buffer was available.

2. '0'B is stored in the variable or pseudovariable if the write was done.

If a WRITE statement with a SUBFILE clause or a DONTWAIT clause is executed for a non-PORT file, an UNDEFINEDFILE condition results.

Examples

1. WRITE FILE(BETA) FROM(UPDATE) KEYFROM(ONKEY);

   Specifies that the record UPDATE be written as the next record in the file BETA. The key identifying the record in the file is taken from ONKEY.

2. WRITE PAGE FILE(SYSPRINT) FROM(BUF);

   Specifies that the record BUF be written as the next record in the printer file SYSPRINT after advancing to the top of the page.

THIS PAGE IS INTENTIONALLY LEFT BLANK FOR FORMATTING PURPOSES.

3. WRITE FROM(BUF) INDEX(5):

   Specifies that the record BUF be written as the fifth record of the disk or tape file SYSPRINT.

7.3.7 Storage Allocation Statements

7.3.7.1 The ALLOCATE Statement

The ALLOCATE statement causes storage to be allocated for based and/or controlled variables.

Syntax

```
<allocate-statement>::=
   ALLOCATE::=<allocation>[,<allocation>]...
   <allocation>::=<identifier>[IN(<scalar-area-variable>)]
                              [SET(<scalar-locator-variable>)]
```

Semantics

The <identifier> to be allocated must be the name of a level 1 scalar, array, or major structure variable with the BASED or CONTROLLED attribute.

The amount of storage that is to be allocated for the <identifier> is determined by evaluating all bounds of arrays, lengths of strings and sizes of areas. Although the limits and initial values of the variable are evaluated at the time of execution of the ALLOCATE statement, the names in these expressions are interpreted in the environment of the DECLARE statement. If these expressions contain references to the variable being allocated, they reference the newly established generation.

Semantics for Controlled Allocation:

The IN and SET options are not allowed.

When a CONTROLLED variable is allocated, all previous generations of the variable are pushed down or stacked and a new generation is established. When a FREE statement is executed for the variable, the most current generation is released and the next oldest generation becomes the current generation.

The limits associated with a CONTROLLED variable are evaluated at the time of allocation, and are permanently associated with the newly allocated generation.

Semantics for Based Allocation:

The allocation of a variable with the BASED storage class attribute has no effect on other generations of the same variable. Storage for BASED variables is not pushed down or stacked. A given generation of a based variable may be accessed by a suitable based reference regardless of allocations of the based variable performed after this generation is allocated. The allocation of a based variable proceeds as follows:

1. The bounds, string lengths, and area sizes of all the fields associated with the allocation are evaluated. Expressions preceding the keyword REFER are used as the values of the bounds, string lengths, or area sizes specified by the REFER options.

2. Sufficient storage for a generation of the based variable with these bounds, string lengths, and area sizes is then allocated. The AREA condition will be raised if the allocation is attempted in an area where the size of the area is insufficient to contain the new generation.

3. After the allocation has been generated, those variables, within the new generation, that are objects of REFER options are initialized to the values specified in the REFER options.

4. Initial values specified in the declaration of the based variable are assigned to the generation that has been allocated.

5. The locator variable specified in the SET option or, if the SET option is not used, the locator variable specified in the BASED attribute part of the declaration for the variable, is assigned a pointer value which identifies the generation that has been allocated.

The allocation of a based variable involves the based variable to be allocated, a locator variable to identify the new generation, and an area if the generation is to be allocated in an area. If no SET option is specified, a SET option is assumed to specify the locator declaration. It is an error, in such a case, if this BASED attribute does not specify a locator variable. If the SET option specifies an offset variable and no IN option is present, then an IN option is assumed to specify the area given in the IN attribute part of the declaration for the offset variable. In such a case it is an error if this offset attribute does not specify an area variable.

If the SET option specifies an offset variable, the pointer value identifying the new generation is assigned to the offset variable. The IN option must be present, or be assumed, and it must specify either the same area as that specified in the IN attribute of the declaration of the offset variable, or an area contained in or containing that area.

If no IN option is present in the ALLOCATE statement and none is assumed, the new generation is allocated in the storage associated with the task which executes the ALLOCATE statement. The SET option in this case must specify a pointer variable.

If an IN option is present, or is assumed, an attempt is made to allocate the new generation in the area specified by the IN option. If there is sufficient storage within the area specified by the IN option, the generation is allocated in the area and a pointer value identifying the generation is assigned to the locator variable specified in the SET option. If insufficient storage exists, the AREA condition is raised. In the case of normal return from an AREA on-unit the IN option is re-evaluated, and the allocation is attempted again.

A pointer value identifying an area is not the same as a pointer value identifying the first generation allocated within the area.

Examples

1.
```
DECLARE A(N,M) CONTROLLED;
    .
    .
    .
N,M = 100;
ALLOCATE A;
    .
    .
    .
N=50;
ALLOCATE A;
    .
    .
    .
FREE A;
```

This example above shows two generations of A. The first is a 100 by 100 array. The second is a 50 by 100 array. The FREE statement releases the most recently allocated generation (the 50 by 100 array), leaving the 100 by 100 array as the current generation.

2.
```
DECLARE A(N) CONTROLLED:
    .
    .
    .
N = 10;
ALLOCATE A;
    .
    .
    .
N = 20;
PUT LIST (HBOUND(A,1));
```

This example outputs the value 10 because the limits of a controlled variable are evaluated only at the time of allocation and are thereafter associated with the generation which they describe.

3.
```
DECLARE NAME CHARACTER (200) BASED;
    .
    .
    .
ALLOCATE NAME SET(P);
    .
    .
    .
ALLOCATE NAME SET(Q);
    .
    .
    .
P->NAME = 'ABC' ;
Q->NAME = 'XYZ' ;
```

In this example, two generations of NAME are created, each having a length of 200 characters. The pointer P identifies the first generation, and the pointer Q identifies the second.

If the length of NAME has been specified by the expressions N, the length of each generation could have been unique.

```
DECLARE NAME CHAR(N) BASED;
    .
    .
    .
N = 100;
ALLOCATE NAME SET(P);
    .
    .
    .
N = 200;
ALLOCATE NAME SET(Q);
```

However, because the limits of BASED variables are evaluated at each reference, the programmer must insure that N has the proper value when he references each generation of NAME.

```
N=100;
P->NAME = 'ABC' ;
N = 200;
Q->NAME = 'XYZ' ;
/*P-> NAME = Q-> NAME*/
```

The expression shown as a comment is illegal because N cannot be 100 and 200 at the same time.

To relieve the programmer from the burden of maintaining the proper limits when referencing BASED variables, the REFER option can be used. (See discussion of BASED attribute in Section 4.11, Storage Class.

```
DECLARE 1 S BASED,
        2 N,
        2 NAME CHAR(M REFER(S.N));
M = 3;
ALLOCATE S SET(P);
   .
   .
   .
M=4;
ALLOCATE S SET(Q)
   .
   .
   .
P->NAME = 'ABC' ;
Q->NAME = 'WXYZ' ;
PUT LIST(P-> NAME, LENGTH(P-> NAME),Q-> NAME,
LENGTH(Q-> NAME));
```

Each allocation causes the length expression M to be evaluated and its value used to create storage for the generation of S being allocated. The value of M is then assigned to the newly allocated generation of S.N. Subsequent references to NAME always use the generation of S.N identified by the pointer used to reference NAME.

```
Q->NAME   uses   Q->S.N.
P->NAME   uses   P->S.N.
```

The output of the last example is;

```
'ABC' 3 'WXYZ' 4
```

7.3.7.2 The FREE Statement

The FREE statement causes the storage allocated for specified based or controlled variables to be freed. For controlled variables, the next youngest generation is made available, and subsequent references to the name of the controlled variable will reference that generation.

Syntax

```
<free-statement> ::=
FREE (<controlled-variable> | ([<locator-qualifier>->]
      <based-variable>[IN(<scalar-area-variable>)]))
     [, <controlled-variable> | ([<locator-qualifier>->]
        <based-variable> [IN(<scalar-area-variable>)])]...
```

Restrictions:

The <controlled-variable> must be an unsubscripted, level 1 controlled variable.

The <based-variable> must be an unsubscripted, level 1 BASED variable, See Section 4.6.3.1, Locator Qualification.

Semantics

If a specified controlled identifier has no allocated storage at the time the FREE statement is executed, no attempt is made to free storage.

A BASED variable can be used to free storage only if that storage has been allocated by a based variable having identical data attributes, including values of bounds, lengths, and area sizes.

An IN option must be specified or implied if and only if the generation to be freed was allocated in an area. The IN option must specify the area in which the generation was allocated. The effect of the FREE statement is to make the relevant storage available for subsequent allocation by an ALLOCATE statement which names the same area in the IN option. If the reference to the variable to be freed is qualified by the POINTER built-in function (either explicitly, or implicitly by the appearance of an offset as the pointer qualifier), and the IN option is absent, the statement is executed as if it contains the IN option naming the area which is the second argument of the POINTER built-in function. Unless allocation has been in an area, the FREE statement cannot include an IN option nor can an IN option be implied by use of an OFFSET variable.

Examples

1.
```
DCL A AREA, O OFFSET (A), V BASED (O);
FREE V;
```

The FREE statement is equivalent to

```
FREE POINTER (O,A)->V IN (A);
```

2. The following example illustrates the FREE statement in conjunction with an ALLOCATE statement:

```
DECLARE A(100) INITIAL ((100)0)
    CONTROLLED, C(100),X(100);
                    .
                    .
                    .
ALLOCATE A;
                    .
                    .
                    .
C=A;
                    .
                    .
                    .
FREE A;
```

7.3.8 System Attribute Statements

7.3.8.1 The System File Attribute Assignment Statement

The <system-file-attribute-assignment-statement> is used to set the value of a system file attribute for a file outside of a file declaration or an OPEN statement.

Syntax

```
<system-file-attribute-assignment-statement> ::=
    <system-file-attribute> (<file-designator>) =
    <scalar-expression>;
```

Semantics

Attempting to set a read-only attribute will generate a syntax error of level three (3) at compile-time and a diagnostic error at run-time.

Multiple assignments are not allowed.

If a file attribute has mnemonics, they must be used by specifying a string constant or a string variable which has been assigned the correct value. If a constant is used, a compile-time check is made to see if it is a valid mnemonic for the specified file attribute. If not an error of level three is given. If a variable is used, it is checked at run-time, and if it is an invalid mnemonic, a non-fatal run-time error is given.

7.3.8.2 The System File Attribute Reference Statement

The <system-file-attribute-reference-statement> is used to reference the value of a system file attribute.

Syntax

```
<system-file-attribute-reference-statement> ::=
    {<scalar-variable> | <pseudo-variable>} =
        <attribute-name> (<internal-file-name>
        [,<scalar-expression> [,<scalar-expression>]]);
```

Semantics

The two <scalar-expression>s are interpreted as follows: if KIND = PORT or REMOTE, the first <scalar-expression> identifies the STATION number, and the second <scalar-expression> is not allowed. If KIND = DISK or PACK, one <scalar-expression> refers to row number; two <scalar-expression>s refer to row number followed by copy number.

An attempt to read a write-only attribute generates a syntax error of level three at compile-time and a diagnostic error at run-time.

7.3.8.3 The System Task Attribute Assignment Statement

The <system-task-attribute-assignment-statement> is used to set the value of a system attribute for a task.

Syntax

```
<system-task-attribute-assignment-statement> ::=
     <system-task-attribute> (<task-designator>) =
     <scalar-expression>;
<task-designator>::= <task-variable>
```

Semantics

Attempting to set a read-only attribute will generate a syntax error of level three (3) at compile-time and a diagnostic at run-time.

Multiple assignments are not allowed.

If a task attribute has mnemonics, they may be used, but mnemonics are not required.

7.3.8.4 The System Task Attribute Reference Statement

The <system-task-attribute-reference-statement> is used to reference the value of a system task attribute.

Syntax

```
<system-task-attribute-reference-statement> ::=
     {<scalar-variable> | <pseudo-variable>} =
          <system-task-attribute> (<task-designator>);
```

Semantics

Attempting to read a write-only attribute will generate a syntax error of level three (3) at compile time and a diagnostic error at run time.

7.3.8.5 The System Library Attribute Assignment Statement

The <system-library-attribute-assignment-statement> sets the value of library attributes outside of a library declaration.

Syntax

```
<system-library-attribute-assignment-statement>::=
<system-library-attribute> (<library-identifier>)=
<scalar-expression>;
```

Semantics

Attempting to set a read-only attribute generates a syntax error at compile time and a diagnostic error at run time.

Multiple assignments are not allowed.

7.3.8.6 The System Library Attribute Reference Statement

The <system-library-attribute-reference-statement> is used to reference the value of a system library attribute.

Syntax

```
<system-library-attribute-reference-statement>::=
 <scalar-variable>/<pseudo-variable> =
<system-library-attribute>(<library-identifier>);
```

Semantics

The values specified for the system library attributes are subject to some constraints:

1. The TITLE must be a valid file title as judged by the system; "*", usercode, and ON PACKNAME specifications are allowed.

2. The INTNAME must be a single identifier.

3. The LIBPARAMETER may be any string and is limited only by the conventions of the particular library.

Attempting to read a write-only attribute generates a syntax error at compile time and a diagnostic error at run time.

7.3.9 The Null-Statement

The <null-statement> is an empty statement. The <null-statement> causes no action and does not affect sequential operations in any way.

Syntax

```
<null-statement> ::= ;
```

Example

```
        .
        .
        .
ON OVERFLOW;
        .
        .
        .
```

The <on-unit> is a <null-statement>.

THIS PAGE IS INTENTIONALLY LEFT BLANK FOR FORMATTING PURPOSES

SECTION 8. INPUT/OUTPUT

A collection of data external to the program constitutes a file. Input activity transmits data from a file to a program. Output activity transmits data from a program to a file. Data transmission statements refer to a file name declared in the program and associated with a file.

In STREAM data transmission, the file can be considered to be a continuous stream of characters. The GET and PUT statements are used to transmit data values from and to the file. Conversions may occur during transmission. (See Section 8.4.2, Modes of Stream Transmission.)

In RECORD data transmission, the file consists of discrete records. The READ and WRITE statements cause a single record to be transmitted from or to the file. Transmission is direct, without any conversion, either directly to or from data variables or an intermediate buffer that may be addressable. When transmission is to or from data variables, the attributes of the variables should accurately describe the composition of the record accessed.

8.1 FILE ATTRIBUTES

The present section describes how attributes are collected and become associated with a file.

The file attributes can be divided into two categories: alternative attributes and additive attributes. Alternative attributes are those in which one group may be selected. If there is no explicit or implied declaration for one of the alternatives, and if one of those alternatives is required, a default attribute is selected. Additive attributes are those that are never applied by default and must always be stated explicitly either in a file declaration or in the OPEN statement. The exceptions are that KEYED is implied by DIRECT, and PRINT may be supplied for the SYSPRINT file. (See Section 8.4.1, Stream Transmission Statements.)

The use of PORT file attributes in PL/I is similar to that in other languages. PORT file attributes are documented in the B 7000/B 6000 Series I/O Subsystem Reference Manual.

Alternative attributes and their defaults:

ATTRIBUTES	DEFAULT
STREAM | RECORD	STREAM
INPUT | OUTPUT | UPDATE	INPUT
SEQUENTIAL | DIRECT	SEQUENTIAL
INTERNAL | EXTERNAL	EXTERNAL

Following is a list of the additive attributes:

PRINT
KEYED
PORT
{ENVIRONMENT | OPTIONS}(<file-attribute-list>)

8.1.1 Merging of Attributes

There may be a conflict between the attributes specified in a file declaration and the attributes merged as the result of the file opening, either explicit or implicit. For example, the attributes INPUT and UPDATE are in conflict, as are the attributes UPDATE and STREAM.

When a conflict occurs between merged attributes, the last specified attribute is given precedence.

After the attributes are merged, the attribute implications, listed below, are applied prior to the application of default attributes discussed earlier in this section. Implied attributes can also cause a conflict. If a conflict of attributes exists after the application of default attributes, the latest attributes have precedence.

Attributes and their implied attributes:

MERGED ATTRIBUTE	IMPLIED ATTRIBUTE(S)
UPDATE	RECORD
SEQUENTIAL	RECORD
DIRECT	RECORD,KEYED
PRINT	OUTPUT,STREAM
KEYED	RECORD

The following two examples illustrate attribute merging for an explicit opening and for an implicit opening:

Example Explicit Opening

```
DECLARE LISTING FILE STREAM;
    .
    .
    .
OPEN FILE (LISTING) PRINT;
```

The DECLARE statement establishes the file name LISTING as EXTERNAL by default, before execution commences.

Attributes after merge, due to execution of the OPEN statement, are EXTERNAL, STREAM and PRINT.

Attributes after implication are EXTERNAL, STREAM, PRINT, and OUTPUT.

This is a complete set of file attributes. No file attribute defaults are applied. The default attribute SEQUENTIAL does not apply as this attribute can be specified only for RECORD files.

Example Implicit Opening

```
DECLARE MASTER FILE KEYED INTERNAL;
.
.
.
READ FILE (MASTER) INTO (MASTERRECORD)
KEYTO (MASTERKEY);
```

Attributes after merge due to the opening caused by execution of the READ statement are KEYED, INTERNAL, RECORD, and INPUT.

Attributes after implication are KEYED, INTERNAL, RECORD and INPUT. There are no additional attributes implied.

Attributes after default application are KEYED, INTERNAL, RECORD, INPUT, and SEQUENTIAL.

8.1.2 Valid Combinations for File Attributes

Valid complete combinations of file attributes:

```
FILE STREAM INPUT
FILE STREAM OUTPUT
FILE STREAM OUTPUT PRINT
FILE RECORD INPUT SEQUENTIAL
FILE RECORD INPUT SEQUENTIAL KEYED
FILE RECORD OUTPUT SEQUENTIAL
FILE RECORD OUTPUT SEQUENTIAL KEYED
FILE RECORD UPDATE SEQUENTIAL
FILE RECORD UPDATE SEQUENTIAL KEYED
FILE RECORD INPUT SEQUENTIAL
FILE RECORD INPUT SEQUENTIAL KEYED
FILE RECORD OUTPUT SEQUENTIAL
FILE RECORD OUTPUT SEQUENTIAL KEYED
FILE RECORD UPDATE SEQUENTIAL
FILE RECORD UPDATE SEQUENTIAL KEYED
FILE RECORD INPUT DIRECT KEYED
FILE RECORD OUTPUT DIRECT KEYED
FILE RECORD UPDATE DIRECT KEYED
```

In addition, the SCOPE attributes and the ENVIRONMENT (or OPTIONS) may be specified with any valid combination.

8.2 Opening a File

The opening of a file is the means by which a file name is associated with that particular file. The identity of the file can be specified in the TITLE option of the OPEN statement. If not specified the file name will indicate the identity of the file. A part of the opening process is the completion of the set of attributes for the file.

Opening a file for STREAM INPUT, for SEQUENTIAL INPUT FORWARDS, or for SEQUENTIAL UPDATE causes the file to be positioned to the first record of that file.

A file can be opened either explicitly or implicitly as described in the following paragraphs.

8.2.1 Explicit Opening

A file is opened explicitly through execution of an OPEN statement that specifies the file name. The OPEN statement may list any of the attributes given above except the INTERNAL, or EXTERNAL attributes. Attributes listed in an OPEN statement are merged with and override any attributes listed in a file declaration for that file name. In an explicit opening, the OPEN statement must be executed prior to the execution of any statements listed in section 8.2.2, Implicit Opening, that refer to that file name.

8.2.2 Implicit Opening

An implicit opening of a file may occur if one of the statements listed below is executed prior to the execution of an OPEN statement specifying the same file name. The statement type is used to determine the usage and function attributes of the file if they have not been explicitly stated in a DECLARE statement. The effect of an implicit opening, caused by one of these statements, is as if the statement were preceded by an OPEN statement specifying the attributes implied from the statement type.

Following is a list of the statement identifiers and the attributes that will be implied from each and that will be applied in the absence of an explicit declaration to the contrary:

STATEMENT IDENTIFIER	ATTRIBUTES IMPLIED
GET	STREAM, INPUT
PUT (W/O DATA option)	STREAM, OUTPUT
READ	RECORD, INPUT
WRITE	RECORD, OUTPUT
LOCATE	RECORD, DIRECT SEQUENTIAL
REWRITE	RECORD, UPDATE
DELETE	RECORD, DIRECT UPDATE

8.3 Closing a File

Closing a file is a means by which a file name is dissociated from the file with which it was associated by the file opening. A file can be closed explicitly by execution of a CLOSE statement that specifies the file name, or implicitly on termination of the task

8.4 Stream Transmission

8.4.1 Statements

This section provides a summary of the allowed STREAM data transmission statements, along with their options, according to the file attributes.

Syntax

```
<stream-input-statement> ::=

     GET (<input-file-option> | <input-string-option>)
     [<data-specification>];

<input-file-option> ::=

     [FILE (<file name>)] [COPY]
     [SKIP [(<scalar expression>)]]

<input-string-option> ::=

     [STRING (<scalar-character-string-variable>)]

<stream-output-statement> ::=

     PUT (<output-file-option> | <output-string-option>)
     [<data-specification>];

<output-file-option> ::=

     [FILE (<file name>)]
     [SKIP [(<scalar-expression>)]]
     [PAGE [LINE (<expression>)]]
     [LINE (<scalar-expression>)]

<output-string-option> ::=

     [STRING (<scalar-character-string-variable>)]

<data-specification> ::=

     (LIST (<data-list>)) |
      (DATA [(<data-list>)]) |
      (EDIT (<data-list>) (<format-list>)
          [(<data-list>) (<format-list>)]...)
```

For a more detailed discussion of data lists and format lists see Section 8.4.3, Data Lists and Section 8.4.6.1, Format Lists.

Semantics

The <data-specification> can be omitted only if the SKIP option or one of the printing options appears.

The PAGE and LINE options may only be used with files that have the PRINT attribute.

A GET statement that does not specify a file or string option is equivalent to the GET statement:

```
GET FILE (SYSIN)...;
```

A PUT statement that does not specify a file or string option is equivalent to the PUT statement:

```
PUT FILE (SYSPRINT)...;
```

The contextual recognition of the FILE attribute applies to the identifiers SYSIN and SYSPRINT in these statements.

If the merged attributes of a file named SYSPRINT contain the attributes STREAM and OUTPUT and if SYSPRINT does not have the INTERNAL attribute, the PRINT attribute is supplied by default.

8.4.2 Modes of Stream Transmission

There are three modes of STREAM transmission: list-directed, data-directed, and edit-directed.

8.4.2.1 List-Directed Transmission

List-directed transmission permits the user to specify the storage area to which data is assigned or from which data is transmitted without specifying the format.

Input: The data in the stream is in the form of optionally signed constants. The program storage areas to which the data is to be assigned are specified by a data list.

Output: The data values to be transmitted are specified by a data list. The form of the data placed in the stream is a function of the data value and precision.

8.4.2.2 Data-Directed Transmission

Data-directed transmission permits the user to read or write self-identifying data.

Input: The data in the stream is in the form of optionally signed constants and includes information identifying the program storage areas to which the data is to be assigned.

Output: The data values to be transmitted are specified by a data list. The data placed in the stream has the form of constants and includes the name of the data being transmitted.

8.4.2.3 Edit-Directed Transmsissions

Edit-directed transmission permits the user to specify the storage area to which data is to be assigned or from which data is to be transmitted and the form of data fields in the stream.

Input: The form of the data in the stream is defined by a format list. The program storage areas to which the data is to be assigned are specified by a data list.

Output: The data values to be transmitted are defined by a data list. The form that the data is to have in the stream is defined by a format list.

8.4.3 Data Lists

List-directed and edit-directed data specifications require a data list to specify the data items to be transmitted. A data-directed data specification may or may not include a data list.

Syntax

<data-list> ::= <element>[,<element>] ...

Semantics

The nature of the elements depends upon whether the data list is used for input or for output.

On Input: Each data list element for edit-directed and list-directed data may be one of the following: a scalar name, an array name, a structure name, a pseudo-variable, or a repetitive specification involving any of these elements. For a data-directed data specification, each data list element may be an unsubscripted scalar, array or structure name.

On Output: Each data list element for edit-directed and list-directed data specifications may be one of the following: a scalar expression, an array expression, a structure expression, or a repetitive specification involving any of these elements. For a data-directed data specification, each data list element may be a scalar, array, or structure name, or a repetitive specification involving any of these elements.

The elements of a data list must be arithmetic or string data type.

A data list must be enclosed in its own set of delimiting parentheses.

8.4.3.1 Repetitive Specification

A repetitive specification may appear in a data list.

Syntax

```
<repetitive-specification> ::=

     (<element> [,<element>]...DO

     (<scalar-variable> | <scalar-pseudo-variable>)

     = <specification>[,<specification>]...)

<specification> ::=

     <expression1>

     [(TO <expression2>[BY<expression3>]) |

     (BY<expression3>[TO<expression2>])]

     [WHILE (<expression4>)]
```

Semantics

Each element in the element list of the repetitive specification is the same as those described for the data list elements above.

The expressions in the specifications are described as follows:

1. Each expression in the specification is a scalar expression.

2. In the specification, <expression1> represents the starting value of the control variable or pseudo-variable. <expression3> represents the increment to be added to the control variable after each repetition of data list elements in the repetitive specification of the control variable. <expression2> represents the terminating value. The exact meaning of the specification is identical to that of a DO statement with the same specification. When the last specification is completed, control passes to the next data list element.

Repetitive specifications may be nested to any depth. That is, each element in the data list may be a <repetitive-specification>. A repetitive specification involving M elements repeated N times is equivalent to M*N elements.

Examples

1.
```
GET LIST (((A(I,J) DO I = 1 TO 2) DO J = 3 TO 4));
```

This is equivalent to:

```
DO J = 3 TO 4;
      DO I = 1 TO 2;
      GET LIST (A(I,J));
      END;
END;
```

It gives values to the elements of the array A in the following order:

```
A(1,3), A(2,3), A(1,4), A(2,4)
```

2. PUT LIST ((A(J), (B(I,J) DO I = 1 TO 10)

```
DO J=5 TO 10));
```

This is equivalent to:

```
DO J=5 TO 10;
PUT LIST(A(J));
    DO I=1 TO 10;
    PUT LIST (B(I,J));
    END;
END;
```

8.4.3.2 Transmission of Data List Elements

If a data list element is an array name, the elements of the array are transmitted in row-major order, that is, by rows with the rightmost subscript of the array varying most frequently.

If a data list element is a structure name, the elements of the structure are transmitted in the order specified in the structure declaration.

Examples

1. If the structure declaration was:

```
DECLARE 1 A(10), 2 B, 2 C;
```

Then the statement

```
PUT FILE (X) LIST (A);
```

would result in the output being ordered as follows:

```
A.B(1) A.C(1) A.B(2) A.C(2) A.B(3)
A.C(3) ..., ETC.
```

However, if the declaration had been:

```
DECLARE 1 A, 2 B(10), 2 C(10);
```

Then the same PUT statement would produce:

```
A.B(1) A.B(2) A.B(3) ..., A.B(10)
A.C(1) A.C(2) A.C(3) ..., A.C(10).
```

If, within a data list used in an input statement, a variable is assigned a value, this new value is used in all later references in the data list and the format list, if present.

2. In the following statement, B is a structure, XSTRING is a character string, and C is an array:

```
DECLARE A FLOAT, 1, B, 2 P, 2 E, 3 F,
XSTRING CHARACTER (6), C(10) FIXED;
```

The following data list, involving these data items, and the scalar variable A, may be used for input or output:

```
(A,B, SUBSTR (XSTRING, 2),
   (C(I) DO I = 2 TO 7))
```

The data list elements are transmitted in the following order:

A - the scalar variable is transmitted.

P, E, F - the elements of structure B are transmitted.

SUBSTR (XSTRING, 2) - second through sixth characters of the XSTRING are transmitted.

C(2), C(3),..., C(7) - the six specified elements of the array C are transmitted.

8.4.4 List-Directed Data Specification

Syntax

```
<list-directed-data-specification> ::= LIST (<data-list>)
```

The <data-list> is described in Section 8.4.3, Data Lists.

8.4.4.1 List-Directed Input Format

When the data list element is an array name and the data consists of constants, the first constant is assigned to the first element of the array, the second constant to the second element, etc., in row-major order (i.e., with the right subscript varying most rapidly).

A structure name in the <data-list> represents a list of the contained scalar variables and arrays in the order specified in the structure description.

Data in the stream has one of the following general forms:

```
[+|-]<arithmetic-constant> |
<simple-character-string-constant>
```

Constants may be surrounded by blanks, which are not treated as part of the data. However, blanks cannot appear between the optional sign and the constant.

Data items in the stream must be separated either by a blank or by a comma. This separator may be preceded and/or followed by an arbitrary number of blanks. A null field in the stream is indicated either by the very first non-blank character in the stream being a comma, or by two adjacent commas separated by an arbitrary number of blanks. A null field specifies that the value of the associated item in the data list specification is to remain unchanged.

The transmission of the list of constants on input is terminated by expiration of the <data-list> or if the ENDFILE condition is raised. In the former case, positioning is always at the character following the first blank or comma following the last data item. More than one

blank can separate two data items, and a comma separator may be preceded or followed by one or more blanks. In such cases, a subsequent list or data-directed GET will ignore intervening blanks and the comma (if present), and will access the next data item. However, if an edit-directed GET should follow, the first character accessed will be the character to which the file has been positioned (i.e., the next data item will begin with the first character following the blank or comma that separated it from the previous data item).

If the data is a string constant, the surrounding quotation marks are deleted. If the data is a character string constant, it is interpreted as a character string. If the data is a bit string constant, it is interpreted as a bit string.

If the data is an arithmetic constant it is converted to coded arithmetic with the base, scale, mode, and precision implied by the constant.

The list item is then examined and the interpreted string value is assigned to it as shown in the chart below.

STREAM ITEM	DATA LIST	CONVERSION
Character string	Arithmetic	Character string to Arithmetic
	Character String	Character string assignment
	Bit String	Character string to Bit string
Bit String	Arithmetic	Bit string to Arithmetic
	Character String	Bit string to Character string
	Bit String	Bit String assignment
Arithmetic	Arithmetic	Arithmetic type conversion
	Character String	Arithmetic to Character string
	Bit string	Arithmetic to Bit string

The type conversions are described in Section 5.2.6, Type Conversion, except arithmetic to character conversion, which is described in Section 8.4.4.2, List Directed Output Format.

8.4.4.2 List-Directed Output Format

The values of the scalar variables in the <data-list> are converted to a character representation of the data value and transmitted to the data stream.

In general, a blank is used to separate data items transmitted. If a numeric data item is longer than the characters remaining on the current line, the entire item will be printed starting at the beginning of the next line. If the length of the item is greater than the size of the line, splitting of the item will occur.

The length of the data field placed in the file is a function of the internal precision and value of the data item.

8.4.4.2.1 Coded Arithmetic Data

The external form of coded arithmetic data in the data stream is an optionally signed decimal constant whose field width, W, is a function of the internal precision declared for the data item and the value of the data item. In the discussion below, the following symbols are used.

1. W represents the field width, which is defined as the length of the data field.

2. D represents the number of positions in the external data field to the right of the decimal point.

3. P represents the total number of digits in the data item after any necessary conversion to decimal.

4. Q represents the scale factor of the data item after any necessary conversion to decimal.

5. S represents a scaling factor as described for floating-point data.

6. YY represents a scaling factor for fixed-point data. The letter F actually appears in the output stream to indicate the presence of a scaling factor. Its value is similar to the value of E in a floating-point number.

7. X represents any decimal digit.

8. <b> represents a blank position in the output.

9. N represents the number of decimal digits in the exponent.

There are four kinds of coded arithmetic data to consider: coded real fixed-point decimal data, coded real fixed-point binary data, coded real floating-point binary data, coded real floating-point decimal data.

The discussions below apply to coded arithmetic data only when the value of the data item to be transmitted is greater than or less than zero. If the converted decimal value of a fixed-point item is equal to zero, the following rules apply:

1. If Q=0, the representation transmitted is a single zero preceded by P+2 blanks.

2. If P>=Q>0, the representation transmitted is a single zero preceded by P-Q+1 blanks, and followed by a decimal point and Q zeros.

3. If P<Q or Q<0, the representation transmitted is a single zero preceded by P blanks and followed by F (+|-) N digits.

4. If the converted decimal value of a floating-point item is equal to zero, the representation transmitted is a single zero, followed by a decimal point, P-1 zeros, the characters "E+", and N zeros.

Coded real fixed-point decimal data: a decimal fixed-point source with

precision (P,Q) is converted to character-string representation as follows:

1. If P>=Q>=0 (i.e., if the assumed decimal point lies within the field of the internal representation), then:

   A. The constant is right-adjusted in a field width of P+3.

   B. Leading zeros are replaced by blanks, except for a single zero that immediately precedes the decimal point of a fractional number.

   C. If the value is negative, a minus sign precedes the first significant digit (or the zero before the point of a fractional number). Positive numbers are unsigned.

   D. If Q>0, the constant has fractional Q digits. If Q=0 there is no decimal point.

2. If Q is negative or greater than P, a scaling factor is appended to the right of the constant. The constant itself is of the same form as an integer. The scaling factor has the form:

   F(+|-) YY where (+|-) YY has the value -Q.

The number of digits in the scaling factor is just sufficient to contain the value of Q without leading zeros.

The length of the intermediate string is:

P+3+K

where K is the number of digits necessary to represent the value of Q (not including a sign or the letter F).

Example

```
DECLARE A FIXED (4, -3), C CHAR (10);
                  A=1234.0E3;
                  C=A;
```

The intermediate string generated in converting A would be:

<b>1234F+3

which, when assigned to C, would give:

<b>1234F+3<b><b>

Coded real fixed-point binary data: the data item is converted to fixed-point decimal and is transmitted as coded real fixed-point decimal data.

Coded real floating-point decimal data: the data item is converted according to the rules for floating-point format items, E(W, D, S). For E-conversion, W = P + N + 4, D = P - 1 and S = P.

Coded real floating-point binary data: The data item is converted to floating-point decimal with a precision (P) and transmitted as coded real floating-point decimal data.

8.4.4.2.2 Numeric Character Data

The base of numeric character data may be decimal or binary.

Numeric Decimal Data: The external format and field width of the numeric decimal data item is that described by the associated picture specification.

Numeric Binary Data: The external format and field width of the numeric binary data item is that described by the associated picture specification. The binary digits 0 and 1 are represented by the characters 0 and 1.

8.4.4.2.3 Character String Data

The contents of the character string are written out. Enclosing quotation marks are supplied, and contained quotation marks are unmodified. The field width is the current length of the string.

8.4.4.2.4 Bit String Data

The format of the data on the external medium is that of a bit string constant. The value is enclosed in quotation marks and followed by the letter B. The binary bits are represented by the characters 0 and 1. The field width is P+3, where P is the current length of the string, and three additional positions are for the two quotation marks and the letter B.

Examples of List-Directed Transmission:

1. LIST (CARD, RATE, DYNAMIC_FLOW)
2. LIST ((THICKNESS (DISTANCE) DO DISTANCE = 1 TO 1000))
3. LIST (P, Z, M, R)
4. LIST (A*B/C, (X+Y)**2)

The specification in example 4 may only be used for output.

8.4.5 Data-Directed Data Specification

Syntax

<data-directed-data-specification> ::=

DATA [(<data-list>)]

Semantics

The <data-list> is described in Section 8.4.3, Data Lists. It cannot include parameters, or based or defined variables. Names of structure elements need only have enough qualification to resolve any ambiguity.

If no <data-list> appears on input, all of the data items known to the block containing the GET statement may be transmitted. The NAME condition will be raised if a name that is not known to the block is transmitted. If no <data-list> appears on output, all data items known to the block and allowed in data-directed transmission are to be transmitted.

Recognition of a semicolon in the stream of input causes transmission to cease. On output a semicolon is written into the stream after the last data item is transmitted.

8.4.5.1 Data-Directed Data in the Stream

The data in the stream associated with data-directed transmission is in the form of a list of scalar assignments.

Syntax

```
<data-directed-data-in-stream> ::=

    <scalar-variable>=<constant>
    [(<b>|,)<scalar-variable> = <constant>]...;
```

Semantics

The <scalar-variable> may be a subscripted name with integer constant subscripts.

On input, the scalar assignments may be separated by either a blank or a comma. On output, the assignments are separated by blanks.

The <constant> in the general format above has one of the forms described in Section 8.4.4.1, List-Directed Input Format.

8.4.5.1.1 Data Directed Input

If no <data-list> is specified, the names in the stream may be any fully qualified names known at the point of transmission.

If a <data-list> is specified, each element of the data list must be a scalar item, an unsubscripted array name, or structure name. The names in the stream must appear in the <data-list>. However, the order of the names need not be the same and the <data-list> may include names that do not appear in the stream. If a name appears in the stream but not in the <data-list>, the NAME condition will be raised.

Example

A, B, C, and D are scalar variables.

```
DATA (B, A, C, D)
```

This data list may be associated with the following input data stream:

```
A=2.5, B=.00476, D=125, Z= ABC ;
```

Z appears in the data list but not in the stream. Because Z does not appear in the data list, the NAME condition will be raised.

If the data list includes the name of an array, subscripted references to that array may appear in the stream. The entire array need not appear.

Example

X is the name of a two-dimensional array.

```
DECLARE X (2, 3);
```

The data list and input data stream could appear as follows:

```
Data List       Input Data Stream

     X          X(1,1) = 7.95, X(1,2)=8085,
                X(1,3) =73;
```

If the data list includes the names of structure elements, then fully qualified names of the items must appear in the stream.

Example Consider the following structure:

```
DECLARE 1 CARDIN, 2 PARTNO, 2 DESCRP;
```

To read a value for PARTNO, the data list may have the following forms, but the input data stream must have the name indicated:

```
        Data List              Input Data Stream

CARDIN.PARTNO          CARDIN.PARTNO=737314
  OR PARTNO
```

Interleaved subscripts cannot appear in qualified names in the stream. All subscripts must be moved all the way to the right, following the last name of the qualified name.

Example

```
DECLARE 1 Y(5,5), 2 A(10), 3 B, 3 C, 3 D;
```

An element name would have to appear in the stream as follows:

```
Y.A.B (2,3,8)=8.72;
```

The name of the data list must not contain the subscript.

8.4.5.1.2 Data Directed Output

An element of the <data-list>, which can be subscripted may be a scalar variable, an array variable, a structure variable, a repetitive specification involving any of these elements or further repetitive specifications. The data with names appearing in the <data-list> is transmitted in the form of a list of scalar assignments separated by blanks (one of which is transmitted immediately following each item) and terminated by a semicolon. Line splitting for PRINT file data items follow the rules set for list-directed transmission. If an item, together with its name is longer than one line, then splitting, if possible, is done after the = sign if that would make each piece fit on one line. See Section 8.4.5.2, Length of Data Fields in Data-Directed Output.

Array variables in the data list are treated as a list of the contained subscripted elements in row-major order.

Example

Let X be an array declared as follows:

```
DECLARE X (2,4) FIXED;
```

Let X appear in a data list as follows:

```
DATA (X)
```

Then, on output, the data stream is as follows:

```
X(1,1)=1 X(1,2)=2 X(1,3)=3 X(1,4)=4

X(2,1)=5 X(2,2)=6 X(2,3)=7 X(2,4)=8;
```

Items that are part of a structure appearing in the <data-list> are transmitted with the full qualification with subscripts following the qualified names rather than being interleaved.

Example

If a data list is specified for a structure element transmitted under data-directed output as follows:

```
DATA (Y(1,3).Q)
```

Then the associated data field in the output stream is as follows:

```
Y.Q(1,3) = 3.756;
```

Structure names in the data list are interpreted as a list of the contained scalar or array elements.

Example

Consider the following structure:

```
1 A, 2 B, 2 C, 3 D
```

If a data list for data-directed output is as follows:

```
DATA (A)
```

Then, if the value of B and D were 2 and 17 respectively, the associated data fields in the output stream would be as follows:

```
A.B= 2 A.C.D= 17;
```

8.4.5.2 Length of Data Fields in Data-Directed Output

For the purpose of defining the length of data fields transmitted to the data stream it is convenient to regard data fields as comprising a left-hand component and a right-hand component, joined by the assignment symbol (=).

The length of the left-hand component is the length of the data name (fully qualified) plus the length of any associated subscript list and enclosing parentheses. Subscripts are integer constants, but plus signs are not transmitted. The left-hand component contains no blanks.

The length of the right-hand component is precisely the same as would be obtained with list-directed output. See Section 8.4.4.2, List-Directed Output Format.

Example of Data-Directed Transmission:

```
AB:  PROCEDURE;
DECLARE A(6), B(7);
GET FILE (X) DATA (B);
     Input Stream
     B(1)=1, B(2)=2, B(3)=3,
     B(4)=1, B(5)=2, B(6)=3, B(7)=4;
DO I = 1 TO 6
A (I) = B (I+1) + B (I);
END;
PUT FILE (Y) DATA (A);
     Output Stream
     A(1)= 3 A(2)= 5 A(3)= 4 A(4)= 3
     A(5)= 5 A(6)= 7;
```

```
END AB;
```

8.4.6 Edit-Directed Data Specification

Syntax

<edit-directed-data-specification> ::=

EDIT [(<data-list>) (<format-list>)]...

Semantics

The <data-list> semantics are given in Section 8.4.3, Data Lists and the format list semantics in Section 8.4.6.1, Format Lists.

On output, the value of each data item in the <data-list> is converted to a format specified by the associated format item in the <format-list>. The first scalar data item is associated with the first format item. If the format item is a control format item, the control item is executed, and the data item associated with the first name in the data list is then associated with the next format item. The second scalar data item is then associated with the second data-format-item, etc.

For input, data in the stream is considered to be a continuous string of characters not separated into individual data items. The number of characters for each data item is specified by a format item in the <format-list>. The characters are treated according to the associated format item.

Suppose the <format-list> specifies J data format items, and the <data-list> specifies K data items. If J<K, then after J scalar data items have been transmitted, the format list is reused, the (J+1)th scalar item being associated with the first format item, etc. This reuse is performed as many times as required. If J>K, the excessive format items are ignored.

An array or a structure in a <data-list> is equivalent to N data items, where N is the number of scalar elements in the array or structure.

The specified transmission is complete when the last data item has been processed using its corresponding format item. Subsequent format items, including control format items, are ignored.

Examples

1. Edit-directed input specification.

```
EDIT (NAME, DATE, SALARY) (A(COLA-COLB), X(2), A(6),
F(M+2,2))
```

2. Edit-directed output specification.

```
EDIT ('INVENTORY -'|| INUM, INVCODE) (A, F(5))
```

8.4.6.1 Format Lists

The edit-directed data specification requires an associated format list.

Syntax

<format-list> ::=

<format-list-item> [,<format-list-item>]...

<format-list-item> ::=

<format-item> |
(<iteration-factor> <format-item>) |
(<iteration-factor>(<format-list>))

<iteration factor> ::=

(<scalar-expression>) | <integer-constant>

Semantics

<format-item> is described below.

The <iteration-factor> specifies the number of times the associated <format-item> is to be used. A zero or negative iteration factor specifies that the associated <format-item> is to be skipped and not used. The data list item will be associated with the next <format-item>. If an expression is used to represent the <iteration-factor>, it is evaluated and converted to an integer each time the format list item is used. The associated format item is that item or list of items to the right of the iteration factor.

There are three types of format items: data format items, control format items and remote format items. Data format items specify the form of the data fields in the stream. Control format items specify page, line, and spacing operations. Remote format items specify the use of format items remote from the GET or PUT statements.

8.4.6.2 Data Format Items

Data format items describe data representation in the data stream.

The discussion of format items requires the following definitions:

1. W represents the length of the data field, in characters, used by the data stream (including signs, decimal points, blanks, and the letter E as used in the representation of constants).

2. D represents the number of positions after the decimal point.

3. S represents the number of significant digits to appear.

4. P represents a scaling factor, which may be positive or negative.

Any of the quantities W, D, S, and P may be specified by a scalar

expression. When the format item is used, the expression is evaluated and converted to an integer. If W < 0 in a format specification, then the associated data and format list items are skipped. On input, if W = 0 and the data item is a string, the data value is taken as the null string. All expressions in a skipped data item are evaluated, and enabled conditions are raised as if the data item was not being skipped. The quantity D must be <= S and S<=W.

On input, the data item in the data stream is treated as if it conformed to the characteristics described by the format item.

There are five format items associated with data:

Fixed-point (F)
Floating-point (E)
Picture Specification (P)
Bit String (B)
Character String (A)

These items are described in the following paragraphs:

8.4.6.2.1 Fixed-Point Format Items

Decimal numeric data may be described by a fixed-point format item.

Syntax

<fixed-point-format-item> ::= F (W[,D[,P]])

In the following, use of the options is referred to as shown below:

F (W): Option 1

F (W,D): Option 2

F (W,D,P):Option 3

Input Semantics

The data item in the data stream is the character representation of a decimal fixed-point number anywhere in a field of width W. Leading and trailing blanks are ignored, but if the data consists only of blanks, it is interpreted as zero.

In option 2, if no decimal point appears in the number, it is assumed to appear immediately before the last D digits (trailing blanks are ignored). If a decimal point does appear, it overrides the D specification. Option 1 is option 2, with D equal to zero.

In option 3, the scaling factor effectively multiplies the external data value by 10 raised to the value of P. If P is positive, the number is treated as though the decimal point appeared P places to the right of its given position. If P is negative, the data is treated as though the decimal point appeared P places to the left of its given position. The given position of the decimal point is that indicated either by an actual point, if it is given, or by D, in the absence of an actual point.

Output Semantics

The external data is a decimal fixed-point number, right-adjusted in a field of width W. If the right-adjustment results in low-order digits being removed, the remaining lowest order digit is rounded.

In option 1, only the integer portion of the number is written. No decimal point appears.

In option 2, both the integer and fractional parts of the number are written. If D is greater than 0, a decimal point is inserted before the last D digits, and the value is appropriately positioned. Trailing zeros are supplied if the number of fractional digits is less than D (where D must be less than W). If the absolute value is less than 1, a zero precedes the point. If W is not large enough to include the zero, the SIZE condition will be raised.

In option 3, the scaling factor effectively multiplies the internal data value by ten raised to the power of P, before it is edited into its external character representation. If D is zero, only the integer portion of the number is considered.

For all options, if the value of the data item is less than zero, a minus sign will be prefixed to the external character representation. If it is greater than or equal to zero, no sign will appear. Therefore, for negative values, W must encompass both sign and decimal point. If the length of the data item is greater than W, the SIZE condition is raised.

8.4.6.2.2 Floating-Point Format Items

Decimal numeric data may be described by a floating-point format item.

Syntax

<floating-point-format-item> ::= E (W, D [,S])

Input Semantics

The data item in the external data field is an optionally signed character representation of a decimal floating-point number anywhere within a field width of W. An all-blank field is not treated as zero and causes the CONVERSION condition to be raised. The mantissa is a fixed decimal constant.

The external form of the number is as follows:

```
[ +|- ] <mantissa>
[   ( [E] (+|-) ) |
    (   E   [+|-] ) <exponent>]
```

If there is no decimal point in the data field, the decimal point is assumed to be before the last D digits of the mantissa. If there is a decimal point in the data field, it overrides the decimal point placement specified by D. Trailing blanks in the data field are ignored.

The <exponent> is an integer constant. If the exponent and the preceding E or sign are omitted, a zero exponent is assumed.
Output Semantics

The data item in the data field has the following general form:

If D < S:

[-] <S-D digits> . <D digits> E{+|-} <exponent>

If D = S:

[-] 0. <D digits>{E +|-} <exponent>

At least one nonfractional digit always appears. The <exponent> is an integer of five digits. When D < S, the exponent is adjusted so that the leading digit of the mantissa is nonzero. When D = S, the exponent is adjusted so that one zero digit appears before the point of the mantissa. In the case of the value zero, one zero digit appears before the point and D zero digits after the point. The exponent is zero and its associated sign is +.

If the above form does not fill the field of width W, it is right-adjusted and the blanks are inserted on the left. If the field width W is such that low-order digits are removed, the remaining lowest-order digit is rounded as if it was followed by a digit greater than or equal to 5.

If S is omitted, it is taken as equal to D + 1. When D < S, the field width W must be greater than or equal to S + 8 for non-negative values, and to S + 9 for negative values, of the data item. However, if D is zero, the decimal point is not transmitted, and W is equal to S + 7. If the length of the data item is greater than W, the SIZE condition results.

8.4.6.2.3 Numeric Picture Format Item

Numeric data may be described by a numeric picture format item.

Syntax

<numeric-picture-format-item> ::=

P'<numeric-picture-specification>'

Semantics

The <numeric-picture-specification> is described in Section 4.3.2, Numeric Picture Data Description.

On input, the form of the data in the data stream must exactly match the picture specification.

On output, the value of the list item is converted to coded arithmetic type and the converted value is edited to the form specified by the picture specification before it is transmitted.

8.4.6.2.4 Bit String Format Items

The bit string format item describes the representation of a bit string in the data stream.

Syntax

```
<bit-string-format-item> ::=

    B [(W)]
```

Semantics

In the case of input, W is always required. For output, if W is omitted, it is taken to be the current length of the associated bit string or the length obtained by converting the item to a bit string.

On input, the data field transmitted is a character representation of a bit string anywhere within the field of width W.

On output, the value of the list item is converted to bit and the resulting bit string is left-adjusted in the field of width W. Truncation, if necessary, is performed on the right. Zeros are used for padding.

8.4.6.2.5 Character String Format Items

Character data may be described by a character string format item.

Syntax

```
<character-string-format-item> ::=

    { A [(W)]} |
    { P'<character-picture-specification>'}
```

Semantics

The <character-picture-specification> is described in Section 4.3.1, Character Picture Data Description.

The representation in the data stream is a string of W characters.

On input, truncation, if necessary, is performed on the right. If the associated list element is too short, it is extended on the right with blanks. If the P format is used, W is implied and checking is performed to ensure that each character matches the corresponding character of the picture specification. On input, W is always required.

On output, W can be omitted, in which case W is taken to be the current length of the string (or the length of the converted character string.)

8.4.6.3 Control Format Items

There are three types of control format items, the spacing format item X, the positioning format items SKIP and COLUMN, and the printing format items PAGE and LINE.

8.4.6.3.1 Spacing Format Item

The spacing format item specifies relative horizontal spacing.

Syntax

<spacing-format-item> ::= X (W)

Semantics

On input, the format item specifies that the next W characters of the stream are to be ignored.

On output, the format item specifies that W blank characters are to be inserted into the stream.

If W is less than zero, it is taken as zero.

8.4.6.3.2 Positioning Format Items

The positioning format items specify positioning to a new line or to a specified column in the current (or next) line. (The length of a line is derived from the linesize of the file.)

Syntax

```
<positioning-format-item> ::=

     (SKIP [ (W) ]) |
     (COLUMN (W)  )
```

Semantics

The SKIP format item operates in the same manner as the SKIP option of a GET or PUT statement.

The COLUMN format item specifies that the file is to be positioned to the W-th column of the current line. If the file is an output file, the blank characters are inserted into the stream until the W-th column of the line is reached. If the file is an input file, the characters are ignored until the W-th column of the line is reached. If the file is already positioned beyond the W-th column of the current line, the file is positioned to the W-th column of the next line. If W is less than 1 or greater than the linesize of the file, W is assumed to be 1.

8.4.6.3.3 Printing Format Items

A printing format item specifies that the next data item transmitted is to appear on a new page or on a particular line of a page.

Syntax

```
<printing-format-item> ::=

     PAGE |
     (LINE (W))
```

Semantics

Printing format items may only be used with the STREAM PRINT files.

The PAGE and LINE format items operate in the same manner as the corresponding options with the PUT statement.

It should be noted that X and COLUMN specify, respectively, relative horizontal spacing. Similarly, SKIP and LINE specify relative vertical positioning. The first line on any page is line number one.

8.4.6.3.4 Remote Format Item

If it is desired to locate format items remotely from a format list, the remote format item, R, may be used.

Syntax

<remote-format-item> ::=

R (<statement-label-designator>)

Semantics

The <statement-label-designator> is a format label constant or a format variable. If a format label constant is designated, it must be the label of a FORMAT statement. The FORMAT statement includes a format list that is taken to replace the format item.

The designated FORMAT statement may be global to the statement in which the remote format item appears.

There can be no recursion. That is, a remote FORMAT statement may not contain an R format item which names itself as a statement label designator, nor may it name another remote FORMAT statement that will lead to the naming of the original FORMAT statement through a statement label designator.

Any conditions enabled for the GET or PUT statement must be correspondingly enabled for the remote FORMAT statements utilized.

8.5 RECORD TRANSMISSION

Files that contain discrete records or which are to be created as a collection of discrete records may be manipulated with record operation statements. The record operation statements are READ, WRITE, REWRITE, LOCATE and DELETE. A general description of these statements is contained in this section, and they are described completely in section 7. The records obtained from files or dispatched to files are defined in terms of the data attributes of a variable. For input operations the record is obtained from the file and placed intact into the variable. For output operations, the variable is transmitted intact into the file.

The variables involved in record transmission must be unsubscripted, of level 1 (scalar variables and array variables are of level 1 by default), and of any storage class. The variables cannot be parameters or defined variables. They may contain varying length strings. They may contain label, event, task, and pointer variables, but such data may lose its validity in transmission. Offset variables, however will maintain their validity.

With record transmission, it is possible to operate upon the record in a buffer. Operations within the buffer are accomplished through the use of a based variable, which describes the data attributes of the record, and a pointer variable, which identifies the location of the record within the buffer. An offset variable cannot be used, since an offset value is relative only to its associated area variable.

For input/output operations specifying based variables, the pointer value is set by the SET option in the READ or LOCATE statements.

8.5.1 Record Transmission Operations

A SEQUENTIAL file specifies that the accessing, creation, or modification of the file records is performed in a particular order, that is, from the first record of the file to the last record of the file.

A DIRECT file specifies that the accessing, creation, or modification of the file records is performed by indicating which particular record of the file is to be operated upon.

A file that is accessed, created, or modified in the sequential access method may or may not have the attribute KEYED. If a file has been created with the KEYED attribute, any recorded keys actually present in the file may be ignored while accessing sequentially, or they may be extracted from the file by use of the KEYTO option. It is possible to create a KEYED file as a SEQUENTIAL OUTPUT file and later to access that file as a DIRECT file.

SEQUENTIAL INPUT and SEQUENTIAL UPDATE files may be positioned to a particular record within the file by a READ operation that specifies the key of the desired file. Thereafter, successive READ statements without the KEY option will access the records sequentially. This kind of accessing may be used only if the file contains keyed records and if the file has the KEYED attribute.

Existing records of a file in a SEQUENTIAL UPDATE file can be rewritten (REWRITE statement), ignored (READ statement with an IGNORE option), or deleted (DELETE statement), but the number of records cannot be increased. For a SEQUENTIAL UPDATE file, only the last record that was read can be deleted. For DIRECT UPDATE files, the record to be deleted by the DELETE statement must be explicitly identified. In addition, records can be added to a DIRECT UPDATE file as well as rewritten by using WRITE and REWRITE statements, respectively.

If the READ INTO option is used in referring to a SEQUENTIAL UPDATE file and the next REWRITE statement does not specify a FROM option, the record in the file is replaced from the buffer and not from the variable that had been specified in the INTO option of the READ statement. The FROM option in a REWRITE statement must specifically name the variable into which the data has been read if that data is to be rewritten.

A WRITE statement adds records to a file, while a REWRITE statement replaces records. Thus, a WRITE statement may be used with OUTPUT or UPDATE files, while a REWRITE statement may only be used with UPDATE files. Moreover, a WRITE statement may use the KEYFROM option to indicate the actual transference of a key from internal storage to the file. The REWRITE statement uses the KEY option to identify the existing record to be replaced.

A READ or WRITE statement may access a specific record of a disk or tape file via the INDEX (<scalar-expression>) option. The expression is converted to an integer value which is used as the number of the file record to be accessed.

Carriage control characters in the first column of a record written to a printer file may be employed in the manner described in Section 7.3.6.11, The WRITE Statement.

SECTION 9. PROCEDURES

9.1 PARAMETERS

The PROCEDURE statement heading a given procedure and defining the primary entry point to the procedure may specify a list of parameters.

One or more ENTRY statements may also be used in the procedure to define secondary entry points. Like the heading statement of the procedure, each of the ENTRY statements must have at least one label to serve as an entry name for that point, and each may specify a list of parameters. Parameter lists for different entry points to a procedure need not be the same.

A parameter may be a scalar, array, or structure name that is unqualified and unsubscripted, or it may be a file name or an entry name. Parameters must be level 1 identifiers (i.e., they cannot be members of structures).

A file parameter may be used within a procedure wherever a file name may be used. An entry parameter may be used wherever an entry name may be used.

A reference within a procedure to a parameter produces an undefined result if the entry point at which the procedure is invoked does not include that parameter in its parameter list.

Parameters are explicitly declared by their appearance in a PROCEDURE statement or ENTRY statement, but attributes can be supplied in a DECLARE statement internal to the procedure. If attributes are not supplied in a DECLARE statement, default arithmetic attributes are applied.

Parameters cannot be declared with the storage class attributes or with the BUILTIN, INITIAL or DEFINED attributes, but a parameter may be used as a base identifier in a DEFINED attribute for simple and iSUB defining.

Scope attributes cannot be declared for parameters. A parameter has internal scope. Any bounds, lengths, and area sizes must be specified either by * or integer constants which, for bounds, may be signed.

Example

```
SBPRIM:  PROCEDURE (X, Y, Z);
         DECLARE (X, Y, A, B) FIXED, Z FLOAT;
         A = X-1; B = Y+1;
         GO TO COMMON;
SBSEC:   ENTRY (X, Z);
         A = X-2; B = X-3;
         COMMON: Z = A**2+A*B+B**2;
         END SBPRIM;
```

In this example, the procedure may be entered at its primary entry point SBPRIM, where the parameter list is (X, Y, Z), or its secondary entry point SBSEC, where the parameter list is (X, Z).

9.2 REFERENCES

9.2.1 Procedure References

Syntax

```
<procedure-reference> ::=

     <function-reference>|

     <subroutine-reference>

<function-reference> ::=

     <entry-name> ([<argument> [,<argument>]...])

<subroutine-reference> ::=

     <call-statement>
```

Semantics

At any point in a program where an entry name for a given procedure is known, the procedure may be invoked by a procedure reference.

The number of arguments (possibly zero) in the procedure reference must be equal to the number of parameters in the list for the entry point denoted by the entry name.

The procedure invoked by the procedure reference may be an external or an internal procedure. If it is an internal procedure, the block to which the entry name is internal must be active at the time of invocation of the procedure.

When a procedure reference invokes a procedure, each argument specified in the reference is associated with its corresponding parameter in the list for the denoted entry point, and control is passed to the procedure at the entry point.

There are two distinctly different uses for procedures, determined by one of two contexts in which a procedure reference may appear:

1. A procedure reference may appear as an operand in an expression. In this case, the reference is said to be a function reference, and the procedure is invoked as a function procedure, or simply a function.

2. A procedure reference may appear following the keyword CALL. In this case, the reference is said to be a subroutine reference, and the procedure is invoked as a subroutine procedure, or simply a subroutine.

9.2.1.1 Function References

When a function reference appears in an expression, the procedure is invoked. The procedure is then executed, using the arguments, if any, specified in the function reference. The result of this execution is the required value, which is passed with return of control back to the point of invocation. This returned value is then used to evaluate the expression.

A procedure that has no parameters may only be invoked in an expression if it is followed by an empty parameter list. Without this empty parameter list, the function identifier is treated as an entry-label constant (or entry-label variable), not as a function invocation. The procedure invoked by a function reference normally will terminate execution with a statement of the form RETURN (<expression>), where <expression> is a scalar expression of arithmetic, character-string, bit-string, locator or area type. It is the value of this expression that will be returned as the function value. The PROCEDURE statement or ENTRY statement at the invoked entry point may specify data attributes for the function value. If no attributes are specified default attributes are applied to the function value. Just prior to return, the <expression> is evaluated, and, before being passed back, the function value is converted, if necessary, to conform to these attributes.

9.2.1.2 Subroutine References

When a procedure is invoked by the execution of a CALL statement the initial action is the same as if the procedure were invoked as a function. The arguments in the procedure reference, if any, are associated with the parameters, and control is passed to the procedure at the denoted entry point. No value is returned by a procedure invoked in this way. Note that for procedures without parameters, the empty parentheses are not required when the procedure is invoked by a CALL statement or CALL option.

9.3 Procedure Reference Arguments

When a procedure is invoked, a relationship is established between the arguments of the invoking statement and the parameters of the invoked entry point. A parameter itself may be passed as an argument.

The correspondence of parameters in a parameter list with arguments in an argument list is from left to right, the leftmost parameter corresponding with the leftmost argument. The number of arguments and parameters must be the same.

In general, the arguments in a procedure reference may be any of the following:

1. Constants
2. Variables
3. Expressions

4. Computational built-in function names.

The attributes of each argument in a procedure reference must, in general, match the attributes of the corresponding parameter at the named entry point.

Example

Assume that the procedure SUB in a program is defined by:

```
SUB:   PROCEDURE (X, Y, Z);
       DECLARE X FIXED, Y ENTRY, Z LABEL;
          .
          .
          .
       END SUB;
```

This implies that the parameter X is used as a fixed-point variable with certain default data attributes. Y is used as an entry name, and Z is a statement label variable in the body of the procedure. Then if SUB is invoked in the program by the statement:

```
CALL SUB (R*S, CALC, L5);
```

It is necessary that:

1. R * S must yield a fixed-point value.

2. CALC be an entry name.

3. L5 be a statement-label designator.

9.3.1 Evaluation of Argument Subscripts

When an argument is a subscripted variable, the subscripts are evaluated before invocation. The specified element is then passed as a dummy argument.

9.3.2 Use of Dummy Arguments

A constructed dummy argument containing the argument value is passed to a procedure if the argument is one of the following:

1. An arithmetic, string, or label constant.

2. An expression involving operators.

3. An expression in parentheses.

4. An expression whose data attributes disagree with the data attributes declared for the parameter in an ENTRY attribute specification in the invoking block.

5. A function reference with arguments.

If the dummy is created, changes to the parameter are not reflected back in the original argument.

In all other cases the argument is passed as it appears. The parameter becomes identical with the passed argument, and changes to the

parameter are reflected in the original argument.

9.3.3 Entry Names as Arguments

When an entry name is specified as an argument of a procedure, one of the following applies:

1. If the entry name argument, call it M, is specified with an argument list of its own, it is recognized as a function reference, M is invoked, and the value returned by M effectively replaces M and its argument list in the containing argument list.

2. If the entry name argument appears without an argument list, but within an operational expression, then it is taken to be a function reference with no arguments.

3. If the entry name argument appears without an argument list and not within an operational expression, the entry name itself is passed to the function or subroutine being invoked. In such cases, the entry name is not taken to be a function reference, even if it is the name of a function that does not require arguments.

4. There is an exception to this rule. If an identifier is known as an entry name and appears as an argument, and if the parameter attribute list for that argument specifies an attribute other than an entry name, the entry name will be invoked and its return value passed.

Examples

```
1.CALL A(B);
```

This passes the entry name B as an argument to procedure A.

```
2.A: PROCEDURE;

        DECLARE B ENTRY,
             C ENTRY (FLOAT);
           .
           .
           .
        X = C(B);
           .
           .
           .
        END A;
```

In this case B is invoked and its returned value is passed to C. Consider the following example:

```
3.CALLP: PROCEDURE;

      DECLARE RREAD ENTRY,
              SUBR ENTRY (ENTRY, FLOAT, FIXED BINARY,
                LABEL);
      DECLARE (R,S) FLOAT;
         .
         .
         .
      GET LIST (R,S);
         .
         .
         .
      CALL SUBR (RREAD, SQRT(R), S, LAB1);
         .
         .
         .

LAB1: CALL ERRT(S);
         .
         .
         .
     END CALLP;
SUBR: PROCEDURE(NAME, X, J, TRANPT);
     DECLARE NAME ENTRY, TRANPT LABEL;
     DECLARE X FLOAT;
         .
         .
         .
      IF X > J THEN CALL NAME(J); ELSE GO TO TRANPT;
         .
         .
         .
      END SUBR;
```

In this example assume that CALLP, SUBR, and RREAD are external entry names. In CALLP, both RREAD and SUBR are explicitly declared to have the ENTRY attribute. (Actually, the explicit declaration for SUBR is used principally to provide information about the characteristics of the parameters of SUBR.) Four arguments are specified in the CALL SUBR statement. These arguments are interpreted as follows:

1. The first argument, RREAD, is recognized as an entry name (because of the ENTRY attribute declaration). The argument is not in conflict with the first parameter as specified in the parameter attribute list in the ENTRY attribute declaration for SUBR in CALLP. Therefore, since RREAD is recognized as an entry name not as a function reference, the entry name is passed at invocation.

2. The second argument, SQRT(R), is recognized as a function reference because of the built-in function name. SQRT is invoked, and the value returned by SQRT is assigned to a dummy argument, which effectively replaces the reference to SQRT. When SUBR is invoked, the dummy argument is passed to it.

3. The third argument, S, is simply a decimal floating-point element variable. However, since its attributes do not agree with those of the third parameter, as is specified in the parameter attribute list declaration, a dummy argument is

created containing the value of S converted to the attributes of the third parameter. When SUBR is invoked, the dummy argument is passed.

4. The fourth argument, LAB1 is a statement label constant. Its attributes agree with those of the fourth parameter. But since it is a constant, a dummy argument is created for it. When SUBR is invoked, the dummy argument is passed.

In SUBR, four parameters are explicitly declared in the PROCEDURE statement. If no further explicit declarations were given for these parameters, default attributes would be supplied for each. Therefore, since the name must represent an entry name, it is explicitly declared with the ENTRY attribute, and since TRANPT must represent a statement label, it is explicitly declared with the LABEL attribute, X is explicitly declared with the FLOAT attribute, and the defaults are allowed to apply to J.

Note that the appearance of NAME in the CALL statement does not constitute a contextual declaration of NAME as an entry name. Such a contextual declaration can be made only if no ENTRY or PROCEDURE statement of SUBR constitutes an explicit declaration of NAME as a parameter. If the attributes of a parameter are not explicitly declared in a complementary DECLARE statement, arithmetic defaults apply. Consequently, NAME must be explicitly declared to have the ENTRY attribute. Otherwise, it would be assumed to be a binary fixed-point variable, and its use in the CALL statement would result in an error.

9.4 USE OF THE ENTRY ATTRIBUTE

If an ENTRY attribute without parameter attribute lists is specified for an identifier, it indicates only that the identifier is an entry name. In this case, the argument must be an entry name. A contextual declaration of an identifier as an entry name supplies an ENTRY attribute specification of this type.

If an ENTRY attribute specification with the parameter attribute lists is supplied for the invoked entry name, each argument is converted, if necessary, to conform to the attributes specified for its corresponding parameter in the ENTRY attribute specification. String lengths and area sizes are considered to match in two circumstances only: if the length or area size is specified by an asterisk in the ENTRY attribute or if declarations for both the argument and the parameter contain the same decimal integer constant.

Dummy arguments are allocated immediately before invocation of the procedure and freed upon return.

The asterisk notation may be used in the ENTRY attribute to specify that for strings, areas, or arrays, the argument length, size, or bounds is assumed for the parameter.

Example

```
A:  PROCEDURE;
    DCL (C,D) FLOAT;
          .
          .
          .
    CALL B (C,D);
          .
          .
          .
    END A;

B:  PROCEDURE (P,Q);
    DECLARE P FIXED, Q FLOAT;
          .
          .
          .
    END B;
```

The specification of the ENTRY attribute in procedure B indicates that B has two parameters, the first with attribute FIXED and the second, with attribute FLOAT. However, in procedure A, the arguments C and D both have the FLOAT attribute. Since C is to be fixed-point when it is passed to procedure B, a dummy argument is constructed by converting C from floating-point to fixed-point. This dummy argument is then passed to B.

9.5 CORRESPONDENCE OF PARAMETERS AND ARGUMENTS

If a parameter of an invoked entry is a scalar item, the argument must be a scalar expression. The data attributes of the argument or dummy argument must agree with the corresponding attributes of the parameter. If a constant is used to specify the length of a string parameter or the size of an area parameter in the invoked procedure the value of the length or size expression of the argument must agree with the constant.

If a parameter of an invoked entry is an array, the argument in general must be an array expression with identical bounds and dimensions. The argument may be a scalar expression so long as the ENTRY attribute specifies the DIMENSION attribute and bounds expressions for the relevant parameter as integer constants. In this case, a dummy array argument will be constructed where the value of each element of the array is the value of the scalar expression. The data attributes of the argument must agree with those of the parameter if a dummy has been created. If constants are used to specify the bounds of the parameter in the invoked procedure, the values of the bounds of the array argument must agree with the values of these constants. ALIGNED and UNALIGNED attributes must agree.

If a parameter is a structure, the argument in general must be a structure expression. When a structure description is given for a parameter in an ENTRY attribute specification, a scalar expression may be specified as the corresponding argument. A dummy structure argument will then be constructed where the value of each element of the structure is the value of the scalar expression. The data attributes of the elements of the structure argument must match those of the associated parameter as specified in the invoked procedure. The relative structuring of the argument and the parameter must be the same, although the level numbers need not be identical. ALIGNED and

UNALIGNED attributes must agree. Contained strings and arrays with lengths, areas, and bounds specified by constants must agree.

If a parameter is an area, the corresponding argument must be an area expression. If its size is declared by a constant in the invoked procedure, the corresponding argument must have the same size. This applies to areas in arrays and structures.

If a parameter is a scalar label variable, the argument must be a scalar expression. If a parameter is an array label variable, the argument may be an array label variable or a scalar label expression. A dummy label array argument will be suitably constructed. A dummy argument is always constructed when the argument is a label constant.

If the argument is a statement label constant, this statement label constant is qualified by an identification of the current invocation of the block containing the label. Any reference to the parameter is a reference to the statement label in that environment.

If a parameter is an entry parameter, the corresponding argument must be an entry name. If an entry attribute specification is given for the invoked entry in the invoking procedure, and if the appropriate parameter attribute list specifies that the parameter is an entry name and specifies further (nested) parameter lists, then the argument may also be a generic name. The alternative whose parameter attribute list matches the nested parameter list is selected and passed to the parameter.

Example

```
      .
      .
      .
DCL GEN GENERIC (
                 P2 WHEN ( ),
                 P3 WHEN (FIXED);
DCL X1 ENTRY (ENTRY()) EXT;
DCL X2 ENTRY (ENTRY(FIXED))EXT;
      .
      .
      .
X2(GEN);
will cause P3
to be selected
X1(GEN);
will cause
P2 to be selected
```

If a parameter is a pointer variable, the corresponding argument must be a locator expression. If the argument is an offset variable its value is converted to pointer using the area named in its offset attribute. This offset attribute must specify an area variable, and the parameter must be described as a pointer in the ENTRY attribute. If the argument is an offset function reference, its value is converted to pointer using the area variable named in the offset attribute within the RETURNS attribute in the declaration of the function. This offset attribute must specify an area variable, and the parameter must be described as a pointer in the entry attribute.

If a parameter is an offset variable, the corresponding argument must be a locator expression. If the argument is a pointer expression, an OFFSET attribute specifying an area variable must be used to describe

the parameter in the ENTRY attribute. This area variable is used to convert the pointer expression to an offset relative to the area. If the argument is an offset expression the area variable specifications associated with the argument and the parameter have no affect on argument passing. If different area variables are specified this does not, of itself, cause the creation of a dummy.

If a parameter is a file parameter, the argument must be a file name or file parameter. With the exception of FILE, any file attributes declared for the parameter are ignored.

9.6 ALLOCATION OF PARAMETERS

A parameter may correspond to an argument of any storage class. If more than one generation of the argument exists, however, the parameter is synonymous only with the generation existing at the point of invocation. At least one generation must exist.

9.7 BUILT-IN FUNCTIONS

Besides the function references to procedures written by the programmer, a function reference may invoke one of a comprehensive set of built-in functions.

The set of built-in functions is an intrinsic part of PL/I. It includes not only the commonly used arithmetic functions but also functions for manipulating strings and arrays, as well as other necessary or useful functions related to special facilities provided in the language. The identifiers corresponding to the built-in function names are not reserved. Any such identifier can be used by the programmer for other purposes.

Each built-in function requires a specified number of arguments. For some built-in functions, only a minimum is specified. Additional arguments are optional. For others, a maximum is specified.

When a reference is made to a built-in function, any arguments whose attributes do not match the attributes required by that function are converted to the appropriate form before the function is invoked. The characteristics of the value returned are determined by the function.

Unlike programmer-specified functions, which always return a scalar value, there are many built-in functions that may return an array or structure value when array or structure expressions are used in certain of their argument positions. This facility is useful in array or structure expressions.

A built-in function with no arguments must be declared BUILTIN.

9.8 LIBRARIES

9.8.1 Introduction

A library is a program which provides a set of procedural "entry points" which can be called on by "user programs". Any user can create and use libraries. (A detailed discussion of the usage of libraries is found in Chapter 20 of the B 5000/B 6000/B 7000 Series System Software Operational Guide, Volume 2. Familiarity with this material is assumed in the discussion that follows.)

9.8.2 Creating Libraries

The OPTION list of the main (that is, outer block) PROCEDURE statement must include the EXPORT option. The EXPORT option specifies which entry points the library makes available to the programs that call on the library. (Refer to Section 7.3.3.4, The PROCEDURE Statement, for a more detailed discussion of the EXPORT option.)

To become a library, a program must be "frozen" at run time using the FREEZE statement. (Refer to Section 7.3.2.4A, The FREEZE Statement, for a more detailed discussion of this statement.)

9.8.3 Referencing Libraries

To call a library, the user program declares library names and specifies which procedures within the program call a given library. Libraries must be explicitly declared using the LIBRARY attribute in a DECLARE statement. (Refer to Section 4.7.6, The LIBRARY Attribute, for a more detailed discussion of this attribute.)

An entry point name is associated with a library by adding the OPTIONS attribute to an entry point declaration. (Refer to Section 4.7.7, The OPTIONS Attribute, for a more detailed discussion of this attribute.)

9.8.4 Changing Library Attributes

Library attributes may also be set and tested dynamically using the <system-library-attribute-reference-statement>. (Refer to Section 7.3.8.6, The System Library Attribute Reference Statement, for a more detailed discussion of this statement.)

9.8.5 Entry Points and their Parameters

Parameters must be passed according to certain rules. When a PL/I program calls a library procedure written in another language, the library procedure expects to receive parameters by name or by reference and not by value. When a PL/I library is called by a program written in another language, parameters must not be passed by name.

Parameters to a library entry point may be any of the following types:

```
Binary Fixed (p,q),
...q = 0, p <= 39
...q = 0, 39 < p <= 78
...q neq 0, p <= 39
...q neq 0, 39 < p <= 78

Decimal Fixed(p, q),
...q = 0, p <= 11
...q = 0, 11 < p <= 23
...q neq 0, q <= 11
...q neq 0, 11 < p <= 23

Binary Float(p),
...p <= 39
...39 < p <= 78

Decimal Float(p),
...p <= 11
...11 < p <= 23

Character(n)
Character(*)
Character(n) Varying
Character(*) Varying

Bit(n), n <= 48

Transaction record
Transaction record array
```

SECTION 10. DYNAMIC PROGRAM STRUCTURE

PROGRAM CONTROL

Every program, when it is being executed, has a control that determines the order of execution of the statements. For a discussion of their order see Section 7.2, Sequence of Control.

Execution of the program is initiated by invoking the initial procedure at some entry point.

10.1 PROLOGUES

On entering a block, certain initial actions are performed (e.g. allocation of storage for automatic variables). Initial actions constitute the prologue.

At the beginning of the prologue, the following items are available for computation:

1. The established generation of automatic and defined variables declared outside the block and known within it.

2. Static variables known within the block.

3. Controlled and based variables known within the block.

4. Arguments passed to the block.

The prologue makes available for computation all other variables known within the block as follows:

a. Automatic variables declared in the block.

b. Defined variables declared within the block.

c. Entry names declared within the block.

In making these items available, the prologue may need to evaluate expressions concerned with automatic and defined data. Such expressions may occur specifying lengths, bounds, sizes of areas, and iteration factors, as well as arguments in a CALL option. Expressions of these kinds also occur in RETURNS attribute specifications. These expressions may depend on items of 1, 2, 3, or 4. They may also be dependent on items a, b, and c under the following circumstances: if an item is referred to in an expression and the allocation or initialization of a second item depends on that expression, then that first item must in no way be dependent on the second item for its own allocation and initialization. Further, the first item must in no way be dependent on any other item that so depends on the second item.

Example

The following is illegal:

```
DECLARE (A(M) INITIAL (1),

M INITIAL (A(I))) AUTO;
```

10.2 ACTIVATION AND TERMINATION OF BLOCKS

A BEGIN block is said to be activated when control passes through the BEGIN statement for the block. A procedure block is said to be activated when the procedure is invoked at any one of its entry points. During certain time intervals of the execution of a program, a block may be active. A block is active if it has been activated and not yet terminated.

There are a number of ways in which a block may be terminated. These are implied by the following rules:

1. A BEGIN block is terminated when control passes through the END statement for the block.

2. A procedure block is terminated on execution of a RETURN statement or an END statement for the block. The END statement implies a RETURN statement. See section 7.2, Sequence of Control.

3. A block is terminated on execution of a GO TO statement contained in the block which transfers control to a point not contained in the block. Any intervening blocks are also terminated. This is known as abnormal termination.

4. The execution of a STOP statement causes termination of the major task.

5. The execution of an EXIT statement causes termination of the task containing the statement and all tasks attached to this task. Thus, all blocks corresponding to these tasks are terminated.

6. When a block B is terminated, all of the dynamic descendants of B are also terminated.

7. When a block is terminated all active subtasks created during execution of that block are terminated.

10.2.1 Dynamic Descent

If a block B is activated, another block, B1, may be activated from a point internal to block B while B still remains active. This is possible only in the following cases:

1. B1 is a procedure block immediately contained in B. (The label of B1 is internal to B and reached through a procedure reference.)

2. B1 is a begin block internal to B and reached through normal flow.

3. B1 is a procedure block not contained in B and reached through a procedure reference. B1, in this case, may be identical to B, i.e., a recursive call. However, it is regarded dynamically as a different block.

4. B1 is a begin block or a statement specified by an ON statement and reached through an interrupt. See Section 7.3.5.2, The ON Statement. For present purposes, even if B1 is a statement, it can be regarded as a block, and this case is dynamically similar to case 1 or case 3 above.

If any of the above cases occur, while B1 is active, B1 is said to be an immediate dynamic descendant of B.

Block B1 may itself have an immediate dynamic descendant B2, etc., so that a chain of blocks (B, B1, B2....) is created where by definition, all of the blocks are active. In this chain, each of the blocks B1, B2, etc., is said to be a dynamic descendant of B.

It is important to note that the termination of a given block may automatically imply the termination of other blocks and that these blocks need not necessarily be contained in the given block. Storage for all automatic variables declared in these blocks will be released at the time of termination. See Section 10.4.2, Storage Classes.

10.2.2 Dynamic Encompassing

If block B is a dynamic descendant of block A, then block A dynamically encompasses block B, and block B is dynamically encompassed by block A.

10.2.3 The Environment of a Block Activation

A block is said to statically contain those blocks that are nested within it. The scope of declarations within a block, B, includes those blocks statically contained in B. Certain attributes are evaluated and certain generations established upon entry to a block. The relevant attributes and generations are:

1. Generations of automatic data.

2. Generations of parameters.

3. Bounds, string-lengths, and area sizes of defined data.

4. Bounds, string-lengths, and area sizes within simple parameter attribute lists of entry attributes.

5. String lengths and area sizes within RETURNS attribute specifications and in PROCEDURE and ENTRY statements.

When several activations of B are in existence, as in recursion, it is essential to know which activation of B holds the storage and evaluated attributes of data declared in B and known to a given descendant activation of a block statically contained in B. If a block, B1, is nested within N statically containing blocks, the particular activation of each of the N blocks that hold the evaluated attributes and generations known to B1 form the environment of the activation of B1.

The immediate environment of an activation of a begin block is provided by the activation of the immediate statically containing block that

activates the begin block.

The immediate environment of an activation of a procedure by one of its entry names (i.e., not by an entry parameter) is provided by the activation of the immediate statically containing block of the procedure.

When an entry name is passed as an argument, the immediate environment to be used in subsequent invocations by an entry parameter is determined and passed with it. This environment is provided by the activation, in the current environment of the block that passes the entry name, of the block that statically contains the procedure whose entry name is passed.

The immediate environment of an activation of an ON unit is provided by that activation of the block containing the ON statement in which the ON unit is established.

The immediate environment of an activation of some block, BA, is provided by an activation of the block, B1, which statically contains BA. If BA is nested within the N blocks B1, B2... BN, there is a sequence of block activations such that the activation of BI+1 provides the immediate environment of the activation of BI. This sequence provides the complete environment of the activation of BA.

10.2.4 The Environment of a Label Constant

A label constant written as a label prefix designates a point within the text of a block B. During execution, there may be several activations of B. It is essential to know to which activation of B a reference to the label refers.

A reference to a label constant, L, made in some activation of a block B1 is to L in that activation of B which forms part of the current environment of the activation of B1. (Of course, if B and B1 are the same block, L refers to the current block.) When a label constant is assigned to a label variable, this environmental information is assigned as well. Subsequent GO TO statements naming the label variable will reestablish the environment assigned to the variable, and hence may cause blocks to be terminated.

When a label variable is assigned to another label variable, the environmental information is assigned as well.

10.3 GENERATION OF A VARIABLE

A level-1 generation, or allocation, of a variable is created whenever storage is allocated for the variable. A level-1 generation, or a subgeneration as described below, consists of the storage for the generation and has associated with it a pointer to the generation and the evaluated set of attributes of the generation. The pointer to the generation serves as a unique identification of the generation. The evaluated set of attributes is established when the generation is allocated and enables the contents of the storage to be interpreted.

In the case of static, automatic, and controlled generations, the pointer to the generation can only be obtained by supplying the variable as the argument of the ADDR built-in function. For based variables a locator variable is specified when a based generation is to be created by an ALLOCATE, LOCATE, or READ statement. A value is

assigned to the locator variable, enabling it to be used to access the generation that is created.

The storage for a generation contains the values of the various fields in the variable. The evaluated set of attributes of a generation comprises the structuring of the variable, its ALIGNED or UNALIGNED attribute, the data types of its components, and the bounds of arrays, lengths of strings, and sizes of areas as evaluated at the point of allocation.

A generation of an aggregate or area variable consists of a number of subgenerations. If a generation is an array, each subscripted item in the array is a subgeneration. If a generation is a structure, each item immediately contained within the structure is a subgeneration. An aggregate subgeneration itself contains further subgenerations. An area generation contains a set of subgenerations corresponding to the generations that have been allocated in the area but not freed. If a subgeneration is an area, the attributes of its subgeneration are significant only if one of these subgenerations is being accessed.

Offset variables may be used to identify the position of a generation within an area. The position is not qualified by the area itself, so the offset may be applied to any suitable area. This is achieved by supplying the offset and the area as arguments of the POINTER built-in function. The result is a pointer identifying the generation within the area.

10.4 ALLOCATION OF DATA AND STORAGE CLASSES

221

The efficient use of storage during execution of a program is frequently a crucial consideration. Multiple use of a storage region for different data during program execution can reduce the total amount of storage required.

Provisions are included in the language to give the programmer virtually any degree of control over the allocation of storage for the data variables in a program if he chooses to do so.

10.4.1 Definitions and Rules

Storage is said to be allocated for a variable when storage is associated with the variable. Allocation for a given variable may take place statically, before execution of the program, or dynamically during execution.

Storage may be allocated dynamically for a variable and subsequently released. Thus, this storage is freed for possible use in later allocations. If storage has been allocated for a variable and not subsequently released, the variable is said to be in an allocated state. When a variable appears in a statement of a source program, the appearance is called a reference. If a reference corresponds to the assignment of a value to the variable (e.g., appearance in an expression to be evaluated), the variable must be in an allocated state.

10.4.2 Storage Classes

Every variable in a program must have a storage class, which specifies the manner of storage allocation.

There are four storage classes. The storage class is specified by declaring the variable with one of the four storage class attributes: STATIC, AUTOMATIC, CONTROLLED, or BASED. The storage class may be declared explicitly or be applied by default.

10.4.2.1 The Static Storage Class

Storage for a variable with the attribute STATIC is allocated before execution of the program and is never released during execution.

The scope attribute of a static variable may be INTERNAL or EXTERNAL. An external variable with unspecified storage class has, by default, the STATIC storage attribute.

10.4.2.2 The Automatic Storage Class

If a variable has the attribute AUTOMATIC, the activation and termination of the block containing the declaration of this variable determine storage allocation for the variable. Whenever this block is activated during execution of a program, storage will be allocated for the variable, and the variable will remain in an allocated state until termination of this block. At the time of termination, the storage is released. Thus, the time interval during which the variable is in an allocated state will necessarily include the intervals when the variable is known.

Termination of a block by means of a GO TO, STOP, or EXIT statement may imply simultaneous termination of other blocks and, consequently, simultaneous release of storage for all automatic variables declared in these blocks.

If the block is activated recursively (reactivated one or more times before return), the previous generation of an automatic variable or parameter is pushed down on each entrance and popped up on each return to yield the proper generation of storage for the variable after each return, until the final return out of the procedure.

The terms "pushed down" and "popped up" refer to the notion of a push-down stack. A push-down stack is a logical device S, similar in behavior to a physical stacking process. When an element is placed in S, it is conceptually placed on top of the elements already in S, which are "pushed down." At any time, if S is not empty, the top element (i.e., the element most recently placed in S) can be removed from S, and the remaining elements are "popped up".

The scope attribute of an automatic variable must be INTERNAL. An internal variable with unspecified storage class has, by default, the AUTOMATIC storage class attribute.

10.4.2.3 The Controlled Storage Class

The ALLOCATE statement may specify one or more controlled variables. Execution of the statement causes the allocation of storage for the variables specified.

The FREE statement may specify one or more controlled variables, and execution of the statement causes the storage allocated for the current generation variables to be released.

More than one ALLOCATE statement specifying the same variable, without an intervening FREE statement creates a push-down stack of generations of that variable. A FREE statement always frees the topmost generation.

Generations that are not explicitly freed are freed automatically upon termination of the task in which they are allocated.

At some point in a program, it may not be known whether a controlled variable X is in an allocated state. The built-in function ALLOCATION is provided to test this state. The function reference ALLOCATION (X) will return the value '1' B if any generation of X is in an allocated state, and the value '0' B if not.

The SCOPE attribute of a controlled variable may be either INTERNAL or EXTERNAL.

Example

```
A:  PROCEDURE;

        DECLARE X STATIC;
             .
             .
             .
        B:  PROCEDURE;
             DECLARE Y (100) CONTROLLED,
               Z CHARACTER (1000);
                  .
                  .
                  .
             ALLOCATE Y:
             .
             .
             FREE Y;
                 .
                 .
                 .
             C: BEGIN
                DECLARE Z (100):
                    .
                    .
                    .
                END C;
                .
                .
                .
             RETURN;
                 .
                 .
                 .
             END B;
             .
             .
             .
         END A;
```

Assume in the above example that the termination of procedure A occurs on the return implied by END A. The termination of procedure B occurs on the RETURN statement, and the termination of block C occurs at END C. Then in this example:

Storage for the static variable X is allocated before execution and is never released.

The character string variable Z is AUTOMATIC by default. Storage is allocated for this Z on entry to procedure B and is released on execution of the RETURN statement.

The array variable Z is AUTOMATIC by default. Storage is allocated for this Z at the beginning of execution of block C and is released at END C.

Storage for the CONTROLLED variable Y is allocated on execution of the ALLOCATE statement and released on execution of the FREE statement. After execution of the FREE statement, the variable Y presumably is not used, but the character string variable Z can be used, since storage is not released for this variable until the

termination of procedure B.

10.4.2.4 The Based Storage Class

The BASED attribute specifies that generations of the declared variable may be allocated under the control of the programmer. A based variable can be allocated by use of the ALLOCATE statement (optionally in a specified area) and freed by use of the FREE statement. A based variable can be allocated in a buffer by use of the LOCATE statement. Such a generation is freed when the record is transmitted by a subsequent LOCATE or WRITE statement for the same file, or when the file is closed.

A based generation may also be allocated by a READ statement with a SET option. Such a generation is freed by the execution of a subsequent READ statement for the same file, or when the file is closed.

A based reference comprises two parts which together enable a generation to be accessed. Firstly, there is a based variable which provides the attributes of the generation. Secondly, there is a locator qualifier which identifies the generation. This qualifier is obtained from the based attribute of the based variable unless a qualifier is specified in the reference. A qualifier specified in the reference overrides any qualifier given in the based attribute.

When a BASED variable is used to access a generation, the data attributes of the BASED variable and the accessed generation must agree.

BASED variables need not be allocated. BASED references may be used to access generations in any storage class. The ADDR built-in is used to obtain a pointer value which will identify a non-based generation. A based reference refers to an allocated generation if its locator qualifier has a defined value.

10.5 INTERRUPT OPERATIONS

During the course of program execution any one of a certain set of conditions may occur that can result in an interrupt. An interrupt operation causes the suspension of normal program activities in order to perform a special action. After the special action, program activities may or may not resume at the point where they were suspended.

For conditions recognized by PL/I, the special action to be taken when an interrupt occurs may be specified in an ON statement. A complete list and description of the conditions can be found in appendix 2. Each condition is named with a unique identifier suggestive of the condition (e.g., ZERODIVIDE specifies the condition obtained whenever an attempt is made to divide by zero). This collection of names is an intrinsic part of the language, but the names are not reserved. The programmer may use them for other purposes, so long as no ambiguity exists.

10.5.1 Purpose of the Condition Prefix

In general, during the execution of a statement, a condition may be in either an enabled or disabled state.

If a particular condition is enabled and an interrupt occurs during execution of the statement, the action specification for the condition is executed. This action specification may either be standard system action or it may have been specified by the programmer through the use of an ON statement.

If a particular condition is disabled during execution of a statement the result is unpredictable if the condition occurs, except for UNDERFLOW which gives a result of zero.

By means of condition prefixes, the programmer can control the enabled/disabled status of the following conditions:

CONVERSION
SIZE
UNDERFLOW
FIXEDOVERFLOW
OVERFLOW
STRINGRANGE
SUBSCRIPTRANGE
ZERODIVIDE

The appearance of any of the above keywords in a prefix list causes the associated condition to be enabled for the scope of the prefix. The appearance of any of the above preceded by a NO (with no separating blank) causes the associated condition to be disabled for the scope of the prefix.

10.5.2 Scope of the Condition Prefix

The scope of the prefix depends upon the statement to which it is attached.

If the statement is a PROCEDURE statement or BEGIN statement, the scope of the prefix is the block defined by this statement, including all nested blocks, except those blocks and statements for which the condition is respecified. The scope does not include procedures that lie outside the scope as defined above but which may be invoked by the execution of statements in this scope.

If the statement is an IF statement or an ON statement, the scope of the prefix does not include the blocks or groups that are part of the statement. Any such block may also have an attached prefix, whose scope rules are implied by the other rules given here.

For any other statement, the scope of the prefix is that of the statement itself, including any expressions evaluated during the execution of the statement but not any procedure explicitly called by the statement.

10.5.3 Use of The ON Statement

In order to define the action to be taken when an interrupt occurs, the programmer may write an ON statement. See section 7.3.5.2, ON statement for the general form of the statement, the syntax, and other details.

When an ON statement that is internal to a given block (for example, a block B) is executed, it causes a preparatory action with the following effect:

If, during the execution of any statement after the execution of the ON statement and before the termination of block B (including the execution of statements in all dynamic descendants of block B), if the condition specified in the ON statement ever occurs and an interrupt results, the statement or begin block specified in the ON statement will be executed as though it were invoked as a procedure block. (If SNAP also has been specified, a program dump will occur). Control normally will be returned to the point of interrupt or to the statement following the one that was interrupted.

When an ON statement specifying a given condition is executed, the action to be taken is established by the execution. The time interval during which this on-unit is effective is defined above in the description of the effect of an ON statement. There are two qualifications to this descriptor.



1. If after a given action is established by execution of an ON statement, and while this on-unit is still effective, another ON statement specifying the same condition is executed, then this latter ON statement will take effect as described above, so that its specified action will determine the interrupt action for the given condition. (The effect of the old ON statement is either temporarily suspended or completely nullified, depending upon whether or not the new ON statement is in a block dynamically descendant from the block to which the old ON statement is internal. See section 7.3.5.2, the ON statement, and section 7.3.5.3, the Revert Statement for more details.

2. There are eight conditions whose names (possibly preceded by the word NO without intervening blanks) may appear in a condition prefix. Even when one of these conditions appears in an ON statement, occurrence of the condition will not necessarily result in an interrupt. For an interrupt to occur, there are certain additional requirements, which are described in the following paragraph.

There are two of these eight conditions, SIZE and STRINGRANGE for which an interrupt will not take place when the condition occurs unless the programmer specifically so designates. He may enable this condition by explicitly specifying the condition in a prefix whose scope will cover the calculation where the condition may occur. If a calculation resulting in the occurrence of either of these conditions does not lie within the scope of such a prefix, no interrupts will occur. Five of these eight conditions, namely OVERFLOW, UNDERFLOW, ZERODIVIDE, CONVERSION, and FIXEDOVERFLOW are always enabled. But the programmer may specifically designate that the compiler is to not generate code to handle the associated interrupt when it occurs. The compiler will generate code to be executed avoiding program termination when the

interrupt produced by the condition occurs unless the occurrence is in a calculation lying within the scope of a prefix specifying NOOVERFLOW, NOUNDERFLOW, NOZERODIVIDE, NOCONVERSION, or NOFIXEDOVERFLOW, respectively.

The remaining condition, SUBSCRIPTRANGE, cannot be named in prefixes, but is always enabled and cannot be disabled.

10.5.4 System Interrupt Action

Each of the conditions has a standard action defined for it if an interrupt should occur. If no established on-unit is in force for a given condition at the time that condition is raised and causes an interrupt, then standard system action will be taken. Standard system action is dependent upon the nature of the condition. If the programmer does not want the system action in the case where one of these conditions may occur and cause an interrupt, he must specify an alternative action for the condition through use of the ON statement.

In some situations, the programmer may want to specify his own action for a given condition, to have it hold for part of the execution of the program, and then have this specification nullified and allow the standard system action. In this case, he may use the keyword SYSTEM, as follows:

```
ON <condition-name> SYSTEM;
```

Example 1:

```
A:  PROCEDURE;
      .
      .
      .
    ON OVERFLOW
          BEGIN;
          DECLARE NUMBOV STATIC
           INITIAL (0);
          NUMBOV=NUMBOV + 1;
          IF NUMBOV = 100 THEN GO
           TO OVERR;
          END;
      .
      .
      .
    ON OVERFLOW;
      .
      .
      .
    ON OVERFLOW SYSTEM;
      .
      .
      .
     OVERR:
     END A;
```

In the above example, assume that the program consists only of procedure A, that the three ON statements are the only ON statements involving the OVERFLOW condition, that they are internal to procedure A, and that they are executed in their physical order.

When program execution begins, the OVERFLOW condition is enabled by the

system. Any floating-point overflow condition that occurs before the first ON OVERFLOW statement is executed will result in an interrupt, with standard system action. However, the execution of the first ON OVERFLOW statement establishes the action specified in the begin block (the number of overflows is counted and if this number has not reached 100, the action is finished.) Any OVERFLOW interrupts will receive this action until the second ON OVERFLOW statement is executed. The action specified here is a null statement. Any subsequent OVERFLOW interrupts will effectively be ignored until control reaches the third ON OVERFLOW statement, which reestablishes the standard system action.

Example 2:

```
(SIZE): A:        PROCEDURE:

                    .
                    .
                  ON SIZE GO TO AERR;
                    .
                    .
                  CALL B;
                    .
                    .
                    .
                  END A;
(SIZE, NOOVERFLOW) :B: PROCEDURE;
                    .
                    .
                    .
                  ON SIZE GO TO BERR;
                    .
                    .
                    .
                  RETURN;
                  END B;
```

In the above example the prefix (SIZE) enables that condition for procedure A and specifies that if a SIZE condition occurs during any calculation in procedure A, an interrupt is to take place. The prefix (SIZE, NOOVERFLOW) for procedure B specifies the same requirement with respect to a SIZE error for procedure B. In addition, it specifies for procedure B that any interrupt that might be caused by an OVERFLOW condition is not to be anticipated by compiler-generated fault-handling code.

After the beginning of execution of procedure A, and before the execution of the first ON statement, any SIZE condition will result in an interrupt with standard system action. After execution of this ON statement, and before execution of the ON statement in the invoked procedure B, any SIZE condition will result in an interrupt with the action GO TO AERR. After execution of the ON statement in procedure B the action GO TO BERR becomes established for the SIZE condition, but the effect of the previous ON statement is suspended only temporarily. After the RETURN statement in procedure B is executed, the effect of this previous ON statement is reinstated, so that SIZE conditions occurring after this point again result in the action GO TO AERR.

If any floating-point overflow condition occurs during execution of procedure A, an interrupt will result with the standard system action for the OVERFLOW condition. However, for any occurrence of an OVERFLOW condition during the execution of procedure B, the interrupt will be handled by compiler-generated code.

Example 3:

```
X: PROCEDURE
  DECLARE A, B;
  ON OVERFLOW BEGIN;
            PUT DATA (A,B);
            END;
  .
  .
  Y:  BEGIN;
  DECLARE A,B;
  .
  .
  END Y;
  .
  .
  .
  END X;
```

This example illustrates the effect of establishment of the generation of variables at the time a condition is executed. If the OVERFLOW condition should arise, the values transmitted by the PUT statement in the on-unit will be the values of the variables A and B that are declared in the outer block. This is true, even if the OVERFLOW condition should arise during execution of the begin block Y, where A and B have been redeclared.

10.5.5 Use of the Revert Statement

The REVERT statement may be used, following an ON statement, to reinstate an action specification that existed in the immediate, dynamically encompassing block at the time the descendant block was invoked.

Example

```
(SIZE):  A: PROCEDURE;
       ON SIZE GO TO AERR;
           .
           .
         CALL B;
           .
           .
           .
         END A;
(SIZE): B: PROCEDURE;
         ON SIZE GO TO BERR;
           .
           .
           .
         REVERT SIZE;
           .
           .
           .
         END B;
```

In the above example, if a SIZE condition occurs in procedure B after execution of the ON statement, an interrupt will take place with the resulting action GO TO BERR. After execution of the REVERT statement, the condition as specified by the ON statement in procedure A is

reinstated. Program control remains in procedure B, but any subsequent SIZE condition that occurs in procedure B will cause an interrupt with the action GO TO AERR.

10.5.6 Programmer-Named Conditions

An identifier can by used to create a condition name by means of the keyword CONDITION used in the ON statement, as follows:

ON CONDITION (<identifier>) <on-unit>

Such a statement contextually declares the identifier to be a condition name and the execution of the statement provides an on-unit. The condition can be caused to occur only by the execution of a SIGNAL statement. See section 7.3.5.4, Signal Statement.

For example, if the following statement is executed:

ON CONDITION (ABC) <block>

and later the following statement is executed:

SIGNAL CONDITION (ABC);

Then the latter execution will (by definition of the SIGNAL statement) cause an interrupt, with the action defined by the block in the ON statement.

10.5.7 Condition Built-In Functions and Pseudo Variables

The condition built-in functions are provided for the investigation of interrupts. Each such function is associated with certain conditions:

BUILT-IN FUNCTION	associated condition(s)
ONFILE	I/O conditions, and "conversion" raised during an I/O operation
ONLOC	All Conditions
ONSOURCE	Conversion-condition
ONCHAR	Conversion-condition
ONKEY	I/O Condition or conversion condition raised by an operation on a KEYED file
ONCODE	All conditions
DATAFIELD	Name-condition
ONCOUNT	I/O condition

Appendix 1 gives the value returned by these functions when they are used in the following contexts:

1. An on-unit for one of the associated conditions for the function, or an "error" on-unit entered as standard system action for one of the associated conditions.

2. A block, B, which is a dynamic descendant of such an on-unit, provided that no intervening block is an on-unit for one of the conditions associated with the function nor is any such block an "error" on-unit entered as standard system action for, or normal return from, any of the conditions associated with the function.

   In all other contexts the value of a condition built-in function is the null character string, except for ONCHAR which is the single character blank and ONCODE and ONCOUNT which have the value zero.

   The values returned by the condition built-in functions are inherited by blocks in a manner analogous to the inheritance of on-units.

   Example If the CONVERSION condition occurs in a CONVERSION on-unit, the values of ONSOURCE and ONCHAR are stacked before the new CONVERSION on-unit is entered. Within the new CONVERSION on-unit ONCHAR and ONSOURCE have values determined by this second conversion interrupt. The values pertaining to the first interrupt are reestablished when control returns from the second on-unit.

   The condition pseudo-variables (i.e., ONCHAR and ONSOURCE, may be used within blocks defined in 1 and 2 above to alter a character string value whose conversion has raised a CONVERSION interrupt). If the source for the conversion is a variable or pseudo-variable, or if it is specified by the SUBSTR built-in function with a variable as its first argument, then within these blocks an assignment to the condition pseudo-variables is an assignment to the variable concerned. It is an error to make assignment to the condition pseudo variables in any of the blocks.

Note that a SIGNAL statement does not prevent inheritance by the signaled on-unit of a value returned by a condition built-in function, unless an on-unit for the condition specified in the SIGNAL statement, entered by normal means, would have reset the value of the function.

The values returned by built-in functions are shown in the following table:

	CONDITION					
FUNCTION	NAME	CONVERSION	ENDFILE, TRANSMIT, RECORD	KEY	ENDPAGE, UNDEFINED-FILE	OTHER
ONFILE	Set	Set if I/O else copy	Set	Set	Set	Copy
ONLOC	Set	Set	Set	Set	Set	Set
ONSOURCE	Copy	*Set	Copy	Copy	Copy	Copy
ONCHAR	Copy	*Set	Copy	Copy	Copy	Copy
ONKEY	Copy	*Set if KEY conversion else copy	*Set if keyed, else copy	*Set	Copy	Copy
DATAFIELD	*Set	Copy	Copy	Copy	Copy	Copy
ONCODE	Set	Set	Set	Set	Set	Set
ONCOUNT	Copy	Copy	Copy or Reduce	Copy or Reduce	Copy	Copy

Note:

1. "Copy" means that the descendant on-unit inherits the value pertaining at the time of its invocation.

2. "Set" means that the value for the preceding on-unit is stacked and a new value is given for the descendant on-unit.

3. "Reduce" means that the descendant on-unit inherits a reducing value of ONCOUNT.

4. "*" means that if the condition is specified in a SIGNAL statement, the character string is set to the null string.

5. When the ERROR condition is raised as standard system action for a condition, the functions return the same values as in an on-unit for the condition.

6. When the RECORD condition is raised for a keyed file, other than by SIGNAL, "ONKEY" is non-zero if storage allocation was successful.

SECTION 11. COMPILE-TIME FACILITIES

The PL/I compiler gives the programmer the opportunity to change his source program at compile-time. For example, the programmer may, at compile-time, include in his program strings of PL/I text that are located in the files which may have already been defined. The programmer may modify the source program by changing identifier names, or he may indicate that certain sections of his program are not to be compiled. These operations, which can be performed at compile-time, are handled by the preprocessor phase of the PL/I compiler.

11.1 PREPROCESSOR INPUT

PL/I statements which indicate how the source program is to be altered at compile-time are known as COMPILE-TIME STATEMENTS. Compile-time statements are identified by the fact that they all begin with a percent sign (%). A compile-time statement can appear anywhere in the source program, except in a character string. A percent sign in a character-string is treated as a character in the string and not as the leading character of a compile-time statement. Compile-time statements cannot be nested (i.e., one compile-time statement cannot contain another compile-time statement).

Besides using compile-time statements to specify compile-time action, the programmer may also use compile-time procedures, which are invoked by function references in compile-time statements. For compile-time procedures, only the PROCEDURE statement and the END statement require a leading percent sign.

Compile-time statements and compile-time procedures are the only input to the preprocessor phase of the PL/I compiler.

11.1.1 Effects of Compile-Time Statements

When a PL/I program is compiled, all source program statements are scanned and then saved for later use by the compiler. However, if during the scanning process a compile-time statement is encountered, the scanning will discontinue and the preprocessor will execute the compile-time statement. The execution of a compile-time statement by the preprocessor may have one of the following effects upon the continuation of the compilation.

1. Inform the scanning process, when it finds a certain identifier in the source program remaining to be scanned, that the value of the identifier is to be scanned as part of the source program text rather than the identifier itself. In this case the identifier must have had a value assigned to it in a previous compile-time statement, or it may have a value assigned to it in a compile-time statement before the next time the identifier is encountered in the source program. The value of the identifier, which is scanned instead of the identifier itself, may contain another identifier whose value is to be scanned as part of the source program text. See Section 11.2.1, Rescanning and Replacement of Compile-Time Identifiers.

2. Direct the scanning process to continue the scan from another point in the source program.

3. Direct the scanning process to continue the scan from a user-defined file containing PL/I program text. This user-defined file may contain compile-time statements which may themselves effect the continuation of the compilation in any of these three ways.

If, however, the scanning process encounters no compile-time statements in the source program, then the source program will be compiled exactly as it appears in the symbol file.

11.2 PREPROCESSOR OUTPUT

There is no separate listing for output from the preprocessor. The actual output of the preprocessor is the source program statements which were modified by the execution of compile-time statements during the scanning of the source program.

11.2.1 Rescanning And Replacement of Compile-Time Identifiers

The scanning of the value of an identifier or the invocation of a compile-time procedure cannot occur unless the identifier (or entry name of the procedure) has been established in a compile-time DECLARE statement (or compile-time PROCEDURE statement) and the identifier (or procedure name) is activated. The identifier or entry name may be activated by appearing in a compile-time ACTIVATE statement or in a compile-time DECLARE statement.

If an identifier or procedure name is encountered in the source test, and it has been activated for replacement by a compile-time DECLARE or ACTIVATE statement, then the value of the identifier may be replaced. This new value of the identifier is then rescanned by the preprocessor, from left to right. If there are any activated identifiers in this value then they are inserted at the second level of replacement and the value is now rescanned by the preprocessor for further replacement. After all possible replacements have been made, the value of the identifier is inserted into the source program text.

The value which is inserted for the activated identifier cannot contain unmatched comment delimiters, unmatched quotation marks, or percent signs, if it is activated for rescanning.

Example

1. Suppose the following three compile-time statements and one source program statement appeared in a PL/I program.

```
% DCL EXPR CHARACTER, X FIXED;
%EXPR = 'X * Y' ;
% X = 4;
ANS = EXPR;
```

The statement ANS = EXPR; would result in the source text;

```
ANS = 4 * Y;
```

The first statement in the above example is a compile-time DECLARE statement which establishes EXPR and X as compile-time identifiers with the specified attributes. This statement also implicitly activates the two identifiers. The second statement is a compile-time statement which assigns the character-string 'X * Y' to EXPR. The third

statement is also a compile-time statement which assigns X the value of 4. The next statement is a source program statement which assigns EXPR to ANS. Since EXPR is a compile-time identifier and is activated for replacement, and since it has a value ('X * Y'), then this value will be rescanned from left to right for possible further replacement. This rescanning will cause the compile-time identifier "X" to be replaced by its value (i.e.,4). The value '4 * Y' is now rescanned again, but since there are no more compile-time identifiers, replacement is stopped and the source program statement now becomes ANS = 4 * Y ;. A blank is appended to each end of the value when it is inserted into the source program.

11.3 COMPILE-TIME VARIABLES

A compile-time variable is an identifier which has been declared in a %DECLARE-statement with either the "CHARACTER" or "FIXED" attribute. Compile-time variables can be declared with these two attributes only, and an attempt to specify any other attributes will result in a syntax error. The default attributes DECIMAL and PRECISION (5,0) are given to compile-time identifiers declared as FIXED, and the default attributes VARYING with no maximum length are given to identifiers declared with the CHARACTER attribute. No contextual or implicit declarations of identifiers in compile-time statements are allowed.

The appearance of an identifier in a %DECLARE statement serves to implicitly activate the identifier. By using %DEACTIVATE and %ACTIVATE statements, the programmer may control the activation of compile-time variables. Deactivation of a compile-time identifier does not mean that the identifier loses its value. Rather, a deactivated compile-time identifier is one that when it is encountered in the source-program text will not be replaced by its value. A deactivated compile-time variable may have its value altered by a compile-time statement while it is deactive.

When a compile-time variable is active, the value of that identifier will replace all occurrences of that identifier in any PL/I source-program statement. If a compile-time variable is deactive, or if it is active but has no value assigned to it, then replacement of that identifier in source-program statements will not take place.

The scope of a compile-time variable is defined as all source program text that is scanned during the scanning process of the compilation of the PL/I program. However, if a compile-time variable is declared in an INCLUDE file, then the scope of that compile-time variable is the INCLUDE file plus all subsequently scanned source-program text. If a previously declared compile-time identifier is redeclared in a compile-time procedure, then the scope of this new declaration is the procedure in which it is redeclared. At all other times the original declaration of the identifier will be used.

11.4 COMPILE-TIME EXPRESSIONS

Compile-time expressions are identical to source program expressions except for the following limitations:

1. There is no exponentiation allowed at compile-time.

2. All arithmetic operations are handled as decimal integer arithmetic. (NOTE: Division will thus be integer division, i.e., 9/10 = 0, not 0.9).

3. Operands are limited to compile-time variables, references to compile-time procedures, decimal integer constants, bit-string constants, and character-string constants. When references to the built-in-functions SUBSTR, INDEX, and LENGTH are made, the arguments to these built-in-functions must be compile-time expressions.

4. Repetition factors are not allowed when the operand of the compile-time expression is a character string.

5. Character strings in compile-time expressions which are assigned to compile-time target variables cannot include compile-time statements as part of the character-string.

11.5 COMPILE-TIME PROCEDURES

A compile-time procedure is a procedure whose PROCEDURE statement and END statement are preceded by a percent sign, A compile-time procedure may be invoked only during the compilation of a PL/I program.

Syntax

```
<compile-time procedure> ::=

% <label>:[<label>:]...PROCEDURE[(<identifier>
    [,<identifier>]...)] RETURNS ((CHARACTER|FIXED));
          .
          .
          .
  [<label>:] RETURN (<compile-time expression>);
          .
          .
          .
% [<label>:] END [<label>];
```

If a label is used following END, it must be one of the labels that preceded the keyword PROCEDURE in the PROCEDURE statement.

The <compile-time expression> in the RETURN statement will be converted to the attribute specified in the RETURNS option of the PROCEDURE statement and then returned to the point of invocation of the procedure.

The optional list of identifiers in the PROCEDURE statement is the parameter list for the procedure. Each parameter in the list must be explicitly declared within the procedure. All names declared in the source program are global to the procedure, unless they have been redeclared within the procedure.

Compile-time procedures may be used only as functions (i.e., they must return a value). These procedures cannot be terminated by executing a GO TO statement transferring control to some point outside the procedure. The only way to terminate a compile-time procedure is by executing a RETURN statement specifying the value to be returned. The value which is returned by the procedure may, upon rescanning, reinvoke the procedure.

A compile-time procedure may consist of the following statements or groups, and no others:

ASSIGNMENT STATEMENT
DECLARE STATEMENT
DO-GROUP
GO TO STATEMENT
IF STATEMENT
NULL STATEMENT
RETURN STATEMENT
PUT STATEMENT

The syntax for all of the above statements and groups, except the RETURN statement are described later in this section. When these statements appear within compile-time procedures, they may not be preceded by a leading percent sign.

11.5.1 Declaring Compile-Time Procedures

In order for a compile-time procedure to be executed, the entry name of the procedure must be established as an entry name for a compile-time procedure. This may be done by declaring the name to have the ENTRY attribute in a %DECLARE statement or by the appearance of the identifier in a %PROCEDURE statement. Unlike a %DECLARE statement the appearance of an identifier in a %PROCEDURE statement will not activate the identifier for replacement. However, the compile-time procedure name may be activated or deactivated explicitly anywhere within the scope of either type of declaration.

If any reference is made to a compile-time procedure in the source program text, then the declaration of the entry name must have been made. The procedure itself may not have been scanned yet. If any replacement is to take place in the source text, the entry name must also be active. Otherwise, the procedure reference itself will be inserted into the source program text.

11.5.2 Execution of Compile-Time Procedures

Compile-time procedures may be executed as part of the source program text or as part of compile-time statements. When executed in the source program text, the procedure returns a value which is inserted into the text of the program, and when executed as part of a compile-time statement, the procedure returns a value which becomes part of a compile-time procedure.

When an activated compile-time procedure name is encountered in the source text, then each argument in the argument list is scanned for possible replacement. When replacement, if any, is completed the procedure is invoked and the arguments are converted to the attributes of the corresponding parameters in the procedure statement. If the value returned by the procedure is fixed, then it is converted to a character-string and inserted into the source program text. If the compile-time procedure returns a character-string, then the string returned will be rescanned for any possible replacements before it is inserted in the source program.

When a compile-time procedure is invoked in a compile-time statement, the action taken is basically the same as that for compile-time procedures invoked in source program text, except that the value returned is used in a compile-time expression and not inserted into the source program text.

11.6 COMPILE-TIME BUILTIN-FUNCTIONS

At compile-time, only the following builtin-functions may be used:

SUBSTR
LENGTH
INDEX

These functions may be referenced in either source text or compile-time statements. In either case the names of these functions will be recognized as built-in functions, unless the names have been redeclared in the source program as variables or entry names.

If any of these names appear in a %INCLUDE statement the interpretation of these names outside the INCLUDE statement will not be affected.

11.7 COMPILE-TIME STATEMENTS

11.7.1 Activate and Deactivate Statements

An ACTIVATE statement is used to activate compile-time identifiers for replacement in the source program text. The DEACTIVATE statement, on the other hand, deactivates compile-time identifiers so that their appearance in source program text will not cause them to be replaced. Instead the identifier will remain unchanged.

Syntax

```
<activate-statement> ::=
%[<label>:]...ACTIVATE<identifier>[RESCAN | NORESCAN]
                      [,<identifier>[RESCAN | NORESCAN]]...;

<deactivate-statement> ::=
%[<label>:]...DEACTIVATE,identifier>[,<identifier>]...;
```

Semantics

Only compile-time variables, compile-time entry names, and compile-time built-in functions may be activated and deactivated.

In order for an activated compile-time variable to be replaced in the source program text, it must be immediately preceded and followed by a PL/I delimiter, and it cannot appear in a comment or character-string. If these conditions are met, the value of the compile-time identifier will be converted to character-string type and rescanned for further replacement (unless the NORESCAN option is specified). After all possible replacements are made, the value will be inserted into the source program text.

The RESCAN and NORESCAN options of the ACTIVATE statement apply to compile-time variables of type CHARACTER or to compile-time procedures which return values of type CHARACTER. For all other types of compile-time identifiers the option is meaningless.

RESCAN is the default specification. RESCAN indicates that the value of the identifier is to be scanned and replacements are to continue until no more replacements can be made. At this time the value of the identifier will be placed into the source program

text.

NORESCAN indicates that the current value of the compile-time variable or the value returned by the compile-time procedure is to be inserted into the source program text without rescanning for any further replacement.

A compile-time identifier may be changed from RESCAN to NORESCAN, or vice versa, simply by specifying the identifier in another %ACTIVATE statement with the RESCAN option set to the desired status. It is not necessary to deactivate the identifier before specifying it in another %ACTIVATE statement.

The appearance of an identifier in a %DECLARE statement will serve to implicitly activate the identifier. However, the RESCAN option and NORESCAN options cannot be specified in a %DECLARE statement.

The DEACTIVATE statement is used to indicate which compile-time identifiers will not cause replacement when they appear in the source program text. Deactivating a compile-time identifier will not have any effect on the value which the identifier had before deactivation, or on any value which it might obtain during deactivation. Deactivation merely causes the identifier itself, rather than the value of the identifier, to be inserted into the source program text.

Activating an already activated compile-time identifier has no effect, unless it is to change the RESCAN option status of the identifier. Deactivating an already deactivated compile-time identifier has no effect.

11.7.2 Assignment Statement

The compile-time assignment statement is used to assign values to compile-time target variables and to evaluate compile-time expressions.

Syntax

<compile-time assignment statement> ::=

%[<label>:]...<compile-time variable>=
<compile-time-expression>;

Semantics

All arithmetic operations are decimal integer of PRECISION (5,0). This means that all arithmetic operands will be converted to DECIMAL FIXED PRECISION (5,0) before any operations are performed.

Character strings which are to be converted to FIXED type must be in the form of an optionally signed decimal integer constant.

Fixed-point decimal values which are converted to character-strings will always result in a string 8 characters long (i.e., 5 + 3).

Values assigned to compile-time variables of type CHARACTER may include percent signs, unmatched comment delimiters, and unmatched quotation marks.

11.7.3 Declare Statement

The compile-time DECLARE statement is used to establish identifiers as compile-time variables or compile-time procedure names. The appearance of an identifier in a %DECLARE statement implicitly activates that identifier for replacement in the source program text (with the option RESCAN set by default if applicable).

Syntax

```
<compile-time declare-statement>  ::=
    %[<label>:]...DECLARE<identifier><attribute-list>
                        [,<identifier><attribute-list>]...;
<attribute-list> ::=CHARACTER | FIXED | ENTRY |
                     <initial attribute>
```

Semantics

Factoring of % DECLARE statements is permitted. (See DECLARE statement, section 7.3.4.1.)

The syntax and semantics of the <initial-attribute> may be found in section 4.13.2. The <initial-call> form of the <initial-attribute> may not be used.

Although labels are allowed preceding a %DECLARE statement, they are essentially ignored.

The scope of compile-time variables is defined in section 11.3, Compile-Time Variables.

Compile-time declarations must appear in the PL/I program to be known. However, the declaration may appear anywhere in the program, excluding any compile-time procedures in which the identifier is redeclared.

Only the four attributes FIXED, CHARACTER, ENTRY, and INITIAL may appear in the <attribute-list>. CHARACTER implies VARYING with no maximum length, and FIXED implies DECIMAL with PRECISION (5,0). These implied attributes are supplied by the compiler as default attributes.

The <initial-attribute> may be used to initialize a compile-time identifier to a certain value upon its activation.

Multiple declarations of compile-time identifiers are not allowed.

11.7.4 Do-Group

Syntax

```
<compile-time do-group> ::=
          %[<label>:]...DO [ I = M1  [TO M2 [BY M3]]
                           |         [BY M3 [TO M2]]]  ;
```

Semantics

I represents a compile-time variable, and M1, M2, and M3 represent compile-time expressions.

The statements of a do-group may consist of both compile-time statements and source-program statements. If source-program statements are used, they are scanned for any possible replacement but they are not executed.

The expansion of a compile-time do-group is the same as a PL/I source-program do-group, with compile-time statements replacing corresponding source-program statements.

Program control cannot be transferred into an iterative do-group, except by return from a compile-time procedure invoked within the group.

11.7.5 GO TO Statement

The GO TO statement is used to direct the scanning of the source program text to a specified label.

Syntax

```
<compile-time GO TO statement> ::=
    %[<label>:]...  {GO TO | GOTO}  <label1>;
```

Semantics

1. The label to which the compile-time GO TO statement directs the scanning process must be the label of a compile-time statement.

2. A compile-time GO TO statement may transfer the scanning process from an include-file to the containing text of the include-file. However, the reverse rule is not true.

11.7.6 If Statement

The compile-time IF statement is used to control the scanning process of the PL/I source program at compile-time.

Syntax

```
<compile-time IF statement> ::=

%[<label>:]...IF<compile-time expression>
              %THEN<compile-time-group1>;
              %ELSE<compile-time-group2>;
```

Semantics

Either compile-time-group may be a single executable compile-time statement or a compile-time do-group.

The <compile-time-expression> is evaluated and converted to a bit-string. If any bit in the string has the value of 1, then compile-time-group1 is executed and compile-time-group2 is skipped over (if it is present). If the bit-string is all zeros, then compile-time-group1 is skipped and compile-time-group2 is executed (if it is present). In the second case if compile-time-group is not present, then the statement following the if-statement is executed.

If the compile-time-group which is executed is not a compile-time GO TO statement, then the scanning continues with the statement immediately following the IF statement. Otherwise, the scanning continues at the label indicated by the GO TO statement.

Compile-time-group1 and/or compile-time-group2 may themselves be compile-time IF statements. For rules concerning this, see section 7.3.2.6, IF Statement.

11.7.7 Include Statement

The INCLUDE statement is used to include external sources of PL/I program text in the source program being compiled.

Syntax

```
<include-statement>::=
%[<label>:] INCLUDE <include-file-name>
[,<include-file-name>]...;

 <include-file-name>::=<file-title>
```

Semantics

Each <include-file-name> is used to identify a source of PL/I text external to the program being compiled. Each external source may contain compile-time-statements or source program statements.

Include-files may not be nested more than five levels deep at one time.

The include-file is scanned in sequence just as if it had appeared in the source program text.

Compile-time GO TO statements may transfer the scanning process from an include-file to the containing text, but the reverse is illegal.

If the include-file contains compile-time IF statements or compile-time do-groups, they must be completely contained within the include-file.

11.7.8 Null Statement

The compile-time NULL statement is used to place compile-time labels into the source program.

Syntax

```
<compile-time null-statement> ::=
%[<label>:]...;
```

11.7.9 Put Statement

The compile-time PUT statement allows the user to output the values of any or all of the currently known compile-time variables.

Syntax

```
<compile-time PUT statement>::=

     PUT DATA[(<compile-time-
     variable>[,<compile-time-
     variable>]...)];
```

Semantics

If no <compile-time-variable> list occurs, then the values of all currently known compile-time-variables will be printed. Otherwise, only the values of the variables listed will be printed.

The format of the output is:

```
<compile-time-variable>=<value>;
```

APPENDIX 1. BUILTIN FUNCTIONS

All of the built-in functions and pseudo-variables that are available to the PL/I programmer are given in this appendix, and are presented in alphabetical order under their respective headings. The general organization of the appendix is as follows:

1. Computational built-in functions.
    A. String-handling built-in functions.
    B. Arithmetic built-in functions.
    C. Mathematical built-in functions.
    D. Array manipulation built-in functions.
2. Condition built-in functions.
3. Based storage built-in functions.
4. Multi-tasking built-in functions.
5. Miscellaneous built-in functions.
6. Pseudo-variables.

The computational built-in functions provide string handling, arithmetic operations (addition, division, etc.), mathematical operands (trigonometric functions, square root, etc.), and array manipulation. The computational built-in functions are given below. Allowable abbreviations are given in Appendix 3.

String Handling:

AFTER	DECAT	REPEAT
BEFORE	EXCEPT	REVERSE
BIT	FROM	STRING
BOOL	HIGH	SUBSTR
CHAR	INDEX	TRANSLATE
COLLATE	LENGTH	UNSPEC
COPY	LOW	UPTO
		VERIFY

Arithmetic:

ABS	MAX
ADD	MIN
BINARY	MOD
CEIL	MULTIPLY
DECIMAL	PRECISION
FIXED	ROUND
FLOAT	SIGN
FLOOR	TRUNC
DIVIDE	

Mathematical:

ATAN	LOG10
ATAND	LOG2
ATANH	RANDOM
COS	SIN
COSD	SIND
COSH	SINH
ERF	SQRT
ERFC	TAN
EXP	TAND
LOG	TANH

Array Manipulation:

ALL	LBOUND
ANY	PROD
DIM	SUM
HBOUND	

The condition built-in functions allow the PL/I programmer to investigate interrupts arising from enabled conditions. The condition built-in functions are:

DATAFIELD	ONFILE
ONCHAR	ONKEY
ONCODE	ONLOC
ONCOUNT	ONSOURCE

The based storage built-in functions are designed to facilitate list processing and the use of based storage. They mainly return special values to locator and area variables. The based storage built-in functions are:

ADDR	NULL	OFFSET
EMPTY	NULLO	POINTER

The miscellaneous built-in functions perform various duties: for example, one function provides the current date, another provides a count of data items transmitted during a "stream' input/output operation, while another provides an indication of whether or not a controlled variable is in an allocated state. The miscellaneous built-in functions are:

ALLOCATION	DATE	PROCTIME
COUNT	IOTIME	TIME
COMPILETIME	LINENO	

Each of the pseudo-variables is described briefly. A more detailed description can be found in the discussion of the corresponding built-in function. The pseudo-variables are:

COMPLETION	PRIORITY
COMPLEX	REAL
IMAG	STATUS
ONCHAR	SUBSTR
ONSOURCE	UNSPEC

COMPUTATIONAL BUILT-IN FUNCTIONS

STRING HANDLING BUILT-IN FUNCTIONS

The functions described in this section may be used for manipulating strings. Unless it is specifically stated otherwise, any argument can be a scalar or aggregate expression. (See Section 5.1.2.1, Built-in Functions with Aggregate Arguments.) The conversion of arguments is discussed with each built-in function. Where necessary, conversions are performed according to the rules for data conversion.

AFTER String Built-In Function

Definition: AFTER returns the substring of a given string which appears after the leftmost occurrence of a second string within the first string.

Reference: AFTER (S,C)

Arguments: The two arguments must be specified. The first argument, S, represents the source string. The second argument, C, represents the configuration for which S is to be searched to determine the substring of S after the leftmost occurrence of C.

Before the function is invoked, the arguments are converted, if necessary, to be either both character or bit strings, in the same manner as for the INDEX built-in function. The result string has the same type as the converted arguments.

Result: If C is the null string, the result is the string S. If C is not the null string but does not appear in S, then the result is the null string. If C is not null and does appear in S, then the result is:

```
SUBSTR(S,INDEX(S,C)+LENGTH(C))
```

The AFTER built-in function produces results identical to the results of the DECAT built-in function with a specification-string of '001'B.

BEFORE String Built-in Function

Definition: BEFORE returns the substring of a given string which appears before the leftmost occurrence of a second string within the first string.

Reference: BEFORE (S,C)

Arguments: Two arguments must be specified. The first argument, S, represents the source string. The second argument, C, represents the configuration for which S is to be searched to determine the substring of S before the leftmost occurrence of C.

Before the function is invoked, the arguments are converted, if necessary, to be either both character or bit strings, in the same manner as for the INDEX built-in function. The result has the same type as the converted arguments.

Result: If C is the null string, the result is the null string. If C is not the null string but does not appear in S, then the result is the string S. If C is not null and does appear in S, then the result is:

SUBSTR(S,1,INDEX(S,C)-1)

The BEFORE built-in function produces results identical to the results of the DECAT built-in function with a specification-string of '100'B.

BIT String Built-In Function

Definition: BIT converts a given value to a bit string and returns the result to the point of invocation. This function allows the programmer to control the size of the result of a bit string conversion.

Reference: BIT (<value> [,<length>])

Arguments: The argument, <value>, is an expression representing the quantity to be converted to a bit string. It is converted according to the normal rules for assignment to bit string. The argument, <length> when specified, is an expression whose value, after any necessary conversion to integer, gives the length of the result. IF <length> is not specified, the length of the bit string returned by BIT is the length of the argument <value> after any necessary conversion to bit string.

Result: The value returned by this function is <value> converted to a bit string and extended to the right with zeros, or truncated on the right, as necessary, to the integral value of <length>.

BOOL String Built-In Function

Definition: BOOL produces a bit string whose bit representation is a result of a given boolean operation on two given bit strings.

Reference: BOOL (X,Y,W)

Arguments: Arguments X and Y are the two bit strings upon which the boolean operation specified by W is to be performed. If X and Y are not bit strings, they are converted according to the normal rules for assignment to bit strings. If X and Y differ in length, the shorter string is extended with zeros on the right to match the length of the longer string.

Argument W represents the boolean operation. It is a bit string of length 4 and is defined as N1 N2 N3 N4, where each N is either 0 or 1. There are 16 possible bit combinations and thus 16 possible boolean operations. If necessary, W is converted to a bit string (of length 4) before the function is invoked.

Result: The value returned by this function is a bit string, Z, whose length is equal to the longer of X and Y. Each bit of Z is determined by the boolean operation on the corresponding bits of X and Y as follows: The I-th bit of Z is set to the value of N1, N2, N3, or N4, depending on the combination of the I-th bits of X and Y as shown in the boolean table below:

Xi	Yi		Zi
0	0		N1
0	1		N2
1	0		N3
1	1		N4

CHAR String Built-In Function

Definitions: CHAR converts a given value to a character string and returns the result to the point of invocation. This function allows the programmer to control the size of the result of a character-string conversion.

Reference: CHAR (<value> [,<length>)]

Arguments: The argument, <value>, is an expression representing the quantity to be converted to a character string. Conversion is performed according to the normal rules for assignment to a character string. The argument, <length>, when specified, is an expression whose integral value gives the length of the result. if <length> is not specified, the length of the character string returned by CHAR is the length of the argument <value> after any necessary conversion to character string.

251

Result: The value returned by this function is <value> converted to a character string and extended to the right with blanks or truncated on the right, as necessary, to the integral value of <length>.

COLLATE String Built-In Function

Definition: COLLATE returns a character string in which each character of the EBCDIC character set occurs once, the order being that of the low-to-high collating sequence.

Reference: COLLATE [()]

Result: The value returned by this function is a character string of length 256 in low-to-high collating sequence order.

COPY String Built-In Function

Definition: COPY takes a given string value and forms a new string consisting of a special number of copies of the string value concatenated together.

Reference: COPY (<string>,<factor>)

Arguments: The argument <string> represents the string from which the new string will be formed. If this argument is not a string, it will be converted, before the function is applied, to a character string.

The argument <factor> is an expression the integral value of which specifies the number of copies of <string> that are to appear concatenated together in the result string; <factor> can be signed.

Result: the returned value will be a character string containing <factor> occurrences of the value <string>. If <factor>is less than or equal to zero, the returned value is the null string.

DECAT String Built-In Function

Definition: DECAT returns substrings of a given string relative to the occurrence of a second string within the first string dependent upon a specified control value.

Reference: DECAT (S,C,<specification-string>)

Arguments: The three arguments must be specified. The first argument, S, represents the source string. The second argument represents the configuration for which S is to be searched to determine the substring(s) of S relative to the occurrence of C. Before the function is invoked, these arguments are converted, if necessary, to be either both character or bit strings, in the same manner as for the INDEX built-in function. The result string has the same type as the converted arguments.

The third argument, <specification-string>, is a three-bit string the value of which specifies the substring(s) to be returned.

Result: If C is the null string, the result is:

Specification-String	Result
'000'B	null
'001'B	S
'010'B	null
'011'B	S
'100'B	null
'101'B	S
'110'B	null
'111'B	S

If C is not the null string and appears in S the result is:

Specification-String	Result
'000'B	null
'001'B	AFTER(S,C)
'010'B	C
'011'B	FROM(S,C)
'100'B	BEFORE(S,C)
'101'B	BEFORE(S,C)||AFTER(S,C)
'110'B	UPTO(S,C)
'111'B	S

If C is not the null string but does not appear in S, then the result is the null string for a <specification-string> in the range '000'B to '011'B, while the result is S for a <specification-string> in the range '100'B to '111'B.

For example, the result of DECAT('ABCDEFGHIJ','EFGHIJ','011'B) is 'EFGHIJ', and the result of DECAT('ABRACADABRA','R','101'B) is 'ABACADABRA'.

EXCEPT String Built-In Function

Definition: EXCEPT takes a character string, extracts from a second character string all characters which appear in the first string, and returns a character string consisting of the remainder of the second string.

Reference: EXCEPT (R[,S])

Arguments: At least one argument must be specified. The first argument, R represents a character string of characters to be removed from the string represented by the second argument, S, which is the source string.

Before the function is invoked, the arguments are converted, if necessary, to be both character strings.

If the second argument, S, is not specified, the character string returned by the COLLATE built-in function is used.

Result: The result is a character string identical to S after all characters appearing in R have been removed from S. If all characters in R are contained in S, the result is the null string. If R is the null string, the result is S.

If S is omitted, the result is a character string in low-to-high collating sequence order containing all characters in the EBCDIC character set except those appearing in R.

For example, the result of EXCEPT('ABC','QCBDAFGQARD,?') is 'QDFGRD,?'.

FROM String Built-In Function

Definition: FROM returns the substring of a given string which appears beginning with the leftmost occurrence of a second string within the first string.

Reference: FROM (S,C)

Arguments: The two arguments must be specified. The first argument, S, represents the source string. The second argument, C, represents the configuration for which S is to be searched to determine the substring of S beginning with the leftmost occurrence of C.

Before the function is invoked, the arguments are converted, if necessary, to be either both character or bit strings, in the same manner as for the INDEX built-in function. The result string has the same types as the converted arguments.

Result: If C is the null string, the result is S. If C is not the null string but does not appear in S, then the result is the null string. If C is not null and does appear in S, then the result is:

```
C||SUBSTR(S,INDEX(S,C)+LENGTH(C))
```

The FROM built-in function produces results identical to the results of the DECAT built-in function with a specification-string of '011'B.

For example, the result of FROM('CHARACTER','A') is 'ARACTER'.

HIGH String Built-in Function

Definition: HIGH forms a character string of a given length from the highest character in the collating sequence; that is, each character in the constructed string is the highest character in the collating sequence.

Reference: HIGH (<length>)

Argument: The argument, <length>, is an expression whose value, after any necessary conversion to integer, specifies the length of the string that is to be formed.

Result: The value returned by this function is a character string whose length is determined by the integral value of <length>; each character in the string is the highest character in the collating sequence.

INDEX String Built-In Function

Definition: INDEX searches a specified string for a specified bit or character string configuration. If the configuration is found, the starting location of that configuration within the string is returned to the point of invocation.

Reference: INDEX (<string>, <config>)

Arguments: Two arguments must be specified. The first argument, <string>, represents the string to be searched; the second argument, <config>, represents the bit or character string configuration for which string is to be searched. Before the function is invoked, the arguments are converted, if necessary, to either character or bit strings. The following table indicates the target attributes in each case:

		"CONFIG"			
		CHAR	BIT	BIN	DEC
"STRING"	CHAR	CHAR	CHAR	CHAR	CHAR
	BIT	CHAR	BIT	BIT	CHAR
	BIN	CHAR	BIT	BIT	CHAR
	DEC	CHAR	CHAR	CHAR	CHAR

Pictured binary, pictured decimal, and pictured character arguments are treated as binary, decimal, and character, respectively, for this purpose.

Result: The value returned by this function is a binary integer of default precision. This binary integer is either of the following:

1. The location in <string> at which <config> has been found. If more than one <config> exists in <string>, the location of the first one found (in a left-to-right sense) will be returned.

2. The value 0, if <config> does not exist within <string> or if either of the arguments has a length of zero.

LENGTH String Built-In Function

Definition: LENGTH finds the string length of a given value and returns it to the point of invocation.

Reference: LENGTH (<string>)

Argument: The argument, <string>, represents a character string or a bit string whose length is to be found. The argument need not represent a string; if it does not, it is converted before the function is invoked to a character string (if the argument is DECIMAL) or a bit string (if the argument is BINARY).

Result: The value returned by this function is a fixed binary integer of precision (38, 0), giving the current length of STRING. If STRING is an array expression, an array of identical bounds is returned.

Example If XYZ is a varying-length character string whose maximum length is 30, but whose current length is 25, then the statement:

```
I = LENGTH (SUBSTR (XYZ,4));
```

will assign a binary value of 22 to I.

LOW String Built-In Function

Definition: LOW forms a character string of specified length from the lowest character in the collating sequence (i.e., each character of the formed string will be the lowest character in the collating sequence).

Reference: LOW (<length>)

Argument: The argument, <length>, is an expression whose value, after any necessary conversion to integer, specifies the length of the string being formed.

Result: The value returned by this function is a character string whose length is determined by the integral value of LENGTH; each character in the string is the lowest character in the collating sequence.

REPEAT String Built-In Function

Definition: REPEAT takes a given string value and forms a new string consisting of the given string value concatenated with itself a specified number of times.

Reference: REPEAT (<string>, <factor>)

Arguments: The argument <string> represents the string from which the new string will be formed. If this argument is not a string, it will be converted, before the function is applied, to a character string if the argument is decimal, or to a bit string if the argument is binary.

The argument <factor> is an expression whose integral value specifies the number of times that <string> is to be concatenated with itself; <factor> can be signed.

Result: The value returned by this function is <string> concatenated with itself <factor> times. In other words, the returned value will be a string containing (factor + 1) occurrences of the value <string>. If <factor> is less than or equal to zero, the returned value is identical to the argument (i.e., the converted argument, if the original argument was not a string).

REVERSE String Built-In Function

Definition; REVERSE forms a string which is the same as a given string with the order of the elements reversed.

Reference: REVERSE (<string>)

Argument: The argument to REVERSE is a string; if it is not, it is converted to a character string before invocation of the function.

Result: The result of this function is a string of the same length as the argument but whose elements are those of the argument taken from right to left.

For example, the result of REVERSE('ASABIYAH') is 'HAYIBASA'.

STRING String Built-In Function

Definition: STRING concatenates all the elements in the result of an expression into a single string element.

Reference: STRING (X)

Arguments: The argument, X, is an element, array, or structure expression, the result of which is composed either entirely of character strings and/or decimal numeric character data, or entirely of bit strings and/or binary numeric character data.

Result: The value returned by this function is an element bit string or character string, the concatenation of all the elements in X. If X contains one or more varying strings, the result is a varying string.

SUBSTR String Built-In Function

Definition: SUBSTR extracts a substring of user-defined length from a given string and returns the substring to the point of invocation. (SUBSTR can also be used as a pseudo-variable.)

Reference: SUBSTR (<string>, I [,J])

Arguments: The argument <string> represents the string from which a substring will be extracted. If this argument is not a string, it will be converted, before the function is applied, to a character string if the argument is decimal, or to a bit string if the argument is binary. Argument I represents the starting point of the substring and J represents the length of the substring. Arguments I and J must be integers or expressions that can be converted to integers.

Assuming that the length of <string> is K, arguments I and J must satisfy the following conditions:

1. J must be less than or equal to K and greater than or equal to 0.

2. I must be less than or equal to K and greater than or equal to 1.

3. The value of I+J-1 must be less than or equal to K.

Thus, the substring, as specified by I and J must lie within <string>.

If J is not specified, it is assumed to be equal to the value of K-I+1. In other words, it is assumed to be the length of the remainder of <string>, beginning at the I-th position in <string>.

When these conditions are not satisfied, the SUBSTR reference causes the STRINGRANGE-interrupt to be raised, if it is enabled.

Result: The value returned by this function is a varying-length string whose current length is defined as follows:

1. If J=0, the returned value is the null string.

2. If J is greater than 0, the returned value is that substring beginning at the I-th character or bit of the first argument and extending J characters or bits.

3. If J is not specified, the returned value is that substring beginning at the I-th character or bit and extending to the end of <string>.

TRANSLATE String Built-In Function

Definition: TRANSLATE forms a character string whose value is the result of applying a given translation to a given character string, and returns the result to the point of invocation.

Reference: TRANSLATE (S,R[,M])

Arguments: The first argument, S, represents the source character string to be translated. The arguments R and M control the translation. Arguments that are not strings are converted to character strings. If R is shorter than M, it is padded on the right to the length of M with blanks. If M is omitted, COLLATE( ) is assumed as the value of M.

Result: The value returned by this function is a character string whose length is that of S. The value r of each character in the result is related to the character c in the corresponding position in S by the following formula:

r=SUBSTR(R,INDEX(M,c),1)

unless INDEX(M,c) is 0, in which case r=c.

Thus, the string S is searched character by character until a character is found which is a member of the string M. When such a character is found it is replaced by the character in R appearing in the position in R corresponding to its position in M.

For example, the result of TRANSLATE('5206900','ADGJMPTW',23456789') is 'JA0MW00'.

UNSPEC String Built-In Function

Definition: UNSPEC returns a bit string that is the internal coded representation of a given value. (UNSPEC can also be used as a pseudo-variable.)

Reference: UNSPEC (X)

Argument: The argument, X, may be an arithmetic, string, locator, or area expression, or an area variable, whose internal coded representation is to be found.

Result: The value returned by this function is the internal coded representation of X. This representation is in bit-string form, the length of this string depends upon the attributes of X.

UPTO String Built-in Function

Definition: UPTO returns the substring of a given string which appears up to and including the leftmost occurrence of a second string within the first string.

Reference: UPTO (S,C)

Arguments: The two arguments must be specified. The first argument, S, represents the source string. The second argument, C, represents the configuration for which S is to be searched to determine the substring of S terminated by the leftmost occurrence of C.

Before the function is invoked, the arguments are converted, if necessary, to be either both character or bit strings, in the same manner as for the INDEX built-in function. The result string has the same type as the converted arguments.

Result: If C is the null string, the result is the null string. If C is not the null string but does not appear in S, then the result is S. If C is not null and does appear in S, then the result is:

SUBSTR(S,1,INDEX(S,C-1))||C

The UPTO built-in function produces results identical to the results of the DECAT built-in function with a specification-string of '110'B.

For example, the result of UPTO('ENOLAG','O') is 'ENO'.

VERIFY String Built-In Function

Definition: VERIFY examines two given strings to verify that each character in the first string occurs in the second string, returning a fixed binary value of zeros if this is the case; otherwise, the value returned is the index of the first character (in the first string) that does not occur in the second string.

Reference: VERIFY (<string-one>,<string-two>)

Arguments: The arguments must be specified and may be any expression. If an argument is not a character string, it is converted, before the function is invoked, to a character string.

Result: Each character, "c", of <string-one> is examined to determine if:

INDEX(<string-two>,c) ¬=0.

The characters of <string-one> are examined from left to right to check whether they occur in <string-two>. If they all do, the function returns a value of zero; if a character does not occur in <string-two>, the function returns the binary integer value, of default precision, corresponding to the index of that character in <string-one>.

For example, the result of VERIFY('ABCDEFGHIJ','ABCDEFFHID') is 7.

ARITHMETIC BUILT-IN FUNCTIONS

All values returned by ARITHMETIC built-in functions are in coded ARITHMETIC form. The arguments of these functions should also be in that form. If an argument is not coded ARITHMETIC, then, before the function is invoked, it is converted to coded ARITHMETIC according to the rules for data conversion. Note, therefore, that in the function descriptions below, a reference to an argument always means the converted argument, if conversion was necessary.

In some function descriptions, the phrase CONVERTED TO THE HIGHEST CHARACTERISTICS is used; this means that the rules for mixed characteristics are followed. See Section 5.2.2.1, Mixed Characteristics.

In general, an argument of an ARITHMETIC built-in function may be a scalar or aggregate expression. (See Section 5.1.2.1, Built-in Functions with Aggregate Arguments.)

Unless it is specifically stated otherwise:

1. The mode of an argument must be real.

2. The BASE, SCALE, MODE, and PRECISION of the returned value are determined according to the rules for the conversion of expression operands.

In many of these built-in function descriptions, the symbol N is used. This symbol represents the maximum precision permitted by the machine for the given base and scale.

ABS Arithmetic Built-In Function

Definition: ABS finds the absolute value of a given quantity and returns it to the point of invocation.

Reference: ABS (X)

Argument: The quantity whose absolute value is to be found is given by X.

Result: The value returned by this function is the absolute value of X. If X is real, the result is the positive value of X. The MODE of the result is REAL, while the BASE, SCALE and PRECISION are the same as those of X.

ADD Arithmetic Built-In Function

Definition: ADD finds the sum of two given values and returns it to the point of invocation. This function allows the programmer to control the precision of the result of an add operation.

Reference: ADD (X,Y,P[,Q])

Arguments: Arguments X and Y represent the values to be added, arguments P and Q must be decimal integer constants specifying the precision of the result; Q may be signed. If the scale of the result is a fixed-point, both P and Q must be specified; if the scale of the result is floating-point, only P must be specified. In either case, P must not exceed N.

Result: The value returned by this function is the sum of X and Y. The precision of the result is determined by P and Q; this precision is maintained throughout the execution of the function. The result has the BASE, SCALE, and MODE of X + Y.

BINARY Arithmetic Built-In Function

Definition: BINARY converts a given value to binary base and returns the converted value to the point of invocation. This function allows the programmer to control the precision of the result of a binary conversion.

Reference: BINARY (X[,P[,Q]])

Arguments: The first argument, X, represents the value to be converted to binary base. Arguments P and Q, when specified, must be decimal integer constants giving the precision of the binary result; Q may be signed. The precision of a fixed-point result is (P,Q); the precision of a floating-point result is (P). If both P and Q are omitted, the precision of the result is determined according to the standard rules for data conversion. Note that Q must be omitted for floating-point arguments.

Result: The value returned by this function is the binary equivalent of X. The scale and mode of this value are the same as those of X. The precision is given by P and Q.

CEIL Arithmetic Built-In Function

Definition: CEIL determines the smallest integer that is greater than or equal to a given real value and returns that integer to the point of invocation.

Reference: CEIL (X)

Result: The value returned by this function is the smallest integer that is greater than or equal to X. The BASE, SCALE, MODE, and PRECISION are the same as those of X, with one exception: if X is a fixed-point value of PRECISION (P,Q), the precision of the result is defined as:

(MIN(N,MAX(P-Q+1,1)),0)

DECIMAL Arithmetic Built-In Function

Definition: DECIMAL ARITHMETIC BUILT-IN FUNCTION

Definition: DECIMAL converts a given value to decimal base and returns the converted value to the point of invocation. This function allows the programmer to control the precision of the result of a decimal conversion.

Reference: DECIMAL (X[,P[,Q]])

Arguments: The first argument, X, represents the value to be converted to decimal base. Arguments P and Q, when specified, must be decimal integer constants giving the precision of the decimal result; Q may be signed. The precision of a fixed-point result is (P,Q); the precision of a floating-point result is (P). If both P and Q are omitted, however, the precision of the result is determined according to the standard rules for data conversion. Note that Q must be omitted for floating-point arguments.

Result: The value returned by this function is the decimal equivalent of the argument X. The scale and mode of this value are the same as argument X; its precision is given by P and Q.

DIVIDE Arithmetic Built-In Function

Definition: DIVIDE divides a given value by another given value and returns the quotient to the point of invocation. This function allows the programmer to control the precision of the result of a divide operation.

Reference: DIVIDE (X,Y,P[,Q])

Arguments: The argument X, is the dividend and argument Y is the divisor. Arguments P and Q (Q is optional and may be signed) must be decimal integer constants specifying the precision of the result. If the result is a fixed-point, P and Q must both be specified; if the result is a floating-point value, only P must be specified. In either case, P must not exceed N.

Result: The value returned by this function is the quotient resulting from the division of X by Y. The precision of the result is determined by P and Q as described above; this precision is maintained throughout the execution of this function. The result has the base, scale and mode of X/Y.

FIXED Arithmetic Built-in Function

Definition: FIXED converts a given value to fixed-point scale and returns the converted value to the point of invocation. This function allows the programmer to control the precision of the result of a fixed-point conversion.

Reference: FIXED (X[,P[,Q]])

Argument: The first argument, X, represents the value to be converted to fixed-point scale. Arguments P and W, when specified, must be decimal integer constants (Q can be signed) giving the precision of the result, (P,Q). If Q is omitted, zero is assumed. If both P and Q are omitted, precision of the result will be default fixed-point precision for the base of X.

Result: The value returned by this function is the fixed-point equivalent of the argument X; its precision is (P,Q).

FLOAT Arithmetic Built-in Function

Definition: FLOAT converts a given value to floating-point scale and returns the converted value to the point of invocation. This function allows the programmer to control the precision of the result of a floating-point conversion.

Reference: FLOAT (X[,P])

Arguments: The first argument, X, represents the value to be converted to a floating-point scale. The second argument, P, when specified, must be a decimal integer constant giving the precision of the result. If P is omitted, precision of the result will be floating-point default precision for the base of X.

Result: The value returned by this function is the floating-point equivalent of X; its precision is P.

FLOOR Arithmetic Built-In Function

Definition: FLOOR determines the largest integer that does not exceed a given value and returns that integer to the point of invocation.

Reference: FLOOR(X)

Result: The value returned by this function is the largest integer that does not exceed X. The BASE, SCALE, MODE, and PRECISION of this value are the same as those of X, with one exception: if X is a fixed-point value of PRECISION (P,Q), the precision of the result is:

(MIN(N,MAX(P-Q+1,1)),0)

MAX Arithmetic Built-In Function

Definition: MAX extracts the highest-valued expression from a given set of two or more expressions and returns that value to the point of invocation.

Reference: MAX (X1, X2, ..., XN)

Arguments: Two or more arguments must be given.

Result: The value returned by MAX is the value of the maximum-valued argument. The returned value is converted to conform to the highest characteristics of all the arguments that were specified. If the arguments are fixed-point values and have the following precisions:

(MIN(N,MAX(P1-Q1, ..., PN-QN)+
MAX (Q1,...,QN)), MAX (Q1,...QN))

MIN Arithmetic Built-In Function

Definition: MIN extracts the lowest-valued expression from a given set of two or more expressions and returns that value to the point of invocation.

Reference: MIN (X1,X2, ...,XN)

Arguments: Two or more arguments must be given.

Result: The value returned by MIN is the value of the lowest-valued argument. The returned value is converted to conform to the highest characteristics of all the arguments that were specified. If the arguments are fixed-point values and have precisions:

(P1,Q1), (P2,Q2),...,(PN,QN)

Then the precision of the result is as follows:

(MIN(N,MAX (P1-Q1 ..., PN-QN) +
MAX (Q1,...,QN)), MAX (Q1,...,QN))

MOD Arithmetic Built-In Function

Definition: MOD extracts the remainder resulting from the division of the real quantity by another and returns it to the point of invocation.

Reference: MOD (X1,X2)

Arguments: Two arguments must be given. Before the function is invoked, the base and scale of each argument are converted according to the standard rules for data conversion.

Result: The value returned by MOD is the positive remainder resulting from the division of X1 by X2. If the result is in floating-point scale, its precision is the higher of the precisions of the arguments; if the result is in fixed-point scale, its precision is defined as follows:

(MIN(N, P2-Q2+MAX (Q1,Q2)), MAX(Q1,Q2))

where (P1,Q1) and (P2,Q2) are the precision or X1 and X2, respectively.

MULTIPLY Arithmetic Built-In Function

Definition: MULTIPLY finds the product of two given values and returns it to the point of invocation. This function allows the programmer to control the precision of the result of a multiplication operation.

Reference: MULTIPLY (X1,X2,P[,Q])

Arguments: Arguments X1 and X2 represent the values to be multiplied. Arguments P and Q (Q is optional and may be signed) are decimal integer constants specifying the precision of the result. If the result is a fixed-point value, P and Q must both be specified; if the result is a floating-point value, only P must be specified. In either case, P must not exceed N.

Result: The value returned by this function is the product of X1 and X2. The precision of the result is as specified; this precision is maintained throughout the execution of the function. The result has the BASE, SCALE and MODE of X*Y.

PRECISION Arithmetic Built-In Function

Definition: PRECISION converts a given value to a specified precision and returns the converted value to the point of invocation.

Reference: PRECISION (X,P[,Q])

Arguments: The first argument, X, represents the value to be converted to the specified precision. Arguments P and Q (Q is optional and may be signed) are decimal integer constants specifying the precision of the result. If X is a fixed-point value, P and Q must be fixed-point values, P and Q must be specified; if X is a floating-point value, only P must be specified.

Result: The value returned by this function is the value of X converted to the specified precision. The BASE, SCALE, and MODE of the returned value are the same as those of X.

ROUND Arithmetic Built-In Function

Definition: ROUND rounds a given value at a specified digit and returns the rounded value to the point of invocation.

Reference: ROUND (<expression>, N)

Arguments: The first argument, <expression>, is an element or array representing the value (or values, in the case of an array expression) to be rounded; the second argument, N, is a signed or unsigned decimal integer constant specifying the digit at which the value of <expression> is to be rounded. Rounding occurs at the N-th digit to the right of the decimal (or binary) point in the value of <expression>. Note that the decimal (or binary) point is assumed to be at the left for floating-point values.

Result: For fixed-point values, ROUND returns the value of <expression> rounded at the N-th digit to the right of the decimal (or binary) point.

If <expression> is a floating-point expression, the second argument is ignored, and the rightmost bit in the internal floating-point representation of the expressions value is set to 1 if it is 0. If the

rightmost bit is 1, it is left unchanged.

If <expression> is a string, the returned value is the same string unmodified.

The BASE, SCALE, MODE and PRECISION of the returned value are those of the value of <expression>, with one exception: if the value of <expression> is fixed-point of PRECISION (P,Q), the result is a fixed-point precision:

(MIN (P+1, N), P)

Note that the rounding of a negative quantity results in the rounding of the absolute value of that quantity.

SIGN Arithmetic Built-In Function

Definition: SIGN determines whether a value is positive, negative, or zero, and it returns an indication to the point of invocation.

Reference: SIGN (X)

Result: This function returns a real fixed-point binary value of PRECISION (38,0) according to the following rules:

1. If the argument is greater than 0, the returned value is 1.

2. If the argument is equal to zero, the returned value is zero.

3. If the argument is less than zero, the returned value is -1.

TRUNC Arithmetic Built-In Function

Definition: TRUNC truncates a given value to an integer as follows: first it determines whether a given value is positive, negative, or equal to zero. If the value is negative, TRUNC returns the smallest integer that is greater than the value; if the value is positive or equal to zero, TRUNC returns the largest integer that does not exceed that value.

Reference: TRUNC (X)

Result: If X is less than zero, the value returned by TRUNC is CEIL (X). If X is greater than or equal to zero, the value returned by TRUNC is FLOOR (X). In either case, the BASE, SCALE, and MODE of the result are the same as those of X. If X is a floating-point value, the precision remains the same. If X is a fixed-point value of PRECISION (P,Q), the precision of the result is:

(MIN(N,MAX(P-Q+1,1)),0)

MATHEMATICAL BUILT-IN FUNCTIONS

All arguments to the mathematical built-in functions should be in coded arithmetic form and in floating-point scale, any argument that does not conform to this rule is converted to coded arithmetic and floating-point before the function is invoked, according to the standard rules for data conversion. Note, therefore, that in the function description below, a reference to an argument always means the converted argument, if conversion was necessary.

In general, an argument to a mathematical built-in function may be a scalar or aggregate expression. (See Section 5.1.2.1, Built-in Functions with Aggregate Arguments.)

All of the mathematical built-in functions return coded arithmetic floating-point values. The mode, base, and precision of these values are always the same as those of the arguments.

ATAN Mathematical Built-In Function

Definition: ATAN finds the arctangent of a given value and returns the result expressed in radians, to the point of invocation.

Reference: ATAN (X[,Y])

Arguments: The argument X must always be specified; the argument Y is optional. If Y is omitted, X represents the value whose arctangent is to be found.

If Y is specified, then the value whose arctangent is to be found is taken to be the expression X/Y. In this case, both X and Y must be real, and both cannot be equal to 0 at the same time.

Result: When X alone is specified, the value returned by ATAN depends on the mode of X. If X is real, the returned value is the arctangent of X, expressed in radians where:

$$-pi/2 < ATAN(X) < pi/2$$

If both X and Y are specified, the possible values returned by this function are defined as follows:

1. IF Y> 0, the value is arctangent (X/Y) in radians.
2. IF X > 0 and Y = 0, the value is (pi/2) radians.
3. If X >= 0 and Y < 0, the value is (pi+ arctangent (X/Y)) radians.
4. If X > 0 and Y = 0, the value is (-pi/2) radians.
5. If X < 0 and Y < 0, the value is (-pi+ arctangent (X/Y)) radians.

ATAND Mathematical Built-In Function

Definition: ATAND finds the arctangent of a given real value and returns the result, expressed in degrees, to the point of invocation.

Reference: ATAND (X[,Y])

Arguments: Arguments X and Y (Y may be omitted) must be real values. If Y is omitted, X represents the value whose arctangent is to be found. If Y is specified, the value whose arctangent is to be found is represented by the expression X/Y; in this case, both X and Y cannot be equal to 0 at the same time.

Result: If Y is not specified, the value returned by the function is simply the arctangent of X, expressed in degrees, where:

$$-90 < \text{ATAND}(X) < 90$$

If Y is specified, the value returned by this function is ATAN (X,Y), except that the value is expressed in degrees and not in radians, that is the returned value is defined as:

$$\text{ATAND}(X,Y) = (180/\pi)*\text{ATAN}(X,Y)$$

COS Mathematical Built-In Function

Definition: COS finds the cosine of a given value, which is expressed in radians, and returns the result to the point of invocation.

Reference: COS (X)

Argument: The value whose cosine is to be found is given by X; this value must be expressed in radians.

Result: The value returned by this function is the cosine of X.

COSD Mathematical Built-In Function

Definition: COSD finds the cosine of a given real value, which is expressed in degrees, and returns the result to the point of invocation.

Reference: COSD (X)

Argument: The value whose cosine is to be found is given by X; this value must be real and must be expressed in degrees.

Result: The value returned by this function is the cosine of X.

COSH Mathematical Built-In Function

Definition: COSH finds the hyperbolic cosine of a given value and returns the result to the point of invocation.

Reference: COSH (X)

Argument: The value whose hyperbolic cosine is to be found is given by X.

Result: The value returned by this function is the hyperbolic cosine of X.

ERF Mathematical Built-In Function

Definition: ERF finds the error function of a given real value and returns it to the point of invocation.

Reference: ERF (X)

Argument: The value for which the error function is to be found is represented by X.

Result: This value returned by this function is defined as follows:

$$ERF(X) = \left(2 / SQRT(PI) * INTEGRAL\left(EXP(-t^2)dt\right)_0^X\right)$$

ERFC Mathematical Built-In Function

Definition: ERFC finds the complement of the error function (ERF) for a given real value and returns the result to the point of invocation.

Reference: ERFC (X)

Argument: The argument, X, represents the value whose error function complement is to be found: X must be real.

Result: The value returned by this function is defined as follows:

$$ERFC(X) = 1-ERF(X)$$

EXP Mathematical Built-In Function

Definition: EXP raises e (the base of the natural logarithm system) to a given power and returns the result to the point of invocation.

Reference: EXP (X)

Argument: The argument, X, specifies the power to which e is to be raised.

Result: The value returned by this function is e raised to the power of X.

LOG Mathematical Built-In Function

Definition: LOG finds the natural logarithm (i.e., base e of a given value) and returns it to the point of invocation.

Reference: LOG (X)

Argument: The argument, X, is the value whose natural logarithm is to be found. X must not be less than or equal to 0.

Result: The value returned by this function is the natural logarithm of X.

LOG10 Mathematical Built-In Function

Definition: LOG10 finds the common logarithm (i.e., base 10) of a given real value and returns it to the point of invocation.

Reference: LOG10 (X)

Argument: The argument, X, represents the value whose common logarithm is to be found; this value must be real and it must not be less than or equal to 0.

Result: The value returned by this function is the common logarithm of X.

LOG2 Mathematical Built-In Function

Definition: LOG2 finds the binary (i.e., base 2) logarithm of a given real value and returns it to the point of invocation.

Reference: LOG2 (X)

Argument: The argument, X, is the value whose binary logarithm is to be found; it must be real and it must not be less than or equal to 0.

Result: The value returned to this function is the binary logarithm of X.

RANDOM Mathematical Built-In Function

Definition: RANDOM returns a pseudo-random value between 0 and 1 to the point of invocation.

Reference: RANDOM (X)

Argument: The argument, X, has a controlling effect on the value returned by the function. If RANDOM is to be called repeatedly in a program, the initial call may be made with any value of X. For subsequent calls, X should be a variable as each call returns a random value between 1 and 2**39-1, and X should be used for no other purpose.

Result: The value returned by this function is a real value between 0 and 1 from a pseudo-random uniform distribution.

SIN Mathematical Built-In Function

Definition: SIN finds the sine of a given value, which is expressed in radians, and returns it to the point of invocation.

Reference: SIN (X)

Argument: The argument, X, is the value whose sine is to be found; it must be expressed in radians.

Result: The value returned by this function is the sine of X.

SIND Mathematical Built-In Function

Definition: SIND finds the sine of a given real value, which is expressed in degrees, and returns the result to the point of invocation.

Reference: SIND (X)

Argument: The argument, X, is the value whose sine is to be found; X must be real and it must be expressed in degrees.

Result: The value returned by this function is the sine of X.

SINH Mathematical Built-In Function

Definition: SINH finds the hyperbolic sine of a given value and returns the result to the point of invocation.

Reference: SINH (X)

Argument: The argument, X, is the value whose hyperbolic sine is to be found.

Result: The value returned by this function is the hyperbolic sine of X.

SQRT Mathematical Built-In Function

Definition: SQRT finds the square root of a given value and returns it to the point of invocation.

Reference: SQRT (X)

Argument: The argument, X, is the value whose square root is to be found. If X is real, it must not be less than 0.

Result: If X is real, the value returned by this function is the positive square root of X.

TAN Mathematical Built-In Function

Definition: TAN finds the tangent of a given value, which is expressed in radians, and returns it to the point of invocation.

Reference: TAN (X)

Argument: The argument, X, represents the value whose tangent is to be found; X must be expressed in radians.

Result: The value returned by this function is the tangent of X.

TAND Mathematical Built-In Function

Definition: TAND finds the tangent of a given real value which is expressed in degrees, and returns the result to the point of invocation.

Reference: TAND (X)

Argument: The argument, X, represents the value whose tangent is to be found; X must be expressed in degrees.

Result: The value returned by this function is the tangent of X.

TANH Mathematical Built-In Function

Definition: TANH finds the hyperbolic tangent of a given value and returns the result to the point of invocation.

Reference: TANH (X)

Argument: The argument, X, represents the value whose hyperbolic tangent is to be found.

Result: The value returned by this function is the hyperbolic tangent of X.

SUMMARY OF MATHEMATICAL FUNCTIONS

The following table summarizes the mathematical built-in functions. In using it, the reader should be aware of the following:

1. The value returned by each function is always in floating-point.
2. The error conditions are those defined by the PL/I language.
3. All arguments must be coded arithmetic and floating-point scale, or such that they can be converted to coded arithmetic and floating-point.

FUNCTION REFERENCE	VALUE RETURNED	ERROR CONDITIONS
ATAN(x)	arctan(x) in radians -(pi/2)<ATAN(x)<pi/2	-
ATAN(x,y)	see function description	error if x=y=0
ATAND(x)	arctan(x) in degrees -90<ATAND(x)<90	-
ATAND(x,y)	see function description	error if x=y=0
COS(x) x in radians	cosine(x)	-
COSD(x) x in degrees	cosine(x)	-
COSH(x)	cosh(x)	-
ERF(x)	2/SQRT(PI) * INTEGRAL $\int_0^x (\exp(-t^2)\,dt)$	-
ERFC(x)	1 - ERF(x)	-
EXP(x)	e^x	-
LOG(x)	log(x)	x<=0
LOG10(x)	$\log_{10}(x)$	x<=0
LOG2(x)	$\log_2(x)$	x<=0

ARRAY MANIPULATION BUILT-IN FUNCTIONS

The built-in functions described here may be used for the manipulations of arrays. All of these functions require array arguments (which may be expressions) and return single element values. Note that since these functions return element values, a function reference to any of them is considered an element expression.

ALL Array Manipulation Function

Definition: ALL tests all bits of a given bit-string array and returns the result, in the form of an element bit-string, to the point of invocation. The element bit-string indicates whether or not the corresponding bits of given array elements are all ones.

Reference: ALL (X)

Argument: The argument, X, is an array of bit strings. If the elements are not bit strings, they are converted to bit strings.

Result: The value returned by this function is a bit string whose length is equal to the length of the longest element in X and whose bit values are determined by the following rule:

> If the Ith bits of all of the elements in X exist and are 1, then the Ith bit of the result is 1; otherwise, the Ith bit of the result is 0.

ANY Array Manipulation Function

Definition: ANY tests the bits of a given bit-string array and returns the result, in the form of an element bit-string, to the point of invocation. The element bit-string indicates whether or not at least one of the corresponding bit of the given array elements is set to 1.

Reference: ANY (X)

Argument: The argument, X, is an array of bit strings. If the elements are not bit strings they are converted to bit strings.

Result: The value returned by this function is a bit string whose length is equal to the length of the longest element in X and whose bit values are determined by the following rule:

> If the Ith bit of any element in X exists and is 1, then the Ith bit of the result is 1; otherwise, the Ith bit of the result is 0.

DIM Array Manipulation Function

Definition: DIM finds the current extent for a specified dimension of a given array and returns it to the point of invocation.

Reference: DIM (X,N)

Arguments: The argument X is the array to be investigated; N is the dimension of X, the extent of which is to be found. If N is not a binary integer, it is converted to a binary integer of default precision. It is an error if X has less than N dimensions. If N is less than or equal to 0, or if X is not currently allocated, N is the dimension number, counting from left to right.

Result: The value returned by this function is a binary integer of precision (38,0), giving the current extent of the Nth dimension of X.

HBOUND Array Manipulation Function

Definition: HBOUND finds the current upper bound for a specified dimension of a given array and returns it to the point of invocation.

Reference: HBOUND (X,N)

Arguments: The argument X is the array to be investigated; N is the dimension of X for which the upper bound is to be found. If N is not a binary integer, it is converted to a binary integer of default precision. It is an error if X has less than N dimensions. If N is less than or equal to 0, or if X is not currently allocated. N is the dimension number, counting from left to right.

Result: The value returned by this function is a binary integer of default precision giving the current upper bound for the Nth dimension of X.

LBOUND Array Manipulation Function

Definition: LBOUND finds the current lower bound for a specified dimension of a given array and returns it to the point of invocation.

Reference: LBOUND (X,N)

Arguments: The argument X is the array to be investigated; N is the dimension of X for which the lower bound is to be found. If N is not a binary integer, it is converted to a binary integer of default precision. It is an error if X has less than N dimensions. If N is less than or equal to 0, or if X is not currently allocated. N is the dimension number, counting from left to right.

Result: The value returned by this function is a binary integer of default precision giving the current lower bound of the Nth dimension of X.

PROD Array Manipulation Function

Definition: PROD finds the product of all of the elements of a given array and returns that product to the point of invocation.

Reference: PROD (X)

Argument: The argument, X, should be an array of coded arithmetic floating-point elements. If it is not, each element is converted to coded arithmetic and floating-point before being multiplied with the previous product.

Result: The value returned by this function is the product of all of the elements in X. The scale of the result is floating-point, while the BASE, MODE, and PRECISION are those of the converted elements.

SUM Array Manipulation Function

Definition: SUM finds the sum of all of the elements of a given array and returns that sum to the point of invocation.

Reference: SUM (X)

Argument: The argument, X, should be an array of coded arithmetic floating-point elements. If it is not, each element is converted to coded arithmetic and floating-point before being summed with the previous total.

Result: The value returned by this function is the sum of all of the elements in X. The scale of the result is floating-point, while the BASE, MODE, and PRECISION are those of the converted elements of the argument.

CONDITION BUILT-IN FUNCTIONS

The condition built-in functions allow the PL/I programmer to investigate interrupts that arise from enabled conditions, none of these functions requires arguments. Each condition built-in function returns the value described only when executed in an on-unit (or a block activated directly or indirectly by an on-unit) that is entered as a result of an interrupt caused by one of the conditions for which the function can be used. Such an on-unit can be one specific to the condition, or it can be for the "ERROR" condition when these conditions are raised as standard system action. If a condition built-in function is used out of context, the value returned is as described for each function.

The on-units in which each function can be used are given in the function definition.

DATAFIELD Condition Built-In Function

Definition: Whenever the name-condition is raised, DATAFIELD may be used to extract the contents of the data field that caused the condition to be raised. It can be used only in an on-unit for the NAME condition.

Reference: DATAFIELD

Result: The value returned by this function is a varying-length character string giving the contents of the data field that caused the NAME condition to be raised. If DATAFIELD is out of context, a null string is returned.

ONCHAR Condition Built-In Function

Definition: Whenever the CONVERSION condition is raised, ONCHAR may be used to extract the character that caused that condition to be raised. It can be used only in an on-unit for the CONVERSION condition or in an on-unit for an ERROR condition raised as standard system action for CONVERSION condition. (ONCHAR can also be used as a pseudo-variable.)

Reference: ONCHAR

Result: The value returned by this function is a character string of length 1, containing the character that caused the CONVERSION condition to be raised. This character can be modified in the on-unit by the use of the ONCHAR or ONSOURCE pseudo-variables. If ONCHAR is used out of context, a blank is returned.

277

ONCODE Condition Built-In Function

Definition: ONCODE can be used in any on-unit to determine the type of interrupt that caused the on-unit to become active.

Reference: ONCODE

Result: ONCODE returns a binary integer of PRECISION (38,0). This CODE defines the type of interrupt that caused the entry into the currently active on-unit. If ONCODE is used out of context, a specified binary integer of default precision is returned.

ONCOUNT Condition Built-In Function

Definition: ONCOUNT can be used in any on-unit entered due to the abnormal completion of an input/output event to determine the number of interrupts (including the current one) that remain to be handled when a multiple interrupt has resulted from that abnormal completion.

Reference: ONCOUNT

Result: ONCOUNT returns a binary value of default precision. If ONCOUNT is used in an on-unit entered as part of a multiple interrupt, this value specifies the corresponding number of equivalent single interrupts (including the current one) that remain to be handled; if ONCOUNT is used in any other case, the returned value is zero.

ONFILE Condition Built-In Function

Definition: ONFILE determines the name of the file for which an input/output or CONVERSION condition was raised and returns that name to the point of invocation. This function can be used in the on-unit for any input/output or CONVERSION condition; it also can be used in an on-unit for an ERROR condition raised as standard system action for an input/output or conversion condition.

Reference: ONFILE

Result: The value returned by this function is a varying-length character string consisting of the name of the file for which an input/output or CONVERSION condition was raised. If that condition is not associated with a file, the returned value is the null string.

ONKEY Condition Built-In Function

Definition: ONKEY extracts the value of the key for the record that caused an input/output condition to be raised. It also extracts the key of a record in which a CONVERSION condition occurred during assignment specified by a KEYTO option. This function can be used in the on-unit for an input/output condition or a CONVERSION condition; it can also be used in an on-unit for an ERROR condition raised as standard system action for one of the above conditions.

Reference: ONKEY

Result: The value returned by this function is a varying-length character string giving the value of the key for the record that caused an input/output or CONVERSION condition to be raised. If the interrupt is not associated with a keyed record, the returned value is the null string.

ONLOC Condition Built-In Function

Definition: Whenever a condition is raised, ONLOC may be used in the on-unit for that condition to determine the entry point to the procedure in which that condition was raised. ONLOC may be used in any on-unit.

Reference: ONLOC

Result: The value returned by this function is a varying-length character string giving the name of the entry point to the procedure in which the condition was raised. If ONLOC is used out of context, a null string is returned.

ONSOURCE Condition Built-In Function

Definition: Whenever the CONVERSION condition is raised, ONSOURCE may be used to extract the contents of the field that was being processed when the condition was raised. This function can be used in the on-unit for a CONVERSION condition or in an on-unit for an ERROR condition raised as standard system action for a CONVERSION condition. (ONSOURCE can also be used as a pseudo-variable.)

Reference: ONSOURCE

Result: The value returned by this function is a varying-length character string giving the contents of the field being processed when CONVERSION was raised. This string may be modified in the on-unit by use of the ONCHAR or ONSOURCE pseudo-variable. If ONSOURCE is used out of context, a null string is returned.

BASED STORAGE BUILT-IN FUNCTIONS

The based storage built-in functions generally return special values to program control variables concerned in the use of based storage and list processing.

ADDR Based Storage Function

Definition: ADDR finds the location at which a given variable has been allocated and returns a pointer value to the point of invocation. The pointer value identifies the location at which the variable has been allocated.

Reference: ADDR (X)

Argument: The argument, X, is the variable whose location is to be found. It can be any variable that represents an element, an array which is not interleaved, a structure, an area, an element of an array, a minor structure, or an element of a structure. It can be of any data type and storage class.

Result: ADDR returns a pointer value identifying the location at which X has been allocated. If X is a parameter, the returned value identifies the corresponding argument (dummy or otherwise). If X is an unallocated controlled variable, a null pointer value is returned.

NULL Based Storage Built-In Function

Definition: NULL returns a null pointer value (that is, a pointer value that cannot identify any allocation) so as to indicate that a pointer variable does not currently identify an allocation.

Reference: NULL

Arguments: None

Result: The value returned by this function is a null pointer value. This value cannot be converted to offset type.

NULLO Based Storage Built-In Function

Definition: NULLO returns a null offset value (that is, an offset value that cannot identify any relative location of a based variable allocation) so as to indicate that an offset variable does not currently identify an allocation.

Reference: NULLO

Arguments: None

Result: The value returned by this function is a null offset value. This value cannot be converted to pointer type.

OFFSET Based Storage Built-In Function

Definition: OFFSET returns an offset value relative to the beginning of a specified area.

Reference: OFFSET (P,A)

Arguments: The argument, P, is a scalar pointer expression; A is a scalar area expression that may be qualified and/or subscripted. The value of P must identify a generation in A.

Result: The value returned by the OFFSET built-in function is an offset value that identifies a generation in A, relative to the beginning of A.

POINTER Based Storage Built-In Function

Definition: POINTER returns a pointer value that identifies a generation in a specified area.

Reference: POINTER (O,A)

Arguments: The argument, O, is an offset expression; A is a scalar area expression that may be qualified and/or subscripted. The value of O must identify an equivalent generation in some area, but not necessarily in A.

Result: The value returned by the POINTER built-in function is a pointer value that identifies, in A, a generation equivalent to the generation originally identified by the offset O.

MISCELLANEOUS BUILT-IN FUNCTIONS

ALLOCATION Built-In Function

Definition: ALLOCATION determines whether or not storage has been allocated for a given controlled variable and returns an appropriate indication to the point of invocation.

Reference: ALLOCATION (X)

Argument: The argument, X, must be a level 01 unsubscripted controlled variable.

Result: The value returned by this function is defined as follows: if an allocation of X is known in the current task, the returned value is '1' B; if no allocation is known, the returned value is '0'B.

COMPILETIME Built-In Function

Definition: COMPILETIME produces a value resulting from an invocation of TIME at that point during the compilation.

Reference: COMPILETIME[()].

Argument: None

Result: The invocation of COMPILETIME is replaced at compile-time by the value of TIME.

COUNT Built-In Function

Definition: COUNT determines the number of data items that were transmitted during the last get or put-operation on a given file and returns the result to the point of invocation.

Reference: COUNT (<file-name>)

Argument: The argument, <file-name>, represents the file to be investigated. This file must have the STREAM attribute.

Result: The value returned by this function is a binary fixed-point integer of default precision specifying the number of element data items that were transmitted during the last get or put-operation on <file-name>. Note that if an on-unit or procedure is entered during a get or put-operation, and within that on-unit or procedure a GET or PUT is executed for the same file. The value of COUNT is reset for the new operation and is not restored when the original GET or PUT is continued.

DATE Built-In Function

Definition: DATE returns the current date to the point of invocation.

Reference: DATE [()]

Arguments: None

Result: The value returned by this function is a character string of length six digits, in the form YYMMDD, where:

YY is the current year
MM is the current month
DD is the current day

IOTIME Built-In Function

Definition: IOTIME returns an integer value indicating the elapsed I/O time of the program.

Reference: IOTIME [()]

Argument: None.

Result: The value returned by this function is an unsigned integer constant which is the elapsed I/O time of the program in units of sixtieths of a second.

LINENO Built-In Function

Definition: LINENO finds the current line number for a file having the PRINT attribute and returns that number to the point of invocation.

Reference: LINENO (<file-name>)

Argument: The argument, <file-name>, must be the name of a file having the PRINT attribute.

Result: The value returned by this function is a binary fixed-point integer of default precision specifying the current line number of <file-name>.

PROCTIME Built-In Function

Definition: PROCTIME returns an integer value indicating the elapsed processing time of the program.

Reference: PROCTIME [()]

Argument: None.

Result: The value returned by this function is an unsigned integer constant which is the elapsed processing time of the program in units of sixtieths of a second.

TIME Built-In Function

Definition: TIME returns the current time to the point of invocation.

Reference: TIME [()]

Arguments: None.

Result: The value returned by this function is a character string of length nine, in the form HHMMSSTTT, where:

HH is the current hour of the day
MM is the number of minutes
SS is the number of seconds
TTT is the number of milliseconds.

PSEUDO-VARIABLES

In general, pseudo-variables are certain built-in functions that can appear wherever other variables can appear in order to receive values. They are built-in functions used as receiving fields. A pseudo-variable may appear to the left of the equal sign in an assignment or DO statement; it may appear in the data list of a GET statement; it may appear as the string name in the KEYTO, STRING, and REPLY options.

Since all pseudo-variables have built-in function counterparts, only a short description of each pseudo-variable is given here; the discussion of the corresponding built-in function should be consulted for details. Pseudo-variables cannot be nested.

Example

The following statement is invalid:

```
UNSPEC(SUBSTR(A,1,2)) = 00 B;
```

ONCHAR Pseudo-Variable

Reference: ONCHAR

Description ONCHAR can be used in the on-unit for a CONVERSION condition or in the on-unit for an ERROR condition raised as standard system action for a CONVERSION condition; it can also be used in a block directly or indirectly activated by such an on-unit. If ONCHAR is used in some other context, it is an error.

The expression being assigned to ONCHAR is evaluated, converted to a character string length of 1, and assigned to the character that caused the error. The new character will displace the current value of the ONCHAR built-in function, and will be used when the conversion is reattempted, upon the resumption of execution at the point of interrupt.

ONSOURCE Pseudo-Variable

Reference: ONSOURCE

Description ONSOURCE can be used in the on-unit for a CONVERSION condition, or in an on-unit for an ERROR condition raised as standard system action for a CONVERSION condition; it can also be used in a block directly or indirectly activated by such an on-unit. If ONSOURCE is used in some other context, it is an error.

The expression being assigned to ONSOURCE is evaluated, converted to a character string, and assigned to the string that caused the CONVERSION condition to be raised. The string will be padded with blanks, if necessary, to match the length of the string that caused the error. This new string displaces the current value of the ONSOURCE built-in function and will be used when the conversion is reattempted, upon the resumption of execution at the point of interrupt.

SUBSTR Pseudo-Variable

Reference: SUBSTR (<string>,I[,J])

Description The value being assigned to SUBSTR is assigned the substring of the character or bit-string variable <string>, as defined for the built-in function SUBSTR. If <string> is an aggregate, I and/or J may be aggregates, the remainder of <string> remains unchanged.

UNSPEC Pseudo-Variable

Reference: UNSPEC (V)

Description The letter V represents an element or aggregate variable of ARITHMETIC, STRING, LOCATOR, or AREA type. The value being assigned to UNSPEC is evaluated and converted to a bit string the length of which is a function of the characteristics of V, and then assigned to V. See the UNSPEC built-in function. If V is a string of varying length, its length after the assignment will be the same as that of the bit string assigned to it.

APPENDIX 2. CONDITIONS

The conditions are those conditions that may be specified in the ON statement. These conditions are also specified in SIGNAL and REVERT statements.

For each condition name, the description in this appendix includes the circumstances under which the condition occurs, the standard system action that would be taken in the absence of programmer-specified action, and, where applicable, the result.

For conditions OVERFLOW, UNDERFLOW, ZERODIVIDE, CONVERSION, or FIXEDOVERFLOW, an interrupt action will always take place on occurrence of the condition, unless the occurrence is in a calculation lying within the scope of a prefix specifying NOOVERFLOW, NOUNDERFLOW, NOZERODIVIDE, NOCONVERSION, or NOFIXEDOVERFLOW. For the conditions SIZE, STRINGRANGE, SUBSCRIPTNAME, or CHECK <identifier list>, an interrupt will not take place on occurrence of the condition unless the occurrence is in a calculation lying within the scope of a prefix specifying the condition. (See Section 10.5.2, Scope of the Condition Prefix.)

For any other condition whose name may not be used in a prefix, an interrupt always will result from the occurrence of the condition.

A multiple interrupt can occur only for an input/output operation that has been associated with an event variable, and is described in the section of this appendix dealing with input/output conditions.

CLASSIFICATION OF CONDITIONS

The conditions are classified as follows: computational, input/output, program-checkout, list processing, programmer-named, and system-action conditions.

The computational conditions are associated with data handling, expression evaluation, and computation.

The input/output conditions are associated with data transmission.

The program-checkout conditions facilitate debugging of programs.

The list processing conditions are associated with area usage.

The programmer named conditions permit the programmer to use conditions of his own naming. These conditions are raised only by a SIGNAL statement.

The system-action conditions provide facilities to the programmer to extend the standard system action taken after the occurrence of a condition or at the completion of a program.

CONVERSION:

This condition is raised whenever an illegal conversion is attempted on character string data, either internally or during input or output. The condition will be raised for such errors as characters other than 0 or 1 in conversion to bit string, characters not permitted in conversion to numeric field, or illegal characters in conversion to arithmetic. The conversion is carried out character-by-character. The condition is raised for each illegal conversion.

This condition may also be raised when the number of digits in a floating-point exponent exceeds the number allowed. On return from the on-unit for this condition, the conversion will be reattempted.

Expressions that are not variables, or variables that are procedure entry parameters, based, or overlay-defined, may not be reevaluated after an on-unit. Therefore, changing the value of such an expression in an on-unit may not affect the conversion retry.

Result: When CONVERSION occurs, results of the entire result field are undefined.

Standard System Action: Comment and raise the ERROR condition.

FIXEDOVERFLOW:

This condition occurs during fixed-point arithmetic operations if the results of these operations exceed the integer capacity of the hardware. (See SIZE below for a related condition that checks for overflow related to declared attributes of data items.)

Result: Result of the invalid fixed-point operation is undefined.

Standard System Action: Comment and raise the ERROR condition.

OVERFLOW:

This condition occurs when the exponent of a floating-point number exceeds the permitted maximum.

Result: The value of such an invalid floating-point number is undefined.

Standard System Action: Comment and raise the ERROR condition.

SIZE:

This condition is raised by conversions between data types, or between differing BASES, SCALES, or PRECISIONS. The condition arises if the high order bits or digits are lost when a value is assigned to an arithmetic data item, or when a value is converted during expression evaluation or during input/output.

The SIZE condition should be distinguished from FIXEDOVERFLOW that occurs during arithmetic calculations. A value too large for the field to which it is assigned will raise a SIZE condition on assignment, regardless of whether there was a FIXEDOVERFLOW in the calculation of the value.

Result: The contents of the receiving field are undefined.

Standard System Action: Comment and raise the ERROR condition.

UNDERFLOW:

This condition occurs when the exponent of a floating-point number is smaller than the permitted minimum.

The condition does not occur when equal numbers are subtracted (often called significance error).

Result: The value of the floating-point number is set to zero.

Standard System Action: Comment and continue execution.

ZERODIVIDE:

This condition occurs on an attempt to divide by zero. The condition does not distinguish between fixed-point and floating-point division.

Result: The result of division by zero is undefined.

Standard System Action: Comment and raise the ERROR condition.

INPUT/OUTPUT CONDITIONS

The following conditions are always enabled and cannot appear in prefix lists. If the same file is known in a program by more than one name (for example, a file parameter and its associated file argument), these names constitute a set of equivalent file names. A condition specified for one file name applies to all names of the set. An on-unit established using one file name of the set can be overridden by specifying an on-unit for another file name of the set.

ENDFILE(<filename>):

This condition may be raised during any GET or READ statement, and indicates that there is no more data on the file.

The end-of-file status remains until the file is closed. Subsequent GET or READ statements will immediately raise the condition. On return from the on-unit, processing will continue at the next statement.

Standard System Action: Comment and raise the ERROR condition.

ENDPAGE(<filename>):

This condition is raised by a PUT statement when an attempt is made to start a new line beyond the limit specified for the current page by the PAGESIZE option in an OPEN statement. This attempt may be made during data transmission (with associated format items, if edit-directed transmission), by the LINE option, or by the SKIP option. It is raised only once per page.

If this condition is raised during data transmission, then, on return from the on-unit, the data is written on the current line, which may have been changed by the on-unit. If it is raised by a LINE or SKIP option, then, on return from the on-unit, the action specified by LINE or SKIP is ignored.

When ENDPAGE is raised, the current line number is one greater than that specified by the PAGESIZE option. During the execution of the on-unit for this condition, or after return from the on-unit without a PAGE option or PAGE format item having been specified, the line number

may increase indefinitely. However, execution of a LINE option or a LINE format item specifying a line number less than or equal to the current line number will cause a result equivalent to that caused by the execution of a PAGE option. In this case, ENDPAGE will not be raised; however, since the current line is now one, ENDPAGE can be raised again.

Standard System Action: Start a new page.

TRANSMIT(<filename>):

This condition may be raised during any input/output operation, and is caused by an irrecoverable transmission error on the specified file. In STREAM input, it is raised after assignment to each data item on record which is potentially of incorrect value because of the transmission error. On return from the on-unit, processing will continue as if no error has occurred.

Standard System Action: Comment and raise the ERROR condition.

UNDEFINEDFILE(<filename>):

This condition is raised whenever an attempt to open a file is unsuccessful. If the attempt is made through an OPEN statement, attempts to open all other files in that statement will be made before this condition is raised. If this condition is raised for more than one file in the same OPEN statement, on-units will be executed according to the order of appearance (taken from left to right) of the file names in that OPEN statement. On return from the final on-unit, processing will continue with the next statement.

If this condition is raised by an implicit file opening in an input/output statement , then upon normal return from the on-unit, processing continues with the remainder of the interrupted input/output statement. If the file was not opened in the on-unit, then the statement cannot be continued and the ERROR condition is raised.

Standard System Action: Comment and raise the ERROR condition.

NAME(<filename>):

This condition may be raised on data-directed GET statements. It is caused by an unrecognizable identifier in the input or by an identifier not in the associated data list. The condition is raised at the time the error occurs. On return from the on-unit, the execution of the GET statement is resumed with the next data field in the stream.

By using the DATAFIELD built-in function in the on-unit, the programmer may access the data field which contained the incorrect name.

Standard System Action: Ignore the field and comment.

KEY(<filename>):

This condition may be raised by any keyed and record operation. It is raised in the following cases:

1. A read for which the key is not found.

2. A write or locate for which the key already exists.

3. A rewrite for which the key is not found.

4. A delete for which the key is not found.

5. Specification of the character string representing the key is in conflict with the format prescribed for PL/I.

   On return from the on-unit, no further action is attempted, and control passes to the next statement.

Standard System Action: Comment and raise the ERROR condition.

RECORD(<filename>):

This condition may be raised by any READ or REWRITE operation. It is raised when the record contains more or less data than the specified variable (i.e., the size of the variable differs from the actual record size). It may be raised on a write when the statement cannot be executed.

The ONCODE built-in function returns an indication of whether the record variable was less than or greater then the record in size.

Before the on-unit is invoked, the following action takes place:

1. If the variable cannot contain the record, the excess data of the record is lost.

2. If the variable is greater than the record in size, the excess data in the variable is not transmitted on output and is unaltered on input.

Standard System Action: Comment and raise the ERROR condition.

PROGRAM CHECKOUT CONDITIONS

SUBSCRIPTRANGE:

This condition occurs when a subscript is evaluated and found to lie outside its specified bounds.

The condition does not distinguish between values that are too large and values that are too small.

Note that if more than one subscript is associated with an identifier, e.g., A(I,J,K), the occurrence of a SUBSCRIPTRANGE condition is signalled after each subscript has been checked.

Result: Undefined.

Standard System Action: Comment and raise the ERROR condition.

STRINGRANGE:

This condition may be raised by any reference to the SUBSTR built-in function or pseudo-variable if the length specified for the substring is less than zero or if the substring does not lie entirely within (or correspond to) the specified string.

The condition can be raised only once for each scalar SUBSTR reference.

Result: On normal return from an on-unit, execution continues with a revised SUBSTR reference whose value is defined as follows:

Let K be the length of the first argument (after execution of the on-unit); the other two arguments are represented by I and J.

If I is greater than K, the value is the null string.

If I is less than or equal to K, the value is that substring beginning with the Mth character or bit of the first argument, and extending N characters or bits, where M and N are defined by:

M = MAX(I,1).

N = MAX(0,MIN(J+MIN(I,1)-1,
K-M+1)) if J is specified.

N = K-M+1 if J is not specified.

The values of I and J are established before entry to the on-unit; they are not reevaluated on return from the on-unit.

Standard System Action: Revise the SUBSTR reference, as described under RESULT, comment, and continue.

This condition is always enabled and may not appear in a condition prefix.

AREA:

This condition is raised when an attempt is made to allocate storage within an area defined by an area variable, and sufficient storage does not remain within the area. It can be raised by an ALLOCATE or assignment statement.

Result: On normal return from the on-unit, the reference to the area is reevaluated. Allocation is then attempted in this area.

Standard System Action: The ERROR condition is raised.

PROGRAMMER NAMED CONDITIONS

CONDITION(<identifier>):

This condition is always enabled and may not appear in a condition prefix. The identifier is specified by the programmer, and is EXTERNAL. The condition is raised by the execution of a SIGNAL statement having the same identifier.

Standard System Action: Comment and continue.

SYSTEM ACTION CONDITIONS

The following conditions are always enabled and cannot appear in a condition prefix.

FINISH:

This condition is raised by execution of a statement that would cause termination of the major task of a PL/I program, that is, a STOP in any task, an EXIT in the major task. The condition is also raised by a SIGNAL FINISH statement in any task. The interrupt occurs in the task in which the statement is executed, and any on-unit specified for the condition is executed as past of the task.

An abnormal return from a finish on-unit will avoid any subsequent task termination processes and permit the interrupted task to continue.

On normal return from a finish on-unit, execution of the interrupted statement is resumed.

Standard System Action: Execution of the interrupted statement is resumed.

ERROR:

This condition is raised either by a SIGNAL ERROR or by some error situation in the execution of the program; it is raised by the standard system action for certain other conditions. As described in this appendix, an abnormal return from an error on-unit will permit the interrupted task to continue execution.

Standard System Action: Comment and terminate the task.

ONCODES FOR CONDITIONS

The ONCODE built-in function may be used by the programmer within an on-unit to determine the nature of his error when a condition is raised. The values of the oncodes are:

SIZE

501	Fatal size condition - single precision exceeded.

IMPLEMENT

601	Exclusive file options are not implemented.
602	This record I/O option is not implemented.
603	Complex operations are not implemented.
604	Unimplemented conversion.
605	Temporary storage area exceeded due to conditions.

MATH

701	Math intrinsic computational error.

ENDFILE

801	End-of-file occurred while processing GET statement.
802	End-of-file occurred while processing PUT statement.
803	End-of-file occurred on a PUT LIST I/O.
804	End-of-file occurred on a GET LIST I/O.
805	End-of-file occurred on a PUT DATA I/O.
806	End-of-file occurred on a keyed file.
807	End-of-file occurred while processing GET STRING statement.
808	End-of-file occurred while processing PUT STRING statement.
809	End-of-file occurred on a KEYED file I/O. Check AREAS and AREASIZE.

TRANSMIT

1301	Undefined I/O result word error on output.
1302	Undefined I/O result word error on input.
1303	Result descriptor error on KEYED I/O.
1304	Result descriptor error on KEYED READ.
1305	Result descriptor error on KEYED WRITE.
1306	Result descriptor error on KEYED TABLE READ.
1307	Result descriptor error on KEYED TABLE WRITE.
1308	Data error occurred on I/O.
1309	Parity error occurred on I/O.
1310	Security error occurred on I/O.

NAME

1401	Unrecognized identifier in the input or an identifier not in the associated input list on a GET DATA statement.

RECORD

1501	Physical record size differs from file record size specification on reocrd I/O into or from fixed string.
1502	File record size exceeds length of varying string on record read into varying string.
1503	Length of varying string exceeds file record size on record write from varying string.
1504	Zero length record has been read.
1505	FROM variable length longer than record size.
1506	INTO variable shorter than record size.

SIZE

1601	Error in processing "F" format item for a GET EDIT statement. Number of decimal digits specified is greater than the format field width.
1602	Error in processing "E" format item for a GET EDIT statement. The width of the number is greater than format field width.
1603	Error in processing "F" format item for a PUT EDIT statement. More decimal digits are specified than significant digits.
1604	Implied "S" missing in an "E" format output statement.
1605	Exponent field error in "E" format output statement.
1606	Size error occurred during a string to arithmetic conversion.
1607	Exponent exceeds implementation maximum.

	1608	Bit string truncated in string to arithmetic conversion.
	1609	"E" format field width too small, E(N+9,N).
AREA		
	1701	Allocate error (allocate of negative size requested).
	1702	Free error (chunk is already free).
	1703	Invalid allocation intrinsic parameter.
	1704	Allocate error (area is exhausted).
	1705	Attempted to free a non-freeable area.
	1706	Allocation size error on unit-size area.
	1707	Free error on unit-size area.
	1708	Free error (invalid free stacked).
	1709	Allocate error (invalid area header).
	1710	Free error (invalid area header).
	1711	Free error (invalid link word).
STRINGSIZE		
	1801	Source string is longer than target string in bit string to character string conversion.
	1802	Source string is longer than target string in character string to bit string conversion.
	1803	Source string is longer than target string in bit string to bit string conversion.
	1804	Source string is longer than target string in character string to character string conversion.
CONVERSION		
	2101	Conversion error on "E" format item input.
	2102	Illegal binary character in string to string conversion.
	2103	Illegal character in string to arithmetic conversion.
	2104	Illegal binary character in string to arithmetic conversion.
	2105	Illegal imaginary string in string to arithmetic conversion.
	2106	Illegal imaginary bit string in arithmetic conversion.
	2107	Invalid character for numeric picture editing.
	2108	Invalid character for character picture editing.
	2109	Invalid character for corresponding picture character.
	2110	Error in string to arithmetic conversion.
	2111	Missing mantissa in string to arithmetic conversion.
	2112	Missing exponent following "E" in floating point number.
	2113	Missing "I" following complex number.
KEY		
	2401	No record was found with this key.
	2402	No space is available for additional keyed records.
	2403	Attempted to write a duplicate keyed record with the NODUPLICATE option set.
	2404	A duplicate keyed record was added.
	2405	A difference in key exists on a REWRITE statement.
	2406	A key error exists on a CREATE write. For example, the keys are out of order.
	2407	Table overflow on a KEYED CREATE WRITE. Check areasize.
	2408	A REWRITE was attempted before a READ was executed.
	2409	Invalid usage of KEYED FILE or KEYED OPTION.
	2410	Write KEYFROM key and record key do not match.

2411	Record contains '11111111' in first byte.
2412	KEYTO variable shorter than key in file.
2413	Key or KEYFROM longer than key in file.

UNDEFINEDFILE

2501	Attribute conflict on this file.
2502	Illegal file access method.
2503	Attempt to open a non-resident file.
2504	Attempted to close an unopened file.
2505	Parameter error on KEYED FILE OPEN.
2506	KEYED FILE opened incorrectly.
2507	Error occurred while opening a KEYED file.
2508	Cannot handle a bcl file.
2509	Attempted to open a previosly opened file.
2510	Attempted a keyed I/O on a non-keyed file.
2511	Keyed file not declared direct.
2512	This I/O action illegal for keyed files.
2513	Open attempted on a non-present file.
2514	Rewrite attempted on variable length records.
2517	Cannot create empty keyed file.
2518	Bad subfile index.
2519	Read or write to subfile failed.

ERROR

2601	An attempt was made to access an unititialized variable.
2602	Unimplemented or unsupported format phrase.
2603	Format error, missing left parenthesis.
2604	SQRT error.
2605	Picture conversion error was not corrected.
2606	Double SQRT error.
2607	String to arithmetic conversion error was not corrected.
2608	String to string conversion error was not corrected.
2609	Arithmetic overflow occurred during conversion.
2610	Arithmetic overflow occurred in arithmetic to character conversion.
2611	No data format item specified.
2612	Explicit field width not specified in format.
2613	Invalid LNGAMMA argument.
2614	Invalid LOG argument.
2615	Invalid ARCCOS argument.
2616	Invalid ARCSIN argument.
2617	Invalid SINH OR COSH argument.
2618	Invalid GAMMA argument.
2619	Invalid SQUARE ROOT argument.
2620	Invalid list item in GET STRING data.
2621	End of input in GET STRING statement.
2622	End of output in PUT STRING statement.
2623	Illegal format phrase with string edit.
2624	Invalid format for "R" format phrase.

DMEXCEPTION

2701	Data management error.

APPENDIX 3. KEYWORD ABBREVIATIONS

Abbreviations are provided for certain keywords. The abbreviations themselves are keywords and will be recognized as synonymous in every respect with the full keywords. The abbreviated keywords are shown to the right of the full keywords in the following list.

AUTOMATIC	AUTO
BINARY	BIN
CHARACTER	CHAR
COLUMN	COL
CONTROLLED	CTL
CONVERSION	CONV
DEACTIVATE	DEACT
DECIMAL	DEC
DECLARE	DCL
DEFAULT	DFT
DEFINED	DEF
DESCRIPTOR	DESC
DIMENSION	DIM
ENVIRONMENT	ENV
EXCLUSIVE	EXCL
EXTERNAL	EXT
FIXEDOVERFLOW	FOFL
INITIAL	INIT
INTERNAL	INT
MEMBER	MEM
NONVARYING	NVAR
OVERFLOW	OFL
PARAMETER	PARAM
PICTURE	PIC
POINTER	PTR
POSITION	POS
PRECISION	PREC
PROCEDURE	PROC
SEQUENTIAL	SEQL
STRINGRANGE	STRG
STRUCTURE	STRUCT
SUBSCRIPTRANGE	SUBRG
UNALIGNED	UNAL
UNDERFLOW	UFL
UNDEFINEDFILE	UNDF
VARYING	VAR
ZERODIVIDE	ZDIV

APPENDIX 4. 48-CHARACTER SET

The 48-character set may be used instead of the 60-character set. The characters that make up the 48-character set are the same as those that make up the 60-character set except for certain restrictions.

The following characters are not included:

Percent	%
Colon	:
Not	¬
Or	|
And	&
Greater Than	>
Less Than	<
Semicolon	;
Number Sign	#
Commercial At Sign	@
Question Mark	?
Break Character	_

The following three characters are replaced as indicated:

60-Character Set	48-Character Set
:	..
;	,.
%	//

The two periods which replace the colon must be immediately preceded by a blank if the preceding character is a period. The two slashes that replace the percent symbol must be immediately preceded by a blank if the preceding character is an asterisk. The sequence "comma period" represents a semicolon when it occurs in a comment or character string, or when it is immediately followed by a digit.

The following character combinations, as used in the 60-character set, are replaced in the 48-character set by alphabetic equivalents as indicated:

60-Character Set	48-Character Set
>	GT
¬>	NG
>=	GE
¬=	NE
<=	LE
<	LT
¬<	NL
¬	NOT
|	OR
&	AND
||	CAT
->	PT

The preceding words are "reserved" in the 48-character set; that is, they must not be used as programmer-specified identifiers.

In each case, one or more blanks must immediately precede the alphabetic operator if the preceding character would otherwise be alphanumeric, and one or more blanks must immediately follow if the following character would otherwise be alphanumeric. Thus, to indicate

the comparison of the variables A6 and BQ2Y for inequality, one would write A6 NE BQZY, but not A6NEBQ2Y, A6NE BQ2Y, or A6 NEBQ2Y. As the equal symbol is usable, however, the comparison of these two variables for equality may be written A6 = BQ2Y.

The break character is not used and consequently may not be employed in identifiers.

APPENDIX 5. ERROR MESSAGES

Error messages are classified into ten levels of severity as follows:

0	information message which has no affect on the compilation or execution and by default are not printed.
1-3	warning messages in which inconsistent parts of a statement may be ignored.
4-6	warning messages in which an entire statement may be ignored.
7-9	error messages in which the results of the compilation are undefined.

APPENDIX 6. CONVERSION FROM IBM

In order to convert IBM PL/I programs to B 7000/B 6000 PL/I programs, it is necessary to understand the differences in the data mappings determined by the IBM PL/I compilers and the B 7000/B 6000 compiler. The differences noted in this document apply to the IBM PL/I checkout and optimizing compilers and the current B 7000/B 6000 PL/I compiler (MARK II.9).

1. TERMS

The IBM systems store data in terms of eight-bit bytes. An IBM "WORD" consists of four bytes (32 bits). A "HALF-WORD" consists of two bytes (16 bits), and a "DOUBLE WORD" consists of eight bytes (64 bits).

The B 7000/B 6000 systems store data in terms of words. A B 7000/B 6000 "WORD" consists of six bytes (48 bits). A double precision word consists of two words (96 bits). The notion of a "HALF-WORD" does not exist on the B 7000/B 6000 systems.

Because of these differences, the PL/I "ALIGNED" attribute has different results on the different systems, i.e., IBM systems as opposed to Burroughs systems. On an IBM system, "ALIGNED" means alignment to either a byte boundary,a half-word boundary, or a word boundary. On a B 7000/B 6000 system, "ALIGNED" means alignment to a word boundary.

2. DATA TYPE STORAGE MAPPING

The following tables detail the amount of storage allocated for the PL/I data types. Table 1 applies to structures and their members with the "ALIGNED" attribute as well as to non- structure and non-member variables. Table 2 applies to structure variables with the "UNALIGNED" attribute.

Table A6-1. Storage Allocation for Aligned Structures

DATA TYPE	INTERNAL STORAGE		ALIGNMENT	
	IBM	BURROUGHS	IBM	BURROUGHS
BIT (N)	ONE BYTE FOR EACH GROUP OF 8 BITS (OR PART THEREOF)	ONE WORD FOR EACH GROUP OF 48 BITS (OR PART THEREOF)	BYTE	WORD
CHARACTER (N)	ONE BYTE FOR EACH CHARACTER	ONE WORD FOR EACH GROUP OF 6 CHARACTERS (OR PART OF)	BYTE	WORD
BIT (N) VARYING	2-BYTE PREFIX PLUS 1 BYTE FOR EACH GROUP OF 8 BITS (OR PART THEREOF)	1-WORD PREFIX PLUS 1 WORD FOR EACH GROUP OF 48 BITS (OR PART THEREOF)	HALF-WORD	WORD
CHARACTER (N) VARYING	2-BYTE PREFIX PLUS 1 BYTE FOR EACH CHARACTER	1-WORD PREFIX PLUS 1 WORD FOR EACH GROUP OF 6 CHARACTERS (OR PART OF)	HALF-WORD	WORD
PICTURE (EXCEPT 'H' AND '1', B6700/B7700 ONLY)	ONE BYTE FOR EACH PICTURE CHARACTER (EXCEPT V, K, AND F SCALING SPEC)	ONE WORD FOR EACH GROUP OF 6 PICTURE CHARACTERS (OR PART THEREOF, EXCEPT B, V, AND Z SPECS)	BYTE	WORD
DECIMAL FIXED	PACKED DECIMAL FORM (4 BITS PER DIGIT, PLUS A 4 BIT SIGN)	ONE WORD SCALED INTEGER	BYTE	WORD
PICTURE 'H'	NOT APPLICABLE	PACKED DECIMAL FORM (4 BITS PER DIGIT, PLUS A 4 BIT SIGN)		WORD
BINARY FIXED (P,Q) P < = 15	HALF-WORD BINARY INTEGER	ONE WORD SEALED BINARY INTEGER	HALF-WORD	WORD
P > = 15	ONE WORD BINARY INTEGER	ONE WORD SCALED BINARY INTEGER	WORD	WORD
PICTURE '1'	NOT APPLICABLE	ONE WORD BINARY INTEGER		WORD
BINARY FLOAT AND DECIMAL FLOAT	ONE WORD FLOATING POINT	ONE WORD FLOATING POINT	WORD	WORD
POINTER	ONE WORD	ONE WORD	WORD	WORD
OFFSET	ONE WORD	ONE WORD	WORD	WORD
FILE	ONE WORD	2 WORDS	WORD	WORD
ENTRY	2 WORDS	ONE WORD	WORD	WORD
LABEL	2 WORDS	ONE WORD	WORD	WORD
AREA	4 WORDS + SIZE OF AREA BYTES	SIZE OF AREA WORDS	DOUBLE WORD	WORD

MV1735

Table A6-2. Storage Allocation for Unaligned Structures

DATA TYPE	INTERNAL STORAGE		ALIGNMENT	
	IBM	BURROUGHS	IBM	BURROUGHS
BIT (N)	N BITS	N BITS	BIT	BIT
CHARACTER (N)	ONE BYTE FOR EACH CHARACTER	ONE BYTE FOR EACH CHARACTER	BYTE	BYTE
BIT (N) VARYING	2-BYTE PREFIX PLUS N BITS	1-WORD PREFIX PLUS N BITS	BYTE	WORD
CHARACTER (N) VARYING	2-BYTE PREFIX PLUS 1 BYTE FOR EACH CHARACTER	1-WORD PREFIX PLUS 1 BYTE FOR EACH CHARACTER	BYTE	WORD
PICTURE (EXCEPT 'H' AND '1', B6700/B7700 ONLY)	ONE BYTE FOR EACH PICTURE CHARACTER (EXCEPT V, K, AND F SCALING SPEC)	ONE BYTE FOR EACH PICTURE CHARACTER (EXCEPT B, V, AND Z SPEC)	BYTE	BYTE
DECIMAL FIXED	PACKED DECIMAL FORM (4 BITS PER DIGIT, PLUS A 4 BIT SIGN)	ONE WORD SCALED INTEGER	BYTE	WORD
PICTURE 'H'	NOT APPLICABLE	PACKED DECIMAL FORM (4 BITS PER DIGIT, PLUS A 4 BIT SIGN)		BYTE
BINARY FIXED (P,Q) P < = 15	HALF-WORD BINARY	ONE WORD SEALED BINARY INTEGER	BYTE	WORD
P >15	ONE WORD BINARY INTEGER	ONE WORD SCALED BINARY INTEGER	BYTE	WORD
PICTURE '(P)1' P < = 15	NOT APPLICABLE	2-BYTE BINARY INTEGER		BYTE
P > 15		4-BYTE BINARY INTEGER		BYTE
BINARY FLOAT AND DECIMAL FLOAT	ONE WORD FLOATING POINT	ONE WORD FLOATING POINT	BYTE	WORD
POINTER	ONE WORD	ONE WORD	BYTE	WORD
OFFSET	ONE WORD	ONE WORD	BYTE	WORD
FILE	ONE WORD	2 WORDS	BYTE	WORD
ENTRY	2 WORDS	ONE WORD	BYTE	WORD
LABEL	2 WORDS	ONE WORD	BYTE	WORD
AREA	4 WORDS + SIZE OF AREA BYTES	SIZE OF AREA WORDS	DOUBLE WORD	WORD

MV1734

The numeric data types specified in the preceding tables apply only to single precision numbers. The following rules apply to double precision numbers on a B 7000/B 6000 system when OPTIONS(DOUBLE) is specified:

1. If the precision of a decimal fixed variable is >= 12, then the variable is allocated a double precision word and is aligned on a double precision word boundary.

2. If the precision of a binary fixed variable is > 38 then the variable is allocated a double precision word and is aligned on a double precision word boundary.

3. Any variables with the "FLOAT" attribute are always allocated a double precision word and are aligned on a double precision word boundary.

On an IBM system, the following rules apply:

1. If the precision, P, of a binary float variable is 21 < P < 54, then the variable is allocated a double word and is aligned on a double word boundary.

2. If the precision, P, of a decimal float variable is 6 < P < 17, the variable is also allocated a double word and is aligned on a double word boundary.

3. If 53 < P < 110 for a binary float variable, the variable is allocated 2 DOUBLE words and is aligned on a double word boundary.

4. If 16 < P < 34 for a decimal float variable, the variable is also allocated 2 DOUBLE words and is aligned on a double word boundary.

5. For "UNALIGNED" structures, alignment is carried out to a byte boundary rather than a double word boundary.

The storage allocation of pointer, offset, file, entry, label, and area data types has no meaning when converting from an IBM system to a B 7000/B 6000 system. They are included in the preceding tables for instructional purposes rather than conversion purposes.

3. REFERENCES

OS PL/I Checkout and Optimizing Compilers: Language Reference Manual, IBM Corp., No. SC33-0009-2, September, 1972.

APPENDIX 7. COMPILER CONTROL IMAGES

The manner in which the B 7000/B 6000 PL/I compiler handles source language input may be controlled by the user. This control information may be input to the PL/I compiler at any point within the primary or secondary input files using a Compiler Control Image (or CCI).

The CCI is a mechanism by which the user controls the options provided for him by a compiler. These options fall into one of five categories. These are for the control of

(1) source language input(s),
(2) source language output(s),
(3) optional compilation mechanisms,
(4) printed output(s),
(5) compiler diagnostic messages.

A CCI contains Compiler Control Statements which are made up of options or groups of options and their associated parameters if any. If no Compiler Control Statements appear on a CCI then the CCI is considered null.

There are two types of CCIs:

(1) those which are permanent and which may remain associated with the source language and

(2) those which are temporary and are only relevant to a given compilation.

There are two types of options:

(1) boolean,
(2) value.

A boolean option is one which is either enabled (set true) or disabled (set false). When enabled, it causes the compiler to apply an associated function to all subsequent processing until disabled. Boolean options may also have parameters associated with them. These parameters are related to the function which the boolean option affects.

A value option causes the compiler to store a value associated with a given option.

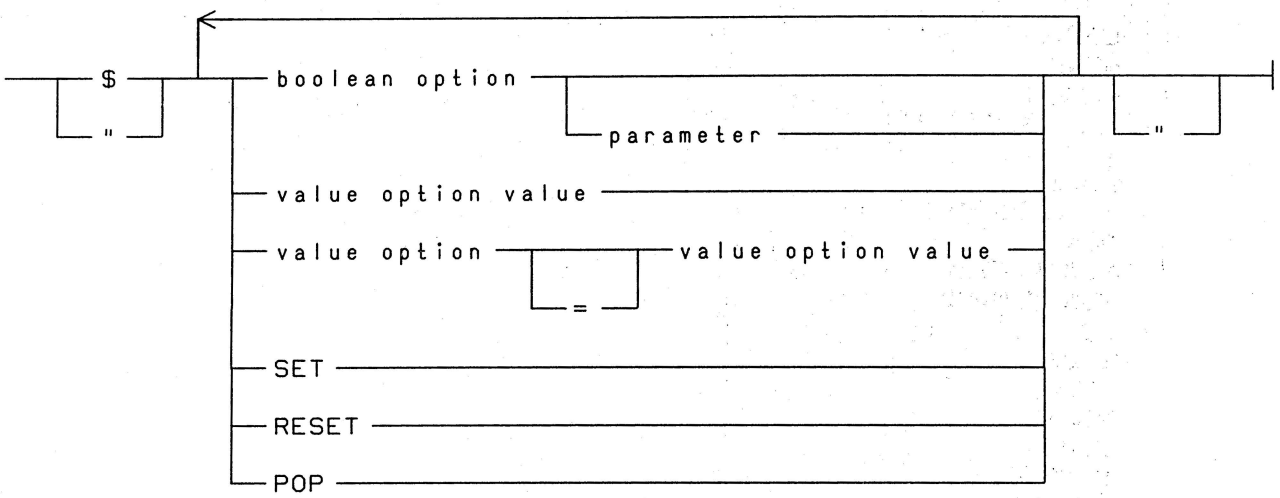
The general format of a CCI is the same as that of a source language image with respect to representation, position, and size of text and sequence number.

A CCI is identified by a dollar sign character ($) in the first text position of the image, or in the second text position of the image with the first position blank.

A CCI with a dollar sign character in the second text position is a permanent CCI and remains associated with the source language.

A CCI may also be identified by enclosing the option in double quotes ("). For example "SET LIST" is a temporary CCI if the first double quote is in text position one and a permanent CCI if the first double quote character is in any position except text position one.

The following is the general syntax of a CCI.



Parameters are optionally enclosed in parentheses, however, the use of an opening parenthesis requires that a matching closing parenthesis be present.

Compiler Control Images are interpreted from left to right beginning at the first text position following the dollar sign character or double quote character and continuing until the last text position or the closing double quote character (").

The status of boolean options is determined in one of three ways depending on the syntax used.

(1) If a boolean option appears on a CCI and is not the object of an explicit "SET" or "RESET", then it is implicitly enabled (set true) and the previous setting is stacked.

(2) If a boolean option appears on a CCI as the object of a "SET" or "RESET", then the specifed boolean option is enabled (set true) or disabled (set false) respectively and the previous setting is stacked.

(3) If a boolean option appears on a CCI as the object of a "POP", then the current setting is discarded and the previous setting is restored from the stack.

Summary of Options

The boolean options are:

ATTRIB
CHECK
CODE
CONTROLS
DUMP
ERRLIST
ERRORS
EXTERN

FLEVEL
LIST
LISTP
LIST1
LIST2
MERGE
MODELII
MULTIPLE
NEW
NEW1
NEW1SEQERR
NEW2
NEW2SEQERR
NOBINDINFO
SEG
SEGS
SEQ
SEQ1
SEQ2
SINGLE
SINGLE1
SINGLE2
SIXTY
STACK
STATISTICS
STMTNO
TIME
TRACE
TRACEENTRYS
TRACELABELS
TAPECOL
TITLE
VOID
VOIDT
XREF

The value options are:

CARDCOL
COL1
COL2
EXEC
INCLCOL
LIMIT
LINECNT
OPTIMIZE
STATCONTROL
WARN

Source Language Input(s)

ATTRIB

Type: boolean

Default: disabled

Functional Description

This option, when enabled, causes the compiler to include as part of the output listing, information concerning the explicit and default attributes for each identifier.

Syntax



CARDCOL

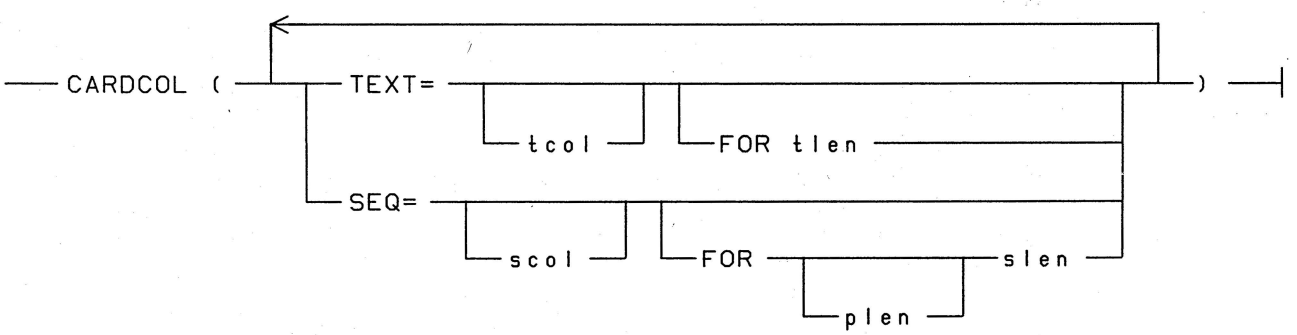
Type: value

Default: TEXT = Column 1, Length 72
SEQ = Column 73, Length 8

Functional Description

This option allows the specification of the column ranges containing the source language text, sequence number prefix, and sequence number within a source language image on the compiler's primary input file (CARD).

Syntax



Syntax Rules

1. The "scol" (sequence number column), "slen" (sequence number length), "tcol" (text column), "tlen" (text length), and "plen" (prefix length) must be unsigned integers.

2. If "scol", "slen", "tcol", "tlen", or "plen" are not specified, their previous values remain unchanged.

3. The allowable ranges are:

Function	Min	Max
tcol	1	80
tlen	1	80
plen	0	6
slen	0	12
scol	1	80

4. The maximum allowable sum of "tlen", "slen", and "plen" is 80.

Semantics

1. The "tcol" is the beginning column number for source language text.

2. The "tlen" is the number of columns for source language text.

3. The "scol" is the beginning column for sequence information.

4. The "plen" is the number of columns for the alphanumeric sequence prefix.

5. The "slen" is the number of columns for the sequence number.

6. The default values are:

tcol	1
tlen	72
scol	73
plen	0
slen	8

7. The column ranges for source language text, sequence number prefix, and sequence number must not overlap.

CHECK

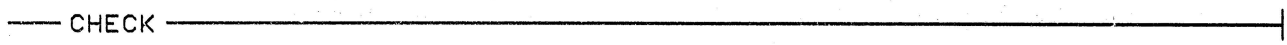
Type: boolean

Default: disabled

Functional description

This option, when enabled, causes sequence errors to be flagged on the TAPE file and both the NEWTAPE files.

Syntax



Semantics

1. If the sequence error occurs on the TAPE file, the message SEQERR followed by the sequence number of the last source image is printed on the output listing. If the sequence error occurs on the NEWTAPE file, the message on the printout is NEWTAPE SEQERR followed by the sequence number of the last source image, and the message NEWTAPE SEQERR is displayed on the SPO.

CODE

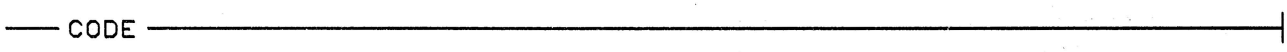
Type: boolean

Default: disabled

Functional Description

This option, when enabled, causes the compiler to list the object code produced by the compilation process.

Syntax



COL1

Type: value

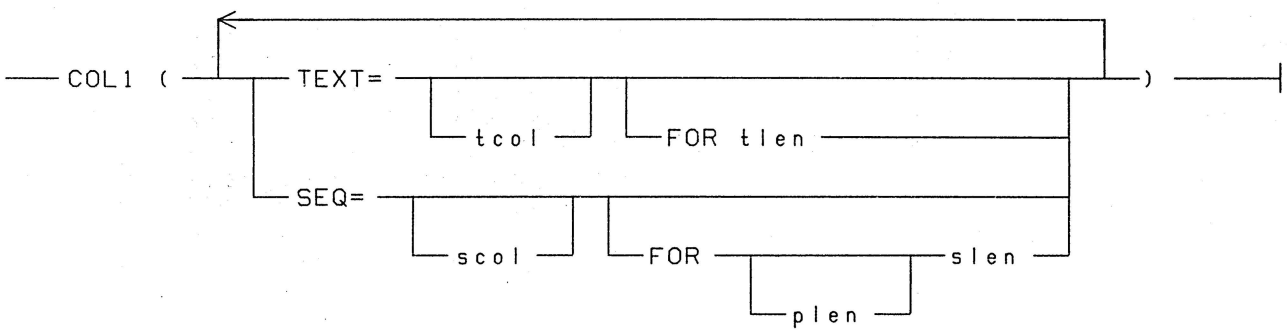
Default: TEXT = Column 1, Length 72

SEQ = Column 73, Length 8

Functional Description

This option allows the specification of the column ranges containing the source language text, sequence number prefix, and sequence number within a source language image on the output source language symbolic (NEWTAPE1).

Syntax



Syntax Rules

1. The "scol" (sequence number column), "slen" (sequence number length), "tcol" (text column), "tlen" (text length), and "plen" (prefix length) must be unsigned integers.

2. If "scol", "slen", "tcol", "tlen", or "plen" are not specified, their previous values remain unchanged.

3. The allowable ranges are:

Function	Min	Max
tcol	1	80
tlen	1	80
plen	0	6
slen	0	12
scol	1	80

4. The maximum allowable sum of "tlen", "slen", and "plen" is 80.

Semantics

1. The "tcol" is the beginning column number for source language text.

2. The "tlen" is the number of columns for source language text.

3. The "scol" is the beginning column for sequence information.

4. The "plen" is the number of columns for the alphanumeric sequence prefix.

5. The "slen" is the number of columns for the sequence number.

6. The default values are:

tcol	1
tlen	72
scol	73
plen	0
slen	8

7. The column ranges for source language text, sequence number prefix, and sequence number must not overlap.

COL2

Type: value

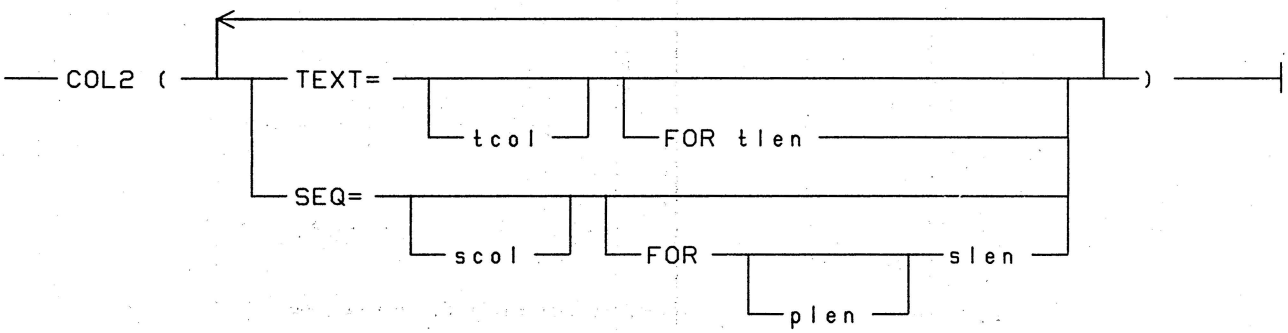
Default: TEXT = Column 1, Length 72

SEQ = Column 73, Length 8

Functional Description

This option allows the specification of the column ranges containing the source language text, sequence number prefix, and sequence number within a source language image on the expanded output source language symbolic (NEWTAPE2).

Syntax



Syntax Rules

1. The "scol" (sequence number column), "slen" (sequence number length), "tcol" (text column), "tlen" (text length), and "plen" (prefix length) must be unsigned integers.

2. If "scol", "slen", "tcol", "tlen", or "plen" are not specified, their previous values remain unchanged.

3. The allowable ranges are:

Function	Min	Max
tcol	1	80
tlen	1	80
plen	0	6
slen	0	12
scol	1	80

4. The maximum allowable sum of "tlen", "slen", and "plen" is 80.

Semantics

1. The "tcol" is the beginning column number for source language text.

2. The "tlen" is the number of columns for source language

text.

3. The "scol" is the beginning column for sequence information.

4. The "plen" is the number of columns for the alphanumeric sequence prefix.

5. The "slen" is the number of columns for the sequence number.

6. The default values are:

tcol	1
tlen	72
scol	73
plen	0
slen	8

7. The column ranges for source language text, sequence number prefix, and sequence number must not overlap.

CONTROLS

Type: boolean

Default: disabled

Functional Description

This option, when enabled, causes the compiler to list all temporary CCIs encountered during the compilation.

Syntax

DUMP

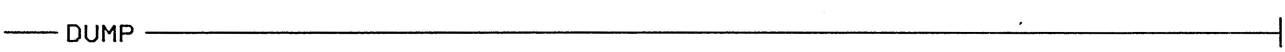
Type: boolean

Default: disabled

Functional Description

This option, when enabled, causes information which associates each symbolic identifier with its actual storage location, to be stored with the object program for use by the DUMP statement.

Syntax



Semantics

The PL/I PROGRAMDUMP is a snap shot of the stack showing the program status during execution when the dump is taken. It shows the exact history of the execution. Starting with the initiation of the program and ending at the point where the dump is taken, all identifiers are shown with their values (printed in the format implied by their declarations) according to the scope in which they belong.

To get the PLI PROGRAMDUMP rather than the standard B 7000/B 6000 PROGRAMDUMP the following criteria must be met.

1. The PL/I Compiler used to compile the program must be compiled with the option 'DUMP' SET; i.e., a '$ SET DUMP' control card must be compiled into the compiler.

2. The program being compiled must have the PL/I control option 'DUMP' SET. Since only the final state of this option is considered all occurrences of the PL/I DUMP statement will result in either a PL/I PROGRAMDUMP or a standard PROGRAMDUMP.

The following describes some special notations in the dump:

1. Headings between two full lines of '=' characters indicate the initiation of the program or the invocations of procedures or begin-blocks.

2. All program identifiers are underlined. If the identifier belongs to a structure, the level number of the structure appears to the left of the identifier.

3. The type of the identifier is enclosed by '<' and '>' characters.

4. Information following a '@' character shows the starting location of the associated storage. Brackets enclose the stack-address in EBCDIC and parentheses enclose the offset in the program memory. In the latter case 'W', 'C', and 'B' characters indicate the word, character, and bit offset respectively.

5. The contents of a simple variable are printed on the same line as the identifier if it fits; otherwise it is shown on the following line.

6. If the DUMP statement option 'ARRAY' or 'ARRAYS' is specified the contents of arrays are shown according to their ascending dimension values. The value of each element is tabulated under the last dimension. Numbers enclosed in parentheses are the indices. Lines with '.' characters as indices and '-' characters as the contents of the elements indicate that the previous line is to be printed indefinitely. There are a fixed number of elements printed in each line and the first and last line of each array row are always shown. If 'ARRAY' or 'ARRAYS' is not specified only the bounds of each array row is printed.

7. If the DUMP statement option 'FILE' or 'FILES' is not specified, then only the TITLE of the file, the KIND of the file, whether the file is OPEN or not, and whether the file is KEYED or not is printed. If 'FILE' or 'FILES' is specified the blocking specifications of the file are also printed.

ERRLIST

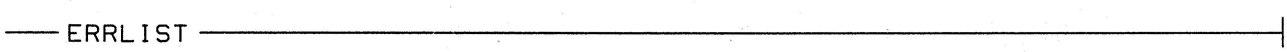
Type: boolean

Default : enabled for CANDE compiles and disabled otherwise

Functional Description

This option, when enabled, causes errors to be written to the terminal being used (see WARN).

Syntax



ERRORS

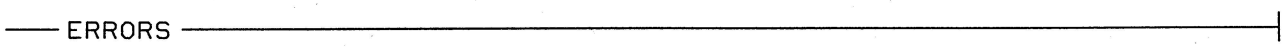
Type: boolean

Default: disabled for CANDE compiles and enabled otherwise

Functional Description

This option, when enabled, causes compiler generated error and warning messages to be written to the output listing.

Syntax



EXEC

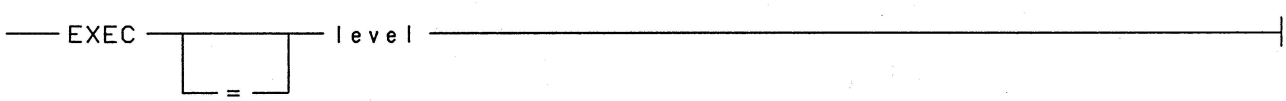
Type: value

Default: 7

Functional Description

This option indicates the lowest level of error message which will be considered a syntax error. Error messages with a lower number than this value are considered warnings (see WARN). When errors occur at this level or higher, the code file is not locked and the compilation, when completed, is terminated with a "SNTX" message.

Syntax



Syntax Rules

1. The "level" is an unsigned integer with a value greater than or equal to zero (0) and less than or equal to nine (9).

EXTERN

Type: boolean

Default: disabled

Functional Description

This option, when enabled, causes the compiler to include as part of the output listing, information concerning the explicit and default attributes for each identifier that has the EXTERNAL attribute.

Syntax



Semantics

1. This option has no effect if the "ATTRIB" option is enabled.

FLEVEL

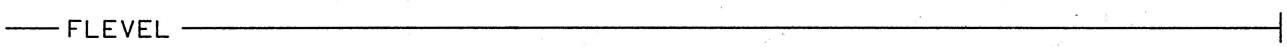
Type: boolean

Default: disabled

Functional Description

This option, when enabled, allows certain IBM constructs to be used.

Syntax



Semantics

1. The FLEVEL option specifies that if during compilation the compiler encounters a PL/I construct which the IBM F-LEVEL PL/I compiler would handle differently from the B6000/B7000 PL/I compiler, then it should be handled according to F-LEVEL.

INCLCOL

Type: value

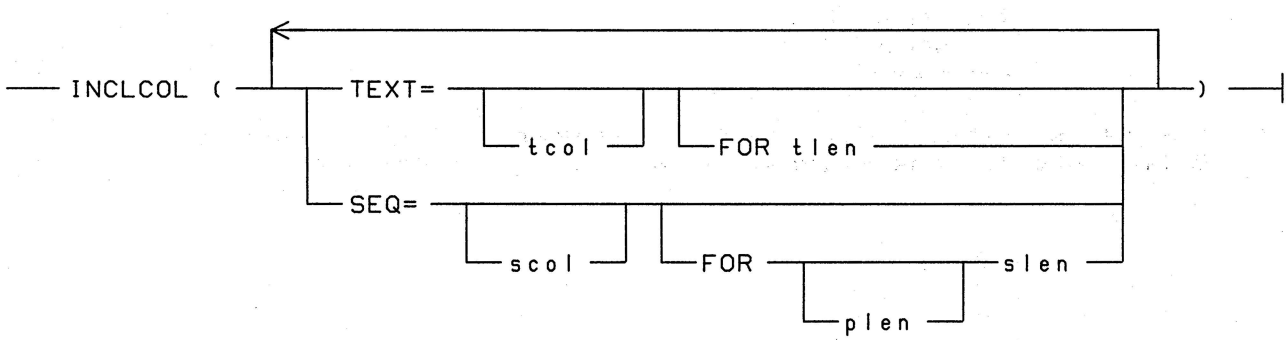
Default: TEXT = Column 1, Length 72

SEQ = Column 73, Length 8

Functional Description

This option allows the specification of the column ranges containing the source language text, sequence number prefix, and sequence number within a source language image in INCLUDE-files.

Syntax



Syntax Rules

1. The "scol" (sequence number column), "slen" (sequence number length), "tcol" (text column), "tlen" (text length), and "plen" (prefix length) must be unsigned integers.

2. If "scol", "slen", "tcol", "tlen", or "plen" are not specified, their previous values remain unchanged.

3. The allowable ranges are:

Function	Min	Max
tcol	1	80
tlen	1	80
plen	0	6
slen	0	12
scol	1	80

4. The maximum allowable sum of "tlen", "slen", and "plen" is 80.

Semantics

1. The "tcol" is the beginning column number for source language text.

2. The "tlen" is the number of columns for source language text.

3. The "scol" is the beginning column for sequence information.

4. The "plen" is the number of columns for the alphanumeric sequence prefix.

5. The "slen" is the number of columns for the sequence number.

6. The default values are:

tcol	1
tlen	72
scol	73
plen	0
slen	8

7. The column ranges for source language text, sequence number prefix, and sequence number must not overlap.

LIMIT

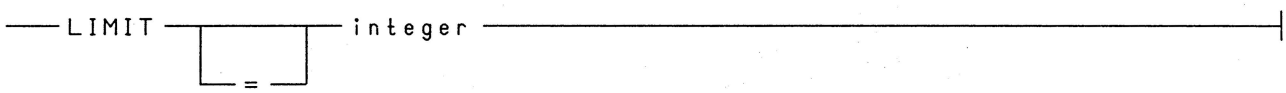
Type: value

Default: 100

Functional Description

This option specifies the maximum number of errors which the compiler may allow before a compilation is terminated.

Syntax



Syntax Rules

1. The "integer" may be any unsigned integer from zero (0) to 1023.

Semantics

1. The compiler keeps a count of only the syntax errors produced (not warnings). When this count exceeds the specified or default error limit, then the compilation is immediately terminated.

2. If the error limit is exceeded and any of the "NEW" options are enabled, those new files are purged (not locked).

LINECNT

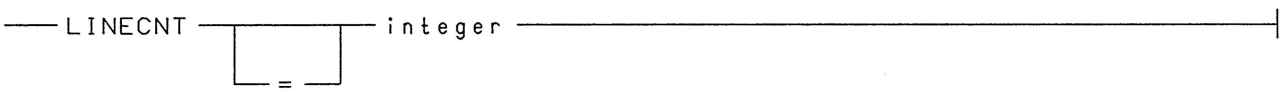
Type: value

Default: 56

Functional Description

This option specifies the maximum number of lines that may appear on a page of the output listing.

Syntax



Syntax Rules

1. The "integer" may be any unsigned integer, the minimum being 1 and the maximum being 100.

LIST

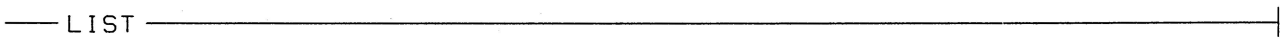
Type: boolean

Default: disabled for CANDE compiles, otherwise LIST1 enabled

Functional Description

This option, when enabled, causes the compiler to list the source language accepted for compilation.

Syntax



Semantics

1. This option affects both the "LIST1" and the "LIST2" options.

LISTP

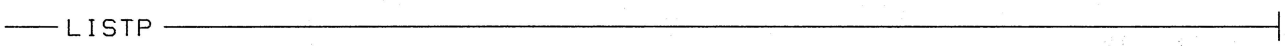
Type: boolean

Default: disabled

Functional Description

This option, when enabled, causes the compiler to list those source language images that originate from the primary input file (CARD).

Syntax



LIST1

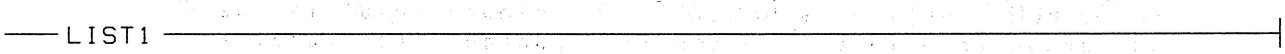
Type: boolean

Default: disabled for CANDE compiles otherwise enabled

Functional Description

This option, when enabled, causes the compiler to list the source language images accepted for compilation.

Syntax



Semantics

1. Enabling or disabling the "LIST" option also enables or disables the "LIST1" option.

LIST2

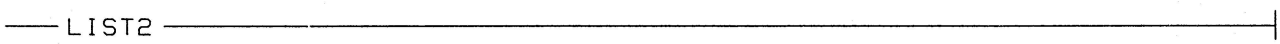
Type: boolean

Default: disabled

Functional Description

This option, when enabled, causes the compiler to list all statements compiled, including those generated or modified by the compile-time preprocessor.

Syntax



Semantics

1. Enabling or disabling the "LIST" option also enables or disables the "LIST2" option.

MERGE

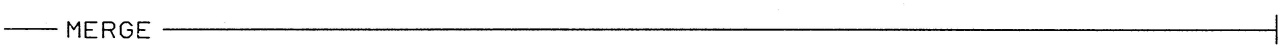
Type: boolean

Default: disabled

Functional Description

This option, when enabled, causes the compiler to begin merging the primary source language images (CARD) with the secondary source language input images (TAPE).

Syntax



Semantics

1. This option causes primary input, the CARD file, to be merged with secondary input, the TAPE file, to form the total input to the compiler.

2. If the MERGE option is RESET, only primary input is used and secondary input is totally ignored. Therefore, the total input to the compiler when the MERGE option is set consists of all card images from the CARD file, and all card images from the TAPE file that do not have sequence numbers that can be found on cards in the CARD file.

MODELII

Type: boolean

Default: enabled

Functional Description

This option, when enabled, causes code to be generated suitable for execution on a MODEL II B6700 processor. When reset, the code generated is suitable for execution by a MODEL I B6700 processor.

Syntax



MULTIPLE

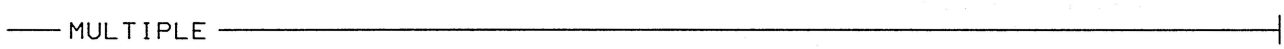
Type: boolean

Default: disabled

Functional Description

This option, when enabled, causes the last name of the program file name to be replaced by the name of the main entry point of the procedure.

Syntax



Semantics

1. If MULTIPLE is not set, the name of the first procedure will be the program file name specified on the WFL compile statement and all subsequent procedures will have the last name of the program file name replaced by the name of the main entry point of the procedure. If NEW , NEW1 or NEW2 is set, all procedures will be contained in a single new symbolic file. Procedures must be separated by a card with an EBCDIC question mark "?" in text column one or two and blank in all other columns.

NEW OR NEW1

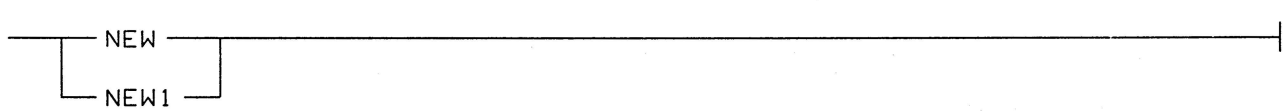
Type: boolean

Default: disabled

Functional Description

This option, when enabled, causes the compiler to create a new source language symbolic (NEWTAPE1) of all source language images accepted for compilation.

Syntax



Semantics

1. This option,when enabled, initiates the writing of all input source images accepted for compilation (but not those discarded by enabling the VOID or VOIDT options) to a new symbolic file(NEWTAPE1).

NEW1SEQERR

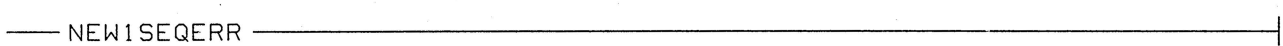
Type: boolean

Default: disabled

Functional Description:

When enabled, this option will cause records written to the NEWTAPE1 file to be checked for sequence errors.

Syntax:



Semantics:

1. As each sequence error is discovered, a warning message is placed on the output listing and also displayed on the SPO. If such errors occur, then at the end of compilation the NEWTAPE1 file will be purged (not locked).

NEW2

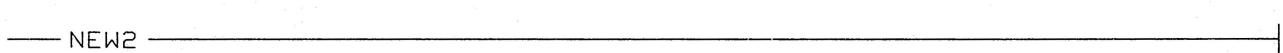
Type: boolean

Default: disabled

Functional Description

This option, when enabled, causes the compiler to create a new source language symbolic file (NEWTAPE2) of all source language statements, including those created or modified by the compile-time preprocessor.

Syntax



Semantics

1. This option initiates the process of writing the symbolic form of all compiled statements to a output symbolic file (NEWTAPE2).

NEW2SEQERR

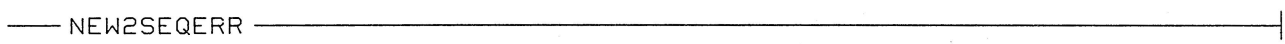
Type: boolean

Default: disabled

Functional Description:

When enabled, this option will cause records written to the NEWTAPE2 file to be checked for sequence errors.

Syntax:



Semantics:

1. As each sequence error is discovered, a warning message is placed on the output listing and also displayed on the SPO. If such errors occur, then at the end of compilation the NEWTAPE2 file will be purged (not locked).

NOBINDINFO

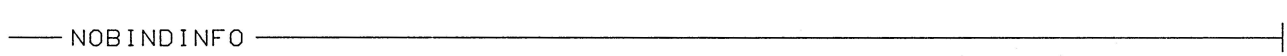
Type: boolean

Default: disabled

Functional Description

This option, when enabled, causes the compiler to suppress the insertion of binder control information into the object file.

Syntax



Semantics

1. This option must be enabled before the compiler encounters any source language text.

2. If this option is enabled or disabled after the compiler has encountered source language text, it will be treated as an error.

OPTIMIZE

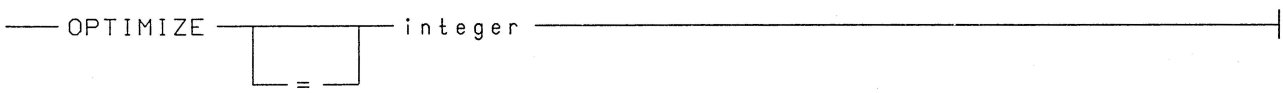
Type: value

Default: 0

Functional Description

This option causes the compiler to apply various optimization functions during the compilation process.

Syntax



Syntax Rules

1. "OPT" and "OPTIMIZE" are synonymous.

Semantics

1. The allowable values for "integer" are:

   0 = no optimization
   1 = minor optimization of pool data sharing
   3 = code analysis optimization

SEG OR SEGS

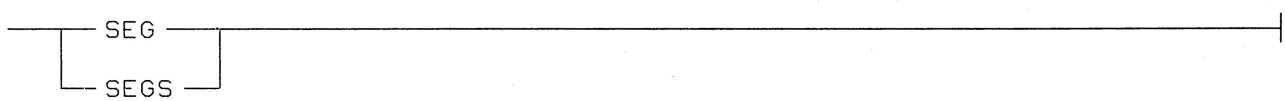
Type: boolean

Default: disabled

Functional Description

This option, when enabled, causes the compiler to include as part of the LIST2 listing information concerning the segmentation of the program.

Syntax



SEQ or SEQUENCE

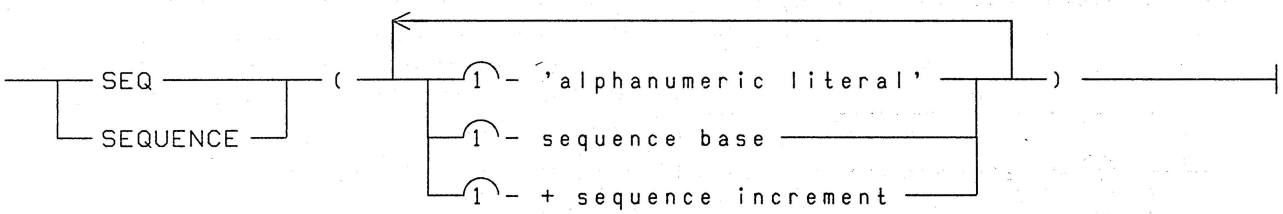
Type: boolean

Default: disabled

Functional Description

This option, when enabled, causes the compiler to assign new sequence numbers to the source language images accepted for compilation. The sequence number prefix, sequence base, and sequence increment may be specified as parameters.

Syntax



Semantics

1. SEQ affects both the SEQ1 and SEQ2 options.

SEQ1

Type: boolean

Default: disabled

Functional Description

This option, when enabled, causes the compiler to assign new sequence numbers to the source language images used by the "LIST1" and "NEW1" options.

Syntax

1. The 'alphanumeric literal' is assigned to the sequence prefix associated with the "COL1" option. The default prefix is all spaces.

2. The value of the "sequence base" is assigned to the sequence base used by this option. The default sequence base is 1000.

3. The value of the "sequence integer" is assigned to the sequence increment used by this option. The default sequence increment is +1000.

4. This option assigns the current sequence prefix and sequence base to the current statement image, then

increments the sequence base by the sequence increment.

5. 'alphanumeric literal' may not be more than 6 characters long.

SEQ2

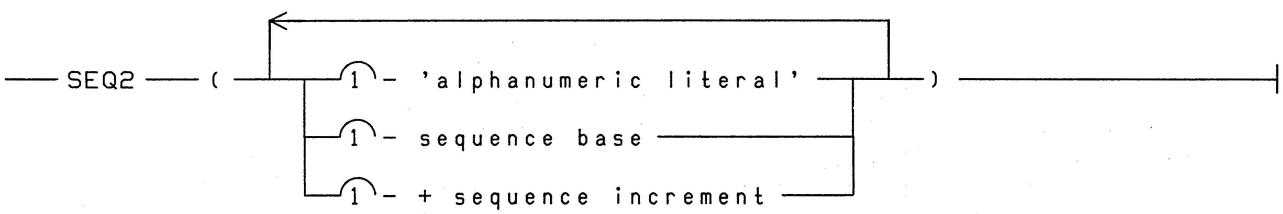
Type: boolean

Default: disabled

Functional Description

This option, when enabled, causes the compiler to assign new sequence numbers to the source language images used by the "LIST2" and "NEW2" options.

Syntax



Semantics

1. The 'alphanumeric literal' is assigned to the sequence prefix associated with the "COL2" option. The default prefix is all spaces ( ).

2. The value of the "sequence base" is assigned to the sequence base used by this option. The default sequence base is 1000.

3. The value of the "sequence integer" is assigned to the sequence increment used by this option. The default sequence increment is +1000.

4. This option assigns the current sequence prefix and sequence base to the current statement image, then increments the sequence base by the sequence increment.

5. 'alphanumeric literal' may not be more than 6 characters long.

SINGLE

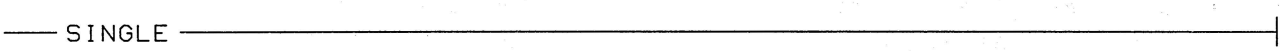
Type: boolean

Default: enabled

Functional Description

This option, when enabled, causes the compiler to single space all printed output. When disabled, the listing will be double spaced.

Syntax



Semantics

1. This option affects SINGLE1 and SINGLE2.

SINGLE1

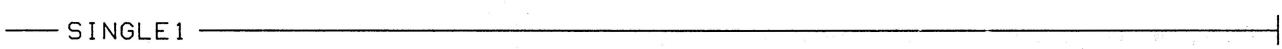
Type: boolean

Default: enabled

Functional Description

This option, when enabled, causes all output from the "LIST1" option to be single spaced. When disabled, the listing will be double spaced.

Syntax



SINGLE2

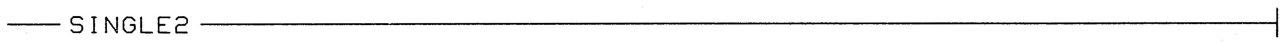
Type: boolean

Default: enabled

Functional Description

This option, when enabled, will cause all output from the "LIST2" option to be single spaced, otherwise, the listing will be double space

Syntax



SIXTY

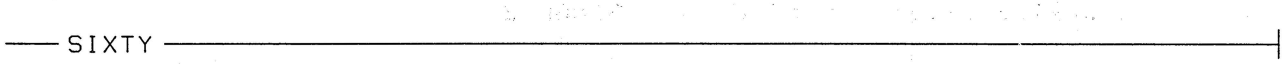
Type: boolean

Default: enabled

Functional Description

This option, when enabled, causes the compiler to use the 60-character-set convention for subsequent source language text. When disabled, the 48-character-set convention is used.

Syntax



Semantics

1. See APPENDIX 4 for the 48-character set.

STACK

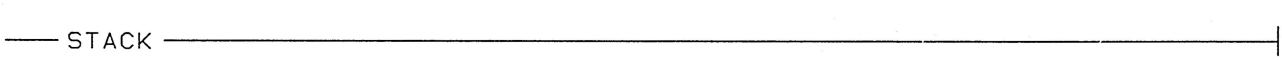
Type: boolean

Default: disabled

Functional Description

This option, when enabled, causes the compiler to include as part of the LIST2 listing, information concerning the stack allocation of variables within the object code produced by the compilation process.

Syntax



STATCONTROL

Type: value

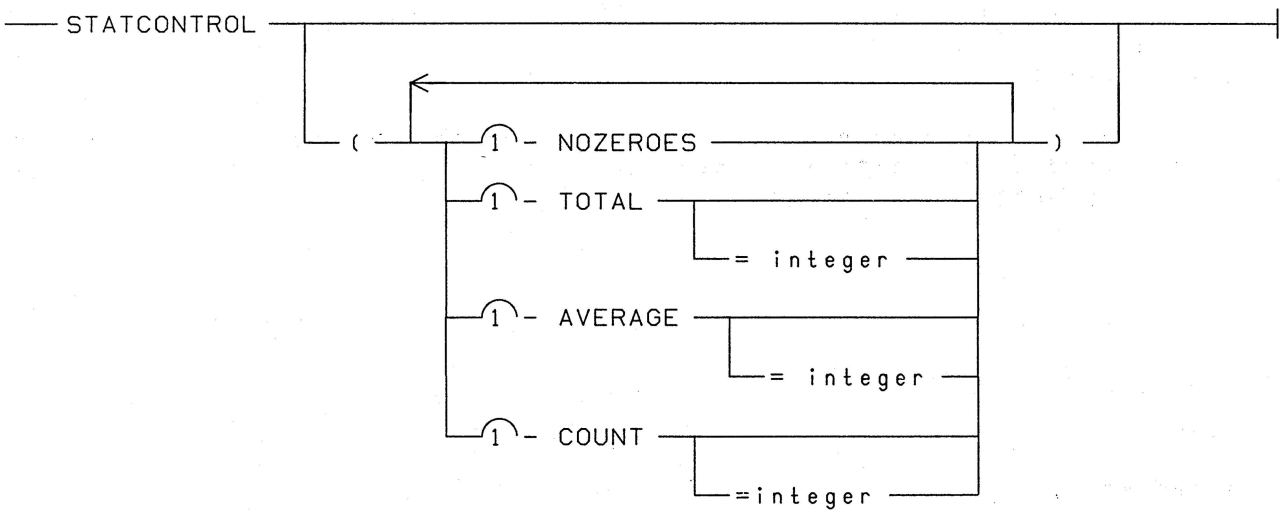
Syntax

The statistics summary produced by the execution of a PL/I program consists of one unconditional printer listing followed by up to three optional, sorted printer listings.

The unconditional listing shows procedure names and block numbers indicating the block structure of the PL/I program. The total elapsed time, average elapsed time and frequency of use for each entry is shown. Each time figure also indicates the percentage of the total program elapsed time that the figure represents.

The three conditional listings show the same basic information, sorted by one of the three figures: total elapsed time, average elapsed time, or frequency of use.

The nature and extent of the printer listing is controlled by the STATCONTROL option. This option may only be SET (ie. not POPped or RESET); if SET more than once, only the last setting is used. This option has no effect at all if STATISTICS is not also SET at least once during the compilation. The format of this control is



NOZEROS

Indicates that the unconditional printer listing is not to include those procedures and blocks which are never called during the program execution. If this option is not specified, blocks and procedures that are not called will appear on the listing with zero counts and zero times.

The three sorted conditional listings will never show zero entries, regardless of the NOZEROS control.

TOTAL

Controls the sorted iisting keyed by the total elapsed time for procedures and blocks. The "=integer" is a simple integer constant indicating the maximum number of entries to appear in this sorted listing. If '=integer' is missing or if 'integer' is greater than 1023, then '=1023' is assumed. If '=0' is specified, no sorted printout will occur. 'Integer' must not be negative.

AVERAGE

Controls the sorted listing keyed by the average elapsed time for procedures and blocks. The meaning of '=integer' is the same as described for the TOTAL option, above.

COUNT

Controls the sorted listing keyed by the number of times that each block and procedure was called in the program execution. The meaning of '=integer' is the same as described for the TOTAL option, above.

Any of the TOTAL, AVERAGE or COUNT options that do not appear will default to TOTAL=0, AVERAGE=0, or COUNT=0 as appropriate. If the STATCONTROL option is not set in a compilation that also sets the STATISTICS option, a default of

```
STATCONTROL ( TOTAL=0  AVERAGE=0  COUNT=0 )
```

will be used; this will cause only the unconditional statistics printer listing to be produced.

The compiler "LIST2" listing will indicate the beginning of each program "logic unit" when the LOGIC or COUNTS specifications of the STATISTICS option is SET. The form of this indication is:

```
procedure-name : lll
          or
   BLOCK nnnn : lll
```

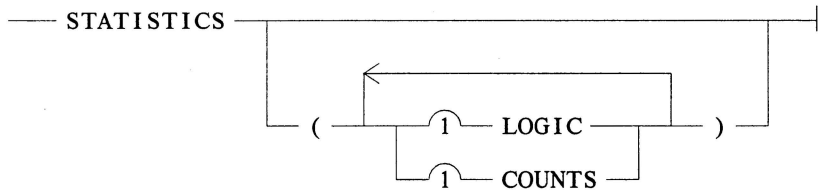
The "lll" represent the number of the logic unit for the indicated procedure or block, and are used to identify the entries in the statistics summary listings produced following program execution.

STATISTICS

Type: boolean

Default: disabled

Syntax



This control performs the normal SET/RESET/POP operations on the compiler toggle that indicates whether or not to compile statistics gathering information for procedures and BEGIN-END blocks. This option returns the processor times from the execution of these sections of code. If the control is enabled when a PROCEDURE or a BEGIN statement is encountered, code will be compiled for that entire procedure or block to gather timing statistics. If the STATISTICS option is disabled at the beginning of a procedure or block, no statistics code will be generated for that procedure or block. If the option is enabled later within the procedure or block, only subsequent procedures and blocks will gather statistics.

LOGIC

Indicates that additional timing statistics are to be gathered for the program's individual logic units. A logic unit is the stream of executable code that begins at a point that can be branched to and ends at the next such "branch point". This is not limited to explicit program labels, as the compiler may generate "branch points" for various language constructs (such as IF...THEN...; ELSE...; and repetitive DO statements). Injudicious use of the LOGIC option can cause a program to run noticably slower. RESET STATISTICS also disables LOGIC and POP STATISTICS pops to the previous setting for both STATISTICS and LOGIC.

COUNTS

Is like the LOGIC specifier in that it gathers information about individual logic units within a program. The only difference between COUNTS and LOGIC is that COUNTS collects only usage counts for the logic units (elapsed time information is only collected for procedures and blocks). Injudicious use of the COUNTS option can cause a program to run noticably slower (although not as seriously as with the LOGIC specifier). RESET STATISTICS also disables COUNTS and POP STATISTICS pops to the previous setting for both STATISTICS and COUNTS.

STMTNO

Type: boolean

Default: enabled

Functional Description

This option, when enabled, causes information which associates a symbolic image sequence number with program addresses to be stored with the object program.

Syntax

Semantics

1. This option causes sufficient information to be stored with the object program to permit the source language sequence number that is associated with a given program address to be determined.

2. Should a program terminate abnormally, the source language sequence number associated with the point of termination is provided.

TAPECOL

Type: value

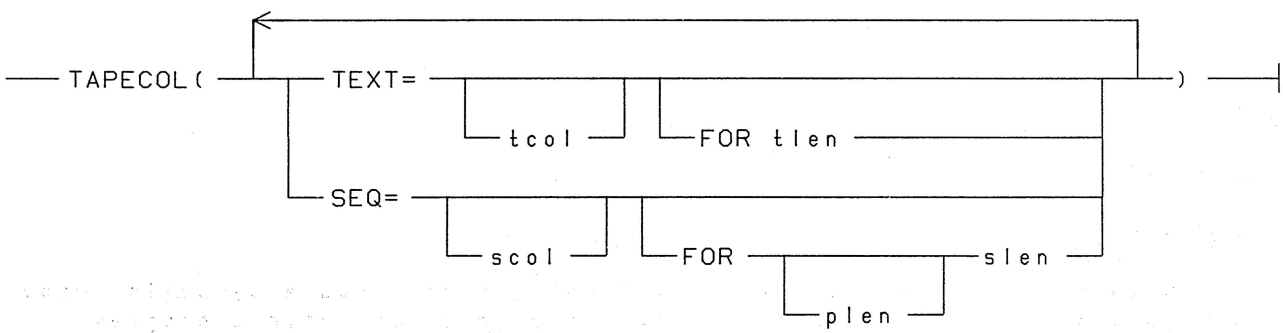
Default: TEXT = Column 1, Length 72

SEQ = Column 73, Length 8

Functional Description

This option allows the specification of the column ranges containing the source language text, sequence number prefix, and sequence number within a source language image on the secondary input file (TAPE).

Syntax



Syntax Rules

1. The "scol" (sequence number column), "slen" (sequence number length), "tcol" (text column), "tlen" (text length), and "plen" (prefix length) must be unsigned integers.

2. If "scol", "slen", "tcol", "tlen", or "plen" are not specified, their previous values remain unchanged.

3. The allowable ranges are:

Function	Min	Max
tcol	1	80
tlen	1	80
plen	0	6
slen	0	12
scol	1	80

4. The maximum allowable sum of "tlen", "slen", and "plen" is 80.

Semantics

1. The "tcol" is the beginning column number for source language text.

2. The "tlen" is the number of columns for source language text.

3. The "scol" is the beginning column for sequence information.

4. The "plen" is the number of columns for the alphanumeric sequence prefix.

5. The "slen" is the number of columns for the sequence number.

6. The default values are:

tcol	1
tlen	72
scol	73
plen	0
slen	8

7. The column ranges for source language text, sequence number prefix, and sequence number must not overlap.

TIME

Type: boolean

Default: disabled

Functional Description

This option, when enabled, causes the compiler to produce a summary of appropriate information about the compilation itself on the output listing.

Syntax



Semantics

1. The summary produced by the enabling of this option is the same as that produced if the "LIST1 or LIST2" option is true. Therefore, this option takes effect only if the "LIST" options are all disabled.

TITLE

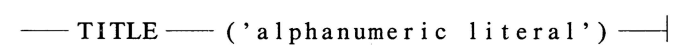
Type: value

Default: the title of the code file being generated.

Functional Description

This option causes the compiler to store a title to be used on all subsequent pages of the output listing.

Syntax



Syntax Rules

1. The 'alphanumeric literal' must contain at least one character.
2. The maximum number of characters in 'alphanumeric literal' is 66.
3. The quote character (') may not appear within the 'alphanumeric literal'.

Semantics

1. The compiler will store the specifed 'alphanumeric literal' for use as a title on all subsequent pages of the output listing.
2. Subsequent uses of this option result in the storing of the new 'alphanumeric literal' and the previous title is lost.

TRACE

Type: boolean

Default: disabled

Functional Description

This option, when enabled or disabled, enables or disables the "TRACEENTRYS" and the "TRACELABELS" options.

Syntax



TRACEENTRYS

Type: boolean

Default: disabled

Functional Description

This option, when enabled, causes the object program to note on the SYSPRINT file each entry into a procedure.

Syntax



TRACELABELS

Type: boolean

Default: disabled

Functional Description

This option, when enabled, causes the object program to note on the SYSPRINT file, each labelled statement that is executed.

Syntax



VOID

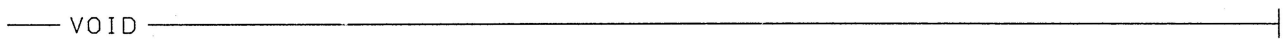
Type: boolean

Default: disabled

Functional description

When enabled, this option will cause all input from files CARD and TAPE (except compiler control cards from the file CARD) to be ignored until VOID is RESET or POP'ED into a RESET state).

Syntax



VOIDT

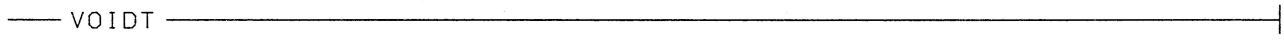
Type: boolean

Default: disabled

Functional description

This option, when enabled, causes only primary input (CARD) to be compiled. All secondary input is ignored.

Syntax



Syntax Rules

1. This option may only appear on a CCI in the primary source language input (CARD).

Semantics

1. This option is ignored if the "MERGE" option is not enabled.

2. This option does not alter the normal merging process. However, it causes the compiler to unconditionally discard all source language images selected from the secondary input (TAPE) including CCIs.

3. The source language images discarded as a result of the enabling of this option are not carried forward to the output symbolic file (NEWTAPE1), should the "NEW1" option be enabled.

WARN

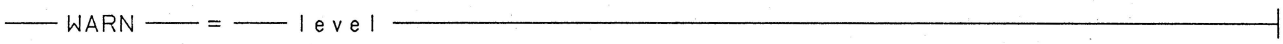
Type: value

Default: 1

Functional Description

This option specifies the lowest severity level for which warning or error messages will be printed. Warning messages with a severity level less than the specified value will not be printed.

Syntax



Syntax Rules

1. The "level" is an unsigned integer with a value greater than or equal to zero (0) and less than or equal to nine (9).

XREF

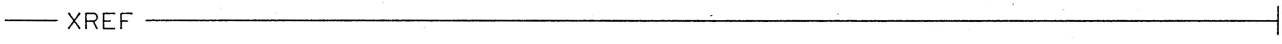
Type: boolean

Default: disabled

Functional Description

This option, when enabled, causes the compiler to produce cross-reference listing to the source language accepted for compilation.

Syntax



INDEX

ABS Arithmetic Built-In Function A1- 14
<access-attribute> . . . . . 4-36- A, 4- 39
<activate-statement> 11- 6
ACTIVATE statement 11- 2, 11- 6, 11- 7
ACTIVATION AND TERMINATION OF BLOCKS 10- 2
ADD Arithmetic Built-In Function A1- 14
ADDR Based Storage Function A1- 33
AFTER String Built-In Function A1- 3
<alignment-attribute> 4- 50, 4- 51
ALL Array Manipulation Function A1- 28
<allocate-statement> 7- 47
ALLOCATE statement . . . 4- 27, 4- 47, 7- 15, 7- 47, 7- 48, 7- 52, 10- 6, 10- 8
<allocation> 7- 47
ALLOCATION Built-In Function A1- 34
alphabetic characters 3- 1
alphanumeric character 3- 1
alternative attributes 7- 29
ANY Array Manipulation Function A1- 28
<area> 7- 25
AREA 3- 6, 4- 2, 4- 29, 4- 30, 4- 32, 4- 60, 4- 61, 7- 5, 7- 24, 7- 48, 7- 49, 7- 52, A1- 38, A2- 6, A2- 9
<area-attribute> 4- 23, 4- 27, 4- 29
Area Data 3- 7
<area-expression> 4- 29
<area-reference> 4- 26, 4- 27, 4- 28
<argument> 7- 9, 9- 2
<argument-list> 4- 53
<arithmetic> 4- 7
ARITHMETIC BUILT-IN FUNCTIONS A1- 14
<arithmetic-constant> 8- 10
Arithmetic Data 3- 5, 8- 12
Arithmetic Operators 3- 2
<arithmetic-type> 4- 60
Arithmetic Variables 3- 11
<array> 4- 7, 4- 43
<array-assignment> 7- 4, 7- 5, 7- 6, 7- 8
ARRAY MANIPULATION BUILT-IN FUNCTIONS A1- 27
<array-variable> 7- 4
<assignment-statement> 7- 4
<asterisk> 4- 11, 4- 29, 4- 43, 4- 48
ATAN Mathematical Built-In Function A1- 21
ATAND Mathematical Built-In Function A1- 22
ATTRIB A7- 2, A7- 4, A7- 13
<attribute> . . . . . 4-36- A, 7- 25, 7- 27
<attribute-expression> 7- 27
<attribute-expression1> 7- 27
<attribute-expression2> 7- 27
<attribute-expression3> 7- 27
<attribute-exression1> 7- 27
<attribute-keyword> 7- 27
<attribute-list> 4- 32, 4- 35, 11- 8
<attribute-name> 7- 53
<attribute-set> 4- 60, 7- 26, 7- 27, 7- 29
AUTOMATIC storage class attribute 10- 6
AVAILABLE 7- 40, 7- 42

BASE 3- 5, 5- 3, 7- 30, A1- 14, A1- 15, A1- 17, A1- 19, A1- 20, A1- 29, A1- 30
<base attribute> 4- 8
<base-attribute> 4- 7, 4- 8, 4- 13
<base-item> 4- 54

Based Allocation 7- 48
<based-attribute> 4- 49
BASED attribute 4- 11, 4- 44, 4- 47, 4- 51, 7- 25, 7- 48, 7- 50, 10- 8
<based-locator-variable> 4- 27
BASED STORAGE BUILT-IN FUNCTIONS A1- 32
<based-variable> 4- 27, 7- 51, 7- 52
BASED variable 7- 39, 7- 40, 7- 52, 10- 8
BEFORE String Built-in Function A1- 3
<begin-block> 6- 4
<begin-option-list> 7- 21
<begin-statement> 6- 4, 7- 21
BEGIN statement . . . . . 6- 4, 7- 21, 7-21- A, 10- 2, 10- 9, A7- 29
BINARY . . . . . 3- 5, 4- 8, 4- 9, 4- 32, 4- 35, 4-36- A, 4- 41, 4- 59, 4- 60, 4- 61, 5- 9, 7- 28, 7- 30, 9- 6, A1- 1, A1- 9, A1- 15, A3- 1
BINARY Arithmetic Built-In Function A1- 15
binary digit 3- 1
<binary-fixed-point-constant> 3- 7
<binary-floating-point-constant> 3- 7
BIT . . . . . 4- 3, 4- 5, 4- 10, 4- 32, 4- 53, 4- 60, 4- 61, 7- 19, 7- 20, 7- 46, 7-46- C, A1- 1, A1- 4, A1- 9
BIT String Built-In Function A1- 4
<bit-string-format-item> 8- 24
<bit-string-operation> 5- 7
<block> 10- 14
<Boolean-valued-system-file-attribute> 7- 40
<bound> 4- 43
Braces 2- 2
Brackets 2- 2, A7- 11
Built-in Function Names 3- 4
<builtin-attribute> 4- 30, 4- 34
BY NAME option 7- 6, 7- 7

<call-statement> 7- 9, 9- 2
CALL statement 4- 2, 7- 3, 7- 9, 9- 3, 9- 7
CARDCOL A7- 3, A7- 4
CEIL Arithmetic Built-In Function A1- 15
CHAR String Built-In Function A1- 5
CHARACTER . . . . . 1- 1, 3- 1, 3- 2, 4- 3, 4- 10, 4- 32, 4-36- A, 4- 58, 4- 60, 4- 61, 7- 25, 7- 50, 8- 9, 10- 7, 11- 2, 11- 3, 11- 4, 11- 6, 11- 8, A1- 8, A3- 1, A4- 1
<character-picture-element> 4- 13
<character-picture-item> 4- 13
<character-picture-specification> 4- 12, 4- 13, 8- 24, 8- 25
<character-string> 7- 40
Character String Constants 3- 8
Character String Data 8- 14
<character-string-format-item> 8- 24
Character String Variables 3- 11
CHARACTERISTICS A1- 14
CHECK A2- 1, A7- 2, A7- 6
CLASSIFICATION OF CONDITIONS A2- 1
<close-option> 7- 34
<close-statement> 7- 34
CLOSE statement 7- 2, 7- 34, 8- 4
Closing a File 8- 4
CODE 7- 30, A7- 2, A7- 6
Coded real fixed-point binary data 8- 13
Coded real floating-point binary data 8- 13
Coded real floating-point decimal data 8- 13
COLLATE String Built-In Function A1- 5
COLUMN format item 8- 26
COL1 . . . . . A7- 3, A7- 7, A7- 24
COL2 . . . . . A7- 3, A7- 9, A7- 25
<comment> 3- 22

<comment-string> 3- 22
Comments 3- 22
<comparison-operation> 5- 6
Comparison Operators 3- 2
<compile-time assignment statement> 11- 7
<compile-time declare-statement> 11- 8
<compile-time do-group> 11- 9
<compile-time expression> 11- 4
<compile-time-expression> 11- 7, 11- 10
Compile-time expressions 11- 3
<compile-time GO TO statement> 11- 9
<compile-time IF statement> 11- 10
<compile-time null-statement> 11- 11
<compile-time procedure> 11- 4
compile-time procedure 11- 2, 11- 3, 11- 4, 11- 5, 11- 8, 11- 9
<compile-time PUT statement> 11- 11
COMPILE-TIME STATEMENTS 11- 1, 11- 6
<compile-time variable> 11- 7
<compile-time- variable> 11- 11
<compile-time-variable> 11- 11
compile-time variable 11- 3, 11- 6, 11- 7, 11- 9
<compound statement> 6- 1
COMPUTATIONAL BUILT-IN FUNCTIONS A1- 3
<concatenation-operation> 5- 8
<condition> 7- 30, 7- 33
Condition 3- 5, 6- 6, 10- 9, 10- 14, A1- 1, A1- 30, A1- 31, A1- 32
CONDITION BUILT-IN FUNCTIONS A1- 30
<condition-list> 7- 31
<condition-name> 6- 7, 10- 11
CONDITION-NAME 4- 2
<condition-prefix> 6- 7
<conditional compound statement> 6- 1
<config> A1- 9
<constant> 4- 52, 8- 15
CONSTANT attribute 4- 3
<constant expression> 7- 18
<constant-expression> 4- 40
Constants 3- 7, 3- 8, 3- 9, 3- 10, 3- 13, 8- 10, 9- 3
CONSTANTS 3- 7, 3- 8, 3- 9
Contextual declarations 4- 2
Control Format Items 8- 25
Controlled Allocation 7- 47
<controlled-variable> 7- 51, 7- 52
CONTROLS A7- 2, A7- 10
CONVERSION condition . . . 4- 13, 8- 22, 10- 15, A1- 30, A1- 31, A1- 32, A1- 37
COPY String Built-In Function A1- 5
COS Mathematical Built-In Function A1- 22
COSD Mathematical Built-In Function A1- 22
COSH Mathematical Built-In Function A1- 22
COUNT Built-In Function A1- 35
COUNTS option A7- 30
<credit> 4- 14, 4- 21
<currency-character> 4- 19

<D digits> 8- 23
<data> 4- 7
Data Aggregates 3- 14
<data-attribute> 4- 50
<data-character> 3- 22
<data-description-attributes> 4- 7
<data-directed-data-in-stream> 8- 15
<data-directed-data-specification> 8- 14
Data Directed Output 8- 17
Data-directed transmission 8- 6

DATA ELEMENTS 3- 5
Data format items 8- 20
<data-list> 8- 5, 8- 7, 8- 10, 8- 11, 8- 14, 8- 15, 8- 17, 8- 19
DATA ORGANIZATION 3- 13
<data-specification> 7- 37, 7- 43, 8- 5
<data-specification-attributes> 7- 22, 7- 23, 7- 24
<data-type> 4- 60
DATAFIELD 10- 14, 10- 16, A1- 2, A1- 30, A2- 4
DATAFIELD Condition Built-In Function A1- 30
DATE Built-In Function A1- 35
<deactivate-statement> 11- 6
DEACTIVATE statement 11- 7
<debit> 4- 14, 4- 21
DECAT String Built-In Function A1- 6
DECIMAL . . . 3- 5, 4- 8, 4- 9, 4- 32, 4- 35, 4- 59, 4- 60, 4- 61, 5- 9, 7- 28, 7- 30, 11- 3, 11- 7, 11- 8, A1- 1, A1- 9, A1- 16, A3- 1
DECIMAL Arithmetic Built-In Function A1- 16
Decimal digits 3- 1
<decimal-fixed-point-constant> 3- 7
<decimal-floating-point-constant> 3- 7
<decimal-specifier> 4- 14, 4- 16
<declaration> 7- 26
<declaration-list> 7- 26
DECLARATIONS 4- 1
<declare-statement> 7- 25, 7- 26
DECLARE statement 3- 15, 4- 1, 4- 2, 4- 35, 4- 46, 4- 47, 7- 25, 7- 26, 7- 47, 8- 2, 8- 4, 9- 1, 9- 7, 9- 11, 11- 2, 11- 3, 11- 5, 11- 7, 11- 8
<declared identifiers> 7- 26
<default-set> 7- 27
<default statement> 7- 27
DEFAULT statement 4- 60, 7- 27, 7- 28, 7- 29, 7- 30
DEFINED 4- 30, 4- 54, 4- 55, 4- 56, 4- 57, 4- 58, 4- 60, 4- 61, 7- 20, 7- 25, 7- 45, 9- 1, A3- 1
<defined-attribute> 4- 50, 4- 54
DEFINED attribute 4- 30, 4- 56, 4- 57, 7- 45, 9- 1
<delay-statement> 7- 10
DELAY statement 7- 10
<delete-statement> 7- 35
DELETE statement 4- 38, 4- 42, 7- 2, 7- 35, 7- 36, 8- 28
<description-list> 4- 32, 4- 34
<digit> 2- 1, 2- 2
DIM Array Manipulation Function A1- 28
<dimension-attribute> 4- 43, 4- 45
DIMENSION attribute 4- 32, 9- 8
DIRECT file 7- 35, 8- 28
<display-statement> 7- 36
DISPLAY statement 7- 2, 7- 36
DIVIDE Arithmetic Built-In Function A1- 16
<do-group> 6- 3
<do-statement> 2- 1, 6- 3, 7- 10
DO statement 3- 3, 6- 3, 7- 3, 7- 10, 7- 11, 7- 12, A1- 36
<dump-option> 7- 30
<dump-statement> 7- 30
DUMP statement 7- 30, A7- 11, A7- 12

<edit-directed-data-specification> 8- 19
Edit-directed transmission 8- 6
<element> 8- 7, 8- 8
<elementary-arithmetic-operation> 5- 3
ellipsis 2- 2
EMPTY built-in function 4- 29
<end-statement> 7- 22
END statement 6- 3, 6- 4, 7- 3, 7- 17, 7- 22, 10- 2, 11- 1, 11- 4
ENDFILE(<filename>) A2- 3

ENDPAGE(<filename>) A2- 3
<entry> 4- 7, 4- 30
<entry-attribute> 4- 30, 4- 31
ENTRY attribute 4- 30, 4- 31, 4- 32, 7- 26, 9- 6, 9- 7, 9- 8, 9- 9
<entry constant> 7- 18
Entry Constants 3- 10
<entry-expression> 4- 34
<entry-expressions> 4- 32
Entry-Label Data 3- 6
<entry-name> 4- 53, 7- 9, 7- 22, 7- 23, 9- 2
entry name 4- 5, 4- 24, 4- 31, 4- 32, 4- 34, 4- 35, 4- 51, 6- 6, 7- 22, 7- 24, 9- 1, 9- 2, 9- 4, 9- 5, 9- 6, 9- 7, 9- 9, 10- 4, 11- 2, 11- 5, 11- 6
<entry-statement> 7- 22
ENTRY statement . . . 4- 1, 4- 2, 4- 32, 4- 35, 6- 4, 7- 1, 7- 3, 7- 22, 7- 23, 7- 24, 7- 29, 9- 1
<entry-type> 4- 60
Entry Variables 3- 12
ENVIRONMENT . . 4- 36, 4- 40, 4- 42, 4- 61, 7- 18, 7- 34, 7- 35, 7- 40, 7- 41, 7- 42, 8- 1, 8- 3, A3- 1
<environment-attribute> . . . . . 4-36- A, 4- 40
ERF Mathematical Built-In Function A1- 23
ERRLIST A7- 2, A7- 12
ERRORS A7- 2, A7- 12
<event-valued-attribute> 7- 21
EXCEPT String Built-In Function A1- 7
EXEC A7- 3, A7- 13
<exit-statement> 7- 14
EXIT statement 7- 3, 7- 14, 10- 2
Explicit Declarations 4- 1
<exponent> 8- 22, 8- 23
<exponent-character> 4- 14, 4- 22
<expression> . . . 4- 11, 4- 29, 4- 43, 7- 11, 7- 46, 8- 5, 9- 3, A1- 19, A1- 20
<expression 1> 7- 10
<expression 2> 7- 10
<expression 3> 7- 10
<expression 4> 7- 10
<expression1> 7- 11, 7- 12, 7- 13, 8- 8
<expression2> 7- 12, 7- 13, 8- 8
<expression3> 7- 11, 7- 12, 7- 13, 8- 8
<expression4> 8- 8
<expression5> 7- 12, 7- 13
<expression8> 7- 13
EXTERN 4- 50, A7- 2, A7- 13
EXTERNAL attribute 4- 1, 4- 4, 4- 5, 4- 49, A7- 13

<factor> A1- 6, A1- 10
<file> . . . . . 4- 7, 4-36- A
<file-attribute> . . . . . 4-36- B
<file-attribute-list> 8- 1
File Constants 3- 10
<file designator> 7- 40
<file-designator> 7- 53
<file-exp> 4- 42
<file expression> 7- 18
<file name> 8- 5
<file-name> . . . 2- 1, 7- 34, 7- 35, 7- 37, 7- 39, 7- 40, 7- 41, 7- 43, 7- 44, 7- 45, 7- 46, 7-46- A, 7-46- B, A1- 35, A1- 36
FILE option 4- 2, 7- 34, 7- 41
<file-title> 7- 40, 11- 10
<file-type> 4- 60
File Variables 3- 12
<filename> A2- 3, A2- 4, A2- 5

FIXED . . . . 2- 2, 3- 6, 4- 5, 4- 6, 4- 8, 4- 9, 4- 26, 4- 32, 4- 35, 4-36- A, 4- 59, 4- 60, 4- 61, 5- 9, 7- 17, 7- 24, 7- 27, 7- 30, 8- 9, 8- 13, 8- 17, 9- 1, 9- 4, 9- 6, 9- 8, 9- 9, 11- 2, 11- 3, 11- 4, 11- 7, 11- 8, A1- 1, A1- 16, A1- 17
FIXED Arithmetic Built-in Function A1- 16
<fixed-point-format-item> 8- 21
<fixed-point-picture> 4- 14
FIXEDOVERFLOW 6- 7, 10- 9, 10- 10, A2- 1, A2- 2, A3- 1
FLEVEL A7- 3, A7- 14
FLOAT 2- 2, 3- 4, 3- 6, 4- 3, 4- 5, 4- 8, 4- 32, 4- 35, 4- 60, 4- 61, 5- 9, 7- 17, 7- 27, 7- 28, 7- 30, 8- 9, 9- 1, 9- 5, 9- 6, 9- 7, 9- 8, A1- 1, A1- 17, A6- 4
FLOAT Arithmetic Built-in Function A1- 17
<floating-point-format-item> 8- 22
<floating-point-picture> 4- 14
<format-attribute> 4- 23, 4- 25
<format-designator> 4- 25
<format-item> 8- 20
Format Label Constants 3- 10
Format Label Variables 3- 13
<format-list> 7- 36, 7- 37, 8- 5, 8- 19, 8- 20
<format-list-item> 8- 20
<format-statement> 7- 36
FORMAT statement 7- 2, 7- 36, 7- 37, 8- 27
<free-statement> 7- 51
FREE statement 4- 29, 7- 47, 7- 51, 7- 52, 10- 6, 10- 8
FROM . . . 4- 42, 7-46- A, 7-46- B, 7-46- C, 7- 47, 8- 28, A1- 1, A1- 7, A1- 8, A2- 8, A6- 1
<fully-prefixed-statement> 6- 7
<function-attribute> . . . . . 4-36- A, 4- 37
<function-reference> 9- 2

GENERIC 4- 32, 4- 34, 7- 9, 9- 9
<generic-attribute> 4- 30, 4- 32
<generic-attribute-list> 4- 32
<generic-attributes> 4- 32
<generic-element> 4- 32
<get-statement> 7- 37
GET statement 7- 37, 7- 41, 8- 5, 8- 15, A2- 8
<go-to-statement> 7- 14
GO TO statement . . . 7- 3, 7- 13, 7- 14, 7-14- A, 7- 15, 7- 37, 11- 4, 11- 9, 11- 10
<group> 6- 3

HBOUND Array Manipulation Function A1- 29

<identifier> . . . 2- 2, 3- 17, 3- 19, 3- 21, 6- 2, 7- 18, 7- 25, 7- 27, 7- 28, 7- 47, 10- 14, 11- 4, 11- 6, 11- 8, A2- 7
<identifiers> 6- 2
IDENTIFIERS 3- 3
<if-statement> 7- 16
IF statement 7- 13, 7- 16, 10- 9, 11- 9, 11- 10
IGNORE option 7- 45
Implicit Opening 8- 3, 8- 4
<in-attribute> 4- 23, 4- 28, 4- 29, 4- 30
<in-option> 4- 29, 4- 30
IN option 7- 48, 7- 49, 7- 52
INCLCOL A7- 3, A7- 14
<include-file-name> 11- 10
<include-statement> 11- 10
INCLUDE statement 11- 6, 11- 10
INDEX option . . . . . 7- 45, 7-46- B
INDEX String Built-In Function A1- 8
<initial attribute> 11- 8

<initial-attribute> 4- 50, 4- 51, 11- 8
INITIAL attribute 4- 24, 4- 30, 4- 47, 4- 51, 4- 53, 7- 39
<initial-call> 4- 30, 4- 51, 4- 53, 11- 8
<initial-list> 4- 51, 4- 52, 4- 53
INPUT attribute 4- 37
<input-file-option> 8- 5
<input option> 7- 18
<input-option> 7- 18
INPUT/OUTPUT CONDITIONS A2- 3
<input-string-option> 8- 5
<insertion-character> 4- 14, 4- 18
<integer-constant> 4- 9, 4- 13, 4- 23, 4- 54, 7- 25, 8- 20
INTERNAL attribute 4- 1, 8- 6
<internal-file-name> 7- 53
IOTIME Built-In Function A1- 35
<item> 4- 52
<iteration-expression> 7- 10, 7- 11, 7- 12, 7- 13
<iteration factor> 8- 20
<iteration-factor> 4- 52, 8- 20
<iteration-specification> 4- 52, 7- 10, 7- 11, 7- 12, 7- 13

<key-attribute> . . . . . 4-36- A, 4- 39
KEY(<filename>) A2- 5
<key option> 7- 18
<key-option> 7- 18
KEY option 7- 35, 7- 45, 8- 28
KEYED attribute 4- 39, 7- 45, 8- 28
KEYFROM . . . . . 4- 39, 4- 42, 7- 39, 7- 45, 7-46- A, 7-46- B, 7-46- C, 8- 28, A2- 9, A2- 10
KEYFROM option . . . . . 7- 45, 7-46- B
KEYLENGTH 4- 40, 4- 41, 4- 42
KEYORDER 4- 40, 4- 42
KEYSPERENTRY 4- 41, 4- 42
KEYSTART 4- 41, 4- 42
KEYTO option 8- 28, A1- 32
KEYTYPE 4- 41
KEYWORDS 3- 4

<label> . . 6- 3, 6- 4, 6- 7, 7- 22, 7- 36, 7- 37, 11- 4, 11- 6, 11- 7, 11- 8, 11- 9, 11- 10, 11- 11
<label-attribute> 4- 23
LABEL attribute 4- 24, 9- 7
Label Data 3- 6
Label Prefixes 4- 1, 6- 2
<labeled-statement> 6- 2
<label1> 11- 9
LBOUND Array Manipulation Function A1- 29
<length> 7- 25, A1- 4, A1- 5, A1- 8, A1- 10
<length-attribute> 4- 10, 4- 11, 4- 12, 4- 35
length of data fields 8- 18
LENGTH String Built-In Function A1- 9
<letter> 7- 27, 7- 28
<level> 7- 25, 7- 26
Libraries 9- 11
Library Attribute 4- 36, 7- 54, 7- 55, 9- 11
<like-attribute> 4- 44, 4- 45, 4- 46
LIMIT 7- 15, A7- 3, A7- 16
LINECNT A7- 3, A7- 17
LINENO Built-In Function A1- 36
<list-directed-data-specification> 8- 10
List-Directed Input Format 8- 10, 8- 15
List-Directed Output Format 8- 11, 8- 18
List-Directed Transmission 8- 6, 8- 14
LISTP A7- 3, A7- 18

LIST1 . . . . . A7- 3, A7- 17, A7- 18, A7- 24, A7- 26, A7- 33
LIST2 . . . . . A7- 3, A7- 17, A7- 18, A7- 19, A7- 25, A7- 26, A7- 29, A7- 33
<locate-statement> 7- 39
LOCATE statement 4- 27, 4- 40, 4- 47, 4- 48, 7- 2, 7- 39
<locator-attribute> 4- 23, 4- 26
Locator Data 3- 6
<locator-qualifier> 4- 27, 4- 47, 7- 51
LOG Mathematical Built-In Function A1- 24
LOGIC option A7- 30
LOG10 Mathematical Built-In Function A1- 24
LOG2 Mathematical Built-In Function A1- 24
LOW String Built-In Function A1- 10

<mantissa> 8- 22
MATHEMATICAL BUILT-IN FUNCTIONS A1- 21
MAX Arithmetic Built-In Function A1- 17
<member-attribute> 4- 44, 4- 45
<memory option> 7- 18
<memory-option> 7- 18, 7- 19
MERGE A7- 3, A7- 19, A7- 35
Merging of Attributes 8- 2
MIN Arithmetic Built-In Function A1- 18
MISCELLANEOUS BUILT-IN FUNCTIONS A1- 34
MOD Arithmetic Built-In Function A1- 18
MODE 3- 6, 4- 42, 5- 3, A1- 14, A1- 15, A1- 17, A1- 19, A1- 20, A1- 29, A1- 30
<mode-attribute> 4- 7, 4- 8
MODELII A7- 3, A7- 19
MULTIPLE A7- 3, A7- 20
MULTIPLY Arithmetic Built-In Function A1- 19

NAME(<filename>) A2- 4
NEW A7- 3, A7- 16, A7- 20
NEW1 . . . . . A7- 3, A7- 20, A7- 24, A7- 35
NEW1SEQERR A7- 3, A7- 20, A7- 21
NEW2 . . . . . A7- 3, A7- 20, A7- 21, A7- 25
NEW2SEQERR A7- 3, A7- 22
NOBINDINFO A7- 3, A7- 22
NODUPLICATE A2- 9
NONVARYING 4- 12, 4- 61, A3- 1
notation constant 2- 1
notation variable 2- 1, 2- 2
NULL Based Storage Built-In Function A1- 33
<null-statement> 7- 55
NULL statement 11- 11
NULLO Based Storage Built-In Function A1- 33
<number-of-digits> 4- 9
Numeric Binary Data 8- 14
Numeric Decimal Data 8- 14
<numeric-picture-element> 4- 14
<numeric-picture-format-item> 8- 23
<numeric-picture-item> 4- 14
<numeric-picture-specification> 4- 12, 4- 14, 8- 23, 8- 24

OFFER 7- 40, 7- 42
OFFSET . . . 3- 6, 4- 2, 4- 26, 4- 27, 4- 29, 4- 32, 4- 60, 7- 6, 7- 22, 7- 24, 7- 25, 7- 30, 7- 52, 9- 9, A1- 2, A1- 33, A1- 34
<offset-attribute> 4- 26
OFFSET Based Storage Built-In Function A1- 33
<offset-expression> 4- 27
<ON compound statement> 6- 1
<on-statement> 7- 30
ON statement . . . . . 4-36- B, 7- 1, 7- 3, 7- 30, 7- 31, 7- 33, 10- 4, 10- 8, 10- 9, 10- 10, 10- 12, 10- 13, 10- 14

<on-unit> 7- 30, 7- 31, 7- 32, 7- 33, 7- 55, 10- 14
ONCHAR 10- 14, 10- 15, 10- 16, A1- 2, A1- 30, A1- 31, A1- 32, A1- 37
ONCHAR Condition Built-In Function A1- 30
ONCODE 4- 42, 10- 14, 10- 15, 10- 16, A1- 2, A1- 31, A2- 5, A2- 7
ONCODE Condition Built-In Function A1- 31
ONCODES FOR CONDITIONS A2- 7
ONCOUNT 10- 14, 10- 16, A1- 2, A1- 31
ONCOUNT Condition Built-In Function A1- 31
ONFILE 10- 14, 10- 16, A1- 2, A1- 31
ONFILE Condition Built-In Function A1- 31
ONKEY . . . . . 7-46- C, 10- 14, 10- 16, A1- 2, A1- 32
ONKEY Condition Built-In Function A1- 32
ONLOC 10- 14, 10- 16, A1- 2, A1- 32
ONLOC Condition Built-In Function A1- 32
ONSOURCE 10- 14, 10- 15, 10- 16, A1- 2, A1- 31, A1- 32, A1- 37
ONSOURCE Condition Built-In Function A1- 32
ONSOURCE Pseudo-Variable A1- 37
<open-statement> 7- 40
OPEN statement . . . 4- 38, 4- 41, 7- 2, 7- 40, 7- 41, 7- 42, 7- 53, 8- 3, 8- 4, A2- 3, A2- 4
Opening a File 7- 25, 8- 3
<operand> 5- 3, 5- 6, 5- 7, 5- 8
OPERATORS 3- 2
OPTIMIZE A7- 3, A7- 23
<option-list> . . . . . 7- 18, 7- 19, 7- 23, 7- 35, 7- 37, 7- 39, 7- 43, 7- 44, 7-46- A
OPTIONS . . . . 4- 36, 4-36- A, 4- 40, 5- 6, 5- 9, 7- 14, 7- 20, 7- 21, 7- 23, 7-24- A, 7- 34, 7- 35, 7- 40, 7- 41, 7- 42, 8- 1, 9- 11, A6- 4
Options Attribute 4- 36
<options-group> 7- 34, 7- 40
<or-symbol> 7- 27, 7- 28
OUTPUT . . . . . 1- 1, 4- 37, 4- 39, 4- 60, 4- 61, 7- 18, 7- 20, 7- 40, 7- 41, 7- 43, 7- 44, 7-46- B, 8- 1, 8- 2, 8- 3, 8- 4, 8- 6, 8- 28, 11- 2, A2- 3
<output-file-option> 8- 5
<output option> 7- 18, 7- 19
<output-option> 7- 18, 7- 19
<output-string-option> 8- 5
OVERFLOW . . . 3- 5, 6- 2, 6- 7, 7-21- A, 7- 55, 10- 9, 10- 10, 10- 11, 10- 12, 10- 13, A2- 1, A2- 2, A3- 1
OVERFLOW statement 10- 11, 10- 12
<overpunch-character> 4- 14, 4- 21

<packed-specifier> 4- 14, 4- 16, 4- 17
PAGE option . . . . . 7- 43, 7-46- B, A2- 3, A2- 4
<parameter> 7- 22, 7- 23
<parameter-attribute> 4- 30, 4- 35
<parameter-attribute-list> 4- 31, 4- 32, 4- 34
parameters . . . . . 4- 30, 4- 31, 4- 34, 4-36- B, 4- 44, 4- 49, 7- 19, 7- 22, 7- 23, 8- 15, 9- 1, 9- 2, 9- 3, 9- 6, 9- 7, 9- 8, 9- 11, 10- 3, 11- 5, A2- 2, A7- 1
parenthesis 3- 1, 3- 3, A2- 10, A7- 2
PHYSICALUPDATE 4- 41
<picture> 4- 7
<picture-attribute> 4- 12
PICTURE attribute 4- 9, 4- 10, 4- 13
<picture-exponent> 4- 14
<picture-mantissa> 4- 14
<picture-scale-factor> 4- 14, 4- 22, 4- 23
<picture-specification> 4- 32
<picture-type> 4- 12, 4- 60
POINTER . . 3- 6, 4- 2, 4- 26, 4- 32, 4- 48, 4- 49, 4- 60, 7- 6, 7- 22, 7- 24, 7- 52, 10- 5, A1- 2, A1- 34, A3- 1
<pointer-attribute> 4- 26
POINTER built-in function 7- 6, 7- 52

<pointer-expression> 4- 27
<position-attribute> 4- 54, 4- 55, 4- 58
<positioning-format-item> 8- 26
positioning format items 8- 25
<precision> 4- 32
Precision 3- 8, 4- 9
PRECISION 3- 6, 4- 9, 4- 32, 4- 60, 4- 61, 5- 3, 7- 25, 7- 28, 7- 30,
11- 3, 11- 7, 11- 8, A1- 1, A1- 14, A1- 15, A1- 17, A1- 19, A1- 20, A1- 29,
A1- 30, A1- 31, A3- 1
<precision-attribute> 4- 7, 4- 9, 4- 13
<print-attribute> . . . . . 4-36- A, 4- 38
PRINT attribute 4- 37, 4- 38, 4- 39, 8- 5, 8- 6, A1- 36
<printing-format-item> 8- 26
Problem Data 3- 5
<procedure-block> 6- 4
<procedure-reference> 9- 2
<procedure-statement> 6- 4, 7- 23, 7- 24
PROCEDURE statement 4- 1, 4- 32, 6- 4, 7- 3, 7- 22, 7- 23, 7- 24, 9- 1,
9- 3, 10- 9, 11- 1, 11- 2, 11- 4, 11- 5
PROCTIME Built-In Function A1- 36
PROD Array Manipulation Function A1- 29
PROGRAM CHECKOUT CONDITIONS A2- 5
<program-control> 4- 7, 4- 23
PROGRAMMER NAMED CONDITIONS A2- 7
PROLOGUES 10- 1
<pseudo-variable> 7- 4, 7- 10, 7- 53, 7- 54, 7- 55
<put-statement> 7- 43
PUT statement . . . 4- 15, 4- 38, 7- 37, 7- 41, 7- 43, 7- 44, 8- 5, 8- 9, 8- 26,
8- 27, 11- 11, A2- 3, A2- 8

<qualified name> 3- 19

RANDOM Mathematical Built-In Function A1- 24
<range-specification> 7- 27, 7- 28
READ INTO option 8- 28
READ operation 8- 28
<read-statement> 7- 44
READ statement . . . 4- 2, 4- 4, 4- 27, 7- 44, 7- 45, 7- 46, 8- 3, 8- 28, 10- 4,
10- 8
REAL 4- 8, 4- 9, 4- 32, 4- 35, 4- 60, 4- 61, A1- 2, A1- 14
<real-arithmetic-constant> 3- 7
RECORD attribute 4- 38, 4- 39, 4- 40
RECORD data transmission 8- 1
RECORD(<filename>) A2- 5
record operation statements 8- 27
<record-type> 4- 60
<refer-option> 4- 11, 4- 29, 4- 43, 4- 44, 4- 47, 4- 48, 4- 49
<remote-format-item> 8- 27
REPEAT String Built-In Function A1- 10
<repetition-factor> 4- 13, 4- 14, 4- 15
<repetitive-specification> 8- 8
RESCAN option 11- 7
<return-statement> 7- 17
RETURN statement 7- 3, 7- 17, 7- 22, 7- 24, 7- 30, 10- 2, 10- 7, 10- 12,
11- 4, 11- 5
<returns-attribute> 4- 30, 4- 35
RETURNS attribute 4- 32, 4- 35, 9- 9, 10- 1
REVERSE String Built-In Function A1- 10
<revert-statement> 7- 31
REVERT statement 7- 31, 7- 32, 10- 13
<rewrite-statement> . . . . . 7-46- A
REWRITE statement . . . . . 7- 2, 7-46- A, 8- 28
ROUND Arithmetic Built-In Function A1- 19

<S-D digits> 8- 23
SAVEDELETIONS 4- 41, 4- 42
<scalar-area-variable> 7- 47, 7- 51
<scalar-assignment> 7- 4, 7- 5
<scalar-character-string-variable> 7- 37, 7- 43, 7- 45, 8- 5
<scalar-character-variable> 7- 36
<scalar-exp> 4- 42
<scalar expression> 8- 5
<scalar-expression> . . . 3- 17, 4- 52, 7- 4, 7- 5, 7- 10, 7- 16, 7- 17, 7- 34, 7- 35, 7- 36, 7- 37, 7- 39, 7- 40, 7- 42, 7- 43, 7- 44, 7- 45, 7- 46, 7-46- A, 7-46- B, 7-46- C, 7- 53, 7- 54, 8- 5, 8- 20, 8- 28
<scalar-expression list> 7- 40
<scalar-expression-list> 7- 42
Scalar Items 3- 13
<scalar-locator-expression> 4- 27, 4- 46, 7- 25
<scalar-locator-variable> 7- 47
<scalar-name> 4- 11, 4- 48, 4- 49
<scalar-pointer-variable> 7- 39, 7- 40, 7- 44
<scalar-pseudo-variable> 8- 8
<scalar-ptr-var> 4- 42
<scalar-ptr-variable> 4- 42
<scalar-ref> 4- 42
<scalar-task-name> 7- 9
<scalar-variable> 4- 52, 7- 4, 7- 21, 7- 53, 7- 54, 7- 55, 8- 8, 8- 15
Scalar Variables 3- 14
<scale-attribute> 4- 7, 4- 8, 4- 13
<scale-factor> 4- 9
<scope> 4- 7
<scope-attribute> . . . . . 2- 3, 4- 35, 4-36- B, 4- 49, 4- 50
SCOPE attribute 10- 7
SEG A7- 3, A7- 23
SEGS A7- 3, A7- 23
<selector-function> 7- 27, 7- 28, 7- 29
SEPARATORS AND OTHER DELIMITERS 3- 3
SEQ 7- 41, A7- 3, A7- 4, A7- 7, A7- 9, A7- 14, A7- 24, A7- 31
SEQUENCE 7- 3, A7- 24
SEQUENTIAL file 8- 28
SEQ1 . . . . . A7- 3, A7- 24
SEQ2 . . . . . A7- 3, A7- 24, A7- 25
<set-option> 4- 27
SET option 7- 39, 7- 45, 7- 48, 7- 49, 8- 27
SIGN Arithmetic Built-In Function A1- 20
<sign-character> 4- 19
<sign-currency-character> 4- 14
<signal-statement> 7- 33
SIGNAL statement 7- 3, 7- 31, 7- 33, 10- 15
<simple-character-string-constant> 8- 10
Simple Names 3- 17
<simple-statement> 6- 1
SIN Mathematical Built-In Function A1- 25
SIND Mathematical Built-In Function A1- 25
SINGLE A7- 3, A7- 26
<single-statement> 6- 3
SINGLE1 . . . . . A7- 3, A7- 26
SINGLE2 . . . . . A7- 3, A7- 26, A7- 27
SINH Mathematical Built-In Function A1- 25
SIXTY A7- 3, A7- 27
<size-attribute> 4- 29, 4- 30
SIZE condition . . . . . 4- 16, 7-21- A, 8- 22, 8- 23, 10- 12, 10- 13, A2- 2
SKIP format item 8- 26
SKIP option . . . . . 7- 38, 7- 41, 7- 43, 7-46- B, A2- 3
<sort identifier> 7- 18
<sort-identifier> 7- 18
<sort option> 7- 18
<sort-option> 7- 18

<sort-statement> 7- 18
SORT statement 7- 18
<spacing-format-item> 8- 25
special characters 3- 1
<specification> 8- 8
<specification-string> A1- 6, A1- 7
SQRT Mathematical Built-In Function A1- 25
STACK A7- 3, A7- 27
STATCONTROL A7- 3, A7- 27, A7- 28, A7- 29
<statement> 6- 2, 6- 3, 6- 7
<statement-body> 6- 1
<statement-identifier> 6- 1
statement identifier 3- 3, 3- 4
<statement-label-constant> 4- 23, 4- 25
Statement Label Constants 3- 9
<statement-label-designator> 8- 27
<statement-label-name> 7- 14
Statement Label Variables 3- 11
STATIC attribute 4- 24, 4- 44, 4- 49, 4- 53
STATIC storage attribute 10- 6
STATISTICS A7- 3, A7- 28, A7- 29, A7- 30
STMTNO A7- 3, A7- 30
<stop-statement> 7- 21
STOP statement 7- 14, 7- 20, 10- 2
<storage> 4- 7, 4- 60
STORAGE 4- 46, 10- 5, A1- 32, A6- 1
<storage-attribute> 4- 44
<storage-class> 4- 46, 4- 49, 4- 60
Storage Classes 10- 5
<stored-data> 4- 60
STREAM attribute 4- 38, 4- 39, 4- 40, A1- 35
STREAM data transmission 4- 30, 8- 1
<stream-input-statement> 8- 5
<stream-output-statement> 8- 5
STREAM transmission 8- 6
<stream-type> 4- 60
<string> 4- 7, 4- 10, A1- 6, A1- 9, A1- 10, A1- 11, A1- 12, A1- 37
String attributes 4- 10
String Data 3- 6, 8- 14
STRING HANDLING BUILT-IN FUNCTIONS A1- 3
<string-one> A1- 13
STRING option 4- 15, 7- 38
<string-two> A1- 13
<string-type> 4- 60
<string-type-attribute> 4- 10
STRINGRANGE 6- 7, 10- 9, 10- 10, A1- 11, A2- 1, A2- 6, A3- 1
<structure> 4- 7, 4- 44
Structure Arrays 3- 14, 3- 15
<structure-assignment> 7- 4, 7- 5, 7- 7
<structure-attribute> 4- 44, 4- 45
<structure-expression> 7- 4
<structure-variable> 4- 45, 4- 46, 7- 4
Structures 3- 14, 7- 26, A6- 2, A6- 3
SUBFILE . . . . . 7- 34, 7- 40, 7- 41, 7- 44, 7- 46, 7-46- A, 7-46- B, 7-46- C
<subroutine-reference> 9- 2
<subscript> 3- 17, 3- 21
<subscripted-name> 3- 17
<subscripted-qualified-name> 3- 21
SUBSCRIPTRANGE 4- 55, 6- 7, 10- 9, 10- 11, A2- 5, A3- 1
SUBSTR Pseudo-Variable A1- 37
SUBSTR String Built-In Function A1- 11
SUM Array Manipulation Function A1- 30
syntax rule 2- 2
SYSTEM ACTION CONDITIONS A2- 7

<system-file-attribute> 4- 40, 7- 40, 7- 53
<system-file-attribute-assignment-statement> 7- 53
<system-file-attribute-reference-statement> 7- 53
<system-file-attribute-specification> 7- 40
<system-file-attribute-specification-list> 7- 40
<system-task-attribute> 7- 54
<system-task-attribute-assignment-statement> 7- 54
<system-task-attribute-reference-statement> 7- 54

TAN Mathematical Built-In Function A1- 26
TAND Mathematical Built-In Function A1- 26
TANH Mathematical Built-In Function A1- 26
TAPECOL A7- 3, A7- 31
Task Data 3- 6
<task-designator> 7- 54
TASK option 4- 2, 7- 9
<task-variable> 7- 54
The System Library Attribute Reference Statement 7- 55, 9- 11
The System Task Attribute Reference Statement 7- 54
TIME 1- 1, 11- 1, 11- 3, 11- 4, 11- 6, A1- 2, A1- 34, A1- 35, A1- 36, A7- 3, A7- 32, A7- 33
TIME Built-In Function A1- 36
TITLE . . . . . 4- 36, 4-36- A, 4- 43, 7- 40, 7- 41, 7- 42, 7- 55, 8- 3, A7- 3, A7- 12, A7- 33
TO-clause 7- 11
TRACE A7- 3, A7- 33, A7- 34
TRACEENTRYS A7- 3, A7- 34
TRACELABELS A7- 3, A7- 34
TRANSLATE String Built-In Function A1- 12
<transmission-attribute> . . . . . 4-36- A, 4- 37, 4- 38
Transmission of Data List Elements 8- 9
TRANSMIT(<filename>) A2- 4
TRUNC Arithmetic Built-In Function A1- 20

UNDEFINEDFILE(<filename>) A2- 4
UNDERFLOW 6- 7, 10- 9, 10- 10, A2- 1, A2- 3, A3- 1
<unit-1> 7- 16
<unit-2> 7- 16
UNSPEC Pseudo-Variable A1- 38
UNSPEC String Built-In Function A1- 12
UPDATE attribute 4- 37

<value> 11- 11, A1- 4, A1- 5
<value-item> 4- 52
<variable> . . . . . 4- 42, 7- 10, 7- 11, 7- 39, 7- 44, 7- 45, 7- 46, 7-46- A, 7-46- B
<variable-attribute> 4- 50, 4- 54
VARIABLES 1- 1, 3- 11, 11- 3, A1- 36
VARYING . . . . . 2- 2, 4- 12, 4-36- A, 4- 54, 4- 55, 4- 60, 5- 8, 7- 5, 11- 3, 11- 8, A3- 1
<varying-attribute> 4- 10, 4- 11, 4- 12
VERIFY String Built-In Function A1- 13
VOID A7- 3, A7- 20, A7- 35
VOIDT A7- 3, A7- 20, A7- 35
VOLUME 7- 34, 7- 35

WAIT statement 7- 21
WAITUPDATEIO 4- 41
WARN A7- 3, A7- 12, A7- 13, A7- 36
<while-expression> 7- 10, 7- 11, 7- 13
<write-statement> . . . . . 7-46- A
WRITE statement . . . . . 7- 40, 7-46- A, 7-46- B, 7-46- C, 8- 28, 10- 8

XREF A7- 3, A7- 36

<zero-suppression> 4- 17
<zero-suppression-character> 4- 14, 4- 17