In 1965 ECMA set up a new technical committee TC16 with the task to study the report "Specifications for the New Programming Language" issued in April 1964 by the Advanced Language Development Committee of SHARE and to consider the suitability of this language as a candidate for standardization. Based on this first study, ECMA decided in November 1968 to proceed with the standardization of the new language named PL/I.

In 1970 ANSI too set up a technical committee X3J1 for PL/I. It was decided that the two committees will work in common on a Joint PL/I Standardization Project. ECMA was entrusted with the secretariat of this Joint Project. The 4th revision of the ECMA draft was then issued as a common ECMA/ANSI draft.

In 1975 the Joint Project distributed a final draft (Basis 1-12) to the public with a request for comments. Numerous answers were received from the DP community in the USA and from several Member Bodies of ISO/TC97/SC5. These comments were taken into consideration as far as possible when preparing the final text of the present Standard.

The text of the technical part of this Standard ECMA-50 is identical to that of the corresponding part of standard ANSI X3.53-1976. Co-operation between ECMA and ANSI is expected to continue on future work related to PL/I and to the maintenance of the Standard.

This Standard ECMA-50 has been accepted by the General Assembly of December 16, 1976.
THIS STANDARD IS A REFERENCE DOCUMENT DEFINING THE FULL PL/1 LANGUAGE.

IT WILL BE THE BASIS FOR THE DEFINITION OF SUB-SETS, WITH THE TWIN OBJECTIVES OF YIELDING PRODUCTS WITH A MORE EFFECTIVE PERFORMANCE AND ALLOWING DEVELOPMENT OF CONFORMANCE TESTS, WHICH WOULD BE LESS DIFFICULT TO IMPLEMENT THAN FOR THE FULL LANGUAGE.

THE DEFINITION OF PL/1 SUB-SETS IS IN THE PROGRAM OF WORK OF ECMA.
## Contents

### CHAPTER 1: SCOPE AND OVERVIEW
1.0 Scope

1.1 An Informal Guide to the PL/I Definition
1.1.1 A Summary of PL/I
1.1.2 The Form of the Definition
1.1.3 Summary of Chapter Structure
1.1.4 Introduction to the Metalanguage
1.2.1.4.1 Tree Concepts
1.2.1.4.2 Syntaxes
1.2.1.4.3 Algorithmic Concepts

1.2 Relationships between an Implementation and This Definition
1.2.1 Flexibility of Interpretation
1.2.1.1 Rejection of Programs
1.2.1.2 Quantitative Restrictions
1.2.1.3 Operating Environment
1.2.1.4 Expression Evaluation
1.2.1.5 Intervals and Assignments
1.2.1.6 Input/Output
1.2.1.7 On-units
1.2.2 Implementation-defined Features

1.3 The Metalanguage

1.3.1 Trees
1.3.1.1 Tree Definitions
1.3.1.2 Node Objects
1.3.1.2.1 Uniques-maxes
1.3.1.2.2 Types
1.3.1.3 Node Rotation
1.3.1.4 Tree Notation
1.3.1.4.1 Emanated Trees
1.3.1.4.2 Forms
1.3.1.5 Tree Copies

1.3.2 Production Rules
1.3.2.1 Production Rules and Syntaxes
1.3.2.2 Complete and Partial Trees
1.3.2.3 Syntactic-expressions and Syntactic-units
1.3.2.4 Application of the Production rules

1.3.3 Operations
1.3.3.1 Nature of an Operation
1.3.3.2 Deterministic Operations
1.3.3.3 Format of an Operation
1.3.3.8 Instructions
1.3.3.5 Control
1.3.3.6 Additional Notational Conventions
1.3.3.7 Arithmetic

1.3.4 The Processor

1.3.5 Mechanization of the Metalanguage

1.4 Initialization of the Machine-state

1.4.1 The Machine-state
1.4.2 Initialization
1.4.3 The Top-level Operations
1.4.3.1 Define-program
1.4.3.2 Translational-phase
1.4.3.3 Interpretation-phase

### CHAPTER 2: CONCRETE SYNTAX
2.0 Introduction

2.1 The Concept of this Definition
2.3.1 Concrete and Abstract Syntaxes
2.2 Organization of the Concrete Syntax
2.3 The High-level Syntax of PL/I

2.3.1 Procedure
2.3.2 Unit
## CHAPTER 6: FLOW OF CONTROL

### 6.0 Introduction

### 6.1 Program Activation and Termination

#### 6.1.1 Program Termination

#### 6.1.2 Exit-step statement

#### 6.1.3 Step-program

### 6.2 Block Activation and Termination

#### 6.2.1 Activate-procedure

#### 6.2.2 Activate-block

#### 6.2.3 Epilogue

### 6.3 Control within a Block

#### 6.3.1 Normal-sequence

#### 6.3.2 Advance-execution

#### 6.3.3 Execute-begin-block

#### 6.3.4 Execute-group

#### 6.3.4.1 Establish-controlled-group

#### 6.3.4.2 Initialize-spec-options

#### 6.3.4.3 Test-spec

#### 6.3.4.4 Establish-next-spec

#### 6.3.4.5 Test-termination-of-controlled-group

#### 6.3.5 Execute-iff-statement

#### 6.3.5.1 Establish-truth-value

#### 6.3.6 Execute-call-statement

#### 6.3.6.1 Entry-references

#### 6.3.6.2 Evaluate-entry-reference

#### 6.3.6.3 Establish-argument

#### 6.3.7 Execute-goto-statement

#### 6.3.7.1 Local-goto

#### 6.3.7.2 Local-control

#### 6.3.8 Execute-null-statement

#### 6.3.9 Execute-return-statement

### 6.4 Conditions and Interrupts

#### 6.4.1 Conditions

#### 6.4.1.1 Raise-condition

#### 6.4.1.2 Test-enable

#### 6.4.1.3 Evaluate-signal-relationship

#### 6.4.1.4 Evaluate-masked-to-condition

#### 6.4.2 Interrupts

#### 6.4.2.1 Execute-on-statement

#### 6.4.2.2 Execute-revert-statement

#### 6.4.3 Interrupt

#### 6.4.4 System-action

#### 6.4.4.1 Comment

## CHAPTER 7: STORAGE AND ASSIGNMENT

### 7.0 Introduction

### 7.1 The Generation

#### 7.1.1 The Nature of Elements in the Storage-index-list of a Generation

#### 7.1.2 Correspondence between an Item-data-description and a Basic-value

#### 7.1.3 Value of a Generation

#### 7.1.4 Value of Storage Index

### 7.2 The Allocation of Storage

#### 7.2.1 Create-allocate-statement

#### 7.2.2 Allocate-controlled-storage

#### 7.2.3 Create-base-storage

#### 7.2.4 Evaluate-in-option

#### 7.2.5 Allocate

#### 7.2.6 Reallocate

#### 7.2.7 Evaluate-data-description-for-allocation

#### 7.2.8 Find-directory-entry

### 7.3 Initialization

#### 7.3.1 Initialize-generation

#### 7.3.2 Initialize-scalar-element
### 8.3.3.4.3 Make-name-and-subscript-list

- 8.7.3.4.4 Expand-name-and-subscript
- 8.7.3.4.5 Subscript-to-comma-subscript
- 8.7.3.4.6 Identifier-to-dename
- 8.7.3.5 Establish-next-format-item
- 8.7.3.6 Evaluate-format-item
- 8.7.3.6.1 Evaluate-format-expression

### CHAPTER 9: EXPRESSIONS AND CONVERSION

9.0 Introduction

9.1 Aggregate Expressions
- 9.1.1 Scalar and Aggregate Types
- 9.1.2 Aggregate Type of a Data Description
- 9.1.3 Scalar Elements
- 9.1.4 Construct 
- 9.1.5 Correspondence
- 9.1.5.1 Correspondence of Scalar Elements
- 9.1.5.2 Correspondence of Data Types
- 9.1.5.3 Generate-aggregate-result

9.2 Integer Type
- 9.2.1 Evaluate-expression-to-integer

9.3 Derived Data Types
- 9.3.1.1 Derived Base, Scalar, and Field
- 9.3.1.2 Converted Precision
- 9.3.1.3 Derived String Type
- 9.3.1.4 Further Definitions for Character and Bit Strings

9.4 Arithmetic Results
- 9.4.1 Conditions in Expressions
- 9.4.2 Expression
- 9.4.3 Value Reference
- 9.4.4 Constants
- 9.4.5 Parenthesized Expressions
- 9.4.6 Arguments

9.5 Prefix Operators
- 9.5.1 Prefix Expressions
- 9.5.2 Definition of the Prefix Operators
- 9.5.2.1 Prefix-silman
- 9.5.2.2 Prefix-tno
- 9.5.2.3 Prefix-plus

9.6 Infix Operators
- 9.6.1 Infix Expressions
- 9.6.2 Definition of the Infix Operators
- 9.6.2.1 Infix-add
- 9.6.2.2 Infix-and
- 9.6.2.3 Infix-cat
- 9.6.2.4 Infix-divide
- 9.6.2.5 Infix-eq

9.7 Examples and Terms

9.4.1 Built-in-function Reference

9.4.2 Special Terms Defined for Built-in-functions

9.4.3 Operations Used in Built-in-function Definitions

9.5 Special Functions

9.6.1 Ahm-bif

9.6.2 Acos-bif

9.6.2.1 Ahm-bif

9.6.2.2 Acos-bif
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add-bif</td>
<td>-</td>
<td>317</td>
</tr>
<tr>
<td>Addr-bif</td>
<td>-</td>
<td>318</td>
</tr>
<tr>
<td>Alloc-bif</td>
<td>-</td>
<td>319</td>
</tr>
<tr>
<td>Alloc-bif</td>
<td></td>
<td>320</td>
</tr>
<tr>
<td>Alloc-bif</td>
<td></td>
<td>321</td>
</tr>
<tr>
<td>Alloc-bif</td>
<td></td>
<td>322</td>
</tr>
<tr>
<td>Alloc-bif</td>
<td></td>
<td>323</td>
</tr>
<tr>
<td>Alloc-bif</td>
<td></td>
<td>324</td>
</tr>
<tr>
<td>Alloc-bif</td>
<td></td>
<td>325</td>
</tr>
<tr>
<td>Alloc-bif</td>
<td></td>
<td>326</td>
</tr>
<tr>
<td>Alloc-bif</td>
<td></td>
<td>327</td>
</tr>
<tr>
<td>Alloc-bif</td>
<td></td>
<td>328</td>
</tr>
<tr>
<td>Alloc-bif</td>
<td></td>
<td>329</td>
</tr>
<tr>
<td>Alloc-bif</td>
<td></td>
<td>330</td>
</tr>
<tr>
<td>Alloc-bif</td>
<td></td>
<td>331</td>
</tr>
<tr>
<td>Alloc-bif</td>
<td></td>
<td>332</td>
</tr>
<tr>
<td>Alloc-bif</td>
<td></td>
<td>333</td>
</tr>
<tr>
<td>Alloc-bif</td>
<td></td>
<td>334</td>
</tr>
<tr>
<td>Alloc-bif</td>
<td></td>
<td>335</td>
</tr>
<tr>
<td>Alloc-bif</td>
<td></td>
<td>336</td>
</tr>
<tr>
<td>Alloc-bif</td>
<td></td>
<td>337</td>
</tr>
<tr>
<td>Alloc-bif</td>
<td></td>
<td>338</td>
</tr>
<tr>
<td>Alloc-bif</td>
<td></td>
<td>339</td>
</tr>
<tr>
<td>Alloc-bif</td>
<td></td>
<td>340</td>
</tr>
<tr>
<td>Alloc-bif</td>
<td></td>
<td>341</td>
</tr>
<tr>
<td>Alloc-bif</td>
<td></td>
<td>342</td>
</tr>
<tr>
<td>Alloc-bif</td>
<td></td>
<td>343</td>
</tr>
<tr>
<td>Alloc-bif</td>
<td></td>
<td>344</td>
</tr>
<tr>
<td>Alloc-bif</td>
<td></td>
<td>345</td>
</tr>
<tr>
<td>Alloc-bif</td>
<td></td>
<td>346</td>
</tr>
<tr>
<td>Alloc-bif</td>
<td></td>
<td>347</td>
</tr>
<tr>
<td>Alloc-bif</td>
<td></td>
<td>348</td>
</tr>
<tr>
<td>Alloc-bif</td>
<td></td>
<td>349</td>
</tr>
<tr>
<td>Alloc-bif</td>
<td></td>
<td>350</td>
</tr>
<tr>
<td>Alloc-bif</td>
<td></td>
<td>351</td>
</tr>
<tr>
<td>Alloc-bif</td>
<td></td>
<td>352</td>
</tr>
<tr>
<td>Alloc-bif</td>
<td></td>
<td>353</td>
</tr>
<tr>
<td>Alloc-bif</td>
<td></td>
<td>354</td>
</tr>
<tr>
<td>Alloc-bif</td>
<td></td>
<td>355</td>
</tr>
</tbody>
</table>
Table 8.1 Concrete Terminals of Significance to Test-generic-description

Table 8.2 Table of <bit-values> as a Function of <symbols> and <prefix-factors> for Create-constant

Table 9.1 Table of Converted Precision as a Function of Target and Source Attributes

Table 9.2 Table of <numeric-picture-field> as a Function of <suppression-type> for Real-bif

Table 9.3 Table of <symbols> as a Function of <numeric-picture-element> and <character-string-values> for Edit-numeric-picture-field

Table 9.4 Table of <symbols> as a Function of <numeric-picture-element> and <character-string-values> for Edit-numeric-picture-field
Chapter 1: Scope and Overviews

1.0 Scope

This document defines the computer programming language PL/I. It is intended to serve as an authoritative reference rather than as a tutorial introduction.

The definition is accomplished by specifying a conceptual PL/I machine which translates and interprets intended PL/I programs. Section 1.1 provides a brief introduction to the statements and data types included in the language, to the structure and use of the document, and to the method of definition. The relationship between an actual implementation and the conceptual machine is described in Section 1.2, and the detailed specification of the notation to be used follows in Section 1.3. The main body of the definition is then begun at Section 1.4, and is completed by Chapters 2 through 9.

1.1 An Informal Guide to the PL/I Definition

1.1.1 A SUMMARY OF PL/I

A PL/I program consists of a set of procedures, each of which is written as a sequence of statements. The \texttt{PROCEDURE} construct may be used to include text from other sources during program translation.

All of the statement types are summarized here in prosigns which are presented as a means of obtaining an overview of the language and which may be related to the organization of the document.

For example:

| Structural: | PROCEDURE |
|            | ENTRY     |
|            | BEGIN     |
|            | DO        |
|            | END       |
| Declarative: | DECLARE   |
|            | DEFAULT   |
|            | FORMAT    |
| Flow of Control: | CALL     |
|                | RETURN   |
|                | IF       |
|                | GO TO    |
|                | Null Statement |
|                | STOP     |
|                | ON       |
|                | REVERSE  |
|                | SIGNAL   |
| Storage:      | ALLOCATE |
|                | FREE     |
|                | Assignment Statement |
Input/Output:

- **OPEN**
- **CLOSE**
- **GET**
- **PUT**
- **READ**
- **WRITE**
- **LOCATE**
- **REMOTE**
- **DELETE**

**Stream I/O**

**Record I/O**

Names may be declared to represent data of the following types, either as single values, or as aggregates in the form of arrays or structures:

- **Arithmetic**
- **Character**
- **Picture**
- **Bit**
- **Area**
- **Entry**
- **File**
- **Format**
- **Label**
- **Offset**
- **Pointer**

Values may be computed by expressions written using a specific set of operators and built-in functions, most of which may be applied to aggregates as well as to single values, together with user-defined procedures which, likewise, may operate on and return aggregate as well as single values.

1.1.1 THE FORM OF THE DEFINITION

The conceptual PL/I machine is a processor which has a set of operations acting on information stored in its memory. The operations are specified as algorithms, and may be viewed as the component parts of one single algorithm, "define-program", which carries out the entire translation and interpretation process. The information in memory is all held in the form of tree structures.

The definition algorithm operates as follows:

Sequences of symbols which are intended to represent PL/I external procedures (i.e., procedures not contained in any other procedure) are processed by a Translator. This processing involves systematically analysing, transforming, and validating each external procedure. The analysis uses a grammar known as the Concrete Syntax to produce the concrete form of an external procedure as a tree structure. This is transformed to the abstract form, which is a tree satisfying the Abstract Syntax, designed to be more convenient for interpretation. Further validation is then carried out on the abstract form. The Translator finally performs some validation of the mutual consistency of the set of external procedures which comprise the complete program.

The semantics of the program when applied to given initial datasets (i.e., collections of data) are then provided by an Interpreter. The datasets are part of the PL/I machine's memory, which is a tree satisfying the Machine-state Syntax, and it is the sequence of changes in the datasets which constitutes the defined semantics.

In addition to translating and interpreting all programs which are valid according to this definition, the PL/I machine detects as non-standard all those which violate a requirement involving the words "must" or "must not" in the algorithms performed during translation or interpretation. The implications for an actual implementation are discussed in Section 1.2.
The method of definition may be seen in outline as follows:

**Symbol-Lists**
**Representing**
**External**
**Procedures**

**→** Translators

**→** Program Tree (Abstract Form)

**→** Interpreter

**→** Datasets

It may also be helpful to visualise the abstract machine on which both the Translators and the Interpreter are "executed":

**Symbol-Lists**

**→** Datasets

**Memory**

(Changeable Machine-State Tree)

**Operations**

(Unchangeable Set of Algorithms)

The inputs from outside the machine occur at initialization of the Translators and Interpreter; the datasets may change during interpretation. However, there are no outputs defined since the datasets are treated for the purposes of this definition as being a part of the storage of the machine, i.e. as being "on-line" when needed.

1.1.3 Summary of Chapter Structure

| 1. Top-Level of Machine-State and Operations |
| 2. Concrete Syntax |
| 3. Abstract Syntax |
| 4. Translator |
| 5. Interpretation-State + Top-Level of Interpreter |
| 6. Flow of Control  | Concerned with the |
| 7. Storage and Assignment  | three parts of the |
| 8. Input/Output  | Interpretation-State |
| 9. Expressions  | Common Subroutines |
| For Chapters 5-8  |  |

The operations of Chapter 1 serve to drive the Translators and Interpreter.

The operations of the Translators are all contained in Chapter 4, and use the syntaxes of Chapters 2 and 3.

The operations of the Interpreter comprise all the operations in Chapters 5-9. After the initialization in Chapter 5, the relevant operations will be in Chapters 6, 7, or 8 depending on the type of statement being interpreted. All of these chapters invoke operations in Chapter 9 where necessary.
Within each chapter, the sections are logically organized, and the Table of Contents may be used to obtain an overview of the structure.

All readers are recommended to acquire a good understanding of Chapter 1 in its entirety. Thereafter, it is possible to read the definition as a systematic whole, or to use the document to locate answers to specific questions by combining an appreciation of the overall structure of the definition with judicious use of the index. To illustrate this latter usage, we consider each chapter in turn together with a sample question answerable from it.

Chapter 2 contains the definition of the Concrete Syntax. The Concrete Syntax consists of rules describing valid forms of PL/I constructs in concrete tree form. The syntax is permissive in the sense that some of the constructs permitted are being syntactically correct may later be found to be meaningless.

QUESTION
Is the following statement correct?
GET LIST (A(I,J)) DO I = 1 TO N, N;

ANSWER
The first possibility is that there may be an error according to the Concrete Syntax. The index entries for "get-statement" lead to the Middle-level Syntax, and study of rules CR103, 111, 112, 122, and 123 reveals that an extra pair of parentheses is required around the form {input-target-constant} DO {do-spec}.
This is in order to resolve the ambiguity exhibited in this example. The correct form is either
GET LIST (A(I,J)) DO I = 1 TO M, M;
or GET LIST (A(I,J)) DO I = 1 TO N, N;

depending on whether the last input value is intended to be assigned to M or to N.

Chapter 3 contains the definition of the Abstract Syntax. Many parts of the Abstract Syntax description intentionally bear a strong resemblance to the corresponding parts of the Concrete Syntax. Names in the Abstract Syntax have been chosen to resemble those of corresponding parts of the Concrete Syntax in order to make obvious as far as possible the relationship between the syntaxes.

QUESTION
May the KEYTO option on a READ statement specify that the key be assigned to a substring of a variable?

ANSWER
The Concrete Syntax for a {keyto-option} shows merely that a {reference} must be specified. However, the Abstract Syntax shows the form of a program after the Translator has completed all declarations, and has thus been able to associate each reference with the appropriate declaration and make some subtle distinctions. The rule All7 for {keyto-option} shows "<target-reference> {scalar & character}"., and All9 shows that this permits a {pseudo-variable-reference}, e.g. the $X$ pseudo-variable. The parenthesized constraint "{scalar & character}" shows that it must be a single target (not an array or structure) which is character-valued.

No further restrictions are found by using the index to search the Translator and Interpreter, so that the answer is: yes, provided the substring is a scalar of character type.
Chapter 4 defines the Translator whereby each of the individual P&L program portions (external procedures) is translated from the submitted character string form to tree form and appended to the program tree. This process involves the parsing of each external procedure using the Concrete Syntax to obtain a concrete tree, insertion of missing options and completion of attribute sets in that concrete tree, conversion from that concrete tree to an abstract tree, and, finally, validation of the whole program. Once formed, the abstract tree is not modified.

QUESTION
What file is implied in PUT LIST(X);?

ANSWER
This seems at first sight as though it might be a semantic question about the <put-statement>. However, the Abstract Syntax shows that a <file-option> must be present in <put-file> (§120), and this means that if it was absent in the concrete form, it must have been supplied by the Translator if the statement was valid.

In fact, immediately after parsing the input, the Translator completes the concrete procedure in various ways, one of which is to insert the equivalent of FILE(SYSPRINT) into our <put-statement> (Step 2 of the operation complete-options, Section 4.3.1.1).

The reason that it has to be handled early in the Translator is that it will lead to a contextual declaration of the name SYSPRINT if our statement is not within the scope of an existing declaration for SYSPRINT. It is necessary to complete all declarations prior to execution in order to resolve references correctly.

Chapter 5 contains the definitions of the Machine-state Syntax, the initialization of the machine-state tree, and the starting of the interpretation process.

QUESTION
May the first procedure executed in a program have arguments passed to it?

ANSWER
For the initialization of execution, we consult Sections 1.4.3.3 and 5.3.1.

In Step 1 of operation interpretation-phase (Section 1.4.3.3) an <entry-values> is obtained from outside the definition. It designates the first procedure in the <program> to be activated. The "must not" condition specifies that the document gives no meaning to a program whose <entry-point> designated by the <entry-value> contains a <parameter-name-list>. (See Section 1.3.1.4 for the definition of "must").

Additionally, in Step 2 of operation interpret (Section 5.3.1), an <evaluated-entry-reference> containing only an <entry-values> and not an <established-argument-list> (see production rule M25 in Section 5.1.21) is used to activate the first procedure.

Thus, the passing of an argument-list to the first procedure would be an extension beyond the language defined in this document.
Chapter 4 describes the operations of the Interpreter affecting the flow of control through blocks, groups, and statements of the program. Normal flow of control consists of the execution of a list of executable units within a block. The definition also defines calling, parameter/argument matching and result returning, and abnormal flow of control caused by interrupts.

**QUESTION**

Is it permissible to REVERT a condition for which no ON statement has been executed?

**ANSWER**

This is a semantic question about interrupt handling. The operation execute-revert-statement (Section 6.4.2.1) deletes a member from the current «established-on-unit-list» if an appropriate one exists, otherwise it performs normal-sequencing to pass on to the next statement. Since the action of execute-on-statement is merely to append «established-on-unit» to such lists, it is clear that execute-revert-statement is indifferent to the absence of such actions. The answer to the question is that it is permissible and has no effect on the interrupt handling mechanism.

Chapter 7 defines the use of storage including the allocation, freeing and initialization of storage, and the referencing of variables and named constants. The assignment statement, aggregate assignment, and pseudo-variables, are also defined.

**QUESTION**

Does DECLARE AC50 INITIAL (0); lead to all 5 elements of A being set to 0 when A is allocated?

**ANSWER**

The allocation and initialization of storage is treated in Chapter 7.

The operation initialize-array (Section 7.3.3) iterates over Step 8 while not. Since it is in this case i = 1 (see Step 2) and n is initially 1 and is incremented in Step 8.7, the iteration is performed once only, thus initializing the first element A(i) only.

Chapter 9 deals with the transmission of data between external media and internal storage, including the opening and closing of files, stream and record transmission, and interrupts applicable to I/O operations.

**QUESTION**

Is DECLARE F RECCED; OPEN FILE(F) PRINS; valid?

**ANSWER**

Although most of the declarative structures is evident in the Abstract Syntax and checked by the Translator, this is not altogether true of file attributes since they may still be incomplete until the file is opened during execution. Therefore their valid combinations have to be tested also at this late stage.

The open operation (Section 8.5.1.3) shows that «open» implies «read» in Step 2, and that «read» and «record» cause the result «fail» to be returned from the attempt to open the file. Thus if the open-statement is executed, it will lead to the raising of the UNDEFINED_FILE condition when an attempt is made to open the file (see Step 6 of execute-single-opening, Section 8.5.1.3).
Chapter 9 describes the evaluation of expressions, also conversions between data types, which can take place in expression evaluation, and the built-in functions of PL/I.

QUESTION

Is it possible to add an array to a structure?

ANSWER

The evaluation of expressions in Chapter 9 begins by considering the general treatment of aggregate operands. The definition of compatibility in Section 9.1.4 leads us to Case 3, yielding the answer that the array must be such that an element of it is compatible with the given structure. Applying the test for compatibility again, this would be true if the element is a scalar (Case 1) or a compatible structure (Case 2 and further recursions).

1.1.4 INTRODUCTION TO THE METALANGUAGE

An informal discussion of the main features of the metalanguage is now offered before proceeding to its more rigorous definition in Section 1.3.

1.1.4.1 TREE CONCEPTS

The definition of PL/I deals with three classes of trees: concrete parse, abstract text, and machine-state, and a uniform concept of tree is employed throughout. This is of a tree which is a directed graph with a label (e.g., <procedure>) at each node, and where the subtrees of any node are ordered. (Although the ordering is often irrelevant, it is needed, e.g., in the concrete parse and in lists, and it was decided that the simple uniformity of concept outweighed the advantages of explicitly distinguishing instances where the order was significant.) Moreover, each node implicitly has a unique-name by which it can be "designated" when required. A copy of the tree has the same ordered structure and labelling, but new unique-names to designate its nodes. Equality of trees requires equality of all except the unique-names of nodes, and identity requires that these also match.

The explicit label of a node may be either a grammatical category-name, or some other type of value such as an integer or a designator (i.e., a copy of the unique-name implicitly associated with some node). Thus a single value may be handled as a degenerate tree with only a root-node labelled with this value, and data objects referenced in the metalanguage are uniformly regarded as trees.

The terminology applied to trees is developed from the starting-point of a tree consisting of a node (the root-node) and an ordered, possibly empty, set of immediately contained trees.

A tree, X, is said to be contained in a tree, Y, if X is immediately contained in Y, or if it is immediately contained in some tree contained in Y. X is simply contained in Y if and only if it is contained in Y and it is not contained in any tree, Z, contained in Y and having a root-node equal to that of X or of Y. For example, if we refer to an <expression> simply contained in a tree representing some statement in a program, we mean the complete <expression> tree and not some subexpression which might be a tree rooted in <expression> at some lower level within it. Since simple containment is a frequently required concept, the use of any form of possessive phrase not employing the verb "contains" or the noun "component", is taken to imply simple containment, e.g., "if y has an <expression>" (meaning a tree with root-node labelled <expression>), "the <expression> of y", or "its <expression>".
The terminology makes it possible to perform the three essential types of operation on a tree, namely, to test them for the presence or type of their subtrees, to select subtrees from them, and to construct or alter them. However, constructing can be laborious if phrased as "root-node <r> with two immediate subtrees, the first being <l> with immediate subtrees <<c> and <d>, and the second being x" (where x is the name of some tree), meaning that a copy of it is to be made in constructing the new tree. This may therefore be abbreviated as

```
<r>
<l>
<c>
<d>
```

The indentation is inessential, although helpful with large trees. One or more trailing semicolons at the end of such a constructed tree may optionally be replaced by a period.

### 1.1.6.2 Syntaxes

Proceeding from consideration of particular trees to classes of trees, we encounter grammars composed of sets of production-rules couched in a slightly extended Backus-Naur Form. The interpretation we make of such rules is that they describe the structure of a tree belonging to the class described by the grammar - it is only in the traditional syntactic use of these rules that the sequence of terminal nodes of a tree acquires a special importance as being the sequence of characters representing an utterance in the language.

As an illustration, we will construct a tree that conforms to a rule of the form

```
<q/>1? <q/>1(<r-list>)<q/>1 <q/>1<q/>1
```

A grammatical rule in the metalanguage is rather like a statement in a language which it may be used to define, i.e. a metalanguage rule itself has a grammatical structure which has to be validated and correctly interpreted. We have chosen to escape from circularity or regress at this point by not giving a formal grammar for the metalanguage, but describing in prose how a rule is to be interpreted. So in the above illustration, the rule indicates that any node labelled <q/> must have subtrees as described by the right-hand side of the rule. There are three types of metalanguage operator, for concatenation (indicated by juxtaposition in the rule), permutation ("*"), and choice between alternatives ("+"), in descending order of precedence. Parenthesized expressions are regarded as simple operands, and the first stage in using a rule to construct a tree is to partition the syntax expression into subexpressions separated by "","1", the operator of lowest precedence, and to choose one of these alternative subexpressions. Suppose we discard the alternative of having <q/> alone, and next partition the other alternative by the "*" operator and decide to use the permutation which reverses these partitions to yield

```
<q/>1<q/>1(<r-list>)<q/>1
```

The brackets indicate that their contents may be optionally omitted, which we choose to do here, and the braces enclose a syntax expression which must be interpreted according to the method just described. Choosing the alternative <r-list> we would then have the tree

```
<q/>
<r-list>
```

The underlining of t indicates that it always labels a terminal node in the grammar in question, i.e. it does not occur on the left-hand side of any production-rule. <r-list> is dedicated by convention that it is to have one or more immediate subtrees of type <r> beneath it. We then apply the rule for <r> separately to each of these. The construction is completed when each terminal node of the tree is either of a type which is always terminal or of a type which has produced an empty set of subtrees.
1.1.6.3 Algorithm Concepts

The operations of the abstract machine are defined by algorithms, which may be compared with the logic or microcode supporting the operation codes of a computer. These operations inspect the machine-state or change the machine-state, or both. i.e. the memory of the abstract machine, which contains all the information directly or indirectly affecting semantics, including some form of the program itself. The metaroots "<" and ">" are used in the P/L/I definition of the machine-state nodes, except for those belonging to the grammar of the abstract program which are distinguished by "<" and ">" to help the reader.

An operation of the machine is defined by a sequence of steps, or a set of mutually exclusive cases, numbered from 1 to n. The i'th Step or Case may itself contain a sequence of steps or a set of cases numbered from i+1 to i.n and so forth. Each Step or Case specifies actions to be performed, using various informal "statement types" in the metalanguage. A Case begins by stating the predicate that must be satisfied for it to be applicable.

6.2.3 PROLOGUE

This operation is invoked at the beginning of every block activation to establish the (automatic) and (defined) variables local to that block. The (automatic) variables are initialized if their (declaration) specify initialization. Any (expression) evaluated during the prologue, such as in (constant-expression) or (expression) in (initial), are not allowed to reference other (automatic) or (defined) variables local to this block. The operation (void)-directory-entry will impose the restriction when it finds a reference to a variable declared in a block for which there exists a (prologue-flag). The (prologue-flag) is only present while the operation prologue is active.

Operation: prologue

Step 1. Attach a (prologue-flag) to the current (linkage-part).

Step 2. For each (declaration), d, of the current block, that contains (automatic) or (defined), perform step 2.1.

Step 2.1: Let ld be the (identifier) immediately contained in d, and let dd be the (data-description) immediately contained in the (variable) of d. Perform evaluate-data-description-for-allocation(dd) to obtain an (evaluated-data-description), edd.

Case 2.1.1: d contains (automatic).

Step 2.1.1.1. Perform allocate(edd) to obtain a (generation), g.

Step 2.1.1.2. Append to the current (automatic-directory-entry-list) an (automatic-directory-entry); id g.

Case 2.1.1.3. If d contains (initial) then perform initialize-
generation(g,d).

Case 2.1.2: d contains (defined).

Step 2.1.2. Append to the current (defined-directory-entry-lists) a (defined-directory-entry); id dd.

Step 3. Delete the (prologue-flag) of the current (linkage-part).

Step 4. Replace the current (statement-control) by a

(statement-control);
(operation-list);
(operation) for advance-execution.

Example 1.1: The Prologue Operation.
As an example of an operation, consider the definition of the prelude operation in Example 1.1. This begins with an introductory paragraph which gives some guidance to the reader, but the definitive material is not reached until the "Operation" heading.

Step 1 consists of an attach action, meaning that one tree is to be copied as a subtree of another, in the position in which it may validly occur according to the syntactic rules governing the type of the letter. The word "current" has been defined precisely with respect to the particular machine-state used in defining FL/I - the current linkage-part is the <linkage-part> simply contained in the <block-state> corresponding to the block currently being executed. Step 2 indicates iteration with a for each specification applied to the first form of the perform action, which refers to one or more steps in the same operation. A name such as d introduced after a comma is a variable, local to the operation, whose value (v, say) is a designator of the tree mentioned immediately preceding the v, i.e. it means "the tree designated by v" except when it is redefined in a way similar to its original definition (as would happen here on the next iteration of "for each"), or by its occurrence after the word "let". The if statement is merely a more explicit syntax for this kind of definition, and Step 2.1 contains an instance of it. If d contains <automatic>, the predicate of Case 2.1.1 is satisfied and Steps 2.1.1 to 2.1.3 will be performed in sequence.

The second form of perform, which is used to invoke another operation as a subroutine or function, occurs in Step 2.1.1.1. Here it is a function that is invoked "to obtain" a resulting value, which is then given a name. ed is passed as an argument to this operation, whose definition will name a parameter to correspond to t. Argument passing is by reference, i.e. a designator to the argument is passed and becomes the value of the parameter, which is then dereferenced on use, behaving just like a local variable.

The second action in Step 2.1.1.2 attaches a tree at the end of a list, and Step 2.1.1.3 exemplifies an if statement containing a subroutine call. Steps 3 and 4 consist of the other two tree actions which we use, delete and replace. When a tree is replaced by another, the first tree is deleted and a copy of the second tree is made with its root at the same position, and having the same implicit name, as had the root of the tree just deleted. Assignment of a value to a variable is permitted in the syntactic form of a set action, e.g. "Set tv to <true>", but this is only an alternative form of replacement with unconnected, implicit dereferencing of the variable name, meaning "replace the tree designated by the strict designator value of tv by <true>".

Other statements not illustrated here are go to (used sparingly), terminate an operation at some point other than the end of the last Step, and return a value from a function (which also terminates the operation). Values are returned by reference, so that if the result is a tree constructed within the function it must be copied back to the caller who will then receive a designator of the copy; values local to an operation are deleted when it terminates.

We now reach a point at which it is necessary to indicate a mechanism of the metalanguage which will suffice to bolster intuition in its weaker moments. Step 4 provides the case for this, since it constructs a tree which is to be placed in the current state at some point and then tails off into infamerity. Of the operation-lists in the <machine-state>, just one is said to be "active" and has a conceptual processor associated with it which may execute operations. In particular, the FL/I machine has a current <block-state> which has a <statement-control> where one could mechanize the actions of the metalanguage involved in interpreting the statements of the current block.

The <operation-list> is to be conceived as a push-down stack, where each operation will have a subtree which is not formally defined, but contains such information as the name of the operation, a list of its parameters and local variables with their current values; all trees constructed during execution of the operation or copied back to it from invoked functions, an indication of whereabouts in which Step or Case it is executing its algorithm, how far it has proceeded through a "for each" iteration, and so forth. When a "perform" invokes another operation, this pushes down the stack and becomes the active operation; when an operation terminates, its whole tree (including its local values) is deleted and the stack pops up, activity in the operation now at the head of the stack being resumed immediately after the point in its algorithm at which it was suspended.

To return to the analogy suggested at the beginning of this section, it is as though an abstract processor had a machine cycle which could cause the execution of microcoded operations with their own local memories per invocation, together with the sophistication of the stacking capability.
1.2 Relationships between an Implementation and this Definition

The inputs of the conceptual FL/I machine are one or more symbol-lists representing FL/I external procedures, an entry-value, and a dataset-list. This combination will be referred to as a program-run.

The standard definition of FL/I for a particular implementation is completed by defining (not necessarily in the style of this document) the implementation-defined features listed in Section 1.2.2, together with the representation of a program-run's elements (symbol-lists, entry-values, and dataset-lists) in the implementation's operating environment. With this information available, the conceptual FL/I machine gives one or more interpretations to a program-run.

The main purpose of this document is to define the semantics of interpreting valid FL/I program-runs. These semantics are constituted solely by the sequence of changes in the dataset-list of the machine-state, and an implementation is free to achieve these by any means. The operations and other parts of the machine-state which are the mechanisms used in this document to define the semantics need not be reflected directly in implementation.

An implementation's interpretation of a program-run conforms to the standard if and only if it conforms to one of the conceptual interpretations as follows:

1. If the conceptual interpretation rejects a program-run (via failure of a "nest" test) or if it never completes the translation-phase, then any interpretation by the implementation conforms. In particular, an implementation may or may not reject a program-run at the same point as it is rejected by the standard, or at all.

2. Otherwise, the implementation's interpretation conforms if it makes the same sequence of changes to datasets as does the conceptual interpretation.

3. The implementation's interpretation also conforms if it deviates from (1) and (2) only as permitted by the flexibilities of interpretation specified in Section 1.2.1.

Note that this implies that an implementation may provide extensions beyond the language defined in this standard, but is still required to conform for a program not using those extensions just as if the extensions were not available.

1.2.1 Flexibilities of Interpretation

Through use of the terms "optionally" and "in any order", the operation definitions of the conceptual FL/I machine permit most of the flexibility necessary for efficient implementation of FL/I. However, there are some rules that, if given formally with a metalanguage of the kind used here, would require constructions so elaborate as to impede understanding seriously. These rules are given in this section using informal language and making direct reference to possible actions of an implementation.

1.2.1.1 Rejection of Programs

If some part of a program is such that its interpretation would cause the program to be rejected, then an implementation may reject the program even if the conceptual interpretation does not reach the offending part.
1.2.1.2 Quantitative Restrictions

An implementation may make quantitative restrictions not contained in the standard. For example, restrictions may be made in the following contexts:

(1) Where syntax allows iterative or recursive constructs of arbitrary size, an implementation may restrict the size of these constructs provided the construct is not deleted. In particular, a standard implementation may limit the maximum length of an identifier, provided it is not less than 31 characters.

(2) The quantity of information existing during translation or execution may be restricted.

(3) The time permitted for interpretation of a program run may be bounded.

1.2.1.3 Operating Environment

An implementation may make restrictions at the interface with its environment, for example, in the composition of external identifiers or titles.

An implementation may require appropriate extralingual information in order to execute a program in conformance with this standard.

1.2.1.4 Expression Evaluation

The operators "evaluate-expression", "evaluate-variable-reference", and "evaluate-target-reference" define one or more strict orders of evaluation. An implementation may deviate from these strict orders in the following respects:

(1) Not all of the interrupts raised by any of the strict evaluations need be raised by an implementation.

(2) The order in which interrupts are raised by an implementation may differ from the order in which they are raised by strict evaluation.

(3) An implementation may evaluate an expression, variable-reference, or target-reference by evaluating its contained expression(s), value-reference(s), and variable-reference(s) in any order. Note that this does not permit the association of operators with their operands to be altered, since the association is fixed in the tree structure of an expression by the translator.

(4) If an implementation can determine the result produced by evaluating an expression, variable-reference, or target-reference without evaluating it, the implementation need not perform the evaluation. However, if the result depends on the value returned by a contained procedure-function-reference, then this procedure-function-reference must be evaluated.

When determining whether or not an evaluation must be performed, the implementation may ignore the possibility that this evaluation could raise one or more interrupts.

1.2.1.5 Interrupts and Assignment

The operation "execute-assignment-statement" determines a strict order of evaluation. An implementation may deviate from one or more strict orders in the manner and order in which it raises interrupts as described in Section 1.2.1.4, parts 1 and 2. When an implementation exceeds one or more of its implementation-defined limits, it may raise one or more of the following conditions: overflow, underflow, fixed-overflow, zerodivide, size, stringsize, stringrange, subscriptrange, storage, area, error.
The `<environment>` attribute and option are provided for the specification of implementation-defined information concerned with the manipulation of the `<file-values>` and `<dataset>`s. If present, an `<environment>` may affect the algorithms described in this standard by either causing (if necessary) the operation evaluate-expression to be performed or affecting the correspondence between open `<file-values>` and `<dataset>`s. If no `<environment>` is present, the algorithms work as given. An `<expression>` appearing in an `<environment>` is evaluated at an implementation-defined point.

(2) If there is more than one `<single-opening>` in an `<open-statement>`, an implementation may deviate from the strict order of execution of the raise-to-condition operation performed by the execute-single-opening operation. In particular, an implementation may defer the raise-to-condition operations until all other operations in the statement have been performed. The conditions must be raised in the same order as they would have been raised had the strict order of execution been followed.

(3) If there is a `<copy-option>` in a `<get-statement>`, an implementation may deviate from the strict order of execution of the output-string-item operation. In particular, an implementation may interleave the execution of this operation with the execution of any other operations in the `<get-statement>` which follow it. If there are two or more output-string operations, they must be executed in the same order with respect to each other as they would have been performed under the strict order of execution, independently of their order with respect to the other operations of the `<get-statement>`.

(4) During the execution of any input/output statement which contains a `<file-option>` or a `<copy-option>` an implementation may raise the `<transmit-condition>`. Continued execution of the program beyond the point where the condition is raised may be undefined depending on the use and validity of the data transmitted by the input/output statement. The `<transmit-condition>` is raised by performing raise-to-condition `<transmit-condition>`, file-value, key) where file-value and key depend on the input/output statement. In the event that an output operation is being executed as part of file closing or program epilogue and circumstances are such that the `<transmit-condition>` is to be raised, the implementation may perform an implementation-defined action.

(5) A `<get-string>` must not immediately contain an `<expression>` that simply contains a `<variable-reference>` that identifies an `<allocation-unit>` that is referenced by the evaluation of the `<input-specification>` of the `<get-string>`.

(6) A `<put-string>` must not immediately contain a `<target-reference>` that identifies an `<allocation-unit>` that is referenced by the evaluation of the `<output-specification>` of the `<put-string>`.
In order to avoid defining certain "side-effects" of \texttt{on-unit}s, and to avoid overly defining the state of the machine upon entry to \texttt{on-unit}s, a program must satisfy the following constraints not explicitly enforced by the PL/I machine:

1. An \texttt{on-unit}, or any of its dynamic descendants, entered for the underflow, conversion, or string spills condition must not allocate, free, or assign a value to any \texttt{allocation-unit} used in the block of interrupt, unless the \texttt{on-unit} terminates by executing a \texttt{goto-statement}.

Let $B$ be the current \texttt{block-state} at the time the interrupt was raised; i.e., the block of interrupt. An \texttt{allocation-unit} is used in block $B$ if it is referenced by any operation of the PL/I machine while $B$ is the current \texttt{block-state}.

2. Let $B$ be a \texttt{block-state}; let $E$ be an \texttt{executable-unit} executed while $B$ is the current \texttt{block-state}. During the execution of $E$, various \texttt{allocation-units} are returned by invocations of \texttt{evaln-target-reference}. Let $T$ be the set of \texttt{allocation-units} designated by these \texttt{execution}s, i.e., the targets of a given statement execution.

During the execution of $E$ whenever the "interrupt" operation is invoked, all \texttt{basic-values} contained in any member of $T$ are set \texttt{undefined}. If control reaches step $B$ of the "interrupt" operation, the original \texttt{basic-values} are restored, unless their containing \texttt{allocation-unit} was assigned one or more \texttt{basic-values} by the \texttt{on-unit} or its dynamic descendants. In the case of such assignment, the \texttt{basic-values} of the \texttt{allocation-unit} are set \texttt{undefined}.

1.3.2 IMPLEMENTATION-DEFINED FEATURES

The PL/I language features listed below are termed \textit{implementation-defined}; their specification is regarded as completing the definition of the language for a particular implementation. A brief description of each feature is given, with references in parentheses to the sections of this document where further details can be found.

1. Circumstances in which \texttt{TRANSMIT} condition is raised (1.2.1.4).

2. Actions performed, instead of raising the \texttt{RECORD}, \texttt{KEY}, or \texttt{TRANSMIT} condition, when an output operation is being executed as a part of a file closing during program epilogue and circumstances are such that, in all other contexts, the condition would be raised (1.2.1.4, 8.5.2.3, and 8.6.6.9).

3. Determination of the \texttt{dataset-list} passed to the "interpret" operation (1.4.3.3).

4. Determination of the \texttt{entry-value} passed to the "interpret" operation (1.4.3.3).

5. \texttt{ENVIRONMENT} attribute and option syntax (2.4.4.4) and semantics (Chapter 8).

6. \texttt{OPTIONS} attribute and option syntax (2.4.4.4) and semantics.

7. Extralinguistic characters in data character set (2.5.5 and 2.6.2).

8. The form of the \texttt{text-name} in the \texttt{INCLINE} construct (2.5.7).

9. Collating sequence, hardware representations, graphic representations, and symbol names of an implementation's character set (2.4 and 8.4.4.17).

10. Default precisions of arithmetic data (4.3.6.3).

11. Default \texttt{AREA} size (4.3.6.3).

12. Consistency requirements for \texttt{ENVIRONMENT} and \texttt{OPTIONS} attributes in \texttt{EXTERNAL} declarations (4.6.1), and for \texttt{OPTIONS} attributes in \texttt{ENTRY} references (4.3.4.1.3).

13. Size of an \texttt{area-value} passed as a dummy argument (4.3.6.1.3).
(14) Information output when SNAP is specified in CM statement (6.4.3).
(15) Value returned by ONCODE builtin function (6.4.3).
(16) Standard system action for STORAGE condition (6.4.4).
(17) Standard system action for ERROR condition (6.4.4).
(18) Form of comment output as standard system action (6.4.4.1).
(19) Situations when ERROR is raised.
(20) Situations when STORAGE is raised (7.2.5).
(21) Use of AREA size specification (7.2.6).
(22) Interpretation of $HUESPEC pseudo-variable (7.5.4.81).
(23) Concrete representation of a <dataset> (8.13).
(24) The "size" of a <record> (8.1.4).
(25) Length of a key (8.1.1).
(26) Representation of stream dataset control items (8.1.2).
(27) Determination of a <dataset> on file opening (8.5.1.3).
(28) Default LEADING for a STREAM OUTPUT file (8.5.1.3).
(29) Default PAGESIZE for a PRINT file (8.5.1.3).
(30) Default tab positions for a PRINT file (8.5.1.3).
(31) Length of file title (8.5.1.5).
(32) Circumstances in which the ERT condition is raised (8.5.2.3, 8.6.2.2, 8.6.3.1, and 8.6.5.10).
(33) Circumstances under which records are written, or not written, when the RECORD condition is raised and normal return occurs, and the values of those records (8.6.3.1, 8.6.4.2, and 8.6.6.9).
(34) Raising of RECORD condition by WRITE and LOCATE statements (8.6.6.8).
(35) Position of records in a KEYED SEQUENTIAL file (8.6.6.9).
(36) Items output by "PUT DATA:" (8.7.2.5).
(37) Maximum <number-of-digits> used in editing relative to a <fixed-point-format> (8.7.2.6.3).
(38) Maximum <number-of-digits> for each combination of <class> and <scale> (9.4.3.5, 9.5.3.2, 9.5.1.9, and elsewhere).
(39) Precision of integer-type (9.1.3.2).
(40) Determination of floating-point results of expressions and builtin functions (9.1.4).
(41) Results of ROUND builtin function with floating-point argument (9.4.4.48).
(42) Length of string returned by TIME builtin function (9.4.4.82).
(43) Result of $HUESPEC builtin function (9.4.4.85).
(44) Results of numeric conversions (9.5.1.2, 9.5.1.3, and 9.5.1.4).
(45) Number of digits in the exponent of a floating-point number (9.5.1.5).
(46) Representation of currency symbol and digit and sign symbols (9.5.2.2).
1.3 The Metalanguage

Following the introductory material in Section 1.1.4, this section now gives a more precise and careful definition of the metalanguage.

The definitive part of this document consists of:

- a set of production-rules
- a set of operations
- constraints
- attribute definitions and argument names
- tables
- definitions of terms

Together with the section describing the relationship between an implementation and this mechanized definition.

The metalanguage in which the definition is expressed has three main notational parts, to be presented later in this section:

- a notation for trees, the fundamental type of data in the metalanguage;
- a notation for production-rules, which define classes of trees;
- a notation for operations, which manipulate trees.

Other definitive material follows headings "Constraints", "Attributes", or "Arguments".

Tables are enclosed in a frame of straight lines.

At the point where a new term (other than a syntactic category) is defined, the term is underlined to indicate this; subsequent uses of the term are not underlined.

Examples, which are not part of the definition, appear in a frame with lines at the sides and asterisks at the top and bottom. Introductory paragraphs to sections and operations are likewise without any definitive force.

1.3.1 TREES

Trees are the sole type of data manipulated by the actions of the process defined by this document. All of the internal operations of the process use only trees, all of the inputs to the process from its environment are suitably constructed trees, and all interpretations of the semantics defined by this process must be in terms of the tree manipulations performed by it. For uniformity even simple values, such as numbers or characters, are regarded within the process as single node trees.

In a strict mechanization of the process defined by this document, there could in fact be only a single tree used to hold the entire "state" of the process, and all of the trees discussed here would be subtrees of this single tree. Since, however, this document leaves certain informal "gaps" in its tree definitions, it is also possible to regard the process as one that operates on a set of independent trees, one for the "state" and others which are more local to particular phases of the definition.

The general abstract form of trees as employed in this document plus the technical terms used in discussing trees are defined in Section 1.3.1.1. These trees are then made more specific by discussing in Section 1.3.1.2 the basic nature of the objects used in composing tree nodes. In Sections 1.3.1.3 and 1.3.1.4 the written notations used for individual nodes and then for whole trees are discussed. Section 1.3.1.5 deals with copies of trees.

Following this general section on trees and tree notations, Section 1.3.2 then discusses production-rules, which function in a declarative manner to specify the particular classes of trees used in this document.
1.3.1.1 Tree Definitions

A node is an ordered pair of objects, termed the type of the node, and the unique-name of the node.

A tree is a finite set of one or more nodes together with some structuring relationships among these nodes. These relationships are such that:

1. There is a specified node termed the root-node of the tree.
2. Excluding this root-node, the remaining nodes (termed the subnodes of the tree) are divided up into zero or more disjoint sets, each of which in turn forms a tree. These trees are termed the immediate subtrees of the defined tree.
3. There is a specified linear ordering among these immediate subtrees.

A tree, X, is said to be a subtree of a given tree, Y, if X is either:

1. an immediate subtree of Y, or
2. a subtree of an immediate subtree of Y.

A subtree, X, of a given tree, Y, is said to be a simple subtree of Y if there does not exist a tree, Z, such that:

1. Z is a subtree of Y,
2. Z is a subtree of X, and
3. the root-node of Z has the same type as either the root-node of X or the root-node of Y.

Terminology based on the word "contained" is used consistently as follows: subnodes and subtrees are said to be contained in the given tree, and immediate subtrees and simple subtrees are said to be immediately contained and simply contained respectively. Similarly for the word "component", i.e., subnodes and subtrees are said to be components of the given tree, immediate subtrees and simple subtrees are said to be immediate components and simple components respectively. The root-nodes of immediate subtrees and simple subtrees are said to be immediate subnodes and simple subnodes respectively.

The concept denoted by the term "simply contains" is used so frequently in the sequel that the words themselves are easily tedious. Any relational statement between two nodes that implies containment or possession, without using explicitly any form of the words "contain" or "component", is to be interpreted as implying the simple containment relation. For example, "X simply contains Y" may be expressed as "A with B", "A has B", "the B in the A", or even "A B". These abbreviated forms may also be compounded in a simple sentence.

Two trees are said to be equal if they contain the same number of nodes and if:

1. when they each contain a single node, their respective types are the same, or
2. when they each contain more than one node, the respective types of their root-nodes are the same, they have the same number of immediate subtrees, and their respective immediate subtrees, taken pairwise, are equal.

A tree, X, is said to immediately follow a tree, Y, if they are both immediate subtrees of some tree, Z, and if X is next after Y in the linear ordering of the subtrees of Z.

A tree, X, is said to immediately precede a tree Y if Y immediately follows X.

A tree, X, is said to follow a tree, Y, if any of the following is true:

1. X immediately follows Y,
2. There is a tree Z such that Z follows Y and Z follows X,
3. There is a tree Z such that Z contains Y and Z follows X,
4. There is a tree Z such that Z contains Y and Z follows X.
A tree, X, is said to precede a tree, Y, if Y follows X.

Note that the above definitions do not define a linear ordering on all of the subtrees of a tree, just a partial ordering. In particular, any tree which contains another does not either precede or follow the contained tree.

The words first and last applied to any distinct set of trees have their usual sense of "have none that precede" and "have none that follow". Although these definitions are not such as to give a unique first or last tree for some sets of trees, "first" and "last" will be applied in this document only to sets such that there is a unique result.

Similarly, the next tree following a given tree is defined in the usual sense of "first that follows" and will also be used in contexts where it is unique.

A node, M, contained in a tree, X, is said to correspond to a node, N, contained in a tree, Y, if M and N occupy the same ordinal position in the ordered set of immediate subtrees of either:

(1) the root-nodes of X and Y, or of
(2) corresponding nodes of X and Y.

(Chapter 9 (see section 9.1.1.5) contains further definitions of "correspond" useful in certain special contexts.)

1.3.1.2 Node Objects

The definition of node given in Section 1.3.1.1 leaves undefined the nature of the objects used for node types and node unique-names. The sets from which these objects are selected are limited as described in section 1.3.1.2.1 and 1.3.1.2.2.

1.3.1.2.1 Unique-names

The set of objects which may be employed as node unique-names plays two roles. All node unique-names are selected from this set, but in addition, some node types may be selected from this set. Any potentially infinite set of objects which are distinguishable from the other objects used as node types will suffice.

The unique-name component of a node is so called for two reasons. First, at no time do two distinct nodes have equal unique-name components. Second, no node in ever created with a unique-name component equal to that of any node which has ever been created. The unique-name component of a node does not change during the life of the node and then serves to identify, or designate, the node (see Section 1.3.1.2.2).

1.3.1.2.2 Types

The set of objects which may be employed as node types is the union of the following disjoint sets:

(1) The set of category-names. This is a finite set of the objects employed in production-rules. This set can be further subdivided into several logically coherent subsets, each with a definite notational convention.

(2) The set of real numbers, or possibly some implementation-dependent subset of them which includes the integers. The integers are used throughout for such purposes as indices and ordinates. Real numbers (including possibly integers) are used as the values of arithmetic variables.

(3) The set of unique-names, as defined in Section 1.3.1.2.1. A number of this set used as a node type is termed a designator. Designators are used explicitly for the purpose of uniquely picking out, or designating, nodes of a tree. A designator for a tree containing a single designator designates exactly that node which has the same object as unique-name. Note that this construction, together with the uniqueness rule in Section 1.3.1.2.1, means that it is possible to examine a designator and constructively determine if the potentially designated node has been in fact deleted.
As a general rule, the type of a node does not change during the life of the node. Modifications to the tree occur by removing old nodes and constructing new nodes with new types and new unique-names. The single exception to this is the replace instruction (see section 1.3.3).

1.3.1.3 Node Notation

Throughout the metalanguage, the unique-names of nodes have only an implicit and essentially invisible function in guaranteeing unique designation and proper subtree distinction. The metalanguage discussion of nodes is always in terms of their types. Particular objects from the set of unique-names are never referred to directly, so no written notation for them is required.

The written notation used for real numbers is just ordinary decimal notation throughout. In the sequel, context is sufficient to distinguish numbers used as node types from numbers used for other purposes.

The written notation used for category-names varies, depending on the logical nature of the use of the particular category-name. These various notations and their meaning is as follows:

1. **Named Categories.** Names formed of lower case letters, hyphens, and numbers, including surrounding brackets of the form "[" and "]", "<" and ">", or "<" and ">" are used as the denotation of some category-names. Optionally the name exclusive of the brackets may be underlined. In general, mnemonic English words or abbreviations are chosen to indicate the function of the category-name. In addition, the three types of brackets indicate whether the category-name functions primarily in the concrete, abstract, or interpretation phases of the definitional process respectively. The underlining, if present, indicates that the category-name occurs only as the type of a node that has no components.

2. **PL/I Characters.** The 57 characters of the PL/I language character set are used as category-names. Two notations are used. In the great majority of situations, where no confusion is liable to arise, they are denoted by straightforward individual character denotations. Capital letters are used for the letters, while the quoted symbol (i.e., that which is inside the following quotes) "E" is used for blank. In situations where confusion might arise, the concrete brackets are used around the straightforward denotations, e.g., §.

3. **PL/I Keywords.** Certain category-names represent PL/I keywords, i.e., selected sequences of letters or numbers that have particular significance in PL/I. Two denotations are used. In the great majority of situations where no confusion is liable to arise, they are denoted by a straightforward concatenation of the denotations for the individual letters or digits that form these keywords in PL/I, written without intervening spaces on the page. Examples are the quoted symbols "LEFT" and "FLOAT". In situations where confusion might arise, the concrete brackets are used around the straightforward denotations.

Nodes which have a type of either of the classes (2) or (3) above are said to possess a concrete-representation, which is a (non-tree) character string composed from any of the 57 PL/I language characters. For these nodes this concrete-representation is just the simple denotation of the node type, with a blank space substituted for E. Any possible subnodes of the category (relational-qualifier) are assumed to have a concrete-representation, each of which is different from that of any PL/I character.

Any tree that satisfies the restriction that all of its nodes which contain no components have a type which possesses a concrete-representation, may also be said to possess a concrete-representation. This representation is just the character string formed by concatenating, in precedence order, the concrete-representations of these nodes.
1.3.1.4 Tree Notation

1.3.1.4.1 Enumerated Trees

A particular tree may be completely specified by stating its root-node and describing each sub-node in terms of immediate components down to the terminal nodes.

This may be expressed more concisely as an enumerated-tree in a notation which specifies:

1. a node type, for the root-node, optionally followed by a name and a label by which this node can be designated (see Section 1.3.1.3),

   optionally followed by

2. a colon,

   the immediate components of the root-node (which may themselves be enumerated-trees, or may be names designating other nodes to be copied) which may be enclosed in brackets ([ ] and { }) denoting a component that is to be omitted if and only if its value is \texttt{absent},

   and a semicolon.

For example, the tree which consists of a \texttt{<data-type>} which immediately contains a \texttt{<non-computational-type>} which immediately contains \texttt{<format>} and \texttt{<local>}, can be written as:

\texttt{<data-type>,dt: \texttt{<non-computational-type>,<format>,<local>};.}

Indentation is often used as a visual aid, e.g.

\texttt{<data-type>,dt:
    \texttt{<non-computational-type>:
        \texttt{<format>\texttt{<local>};.}}

Semicolons at the end of an enumerated-tree specification may be omitted immediately before a period.

Use of an enumerated-tree in the metaslangage indicates the creation of a local-tree (see Section 1.3.3.1) having the structure and node types indicated, with appropriate copies inserted of the trees to be copied (see [2] above), and with the designators corresponding to the names optionally used as in (1) above not to designate the nodes there created.

A frequently used abbreviation for a particular class of enumerated-trees is to enclose a potential \texttt{C++} concrete-representation in double quotes. This is an abbreviation for that tree rooted in \texttt{<symbol-list>}, which has this enclosed string as its concrete-representation.

1.3.1.4.2 Forms

Patterns to be searched for in trees may be indicated in the metaslangage by a notation which is the same as that for enumerated-trees, except that the names of trees to be copied may not be included.

Use of such a notation in the metaslangage is always preceded by the word \texttt{form}. Its use indicates that a search for, or test of conformance to, the pattern is to be made, yielding true or false, and that the designators corresponding to the names used as in (1) of Section 1.3.4.1 are to be set to designate the nodes corresponding to those if and only if the search, or test, returns true. Use of brackets in a form indicates the bracketed components may be either present or absent.
1.3.1.5 Tree Copies

A copy of a given tree is constructed as follows:

1. Construct a tree which is equal to the given tree.
2. For each designator node X in the given tree which designates a node Y also in
the given tree, change the constructed node which corresponds to X so that it
designates the constructed node which corresponds to Y.

1.3.2 Production Rules

The trees actually employed in this document are a limited subset of all the possible
trees that could be formed according to the definitions given in Section 1.3.1.
Production rules serve as the declarative portion of the metalanguage and do so by
specifying restrictions on the forms assumed by the trees used in the definitional
process of this document.

1.3.2.1 Production Rules and Syntaxes

A production rule is written with an optional label formed of capital letters and digits,
and consists of a category-name, followed by the quoted symbol “:=" and then followed by
either a syntactic-expression (see Section 1.3.2.2) or, in a few instances, an English
Language phrase. Each production rule is termed a defining production rule for
that category-name written before the “:". Within this document, there is at most one
defining production rule for any given category-name.

The basic function of a production rule is to define a set of possibilities for the
number, type(s), and order of the immediate subnodes of a node whose type is the defined
category-name. This is done by interpreting the syntactic-expression of the production-
rule according to the algorithm given in Section 1.3.2.4.

Production rules are augmented in their function of specifying immediate subnodes by a
notational convention used for creating lists of repetitive immediate subnodes. This
convention applies to the bracketed category-names whose denotation, exclusive of the
brackets, terminates in the quoted symbols “-list” or “-comma-list”, and it substitutes
for the explicit appearance of a defining production rule for each category-name (i.e.
such category-names have no defining production rule written in this document).

A syntax is any set of production rules. For example, the set of all of the production-
rules in this document is a syntax. Five (disjoint) subsets of the production rules in
this document have enough logical coherence that they have been given names, i.e. the
High-level Concrete Syntax, the Middle-level Concrete Syntax, the Low-level Concrete
Syntax, the Abstract Syntax, and the Machine-state Syntax. In order to distinguish these
syntaxes, the production rules comprising them have been given numbered labels starting
with the quoted symbols “CH”, “CM”, “CL”, “A”, and “M” respectively. The unlabelled
production rules of this document do not belong to any of these syntaxes.

If a defining production rule for a category-name occurs in a syntax, then that category-
name is said to be non-terminal with respect to that syntax. Any category-name that
occurs somewhere within the syntactic-expressions of the production rules of the syntax,
but has no defining production rule in the syntax, is said to be:

1. non-terminal with respect to that syntax if its denotation exclusive of the
   brackets, ends with “-list”, “-comma-list”, or “-designator”, and
2. terminal with respect to that syntax otherwise.

A category-name that is non-terminal with respect to the syntax composed of all the
production rules occurring in this document is said to be just non-terminal; similarly
for terminal.

The Abstract Syntax additionally allows constraints to be specified for certain category-
names. The constraint, written in parentheses after the relevant category-name, is
applied by the Translator, or during the interpretation phase, but has no effect on the
constitution of a tree specified by the syntax.
The production-rule

A67. \text{<bound-pair> ::= <lower-bound> <upper-bound> | <asterisk>}

defines two possibilities, which may be written as (a) or drawn as (b)

(a) \quad \text{<bound-pair> :}
\quad \quad \quad \text{<lower-bound>} \quad \text{or} \quad \text{<upper-bound>}
\quad \quad \quad \text{<asterisk>}

(b) \quad \text{<bound-pair> :}
\quad \quad \quad \text{<lower-bound>}
\quad \quad \quad \text{or}
\quad \quad \quad \text{<upper-bound>}
\quad \quad \quad \text{<asterisk>}

A node whose type is \text{<identifier-list>} may have any non-zero number of immediate subnodes of type \text{<identifier>}, i.e.

\text{<identifier-list> : or <identifier-list> : or <identifier-list> :}
\text{<identifier> : or <identifier> : or <identifier> :}

and so on.

A node whose type is \text{<parameter-conalist>} may have any non-zero number of \text{<parameter>} immediate subnodes, but with nodes whose type is the Pl/i character $\{,\}$ interspersed between adjacent ones, i.e.

\text{<parameter-conalist> : or <parameter-conalist> : or <parameter-conalist> :}
\text{<parameter> : or <parameter> : or <parameter> :}
\text{\{,\} : or \{,\} : or \{,\} :}
\text{<parameter> : or <parameter> : or <parameter> :}

and so on.

The production-rule

A68. \text{<repeat-option> ::= <expression> \{scalar\}}

specifies that only \text{<expression>} yielding scalar values (i.e., not aggregate values) are valid immediate subnodes of a \text{<repeat-option>}.

Example 1.2. Examples of Syntax.

1.2.3.2 Complete and Partial Trees

Given a syntax, a complete tree with respect to that syntax is any tree which can be obtained by starting from a node of a given type, and repeatedly attaching subnodes to the nodes of the tree being developed according to the algorithm of Section 1.2.3.2, until an interpretation has been obtained for every node of the tree. A complete tree with respect to the syntax composed of all the production-rules occurring in this document is said to be just a complete tree.
A partial tree is any tree which is not a complete tree but which can be obtained by deleting some nodes from some complete tree.

The trees utilized by the definition process of this document are only complete trees or partial trees. Other possible forms of trees are never utilized. Furthermore, it is the usual case that complete trees are utilized, or at least utilized at the interfaces between the various operations of the definition. Although there are a few specific exceptions, it is a general rule that partial trees occur only in a very local context in the process of building up a complete tree. Several of the "instructions" (see Section 1.3.3.4) of the metalanguage are in fact designed to assist in the process of building complete trees.

1.3.2.1 Syntactic-expressions and Syntactic-units

Given a syntax, a syntactic-expression is defined to be either a single syntactic-unit, or several syntactic-units any of the adjacent pairs of which is possibly separated by a "i" or a "a". The symbols are called the or-symbol and the bullet respectively.

Given a syntax, a syntactic-unit is defined to be one of the following:

- a single category-name,

- a syntactic-expression enclosed in the brackets "a" and "a", or

- a syntactic-expression enclosed in the brackets "i" and "i".

1.3.2.2 Application of the Production Rules

Given a syntax and a category-name, the algorithm shown just below obtains (possibly empty) ordered set of category-names, termed here an interpretation with respect to the given syntax of the given category-name. In the process of constructing a complete tree with respect to the given syntax, any such ordered set may then be used as the corresponding types of an ordered set of immediate subnodes connected to any node whose type is the given category-name.

An interpretation of a category-name is defined as follows:

Case 1. The denotation of the given category-name, excluding any terminating bracket, ends with "i-link".

An interpretation consists of an ordered set of any non-zero number of instances of that category-name whose denotation is obtained by deleting the "i-link" from the denotation of the given category-name.

Case 2. The denotation of the given category-name, excluding any terminating bracket, ends with "a-comma".

An interpretation consists of an ordered set which:

1) contains any non-zero number, n, of instances of the category-name whose denotation is obtained by deleting the "a-comma" from the denotation of the given category-name, and which

2) contains n-1 instances of the category-name f, g, and which

3) is arranged so that no two instances of the same category-name are adjacent.

Case 3. The denotation of the given category-name, excluding any terminating bracket, ends with "a-designator".

An interpretation is a single member of the set of unique-names. If the category-name is of the form "a-designator", the unique-name must be of that of a node of type "a".

Chapter 1: Scope and Overviews 23
Case 5. (Otherwise).

The given category-name is a terminal with respect to the given syntax; the
interpretation is the empty set.

An interpretation of a syntactic-expression is defined as follows:

Case 1. The syntactic-expression is a syntactic-unit.

Case 1.1. The syntactic-unit is a single category-name.

An interpretation consists of the ordered set containing just this single
category-name.

Case 1.2. The syntactic-unit is a syntactic-expression enclosed in the brackets "["]
and "]".

An interpretation consists either of an interpretation of the enclosed
syntactic-expression, or of the empty set.

Case 1.3. The syntactic-unit is a syntactic-expression enclosed in the braces "{" and
"}".

An interpretation consists of an interpretation of the enclosed syntactic-
expression.

Case 2. The syntactic-expression is a sequence of two or more syntactic-units possibly
separated by a "," or a ";".

Case 2.1. An or-symbol occurs at least once such a separator.

Consider all or-symbols occurring thus in the given syntactic-expression to
partition it into a sequence of inner syntactic-expressions. An
interpretation is one of any of these inner syntactic-expressions chosen
arbitrarily.

Case 2.2. A bullet occurs as such a separator and an or-symbol does not.

Consider all bullets occurring thus in the given syntactic-expression to
partition it into a sequence of inner syntactic-expressions. An
interpretation is the same as one of a syntactic-expression formed by
omitting these inner syntactic-expressions in an arbitrary order and
omitting these bullets.

Case 2.3. (Otherwise).

(The syntactic-expression is a sequence of syntactic-expressions optionally
separated by blanks.) An interpretation consists of the concatenation, in
order, of interpretations of the syntactic-expressions of the sequence.

1.3.3 OPERATIONS

The procedural part of the metalanguage provides for the writing of algorithms termed
operations. These are expressed in a semi-formal programming language which uses the
grammatical flexibility of ordinary English prose, while at the same time attaching
precise meaning to certain words and phrases, in order that the flow of control and the
tree manipulations in the operations be well defined. Completely formal notation is used
to describe trees in accordance with their syntactic definitions.
1.3.3.1 Nature of an Operation

An operation, applied to zero or more operands, may be performed by the processor (see section 1.3.4.), with the effect of:

1. changing the machine-state, or
2. changing an operand, or
3. returning a result, or
4. any combination of (1), (2), and (3)

Within an operation either operand-names or local-variable-names may be used for accomplishing this effect. These names serve as designators of trees, which may be either portions of the machine-state, or local-trees created within this or another operation.

When an operation is performed, its operand-names are set to designate the operand trees it has been passed. It then has the following data available to it:

1. The whole machine-state, which is directly accessible for inspection or change at any time.
2. The operands which it has been passed, which can be inspected or changed.
3. Local-trees local to itself, which it can freely construct, inspect, or modify. These trees are deleted when the operation terminates.

The operation may also apply any operation defined in this document (including itself) to any operands which it may select from among the trees available to it.

Upon completion of its actions, an operation may return a result, which then becomes available to whichever operation applied the given operation to its operands. As with operands, this result may be selected from among the trees available to the operation.

In the case that it is portion of a local-tree local to this operation is selected, this tree is not deleted, but is copied to become a local-tree local to the applying operation. (See section 1.3.3.4 on "perform").

1.3.3.2 Nondeterministic Operations

The phrases "optionally" and "in any order" are used in some operation descriptions. They indicate that the processor is to make a choice each time that part of the operation is executed. In general, then, the conceptual PL/1 machine defines a set of possible interpretations for a program.

1.3.3.3 Format of an Operation

The written description of an operation has a format consisting of a heading and a body.

The heading may have three parts:

1. There is always a specification of the form "operation:" followed by the underlined name of the operation, optionally followed by a parenthesized list of the names used to refer to the operands passed to the operation, the names being separated by commas.

2. For an operation which has operands, the word "where" then precedes a description of the types(s) of tree to which each operand name may refer. Brackets around a node type indicate that the operand is optional, which is an abbreviation for stating that it may alternatively have the value "none".

3. For an operation which returns a result, a final part of the heading is of the form "returns:" followed by a description of the type(s) of tree which may be returned.
Operation: \texttt{evaluate-in-option}(\texttt{al}, \texttt{vr})

where \texttt{al} is an \texttt{allocation},
\texttt{vr} is a \texttt{variable-reference}.

result: a \texttt{generation}.

Example 1.3. An Example of an Operation Returning a Result.

The body of an operation consists of either a \texttt{Step-list} or a \texttt{Case-set}. Each \texttt{Step} or \texttt{Case}
is a numbered section containing written instruction descriptions of arbitrary complexity, and may itself contain a \texttt{Step-list} or \texttt{Case-set}, numbered with an additional index position and indexed to indicate this containment.

Steps are normally performed sequentially.

Each \texttt{Case} consists of a condition part followed by an executable part. Within each case-set, the set of condition parts is such that, at any execution point, exactly one of the condition parts is satisfied. (The conventional condition part "Otherwise" is satisfied wherever none of the other condition parts of the case-set is satisfied.) The execution of a case-set consists of executing the executable part of the one case whose condition part is then satisfied.

The normal sequence for \texttt{Steps} and \texttt{Cases} may be modified by explicit instructions using such terms as "go to" or "perform" which are defined in Section 1.1.3.4. The terminology for selecting some actions, either in a defined left-to-right order or in an unspecified order, is "in left-to-right order" and "in any order" respectively. When selection is between two options, "in either order" may be used instead of "in any order". Optional selection as to whether an action will be carried out is indicated by words such as "optionally perform".

Operation: \texttt{every-bit}(\texttt{r0}, x)

Step 1. Perform evaluate-expressions to obtain an \texttt{aggregate-value}, \texttt{u}.
Step 2. In any order, convert each scalar-element of \texttt{u} to \texttt{bit}, to obtain \texttt{v}.
Step 3.

Case 3.1. Every scalar-element of \texttt{v} that does not contain \texttt{null-bit-string} has a \texttt{bit-string-value} with every \texttt{bit-value} containing \texttt{one-bit}.

Let \texttt{r} be \texttt{one-bit}.

Case 3.2. (Otherwise).

Let \texttt{r} be \texttt{zero-bit}.

Step 4. Return an \texttt{aggregate-value} containing a \texttt{bit-string-value} containing \texttt{r}.

Example 1.4. An Example of an Operation Containing a \texttt{Step-list} and a \texttt{Case-set}.
1.2.3.4 Instructions

An instruction is a specification of some action involving the creation, destruction, inspection, or modification of some tree(s), or causing some departure to be made from the normal sequential flow of control through an operation.

Operand-names and local-variable-names strictly denote designators of trees rather than tree values themselves; however, apart from their use in a context where they acquire designator values, are passed as operands, or are returned as results, references to these names are always taken to be an abbreviation for references to the tree designator by the strict value.

The "let" instruction is used to indicate that the named variable is henceforth to reference the specified tree, which may be an existing tree or one newly created, e.g. by one of the enumerated-tree notation or by copying an existing tree. Use of a name for a tree following some description of the root-node and a comma, e.g. in the enumerated-tree notation, is equivalent to use of a "let" instruction. No change to any tree previously designated by the named variable or operand occurs as a result of the "let" instruction.

In contrast, the "replace" instruction is used to indicate that the tree referenced by the named variable or operand is to be replaced with the specified tree. The replacement occurs exactly at the root of the referenced tree, and the named variable or operand henceforth references the replacement.

The "append" instruction, as in "append b to c", indicates that b is to be attached as the last immediate subtree of c, c may be any existing tree, or if it is a uniquely specified, may be non-existent. This latter case causes the construction of the (simple) node c. In order to make the specification of the potentially missing node c unique, the notation illustrated by "append b to c in d" can be employed to indicate that c may be missing and is to be constructed as a simple component of d.

The "attach" instruction, as in "attach b to c", indicates that b is to be attached to c as a component of c. If b can be an immediate component of c, then it is attached as an immediate component. Otherwise, there will be a unique way that a node of type b can be a simple component of c, and exactly the minimal necessary nodes which are both contained by c and also contain b are created so as to attach b as a simple component.

The "delete" instruction, as in "delete b", indicates that b is to be removed from its containing tree (and discarded). In addition, if b is a mandatory component of the tree which immediately contained b, say c, then the "delete" instruction is applied to c (i.e., c is discarded and the process continues with the tree which immediately contained c).

The "perform" instruction either calls another operation, possibly passing operands, possibly receiving a result, or calls for the execution of some steps with an operation out of the normal sequence. In calling other operations, references to trees specified by the argument list are passed to the named operation as operands (any missing arguments are given the value "absent" in the called operation). If the operation returns a result, then the term "to obtain" is used to indicate the obtained result. When other steps in an operation are performed, control returns to the instruction following the "perform" instruction.

The "go to" instruction indicates that the normal sequence of control is broken and that the named Step is to be executed next.

The "terminate" instruction indicates that the execution of the current operation is to be terminated and control is to be returned to the calling operation. If control reaches the end of an operation that does not return a result, an implicit "terminate" instruction is assumed. The "return" instruction indicates that the current operation is to be terminated with a reference to a specified tree as the result.

The "if" instruction indicates that if the specified condition is true then the instruction list following the "then" is executed and the instruction list (if any) following the "otherwise" is skipped. If the condition is false the instruction list following the "then" is skipped and, if there is one, the instruction list following the "otherwise" is executed.

The "for each" instruction specifies actions that are to be carried out once for each member of a set of objects, in some sequence which may be in any order or in some specified order.
The "must" instruction specifies a test to be performed. If the test is not satisfied, the program when combined with the particular initial entry-point and datasets if the interpretation-phase has begun is rejected by the standard, and the conceptual PL/I machine stops. This is the only sense in which "must" and "must not" are used in operation descriptions.

1.3.3.5 Convert

Convert is an exception to the general rules for naming and performing (see Section 1.3.3.4). Use of this operation is generally specified in an informal style, e.g., "convert the value of the <expression> x to integer-type." Details are given in Section 9.5.1.1.

1.3.3.6 Additional Notational Conventions

Common mathematical symbols are used with their usual meaning. In addition, the following notational conventions are used:

\[\begin{align*}
\ast & \text{ denotes multiplication;} \\
/ & \text{ denotes division;} \\
\div & \text{ denotes exponentiation;} \\
\sum & \text{ denotes iterated addition (summation); the result is taken as zero if the iteration range is empty;} \\
\prod & \text{ denotes iterated multiplication (product); the result is taken as one if the iteration range is empty;} \\
[ ] & \text{ denotes subscripting in the metalanguage as a notational convention used with local names;} \\
\pi & \text{ denotes the mathematical constant of that name;} \\
\theta & \text{ denotes the mathematical constant of that name;} \\
\mathbf{i} & \text{ denotes the square root of -1;} \\
\text{ceil}(x) & \text{ denotes the smallest integer larger than or equal to } x; \\
\text{floor}(x) & \text{ denotes the largest integer smaller than or equal to } x; \\
\text{min}(x,y) & \text{ denotes the value of } x \text{ if } x \leq y, \text{ otherwise the value of } y; \\
\text{max}(x,y) & \text{ denotes the value of } x \text{ if } x \geq y, \text{ otherwise the value of } y; \\
\log(x) & \text{ denotes the natural logarithm of } x; \\
|x| & \text{ denotes the absolute value of } x.
\end{align*}\]

1.3.3.7 Arithmetic

In the metalanguage, an arithmetic expression denotes the exact mathematical value. The situations in which the result may be approximated by the PL/I machine are defined in such operations as arithmetic-result.

In general the distinction between numbers and trees containing them is ignored. For example, if \( z \) is a tree of the form

\[
<\text{basic-value}>;
<\text{complex-value}>;
<\text{real-number}>.x.
<\text{real-number}>.y,
\]

then the arithmetic expression "z+1" denotes the complex number (x+1)+yi.
1.3.4 THE PROCESSOR

The processor is the active agent capable of performing various actions on the machine-state tree. These actions are carried out as directed by the written algorithms which comprise the FlI definition.

A portion of the machine-state tree, either the control-state or a statement-control, holds the information which controls the processor. An operation is known by the processor if it is a simple component of the control-state or a statement-control. There is at most one active operation. An operation is active if:

1. it is the last member of its immediately containing operation-list, and
2. either:
   2.1. the operation is simply contained in an operation-list which is simply contained in a statement-control which is simply contained in a block-state that is the last member of its immediately containing block-state-list, or
   2.2. there are no statement-controls contained in the machine-state, and the operation is simply contained in an operation-list which is simply contained in the control-state.

(See section 1.4.1 and section 5.1 for the definitions of these category-names.)

The processor carries out actions as follows:

Whenever there exists a known active operation, the processor carries out the actions specified by the written description of the corresponding operation.

Whenever there exists no known active operation, the processor does nothing.

1.3.5 MECHANIZATION OF THE METALANGUAGE

A deeper understanding of the metalanguage may be obtained by considering how its formal mechanization may be carried out as an extension of the machine-state. Each operation is an operation-list may be given a subtree containing the name of the operation, the names and designator values of its operands, the names and designator values of its local variables, the local-trees constructed during the performance of this operation, and control information indicating which Step or Case is currently being executed, which members of an iterative "for each" have still to be performed, and so on. When a perform instruction causes a new operation to be invoked, a new operation tree will be appended to the operation-list and activity in the present operation will be suspended. On return from an operation, any local-trees returned by it as a result is copied back as a subnode of the preceding operation tree, and the operation tree for the terminating operation is deleted together with all its local information. Activity in the invoking operation is then resumed from the point at which it was suspended.
During the translation-phase, the only <operation-list> is the one in the <control-state>. During most of the interpretation-phase, the <operations> in the <control-state> will be dormant while there is an active <operation> in a <statement-control> as shown below.

Example 1.5. Example showing <operation-lists> of a <machine-state>.
1.4 Initialization of the Machine-state

1.4.1 THE MACHINE-STATE

M1. \(<\text{machine-state}>::= <\text{program}> <\text{control-state}>\) 
\(<\text{translation-state}> | <\text{interpretation-state}>\)

M2. \(<\text{control-state}>::= <\text{operation-list}>\)

M3. \(<\text{translation-state}>::= <\text{concrete-external-procedure}>\)

M4. \(<\text{concrete-external-procedure}>::= <\text{declaration-commalist}> <\text{procedure}>\)

M5. \(<\text{operation}>::=\)

The exact structure of <operation> is left formalized and unspecified. It must have adequate structure and capacity to represent the carrying out of the actions of an operation. This includes designating the particular operation and the current position within it, holding the operands given to the operation, and holding the values of any variables used by the operation (see Section 1.3.5).

The definitions of <declaration> and <procedure> are given in Chapter 2; the definition of <program> is given in Chapter 3; the definition of <interpretation-state> is given in Chapter 5.

1.4.2 INITIALIZATION

The PL/1 definition process begins by creating an initial <machine-state> tree, consisting of:

\(<\text{machine-state}>;
\(<\text{program}>;
\(<\text{control-state}>;
\(<\text{operation-list}>;
\(<\text{operation}>\) for \text{define-program.}\)

The processor then performs the <operation> for define-program.

1.4.3 THE TOP-LEVEL OPERATIONS

1.4.3.1 Define-program

Operation: define-program

Step 1. Perform translation-phase.
Step 2. Perform interpretation-phase.
Step 3. No action. (Reaching this point indicates the successful completion of the definition algorithm.)
1.4.1.2 Translation-phase

Operation: Translation-phase

Step 1. Append <translation-state> to the <machine-state>.

Step 2.

Step 2.1. Obtain, from a source outside this definition, a sequence of characters forming a putative PL/I external procedure, constructed in the form of a symbol-list, s1.

Step 2.2. Perform translate(s1) to obtain an <abstract-external-procedure>, sep. Append sep to the <abstract-external-procedure-list> in the <program>.

Step 2.3. Optionally go to Step 2.

Step 3. Perform validate-program.

Step 4. Delete the <translation-state>.

1.4.1.3 Interpretation-phase

Operation: Interpretation-phase

Step 1. Obtain, from a source outside this definition, the following items:

1) A collection of information to be used for input/output, constructed in the form of a suitable <dataset-list>, sl.

2) A designation, as the first to be activated, of one of the <entry-point>s of a <procedure> simple component of <program>, constructed in the form of a suitable <entry-value>, ev. Such an <entry-point> must exist and must not have <parameter-name-list> or <returns-descriptor> components.

Step 2. Perform interpret(sl, ev.) (See Section 5.3.1).
Chapter 2: Concrete Syntax

2.0 Introduction

The Concrete Syntax of PL/I is specified mainly by means of production-rules using the notation defined in Chapter 1. The first such rule defines a procedure, and subsequent rules define the permitted forms of a procedure and its components in increasingly fine detail, until every component is ultimately described in terms of sequences of characters of the language character set.

2.1 The Intent of this Definition

As the first stage of translation (Chapter 8), any given sequence of symbols is parsed to determine whether that sequence indeed represents a procedure valid according to a set of rules known in this document as the "Concrete Syntax".

2.1.1 Concrete and Abstract Syntaxes

This formal Concrete Syntax is permissive in the sense that some of the constructs permitted are not actually valid procedures. Thus, for example, the sequence of symbols "DCL X FLOAT FIXED;" is a syntactically correct construct that may be parsed as a declare-statement. Errors of this sort will be detected later in the translation, because of a failure to satisfy the Abstract Syntax (Chapter 8).

2.2 Organization of the Concrete Syntax

The rules of the Concrete Syntax, which are context-free, are arranged in three levels, so that two context-dependent features of this grammar, namely the presence of blanks and comments, and the so-called "multiple closure", may be resolved at the interface between the levels.

The three levels of syntax correspond to the three levels of the parse algorithm described in Chapter 4.

2.3 The High-level Syntax of PL/I

2.3.1 PROCEDURE

Cel. \{procedure\} ::= \{prefix-list\} \{procedure-statement\} \{unit-list\} \{ending\}

2.3.2 UNIT

Cel2. \{unit\} ::= \{statement-name-list\} \{declare-statement\} \{default-statement\} | \{statement-name-list\} \{entry-statement\} | \{prefix-list\} \{format-statement\} | \{procedure\} | \{executable-unit\}
Example 2.1 illustrates the tree of the high-level structure of a simple `procedure`.

Example 2.2 illustrates the high-level structure of a simple `if-statement`.

---

```plaintext
T: procedure
  DCL PI INIT (3.14159);
  PUT LIST (START(PI/3));
  END T;
```

---

Example 2.1. The high-level structure of a simple `procedure`.
2.4 The Middle-level Syntax of PL/I

2.4.1 SENTENCE

A \texttt{sentence} in the middle-level syntax corresponds to that which comes between semicolons in a PL/I \texttt{procedure}.

CM1. \texttt{sentence} ::= \texttt{[prefix-list]} \texttt{[prefix]} \texttt{[single-statement]} \texttt{| \{else-part\}}

CM2. \texttt{else-part} ::= ELSE \texttt{[prefix-list]} \texttt{[single-statement]}

CM3. \texttt{[prefix-list]} ::= \texttt{[prefix]} \texttt{[prefix]} \texttt{[single-statement]}

CM4. \texttt{[prefix]} ::= \texttt{[condition-prefix-list]} \texttt{| \{statement-name\}}

CM5. \texttt{[if-clause]} ::= IF \texttt{[expression]} THEN
2.4.2 STATEMENT

CM8. \texttt{\textbackslash \{single\-statement\}} := \texttt{\{statement-name-list\}}
\begin{itemize}
\item \texttt{\{declare\-statement\}}
\item \texttt{\{default\-statement\}}
\item \texttt{\{end\-statement\}}
\item \texttt{\{statement-name-list\}} \texttt{\{entry\-statement\}}
\item \texttt{\{prefix\-list\}} \texttt{\{executable\-single\-statement\}}
\item \texttt{\{begin\-statement\}} \texttt{\{formated\}}
\item \texttt{\{do\-statement\}} \texttt{\{unmatched\}} \texttt{\{format\-statement\}}
\item \texttt{\{system\}} \texttt{\{procedure\-statement\}} \texttt{\{unmatched\}}
\end{itemize}

Notes \texttt{\{unmatched\}} is used only by the operation high-level-parse.

CM9. \texttt{\{executable\-single\-statement\}} := \texttt{\{allocate\-statement\}}
\begin{itemize}
\item \texttt{\{open\-statement\}}
\item \texttt{\{assignment\-statement\}}
\item \texttt{\{call\-statement\}}
\item \texttt{\{clone\-statement\}}
\item \texttt{\{delete\-statement\}}
\item \texttt{\{free\-statement\}}
\item \texttt{\{rewrite\-statement\}}
\item \texttt{\{put\-statement\}}
\item \texttt{\{signal\-statement\}}
\item \texttt{\{goto\-statement\}}
\item \texttt{\{locate\-statement\}}
\item \texttt{\{write\-statement\}}
\item \texttt{\{null\-statement\}}
\end{itemize}

2.4.3 PREFIXES

2.4.3.1 Condition Prefixes

CM8. \texttt{\{condition\-prefix\}} := \texttt{\{computational\-condition\}}
\begin{itemize}
\item \texttt{\{disabled\-computational\-condition\}}
\end{itemize}

CM9. \texttt{\{computational\-condition\}} := \texttt{CONVERSION | FIXEDOVERFLOW | OVERFLOW | SIZE | STRINGSIZE | SUBSCRIPTSIZE | UNDERFLOW | INORDIVIDE}

CM10. \texttt{\{disabled\-computational\-condition\}} := \texttt{NOCONVERSION | NOFIXEDOVERFLOW | NOOVERFLOW | NONSIZE | NOSTRINGSIZE | NOSUBSCRIPTSIZE | NONUNDERFLOW | NONINORDIVIDE}

2.4.3.2 Statement Name Prefixes

CM11. \texttt{\{statement-name\}} := \texttt{\{identifier\}} \texttt{\{\{signed\-integer\-comma\-list\}\}}

CM12. \texttt{\{signed\-integer\}} := \texttt{\{+ | -\}} \texttt{\{integer\}}

2.4.4 DATA DECLARATION

CM13. \texttt{\{declare\-statement\}} := \texttt{DECLARE \{declaration\-comma\-list\}}

CM14. \texttt{\{declaration\}} := \texttt{\{level\}} \texttt{\{\{identifier\} | \{declaration\-comma\-list\}\}}
\texttt{\{dimension\-suffix\} \{attribute\-list\}}

CM15. \texttt{\{level\}} := \texttt{\{integer\}}
2.4.4.1 Dimension Attribute and Dimension Suffix

CM16. $\{\text{dimension-attribute}\} ::= \text{DIMENSION}\ {\{\text{dimension-suffix}\}}$
CM17. $\{\text{dimension-suffix}\} ::= \{\text{bound-pair-constant}\}$
CM18. $\{\text{bound-pair}\} ::= \{\{\text{lower-bound}\}\ {\{\text{upper-bound}\}}\ |\ *$
CM19. $\{\text{lower-bound}\} ::= \{\text{extent-expression}\}$
CM20. $\{\text{upper-bound}\} ::= \{\text{extent-expression}\}$
CM21. $\{\text{extent-expression}\} ::= \{\text{expression}\} \{\text{prefer-option}\}$
CM22. $\{\text{prefer-option}\} ::= \{\text{unsubscribed-reference}\}$

2.4.4.2 Attributes

CM23. $\{\text{attribute}\} ::= \{\text{data-attribute}\}$
               $\{\text{arbitrary}\}$
               $\{\text{built-in}\}$
               $\{\text{condition}\}$
               $\{\text{controlled}\}$
               $\{\text{defined}\} \{\text{reference}\}\ |
               \{\text{reference}\}$
               $\{\text{direct}\}$
               $\{\text{environment}\}$
               $\{\text{external}\}$
               $\{\text{generic-attribute}\}$
               $\{\text{initial}\}$
               $\{\text{key}\}$
               $\{\text{internal}\}$

2.4.4.3 Data Attributes

CM24. $\{\text{data-attribute}\} ::= \{\text{aligned}\}$
               $\{\text{area}\} \{\text{area-size}\}$
               $\{\text{binary}\} \{\text{binary}\}$
               $\{\text{bit}\} \{\text{bit-length}\}$
               $\{\text{character}\} \{\text{character}\}$
               $\{\text{complex}\} \{\text{precision}\}$
               $\{\text{decimal}\} \{\text{precision}\}$
               $\{\text{dimension-attribute}\}$
               $\{\text{entry}\} \{\text{entry}\}$
               $\{\text{file}\}$
               $\{\text{fixed}\} \{\text{precision}\}$
               $\{\text{format}\}$
               $\{\text{floating}\}$
               $\{\text{format}\}$
               $\{\text{format}\}$

CM25. $\{\text{area-size}\} ::= \{\text{extent-expression}\}\ |\ *$
CM26. $\{\text{precision}\} ::= \{\text{number-of-digits}\} \{\text{scale-factor}\}$
CM27. $\{\text{number-of-digits}\} ::= \{\text{integer}\}$
CM28. $\{\text{scale-factor}\} ::= \{\text{signed-integer}\}$
CM29. $\{\text{maximum-length}\} ::= \{\text{extent-expression}\}\ |\ *$
CM30. $\{\text{description}\} ::= \{\text{level}\} \{\text{dimension-suffix}\} \{\text{data-attribute-list}\}$

Constraint: At least one subtype must be present.

CM31. $\{\text{picture}\} ::= \{\text{simple-character-string-constant}\}$
2.4.4.4 Environment and Options

CM02. \$environment\$ := ENVIRONMENT \{$environment-specification\$
CM03. \$environment-specification\$ :=
CM04. \$options\$ := OPTIONS \{$options-specification\$
CM05. \$options-specification\$ :=

2.4.4.5 Generic

CM06. \$generic-attribute\$ := GENERIC \{$generic-element-comma-list\$
CM07. \$generic-element\$ := \{reference\} WHEN \{$generic-description-comma-list\$
CM08. \$generic-description\$ := \{level\} \{asterisk-bounds\} \{generic-data-attribute-list\} | *

Constraint: At least one subnode must be present.

CM09. \$generic-data-attribute\$ := ALIGNED | REAL
      | AREA | PRECISION | LABEL
      | BIT | BINVARING | MEMBER
      | CHAR | OFFSET | PICTURE | MEMMEM
      | COMPLEX | PRECISION | STRUCTURE | MEMPOS
      | DECIMAL | PRECISION | UNIVERSAL | FILE
      | FIXED | PRECISION | VARTAP | FIXED
      | FLOOR | PRECISION | VARTAP | FLOAT
      | FORMAT | PRECISION | VARTAP | FORMAT

CM10. \{asterisk-bounds\} := \{\*\-comma-list\}

CM11. \$generic-precision\$ := \{number-of-digits\} \{number-of-digits\}
      \{scale-factor\} \{scale-factor\}

2.4.4.6 Initial

CM12. \$initial\$ := INITIAL \{$initial-element-comma-list\$
CM13. \$initial-element\$ := * | \{parenthesized-expression\} |
      \{iteration-factor\} \{initial-constant-two\} | *
      \{initial-element-comma-list\} |
      \{initial-constant-one\}

CM14. \{initial-constant-one\} := \{prefix-operator\} \{simple-string-constant\} |
       \{initial-constant-two\}

CM15. \{initial-constant-two\} := \{prefix-operator\}
      \{reference\} \{replicated-string-constant\} |
       \{arithmetic-constant\} |
      \{real-constant\} \{\* \} \{imaginary-constant\}

CM16. \{iteration-factor\} := \{expression\}

38
2.4.4.7 The Default Statement

CH47. \texttt{\{default-statement\} := {\texttt{DEFAULT}} \{\texttt{default-specification}\} | \texttt{NONE} | \texttt{SYSTEM};}

CH48. \texttt{\{default-specification\} := \{\texttt{predicate-expression}\} \texttt{ERROR} | \{\texttt{default-attributes-constant}\};}

CH49. \texttt{\{predicate-expression\} := \{\texttt{predicate-expression-three}\} | \{\texttt{predicate-expression-two}\} & \{\texttt{predicate-expression-three}\};}

CH50. \texttt{\{predicate-expression-three\} := \{\texttt{predicate-expression-two}\} | \{\texttt{range-specification}\};}

CH51. \texttt{\{predicate-expression-two\} := \{\texttt{predicate-expression-one}\} | \{\texttt{range-expression}\};}

CH52. \texttt{\{predicate-expression-one\} := \{\texttt{predicate-expression}\} | \{\texttt{attribute-keyword}\} | \{\texttt{range-specification}\};}

CH53. \texttt{\{range-specification\} := \texttt{RANGE} \{\{\texttt{identifier}\} | \{\texttt{letter}\} | \{\texttt{letter}\} | \texttt{*}\};}

CH54. \texttt{\{attribute-keyword\} := \texttt{ALIGNED} | \texttt{DEFINED} | \texttt{EXTERNAL} | \texttt{PRECISION} | \texttt{AREA} | \texttt{DIMENSION} | \texttt{RETED} | \texttt{PRINT} | \texttt{ARRAY} | \texttt{DIRECT} | \texttt{LABEL} | \texttt{REAL} | \texttt{BASED} | \texttt{ENTRY} | \texttt{LOCAL} | \texttt{RECORD} | \texttt{BINARY} | \texttt{ENVIRONMENT} | \texttt{PARAMETER} | \texttt{RETURN} | \texttt{BYTE} | \texttt{EXTERNAL} | \texttt{NVARYING} | \texttt{SEQUENTIAL} | \texttt{BULLETIN} | \texttt{FILE} | \texttt{OFFSET} | \texttt{STATIC} | \texttt{CHARACTER} | \texttt{FIXED} | \texttt{OPTIONS} | \texttt{STREAM} | \texttt{COMPLEX} | \texttt{FLOAT} | \texttt{OFFSET} | \texttt{STRUCTURE} | \texttt{CONDITION} | \texttt{FORMAT} | \texttt{PARAMETER} | \texttt{UNALIGNED} | \texttt{CONSTANT} | \texttt{GENERIC} | \texttt{PICTURE} | \texttt{UPGRADE} | \texttt{CONTROLLED} | \texttt{INITIAL} | \texttt{JOINER} | \texttt{VARIABLE} | \texttt{DECIMAL} | \texttt{INPUT} | \texttt{POSITION} | \texttt{VARYING};}

CH55. \texttt{\{default-attributes\} := \{\texttt{attribute-list}\};}

2.4.5 THE PROCEDURE STATEMENT

CH56. \texttt{\{procedure-statement\} := \texttt{PROCEDURE} \{\texttt{entry-information}\};}

CH57. \texttt{\{entry-information\} := \{\{\texttt{parameter-name-constant-list}\}\} \{\{\texttt{returns-descriptor}\} | \{\texttt{options}\} | \{\texttt{RECURSIVE}\}\};}

CH58. \texttt{\{parameter-name\} := \texttt{identifier};}

CH59. \texttt{\{returns-descriptor\} := \texttt{RETURNS} \{\{\texttt{description-constant-list}\}\};}

2.4.6 THE ENTRY STATEMENT

CH60. \texttt{\{entry-statement\} := \texttt{ENTRY} \{\texttt{entry-information}\};}

2.4.7 THE BEGIN STATEMENT

CH61. \texttt{\{begin-statement\} := \texttt{BEGIN} \{\texttt{options}\};}

Chapter 2: Concrete Syntax 39
2.4.8 THE DO STATEMENT

CM62. \$do-statement\ ::= DO \|$do-spec\]| DO \$do-spec\]
CM63. \$do-spec\ ::= \$reference\ = \$spec-comma-list\]
CM64. \$spec\ ::= \$expression\ | \$to-by\ | \$repeat-option\ | \$while-option\]
CM65. \$to-by\ ::= \$to-option\ \$by-option\ | \$by-option\ \$to-option\]
CM66. \$to-option\ ::= TO \$expression\]
CM67. \$by-option\ ::= BY \$expression\]
CM68. \$while-option\ ::= WHILE \$expression\]
CM69. \$repeat-option\ ::= REPEAT \$expression\]

2.4.9 THE END STATEMENT

CM70. \$end-statement\ ::= END \$identifier\]

2.4.10 FLOW OF CONTROL STATEMENTS

2.4.10.1 The Call and Return Statements

CM71. \$call-statement\ ::= CALL \$reference\]
CM72. \$return-statement\ ::= RETURN \$expression\]

2.4.10.2 The Go To Statement

CM73. \$goto-statement\ ::= GO TO \$reference\]

2.4.10.3 The Null Statement

CM74. \$null-statement\ ::= ]

2.4.10.4 The Revert and Signal Statements

CM75. \$revert-statement\ ::= REVERT \$condition-name-comma-list\]
CM76. \$signal-statement\ ::= SIGNAL \$condition-name\]
CM77. \$condition-name\ ::= \$computational-condition\ | \$named-io-condition\ | \$programmer-named-condition\ | \$AREA | \$ERROR | \$FINISH | \$STORAGE
CM78. \$named-io-condition\ ::= \$io-condition\ \$reference\]
CM79. \$io-condition\ ::= \$ENDFILE | \$ENDPAGE | \$KEY | \$NAME | \$RECORD | \$TRANSMIT | \$UNDEFINEDFILE
CM80. \$programmer-named-condition\ ::= \$CONDITION \$identifier\]

2.4.10.5 The Stop Statement

CM81. \$stop-statement\ ::= STOP;
2.4.11 STORAGE CONTROL STATEMENTS

CM12. $\{\text{assignment-statement}\} ::= \{\text{reference-comma-list}\} \rightarrow \{\text{expression}\} \{\text{HT NAME}\}$

CM13. $\{\text{allocate-statement}\} ::= \text{ALLOCATE} \{\text{allocation-comma-list}\}$

CM14. $\{\text{allocation}\} ::= \{\text{identifier}\} \{\{\text{set-option}\} \rightarrow \{\text{in-option}\}\}$

CM15. $\{\text{set-option}\} ::= \text{SET} \{\text{reference}\}$

CM16. $\{\text{in-option}\} ::= \text{IN} \{\text{reference}\}$

CM17. $\{\text{free-statement}\} ::= \text{FREE} \{\text{freeing-comma-list}\}$

CM18. $\{\text{freeing}\} ::= \{\text{locator-qualifier}\} \{\text{identifier}\} \{\text{in-option}\}$

2.4.12 INPUT/OUTPUT STATEMENTS

2.4.12.1 The Open and Close Statements

CM19. $\{\text{open-statement}\} ::= \text{OPEN} \{\text{single-opening-comma-list}\}$

CM20. $\{\text{single-opening}\} ::= \{\text{file-option}\} \rightarrow \{\text{tab-option}\} \rightarrow \{\text{title-option}\} \rightarrow \{\text{linesize-option}\} \rightarrow \{\text{pagesize-option}\} \rightarrow \{\text{STREAM}\} \rightarrow \{\text{RECORD}\} \rightarrow \{\text{INPUT}\} \rightarrow \{\text{OUTPUT}\} \rightarrow \{\text{UPDATE}\} \rightarrow \{\text{SEQUENTIAL}\} \rightarrow \{\text{DIRECT}\} \rightarrow \{\text{PRINT}\} \rightarrow \{\text{TED}\} \rightarrow \{\text{environment}\}$

CM21. $\{\text{file-option}\} ::= \text{FILE} \{\text{reference}\}$

CM22. $\{\text{tab-option}\} ::= \text{TAB} \{\text{expression-comma-list}\}$

CM23. $\{\text{title-option}\} ::= \text{TITLE} \{\text{expression}\}$

CM24. $\{\text{linesize-option}\} ::= \text{LINESIZE} \{\text{expression}\}$

CM25. $\{\text{pagesize-option}\} ::= \text{PAGESIZE} \{\text{expression}\}$

CM26. $\{\text{close-statement}\} ::= \text{CLOSE} \{\text{single-closing-comma-list}\}$

CM27. $\{\text{single-closing}\} ::= \{\text{file-option}\} \rightarrow \{\text{environment}\}$

2.4.12.2 Record I/O

CM30. $\{\text{delete-statement}\} ::= \text{DELETE} \{\text{file-option}\} \rightarrow \{\text{key-option}\}$

CM31. $\{\text{locate-statement}\} ::= \text{LOCATE} \{\text{identifier}\} \{\text{file-option}\} \rightarrow \{\text{pointer-set-option}\} \rightarrow \{\text{keyfrom-option}\}$

CM34. $\{\text{pointer-set-option}\} ::= \text{SET} \{\text{reference}\}$

CM35. $\{\text{read-statement}\} ::= \text{READ} \{\text{file-option}\} \rightarrow \{\text{into-option}\} \rightarrow \{\text{pointer-set-option}\} \rightarrow \{\text{ignore-option}\} \rightarrow \{\text{key-option}\} \rightarrow \{\text{keyto-option}\}$

CM36. $\{\text{into-option}\} ::= \text{INFO} \{\text{reference}\}$

CM37. $\{\text{ignore-option}\} ::= \text{IGNORE} \{\text{expression}\}$

CM38. $\{\text{key-option}\} ::= \text{KEY} \{\text{expression}\}$
CM105. {key-to-option} := RETURN {reference};
CM106. {rewrite-statement} := REWRITE {file-option} | {file-option} * {key-option} * {from-option};
CM107. {write-statement} := WRITE {file-option} * {from-option} * {keyfrom-option};
CM108. {from-option} := FROM {reference};
CM109. {keyfrom-option} := KEYPERM {expression};

2.4.12.3 Stream I/O

CM110. {get-statement} := GET {get-file} | {get-string};
CM111. {get-file} := {file-option} * {copy-option} * {skip-option} * {input-specification};
CM112. {copy-option} := COPY {reference};
CM113. {skip-option} := SKIP {expression};
CM114. {get-string} := STRING {expression} * {input-specification} * {copy-option};
CM115. {put-statement} := PUT {put-file} | {put-string};
CM116. {put-file} := {file-option} * {skip-option} * {line-option} * {output-specification};
CM117. {line-option} := LINE {expression};
CM118. {put-string} := STRING {reference} * {output-specification};

2.4.12.3.1 Stream Input Specification

CM119. {input-specification} := {data-directed-input} | {list-directed-input} | {edit-directed-input};
CM120. {data-directed-input} := DATA {data-target-comma-list};
CM121. {data-target} := {unsubscripted-reference};
CM122. {list-directed-input} := LIST {input-target-comma-list};
CM123. {input-target} := {reference} | {input-target-comma-list} DO {do-spec};
CM124. {edit-directed-input} := EDIT {edit-input-pair-list};
CM125. {edit-input-pair} := {input-target-comma-list} {format-specification-comma-list}
2.9.12.3.2 Stream Output Specification

CH126. \( \text{output-specification} ::= \text{data-directed-output} \mid \text{list-directed-output} \mid \text{edit-directed-output} \)

CH127. \( \text{data-directed-output} ::= \text{DATA} \{ \{ \text{data-source-commalist} \} \} \)

CH128. \( \text{data-source} ::= \{ \text{reference} \mid \{ \text{data-source-commalist} \} \text{ DO } \{ \text{do-spec} \} \} \)

CH129. \( \text{list-directed-output} ::= \text{List} \{ \{ \text{output-source-commalist} \} \} \)

CH130. \( \text{output-source} ::= \{ \text{expression} \mid \{ \text{output-source-commalist} \} \text{ DO } \{ \text{do-spec} \} \} \)

CH131. \( \text{edit-directed-output} ::= \text{EDIT} \{ \text{edit-output-pair-list} \} \)

CH132. \( \text{edit-output-pair} ::= \{ \text{output-source-commalist} \}\{ \text{format-specification-commalist} \} \)

2.9.12.3.3 Format Specification Lists and the Format Statement

CH133. \( \text{format-specification} ::= \{ \text{format-item} \mid \{ \text{format-iteration} \} \}

CH134. \( \text{format-iteration} ::= \{ \text{format-iteration-factor} \}\{ \text{format-specification-commalist} \} \)

CH135. \( \text{format-iteration-factor} ::= \{ \text{format-item} \mid \{ \text{format-specification-commalist} \} \}

CH136. \( \text{format-item} ::= \{ \text{integer} \mid \{ \text{expression} \} \}

CH137. \( \text{data-format} ::= \{ \text{real-format} \mid \{ \text{complex-format} \mid \{ \text{picture-format} \mid \{ \text{string-format} \} \}

CH138. \( \text{real-format} ::= \{ \text{fixed-point-format} \mid \{ \text{floating-point-format} \}

CH139. \( \text{fixed-point-format} ::= \text{F} \{ \text{expression} \mid \{ \text{expression} \} \text{,} \{ \text{expression} \} \}

CH140. \( \text{floating-point-format} ::= \text{E} \{ \text{expression} \mid \{ \text{expression} \} \text{,} \{ \text{expression} \} \}

CH141. \( \text{complex-format} ::= \text{C} \{ \{ \text{real-format} \mid \{ \text{picture-format} \}

CH142. \( \text{picture-format} ::= \text{P} \{ \text{picture} \}

CH143. \( \text{string-format} ::= \{ \text{character-format} \mid \{ \text{bit-format} \}

CH144. \( \text{character-format} ::= \text{A} \{ \{ \text{expression} \} \}

CH145. \( \text{bit-format} ::= \{ \text{redix-factor} \{ \{ \text{expression} \} \}

CH146. \( \text{control-format} ::= \{ \text{tab-format} \mid \{ \text{line-format} \mid \{ \text{space-format} \mid \{ \text{skip-format} \mid \{ \text{column-format} \mid \text{PDE} \}

CH147. \( \text{tab-format} ::= \text{TAB} \{ \{ \text{expression} \} \}

CH148. \( \text{line-format} ::= \text{LINE} \{ \{ \text{expression} \} \}

CH149. \( \text{space-format} ::= \text{X} \{ \{ \text{expression} \} \}

CH150. \( \text{skip-format} ::= \text{SKIP} \{ \{ \text{expression} \} \}

CH151. \( \text{column-format} ::= \text{COLUMN} \{ \{ \text{expression} \} \}

CH152. \( \text{remote-format} ::= \text{R} \{ \text{reference} \}

CH153. \( \text{format-statement} ::= \text{FORMAT} \{ \text{format-specification-commalist} \} \)
2.6.13 EXPRESSIONS

CH154. expression::= expression-sequence | expression \(
\) expression
CH155. expression-sequence::= expression-six | expression-sequence \& expression-six
CH156. expression-six::= expression-five | \{comparison-operator\} expression-five
CH157. \{comparison-operator\}::= \(\, > \, | \, \geq \, | \, < \, | \, \leq \, | \, = \, | \, \neq \, \)
CH158. expression-five::= expression-four | \{expression-five \} \{expression-four\}
CH159. expression-four::= expression-three | \{expression-four \} \{expression-three\} \{+\} \{-\}
CH160. expression-three::= expression-two | expression-three \{\ast\} expression-three \{/\}
CH161. expression-two::= \{primitive-expression\} | \{prefix-expression\} | \{parenthesised-expression\} | \{expression-one\}
CH162. \{primitive-expression\}::= \{parenthesised-expression\} | \{primitive-expression\} \{expression-one\} \{**\}
CH163. \{prefix-expression\}::= \{prefix-operator\} \{expression-two\}
CH164. \{prefix-operator\}::= \{+\} | \{-\} \{-\}
CH165. \{parenthesised-expression\}::= \{expression\}
CH166. \{primitive-expression\}::= \{reference\} | \{constant\} \{\#name\}
CH167. \{reference\}::= \{locator-qualifier\} \{basic-reference\} \{arguments-list\}
CH168. \{locator-qualifier\}::= \{reference\} \{\rightarrow\}
CH169. \{arguments\}::= \{\{subscribe-costalist\}\}
CH170. \{basic-reference\}::= \{structure-qualification\} \{identifier\}
CH171. \{structure-qualification\}::= \{basic-reference\} \{arguments\}
CH172. \{subscribe\}::= \{expression\} \{\ast\}
CH173. \{unsubscribe-reference\}::= \{unsubscribe-reference\} \{identifier\}
CH174. \{constant\}::= \{arithmetic-constant\} \{string-constant\}
CH175. \{string-constant\}::= \{simple-string-constant\} | \{replicated-string-constant\}
CH176. \{simple-string-constant\}::= \{simple-character-string-constant\} | \{simple-bit-string-constant\}
CH177. \{replicated-string-constant\}::= \{integer\} \{simple-string-constant\}
The following are examples of two middle-level parses. As in the previous examples, each is accompanied by an example of a construct that matches the given syntax.

---

Example 2.3. An example of the middle-level structure of an {expression}.

---

A sample concrete representation of the above structure:

\[(A+B)+C-D/3\]
Example 2.4. An Example of the Middle-level Structure of a `do-statement`.
2.5 The Low-level Syntax of PL/I

2.5.1 PL/I TEXT

CL1. {pli-text}::= [{delimiter-list}] [{delimiter-pair-list}]

CL2. {delimiter-pair} ::= {non-delimiter} {delimiter-list}

CL3. {delimiter-list} ::= + | - | * | / | ** | > | < | = | >= | <= | -> | <> | =< | => | ~ | & | { | } | [ | ] | ? | -> | < | >= | <= | ! | -> | B | {comment} | {include}

CL4. {non-delimiter} ::= {identifier} | {arithmetic-constant} | {simple-bit-string-constant} | {simple-character-string-constant} | {list}

2.5.2 COMMENT

CL5. {comment} ::= /* {comment-body-list} */"list" /

CL6. {comment-body-list} ::= [{"list"}] {comment-character} /

CL7. {comment-character} ::= {letter} | {digit} | - | B | ' | = | + | - | @ | | |

Note: Rules CL5-7 effectively state that a comment begins with /* and ends with */ and that any characters may appear between these except the consecutive pair */.

2.5.3 IDENTIFIERS

CL8. {identifier} ::= {letter} | {digit} | {.identifier} | {.identifier} | $.identifier | _

CL9. {letter} ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

CL10. {digit} ::= 0|1|2|3|4|5|6|7|8|9

2.5.4 ARITHMETIC CONSTANT

CL11. {arithmetic-constant} ::= {real-constant} | {imaginary-constant}

CL12. {real-constant} ::= {decimal-constant} | {binary-constant}

CL13. {decimal-constant} ::= {decimal-number} {scale-type} {exponent} (P)

CL14. {decimal-number} ::= {integer} [.]{digit-list} [.]{digit-list}

CL15. {integer} ::= {digit-list}

CL16. {scale-type} ::= E | F

CL17. {exponent} ::= 1 | -1 | {integer}

CL18. {binary-constant} ::= {binary-number} {scale-type} {exponent} (B)

CL19. {binary-number} ::= {binary-digit-list} [.]{binary-digit-list}

CL20. {binary-digit} ::= 0 | 1

CL21. {imaginary-constant} ::= {real-constant} i
2.5.5 STRING CONSTANTS AND PICTURES

CL22. \{$simple-bit-string-constant\}::= '\{\{simple-or-picture-symbol-list\}\'} \{radix-factor\}

CL23. \{radix-factor\}::= 8|10|16|84

CL24. \{$simple-character-string-constant\}::= '\{\{string-or-picture-symbol-list\}\}'

CL25. \{string-or-picture-symbol\}::= \{letter\} | \{digit\} | . | # | ' ' | + | - | * |
\/
| | | | | | | | | | | | | | | |
< | % | \{extralingual-character\}

Note: A \{string-or-picture-symbol\} may be two consecutive characters "" or any character other than '

CL26. \{extralingual-character\}::=

The category \{extralingual-character\} in rule CL26 is implementation-defined, as specified in Section 2.6.2.

2.5.6 ISO78

CL27. \{isbn\}::= \{integer\} \&8

2.5.7 INCLUDE

CL28. \{include\}::= \{INCLUDE \{text-name\}\}

CL29. \{text-name\}::=

The category \{text-name\} in rule CL29 is implementation-defined and such that if it contains \{\}, then that \{\} must be contained in a \{simple-character-string-constant\} which is the only immediate subnode of \{text-name\}.

2.6 Character Sets

The character set used in the formation of PL/I text is a finite set of symbols, comprising 57 language characters, and zero or more \{extralingual-character\} which are distinct from these and from each other and are implementation-defined.

\{symbol\}::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | . | , | ' | = | \* | / | \{ | \} | - | \[ | \| | + | \% | \} | \|

This document does not specify internal or external hardware representations of the characters, nor does it define a collating sequence for them. These may, however, be the subject of other standards.

2.6.1 LANGUAGE CHARACTER SET

The names of the symbols in the language character set, together with the graphic representations of them to be used in this document, are given in Sections 2.6.1.1 and 2.6.1.2.
2.4.1 Letters and Digits

<table>
<thead>
<tr>
<th>Name</th>
<th>Graphic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Letter A</td>
<td>A</td>
</tr>
<tr>
<td>Letter B</td>
<td>B</td>
</tr>
<tr>
<td>Letter C</td>
<td>C</td>
</tr>
<tr>
<td>Letter D</td>
<td>D</td>
</tr>
<tr>
<td>Letter E</td>
<td>E</td>
</tr>
<tr>
<td>Letter F</td>
<td>F</td>
</tr>
<tr>
<td>Letter G</td>
<td>G</td>
</tr>
<tr>
<td>Letter H</td>
<td>H</td>
</tr>
<tr>
<td>Letter I</td>
<td>I</td>
</tr>
<tr>
<td>Letter J</td>
<td>J</td>
</tr>
<tr>
<td>Letter K</td>
<td>K</td>
</tr>
<tr>
<td>Letter L</td>
<td>L</td>
</tr>
<tr>
<td>Letter M</td>
<td>M</td>
</tr>
<tr>
<td>Letter N</td>
<td>N</td>
</tr>
<tr>
<td>Letter O</td>
<td>O</td>
</tr>
<tr>
<td>Letter P</td>
<td>P</td>
</tr>
<tr>
<td>Letter Q</td>
<td>Q</td>
</tr>
<tr>
<td>Letter R</td>
<td>R</td>
</tr>
<tr>
<td>Letter S</td>
<td>S</td>
</tr>
<tr>
<td>Letter T</td>
<td>T</td>
</tr>
<tr>
<td>Letter U</td>
<td>U</td>
</tr>
<tr>
<td>Letter V</td>
<td>V</td>
</tr>
<tr>
<td>Letter W</td>
<td>W</td>
</tr>
<tr>
<td>Letter X</td>
<td>X</td>
</tr>
<tr>
<td>Letter Y</td>
<td>Y</td>
</tr>
<tr>
<td>Letter Z</td>
<td>Z</td>
</tr>
</tbody>
</table>

2.4.1.2 Special Characters

<table>
<thead>
<tr>
<th>Name</th>
<th>Graphic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plus</td>
<td>+</td>
</tr>
<tr>
<td>Minus</td>
<td>-</td>
</tr>
<tr>
<td>Asterisk</td>
<td>*</td>
</tr>
<tr>
<td>Slash</td>
<td>/</td>
</tr>
<tr>
<td>Greater than</td>
<td>&gt;</td>
</tr>
<tr>
<td>Less than</td>
<td>&lt;</td>
</tr>
<tr>
<td>Equal</td>
<td>=</td>
</tr>
<tr>
<td>Not</td>
<td>!</td>
</tr>
<tr>
<td>And</td>
<td>&amp;</td>
</tr>
<tr>
<td>Or</td>
<td></td>
</tr>
<tr>
<td>Percent</td>
<td>%</td>
</tr>
<tr>
<td>Period</td>
<td>.</td>
</tr>
<tr>
<td>Comma</td>
<td>,</td>
</tr>
<tr>
<td>Semicolon</td>
<td>;</td>
</tr>
<tr>
<td>Colon</td>
<td>:</td>
</tr>
<tr>
<td>Semicolon</td>
<td>;</td>
</tr>
<tr>
<td>Single quote</td>
<td>’</td>
</tr>
<tr>
<td>Left parenthesis</td>
<td>(</td>
</tr>
<tr>
<td>Right parenthesis</td>
<td>)</td>
</tr>
<tr>
<td>Break</td>
<td>\</td>
</tr>
<tr>
<td>Dollar</td>
<td>$</td>
</tr>
</tbody>
</table>

2.4.2 DATA CHARACTER SET

Data in stream datasets or in character-string-values may be represented by characters from the language character set plus any other characters permitted by the particular implementation's extralingual character set.
2.7 Abbreviations

Abbreviations are provided for certain keywords (see Section 4.2.2) and builtin-function-names. The abbreviations will be recognized as synonymous in every respect with the full denotations, except that in the case of builtin-function-names the abbreviations have separate declarations (explicit or contextual) and same scopes. The abbreviations are shown to the right of the full denotations in the following list.

<table>
<thead>
<tr>
<th>ALLOCATE</th>
<th>ALLOC</th>
<th>ALLOCN</th>
<th>N有利于</th>
<th>N有利于</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUTOMATIC</td>
<td>AUTO</td>
<td></td>
<td>N有利于</td>
<td>N有利于</td>
</tr>
<tr>
<td>RINARY</td>
<td>BIN</td>
<td></td>
<td>N有利于</td>
<td>N有利于</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>CHAR</td>
<td></td>
<td>N有利于</td>
<td>N有利于</td>
</tr>
<tr>
<td>COLUMN</td>
<td>COL</td>
<td></td>
<td>N有利于</td>
<td>N有利于</td>
</tr>
<tr>
<td>COMPLEX</td>
<td>CPLE</td>
<td></td>
<td>N有利于</td>
<td>N有利于</td>
</tr>
<tr>
<td>CONDITION</td>
<td>COND</td>
<td></td>
<td>N有利于</td>
<td>N有利于</td>
</tr>
<tr>
<td>CONTROLLED</td>
<td>CTL</td>
<td></td>
<td>N有利于</td>
<td>N有利于</td>
</tr>
<tr>
<td>CONVERSION</td>
<td>CONV</td>
<td></td>
<td>N有利于</td>
<td>N有利于</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>DEC</td>
<td></td>
<td>N有利于</td>
<td>N有利于</td>
</tr>
<tr>
<td>DECLARE</td>
<td>DCL</td>
<td></td>
<td>N有利于</td>
<td>N有利于</td>
</tr>
<tr>
<td>DEFINED</td>
<td>DEF</td>
<td></td>
<td>N有利于</td>
<td>N有利于</td>
</tr>
<tr>
<td>DIMENSION</td>
<td>DIM</td>
<td></td>
<td>N有利于</td>
<td>N有利于</td>
</tr>
<tr>
<td>ENVIRONMENT</td>
<td>ENV</td>
<td></td>
<td>N有利于</td>
<td>N有利于</td>
</tr>
<tr>
<td>EXTERNAL</td>
<td>EXT</td>
<td></td>
<td>N有利于</td>
<td>N有利于</td>
</tr>
<tr>
<td>FIXEDOVERFLOW</td>
<td>FOFL</td>
<td></td>
<td>N有利于</td>
<td>N有利于</td>
</tr>
<tr>
<td>INITIAL</td>
<td>INIT</td>
<td></td>
<td>N有利于</td>
<td>N有利于</td>
</tr>
<tr>
<td>INTERNAL</td>
<td>INT</td>
<td></td>
<td>N有利于</td>
<td>N有利于</td>
</tr>
<tr>
<td>NOCONVERSION</td>
<td>NOCONV</td>
<td></td>
<td>N有利于</td>
<td>N有利于</td>
</tr>
<tr>
<td>NOFIXEDOVERFLOW</td>
<td>NOFOFL</td>
<td></td>
<td>N有利于</td>
<td>N有利于</td>
</tr>
<tr>
<td>NOVARYING</td>
<td>NOVAR</td>
<td></td>
<td>N有利于</td>
<td>N有利于</td>
</tr>
</tbody>
</table>
3.0 Introduction

This chapter specifies the Abstract Syntax of PL/I, which is the syntax of programs in a tree representation which is convenient for the definition of semantics. The notation for this syntax is defined in Chapter 1, together with some examples of its use. Further examples are provided at the end of this chapter.

Many parts of the Abstract Syntax bear a strong resemblance to the corresponding parts of the Concrete Syntax, and the relationship between them is intended to be, or become, intuitively obvious. In other parts, notably in the treatment of <declaration>, the Abstract Syntax exhibits a structuring and completeness of information which involves a more complex transformation. The detailed description of the transformation between the concrete and abstract forms of a program will be given in the next chapter.

There are also many instances of context-dependent constraints which have been inserted in parentheses in the Abstract Syntax. These are attached to categories such as <expression>, <target-reference>, <value-reference> and <declaration-designator>, where these are required to fall within the scope of appropriate <declaration>s so that they have the properties indicated. The Translator (Chapter 4) checks that these constraints are satisfied.

3.1 Abstract Syntax Rules

3.1.1 PROGRAM

A1. <program> ::= {<abstract-external-procedure-list>}

A2. <abstract-external-procedure-list> ::= {<declaration-list>} <procedure>

3.1.2 PROCEDURE

A3. <procedure> ::= {<declaration-list>} {<procedure-list>} {<format-statement-list>} {<condition-prefix-list>} {<recursive>} {<entry-or-executable-unit-list>}

3.1.3 DECLARATION

A4. <declaration> ::= <identifier> <scope> <declaration-type> {<declaration-designator>}

A5. <scope> ::= <external> | <internal>

A6. <declaration-type> ::= <variable> | <named-constant> | <nullref> | <condition>
3.1.4 VARIABLE

A7.  <variable>::= <storage-type> <data-description>
A8.  <storage-type>::= <storage-class> | <defined> | <parameter>
A9.  <storage-class>::= <automatic> | <base> | <controlled> | <static>
A10. <defined>::= {value-reference} {scalar & locator} | {reference-designator}
A11. <base>::= {base-item} {position}
A12. <base-item>::= {variable-reference} {<defined & <base>}
     | {reference-designator}
A13. <position>::= <expression> {scalar & computational-type} | {expression-designator}

3.1.5 DATA-DESCRIPTION

A15.  <dimensioned-data-description>::= <element-data-description> <bound-pair-list>
A17.  <bound-pair>::= <lower-bound> <upper-bound> | <asterisk>
A18.  <lower-bound>::= <extent-expression>
A19.  <upper-bound>::= <extent-expression>
A20.  <extent-expression>::= {expression} {scalar & computational-type} | <expression-designator> | {integer-value} {<refer-option>}
A21.  <refer-option>::= <identifier-list> | <integer-value>
A22.  <structure-data-description>::= {<identifier-list>} <member-description-list>
A23.  <member-description>::= <data-description>
A24.  <item-data-description>::= {<alignment>} <data-type> {<initial>}

Constraint: An <item-data-description> must not have an <initial> component with an <iteration-factor>, or an <initial-element-list> with more than one <initial-element> immediate component, unless the <item-data-description> is a (not necessarily immediate) component of a <dimensioned-data-description>.

A25.  <initial>::= <initial-element-list> | {initial-designator}
A26.  <initial-element>::= <asterisk> | <parenthesized-expression> {scalar} | <iteration-factor> <initial-element-list>
A27.  <iteration-factor>::= <expression> {scalar & computational-type}
A28.  <alignment>::= <aligned> | <unaligned>
3.1.6 DATA-TYPE

A29. <data-type>::= <computational-type> | <non-computational-type>
A30. <computational-type>::= <arithmetic> | <string> | <picture>
A31. <non-computational-type>::= <area> | <entry> | <file> |
    <format> [(local)] | <label> [(local)] | <locator>
A32. <arithmetic>::= <mode> <base> <scale> <precision>
A33. <mode>::= <real> | <complex>
A34. <base>::= <binary> | <decimal>
A35. <scale>::= <fixed> | <float>
A36. <precision>::= <number-of-digits> [scale-factors]
A37. <number-of-digits>::= <integer>
    Constraint: The <integer> must not be zero.
A38. <scale-factors>::= <signed-integer>
A39. <string>::= <string-type> <maximum-length> [varying] [unvarying]
A40. <string-type>::= <character> | <bit>
A41. <maximum-length>::= <extent-expression> [asterisk]
A42. <picture>::= <pictured-character> | <pictured-numeric>
A43. <locator>::= <pointer> | <offset>
A44. <offset>::= [[variable-reference] {scalar & area} | {reference-designator}]
A45. <entry>::= [parameter-descriptor-list] {returns-descriptor} [options]
A46. <parameter-descriptor>::= <data-description>
A47. <returns-descriptor>::= <data-description>
A48. <options>::= 
    This category is implementation-defined.
A49. <area>::= <area-size>
A50. <area-size>::= <extent-expression> [asterisk]

3.1.7 NAMED-CONSTANT

A51. <named-constant>::= <entry> | <file> <file-description> | <format> | <label>
    [bound-pair-list]
A52. <file-description>::= [at-name] [record] [input] [output] [update]
    [sequential] [direct] [qsl] [level] [environment]
A53. <environment>::= 
    This category is implementation-defined.
3.1.8 ENTRY-OR-EXECUTABLE-UNIT

A54. `<entry-or-executable-unit>` ::= `<entry-point> | <executable-unit>`

A55. `<entry-point>` ::= `{statement-name} `<entry-information>`

A54. `<statement-name>` ::= `<identifier> `{signed-integer-list}`

A57. `<entry-information>` ::= `{parameter-name-list} `{returns-descriptor} | `{options}`

A58. `<parameter-name>` ::= `<identifier>`

A59. `<executable-unit>` ::= `{condition-prefix-list} `{statement-name-list}`

<table>
<thead>
<tr>
<th><code>begin-block</code></th>
<th><code>call-block</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>allocate-statement</code></td>
<td><code>call-statement</code></td>
</tr>
<tr>
<td><code>assignment-statement</code></td>
<td><code>close-statement</code></td>
</tr>
<tr>
<td><code>copy-statement</code></td>
<td><code>delete-statement</code></td>
</tr>
<tr>
<td><code>end-statement</code></td>
<td><code>end-statement</code></td>
</tr>
<tr>
<td><code>free-statement</code></td>
<td><code>free-statement</code></td>
</tr>
<tr>
<td><code>if-statement</code></td>
<td><code>if-statement</code></td>
</tr>
<tr>
<td><code>locate-statement</code></td>
<td><code>locate-statement</code></td>
</tr>
</tbody>
</table>

3.1.9 BEGIN-BLOCK

A60. `<begin-block>` ::= `{declaration-list} `{procedure-list} `{format-statement-list} `{options}` `<executable-unit-list>`

3.1.10 GROUPS

A61. `<group>` ::= `<iterative-group> | <non-iterative-group>`

A62. `<iterative-group>` ::= `<controlled-group> | <while-only-group>`

A63. `<controlled-group>` ::= `{do-spec} `<executable-unit-list>`

A64. `<do-spec>` ::= `<target-reference> `{scalar} `<spec-list>`

A65. `<spec>` ::= `<expression> `{scalar} `{to-by} | `<repeat-option> | `{while-option>`

A66. `<to-by>` ::= `<to-option> `{by-option} | `{by-option>`

A67. `<repeat-option>` ::= `<expression> `{scalar>`

A68. `<by-option>` ::= `<expression> `{scalar & computational-type}`

A69. `<to-option>` ::= `<expression> `{scalar & computational-type}`

A70. `<while-option>` ::= `<expression> `{scalar & computational-type}`

A71. `<while-only-group>` ::= `<while-option> `<executable-unit-list>`

A72. `<non-iterative-group>` ::= `<entry-or-executable-unit-list>`

3.1.11 ON STATEMENT

A73. `<on-statement>` ::= `{condition-name-list} `{map} | `{con-unit} | `{system-action}`

A74. `<on-unit>` ::= `<procedure>`
3.1.12 IF STATEMENT
A75. <if-statement>::= <test> <then-unit> [else-unit]
A76. <test>::= <expression>  (scalar & computational-type)
A77. <then-unit>::= <executable-unit>
A78. <else-unit>::= <executable-unit>

3.1.13 FLOW OF CONTROL STATEMENTS
A79. <call-statement>::= <subroutine-reference>
A80. <goto-statement>::= <value-reference> (scalar & label)
A81. <return-statement>::= [<expression>]
A82. <revert-statement>::= <condition-name-list>
A83. <signal-statement>::= <condition-name>
A84. <condition-name>::= <computational-condition> | <named-in-condition> |
| <programmer-named-condition> | <area-condition> |
| <error-condition> | <finish-condition> | <storage-condition>
A85. <condition-prefix>::= <computational-condition>  [:] (enabled) [:] (disabled)
A86. <computational-condition>::= <conversion-condition> | <overflow-condition> |
| <underflow-condition> | <size-condition> |
| <integer-range-condition> | <attributed-range-condition> |
| <integer-size-condition> | <underflow-condition> |
| <aerosize-condition>
A87. <named-in-condition>::= <in-condition> <value-reference> (scalar & file)
A88. <in-condition>::= <endfile-condition> | <endpage-condition> | <key-condition> |
| <name-condition> | <record-condition> | <transmit-condition> |
| <underflow-condition>
A89. <programmer-named-condition>::= <declaration-designator> (condition)

3.1.14 STORAGE STATEMENTS
A90. <assignment-statement>::= <target-reference-list> <expression>
A91. <allocate-statement>::= <allocation-list>
A92. <allocation>::= <declaration-designator> (based | controlled)
| <set-option> | <in-option>
A93. <set-option>::= <variable-reference> (scalar & locator)

Constraints: The <data-description> immediate component of the <variable-reference>
must not have <offset> without a <variable-reference> subnode.
A94. <in-option>::= <variable-reference> (scalar & area)
A95. <free-statement>::= <freeing-list>
A96. <freeing>::= <locator-qualifier> <declaration-designator> (based | controlled)
| <in-option>

Chapter 3: Abstract Syntax  55
3.1.15 I/O STATEMENTS

A97. <open-statement> ::= <single-opening-list>

A98. <single-opening> ::= <file-option> {<tab-option> | <title-option>}

A99. <file-option> ::= <value-reference> {<classname-option> | <pagesize-option> | <stream-option> | <record-option> | <input-option> | <output-option> | <update-option> | <sequential-option> | <direct-option> | <print-option> | <keyset-option> | <environment-option>}

A100. <tab-option> ::= <expression-list> {<scalar-option> | <computational-option>}

A101. <title-option> ::= <expression> {<scalar-option> | <computational-option>}

A102. <pagesize-option> ::= <expression> {<scalar-option> | <computational-option>}

A103. <record-option> ::= <expression> {<scalar-option> | <computational-option>}

A104. <stream-option> ::= <single-closing-list>

A105. <single-closing> ::= <file-option> | <environment-option>

3.1.16 RECORD I/O STATEMENTS

A106. <delete-statement> ::= <file-option> | <key-option>

A107. <locate-statement> ::= <declaration-designator> {<based> | <file-option> | <pointer-set-option> | <keyfrom-option>}

A108. <pointer-set-option> ::= <variable-reference> {<scalar-option> | <pointer-option> | <ignore-option> | <key-option> | <keyset-option>}

A109. <read-statement> ::= <file-option>

A110. <write-statement> ::= <file-option>

A111. <write-convert-statement> ::= <file-option> {<key-option> | <from-option>}

A112. <from-option> ::= <variable-reference>

A113. <key-option> ::= <expression> {<scalar-option> | <computational-option>}

A114. <keyfrom-option> ::= <expression> {<scalar-option> | <computational-option>}

A115. <write-key-option> ::= <expression> {<scalar-option> | <computational-option>}

A116. <write-keyset-option> ::= <expression> {<scalar-option> | <character-option>}

3.1.17 STREAM I/O STATEMENTS

A117. <get-statement> ::= <get-file> | <get-string>

A118. <get-file> ::= <file-option> {<copy-option> | <skip-option> | <input-specification>}

A119. <copy-file> ::= <file-option> {<copy-option> | <skip-option> | <input-specification>}

A120. <skip-option> ::= <expression> {<scalar-option> | <computational-option>}

A121. <copy-option> ::= <value-reference> {<scalar-option> | <file-option>}

56
A122. <get-string> ::= <expression> (scalar & computational-type) <input-specification> [copy-option]
A123. <put-string> ::= <get-file> | <get-string>
A124. <get-file> ::= <file-option> (skip-option?) (line-option?) (query) [output-specification]
Constraint: At least one of the last four options must be present and the skip-option must not be used together with a line-option or query.
A125. <line-option> ::= <expression> (scalar & computational-type)
A126. <get-string> ::= <target-reference> (scalar & character) <output-specification>
A127. <input-specification> ::= <data-directed-input> | <list-directed-input> | <edit-directed-input>
A128. <data-directed-input> ::= [<data-target-list>]
A129. <data-target> ::= <variable-reference> (computational-type)
Constraint: The variable-reference must not contain a <locator-qualifier> or a <subscript-list>, and must not contain a <by-name-para-list>.
A130. <list-directed-input> ::= <input-target-list>
A131. <input-target> ::= <target-reference> (computational-type) | <input-target-list> <do-spec>
A132. <edit-directed-input> ::= <edit-input-pair-list>
A133. <edit-input-pair> ::= <input-target-list> <format-specification-list>
A134. <output-specification> ::= <data-directed-output> | <list-directed-output> | <edit-directed-output>
A135. <data-directed-output> ::= [<data-source-list>]
A136. <data-source> ::= <variable-reference> (computational-type) | <data-source-list> <do-spec>
Constraint: The variable-reference must not have a <locator-qualifier>.
A137. <list-directed-output> ::= <output-source-list>
A138. <output-source> ::= <expression> (computational-type) | <output-source-list> <do-spec>
A139. <edit-directed-output> ::= <edit-output-pair-list>
A140. <edit-output-pair> ::= <output-source-list> <format-specification-list>
A141. <format-specification> ::= <format-item> | <format-iteration>
A142. <format-iteration> ::= <format-iteration-factor> <format-specification-list>
A143. <format-iteration-factor> ::= <expression> (scalar)
A144. <format-item> ::= <data-format> | <control-format> | <remote-format>
A145. <data-format> ::= <real-format> | <complex-format> | <picture-format> | <string-format>
A146. <real-format> ::= <fixed-point-format> | <floating-point-format>
A147. <fixed-point-format> ::= <expression> (scalar & computational-type) | <integer-value> | <expression> (scalar & computational-type) | <integer-value> | <expression> (scalar & computational-type) | <integer-value>
\[8. \quad \langle\text{floating-point-format}\rangle::=\begin{cases} \langle\text{expression}\rangle \quad \langle\text{scalar} \& \text{computational-type}\rangle \\
\langle\text{integer-value}\rangle \\
\langle\text{expression}\rangle \quad \langle\text{scalar} \& \text{computational-type}\rangle \\
\langle\text{integer-value}\rangle \\
\langle\text{expression}\rangle \quad \langle\text{scalar} \& \text{computational-type}\rangle \\
\langle\text{integer-value}\rangle \\
\end{cases}\]

\[9. \quad \langle\text{complex-format}\rangle::=\langle\text{real-format}\rangle | \langle\text{picture-format}\rangle
\langle\text{real-format}\rangle | \langle\text{picture-format}\rangle
\]

Constraint: A \langle\text{complex-format}\rangle must not contain \langle\text{pictured-character}\rangle.

\[10. \quad \langle\text{picture-format}\rangle::=\langle\text{pictured}\rangle
\]

\[11. \quad \langle\text{string-format}\rangle::=\langle\text{character-string}\rangle | \langle\text{bit-format}\rangle
\]

\[12. \quad \langle\text{character-string}\rangle::=\begin{cases} \langle\text{expression}\rangle \quad \langle\text{scalar} \& \text{computational-type}\rangle \\
\langle\text{integer-value}\rangle \\
\end{cases}\]

\[13. \quad \langle\text{bit-format}\rangle::=\langle\text{radix-factor}\rangle | \begin{cases} \langle\text{expression}\rangle \quad \langle\text{scalar} \& \text{computational-type}\rangle \\
\langle\text{integer-value}\rangle \\
\end{cases}\]

\[14. \quad \langle\text{radix-factor}\rangle::=1 \mid 2 \mid 3 \mid 4 \mid 8\]

\[15. \quad \langle\text{control-format}\rangle::=\langle\text{tab-format}\rangle | \langle\text{line-format}\rangle | \langle\text{space-format}\rangle | \langle\text{skip-format}\rangle | \langle\text{comma-format}\rangle | \langle\text{page}\rangle
\]

\[16. \quad \langle\text{tab-format}\rangle::=\begin{cases} \langle\text{expression}\rangle \quad \langle\text{scalar} \& \text{computational-type}\rangle \\
\langle\text{integer-value}\rangle \\
\end{cases}\]

\[17. \quad \langle\text{line-format}\rangle::=\begin{cases} \langle\text{expression}\rangle \quad \langle\text{scalar} \& \text{computational-type}\rangle \\
\langle\text{integer-value}\rangle \\
\end{cases}\]

\[18. \quad \langle\text{space-format}\rangle::=\begin{cases} \langle\text{expression}\rangle \quad \langle\text{scalar} \& \text{computational-type}\rangle \\
\langle\text{integer-value}\rangle \\
\end{cases}\]

\[19. \quad \langle\text{skip-format}\rangle::=\begin{cases} \langle\text{expression}\rangle \quad \langle\text{scalar} \& \text{computational-type}\rangle \\
\langle\text{integer-value}\rangle \\
\end{cases}\]

\[20. \quad \langle\text{comma-format}\rangle::=\begin{cases} \langle\text{expression}\rangle \quad \langle\text{scalar} \& \text{computational-type}\rangle \\
\langle\text{integer-value}\rangle \\
\end{cases}\]

\[21. \quad \langle\text{remote-format}\rangle::=\begin{cases} \langle\text{variable-reference}\rangle \quad \langle\text{scalar} \& \text{format}\rangle \\
\langle\text{named-constant-reference}\rangle \quad \langle\text{scalar} \& \text{format}\rangle \\
\end{cases}\]

\[22. \quad \langle\text{format-statement}\rangle::=\begin{cases} \langle\text{condition-prefix-list}\rangle \\
\langle\text{statement-name-list}\rangle \\
\langle\text{format-specification-list}\rangle \\
\end{cases}\]

3.1.18 EXPRESSION

\[23. \quad \langle\text{expression}\rangle::=\begin{cases} \langle\text{value-reference}\rangle | \langle\text{constant}\rangle | \langle\text{list}\rangle \\
\langle\text{infix-expression}\rangle | \langle\text{prefix-expression}\rangle \\
\langle\text{parenthesized-expression}\rangle | \langle\text{data-description}\rangle \\
\end{cases}\]

\[24. \quad \langle\text{infix-expression}\rangle::=\langle\text{expression}\rangle \quad \langle\text{infix-operator}\rangle \quad \langle\text{expression}\rangle \quad \langle\text{data-description}\rangle
\]

\[25. \quad \langle\text{infix-operator}\rangle::=\langle\text{or}\rangle | \langle\text{and}\rangle | \langle\text{eq}\rangle | \langle\text{lt}\rangle | \langle\text{le}\rangle | \langle\text{gt}\rangle | \langle\text{ge}\rangle | \langle\text{add}\rangle | \langle\text{sub}\rangle | \langle\text{mult}\rangle | \langle\text{div}\rangle | \langle\text{pow}\rangle
\]

\[26. \quad \langle\text{prefix-expression}\rangle::=\langle\text{prefix-operator}\rangle \quad \langle\text{expression}\rangle \quad \langle\text{data-description}\rangle
\]

\[27. \quad \langle\text{prefix-operator}\rangle::=\langle\text{plus}\rangle | \langle\text{minus}\rangle | \langle\text{not}\rangle
\]

\[28. \quad \langle\text{parenthesized-expression}\rangle::=\langle\text{expression}\rangle \quad \langle\text{data-description}\rangle
\]
### 3.1.19 Types of Reference

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>value-reference</code></td>
<td>A reference to a variable, procedure function, or built-in function.</td>
</tr>
<tr>
<td><code>variable-reference</code></td>
<td>A reference to a variable.</td>
</tr>
<tr>
<td><code>procedure-function-reference</code></td>
<td>A reference to a procedure or function.</td>
</tr>
<tr>
<td><code>named-constant-reference</code></td>
<td>A reference to a named constant.</td>
</tr>
</tbody>
</table>

#### 3.1.19.1 By-name parts

- `identifier-list`:
  - A list of identifiers.
- `subscript-list`:
  - A list of subscripts.
- `by-name-parts-list`:
  - A list of by-name parts.

#### 3.1.19.2 Arguments

- `argument-list`:
  - A list of arguments.
- `argument`:
  - An argument.
  - `expression`:
    - An expression.
    - `integer-value`:
      - An integer value.

#### 3.1.19.3 Built-in Function Reference

- `built-in-function`:
  - A built-in function.
  - Examples:
    - `abs-bit`...
    - `empty-bit`...
    - `memry-bit`...

#### 3.1.19.4 Named Constant Reference

- `named-constant-reference`:
  - A reference to a named constant.

#### 3.1.19.5 Target Reference

- `target-reference`:
  - A reference to a variable or pseudo-variable.

#### 3.1.19.6 Pseudo-Variable Reference

- `pseudo-variable-reference`:
  - A reference to a pseudo-variable.

#### 3.1.19.7 Subroutine Reference

- `subroutine-reference`:
  - A reference to a subroutine.
3.1 Constant and Enum

A183. <constant> ::= <basic-value> <data-type>
A184. <enum> ::= <integer>

3.1.21 Types of Value

A185. <identifier> ::= ...
A186. <signed-integer> ::= [ + | - ] <integer>
A187. <integer> ::= ...

The two categories, A185 and A187, are defined as {symbol-list}s corresponding to
the sequences of characters in an {identifier} or {integer} respectively. See rules
CLS and CLS5 in Chapter 2.

3.1.22 Types of Picture

A188. <picture-character> ::= <character-picture-element-list>
A189. <character-picture-element> ::= A | X | 9
A190. <picture-numeric> ::= <numeric-picture-specification> <arithmetic>
A191. <numeric-picture-specification> ::= <fixed-point-picture> [ <picture-scale-factor> ]
| <floating-point-picture>
A192. <fixed-point-picture> ::= <numeric-picture-element-list>
A193. <floating-point-picture> ::= <picture-mantissa> <picture-exponent>
A194. <picture-mantissa> ::= <numeric-picture-element-list>
A195. <picture-exponent> ::= <numeric-picture-element-list>
A196. <picture-scale-factor> ::= <signed-integer>
A197. <numeric-picture-element> ::= X | I | B | N | T | V | Y | Z | $ | 9 | * | = | |
| <insertion-character> | <credit> | <debit>
A198. <insertion-character> ::= R | ` | - |
A199. <credit> ::= CR
A200. <debit> ::= DE
The abstract text corresponding to the concrete representation

\[ \text{GO TO } P \rightarrow S \rightarrow W \]

is as follows:

```plaintext
<goto-statement>
  <value-reference>
  <variable-reference>

<locator-qualifier>
  <declaration-designator>
    <identifier-list>
      <identifier>

<declaration-designator>
  <for P>

Note: <value-reference> and <variable-reference> nodes immediately contain a <data-description> (not shown).
```

The abstract text corresponding to the concrete representation

\[ \text{DECLARE } Z \text{ EXTERNAL STATIC ALIGNED COMPLEX DECIMAL FLOAT(15);} \]

is as follows:

```plaintext
<Declaration>

<identifier>
  <scope> <declaration-type>
    <external> <variable>

<storage-type>
  <data-description>
    <item-data-description>
      <static>

<alignment>
  <data-type>
    <alignment>
      <computational-type>
        <arithmetic>

<mode>
  <class>
    <scale>
      <precision>
        <computed>
          <decimal> <list> <number-of-digits>
            <integer>
              15
```

Example 3.1. Examples of Abstract Text.
Chapter 4: The Translator

4.0 Introduction

This Chapter defines an operation which translates a \{symbol-list\} into an \{abstract-external-procedure\}. All the \{abstract-external-procedures\} that are part of a PL/I \{program\} are then combined. The \{program\} is used by the PL/I machine to determine the course of execution.

4.1 Translate

A concrete-block is a \{begin-block\}, \{procedure\}, or \{concrete-external-procedure\}.

x is a concrete-block-component of y if y is a concrete-block, and y contains x but does not contain any other concrete-block which also contains x. In this case y concrete-block-contains x.

An abstract-block is a \{begin-block\}, \{procedure\}, or \{abstract-external-procedure\}.

y is an abstract-block-component of y if y is an abstract-block, and y contains x but does not contain any other abstract-block which also contains x. In this case y abstract-block-contains x.

The informal term block-component is used for either concrete-block-component or abstract-block-component, and block-contains is used for either concrete-block-contains or abstract-block-contains where the context makes it obvious which formal term is required.

A \{declaration\}, \{description\}, \{default-attributes\} or \{generic-description\}, d declaration-contains a node a, if d contains a and d does not contain a \{description\} or \{generic-description\} that also contains a.

A node, a, is a declaration-component of a node, d, if d declaration-contains a.

Operation: \texttt{translate}(t)

where t is a \{symbol-list\}.

result: an \{abstract-external-procedure\}.

Step 1. Perform \texttt{parse}(t, procedure) to obtain a \{procedure\}. Append a \{concrete-external-procedure\}: cep to the \{translation-state\}.

Step 2. Perform complete-concrete-procedure.

Step 3. Let cep be an abstract-external-procedure.

Step 4. For each \{declaration\},d which is a block-component of the \{concrete-external-procedure\} perform create-declaration(id) to obtain a \{declaration\},ad, and append ad to the \{declaration-list\} in cep.

Step 5. For each \{declaration\},d which is a block-component of cep and which contains at least one \{expression-designator\} or \{reference-designator\}, perform replace-concrete-designator(d).

Step 6. Let p be the \{procedure\} immediate component of the \{concrete-external-procedure\}. Perform create-procedure(p) to obtain a \{procedure\},ap, and attach ap to cep.

Step 7. Delete the \{concrete-external-procedure\}.


Step 9. Return cep.
4.2 Forming the Concrete Procedure

The parse operation is applied to a `symbol-list` to construct a complete tree with respect to the Concrete Syntax for a specified category-name. If this category-name is defined in the high-level syntax or the middle-level syntax then some additional mapping at the interfaces between these syntaxes is required.

If parse is called for a `procedure`, it calls itself recursively to build trees consistent with the low-level, middle-level, and high-level syntaxes, in that order.

Operation: `parse(sl,n)`

where `sl` is a `symbol-list`,

n is a tree with a single node, whose type is a non-terminal category

in the Concrete Syntax.

result: a complete tree with respect to the Concrete Syntax for n.

Case 1. The type of n is a non-terminal of the high-level syntax.

Perform `parse(sl, sentence-list)` to obtain a `sentence-list`, sl. Perform high-level-parse(sl,n) to obtain nt.

Return nt.

Case 2. The type of n is a non-terminal of the middle-level syntax.

Perform `parse(sl, p(p1-text))` to obtain a `p(p1-text)`, pt. Perform middle-level-parse(pt,n) to obtain nt.

Return nt.

Case 3. The type of n is a non-terminal of the low-level syntax.

Perform low-level-parse(sl,n) to obtain nt.

Return nt.

4.2.1 LOW-LEVEL-PARSE

Operation: `low-level-parse(sl,n)`

where `sl` is a `symbol-list`,

n is a tree with a single node, whose type is a non-terminal category-name at the low-level syntax.

result: a complete tree with respect to the low-level syntax for n.

Step 1. There must exist one and only one tree, nt, which is a complete tree with respect to the low-level syntax for n, such that the following conditions are true:

(1) the concrete-representation of nt is exactly the same as the concrete-representation of sl, and

(2) every occurrence of `{/}` or `{*}` in the concrete-representation of nt must be such that the `{}` and `{*}` are nodes of a `{comment}` category or are contained in a `{delimiter-pair}`, and

(3) of all possible trees satisfying conditions (1) and (2), nt is that one containing the least number of `{delimiter-pair}`s and `{delimiter}`s.

Step 2. Return nt.
4.2.2 MIDDLE-LEVEL-PARSE

A keyword is a category-name specified in the middle-level syntax as a sequence of uppercase letters.

$\textbf{delimiter-or-non-delimiter} ::= \textbf{delimiter} | \textbf{non-delimiter}$

Operation: middle-level-parse(pt, cm)

where pt is a $\textbf{pul-test}$, cm is a tree with a single node, whose type is a non-terminal of the middle-level syntax.

results: a complete tree with respect to the syntax composed of all the production-rules occurring in the middle-level syntax and in the low-level syntax with the root-node cm.

Step 1. Let t be a $\textbf{delimiter-or-non-delimiter-list}$ which contains a copy of the $\textbf{delimiter}$ and $\textbf{non-delimiter}$ components of pt in the same order.

Step 2. Repeat Steps 2.1 through 2.5 as long as there is a $\textbf{delimiter-or-non-delimiter}$ of $\textbf{delimiter}$ and $\textbf{non-delimiter}$ components of pt in the same order.

Step 2.1. Let x be $\textbf{symbol-list}: \textbf{symbol}: x$.

Step 2.2. Append to x any $\textbf{symbol-list}$ obtained in an implementation-defined way from the $\textbf{test-name}$ in d. Append $\textbf{symbol}: x$ to x.

Step 2.3. Perform low-level-parse(s, $\textbf{pul-test}$) to obtain a $\textbf{pul-test}: tk$.

Step 2.4. Let tl be a $\textbf{delimiter-or-non-delimiter-list}$ which contains a copy of the $\textbf{delimiter}$ and $\textbf{non-delimiter}$ components of tk in the same order.

Step 2.5. Replace d by the immediate components of tl in the same order.

Step 3. Delete from t any $\textbf{delimiter}$ containing a $\textbf{#}$ or a $\textbf{comment}$. This must not cause t to be deleted.

Step 4. Let ntill, i=1,...,n, be the ordered list of nodes which are the immediate components of the $\textbf{delimiter}$ and $\textbf{non-delimiter}$ in t.

Step 5. There must exist one and only one tree, at, which is a complete tree with respect to the middle-level syntax for the root-node cm and satisfies any additional constraints specified for that syntax, and which is such that it contains a terminal nodes ntij, j=1,...,m and there is a one-to-one correspondence between the ntij, j=1,...,m and ntij, j=1,...,m taken in left-to-right order as specified by Case 5.1 through Case 5.5 except for the following instance:

if ntij is in an $\textbf{environment-specification}$ or an $\textbf{option-specification}$ then it corresponds to k nodes atill, i=1,...,i+k-1 such that

(i) if still, i=1,...,i+k-1 is a $\textbf{#}$;

(ii) all nodes atill, i=1,...,i+k-1 which are either a $\textbf{#}$ or a $\textbf{#}$ must be matched in the normal way for balancing parentheses.

Case 5.1. ntij is a keyword.

atill must be an $\textbf{identifer}$ containing the same terminals as the characters appearing either in the denotation of ntij or in the abbreviation for ntij. (See Section 2.7.)

Case 5.2. ntij is a non-bracketed category-name other than a keyword.

ntij and still must be equal.

Chapter 4: The Translator
Case 5.2. $\text{nti}[	ext{j}]$ is a $\text{radix-factor}$. 

$\text{nti}[	ext{j}]$ must be an $\text{identifier}$ such that the ordered sequence, $\text{seq}$, of its terminals can be a denotation of an immediate component of a $\text{radix-factor}$. 

Replace $\text{nti}[	ext{j}]$ by a $\text{radix-factor}$. $\text{seq}$.

Case 5.4. $\text{nti}[	ext{j}]$ is an $\text{imaginary-constant}$, a $\text{real-constant}$, or an $\text{integer}$. 

$\text{nti}[	ext{j}]$ must be an $\text{arithmetic-constant}$ containing just a node, $\text{lc}$, with the same type as $\text{nti}[	ext{j}]$. (There may be intermediate nodes between $\text{nti}$ and $\text{lc}$, but no side branches.) 

Replace $\text{nti}[	ext{j}]$ by the tree with root-node $\text{lc}$.

Case 5.5. $\text{nti}[	ext{j}]$ is one of the following: 

- $\text{identifier}$ 
- $\text{arithmetic-constant}$ 
- $\text{simple-bit-string-constant}$ 
- $\text{simple-character-string-constant}$ 
- $\text{function}$ 
- $\text{letter}$ 

$\text{nti}$ and $\text{nti}[	ext{j}]$ must be of the same type. 

Replace $\text{nti}[	ext{j}]$ by $\text{nti}$.

Step 6. Each $\text{description}$ and $\text{generic-description}$ must contain a subtree.

Step 7. Return $\text{nti}$.

4.2.3 HIGH-LEVEL-PARSE

Operation: high-level-parse($\text{si}[	ext{cl}]$).

where $\text{si}$ is a $\text{sentence-list}$,

$\text{cl}$ is a tree with a single node, whose type is a non-terminal of the high-level syntax.

result: a complete tree with respect to the Concrete Syntax for $\text{cl}$.

Step 1. For each $\text{single-statement}$, a component of $\text{si}$, in left-to-right order perform steps 1.1 through 1.3.

Step 1.1. If $\text{si}$ contains a $\text{block-statement}$, $\text{do-statement}$, or $\text{procedure-statement}$, then attach $\text{ unmatched}$ to $\text{si}$.

Step 1.2. If $\text{si}$ contains an $\text{end-statement}$ not containing an $\text{identifier}$ then remove the rightmost preceding $\text{ unmatched}$.

Step 1.3. If $\text{si}$ contains an $\text{end-statement}$ containing an $\text{identifier}$, $\text{id}$ then perform steps 1.3.1 and 1.3.2.

Step 1.3.1. Let $\text{rpc}$ be the rightmost preceding $\text{single-statement}$ containing $\text{ unmatched}$ such that $\text{rpc}$ contains also a tree of the form $\text{statement-name}, \text{sn} ; \text{identifier}, \text{id}$ and $\text{id}$ is equal to $\text{id}$. we must not contain a $\text{function-integer-expression-list}$. $\text{rpc}$ must exist.

Step 1.3.2. Let $\text{k}$ be the number of $\text{ unmatched}$ components of $\text{si}$ following $\text{rpc}$ and preceding $\text{si}$. Let $\text{es}$ be a

$\text{sentence}$:

$\text{single-statement}$:

$\text{end-statement}$:

ENQ();

Attach $\text{k}$ copies of $\text{es}$ to $\text{si}$ immediately preceding $\text{si}$. Delete the $\text{k+1}$ instances of $\text{ unmatched}$ which immediately precede $\text{si}$.
Step 2. Let still, i=1,...,n be the ordered sequence of components of \( a_i \) which are such that for each component, \( c_i \), the following conditions are satisfied:

1. The category-name of \( c_i \) is terminal with respect to the middle-level syntax, and
2. \( c_i \) is not contained in any component of \( a_i \) whose category-name is terminal with respect to the middle-level syntax.

Step 3. There must exist one and only one tree, \( h_t \), with root-node of the same type as \( c_t \) such that the following conditions are true:

1. \( h_t \) is a complete tree with respect to the syntax composed of all the production-rules occurring in the high-level syntax, and in the middle-level syntax, and
2. \( h_t \) contains a terminal nodes \( h_{t\alpha_j} \), j=1,...,n, such that for every i, i=1,...,n, the type of the node still is the same as the type of \( h_{t\alpha_j} \).

Step 4. For each i, i=1,...,n, replace \( h_{t\alpha_j} \) by still. Return \( h_t \).
4.3 Completion of the Concrete Procedure

The concrete-external-procedure is "completed" in the sense that all declarations are constructed or completed.

Operation: complete-concrete-procedure

Step 1. Perform reorganize.

Step 2. Perform construct-explicit-declarations.

Step 3. Perform complete-structure-declarations.


Step 5. Perform construct-implicit-declarations.

Step 6. Perform complete-declarations.


4.3.1 REORGANIZE

The concrete-external-procedure is reorganized in various ways to simplify and complete it.

Operation: reorganize

Step 1. Perform complete-options.

Step 2. Perform modify-statement-names.

Step 3. Perform complete-attribute-implications.

Step 4. For each (declaration-list), do immediate component of a (declaration) component of the concrete-external-procedure perform defactor-declarations() to obtain a (declaration-list), and replace ds by ds.

4.3.1.1 Complete-options

Various modifications are made to (put-statement)s, (get-statement)s, and (format-statement)s to complete their options. These are performed before the application of any (default-statement)s.

Operation: complete-options

Step 1. For each (get-statement)ps component of the concrete-external-procedure perform steps 1.1 and 1.2.

Step 1.1. If ps does not contain a (file-option) or a (get-string), then perform parse("FILEISPRINT",{file-option}) to obtain a (file-option), and attach fo to ps.

Step 1.2. If ps contains a (copy-option)co which does not contain a (reference), then perform parse("COPYISPRINT",{copy-option}) to obtain a (copy-option), and replace co by co.

Step 2. For each (put-statement)ps component of the concrete-external-procedure, if ps does not contain a (file-option) or a (put-string) then perform parse("FILEISPRINT",{file-option}) to obtain a (file-option), and attach fo to ps.

Step 3. For each (tab-format)tf component of the concrete-external-procedure where tf does not contain an (expression) perform parse("TAB",{tab-format}) to obtain a (tab-format), and replace tf by tf.
Step 6. For each \texttt{\textbackslash{}skip-option} component of the \texttt{\textbackslash{}concrete-external-procedure}, if \texttt{sk} does not contain an \texttt{\textbackslash{}expression} then perform \texttt{parse("\textbackslash{}SKIP\textbackslash{}","\textbackslash{}skip-option")} to obtain \texttt{\textbackslash{}sk}, and replace \texttt{\textbackslash{}sk} by \texttt{\textbackslash{}sk}.

Step 5. For each \texttt{\textbackslash{}skip-format} component of the \texttt{\textbackslash{}concrete-external-procedure}, if \texttt{sf} does not contain an \texttt{\textbackslash{}expression} then perform \texttt{parse("\textbackslash{}SKIP\textbackslash{}","\textbackslash{}skip-format")} to obtain \texttt{\textbackslash{}sf}, and replace \texttt{\textbackslash{}sf} by \texttt{\textbackslash{}sf}.

Step 6. For each \texttt{\textbackslash{}radix-factor} component of the \texttt{\textbackslash{}concrete-external-procedure}, if \texttt{rf} contains only \texttt{8} then replace \texttt{rf} by a \texttt{\textbackslash{}radix-factor} \texttt{\textbackslash{}8}.

3.1.1.2 \texttt{\textbackslash{}modify-statement-name}

The \texttt{\textbackslash{}statement-name} components of a \texttt{\textbackslash{}declare-statement} or \texttt{\textbackslash{}default-statement} are removed and attached to \texttt{\textbackslash{}null-statement}s. (It is not possible for control to branch to a declaration or a default during execution.) Multiple occurrences of \texttt{\textbackslash{}statement-name}s in the \texttt{\textbackslash{}prefix-list} component of a \texttt{\textbackslash{}procedure}, or in \texttt{\textbackslash{}unit}s which contain \texttt{\textbackslash{}entry-statement}s, are also simplified.

Operation: \texttt{\textbackslash{}modify-statement-name}

Step 1. For each \texttt{\textbackslash{}unit}, a immediate component of a \texttt{\textbackslash{}unit-list}, a component of the \texttt{\textbackslash{}concrete-external-procedure}, where a immediately contains a \texttt{\textbackslash{}statement-name-list}, anl and a \texttt{\textbackslash{}declare-statement} or \texttt{\textbackslash{}default-statement}, perform Steps 1.1 and 1.2.

Step 1.1. Let \texttt{pl} be a \texttt{\textbackslash{}prefix-list}. For each \texttt{\textbackslash{}statement-name}, an of anl perform Step 1.1.1.

Step 1.1.1. Append \texttt{\textbackslash{}prefix} as \texttt{\textbackslash{}mc} to \texttt{pl}, where \texttt{\textbackslash{}mc} is a copy of \texttt{\textbackslash{}mc}.

Step 1.2. Let \texttt{\textbackslash{}un} be a

\begin{verbatim}
\texttt{\textbackslash{}unit}
\texttt{\textbackslash{}executable-unit};
\texttt{pl}
\texttt{\textbackslash{}executable-single-statement};
\texttt{\textbackslash{}null-statement};
\texttt{\textbackslash{};}
\end{verbatim}

Attach \texttt{\textbackslash{}un} to \texttt{\textbackslash{}un} immediately preceding \texttt{\textbackslash{}un}. Delete anl.

Step 2. For each \texttt{\textbackslash{}procedure}, a, where a is a component of the \texttt{\textbackslash{}concrete-external-procedure} and a immediately contains a \texttt{\textbackslash{}prefix-list}, pf, perform Steps 2.1 and 2.2.

Step 2.1. pf must contain at least one \texttt{\textbackslash{}statement-name}. For each \texttt{\textbackslash{}statement-name}, an component of pf, after the leftmost one, perform Steps 2.1.1 through 2.1.3.

Step 2.1.1. Let \texttt{\textbackslash{}mc} be a copy of \texttt{\textbackslash{}mc}. Let \texttt{\textbackslash{}mc} be a

\begin{verbatim}
\texttt{\textbackslash{}unit}
\texttt{\textbackslash{}statement-name-list};
\texttt{\textbackslash{}mc};
\texttt{\textbackslash{}entry-statement-list};
\texttt{\textbackslash{} ENTRY}
\texttt{\textbackslash{};}
\end{verbatim}

Step 2.1.2. If the \texttt{\textbackslash{}procedure-statement} immediate component of pc contains an \texttt{\textbackslash{}entry-information}, an then let \texttt{\textbackslash{}mc} be a copy of \texttt{\textbackslash{}mc}, delete any EXCLUSIVE subnode of \texttt{\textbackslash{}mc}, and attach \texttt{\textbackslash{}mc} to \texttt{\textbackslash{}mc}.

Step 2.1.3. Attach anl to pc as the first component of the \texttt{\textbackslash{}unit-list} immediate component of pc.

Step 2.2. Delete every \texttt{\textbackslash{}statement-name} except the first from pf.

Step 3. For each \texttt{\textbackslash{}unit}, a component of the \texttt{\textbackslash{}concrete-external-procedure}, where a immediately contains a \texttt{\textbackslash{}statement-name-list}, anl and a \texttt{\textbackslash{}entry-statement-list}, an and anl contains more than one \texttt{\textbackslash{}statement-name}, perform Steps 3.1 and 3.2.
Step 3.1. For each $\text{statement-name}$, as component of a list after the first, perform Steps 3.1.1 and 3.1.2.

Step 3.1.1. Let $\text{sec}$ be a copy of $\text{sn}$. Let $\text{enc}$ be a copy of $\text{es}$. Let $\text{snl}$ be a $\{\text{unit}\}$:

$\langle \text{statement-name-list} \rangle$

$\langle \text{snl} \rangle$

$\langle \text{enc} \rangle$

Step 3.1.2. Attach $\langle \text{snl} \rangle$ to the $\langle \text{unit-list} \rangle$ which immediately contains $\langle \text{snl} \rangle$ so that $\langle \text{snl} \rangle$ immediately precedes $\langle \text{sn} \rangle$.

Step 3.2. Delete every $\langle \text{statement-name} \rangle$ from $\langle \text{snl} \rangle$ except the first.

4.3.1.3 Complete-attribute-implications

The $\langle \text{dimension-attribute} \rangle$, $\langle \text{precision} \rangle$, and the $\langle \text{data-attribute} \rangle$ FIXED can be implied without the use of the keywords DIMENSION, PRECISION, and FIXED. These implications are replaced by explicit declarations of those attributes.

Operation: complete-attribute-implications

Step 1. For each $\langle \text{dimension-suffix} \rangle$,ds component of the $\langle \text{concrete-external-procedure} \rangle$

such that $\langle \text{ds} \rangle$ is not a component of a $\langle \text{dimension-attribute} \rangle$ append a $\langle \text{data-attribute} \rangle$

$\langle \text{dimension-attribute} \rangle$

$\langle \text{dimension} \rangle$

$\langle \text{ds} \rangle$

where $\langle \text{ds} \rangle$ is a copy of $\langle \text{ds} \rangle$, to the node immediately containing $\langle \text{ds} \rangle$, and delete $\langle \text{ds} \rangle$.

Step 2. For each $\langle \text{generic-declaration} \rangle$,gs component of the $\langle \text{concrete-external-procedure} \rangle$, if $\langle \text{gs} \rangle$ immediately contains an $\langle \text{generic-bound} \rangle$,ab then append a $\langle \text{generic-data-attribute} \rangle$, DIMENSION $\langle \text{ab} \rangle$ to the $\langle \text{generic-data-attribute-list} \rangle$ component of $\langle \text{gs} \rangle$ and delete $\langle \text{ab} \rangle$.

Step 3. For each $\langle \text{attribute} \rangle$, $\langle \text{data-attribute} \rangle$,atr or $\langle \text{data-attribute} \rangle$,atr which is an immediate component of a list, l, component of the $\langle \text{concrete-external-procedure} \rangle$, if $\langle \text{atr} \rangle$ simply contains a $\langle \text{precision} \rangle$,p not not PRECISION then append to l a $\langle \text{data-attribute} \rangle$

$\langle \text{precision} \rangle$

$\langle \text{pl} \rangle$

where $\langle \text{pl} \rangle$ is a copy of $\langle \text{p} \rangle$, and delete $\langle \text{p} \rangle$.

Step 4. For each $\langle \text{data-attribute-list} \rangle$,al or $\langle \text{attribute-list} \rangle$,al of the $\langle \text{concrete-external-procedure} \rangle$, if $\langle \text{al} \rangle$ simply contains a $\langle \text{data-attribute} \rangle$ with $\langle \text{precision} \rangle$,p and $\langle \text{p} \rangle$ contains a $\langle \text{scale-factor} \rangle$ then append to $\langle \text{al} \rangle$ a $\langle \text{data-attribute} \rangle$

$\langle \text{fixed} \rangle$. 

70
The syntax of \textit{declare-statement} allows \textit{identifier}s to be factored together to give them the same structuring or attributes. This factoring is unravelled to provide a single \textit{declaration} for each \textit{identifier}.

\textbf{Operation:} \texttt{defactor-declarations(dcl)}

\hspace{1em} where \texttt{dcl} is a \textit{declaration-comma-list}.

\hspace{1em} result: a \textit{declaration-comma-list}.

\textbf{Step 1.} Let \texttt{dcl} be a copy of \texttt{dcl}.

\textbf{Step 2.} For each \textit{declaration}, \texttt{d} immediate component of \texttt{dcl}, if \texttt{d} immediately contains a \textit{declaration-comma-list}, \texttt{dcl} then perform \texttt{defactor-declarations(dcl)} to contain a \textit{declaration-comma-list}, \texttt{dcl} and replace \texttt{dcl} by \texttt{dcl}.

\textbf{Step 3.} For each \textit{declaration}, \texttt{d} immediate component of \texttt{dcl}, if \texttt{d} immediately contains a \textit{declaration-comma-list}, \texttt{dcl} then perform \texttt{Steps 3.1 through 3.3}.

\textbf{Step 3.1.} If \texttt{d} immediately contains a \textit{level}, \texttt{lv} then there must not be a \textit{level} declaration-contained in any of the \textit{declaration}s immediately contained in \texttt{d}, and attach a copy of \texttt{lv} to each \textit{declaration} immediate component of \texttt{d}.

\textbf{Step 3.2.} If \texttt{d} immediately contains an \textit{attributes-list}, \texttt{al} then append a copy of each \textit{attributes} of \texttt{al} to each \textit{declaration} immediate component of \texttt{d}.

\textbf{Step 3.3.} Replace \texttt{d} by the immediate components of \texttt{dcl} in the same order.

\textbf{Step 4.} Return \texttt{dcl}.

\textbf{4.3.2} \texttt{construct-explicit-declarations}

The occurrence of an \textit{identifier} in the \textit{concrete-external-procedure} as an immediate component of a \textit{declaration} explicitly specifies that \textit{identifier} as the name of some data item. Other contexts may also constitute explicit declarations of an \textit{identifier}, in which case a \textit{declaration} is created for it.

\textit{declared-statement-names} ::= \{\textit{name-list}\} \{\textit{procedure} \mid \textit{begin-block}\}

\textit{name} ::= \textit{statement-name} \{\textit{entry-information}\}

\textbf{Operation:} \texttt{construct-explicit-declarations}

\textbf{Step 1.} For each \textit{procedure}, \texttt{p} contained in the \textit{concrete-external-procedure} perform \texttt{declare-parameters(p)}.

\textbf{Step 2.} Let \texttt{p} be the \textit{procedure} immediate component of the \textit{concrete-external-procedure}. Perform \texttt{declare-statement-names(p)} to obtain a \textit{declared-statement-names}\texttt{dnn}, \texttt{dnn}. Replace \texttt{p} by the \textit{procedure} of \texttt{dnn}. The \textit{name-list}, \texttt{al} of \texttt{dnn} must not contain a \textit{signed-integer-comma-list}.

\textbf{Step 3.} Perform \texttt{construct-statement-name-declarations(in, attr)} where \texttt{in} is an\begin{itemize}
    \item \textit{attributes},
    \item \textit{data-attributes},
    \item \texttt{ENTRY}()
\end{itemize}

\texttt{to obtain a \textit{suite} containing a \textit{declaration-comma-list}, \texttt{dcl}. Replace all occurrences of \textit{attributes}: \texttt{EXTERNAL} in \texttt{dcl} by \textit{attributes}: \texttt{EXTERNAL}.

\textbf{Step 4.} Attach \texttt{dcl} to the \textit{concrete-external-procedure}.\hfill Chapter 8: The Translator 71
4.3.2.1 Declare-parameters

Each \texttt{identifier} occurring in a \texttt{parameter-name-conma-list} represents an explicit declaration of a parameter. Unless it is already declared, a declaration is introduced for it. If it has been declared erroneously then a conflict will occur when this declaration is being transformed to its abstract equivalent.

Operation: \texttt{declare-parameters\{p\}}

where \( p \) is a \texttt{procedure}.

Step 1. For each \texttt{parameter-name-conma-list}, \( pl \) block-component of \( p \) perform step 1.1.

Step 1.1. \( pl \) must be such that no two \texttt{identifier} components of \( pl \) are equal. For each \texttt{identifier}, id component of \( pl \) perform steps 1.1.1 through 1.1.3.

Step 1.1.1. Perform \texttt{find-applicable-declaration\{id\}} to obtain \( d \).

Step 1.1.2. If \( d \) is \texttt{absent}, or \( d \) is not a concrete-block-component of \( p \), or \( d \) is a concrete-block-component of \( p \) but declaration-contains a \texttt{level} whose value is not \( 1 \), then let \( d \) be a

\begin{verbatim}
$unit$
 DECLARE
$declaration-conma-list$
 \{declaration\}.d:
 $id$
 \end{verbatim}

and append \( d \) to the \texttt{unit-list} of \( p \).

Step 1.1.3. If \( d \) does not contain \texttt{attribute\{PARAMETER\}} then attach \texttt{attribute\{PARAMETER\}} to \( d \).

4.3.2.3 Declare-statement-names

A \texttt{statement-name} may occur as a component of a \texttt{unit} which contains an \texttt{entry-statement}, \texttt{format-statement}, or some other \texttt{presentable-unit}, or as a component of a \texttt{procedure}. These contexts are used to determine the \texttt{attribute\{PARAMETER\}} to be attached to the explicit declarations for the \texttt{identifier} of the \texttt{statement-name}.

Operation: \texttt{declare-statement-names\{p\}}

where \( p \) is a \texttt{procedure} or \texttt{begin-block}.

result: a \texttt{declared-statement-names}.

Step 1. Let \( pc \) be a copy of \( p \) and let \( loi, pai, foi, \text{and} sei \) each be a \texttt{name-list} with no components.

Step 2. For each \texttt{statement-name}, on concrete-block-component of \( pc \) perform step 2.1.

Step 2.1. One of the following Cases must apply:

Case 2.1.1. \( sn \) is simply contained in an \texttt{executable-unit}, etc.

Append \( sn \) to \( loi \).

Case 2.1.2. \( sn \) is simply contained in a \texttt{unit} that has a \texttt{format-statement} immediate component.

Append \( sn \) to \( foi \).

Case 2.1.3. \( sn \) is contained either in a \texttt{prefix-list} immediate component of \( pc \) where \( pc \) has \texttt{procedure-statement}, \texttt{ep}, or in a \texttt{unit} with an \texttt{entry-statement}, \texttt{ep} as an immediate component.
Case 2.1.3.1. \( \text{sp} \) has an \{entry-information\}, \( \text{ei} \).

If \( \text{sp} \) is an \{entry-statement\}, then \( \text{ei} \) must not contain \text{RECURSIVE}. Append \{name\} : \( \text{sp} \) \( \text{ei} \) to \( \text{ei} \).

Case 2.1.3.2. \( \text{sp} \) has no \{entry-information\}.

Append \{name\} : \text{sp} to \( \text{ei} \).

Step 3. For each \{procedure\} or \{begin-block\}, \( \text{pc} \) \text{concrete-block-component} \text{of} \text{pc} \text{perform Step 3.1.}

Step 3.1. Perform \{declaration-statement-names\}(\( \text{pc} \)) to obtain a \{declared-statement-names\}, \( \text{d} \). Append the elements of the \{name-list\} of \( \text{d} \) due to \( \text{p} \). Replace \( \text{p} \) by the \{procedure\} or \{begin-block\} of \( \text{d} \). If \( \text{p} \) is a \{begin-block\}, then the \{name-list\} \text{component} \text{of} \text{d} \text{must be empty}.

Step 4. If \( \text{l} \) is not empty then perform \text{construct-statement-name-declarations}(\( \text{l} \), \text{latr} \), what \text{tr} is an \{attribute\}:

\text{data-attribute:}
\text{LABEL:}

\text{to obtain a \{unit\}, \text{d}, and append \( \text{d} \) to the \{unit-list\} of \text{pc}.

Step 5. If \( \text{f} \) is not empty then perform \text{construct-statement-name-declarations}(\( \text{f} \), \text{fatr} \), what \text{tr} is an \{attribute\}:

\text{data-attribute:}
\text{FORMAT:}

\text{to obtain a \{unit\}, \text{d} and append \( \text{d} \) to the \{unit-list\} of \text{pc}.

Step 6. If \( \text{p} \) is not empty then perform \text{construct-statement-name-declarations}(\( \text{p} \), \text{patr} \), what \text{tr} is an \{attribute\}:

\text{data-attribute:}
\text{ENTRY:}

\text{to obtain a \{unit\}, \text{d} and append \( \text{d} \) to the \{unit-list\} of \text{pc}.

Step 7. Return the \{declared-statement-names\} consisting of \text{ei} and \text{pc}.

8.3.2.3 \text{Construct-statement-name-declarations}

This operation takes a \{name-list\} containing \{statement-names\} and possibly \{entry-information\} and constructs a \{declaration-statement\} for them. The type of the \{statement-name\} is given by a supplied \{attribute\}. Any type of \{statement-name\} may contain one or more \{integer\}, signifying that it is one element of an array of \{statement-name\}'s. The explicit declaration for this array is constructed with a \{dimension-attribute\} component.

Operation: \text{construct-statement-name-declarations}(\text{nl}, \text{att})

where \( \text{nl} \) is a \{name-list\},
\text{att} is an \{attribute\}.

result: a \{unit\}.

Step 1. Let \( \text{nl} \) be a copy of the \{name-list\}, \( \text{nl} \). Let \( \text{ei} \) be a

\{unit\}:
\{declaration-statement\}:
\text{DECLARE}
\{declaration-comma-list\}, \text{dci}
\text{[;]}.

Chapter 8: The Translator 73
Step 2. While alc contains any element, perform Steps 2.1 through 2.3.

Step 2.1. Let id be the {identifier} component of the first element of alc.

Step 2.2. Let tal be a {name-list} containing a copy of all elements of alc whose {statement-name} immediately contains an {identifier} equal to id.

Step 2.3. Let d be a

{declaration};

id
{attribute-list},ai
{attribute};
INTERNAL;
{attribute};
CONSTANT;
att.

Step 2.4. If tal contains more than one element or if tal contains only one element and this element has a {signed-integer-comma-list}, then perform Steps 2.4.1 through 2.4.6.

Step 2.4.1. Let a be the number of elements of tal. Each element of tal must contain a {signed-integer-comma-list} with the same number of elements. a. tal must be such that no two {signed-integer-comma-list} components of tal represent the same ordered sequence of numerical values.

Step 2.4.2. Let sli1,j1 be the j'th {signed-integer} of the {signed-integer-comma-list} of the i'th element of tal.

Step 2.4.3. Let ub[k] and lb[k], kv1, ..., n be, respectively, the largest and least value of sli1,k1,...,silm,kl.

Step 2.4.4. Let bpl be the {bound-pair-list} containing a {bound-pair} elements such that the j'th element in a

{bound-pair};
{lower-bound};
e1;
{upper-bound};
e2;

where e1 and e2 are {extent-expression} representing ub[k] and lb[k] respectively.

Step 2.4.5. Let bpc be the {bound-pair-comma-list} produced by inserting commas as appropriate in bpl.

Step 2.4.6. Append an

{attribute};
{data-attribute};
{dimension-attribute};
DIMENSION
{dimension-suffix};

bpc
"";

so, the {attribute-list} of d.

Step 2.5. If alc contains ENTRY then perform Steps 2.5.1 through 2.5.7.

Step 2.5.1. If any {entry-information} of tal contains a {returns-descriptor}, rd then perform Steps 2.5.1.1 and 2.5.1.2.

Step 2.5.1.1. Each element of tal must contain a {returns-descriptor}.

Note: a check for consistency is made in copy-descriptors.
Step 2.5.1.2. Append to \( \text{ali} \) an
\[
\text{attribute}
\]
\[
\text{data-attribute}
\]\n\[
\text{rd}.
\]

Step 2.5.2. If any \{entry-information\} of \( \text{tin} \) contains an \{option\}, then perform
Steps 2.5.2.1 and 2.5.2.2.

Step 2.5.2.1. Each element of \( \text{tin} \) must contain an \{option\}.

Note: a check for consistency is made in \{copy-descriptors\}.

Step 2.5.2.2. Append to \( \text{ali} \) an
\[
\text{attribute}
\]
\[
\text{op}.
\]

Step 2.6. If \( \text{dci} \) has a subcomponent, append \{\} to \( \text{dci} \). Append \( \text{d} \) to \( \text{dci} \).

Step 2.7. Delete from \( \text{ail} \) all those elements of which there is a copy in \( \text{tin} \).

Step 2. Return \( \text{ali} \).

8.3.3 COMPLETE-STRUCTURE-DECLARATIONS

A structure is specified by a hierarchical set of names that refers to a group of individual items each of which may have a different data type. Conversely, an array is specified by a single name referring to a group of items all of the same data type. The component items of a structure may themselves be structures or arrays.

Operation: complete-structure-declarations

Step 1. For each \{declaration-comma-list\}, \{description-comma-list\}, or \{generic-description-comma-list\}, \( \text{di} \) component of the \{concrete-external-procedure\} perform determine-structure(d) to obtain a \{declaration-comma-list\}, \{description-comma-list\}, or \{generic-description-comma-list\}, \( \text{di} \). Replace \( \text{di} \) by \( \text{di} \).

Step 2. Let \( \text{dij} \) be the \( j \)th \{declaration\} component of the \{concrete-external-procedure\} that declaration-contains LIKE. For each \( \text{dij} \) perform Step 2.1.

Step 2.1. Perform expand-like-attribute(dij) to obtain a \{declaration-comma-list\}, \( \text{dcmij} \).

Step 3. For each \( \text{dij} \) perform Step 3.1.

Step 3.1. Let \( \text{d} \) be the \{declaration-comma-list\} that immediately contains \( \text{dij} \). Attach a \{\} followed by the elements of \( \text{dcmij} \) in sequence so that they immediately follow \( \text{dij} \).

Step 4. Delete all \{attribute\} LIKE \{unspecified-reference\} components of \{declaration\} in the \{concrete-external-procedure\}.

Step 5. For each \{declaration-comma-list\}, \{description-comma-list\}, or \{generic-description-comma-list\}, \( \text{dl} \) component of the \{concrete-external-procedure\} that declaration-contains STRUCTURE, perform convert-to-logical-level(dil) to obtain \( \text{dil} \), a node of the same type as \( \text{di} \). Perform propagate-alignment(dil) to obtain \( \text{dil} \), a node of the same type as \( \text{di} \). Replace \( \text{di} \) by \( \text{dil} \).
4.3.2.1 Determine-structure

Operation: determine-structure(cml)

where cml is a @declaration-connalist#, @description-connalist#, or @generic-description-connalist#.

result: a @declaration-connalist#, a @description-connalist#, or @generic-description-connalist#.

Step 1. Let cml be a copy of cml. Let eij be the j'th immediate component of cml that is not @}. Let n be the number of such components.

Step 2. For each eij that immediately contains a @level#, lv, perform steps 2.1 through 2.4.

Step 2.1. lv must not be @.

Step 2.2. If j is less than n and eij+1 immediately contains a @level# whose numeric value is greater than that of lv, then attach STRUCTURE to eij.

Step 2.3. If eij declaration-contains LIKE then attach STRUCTURE to eij. In this case, eij must not contain more than one instance of LIKE and cml must be a @declaration-connalist#.

Step 2.4. If the numeric value of lv is greater than one, attach MEMBER to eij.

Step 3. For each eij, all of the following must be false:

1. eij immediately contains @level# and does not declaration-contains STRUCTURE or MEMBER.
2. eij immediately contains @level# whose value is 1 and eij declaration-contains MEMBER.
3. eij declaration-contains STRUCTURE or MEMBER and does not immediately contain @level#.
4. eij declaration-contains STRUCTURE, does not declaration-contains LIKE and either j is equal to n or eij+1 does not declaration-contains a @level# whose numeric value is greater than the @level# declaration-component of eij.
5. eij declaration-contains MEMBER and either j is equal to one or eij-1 does not declaration-contains MEMBER or STRUCTURE.
6. eij declaration-contains STRUCTURE and LIKE, j is less than n, and eij+1 declaration-contains MEMBER and a @level# whose numeric value is greater than the numeric value of the @level# declaration-contained in eij.
7. eij declaration-contains LIKE but not a @level#.

Step 4. Return cml.

4.3.2.2 Expand-like-attribute

Operation: expand-like-attribute(d)

where d is a @declaration#.

result: a @declaration-connalist#.

Step 1. d declaration-contains an

@attribute#
LIKE
@unsubscripted-reference#.

76
Step 2. Perform find-applicable-declaration(id) to obtain Id. Id must be a {Declaration} that declaration-contains STRUCTURE and must not declaration-contain LREQ.

Step 3. Let Id be immediately contained in the {declaration-comma-list}. Let d be the j'th element of Id that is not $\xi_i$. Let k be such that sk1 is identical to Id and let n be the number of elements of Id that are not $\xi_i$.

Step 4. Let m be the numeric value of the immediately contained {level} of d. Let lv be the numeric value of the immediately contained {level} of sk1 and let cf be the numeric value (m-lv).

Step 5. Let cl be a {declaration-comma-list} with no elements. Let i be k+1. Let t be the smallest integer greater than k such that one of the following is true: t is equal to n, s[t+1] does not declaration-contain MEMBER, or s[t+1] immediately contains a {level} whose numeric value is less than or equal to lv. Perform Steps 5.1 through 5.3 while i is less than or equal to t.

Step 5.1. Let ec be a copy of sk1. Let the numeric value of the immediately contained {level} of ec be lv+cf. Replace this {level} by one whose numeric value is lv+cf.

Step 5.2. Append ec to cl, appending commas where necessary.

Step 5.3. Let i be i+1.

Step 6. cl must not contain any instance of LREQ.

Step 7. Return cl.

4.1.1.3 Convert-to-logical-levels

Operation: convert-to-logical-levels(osi)

where osi is a {description-comma-list}, {generic-description-comma-list}, or {generic-description-comma-list}.

result: a {generic-description-comma-list}, {description-comma-list}, or {generic-description-comma-list}.

Step 1. Let calc be a copy of osi. Let c[1] be the j'th component of calc that is not % and let m be the number of such components. Let d[1] be the {level} declaration-contained in c[1], if it exists, and let d[1] be the numeric value of d[1].

Step 2.

Case 2.1. There exists an integer t less than m such that c[t] and c[t+1] each declaration-contains a {level}, and c[t+1] is greater than c[t+1].

Let j be the least such t.

Case 2.2. (Otherwise).

Return calc.

Step 3. Let j be the least integer greater than j such that either c[j] is equal to %, or c[j+1] does not declaration-contain MEMBER, or c[j+1] is less than or equal to c[j+1]. For 1=j+1,...,j, perform Step 3.1.

Step 3.1. Replace l[i] by a {level} whose numeric value is l[i]-1.

Step 4. Go to Step 2.
4.3.1.4 Propagate-alignment

Operation: \texttt{propagate-alignment\texttt{\{cal\}}}

where \texttt{cal} is a \{declaration-comma-list\}, \{description-comma-list\}, or \{generic-description-comma-list\}.

result: a \{declaration-comma-list\}, \{description-comma-list\}, or \{generic-description-comma-list\}.

Step 1. Let \texttt{cal'} be a copy of \texttt{cal}. Let \texttt{eij} be the \textit{j}'th component of \texttt{cal'} that is not \texttt{\{\}} and let \texttt{a} be the number of such components. Let \texttt{iij} be the \{level\} declaration-contained in \texttt{eij}, if it exists, and let \texttt{ij} be the numeric value of \texttt{iij}.

Step 2. For \texttt{i=1,...,a}, perform Steps 2.1 and 2.2.

Step 2.1. \texttt{eij} must not declaration-contain both ALIGNED and UNALIGNED.

Step 2.2.

Case 2.2.1. \texttt{eij} declaration-contains REMAIN, and \texttt{eij} declaration-contains neither ALIGNED nor UNALIGNED.

Let \texttt{k} be the greatest integer such that \texttt{k} is less than \texttt{i} and \texttt{ij} is equal to \texttt{iij-1}. If \texttt{eij} declaration-contains \{ALIGNED\} or \{UNALIGNED\}, then attach \{ALIGNED\} or \{UNALIGNED\}, respectively, to \texttt{eij}.

Case 2.2.2. (Otherwise).

No action.

Step 3. For \texttt{i=1,...,a}, perform Step 3.1.

Step 3.1. If \texttt{eij} contains STRUCTURE and ALIGNED, or STRUCTURE and UNALIGNED, delete ALIGNED or UNALIGNED, respectively, from \texttt{eij}.

Step 4. Return \texttt{cal'}.

4.3.1.5 Find-applicable-declaration

Operation: \texttt{find-applicable-declaration\texttt{\{r\}}}

where \texttt{r} is an \{identifier\}, \{unsubscripted-reference\}, \{basic-reference\}, or \{reference\}.

result: a \{declaration\} or \{absent\}.

Step 1.

Case 1.1. \texttt{r} is an \{identifier\}.

Let \texttt{iid} be an \{identifier-list\}: \texttt{r}.

Case 1.2. \texttt{r} is an \{unsubscripted-reference\}. Append, in order, a copy of each \{identifier\} component of \texttt{r} to an \{identifier-list\}, \texttt{iid}.

Case 1.3. \texttt{r} is a \{basic-reference\}.

Append, in order, a copy of each \{identifier\} component of \texttt{r} which is not a component of an \{arguments\}, in order, to an \{identifier-list\}, \texttt{iid}.

Case 1.4. \texttt{r} is a \{reference\}.

Let \texttt{br} be the \{basic-reference\} immediate component of \texttt{r}. Perform \texttt{find-applicable-declaration\texttt{\{br\}}} to obtain \texttt{d}. Return \texttt{d}.
Step 2. Let \( b \) be the concrete-block that \( \text{block-contains} \). If \( b \) is contained in a \( \text{statement-name} \), go then perform Step 2.1.

Step 2.1. If \( b \) is contained in a \( \text{unit} \) which simply contains \( \text{entry-statement} \) or \( b \) is contained in a \( \text{prefix-list} \) immediate component of a \( \text{procedure} \) then let \( b \) be the concrete-block which \( \text{block-contains} \).

Step 3. Let \( id \) be the rightmost \( \text{identifier} \) component of \( id \).

Step 4. Let \( dl \) be a \( \text{declaration-designator-list} \) each component of which designates a \( \text{declaration} \) that is a concrete-block-component of \( b \) and that immediately contains an \( \text{identifier} \) that is equal to \( id \).

Step 5. Delete from \( dl \) any component that designates a \( \text{declaration} \), \( d \) such that \( \text{find-fully-qualified-name}(d) \) returns an \( \text{identifier-list} \), \( idl \) such that \( idl \) does not contain an ordered sublist of the \( \text{identifier} \)s contained in \( dl \).

Step 6.

Case 6.1. \( dl \) is empty.

Let \( b \) be the concrete-block that \( \text{block-contains} \). If there is no such block then return \( \text{abort} \); otherwise, go to Step 4.

Case 6.2. \( dl \) contains a single component.

Let \( d \) be the \( \text{declaration} \) designated by the single component of \( dl \). Return \( d \).

Case 6.3. \( dl \) contains more than one component.

\( dl \) must contain exactly one component that designates a \( \text{declaration} \), \( d \) such that \( \text{find-fully-qualified-name}(d) \) returns an \( \text{identifier-list} \) equal to \( idl \). Return \( d \).

4.3.4 \text{find-fully-qualified-name}

Each \( \text{declaration} \) has a fully qualified name associated with it. If it is an array or item declaration then this is a single \( \text{identifier} \). If the \( \text{declaration} \) is a member of a structure then it may have as many \( \text{identifier} \)s as its logical level-number indicates its depth of embedding to be.

Operation: \( \text{find-fully-qualified-name}(dl) \)

where \( dl \) is a \( \text{declaration} \).

result: an \( \text{identifier-list} \).

Case 1. \( d \) declaration-contains MEMBER.

Step 1.1. Let \( dl \) be the \( \text{declaration-context-list} \) that immediately contains \( d \). Let \( dl \) be the rightmost preceding \( \text{declaration} \) of \( dl \) that declaration-contains a \( \text{level} \) whose numeric value is less than the numeric value of the \( \text{level} \) of \( d \).

Step 1.2. Perform \( \text{find-fully-qualified-name}(dl) \) to obtain an \( \text{identifier-list} \), \( idl \).

Step 1.3. Append the \( \text{identifier} \) immediate component of \( d \) to \( idl \).

Step 1.4. Return \( idl \).

Case 2. \( d \) does not declaration-contain MEMBER.

Step 2.1. Let \( id \) be the \( \text{identifier} \) immediate component of \( d \).

Step 2.2. Return \( \text{identifier-list} \), \( id \).
4.1.4 CONSTRUCT-CONTEXTUAL-DECLARATIONS

Certain contexts in a procedure specify the attributes of an identifier appearing in those contexts. If an identifier is not explicitly declared, but is used in such a context, then it is contextually declared and a declaration for it is introduced into the concrete-external-procedure. All the attributes implied by the context are added onto this generated declaration.

Operation: construct-contextual-declarations

Step 1. Let cep be the concrete-external-procedure. Let u be a

{unit}:
  {declare-statement}:
    DECLARE
    {declaration-commalist}.dcl
  ;

where dcl contains no elements.

Step 2. For each {identifier}.id in cep perform steps 2.1 through 2.3:

Step 2.1. Let attrsa be <append>.

Step 2.2.

Case 2.2.1. id is a component of a

{reference}.r:
  {basic-reference}:
    {identifier}.id;

where r has no other components.

Step 2.2.1.1.

Case 2.2.1.1.1. r is an immediate component of an {in-option} or r is an immediate component of a {data-attribute}.da and da contains OFFSET.

Let attrs be an

{attribute-list}:
  {attribute}:
    {data-attribute}:
      AREA;
    {attribute}:
      VARIABLE;

Case 2.2.1.1.2. r is an immediate component of a {call-statement}.

This case must not occur.

Case 2.2.1.1.3. r is an component of a {file-option}, {copy-option}, or {named-io-condition}.

Let attrs be as

{attribute-list}:
  {attribute}:
    {data-attribute}:
      FILE;
  {attribute}:
    CONSTANT;
Case 2.2.1.1.4. \( x \) is an immediate component of a \{text-option\} or a \{locator-qualifier\}, or \( x \) is an immediate component of an \{attribute\} that contains BASED.

Let \( \text{attr} \) be an

\[ \{\text{attribute-list}\} : \{\text{attribute}\} : \{\text{data-attribute}\} : \{\text{POINTER}\} : \{\text{attribute}\} : \{\text{VARNAME}\} \]

Case 2.2.1.1.5. (Otherwise).

No action.

Case 2.2.2. \text{id} is an immediate component of a \{programmable-condition\}.

Let \( \text{attr} \) be an

\[ \{\text{attribute-list}\} : \{\text{attribute}\} : \{\text{CONDITION}\} \]

Case 2.2.3. \text{id} is the only component of the \{basic-reference\} of a \{reference\}:

\[ \{\text{basic-reference}\} : \{\text{identifier}\} : \{\text{identifier}\} : \{\text{arguments-list}\} \]

where the \{reference\} has no other immediate components.

Let \( \text{attr} \) be an

\[ \{\text{attribute-list}\} : \{\text{attribute}\} : \{\text{NULLPTR}\} \]

Case 2.2.4. (Otherwise).

No action.

Step 2.3. If \( \text{attr} \) is not \{absent\} then perform \{find-applicable-declaration\}(\text{id}) to obtain \( d \). If \( d \) is \{absent\} then perform Steps 2.3.1 through 2.3.3.

Step 2.3.1. Let \( d \) be a

\[ \{\text{declaration}\} : \{\text{id}\} : \text{attr} \]

Step 2.3.2. If an element of \( d \) contains an \{identifier\} equal to \( \text{id} \) then the \{attribute-list\} of this \{declaration\} must equal \( \text{attr} \).

Step 2.3.3. If no element of \( d \) contains an \{identifier\} equal to \( \text{id} \) then append \( d \) to \( d \).

Step 3. If \( d \) contains any elements, append \( a \) to the \{unit-list\} of the \{procedure\} immediately contained in \( c \).
8.3.5 \texttt{construct-implicit-declarations}

Identifiers that do not resolve to any declaration or that have not had declarations constructed because of the context in which they appear are implicitly declared.

\textbf{Operation:} \texttt{construct-implicit-declarations}

\textbf{Step 1.} For each tree of the form

\begin{verbatim}
{reference};
{basic-reference};
{identifier};
\end{verbatim}

where the \{reference\} has no other components, or that is the only \{identifier\} of an \{unsubscribed-reference\}, or that is an immediate component of an \{allocation\}, \{freeing\}, or \{locate-statement\}, perform Step 1.1.

\textbf{Step 1.1.} Perform \texttt{find-applicable-declaration(id)} to obtain \(d\). If \(d\) is \texttt{absent} then perform Step 1.1.1.

\textbf{Step 1.1.1.} Let \(a\) be a

\begin{verbatim}
{unit};
{declared-statement};
DECLARE
{declaration-list};
{declaration};
\end{verbatim}

Append \(a\) to the \{unit-list\} of the \{procedure\} immediately contained in the \{concrete-external-procedure\}.

8.3.6 \texttt{complete-declarations}

\textbf{Operation:} \texttt{complete-declarations}

\textbf{Step 1.} For each \{default-attributes\}, \(d\) component of the \{concrete-external-procedure\}, perform Steps 1.1 and 1.2.

\textbf{Step 1.1.} Let \(d\) be a \{declaration\}; \(x\), where \(x\) is a copy of the \{attribute-list\} of \(d\); \(\bar{d}\) is a partial \{declaration\}.

\textbf{Step 1.2.} Perform \texttt{test-attribute-consistency}(\(d\), \texttt{absent}) to obtain \(tv\). \(tv\) must not be \texttt{false}.

\textbf{Step 2.} For each \{declaration\}, \{description\}, or \{generic-description\}, \(d\) component of the \{concrete-external-procedure\}, perform Step 2.1.

\textbf{Step 2.1.} Perform \texttt{test-attribute-consistency}(\(d\)) to obtain \(tv\). \(tv\) must not be \texttt{false}.

\textbf{Step 3.} Perform \texttt{append-system-defaults}.

\textbf{Step 4.} Each \{default-attributes\} component of the \{concrete-external-procedure\} must not contain LIKE, MEMBER, STRUCTURE, or PARAMETER.

\textbf{Step 5.} For each \{Declaration\}, \(d\) component of the \{concrete-external-procedure\} perform Steps 5.1 and 5.2.

\textbf{Step 5.1.} Perform \texttt{apply-defaults}(\(d\)).

\textbf{Step 5.2.} Perform \texttt{check-attribute-completeness-and-delete-attributes}(\(d\)).

\textbf{Step 6.} For each \{description\}, \(d\) component of the \{concrete-external-procedure\} that satisfies the following conditions, perform Steps 6.1 and 6.2.

\begin{enumerate}
\item \(d\) is contained in a \{Declaration\} but not in a \{generic-attribute\}.
\item \(d\) was not produced by Step 4.1.
\end{enumerate}
Step 6.1. Perform apply-defaults().

Step 6.2. Perform check-attribute-completeness-and-delete-attributes().

Step 7. For each {returns-descriptor}, d component of the {concrete-external-procedure} which does not contain a {description} and which satisfies the following conditions, perform Step 7.1.

{1} d is contained in a {declaration} but not in a {generic-attribute}.

{2} d was not produced by Step 6.1 or Step 7.1.

Step 7.1. Perform apply-defaults(d).

Step 8. For each {description}, d component of the {concrete-external-procedure} which is contained in a {declaration} but not in a {generic-attribute}, perform check-attribute-completeness-and-delete-attributes().

Step 9. For each tree of the form

{procedure};

{prefix-list};

{prefix};

{statement-name-list};

{procedure-statement};

{unit-list};

{ending};

or of the form

{unit};

{statement-name-list};

{statement-name};

{entry-statement};

perform Steps 9.1 through 9.3.

Step 9.1. Let id be the {identifier} immediately contained in as.

Step 9.2. Perform find-applicable-declaration(id) to obtain d. d must be a {declaration}.

Step 9.3. Perform copy-descriptors(id).

4.3.4.1 Text-attribute-consistency

Operation: text-attribute-consistency(d, ds)

where d is a {declaration}, {description}, or {generic-description},

ds is a {default-attributes}.

result: {true} or {false}.

Step 1. If ds is {absent} then:

Case 1.1. d is a {declaration}.

For each pair of {attribute-descriptors}, a1 and a2, which are declaration-components of d, perform test-invalid-duplicates(a1, a2) to obtain tv. tv must not be {true}.

Case 1.2. d is a {description}.

For each pair of {data-attribute-descriptors}, a1 and a2, which are declaration-components of d, perform test-invalid-duplicates(a1, a2) to obtain tv. tv must not be {true}.
Case 1.3. \( a \) is a \{generic-description\}.

For each pair of \{generic-data-attribute\}, \( a_i \) and \( a_j \), which are declaration-components of \( a \), perform \text{test-invalid-duplicate\(\_\text{data}\text{-}\text{attribute}\text{-}\text{a}\text{_}\text{a}\text{)} to obtain \( t \). \( t \) must not be \text{true}.\}

Step 2.

Case 2.1. \( a \) is \{abstract\}.

Let \( a_k \) be an \{attribute-keyword-list\} consisting of a copy of each keyword declaration-component of \( a \).

Case 2.2. \( a \) is not \{abstract\}.

Step 2.2.1. Let \( a_l \) be an \{attribute-list\} consisting of copies of each \{attribute\} declaration-component of \( a_k \), and each \{attribute\} or \{data-attribute\} declaration-component of \( d \) where \( d \) is a \{declaration\} or \{description\} respectively.

Step 2.2.2. For each pair of \{attribute\} components of \( a_l \) and \( a_j \), perform \text{test-invalid-duplicate\(\_\text{data}\text{-}\text{attribute}\text{-}\text{a}\text{_}\text{a}\text{)} to obtain \( t \). If \( t \) is \text{true}, return \text{false}.\}

Step 2.2.3. Let \( a_k \) be an \{attribute-keyword-list\} consisting of a copy of each keyword declaration-component of \( a_l \).

Step 3. Replace all multiple occurrences of a given keyword in \( a_k \) by a single occurrence of that keyword.

Step 4. If \( a_k \) contains GENERIC or MEMBER and also contains EXTERNAL, return \text{false}.\}

Step 5. If \( a_k \) contains STRUCTURE and ALIGNED, or contains STRUCTURE and UNALIGNED, return \text{false}.\}

Step 6.

Case 6.1. \( a \) is a \{description\} or \{generic-description\}.

If the set of all keywords in \( a_k \) is not a subset of the keywords that form the concrete-representation of some tree whose root-node is \{consistent-description\}, return \text{false}.\}

Case 6.2. \( a \) is a partial \{declaration\} produced by the operation \text{create-constant}.\}

If the set of all keywords in \( a_k \) is not a subset of the keywords that form the concrete-representation of some tree whose root-node is \{consistent-literal-constant\}, return \text{false}.\}

Case 6.3. \( a \) is a \{declaration\}.

If the set of all keywords in \( a_k \) is not a subset of the keywords that form the concrete-representation of some tree whose root-node is \{consistent-declaration\}, return \text{false}.\}

Step 7. Return \text{true}.\}

\{consistent-declaration\} := \{scope\} \{declaration-type\}

\{scope\} := \text{EXTERNAL} \mid \text{INTERNAL}.

\{declaration-type\} := \{variable\} \mid \{named-constant\} \mid \text{BUILTIN} \mid \text{CONDITION} \mid \text{GENERIC}

\{variable\} := \text{VARIABLE} \{storage-type\} \{data-description\}

\{storage-type\} := \{storage-class\} \mid \text{DEFINED} \{POSITION\} \mid \text{PARAMETER} \mid \text{MEMBER}

\{storage-class\} := \text{AUTOMATIC} \mid \text{BASED} \mid \text{CONTROLLED} \mid \text{STATIC}

\{data-description\} := \{DIMENSION\} \{alignment\} \{\{data-type\} \{INITIAL\} \mid \text{STRUCTURE}\}
4.3.4.2 Test-invalid-duplicates

Operation: test-invalid-duplicates(a1, a2)

where a1 and a2 are attribute| or generic-data-attribute| or any of those.

result: <true> or <false>.

Step 1. Let k1 and k2 be the leftmost keyword components of a1 and a2 respectively.

Step 2.

Case 2.1. k1 and k2 are equal.

Case 2.1.1. a1 and a2 have no terminal components other than k1 and k2.

Return <false>.

Case 2.1.2. k1 (respectively, k2) is the only terminal component of a1 (respectively a2) but k2 (respectively, k1) is not the only terminal component of a2 (respectively, a1).

Return <false>.

Case 2.1.3. Otherwise.

Return <true>.

Case 2.2. k1 and k2 are not equal.

Return <false>.
4.3.5.1 Append-system-defaults

Operation: append-system-defaults

Step 1. If the procedure p immediately contained in the concrete-external-procedure does not have as a concrete-block-component a default-block that contains SYSTEM, append a

```
{unit};
{default-statement};
DEFAULT SYSTEM $1$;
```

to the {unit-list} of p.

Step 2. Modify the {symbol-list} below by replacing d1, d2, d3, d4, and d5 by implementation-defined integers to obtain td.

```
/* ENTRY DEFAULTS */
DEFAULT (returns) ENTRY;

/* FILE DEFAULTS */
DEFAULT (DIRECT|INPUT|INPUT|RECORD|RECORD|SEQ|SEQ|UPDATE) FILE;

/* ARITHMETIC DEFAULTS */
DEFAULT (+CONSTANT & ~PICTURE) FIXED, BINARY, REAL;
DEFAULT (FIXED & BINARY & ~CONSTANT) PRECISION(00000000);
DEFAULT (FIXED & DECIMAL & ~CONSTANT) PRECISION(00000000);
DEFAULT (FLOAT & BINARY & ~CONSTANT) PRECISION(00000000);

/* STRING AND AREA DEFAULTS */
DEFAULT CHARACTER CHARACTER(1), NONVARYING;
DEFAULT (BIT) BIT(1), NONVARYING;
DEFAULT (AREA) AREA(d5);
DEFAULT (POSITION) POSITION(d5);

/* SCOPE AND STORAGE CLASS DEFAULTS */
DEFAULT ({ENTRY|FILE} &
  (AUTOMATIC|BASED|DEFINED|PARAMETER|STATIC|CONTROLLED|REMOVED|ALIGNED|UNALIGNED|EXTERNAL)) VARIABLE;
DEFAULT ({ENTRY|FILE} & RANGE(*) ) CONSTANT;
DEFAULT (CONDITION) (FILE|ENTRY) & CONSTANT) EXTERNAL;
DEFAULT (NAME(*) ) INTERNAL;
DEFAULT (VARIABLE & EXTERNAL) STATIC;
DEFAULT (VARIABLE) AUTOMATIC;

/* ALIGNMENT DEFAULTS */
DEFAULT (- (CHARACTER|BIT|PICTURE) & ~CONSTANT) UNALIGNED;
DEFAULT (+ (CONSTANT) ALIGNED;
```
Step 3. For each \texttt{unit\_p} component of the \texttt{concrete-external-procedure} where \texttt{u} immediately contains a \texttt{default-statement} that contains \texttt{SELECT}, perform Steps 3.1 and 3.2.

Step 3.1. Perform \texttt{person\_id,unit\_list})} to obtain \texttt{u}.

Step 3.2. Replace the single \texttt{unit\_p} by the \texttt{unit\_p} immediate components of \texttt{u} such that all of the immediate components of \texttt{u} are effectively inserted, in order, in place of \texttt{u}.

3.3.4.4 Apply-defaults

Operation: \texttt{apply-defaults(})

where \texttt{d} is a \texttt{declaration}, or a \texttt{description}, or a \texttt{returns-descriptor} with no \texttt{description} component.

Step 1. If \texttt{d} is a \texttt{returns-descriptor}, then attach to \texttt{d} a \texttt{description}, decl with no components, and the surrounding parentheses as required; otherwise let \texttt{decl} be \texttt{d}.

Step 2. For each \texttt{default-statement}, \texttt{dft} component of the \texttt{concrete-external-procedure} taken in left-to-right order perform Steps 2.1 through 2.3.

Step 2.1. \texttt{dft} must be a block component of the \texttt{procedure} immediate component of the \texttt{concrete-external-procedure}.

Step 2.2. If \texttt{dft} immediately contains \texttt{NONE} then go to Step 3.

Step 2.3. Let \texttt{dpe} be the \texttt{predicate-expression} component of \texttt{dft}. Perform \texttt{test-default-applicability(dpe, decl)} to obtain \texttt{tv}. If \texttt{tv} is \texttt{true} perform Steps 2.3.1 and 2.3.2.

Step 2.3.1. \texttt{dft} must not contain \texttt{ERROR}.

Step 2.3.2. For each \texttt{default-attribute}, \texttt{das} component of \texttt{dft} in left-to-right order perform Steps 2.3.2.1 through 2.3.2.3.

Step 2.3.2.1. Perform \texttt{test-attribute-consistency(decl, das)} to obtain \texttt{tv}.

Step 2.3.2.2. If \texttt{tv} is \texttt{true} and \texttt{decl} is a \texttt{declaration}, then append to the \texttt{attribute-list} in \texttt{decl} copies of the \texttt{attribute} simple components of \texttt{das}.

Step 2.3.2.3. If \texttt{tv} is \texttt{true} and \texttt{decl} is a \texttt{description}, then append to the \texttt{data-attribute-list} in \texttt{decl} copies of the \texttt{data-attribute} simple components of \texttt{das}.

Step 3. If \texttt{d} is a \texttt{returns-descriptor}, then \texttt{decl} must contain at least one subnode.

3.3.5 Test-default-applicability

Operation: \texttt{test-default-applicability(dpe, decl)}

where \texttt{dpe} is a \texttt{predicate-expression}, \texttt{predicate-expression-three}, \texttt{predicate-expression-two}, \texttt{predicate-expression-one}, \texttt{range-specification}, or \texttt{attribute-keyword}, \texttt{decl} is a \texttt{declaration}, or a \texttt{description}.

result: \texttt{true} or \texttt{false}.

Case 1. \texttt{dpe} immediately contains a \texttt{if}.

Let \texttt{pe} be the \texttt{predicate-expression} simple component of \texttt{dpe}, and \texttt{p3} be the \texttt{predicate-expression-three} simple component of \texttt{dpe}. If either, or both, \texttt{test-default-applicability(pe, decl)} or \texttt{test-default-applicability(p3, decl)} yield \texttt{true} then return \texttt{true}; otherwise return \texttt{false}.

Chapter 4: The Translator 57
Case 2. dpe immediately contains an #.

let p1 be the predicate-expression-three simple component of dpe, and p2 be the predicate-expression-two simple component of dpe. If both test-default-applicability(p2, decl) and test-default-applicability(p1, decl) yield true, then return true; otherwise return false.

Case 3. dpe immediately contains an #.

let p be the other simple component of dpe. Perform test-default-applicability(p, decl) to obtain tv. If tv is true then return false; otherwise return true.

Case 4. dpe is a range-specification, re.

Case 4.1. re contains an #.

If decl is a declaration which immediately contains an identifier return true, otherwise return false.

Case 4.2. re contains an identifier, id.

Step 4.2.1. If decl is a description or a declaration that does not immediately contain an identifier, then return false.

Step 4.2.2. Let id be the identifier of decl. Immediate component of decl. Compare the terminal nodes of id and id, taken in order, until the terminal nodes of id have been exhausted. If all the comparisons are equal then return true; otherwise return false.

Case 4.3. re contains a letter, l1 # l letter, l2.

Step 4.3.1. If decl is a description or a declaration that does not immediately contain an identifier, then return false.

Step 4.3.2. Let dl be the first letter of the identifier of decl. If dl is, or is after l1 and l1, or is before l2 in the English alphabet then return true; otherwise return false.

Case 5. dpe is an attribute-keyword, ak.

If decl declaration-contains an attribute, str or a data-attribute, str such that the leftmost keyword, ak of str is equal to ak then return true; otherwise return false.

Case 6. (Otherwise).

Let p be the predicate-expression component of dpe. Perform test-default-applicability(p, decl) to obtain tr. Return tr.

4.3.4 Copy-descriptors

Operation: copy-descriptors(d)

where d is a declaration.

Step 1. Let a be the data-attribute declaration-component of d that immediately contains empty.

Step 2.

Case 2.1. d declaration-contains a dimension-attribute.

Let ep # l1, l_1,..., l_n be the n occurrences of

statement-name:

#identifier, id

{signed-integer-constant} #

such that d is obtained by performing find-applicable-declaration(id) -
Case 2.2. {Otherwise}.

Let sp[i] be the single occurrence of a

\{statement-name\}

\{identifier\}, i.e. \$i\$

such that \$d\$ is obtained by performing \texttt{find-applicable-declaration(id)}. Let \$n\$ be \$1\$.

Step 3. For each sp[i], i=1,...,n, perform Steps 3.1 through 3.4.

Step 3.1.

Case 3.1.1. sp[i] is contained in a \{prefix-list\} immediate component of a \{procedure\}, p.

Let us be the immediately contained \{procedure-statement\} of p.

Case 3.1.2. sp[i] is contained in a \{statement-name-list\} immediate component of a \{unit\}, u.

Let us be the immediately contained \{entry-statement\} of u.

Step 3.2.

Case 3.2.1. us contains a \{parameter-name-comma-list\}, p1.

Step 3.2.1.1. Let \$n1\$j be the \{identifier\} of the \$j\$th immediately contained \{parameter-name\} component of p1. Let \$n\$ be the number of such components.

For each \$n1\$j, j=1,...,n, taken in order, perform Steps 3.2.1.1.1 through 3.2.1.1.4.

Step 3.2.1.1.1. Perform \texttt{find-applicable-declaration(p1,y)} to obtain pd. pd must be a \{declaration\}.

Step 3.2.1.1.2. Let dcmi be the \{declaration-comma-list\} that immediately contains pd. Let dcmi be the \$k\$th \{declaration\} component of dcmi. Let there be \$m\$ such components and let \$k\$ be such that dcmi is identical to \$pd\$.

Step 3.2.1.1.3. Let \$k\$ and be the minimum integer greater than or equal to \$k\$ such that one of the following is true:

1. \$k\$ and is equal to \$m\$;
2. dcmi does not contain \{level\};
3. dcmi contains a \{level\} whose \{integer\} has the value 1.

Step 3.2.1.1.4. For each dcmi, i=x-k,...,x-m, taken in order, perform Steps 3.2.1.1.4.1 through 3.2.1.1.4.5.

Step 3.2.1.1.4.1. Let \$pd\$ be a copy of dcmi. Delete from \$pd\$ any occurrence of an

\{attribute\}:

\{PARAMETER\}.
Step 3.2.1.1.4.2. If do{lin} contains an

```
attribute;
  data-attribute, da:
  offset
  preference,r
  ;
```

then perform test-offset-in-description(da) to obtain tv. If tv is `trim', then replace the
data-attribute in pad corresponding to da by `data-attribute'; offset.

Note: This language feature allows pointer-to-offset conversion to be performed on calls to internal
procedures only if the area to which the offset is relative is known to the calling block.

Step 3.2.1.1.4.3. If pad contains a

```
data-attribute, da:
  entry
  { description-comalist }
  /
```

then replace da with a

```
data-attribute;
  entry.
```

Step 3.2.1.1.4.4. If pad contains an

```
attribute, atr;
  data-attribute;
  returns-descriptor
```

then delete atr.

Step 3.2.1.1.4.5. Append, with intermediate \_/s as required, to a
\{description-comalist\}, des{ll}, a \{description\} simply containing copies of the \{level\} and \{data-
attribute\} simple components of pad. pad must not contain any \{extent-expression\} containing an
\{identifier\}.

Case 3.2.2. as does not contain a \{parameter-name-comalist\}.

Let des{ll} be `empty'.

Step 3.3. If as contains a \{returns-descriptor\}, let r{ll} be that \{returns-
descriptor\}; otherwise, let r{ll} be `empty'.

Step 3.4. If as contains an \{options\} then let op{ll} be a copy of that \{options\};
otherwise let op{ll} be `empty'.

Step 4.

Step 4.1. Let l{ll} be des{ll} for i=1,...,n. Perform Steps 4.5 and 4.6.
Step 4.2. Let l{ll} be r{ll} for i=1,...,n. Perform Steps 4.5 and 4.6.
Step 4.3. Let l{ll} be op{ll} for i=1,...,n. Perform Steps 4.5 and 4.6.
Step 4.4. Go to Step 5.
Step 4.5. It must be possible to make copies of all the EII in which the ordering of the immediate components of \$data-attribute-lists and \$options specifications has been altered such that all the EII are equal, except for the subnodes of any \$extent-expressions.

Step 4.6. For each pair of corresponding \$extent-expressions, ex1 and ex2, in each pair of members of the set EII, perform test-descriptor-extent-expression(text1, ext1, ext2) to obtain tv, which must be \$true.

Step 5. If descII is not \$absent, replace e with a

\$data-attribute;
  \$ENTRY
  \$DescII;

Step 6. If d declaration contains a \$returns-descriptor, drd, then for i=1,...,n, replace rdIl (which must not be \$absent) by drd.

Case 4.1. d is not block-contained in the \$concrete-external-procedure.

For each \$data-attribute, da; OFFSET(\$reference); in rdIl, perform Step 6.1.1.

Step 6.1.1. Perform test-offset-ja-declaration(da) to obtain tv, which must be \$notrip.

Case 4.2. (Otherwise).

For each \$data-attribute; OFFSET(\$reference); in drd, delete the \$reference.

Step 7. If d declaration contains an \$options, do a, then for i=1,...,n, replace opII (which must not be \$absent) by do a.

4.3.6.7 Test-offset-ja-declaration

This operation tests whether the \$reference from an OFFSET attribute has to be trimmed because it would fall within the scope of a different declaration if copied into an ENTRY declaration in the surrounding block.

Operation: test-offset-ja-declaration(da);

where da is a \$data-attribute; OFFSET(\$reference);

result: \$trip or \$notrip.

Step 1. For each \$reference, if contained in da, perform Step 1.1.

Step 1.1. Perform find-applicable-declaration(r) to obtain rd. rd must be a \$declaration. If rd is block-contained in the same block as da, return \$trip.

Step 2. Return \$notrip.
4.3.8 Test-descriptor-extent-expressions

Operation:  test-descriptor-extent-expressions(text1, ext2)

where ext1 and ext2 are extent-expressions.

results: <true> or <false>.

Step 1. If ext1 or ext2 contains identifier, return <false>.

Step 2. Let c1 and c2 be the expression immediate components of ext1 and ext2.

Perform create-expression(c1) to obtain an expression, e1, and create-expression(c2) to obtain an expression, e2.

Step 3. Perform evaluate-restricted-expression(e1) and evaluate-restricted-expression(e2) to obtain v1 and v2 respectively. If v1 or v2 is not a constant having computational-type, return <false>. Otherwise, perform evaluate-expression-to-integer(e1) and evaluate-expression-to-integer(e2) to obtain integer-value v1 and v2 respectively. If v1 and v2 are equal, return <true>; otherwise return <false>.

4.3.9 Validate-concrete-declarations

Various checks are applied to the concrete-external-procedure to ensure that a valid set of declarations has been generated. Each declaration must be unique within its own scope. A check is also made to ensure that declaration of INTERNAL CONSTANT ENTRY, CONSTANT FORMAT, and CONSTANT LABEL were constructed by the operation construct-statement-name-declarations.

Operation:  validate-concrete-declarations

Step 1. The concrete-external-procedure must not contain two declarations d1 and d2, such that d1 and d2 are block-components of the same block, and find-fully-qualified-name(d1) yields the same result as find-fully-qualified-name(d2).

(All d1 and d2 are duplicate declarations).

Step 2. The concrete-external-procedure must not contain a reference r, (basic-reference)r, or (unsubscripted-reference)r such that find-applicable-declarations(r) yields <empty>.

(There is no declaration for r).

Step 3. The concrete-external-procedure must not contain a declaration d that declaration-contains LABEL and CONSTANT, or FORMAT and CONSTANT, or INTERNAL and ENTRY and CONSTANT, unless was created by the operation construct-statement-name-declarations.
§4.3.1. Check-attribute-completeness-and-delete-attributes

Operation: \texttt{check-attribute-completeness-and-delete-attributes}(d)

where d is a \texttt{declaration} or a \texttt{description}.

Step 1. For each distinct keyword, k which is a declaration-component of d perform Step 1.1.

Step 1.1. If there is an \texttt{attribute} or a \texttt{data-attribute} declaration-component of d such that \texttt{attribute} contains k, but not as its sole terminal component, then delete all \texttt{attribute} or \texttt{data-attribute} declaration-components of d which declaration-contains k, except for att; otherwise, delete all but one of the \texttt{attribute} or \texttt{data-attribute} declaration-components of d which declaration-contains k.

Step 2. d must not declaration-contain an \texttt{attribute} or \texttt{data-attribute} with \texttt{AREA}, \texttt{EDIT}, \texttt{CHARACTER}, \texttt{DIMENSION}, \texttt{GENERAL}, \texttt{INITIAL}, \texttt{PICTURE}, \texttt{POSITION}, \texttt{PRECISION}, or \texttt{RETURN} as its sole terminal node.

Step 3. If d declaration-contains \texttt{EXTERNAL}, it must not declaration-contain \texttt{AUTOMATIC}, \texttt{BASED}, \texttt{DEFINED}, \texttt{PARAMETER}, or \texttt{MULTIP}.

Step 4. If d declaration-contains \texttt{EXTERNAL} and \texttt{CONSTANT}, it must not declaration-contains \texttt{FORMAT} or \texttt{LABEL}.

Step 5. If d declaration-contains \texttt{EXTERNAL} and \texttt{CONSTANT} and \texttt{ENTRY}, it must not declaration-contains \texttt{DIMENSION}.

Step 6. If d is a \texttt{description}, d must not declaration-contains \texttt{INITIAL}.

Step 7. If d is a \texttt{declaration} which declaration-contains \texttt{DEFINED} or \texttt{PARAMETER} for declaration-contains \texttt{MEMBER} and the rightmost preceding \texttt{declaration} whose \texttt{level} has the value one declaration-contains \texttt{DEFINED} or \texttt{PARAMETER}, then d must not declaration-contains \texttt{INITIAL}.

Step 8. If d is a \texttt{declaration} which declaration-contains \texttt{STATIC} and either \texttt{ENTRY}, \texttt{FORMAT}, or \texttt{LABEL} for declaration-contains \texttt{MEMBER} and either \texttt{ENTRY}, \texttt{FORMAT}, or \texttt{LABEL}, and the rightmost preceding \texttt{declaration} whose \texttt{level} has the value one declaration-contains \texttt{STATIC}, then d must not declaration-contains \texttt{INITIAL}.

Step 9. Let \texttt{akl} be an \texttt{attribute-keyword-list} consisting of copies of the keyword declaration-components of d.

Step 10. Delete from \texttt{akl} any keyword that can be contained in a tree whose root-node is \texttt{file-description-set}.

Step 11.

Case 11.1. d is a \texttt{description}.

There must exist a tree whose root-node is \texttt{consistent-description} and whose concrete-representation consists of the same set of keywords as are contained in \texttt{akl}.

Case 11.2. d is a \texttt{declaration}.

There must exist a tree whose root-node is \texttt{consistent-declaration} and whose concrete representation consists of the same set of keywords as are contained in \texttt{akl}.

Chapter 8: The Translator
4.4 Create-abstract-equivalent-tree

Where the Concrete and Abstract Syntaxes are similar, a simple transformation generates a specific abstract tree from the given concrete one. Essentially this consists of ignoring those concrete terms which represent the "punctuation" of the concrete program, and transforming each concrete node into its abstract equivalent. Where the syntaxes are not similar an operation specifies the translation.

Operation: \text{create-abstract-equivalent-tree}(ct)

where \text{ct} is a tree belonging to a category of the Concrete Syntax.

result: a tree belonging to a related category of the Abstract Syntax, or \text{<none>}

Case 1. \text{ct} is an \text{<allocation>}, \text{<assignment-statement>}, \text{<begin-block>}, \text{<constant>}, \text{<expression>}, \text{<format-iteration>}, \text{<freeing>}, \text{<identifier>}, \text{<initial-element>}, \text{<if-statement>}, \text{<group>}, \text{<locate-statement>}, \text{<on-statement>}, or \text{<picture>}

Perform \text{create-f(ct)} to obtain \text{abt}, where \text{"f"} is the name of the category of \text{ct}. Return \text{abt}.

Case 2. \text{ct} belongs to a terminal category.

Case 2.1. \text{ct} is a keyword and there is a terminal category whose name is of the form \text{"<kw>"} where \text{cn} is the lowercase equivalent of category name of \text{ct} (e.g., if \text{ct} is \text{FIXED}, \text{cn} is "fixed")

Return \text{<kw>}

Case 2.2. \text{ct} is an \text{*}

Return \text{<asterisk>}

Case 2.3. Otherwise

Return \text{<none>}

Case 3. The category-name of \text{ct} is of the form \text{"<cn-list>"} or \text{"<cn-commlist>"}, where \text{"cn"} is some name.

Let \text{x} be a \text{<cn-list>}. For each \text{<cn>,y, in ct, taken in order, perform \text{create-abstract-equivalent-tree(y)} to obtain \text{y}, and append \text{y} to \text{x}. Return \text{x}.

Case 4. \text{ct} is a \text{<radix-factor>}

Let \text{d} be the digit in the denotation of the immediate component of \text{ct}. Return \text{<radix-factor> \text{d}}.

Case 5. \text{ct} is a \text{<condition-name>}

Perform \text{create-condition(ct)} to obtain \text{abt}. Return \text{abt}.

Case 6. Otherwise

Step 6.1. Let \text{"cn"} be the name such that \text{"<cn>"} is the category-name of \text{ct}. Let \text{x} be a \text{<cn>}. If \text{<cn>} is a terminal category, return \text{x}.

Step 6.2. For each immediate subnode, \text{y, of ct, taken in order, perform Step 6.2.1.}

Step 6.2.1. 

Case 6.2.1.1. \text{y} is a \text{<reference>}

Let \text{ref} be a \text{<variable-reference>}, or a \text{<target-reference>}, or a \text{<subroutine-reference>}, choosing the alternative that permits \text{ref} to be attached to \text{x} as an immediate subnode. Perform \text{create-reference(x,ref)} to obtain \text{x}. Attach \text{x} to \text{y}. 
Case 6.2.1.2. \( y \) is an {unsubscripted-reference}.

Perform create-reference\((y, variable-reference)\) to obtain \( x \). Attach \( x \) to \( y \).

Case 6.2.1.3. Otherwise.

Perform create-abstract-equivalent-tree\((y)\) to obtain \( x \). If \( x \) is not equal to \( \emptyset \), attach \( x \) to \( y \).

Step 6.2. Return \( y \).

4.4.1 CREATION OF BLOCKS AND GROUPS

4.4.1.1 Create-procedure

Operation: create-procedure\((cp)\)

where \( cp \) is a {procedure}.

result: a {procedure}.

Step 1. \( cp \) immediately contains a {procedure-statement}, \( cpe \). Perform create-entry-point\((cpe)\) to obtain an {entry-point}, \( ep \). Let \( ap \) be a

\(<\text{procedure}>\)
\<entry-or-executable-unit-list>, equl;
\<entry-or-executable-unit>\;, ep.

Step 2. Perform create-block\((cp, ep)\).

Step 3. Let \( pl \) be the {prefix-list} immediately contained in \( cp \). Perform create-condition-prefix-list\((pl)\) to obtain a {condition-prefix-list}, \( cpl \) or {absent}. \( cpl \). If \( cpl \) is not \( {\text{absent}} \), attach \( cpl \) to \( ep \).

Step 4. If \( cp \) simply contains {recursive}, then attach \( {\text{recursive}} \) to \( ep \).

Step 5. Return \( ep \).

4.4.1.2 Create-begin-block

A {begin-block} is constructed in the same way as a {procedure}, which it resembles except for the presence of any entry or return information.

Operation: create-begin-block\((cbb)\)

where \( cbb \) is a {begin-block}.

result: a {begin-block}.

Step 1. Let \( abb \) be a {begin-block}. If the {begin-statement} in \( cbb \) contains an {options}, attach an implementation-dependent tree of type {options} to \( abb \).

Step 2. Perform create-block\((cbb, abb)\).

Step 3. For each \(<\text{entry-point}>, ep\), contained in \( abb \), \( abb \) must contain a {procedure} that contains \( ep \).

Step 4. Return \( abb \).
8.4.1.3 \textit{create-block}

\textbf{Operation:} \texttt{create-block}(cb, ab)

\begin{itemize}
  \item where cb is a \{procedure\} or \{begin-block\},
  \item ab is a \{procedure\} or \{begin-block\}.
\end{itemize}

\textbf{Step 1.} For each \{declaration\}, d that is a block-component of cb and that does not
  declaration-contain MEMBER or GENERIC, perform \texttt{create-declaration}(d) to obtain a
  \{declaration\}, ad, and append ad to the \{declaration-list\} immediately contained
  in ab.

\textbf{Step 2.} For each \{declaration\}, ad component of ab such that ad contains at least one
  \{expression-designator\}, \{reference-designator\}, or \{initial-designator\} perform
  \texttt{replace-concrete-designators}(ad).

\textbf{Step 3.} For each \{procedure\}, p that is a block-component of cb perform \texttt{create-
  procedure}(p) to obtain a \{procedure\}, ap and append ap to the \{procedure-list\} in
  ab.

\textbf{Step 4.} For each \{format-statement\}, fs that is a block-component of cb, perform
  \texttt{create-format-statement}(fs) to obtain a \{format-statement\}, afs and append afs to the
  \{format-statement-list\} in ab.

\textbf{Step 5.} If cb immediately contains a \{unit-list\}, let \( n \) be that \{unit-list\}. Otherwise
  let \( n \) be \{absent\}.

\textbf{Case 5.1.} \texttt{cb} is a \{procedure\}.
  Perform \texttt{create-entry-or-executable-unit-list}(u) to obtain an \{entry-or-
  executable-unit-list\} and attach u to ab.

\textbf{Case 5.2.} \texttt{cb} is a \{begin-block\}.
  Perform \texttt{create-executable-unit-list}(u) to obtain an \{executable-unit-
  list\} and attach u to ab.

8.4.1.4 \textit{replace-concrete-designators}

\textbf{Operation:} \texttt{replace-concrete-designators}(ad)

\begin{itemize}
  \item where ad is a \{declaration\} or a \{data-description\}.
\end{itemize}

\textbf{Step 1.} For each \{expression-designator\}, ed component of ad, perform Steps 1.1 and 1.2.

\textbf{Step 1.1.} Let e be the \{expression\} designated by ed.

\textbf{Step 1.2.} Perform \texttt{create-abstract-equivalent-free}(e) to obtain an \{expression\}, se and
  replace ed by se.

\textbf{Step 2.} For each \{reference-designator\}, rd component of ad, perform Steps 2.1 and 2.2.

\textbf{Step 2.1.} Let r be the \{reference\} designated by rd.

\textbf{Case 2.1.1.} rd is an immediate component of \{based\}.
  Perform \texttt{create-reference}(r, \{value-reference\}) to obtain a \{value-
  reference\}, vr.

\textbf{Case 2.1.2.} rd is an immediate component of \{base-item\} or \{offset\}.
  Perform \texttt{create-reference}(r, \{variable-reference\}) to obtain a \{variable-
  reference\}, vr.

\textbf{Step 2.2.} Replace rd by vr.
Step 3. For each \{initial-designator\}, i.e., component of \(ad\), perform Step 3.1.

Step 3.1. Let \(i\) be the \{initial\} designated by \(id\). Perform create-abstract-equivalent-treem\(lit\) to obtain an \{initial\}, \(i\) and replace \(id\) by \(i\).

4.4.1.5 Create-group

Operation: create-group

where \(q\) is a \{group\}.

result: a \{group\}.

Step 1. Let \(d\) be the \{do-statement\} in \(q\). If \(d\) immediately contains a \{unit-list\}, let \(u\) be that \{unit-list\}. Otherwise let \(u\) be \{empty\}.

Case 1.1. \(d\) immediately contains a \{do-spec\}, \(dp\).

Perform create-abstract-equivalent-treem\(lit\) to obtain a \{do-spec\}, \(adp\). Perform create-executable-unit-list\(lit\) to obtain an \{executable-unit-list\}, \(eul\). For each \{entry-point\}, \(ep\) contained in \(eul\), \(eul\) must contain a \{procedure\} that contains \(ep\). Return a \{group\}: \{iterative-group\}: \{controlled-group\}: \(adp\): \(eul\).

Case 1.2. \(d\) immediately contains a \{while-option\}, \(wo\).

Perform create-abstract-equivalent-treem\(lit\) to obtain a \{while-option\}, \(awo\). Perform create-executable-unit-list\(lit\) to obtain an \{executable-unit-list\}, \(eul\). For each \{entry-point\}, \(ep\) contained in \(eul\), \(eul\) must contain a \{procedure\} that contains \(ep\). Return a \{group\}: \{iterative-group\}: \{while-only-group\}: \(awo\): \(eul\).

Case 1.3. (Otherwise).

Perform create-entry-or-executable-unit-list\(lit\) to obtain an \{entry-or-executable-unit-list\}, \(eul\). Return a \{group\}: \{non-iterative-group\}: \(eul\).

4.4.1.6 Create-entry-or-executable-unit-list

Operation: create-entry-or-executable-unit-list

where \(u\) is a \{unit-list\}.

result: an \{entry-or-executable-unit-list\}.

Step 1. Let \(eul\) be an \{entry-or-executable-unit-list\}. If \(u\) is not \{empty\} then for each \{unit\}, \(u\) in \(u\), taken in order, perform Step 1.1.

Step 1.1. If \(u\) immediately contains an \{entry-statement\}, \(es\) then perform create-entry-points\(lit\) to obtain an \{entry-point\}, \(ep\), and append an \{entry-or-executable-unit\}: \(ep\) to \(eul\); otherwise, if \(u\) immediately contains an \{executable-unit\}, \(ew\) then perform create-executable-units\(lit\) to obtain an \{executable-unit\}, \(eu\) and append an \{entry-or-executable-unit\}: \(ew\) to \(eul\).

Step 2. Append an \{entry-or-executable-unit\}: \{executable-unit\}: \{end-statement\} to \(eul\) and return \(eul\).
4.4.1.7 Create-executable-unit-list

Operation: create-executable-unit-list(u)
   where u is a \(\text{\{unit-list\}}\).
   result: an \(\text{\{executable-unit-list\}}\).

Step 1. Let \(\text{\{executable-unit-list\}_u}\) be an \(\text{\{executable-unit-list\}}\). If \(\text{\{executable-unit-list\}_u}\) is not \(\text{\{empty\}}\) then for each \(\text{\{unit\}_u}\) in \(\text{\{executable-unit-list\}_u}\), taken in order, perform Step 1.1.
   Step 1.1. u must not immediately contain an \(\text{\{entry-statement\}}\). If \(\text{\{executable-unit-list\}_u}\) immediately contains an \(\text{\{executable-unit\}_u}\), let \(\text{\{executable-unit\}_u}\) be \(\text{\{executable-unit\}_u}\), and append \(\text{\{executable-unit\}_u}\) to \(\text{\{executable-unit-list\}_u}\).
Step 2. Append an \(\text{\{executable-unit\}_u}\); \(\text{\{end-statement\}_u}\); to \(\text{\{executable-unit-list\}_u}\), and return \(\text{\{executable-unit-list\}_u}\).

4.4.1.8 Create-executable-unit

Operation: create-executable-unit(es)
   where es is an \(\text{\{executable-unit\}}\).
   result: an \(\text{\{executable-unit\}}\).

Step 1. If es immediately contains an \(\text{\{executable-single-statement\}_es}\), let st be the immediate component of es; otherwise let st be the immediate component of es that is not a \(\text{\{prefix\}_es}\).
Step 2. Perform create-abstract-equivalent-tree(st) to obtain \(\text{\{abstract-equivalent\}_st}\). Let \(\text{\{abstract-equivalent\}_st}\) be an \(\text{\{executable-unit\}_st}\).
Step 3. If es immediately contains a \(\text{\{prefix\}_es}\), perform Steps 3.1 and 3.2.
   Step 3.1. Perform create-condition-prefix-list(pl) to obtain a \(\text{\{condition-prefix-list\}_pl}\) or \(\text{\{abstract\}_pl}\). If \(\text{\{abstract\}_pl}\) is not \(\text{\{abstract\}_es}\) then attach \(\text{\{abstract\}_pl}\) to \(\text{\{abstract\}_es}\).
   Step 3.2. Perform create-statement-name-list(pl) to obtain a \(\text{\{statement-name-list\}_pl}\); or \(\text{\{abstract\}_pl}\). If \(\text{\{abstract\}_pl}\) is not \(\text{\{abstract\}_es}\), then attach \(\text{\{abstract\}_pl}\) to \(\text{\{abstract\}_es}\).
Step 4. Return \(\text{\{executable-unit\}_es}\).

4.4.1.9 Create-entry-point

An \(\text{\{entry-point\}_es}\) may be the primary \(\text{\{entry-point\}_es}\) of a \(\text{\{procedure\}_es}\), or a secondary \(\text{\{entry-point\}_es}\) specified in the concrete \(\text{\{procedure\}_es}\) by an \(\text{\{entry-statement\}_es}\). Each \(\text{\{entry-point\}_es}\) has its own information for entry and return which becomes the \(\text{\{entry-information\}_es}\) and \(\text{\{returns-descriptor\}_es}\) respectively.

Operation: create-entry-point(es)
   where es is an \(\text{\{entry-statement\}_es}\) or a \(\text{\{procedure-statement\}_es}\).
   result: an \(\text{\{entry-point\}_es}\).

Step 1. Let ep be an \(\text{\{entry-point\}_es}\); \(\text{\{entry-information\}_es}\);.
   Case 1.1. es is an \(\text{\{entry-statement\}_es}\).
   Let pl be the \(\text{\{statement-name-list\}_pl}\) of the \(\text{\{unit\}_pl}\) immediately containing es.
   Case 1.2. es is a \(\text{\{procedure-statement\}_es}\).
   Let pl be the \(\text{\{prefix-list\}_pl}\) of the \(\text{\{procedure\}_pl}\) immediately containing es.
Step 2. Perform create-statement-name-list(pl) to obtain a <statement-name-list> in sl. Let add be a copy of the <statement-name> in sl, and attach add to el.

Step 3. If add has a <parameter-name-conmlist>, perform create-abstract-equivalent-tree(mln) to obtain a <parameter-name-list> in ml and attach add to el.

Step 4. If add has a <returns-descriptor>, then perform Step 4.1.

Step 4.1. Let id be the first <identifier> in rd. Perform create-data-description(d) to obtain a <data-description> in d. Perform replace-concrete-designators(d) in d. Let rdn be a <returns-descriptor>; rdn and attach rdn to el.

Step 5. If add has an <options>, attach an implementation-dependent tree of type <options> to el.

Step 6. Return el.

4.6.1.10 Create-statement-name-list

Operation: create-statement-name-list(pl)
where pl is a <prefix-list> or a <statement-name-list>
result: a <statement-name-list>

Step 1. Let sl be a <statement-name-list>. For each <statement-name> in pl, perform Steps 1.1 and 1.2.

Step 1.1. Let id be the <identifier> in pl. Perform create-identifier(id) to obtain an <identifier> in d. Let add be a <statement-name>; add. For each <expression> in sl, taken in order, perform Step 1.1.1.

Step 1.1.1. Let add be a <expression> whose concrete-representation is of type <expression> in add. Append add to the <expression-list> in add.

Step 1.2. Append add to sl.

Step 2. If sl has any subnodes, return sl; otherwise return <empty>.

4.6.1.11 Create-condition-prefix-list

Operation: create-condition-prefix-list(pl)
where pl is a <prefix-list> or a <condition-prefix-conmlist>
result: a <condition-prefix-list>

Step 1. Let pl be a <condition-prefix-list>.

Step 2. For each <computational-condition> in pl, perform Step 2.1.

Step 2.1. Perform create-condition(d) to obtain a <computational-condition> in pl and append a <condition-prefix-list> to pl.

Step 3. For each <disabled-computational-condition> in pl, perform Step 3.1.

Step 3.1. Perform create-condition(d) to obtain a <computational-condition> in pl. Append a <condition-prefix-list> to pl.

Step 4. If sl has any subnodes, return sl; otherwise return <empty>.
4.3.1.12 Create-condition

**Operation**: create-condition(c)

where c is a \{computational-condition\}, \{disabled-computational-condition\}, or \{condition-name\}, or \{io-condition\}.

result: a \{computational-condition\}, \{io-condition\}, or \{condition-name\}.

**Case 1.** c is a \{computational-condition\}.

Return a \{computational-condition\}; \{x-condition\}; where "x" is the lowercase equivalent of the concrete-representation of c.

**Case 2.** c is a \{disabled-computational-condition\}.

Return a \{computational-condition\}; \{x-condition\}; where x is a name such that "nox" is the lowercase equivalent of the concrete-representation of c.

**Case 3.** c is an \{io-condition\}.

Return an \{io-condition\}; \{x-condition\}; where "x" is the lowercase equivalent of the concrete-representation of c.

**Case 4.** c is a \{condition-name\} that contains AREA, ERROR, FINISH, or STORAGE.

Return a \{condition-name\}; \{x-condition\}; where "x" is the lowercase equivalent of the concrete-representation of c.

**Case 5.** c is a \{condition-name\}; \{computational-condition\}, co.

Perform create-condition(co) to obtain a \{computational-condition\}, co, and return a \{Condition-name\}; co.

**Case 6.** c is a \{condition-name\}; \{named-io-condition\}; \{io-condition\}, loc (\{reference\}, ref).

Perform create-condition(loc) to obtain an \{io-condition\}, loc, and perform create-reference(ref, \{value-reference\}) to obtain a \{value-reference\}, vr. Return a \{condition-name\}; \{named-io-condition\}; loc vr.

**Case 7.** c is a \{condition-name\}; \{programmer-named-condition\}.

Let Id be the \{identifier\} in c. Perform find-applicable-declaration(Id) to obtain a \{declaration\}, dd. Let des be a \{declaration-designator\} designating the \{declaration\} whose \{declaration-designator\} designates dd. Return a \{condition-name\}; \{programmer-named-condition\}; des.
4.6.2 Creation of Statements

4.6.2.1 Create-assignment-statement

Operation: create-assignment-statement (ast)

where ast is an <assignment-statement>.

result: an <assignment-statement>.

Step 1. ast immediately contains {,}, by, and NAME.

Case 1.1. ast immediately contains {,}, by, and NAME.

Perform create-by-name-assignment (ast) to obtain an <assignment-statement>, ast or a <null-statement>, ast. If ast is a <null-statement> then return ast.

Case 1.2. (Otherwise).

Let ast be an <assignment-statement>, <target-reference-list>, trl. For each <reference>, r, immediately contained in the <reference-context-list> in ast, taken in left-to-right order, perform create-reference (<reference>, trl, and append tr to trl. Let e be the <expression> immediately contained in ast. Perform create-expression(e) to obtain an <expression>, ae, and attach ae to ast.

Step 2. The <data-description> of ae must be proper for assignment to the <data-description> of each <target-reference> in ast.

Step 3. Return ast.

4.6.2.2 Create-by-name-assignment

Operation: create-by-name-assignment (ast)

where ast is an <assignment-statement>.

result: an <assignment-statement> or <null-statement>.

Step 1. Perform create-by-name-parts-list (ast) to obtain a <by-name-parts-list>, bpl or <absent>, bpl. If bpl is <absent>, return a <null-statement>.

Step 2. Let ast be an <assignment-statement>, <target-reference-list>, trl. For each <reference>, r, immediately contained in the <reference-context-list> in ast, taken in left-to-right order, perform Step 2.1.

Step 2.1. Perform create-reference (<reference>, trl, trl, and append tr to trl.

Step 3. Let e be the <expression> immediately contained in ast. Perform create-expression(e, bpl) to obtain an <expression>, ae and attach ae to ast.

Step 4. Each <builtin-function-reference> or <procedure-function-reference> simple component of ast, which is not contained in a <locator-qualifier>, is a <subscript>, or in a <builtin-function-reference> whose result has <aggregate-type> <scalar>, is an "Attribute" for each builtin function in Section 4.6.4, ast immediately contains a <data-description> which simply contains an <item-data-description>.

Step 5. Return ast.
8.4.2.3 Data-descriptions Proper for Assignment

The <data-description>,dds is proper for assignment to the <data-description>,ddt if and only if the following conditions exist:

1. The <aggregate-type> of dds is promotable (see Section 7.5.3.1) to the <aggregate-type> of ddt.

2. Corresponding <data-type>s of dds and ddt:
   2.1 both have <computational-type>, or
   2.2 both have <locator>, or
   2.3 both have <non-computational-type>, with the immediate subsdes of the <name> <computational-type>s belonging to the same category other than <locator>.

   Further, if one <data-type> has <offset> and the other has <pointer>, then the <offset> must contain a <variable-reference>.

8.4.2.4 Create-by-name-parts-list

Operation: create-by-name-parts-list(ars)
   where ars is an <assignment-statement> or an <arguments>.

   result: a <by-name-parts-list> or <absent>.

Step 1. Let bnp be a <by-name-parts-list-list> with no components.

Step 2. For each <reference> in immediately contained in the <reference-consortlist> of ars perform Step 2.1.

   Step 2.1. Perform find-applicable-declaration[re] to obtain a <declaration>,cd, which must <declaration-contains STRUCTURE>. Perform find-by-name-parts[lcd] to obtain a <by-name-parts-list>,tcp and append tcp to bnp.

Step 3. For each <reference> contained in the <expression> immediate component of ars, but not contained in a <locator-qualifier> or <arguments>, perform Steps 3.1 and 3.2.

   Step 3.1. Perform find-applicable-declaration[re] to obtain a <declaration>,cd.

   Step 3.2. If cd <declaration-contains STRUCTURE then perform find-by-name-parts[lcd] to obtain a <by-name-parts-list>,tcp and append tcp to bnp.

Step 4. Let rbnp be a <by-name-parts-list> consisting of those <by-name-parts> which are common to every <by-name-parts-list> of bnp.

Step 5. If rbnp is not empty then return rbnp; otherwise return <absent>.

102
4.4.2.5 Find-by-name-parts

Operation: find-by-name-parts(d)

where d is a <declaration>.

result: a <by-name-parts-list>.

Step 1. Let d1 be the <declaration-components> node that immediately contains d. Let e1j be the j'th immediate component of d1 that is not a {,}. Let n be the number of such components and let k be such that e1k is d. Let id be the numeric value of the <level> of d. Let i be k+1.

Step 2. Let impl be a <by-name-parts-list> with no components.

Step 3. While i<n perform Steps 3.1 through 3.4.

Step 3.1. If e1i does not declaration-contain MEMBER then go to Step 4.

Step 3.2. Let le be the numeric value of the <level> of e1i. If le<id, go to Step 4.

Step 3.3. If le = id+1 then perform Steps 3.3.1 and 3.3.2.

Step 3.3.1. Let c1d be the <identifier> of e1i. Perform create-identifier(c1d) to obtain an <identifier>.

Step 3.3.2. Case 3.3.2.1: e1i declaration-contains STRUCTURE.

Perform find-by-name-parts(e1i) to obtain a <by-name-parts-list>,rstep. Attach a copy of id to each <identifier-list> component of rstep as the initial element. Append a copy of each immediate component of rstep to impl.

Case 3.3.2.2. (Otherwise).

Append a <by-name-parts> containing id to impl.

Step 3.4. Let i = i+1.

Step 4. Return impl.

4.4.2.6 Create-allocation

Operation: create-allocation(al)

where al is an <allocation>.

result: an <allocation>.

Step 1. Let id be the <identifier> immediately contained in al. Perform find-applicable-declaration(id) to obtain a <declaration>,ocl. ocl must not declaration-contain MEMBER.

Step 2. Let addr be the <declaration> whose <declaration-designator> designates ocl. If addr contains <controlled>, then al must not contain a <set-option> or an <in-option>. If addr contains a <cased>, b and al does not contain a <foot-option>, then b must immediately contain a <value-reference> that immediately contains a <variable-reference>.

Step 3. Let decl be a <declaration-designator> designating addr. Let acl be an <allocation>: den. If acl contains a <foot-option>,p, then perform create-abstract-equivalent-tree(p) to obtain a <set-option>,p, and attach p to acl. If acl contains an <in-option>,i, then perform create-abstract-equivalent-tree(i) to obtain an <in-option>,i, and attach i to acl.

Step 4. Return acl.

Chapter 4: The Translactor 103
8.4.2.7 Create-format-statement

Operation: create-format-statement(fs)

where fs is a <format-statement>.

result: a <format-statement>.

Step 1. Let fsl be the <format-specification-consmalst> component of fs. Perform create-abstract-equivalent-tree(fsl) to obtain a <format-specification-list>, afs1.

Step 2. The <unit> that immediately contains fs must immediately contain a <prefix-list>, pl. Perform create-statement-name-list(pl) to obtain a <statement-name-list>, s1, which must not be <absent>. Let afs be a <format-statement>; s1 afs1.

Step 3. Perform create-condition-prefix-list(pl) to obtain a <condition-prefix-list>, cpl, or <absent>, cpl. If cpl is not <absent>, attach cpl to afs.

Step 4. Return afs.

8.4.2.8 Create-format-iteration

Operation: create-format-iteration(fi)

where fi is a <format-iteration>.

result: a <format-iteration>.

Step 1. Let afi be a <format-iteration>.

Step 2. Let ff be the <format-iteration-factor> immediately contained in fi. If ff immediately contains an <expression>, let e be that <expression>; otherwise let e be an <expression> to which the immediately contained <integer> of ff has been attached. Perform create-iteration(e) to obtain an <expression>, se and attach a <format-iteration-factor>; se to afi.

Step 3:

Case 3.1. fi immediately contains a <format-specification-consmalst>, fsc.

Perform create-abstract-equivalent-tree(fsc) to obtain a <format-specification-list>, fsl, and attach fsl to fi.

Case 3.2. fi immediately contains a <format-item>, ft.

Perform create-abstract-equivalent-tree(ft) to obtain a <format-item>, ftt and attach a <format-specification-list>; <format-specification>; ftt to afi.

Step 4. Return afi.
8.4.2.9 create-freeling

Operation: \texttt{create-freeling}\(fr\)

where \(fr\) is a <freeling>.

result: a <freeling>.

Step 1. Let \(id\) be the <identifier> immediately contained in \(fr\). Perform find-applicable-declaration(\(id\)) to obtain a <declaration>|\(cdcl\). \(cdcl\) must not declaration-contain MEMBER.

Step 2. Let \(adcl\) be the <declaration> whose <declaration-designator> designates \(cdcl\). If \(adcl\) contains <controlled>, then \(fr\) must not contain an <in-option> or a <locator-qualifier>.

Step 3. Let \(des\) be a <declaration-designator> designating \(adcl\). Let \(atr\) be a <freeling>|\(des\). If \(fr\) immediately contains a <locator-qualifier> with <reference>|\(vr\), perform create-reference(\(des\),<value-reference>|\(vr\)) to obtain a <value-reference>|\(vr\) and attach a <locator-qualifier>|\(vr\) to \(atr\). If \(atr\) contains an <in-option>|\(io\), perform create-abstract-equivalent-trivial to obtain an <in-option>|\(io\), and attach \(io\) to \(atr\). Return \(atr\).

8.4.2.10 create-if-statement

Operation: \texttt{create-if-statement}\(ifs\)

where \(ifs\) is a <if-statement>.

result: an <if-statement>.

Step 1. \(ifs\) immediately contains an <if-clause> which contains an <expression>|\(es\). Perform create-expression(\(es\)) to obtain an <expression>|\(es\).

Step 2.

Case 2.1. \(ifs\) immediately contains an <executable-unit>|\(eu\), but not an ELSE.

Perform create-executable-unit(\(eu\)) to obtain an <executable-unit>|\(eu\). Return an <if-statement>

\texttt{<if-statement>};
<test>: ae;
<then-unit>: eu1;
<else-unit>: eu2;

Case 2.2. \(ifs\) immediately contains a <balanced-unit>|\(bu\), an ELSE, and an <executable-unit>|\(eu\).

Perform create-balanced-unit(\(bu\)) to obtain an <executable-unit>|\(eu\). Perform create-executable-unit(\(eu\)) to obtain an <executable-unit>|\(eu\). Return an <if-statement>

\texttt{<if-statement>};
<test>: ae;
<then-unit>: eu1;
<else-unit>: eu2.
4.4.2.11 Create-balanced-unit

Operation: \texttt{create-balanced-unit(bu)}

where \texttt{bu} is a \texttt{\{balanced-unit\}}.

result: an \texttt{\{executable-unit\}}.

Step 1.

Case 1.1. \texttt{bu} immediately contains an \texttt{\{executable-single-statement\}}.

Let \(a\) be the immediate component of \texttt{\{executable-single-statement\}}. Perform \texttt{create-abstract-equivalent-tree}() to obtain \texttt{stat}.

Case 1.2. \texttt{bu} immediately contains \{group\}, \{begin-block\}, or \{on-statement\}.

Let \(a\) be that \{group\}, \{begin-block\}, or \{on-statement\}. Perform \texttt{create-abstract-equivalent-tree}() to obtain \texttt{stat}.

Case 1.3. (Otherwise).

Let \(a\) be the \{expression\} immediately contained in the \{if-clause\} immediately contained in \texttt{bu}. Perform \texttt{create-expression}() to obtain an \texttt{\{expression\}}, \texttt{\{test\}}. Let \texttt{\{then-unit\}} and \texttt{\{else-unit\}} be, in order, the \{balanced-unit\}'s immediately contained in \texttt{bu}. Perform \texttt{create-balanced-unit(bu1)} to obtain \texttt{eu1} and \texttt{create-balanced-unit(bu2)} to obtain \texttt{eu2}. Let \texttt{stat} be so

\[
\texttt{\{if-statement\}}:
\begin{align*}
\texttt{\{test\}}: \texttt{\{test\}}; \\
\texttt{\{then-unit\}}: \texttt{\{then-unit\}}; \\
\texttt{\{else-unit\}}: \texttt{\{else-unit\}};
\end{align*}
\]

Step 2. Let \texttt{eu} be an \texttt{\{executable-unit\}}: \texttt{stat}. If \texttt{eu} immediately contains a \{prefix-list\}, perform Steps 2.1 and 2.2.

Step 2.1. Perform \texttt{create-statement-name-list(pl)} to obtain a \texttt{\{statement-name-list\}}, \texttt{\{nil\}} or \texttt{\{absent\}}, \texttt{\{nil\}}. If \texttt{\{nil\}} is not \texttt{\{absent\}}, attach \texttt{\{nil\}} to \texttt{eu}.

Step 2.2. Perform \texttt{create-condition-prefix-list(pl)} to obtain a \texttt{\{condition-prefix-list\}}, \texttt{\{nil\}} or \texttt{\{absent\}}, \texttt{\{nil\}}. If \texttt{\{nil\}} is not \texttt{\{absent\}}, attach \texttt{\{nil\}} to \texttt{eu}.

Step 3. Return \texttt{eu}.

4.4.2.12 Create-locate-statement

Operation: \texttt{create-locate-statement(1x)}

where \(x\) is a \texttt{\{locate-statement\}}.

result: a \texttt{\{locate-statement\}}.

Step 1. Let \(id\) be the \{identifier\} immediately contained in \texttt{1x}. Perform \texttt{find-applicable-declaration(id)} to obtain a \texttt{\{declaration\}}, \texttt{\{cdcl\}}. \texttt{\{cdcl\}} must not \texttt{\{declarations\}} contain \texttt{\{REDUX\}}.

Step 2. Let \(des\) be a \{declaration-designator\} designating the \texttt{\{declaration\}}, \texttt{\{ad\}}, whose \{declaration-designator\} designates \texttt{\{cdcl\}}, \texttt{\{ad\}} must contain \texttt{\{based\}}. Let \(als\) be a \{locate-statement\} \{des\}. For each immediate submode, \(x\), \(al\) is other than \(id\) or \texttt{LOCATE}, perform \texttt{create-abstract-equivalent-tree}() to obtain a \texttt{\{file-option\}}, \texttt{\{pointer-set-option\}}, \texttt{\{y\}} or a \texttt{\{stepfrom-option\}}, \texttt{\{y\}}, and attach \texttt{\{y\}} to \texttt{\{als\}}.

Step 3. Return \texttt{als}.
4.4.2.13 Create-on-statement

Operation:  create-on-statement(on)

where on is an <on-statement>.

result: an <on-statement>.

Step 1.

Case 1. on immediately contains SYSTEM.

Let p be a <system-action>.

Case 1.2. on immediate contains an <on-unit>, on.

Step 1.2.1.

Case 1.2.1.1. on immediately contains an <executable-single-statement>, ess.

Let e be the immediate component of ess.

Case 1.2.1.2. on immediately contains a <begin-block>.

Let e be that <begin-block>.

Step 1.2.2. Perform create-abstract-equivalent-treat(s) to obtain ee. Let ep be an <entry-point> whose only subsode is <entry-information>. Let eul be an

<entry-or-executable-unit-list>:
<entry-or-executable-unit>: ep;
<entry-or-executable-unit>:
<executable-unit>: ee;
<executable-unit>:
<executable-unit>:
<end-statement>.

Step 1.2.3. If on immediately contains a <condition-prefix-constraint-list>, cpl, perform create-condition-prefix-list(cpl) to obtain acpl, and attach acpl to ee.

Step 1.2.4. Let p be an <on-unit>; <procedure>: eul.

Step 2. Let aso be an <on-statement>; p. If on immediately contains SNAP attach <snap> to aso. For each <condition-name>, cn in the <condition-name-constraint-list> in on, perform create-condition(cn) to obtain a <condition-name>, acn, and append acn to the <condition-name-list> in aso. Return aso.

4.4.3 CREATE-DECLARATION

Operation:  create-declaration(d)

where d is a <declaration>.

result: a <declaration>.

Step 1. Let cid be the <identifier> of d. Perform create-identifier(cid) to obtain an <identifier>, id.

Step 2.

Case 2.1. d contains INTERNAL.

Let sc be <scope>: <internal>.

Case 2.2. d contains EXTERNAL.

Let sc be <scope>: <external>.
Step 3.

Case 3.1. d contains VARIABLE.

Perform create-variable(d) to obtain a <variable>, v. Let dt be <declaration-type>: v.

Case 3.2. d contains CONSTANT.

Perform create-named-constant(d) to obtain a <named-constant>, nc. Let dt be <declaration-type>: nc.

Case 3.3. d contains BUILTIN.

Let dt be a <declaration-type>: <builtin>.

Case 3.4. d contains CONDITION.

Let dt be a <declaration-type>: <condition>.

Step 4. Let dd be a <declaration-designator> that designates d.

Step 5. Return <declaration>: id nc dt dd.

4.4.3.1 Create-named-constant

Operation: create-named-constant(d)

where d is a <declaration>.

result: a <named-constant>.

Step 1. Let nc be a <named-constant>.

Step 2. If d declaration-contains a <dimension-attribute>, ds then perform create-bounded-pair-list(ds) to obtain a <bounded-pair-list>, bpl and attach bpl to <c>.

Step 3. If d declaration-contains ENTRY.

Perform create-entry(d) to obtain an <entry>, ee. Attach ee to nc.

Case 3.2. d declaration-contains FILE.

Attach <file> to nc. Let fd be a <file-description>. For each <attribute> component of d which declaration-contains STREAM, RECORD, INPUT, OUTPUT, UPDATE, SEQUENTIAL, DIRECT, PRINT, KEYED, or ENVIRONMENT attach <stream>, <record>, <input>, <output>, <sequential>, <direct>, <print>, <keyed>, or <environment>, respectively, to fd. Attach fd to nc. If <environment> was attached then perform some implementation-defined action.

Case 3.3. d declaration-contains FORMAT or LABEL.

Attach <format> or <label> respectively, to nc.

Step 6. Return nc.
8.1.3.2 Create-variable

For each distinct variable a <declaration> which contains <variable> is constructed and completed according to the declared attributes of the item. A <variable> may be referred to by a <value-reference>, a <target-reference>, or a <subroutine-reference> (see Section 8.4.3).

Operation: create-variable(d)

where d is a <declaration>.

results: a <variable>.

Step 1.

Case 1.1. d contains AUTOMATIC.

Let st be a <storage-type>; <storage-class>; <automatic>.

Case 1.2. d contains CONTROLLED.

Let st be a <storage-type>; <storage-class>; <controlled>.

Case 1.3. d contains STATIC.

Let st be a <storage-type>; <storage-class>; <static>.

Case 1.4. d contains a <attribute>, atr that immediately contains BASED.

Let st be a <storage-type>; <storage-class>; <based>; if atr also simply contains a <reference>, r then attach a <reference-designator>; designator of r; to st.

Note: The translation of r will be completed after the processing of all <declare-statements>.

Case 1.5. d contains DEFINED <reference>, e or DEFINED reference, r.

Let st be

<storage-type>:
  <defined>;
  <base-item>;
  <reference-designator>:
  designator of r;

if d contains the form <attribute>, atr: POSITION then atr must have an <expression>.

if d contains <attribute>, atr which immediately contains POSITION <expression>, e then attach a <position>; <expression-designator>; designator of e.

Note: The translation of e will be completed after the processing of all <declare-statements>.

Case 1.6. d contains <attribute> PARAMETER.

Let st be a <storage-type>; <parameter>.

Step 2. Perform create-data-description(d) to obtain a <data-description>, de.

Step 3. Return a <variable>; st de.
4.4.3.1 Create-bound-pair-list

Operation: create-bound-pair-list(da)

where da is a dimension-attribute.

result: a bound-pair-list.

Step 1. Let bpl be the bound-pair-constructor component of da.

Step 2. For each bound-pair, bp of bpl such that bp contains an upper-bound and does not contain a lower-bound, attach a lower-bound; un to bp, where un is a copy of an extent-expression whose concrete representation is the character '*'.

Step 3. Let abpl be a bound-pair-list. For each bound-pair, bp component of bpl in left-to-right order perform Step 3.1.

Step 3.1.

Case 3.1.1. bp contains *.

Append a bound-pair: asterisk to abpl.

Case 3.1.2. (Otherwise).

Step 3.1.2.1. Let ab be a

<bound-pair>

<lower-bound>, ab

<upper-bound>, abs.

Attach to ab an expression-designator designating the expression simply contained in the lower-bound component of bp. Attach to abs an expression-designator designating the expression simply contained in the upper-bound component of bp.

Step 3.1.2.2. If the lower-bound component of bp contains a refer-option, re then perform create-refer-optionIcon to obtain a refer-optionIcon and attach are to ab.

Step 3.1.2.3. If the upper-bound component of bp contains a refer-option, re perform create-refer-optionIcon to obtain a refer-optionIcon and attach are to abs.

Step 3.1.2.4. Append ab to abpl.

Step 4. Return abpl.

4.4.3.4 Create-data-description

The <data-description> component of a <declaration> specifies the aggregate properties of a variable or the descriptor of a variable, and also the data properties associated with elements of the variable. This distinction between structures, arrays, and scalars is made at a high level, and the individual element data properties are attached to each scalar.

Operation: create-data-description(d)

where d is a declaration, a description, or a generic-description whose subtree would be a valid subtree of a description.

result: a <data-description>.

Step 1. Let dd be a <data-description>.

Step 2. If d declaration-contains a dimension-attribute, da then perform Steps 2.1 and 2.2.
Step 2.1. Perform create-bound-pair-list(0d) to obtain a <bound-pair-list>, abpl.


Step 3. If d declaration-contains STRUCTURE then perform Steps 3.1 through 3.4.

Step 3.1. Let di be the node that immediately contains d. Let eij be the j'th immediate component of di that is not a f,j. Let n be the number of such components and let k be such that e((k) is identical to d. Let i be k+1. Let id be the numeric value of the flevel of d.

Step 3.2. Let id be an <identifier-list> and let nli be a <member-description-list>.

Step 3.3. While lih perform Steps 3.3.1 through 3.3.4.

Step 3.3.1. If nli does not declaration-contains MEMBER then go to Step 3.4.

Step 3.3.2. Let le be the numeric value of the flevel of nli. If le id then go to Step 3.4.

Step 3.3.3. If le id+1 then perform Steps 3.3.3.1 through 3.3.3.3.

Step 3.3.3.1. Perform create-data-description(nli) to obtain a <data-description>, add.

Step 3.3.3.2. Append a <member-description>: add; to nli.

Step 3.3.3.3. If nli is a declaration then let cid be the <identifier> of nli, perform create-identifier(cid) to obtain an <identifier>, id, and append id to idl.

Step 3.3.4. Let i be i+1.

Step 3.4. If idl has no components then let add be a <structure-data-description>: addl; otherwise let add be a <structure-data-description>: addl addi. Attach add to dd.

Step 4. If d does not declaration-contains STRUCTURE then perform Steps 4.1 through 4.4.

Step 4.1. One of the following cases must apply:

Case 4.1.1. d declaration-contains ALIGNED.

Let ai be <alignment>: <aligned>.

Case 4.1.2. d declaration-contains UNALIGNED.

Let ai be <alignment>: <unaligned>.

Step 4.2. Perform create-data-type(ai) to obtain a <data-type>, dt. Let idd be an <item-data-description>: a dt.

Step 4.3. If d declaration-contains {initial}, list then let idd be an {initial-designator} that designates list and attach into to idd.

Step 4.4. Attach idd to dd.

Step 5. Return dd.
4.4.3.1 Create-data-type

Operation: \texttt{create-data-type}[^2]

\[ \text{where } d \text{ is a } \texttt{declaration}, \text{ a } \texttt{description}, \text{ or a } \texttt{generic-description} \text{ that is restricted to those forms that are equivalent to } \texttt{description}. \]

result: a \texttt{data-type}.

One and only one of the following cases must apply:

Case 1. \texttt{d declaration-contains AREA}\{\texttt{area-size}\},\{\texttt{area}\}.

\begin{itemize}
  \item \textbf{Step 1.1.} Let \texttt{ar} be \texttt{area}. If \texttt{ar} immediately contains an \texttt{#} then let \texttt{ed} be an \texttt{expression-designator} that designates the \texttt{expression} simple component of \texttt{ar}. Attach \texttt{ed} to \texttt{ar}.
  \item \textbf{Step 1.2.} If \texttt{ar} contains a \texttt{refer-option}, perform \texttt{create-refer-option(re) to obtain a \texttt{refer-option},ar and attach ar to \texttt{ar}}.
  \item \textbf{Step 1.3.} Return \texttt{data-type}: \texttt{one-computational-type}: \texttt{ar}.
\end{itemize}

Case 2. \texttt{d declaration-contains ENTRY}.

Perform \texttt{create-entry} to obtain an \texttt{entry},e and return a \texttt{data-type}: \texttt{non-computational-type}: \texttt{e}.

Case 3. \texttt{d declaration-contains FILE}.

Return a \texttt{data-type}: \texttt{non-computational-type}: \texttt{file}.

Case 4. \texttt{d declaration-contains FORMAT} or \texttt{LABEL}.

Attach \texttt{format} or \texttt{label} respectively to \texttt{data-type},dt: \texttt{non-computational-type}. If \texttt{d declaration-contains LOCAL then attach \texttt{local} to dt. Return dt}.

Case 5. \texttt{d declaration-contains POINTER or OFFSET}\{\texttt{reference},r\}.

Let \texttt{dt} be a \texttt{data-type}: \texttt{non-computational-type}: \texttt{locator},loc. Attach \texttt{pointer} or \texttt{offset},ofs, respectively to \texttt{loc}. If \texttt{r} exists then attach a \texttt{expression-designator} that designates \texttt{r} to \texttt{ofs}.

Case 6. \texttt{d declaration-contains PICTURES}\{\texttt{picture},p\}.

Perform \texttt{create-picture(p) to obtain a \texttt{picture},p and return a \texttt{data-type}: \texttt{computational-type}: \texttt{p}.}

Case 7. \texttt{d declaration-contains a }\texttt{data-attribute},sa which \texttt{declaration-contains CHARACTER} or \texttt{BIT}.

\begin{itemize}
  \item \textbf{Step 7.1.} If \texttt{sa} contains \texttt{CHARACTER} then let \texttt{at} be a \texttt{string-type}: \texttt{character}. Otherwise, let \texttt{at} be a \texttt{string-type}: \texttt{bit}.
  \item \textbf{Step 7.2.} Let \texttt{ml} be the \texttt{maximum-length} component of \texttt{sa}. Let \texttt{ml} be a \texttt{maximum-length}. If \texttt{ml} immediately contains an \texttt{#} then let \texttt{ed} be an \texttt{expression-designator} that designates the \texttt{expression} simple component of \texttt{ml}. Attach \texttt{ed} to \texttt{ml}.
  \item \textbf{Step 7.3.} If \texttt{ml} contains a \texttt{refer-option}, CE then perform \texttt{create-refer-option(re) to obtain a \texttt{refer-option},ar and attach ar to \texttt{ml}}.
  \item \textbf{Step 7.4.} If \texttt{d declaration-contains VARYING let \texttt{v} be \texttt{varying} otherwise let \texttt{v} be \texttt{nonvarying}.
  \item \textbf{Step 7.5.} Return a \texttt{data-type}: \texttt{computational-type}: \texttt{string}: \texttt{at and v}.}
Case 8. (Otherwise).

Step 8.1. Let \( \text{dt} \) be a

\[
\text{<data-type>}
\]

\[
\quad \text{<computational-type>}
\]

\[
\quad \text{<arithmetic>}
\]

\[
\quad \text{<mode>}, m
\]

\[
\quad \text{<base>}, b
\]

\[
\quad \text{<scale>}, s
\]

\[
\quad \text{<precision>}, p
\]

Step 8.2. If \( \text{d} \) declaration-contains REAL then attach \( \text{<real>} \) to \( m \); otherwise, attach \( \text{<complex>} \).

Step 8.3. If \( \text{d} \) declaration-contains FIXED then attach \( \text{<binary>} \) to \( b \); otherwise, attach \( \text{<decimal>} \).

Step 8.4. If \( \text{d} \) declaration-contains FIXED then attach \( \text{<fixed>} \) to \( s \); otherwise, attach \( \text{<fraction>} \).

Step 8.5. Let \( \text{ap} \) be the \( \text{<precision>} \) declaration-component of \( \text{d} \). Perform create-

abstract-equivalent-trees() to obtain a \( \text{<precision>} \) ap. The \( \text{<number-of-
digits>} \) in ap must not be greater than the maximum \( \text{<number-of-digits>} \) allowed for the \( \text{<base>} \) and \( \text{<scale>} \) of \( \text{dt} \). Replace \( p \) by \( ap \).

Step 8.6. Return \( \text{dt} \).

9.4.3.6 Create-entry

Operation: \( \text{create-entry}(\theta) \)

\[
\text{where } \theta \text{ is a } \{\text{declaration}\}, \text{ a } \{\text{description}\}, \text{ or a } \{\text{generic-description}\} \text{ that declaration-contains an ENTRY } \{\text{description-commentlist}, \text{doc}\}.
\]

result: an \( \text{<entry>} \).

Step 1. Let \( \text{ent} \) be an \( \text{<entry>} \) with no subnodes.

Step 2. If \( \text{d} \) exists, then for each \( \{\text{description}\}, \text{pd} \) in \( \text{d} \) that does not declaration-

contain MEMBER, perform Steps 2.1 and 2.2.

Step 2.1. Perform create-data-description(pd) to obtain a \( \text{<data-description>} \) \( \theta \).

Step 2.2. Append a \( \text{<parameter-descriptor>} \) \( \theta \) to the \( \text{<parameter-descriptor-list>} \) in

\( \text{ent} \).

Step 3. If \( \text{d} \) declaration-contains RETURNS \( \{\text{description-commentlist}, \text{gt}\} \), perform steps

3.1 through 3.3.

Step 3.1. Let \( \text{id} \) be the \( \{\text{description}\} \) immediate component of \( \text{d} \) that does not declaration-contain MEMBER. There must be exactly one such \( \{\text{description}\} \).

Step 3.2. Perform create-data-description(id) to obtain a \( \text{<data-description>} \) \( \theta \).

Step 3.3. Attach a \( \text{<returns-descriptor>} \) \( \theta \) to \( \text{ent} \).

Step 4. If \( \text{d} \) contains OPTIONS, attach \( \text{<options>} \) with some implementation-defined

subnodes, to \( \text{ent} \).

Step 5. Return \( \text{ent} \).
8.8.1.7 Create-refer-option

Operation: create-refer-option creo
where creo is a refer-option.
result: a refer-option.

Step 1. Let ur be the (unsubscripted-reference) of creo. Perform find-applicable-declaration(ur) to obtain d.

Step 2. Perform find-fully-qualified-name(d) to obtain an identifier-list, idli.

Step 3. Perform create-abstract-equivalent-truth(idli) to obtain an <identifier-list>, aidli.

Step 4. Return a refer-option: aidli.

8.8.1.8 Create-identifier

Operation: create-identifier(id)
where id is an identifier.
result: an identifier.

Step 1. Return an identifier whose concrete-representation is the same as that of id.

8.8.1.9 Create-initial-element

Operation: create-initial-element(dom)
where dom is an initial-element.
result: an initial-element.

Step 1. Let aine be an initial-element.

Case 1.1. aine immediately contains * as its only component.
    Attach an <asterisk> to aine.

Case 1.2. aine immediately contains a parenthesized-expression, pee.
    Perform create-abstract-equivalent-truth(pee) to obtain a parenthesized-expression, pe, and attach pe to aine.

Case 1.3. aine immediately contains an initial-constant-one, ico.
    Let e be the expression whose concrete-representation is the same as the concrete-representation of ico. Perform create-expression(e) to obtain an expression, ee, and attach a parenthesized-expression, ee, to aine.

Case 1.4. aine immediately contains an iteration-factor, itf.
    Perform create-abstract-equivalent-truth(itf) to obtain aitf and attach aitf to aine.

Case 1.4.1. aine immediately contains an *.
    Attach an initial-element-list, iel, initial-element, <asterisk>; to aine.
Case 1.4.2. \textit{icn} immediately contains an \texttt{initial-constant-twelv}, \texttt{icn}.

Let \( e \) be the \texttt{expression} whose concrete representation is the same as the concrete representation of \( \texttt{icn} \). Perform \texttt{create-expression(e)} to obtain an \texttt{expression}, \texttt{as}. Attach an \texttt{initial-element-list} \texttt{<initial-element> <parenthesized-expression> as \texttt{> to \texttt{icn}>}}.

Case 1.4.3. \textit{icn} immediately contains an \texttt{initial-element-conma-list}, \texttt{icn}.

Perform \texttt{create-abstract-equivalent-tree\text{<\texttt{icn}>}} to obtain \texttt{aln}, and attach \texttt{aln} to \texttt{icn}.

\textbf{Step 2.} Return \texttt{aln}.

4.4.6 CREATE-\texttt{EXPRESSION}

\textbf{Operation:} \texttt{create-expression(e, helpl)}

where \( e \) is an \texttt{expression}, \texttt{expression-seven}, \texttt{expression-six}, \texttt{expression-five}, \texttt{expression-four}, \texttt{expression-three}, \texttt{expression-two}, \texttt{expression-one}, \texttt{primitive-expression}, \texttt{prefix-expression}, or \texttt{parenthesized-expression}.

\texttt{helpl} is a \texttt{<<by-name-parts-list>}}.

\textbf{result: an \texttt{expression}.}

\textbf{Case 1.} \( e \) is an \texttt{expression}, \texttt{expression-seven}, \texttt{expression-six}, \texttt{expression-five}, \texttt{expression-four}, \texttt{expression-three}, or an \texttt{expression-two} and \( e \) has only one component, \texttt{ec}.

Perform \texttt{create-expression(ec, helpl)} to obtain an \texttt{expression}, \texttt{ae}. Return \texttt{ae}.

\textbf{Case 2.} \( e \) is an \texttt{expression}, \texttt{expression-seven}, \texttt{expression-six}, \texttt{expression-five}, \texttt{expression-four}, \texttt{expression-three}, or an \texttt{expression-one} and \( e \) has three components, \texttt{ed}, \texttt{op}, and \texttt{e2}.

\textbf{Step 2.1.} Perform \texttt{create-expression(e2, helpl)} to obtain an \texttt{expression}, \texttt{ae2}. Perform \texttt{create-expression(ed, helpl)} to obtain an \texttt{expression}, \texttt{ae}.

\textbf{Step 2.2.} If \texttt{op} is or has \( \{ +, -, \times, \div, \leq, \geq, <, \leq, =, \neq, \neq, \neq, \neq, \neq, \neq \} \) then let \texttt{sop} be \texttt{<add>, <sub>, <mul>, <div>, <le>, <le>, <op>, <op>, <gt>, <gt>, <gt>, <gt>, <gt>, <gt>, <gt>, <gt>, <gt>, <gt>, <gt>, <gt>}. \texttt{sop} is \texttt{<adj>, <subtract>, <multiply>, <divide>, or <cover> respectively.}

\textbf{Step 2.3.} Let \texttt{d1} and \texttt{d2} be the \texttt{<data-description>\textit{e}} immediately contained in \texttt{ae} and \texttt{ae2}, respectively.

The associated \texttt{<aggregate-type>s} of \texttt{d1} and \texttt{d2} must be compatible. Individual \texttt{<data-type>s} in \texttt{d1} and \texttt{d2} and corresponding \texttt{<data-type>s} in \texttt{d1} and \texttt{d2} must satisfy any constraints specified in the "Constraints" paragraphs of the section of Chapter 9 for the \texttt{<infix-operator>\textit{s}}.

Let \texttt{dd} be a \texttt{<data-description>\textit{e}} whose associated \texttt{<aggregate-type>\textit{e}} is the common \texttt{<aggregate-type>\textit{e}} of \texttt{d1} and \texttt{d2} and whose \texttt{<data-type>s} are defined as "scalar-result-types" in the "Attributes" paragraphs of the section of Chapter 9 for the \texttt{<infix-operator>\textit{s}} in terms of the corresponding \texttt{<data-type>s} in \texttt{d1} and \texttt{d2}.

\textbf{Step 2.4.} Return an.

\begin{verbatim}
<expression>:
  <infix-expression>:
    <initial-expression>:
      <initial-element>:
        <parenthesized-expression>:
          as
        ...
    ...
</infix-expression>:
  ...
</expression>:
\end{verbatim}

\textbf{Chapter 9: The Translator 115}
Case 3. e is a $\{\text{primitive-expression}\}$.

Let $ec$ be the immediate component of e.

Case 3.1. $ec$ is a $\{\text{reference}\}$.

Perform create-reference($ec$, $\langle\text{value-reference}, \text{map}\rangle$) to obtain a $\langle\text{value-reference}, \text{vr}\rangle$. Let $dd$ be the $\langle\text{data-description}\rangle$ immediately contained in vr. Return an $\langle\text{expression}\rangle$: vr $dd$.

Case 3.2. $ec$ is a $\{\text{constant}\}$.

Perform create-constant($ec$) to obtain a $\langle\text{constant}\rangle,e$. Let $dt$ be the $\langle\text{data-type}\rangle$ immediately contained in e. Return an $\langle\text{expression}\rangle$:

$$
\begin{align*}
\text{let } d & \text{ be an } \langle\text{data-description}\rangle \\
\text{let } dt & \text{ be a } \langle\text{data-type}\rangle
\end{align*}
$$

Case 3.3. $ec$ is an $\{\text{list}\}$.

$ec$ must be contained in an $\{\text{attribute}\}$ that immediately contains $\text{_DEFINED}$. Let $l$ be an $\langle\text{integer}\rangle$ whose concrete-representation is the same as that of the $\langle\text{integer}\rangle$ in $ec$. Let $dt$ be a $\langle\text{data-type}\rangle$ that is an integer-type. Return an $\langle\text{expression}\rangle$:

$$
\begin{align*}
\text{let } l & \text{ be an } \langle\text{integer}\rangle \\
\text{let } dt & \text{ be a } \langle\text{data-type}\rangle
\end{align*}
$$

Case 4. e is a $\{\text{prefix-expression}\}$: op $\langle\text{expression}\rangle$.

Step 4.1. Perform create-expression($el$, $\text{map}$) to obtain $el$.

Step 4.2. If op has $\langle\text{not}\rangle$, $\langle\text{plus}\rangle$, or $\langle\text{minus}\rangle$ respectively.

Step 4.3. Let $dd$ be the $\langle\text{data-description}\rangle$ immediately contained in $el$.

The $\langle\text{data-type}\rangle$s in $dd$ must satisfy any constraints in the "Constraints" paragraphs of the section of Chapter 9 for the $\langle\text{prefix-operator}\rangle$, $\text{op}$.

Let $dd$ be a $\langle\text{data-description}\rangle$ whose associated $\langle\text{aggregate-type}\rangle$ is the same as that of $dd$ and whose $\langle\text{data-type}\rangle$s are defined as "scalar-result-types" in the "Attributes" paragraphs of the section of Chapter 9 for the $\langle\text{prefix-operator}\rangle$, $\text{op}$ is in terms of the corresponding $\langle\text{data-type}\rangle$s in $dd$.

Step 4.4. Return as

$$
\begin{align*}
\text{let } dl & \text{ be a } \langle\text{data-description}\rangle \\
\text{let } dt & \text{ be a } \langle\text{data-type}\rangle
\end{align*}
$$

Case 5. e is a $\{\text{parenthesized-expression}\}$: ($\text{expression}$).

Step 5.1. Perform create-expression($el$, $\text{map}$) to obtain an $\langle\text{expression}\rangle,ae$. Let $dd$ be the $\langle\text{data-description}\rangle$ immediate component of ae.

Step 5.2. Return as

$$
\begin{align*}
\text{let } dl & \text{ be a } \langle\text{data-description}\rangle \\
\text{let } dt & \text{ be a } \langle\text{data-type}\rangle
\end{align*}
$$
### 4.5 Create-Reference

**Operation**: `create-reference(cr, tarq, begl)`

where `cr` is a `{reference}` or an `{unspecified-reference}`, `tarq` is a `{variable-reference}`, `{value-reference}`, `{subroutine-reference}`, or `{target-reference}`, `begl` is a `{by-name-parts-list}`.

result: a tree of the same type as `tarq`.

**Step 1.** If `cr` immediately contains an `{arguments-list}`, let `al` be a copy of `cr`. Otherwise let `al` be an `{arguments-list}`. Perform `find-applicable-declaration(cr)` to obtain a `{declaration}` `cd`.

**Case 1.1.** `cd` contains a `{generic-attribute}`.

Let `dcl` be `{abstract}`.

**Case 1.2.** `cd` declaration-contains `MEMBER`.

Let `red` be the rightmost preceding `{declaration}` that does not declaration-contain `MEMBER`. Let `dcl` be the `{declaration}` whose `{declaration-designator}` designates `red`.

**Case 1.3.** (Otherwise).

Let `dcl` be the `{declaration}` whose `{declaration-designator}` designates `cd`. It must not contain `{condition}`.

**Step 2.**

**Case 2.1.** `dcl` is a `{declaration}` that has `{variable}`.

**Step 2.1.1.** Let `den` be a `{declaration-designator}` designating `dcl`. Let `dd` be a copy of the `{data-description}` in `den`. Let `ref` be a `{variable-reference}`.

**Step 2.1.2.** Perform `find-fully-qualified-name(dd)` to obtain an `{identifier-list}`, `{il}`. Perform `create-abstract-equivalent-tree(il)` to obtain an `{identifier-list}`, `{il1}`. Delete the first `{identifier}` in `{il1}`. If `{il1}` still contains any `{identifier}`, attach `{id}` to `ref`, and for each `{identifier}`, `{id}`, in `{il1}`, taken in order, perform Steps 2.1.2.1 through 2.1.2.3.

**Step 2.1.2.1.** Let `d2` be the `{data-description}` immediately contained in `ref`. It will have a `{structure-data-description}`, `{sd}`. If `d2` has a `{dimensioned-data-description}`, let `bpl` be the `{bound-pair-list}` in `d2`; otherwise let `bpl` be a `{bound-pair-list}`. Let `i` be the first `{identifier}` in `d2`, and let `tdd` be the `{member-data-description}` in `d2`. If `i` equals `{id}`, and then let `d2` be the `{data-description}` immediately contained in the `i`-th `{member-data-description}` of `{id}`.

**Step 2.1.2.2.** If `tdd` has a `{dimensioned-data-description}`, append copies of the `{dimensioned-data-description}` in `d2` to `bpl`, and let `tdd` be the `{item-data-description}` or `{structure-data-description}` immediately contained in the `{element-data-description}` of `tdd`. Otherwise, let `tdd` be the `{item-data-description}` or `{structure-data-description}` immediately contained in `tdd`.

**Step 2.1.2.3.** If `bpl` contains `{bound-pairs}`, replace `d2` by a `{data-description}`:

```
<dimensioned-data-description>
<element-data-description>:
tdd;
bpl.
```

Otherwise replace `d2` by a `{data-description}`: `tdd`.

---

Chapter 4: The Translator 117
Step 2.1.3. If cr is a \{reference\}, then let hr be the \{basic-reference\} immediately contained in cr, and perform collect-subscript\{s\} to obtain a \{subscript-constraint\}, sl. Otherwise let sl be a \{subscript-constraint\}.

Case 2.1.3.1. The \{data-description\} immediately contained in ref does not have a \{dimensioned-data-description\}.

In this case sl must not contain any \{subscript\}s.

Case 2.1.3.2. The \{data-description\}, dd, immediate component of ref has a \{dimensioned-data-description\}.

Perform apply-subscript\{cr, ref, sl, sl\}.

Case 2.2. dcl is a \{declaration\} that has \{named-constant\}.

Perform create-named-constant-reference\{dcl\} to obtain a \{named-constant-reference\}, ref. If the \{data-description\} immediately contained in ref has a \{dimensioned-data-description\}, let sl be a \{subscript-constraint\}, and perform apply-subscript\{cr, ref, sl, sl\}.

Case 2.3. dcl is a \{declaration\} that has \{builtin\}.

Step 2.3.1.

Case 2.3.1.1. sl has an \{arguments\}.

In this case, sl must have only one \{arguments\}. Let ar be the \{arguments\} in ca. Perform create-argument-list\{ar\} to obtain args. Delete the \{arguments\} from sl.

Case 2.3.1.2. sl does not have an \{arguments\}.

Let args be \{absent\}.

Step 2.3.2.

Case 2.3.2.1. tar is a \{value-reference\}.

Perform create-builtin-function-reference\{dcl, args\} to obtain a \{builtin-function-reference\}, ref.

Case 2.3.2.2. tar is a \{target-reference\}.

Perform create-pseudo-variable-reference\{dcl, args\} to obtain a \{pseudo-variable-reference\}, ref.

Case 2.3.2.3. tar is a \{variable-reference\} or a \{subroutine-reference\}.

This case must not occur.

Case 2.4. cd has a \{generic-attribute\}, ga.

Step 2.4.1.

Case 2.4.1.1. sl has an \{arguments\}.

Let ar be the first \{arguments\} in ca, and perform create-argument-list\{ar\} to obtain args.

Case 2.4.1.2. sl does not have an \{arguments\}.

In this case tar must be a \{subroutine-reference\}. Let args be \{absent\}.

Step 2.4.2. Perform select-generic-alternative\{ga, args\} to obtain a \{value-reference\}, vr. Let ref be the first immediate component of vr.

Step 3.

Case 3.1. cr immediately contains a \{locator-qualifier\}, bq.
Let \( r \) be the \(<\text{reference}>\) immediately contained in \(\text{ln}.\) Perform \(<\text{create-reference}>\), \(<\text{value-reference}>\) to obtain a \(<\text{value-reference}>\), \(r\). All the following conditions must hold:

1. \(<\text{ref}>\) must be a \(<\text{variable-reference}>\);
2. the \(<\text{declaration}>\) designated by the \(<\text{declaration-designator}>\) immediately contained in \(r\) must have \(<\text{based}>\);
3. the \(<\text{data-description}>\) immediately contained in \(r\) must immediately contain a \(<\text{item-data-description}>\) whose \(<\text{data-type}>\) must have \(<\text{locator}>\), and if the \(<\text{data-type}>\) has an \(<\text{offset}>\) or, then \(r\) must have a \(<\text{variable-reference}>\) or a \(<\text{reference-designator}>\).

Attach a \(<\text{locator-qualifier}>\), \(v\), to \(r\).

Case 3.2. or does not immediately contain a \(<\text{locator-qualifier}>\) and \(\text{dcl}\) is a \(<\text{declaration}>\) that has \(<\text{variable}>\).

If \(\text{dcl}\) contains a \(<\text{based}>\), then \(h\) must immediately contain a \(<\text{value-reference}>\) or a \(<\text{reference-designator}>\).

Case 3.3. (Otherwise).

No action.

Step 4. If \(\text{al}\) contains an \(<\text{argument}>\), perform Steps 4.1 to 4.6.

Step 4.1. \(<\text{ref}>\) must not be a \(<\text{subroutine-reference}>\). Perform \(<\text{create-value-reference}>\) \(\text{ref}\) to obtain \(\text{evr}\). The \(<\text{data-description}>\) immediate component of \(\text{evr}\) must immediately contain an \(<\text{item-data-description}>\) whose \(<\text{data-type}>\) must have \(<\text{entry}>\).

Step 4.2. Let \(ar\) be the first \(<\text{argument}>\) in \(\text{al}\). Perform \(<\text{create-argument-list}>\) \(\text{ar}\) to obtain \(\text{args}\). Delete \(ar\) from \(\text{al}\).

Step 4.3. Perform \(<\text{create-entry-reference}>\) \(\text{evr}, \text{args}\) to obtain \(\text{ref}\).

Step 4.4. Go to Step 4.

Step 5. If \(\text{impl}\) is not \(<\text{absent}>\) and \(\text{ref}\) is a \(<\text{variable-reference}>\) then perform \(<\text{apply-by-name}>\) \(\text{impl}, \text{ref}\) to obtain a \(<\text{variable-reference}>\), \(\text{ref}\). If \(\text{impl}\) is not \(<\text{absent}>\) and \(\text{ref}\) is not a \(<\text{variable-reference}>\), then the \(<\text{data-description}>\) of \(\text{ref}\) must immediately contain \(<\text{item-data-description}>\).

Step 6.

Case 6.1. \(\text{targ}\) is a \(<\text{variable-reference}>\).

In this case \(\text{ref}\) must be a \(<\text{variable-reference}>\). Return \(\text{ref}\).

Case 6.2. \(\text{targ}\) is a \(<\text{value-reference}>\).

\(<\text{ref}>\) must not be a \(<\text{subroutine-reference}>\). Perform \(<\text{create-value-reference}>\) \(\text{ref}\) to obtain a \(<\text{value-reference}>\), \(r\). Return \(r\).

Case 6.3. \(\text{targ}\) is a \(<\text{subroutine-reference}>\).

If \(\text{ref}\) is a \(<\text{subroutine-reference}>\), return \(\text{ref}\). Otherwise perform \(<\text{create-value-reference}>\) \(\text{ref}\) to obtain a \(<\text{value-reference}>\), \(r\) whose immediately contained \(<\text{data-description}>\) must immediately contain an \(<\text{item-data-description}>\) whose \(<\text{data-type}>\) must have \(<\text{entry}>\). Perform \(<\text{create-entry-reference}>\) \(\text{ref}\) to obtain \(\text{mr}\), which must be a \(<\text{subroutine-reference}>\). Return \(\text{mr}\).

Case 6.4. \(\text{targ}\) is a \(<\text{target-reference}>\).

In this case \(\text{ref}\) must be a \(<\text{variable-reference}>\) or \(<\text{pseudo-variable-reference}>\). Let \(d\) be a copy of the \(<\text{data-description}>\) immediately contained in \(\text{ref}\). If \(\text{ref}\) is a \(<\text{variable-reference}>\), perform \(<\text{trim-didd}>\). Return a \(<\text{target-reference}>\), \(\text{ref} d\).

Chapter 4: The Translator 119
4.4.5.1 Collect-subscripts

Operation: \texttt{collect-subscripts}(br)

where \texttt{br} is a \texttt{basic-reference}.

result: a \texttt{structure-comma-list} which may have no components.

Case 1. \texttt{br} has a \texttt{structure-qualification}, \texttt{sq}.

Let \texttt{br'} be the \texttt{basic-reference} immediately contained in \texttt{sq}. Perform \texttt{collect-subscripts}(\texttt{br'}) to obtain a \texttt{structure-comma-list}, \texttt{sl}. If \texttt{sq} has an \texttt{argument-list}, \texttt{args}, then \texttt{args} must have a \texttt{structure-comma-list}, \texttt{al}, append the \texttt{structure-comma-list} in \texttt{al} to \texttt{sl}, appending \texttt{a} as required. Finally, return \texttt{sl}.

Case 2. \texttt{br} does not have a \texttt{structure-qualification}.

Return a \texttt{structure-comma-list}.

4.4.5.2 Apply-by-name-parts

Operation: \texttt{apply-by-name-parts}(vro, bap)

where \texttt{vro} is a \texttt{variable-reference},
\texttt{bap} is a \texttt{by-name-parts-list}.

result: a \texttt{variable-reference}.

Case 1. The \texttt{data-description}, \texttt{dd} immediate component of \texttt{vro} has a \texttt{structure-data-description}.

Step 1.1. Let \texttt{wr} be a copy of \texttt{vro}. Let \texttt{ddd} be a
\begin{verbatim}
<data-description>
<structure-data-description>
<member-description-list>
 slot

For each \texttt{identifier-list}, \texttt{id} component of \texttt{bap}, taken in order, perform Steps 1.1.1 and 1.1.2.

Step 1.1.1. For each \texttt{identifier}, \texttt{id} is \texttt{idl}, taken in order, perform Steps 1.1.1.1 through 1.1.1.3.

Step 1.1.1.1. \texttt{dd} will have a \texttt{structure-data-description}, \texttt{dd}. If \texttt{dd} has a \texttt{dimensioned-data-description}, let \texttt{bpl} be the \texttt{bound-pair-list} in \texttt{dd}; otherwise let \texttt{bpl} be a \texttt{bound-pair-list} with no component. Let \texttt{bi} be the \texttt{identifier-list} in \texttt{dd}, and let \texttt{adj} be the \texttt{member-description-list} in \texttt{dd}. Let \texttt{i} be the integer such that the \texttt{i}th \texttt{identifier} in \texttt{bi} equals \texttt{id}, and let \texttt{dd} be the \texttt{data-description} in the \texttt{i}th \texttt{member-description} of \texttt{adj}.

Step 1.1.1.2. If \texttt{adj} has a \texttt{dimensioned-data-description}, append copies of the \texttt{bound-pairs} in \texttt{adj} to \texttt{bpl}, and let \texttt{add} be the \texttt{item-data-description} or \texttt{structure-data-description} immediately contained in the \texttt{element-data-description} of \texttt{adj}. Otherwise, let \texttt{add} be the \texttt{item-data-description} or \texttt{structure-data-description} immediately contained in \texttt{adj}.

Step 1.1.1.3. If \texttt{bpl} contains \texttt{bound-pair} then let \texttt{dd} be a
\begin{verbatim}
<data-description>
<dimensioned-data-description>
<element-data-description>
 tdd
 bpl.

Otherwise let \texttt{dd} be a \texttt{data-description}: \texttt{tdd}.
Step 1.1.2. Append \( \dd \) to \( \text{ddis} \).

Step 1.2. Replace \( \dd \) by \( \text{ddis} \) and append step 2 to \( \text{vr} \).

Step 1.3. Return \( \text{vr} \).

Case 2. Otherwise.

Step 2.1. Return \( \text{vr} \).

### 6.3.5.3 Apply-subscripts

**Operation:** \( \text{apply-subscript} (\text{er}, \text{ref}, \text{sl}, \text{al}) \)

where \( \text{er} \) is a \( \langle \text{reference} \rangle \),

\( \text{ref} \) is a \( \langle \text{variable-reference} \rangle \) or a \( \langle \text{named-constant-reference} \rangle \),

\( \text{sl} \) is a \( \langle \text{subscript-constructor} \rangle \),

\( \text{al} \) is an \( \langle \text{arguments-list} \rangle \).

**Step 1.** Let \( \dd \) be the \( \langle \text{data-description} \rangle \) immediately contained in \( \text{ref} \). Let \( n \) be the number of \( \langle \text{bound-pair} \rangle s \) in \( \dd \), and let \( m \) be the number of \( \langle \text{subscript} \rangle s \) in \( \text{al} \) in any order. If \( n < m \), and if \( \text{al} \) has a first immediate component, \( \text{args} \), that has a \( \langle \text{subscript-constructor} \rangle \), then perform step 1.1.

**Step 1.1.** Append the \( \langle \text{subscript} \rangle s \) in \( \text{args} \) to \( \text{sl} \), appending \( \# \) as required. Delete \( \text{args} \) from \( \text{al} \).

**Step 2.** Let \( n \) be the number of \( \langle \text{subscript} \rangle s \) in \( \text{al} \) which are not in any \( \langle \text{expression} \rangle \) also in \( \text{sl} \).

**Case 2.1.** \( n = 0 \).

Attach a \( \langle \text{subscript-list} \rangle \) containing \( n \) occurrences of \( \langle \text{asterisk} \rangle \) to \( \text{ref} \).

**Case 2.2.** \( n > 0 \).

Step 2.2.1. In this case \( n \) must equal \( m \). Attach a \( \langle \text{subscript-list} \rangle \), \( \text{sl2} \) to \( \text{ref} \). For \( i = 1, \ldots, n \), perform step 2.2.1.1.

**Step 2.2.1.1.** Of those \( \langle \text{subscript} \rangle s \) contained in \( \text{cr} \) which are not contained in any \( \langle \text{expression} \rangle \) contained in \( \text{cr} \), let \( s \) be the \( i \)th one. If \( s \) immediately contains an \( \langle \text{expression} \rangle \), perform create-expressions to obtain an \( \langle \text{expression} \rangle \), \( \text{e2} \), and append a \( \langle \text{subscript} \rangle \), \( s \) \( \langle \text{expression} \rangle \), \( \text{e2} \) \( \langle \text{asterisk} \rangle \) to \( \text{sl2} \). Otherwise append a \( \langle \text{subscript} \rangle \), \( \langle \text{asterisk} \rangle \) to \( \text{sl2} \).

Step 2.2.2. For \( i = n + 1, \ldots, m \), perform step 2.2.2.1.

**Step 2.2.2.1.** If the \( i \)th \( \langle \text{subscript} \rangle \) in \( \text{sl2} \) contains an \( \langle \text{expression} \rangle \), delete the \( i \)th \( \langle \text{bound-pair} \rangle \) in \( \dd \).

Step 2.2.3. If the \( \langle \text{bound-pair-list} \rangle \) in \( \dd \) now has no \( \langle \text{bound-pair} \rangle s \), let \( \dd2 \) be the \( \langle \text{data-description} \rangle \) or \( \langle \text{structure-data-description} \rangle \) in \( \dd \), and then replace \( \dd \) by a \( \langle \text{data-description} \rangle \), \( \dd2 \).

### 6.3.5.4 Create-value-reference

**Operation:** \( \text{create-value-reference} (\text{ref}) \)

where \( \text{ref} \) is a \( \langle \text{variable-reference} \rangle \), \( \langle \text{procedure-function-reference} \rangle \), \( \langle \text{builtin-function-reference} \rangle \), or \( \langle \text{named-constant-reference} \rangle \).

**result:** a \( \langle \text{value-reference} \rangle \).

**Step 1.** Let \( \dd \) be a copy of the \( \langle \text{data-description} \rangle \) immediately contained in \( \text{ref} \).

**Step 2.** If \( \text{ref} \) is a \( \langle \text{variable-reference} \rangle \) perform trim-dd0000. Return a \( \langle \text{value-reference} \rangle \), \( \text{ref} \), \( \dd \).

---

Chapter 9: The Translator 121
8.4.3.3.1 Trim-dd

Operation: \texttt{trim-dd} of dd

where dd is a \texttt{<data-description>}

Step 1. Delete from dd any occurrences of the following categories that are not contained in an \texttt{<entry>}: \texttt{<alignment>}, \texttt{<initial>}, \texttt{<identifier-list>}, \texttt{<local>}, \texttt{<options>}, \texttt{<varying>}, \texttt{<convarying>}. Replace by an \texttt{<asterisk>} the immediate component of each \texttt{<maximum-length>}, \texttt{<area-size>} or \texttt{<bound-pair>} that is not a component of an \texttt{<entry>}.

8.4.5.1 Create-named-constant-reference

Operation: \texttt{create-named-constant-reference} of dcl

where dcl is a \texttt{<declaration>}

result: a \texttt{<named-constant-reference>}

Step 1. Let zt be a copy of the leftmost immediate component of the \texttt{<named-constant>} component of dcl. Let dt be an

\texttt{<item-data-description>:}

\texttt{<data-type>:}

\texttt{<non-computational-type>:}

zt.

Step 2.

Case 2.1. dcl contains a \texttt{<bound-pair-list>}, bp.

Let n be the number of \texttt{<bound-pair>}s in bp. Let bpl be a \texttt{<bound-pair-list> containing a subnodes \texttt{<bound-pair> \texttt{<asterisk>}}}. Let dd be a

\texttt{<dimensioned-data-description>:}

\texttt{<element-data-description>:}

\texttt{dt:}

bpl.

Case 2.2. (Otherwise).

Let dd be dt.

Step 3. Let ddg be a \texttt{<declaration-designator>} designating dcl. Return a

\texttt{<named-constant-reference>:}

ddg

\texttt{<data-description>:}

dd.
4.4.5.6 Create-argument-list

Operation: $create\text{-}argument\text{-}list\{all\}$

where $a_i$ is an $\{arguments\}$.

result: an $\{argument\text{-}list\}$ or $\{absent\}$.

Step 1. If $a_i$ does not contain a $\{subscript\text{-}commlist\}$ then return $\{absent\}$; otherwise let $scl$ be the $\{subscript\text{-}commlist\}$ immediately contained in $a_i$.

Step 2. Let $n$ be the number of $\{subscript\}$ immediate components of $scl$ that not immediately contain $*$. 

Step 3. Let $xal$ be an $\{argument\text{-}list\}$ and let $scx_i, i=1,\ldots,n$ be the $\{expression\}$-single components of $scl$ taken in left-to-right order.

Step 4. For each element, $scx_i, i=1,\ldots,n$, perform Steps 4.1 and 4.2.

Step 4.1. Perform $create\text{-}expression\{scx_i\}$ to obtain an $\{expression\}$, $scx_i$.

Step 4.2. Let $rd\bar{e}$ be the $\{data\text{-}description\}$ immediately contained in $scx_i$. Append to $xal$ an

\begin{description}
\item \textit{\{argument\}}: \textit{scx}_i \textit{rd\bar{e}}.
\end{description}

Step 5. Return $xal$.

4.4.5.7 Create-built-in-function-reference

A $built-in\text{-}function\text{-}name$ is a sequence of uppercase letters and digits such that the corresponding sequence of lowercase letters and digits followed by "$\text{bif}$" is in the category-name of a subcode of $\{built-in\text{-}function\}$.

Operation: $create\text{-}built-in\text{-}function\text{-}reference\{ad,ai\}$

where $ad$ is a $\{declaration\}$, $ai$ is an $\{argument\text{-}list\}$. 

result: a $\{built-in\text{-}function\text{-}reference\}$.

Step 1. Let $id$ be the $\{identifier\}$ contained in $ad$. There must be a $\{built-in\text{-}function\}$, $bf$, whose name or abbreviation (as listed in Section 2.3) corresponds to the concrete-representation of $id$. Let $bf$ be a $\{built-in\text{-}function\text{-}reference\}$.

Step 2. The number of $\{argument\}$s in $ai$ must be as shown in the "Arguments" section of the description of $bf$ (see Chapter 9).

Step 3.

Case 3.1. $ai$ is not $\{absent\}$.

Step 3.1.1. All $\{data\text{-}description\}$s immediately contained in the $\{argument\}$s in $ai$ must satisfy the constraints given in the "Constraints" section of the same $\{built-in\text{-}function\text{-}description\}$.

Step 3.1.2. Append $ai$ to $bf$.

Case 3.2. (Otherwise).

No action.

Step 4. Construct a $\{data\text{-}description\}$, $rd\bar{e}$, as specified in the "Attributes" section of the same $\{built-in\text{-}function\text{-}description\}$, Append $rd\bar{e}$ to $bf$.

Step 5. Return $bf$. 

Chapter 4: The Translator
3.4.5.8 Create-pseudo-variable-reference

Operation: create-pseudo-variable-reference(ad, al)

where ad is a <declaration>,
al is an <argument-list>.
result: a <pseudo-variable-reference>.

Step 1. Let id be the <identifier> contained in ad. There must be a <pseudo-variable-reference>, pv, whose name corresponds to the concrete-representation of id. Let per be a <pseudo-variable-reference>: pv.

Step 2. The number of arguments in al must be as shown in the "Arguments" section of the description of pv (see Chapter 7).

Step 3.

Case 3.1. al is not <empty>.

Step 3.1.1. All <data-description> immediately contained in the <argument-list> in al must satisfy the constraints given in the "Constraints" section of the description of pv.

Step 3.1.2. Attach al to pv.

Case 3.2. (Otherwise).

No action.


Step 5. Return pv.

3.4.5.9 Create-entry-reference

Operation: create-entry-reference(vr, al)

where vr is an <value-reference> whose <data-type> has <entry>,
al is an <argument-list>.
result: a <procedure-function-reference> or a <coroutine-reference>.

Step 1. Let dd be the <data-description> immediately contained in vr.

Step 2.

Case 2.1. dd simply contains a <parameter-descriptor-list>, pdl. al must not be <empty>. The number of elements in pdl must be equal to the number of elements in al.

Case 2.2. (Otherwise).

al must be <empty>; Go to Step 4.

Step 3. For each <argument>, arg, immediate component of al, perform Steps 3.1 through 3.3.

Step 3.1. Let pdl be the <data-description> immediate component of the <parameter-descriptor> corresponding to arg.

Step 3.2.

Case 3.2.1. arg immediately contains <expression>, <value-reference>, <variable-reference>, <variable-reference>, var. Perform test-watching(var, pdl) to obtain tv. If tv is <false>, then attach <dummy> to arg.
Case 3.3.2. (Otherwise).

Attach <dummy> to arg.

Step 3.3.

Case 3.3.1. <dummy> was attached to arg in Step 3.2.

Let rdd be the <data-description> immediately contained in arg. rdd must be proper for assignmet to pd for (see Section 3.4.2.3).

Case 3.3.2. (Otherwise).

No action.

Step 4.

Case 4.1. dd simply contains a <returns-descriptor>, rd.

Let rde be a copy of the <data-description> immediately contained in rd.

Return <procedure-function-reference> = (all rds).

Case 4.2. (Otherwise).

Return <subroutine-reference> = (all).

8.4.5.10 Text-matching

Operation: text-matching(var, pd)

where var is a <variable-reference>,

pd is a <data-description> immediate component of a <parameter-descriptor>

result: <true> or <false>

Step 1. Let dcl be the <declaration-designator> designated by the <declaration-designator> in var. If dcl contains a <defined> whose <base-item> contains an <implicit>, return <false>.

Step 2. Let dd be a copy of the <data-description> immediately contained in var. Let pd be a copy of pd.

Step 3. If any of the following substrings exist as a component of dd or pd then delete every occurrence.

<local>
<initial>
<variable-reference> as component of an <offset>
<parameter-descriptor-list>
<returns-descriptor>
<options>
<identifier-list> as a component of a <structure-data-description>

Step 4. If pd and dd are not equal, disregarding comparison of the subnodes of any <maximum-length>, <bound-pair>, or <tree-nice>, then return <false>.

Step 5. For each <extent-expression>, ei, in pd, perform Step 5.1.

Step 5.1. If there does not exist a corresponding <extent-expression>, e2, in dd, return <false>. If e2 contains a <refer-option>, return <false>. Let ei and e2 be the <expression-node> in e1 and e2, respectively. If ei or e2 contains a <declaration-designator>, return <false>. Perform evaluate-restricted-expression(ei) and evaluate-restricted-expression(e2) to obtain v1 and v2. If v1 or v2 is not a <constant> having <computational-type>, return <false>. Otherwise, perform evaluate-expression-to-integer(ei) and evaluate-expression-to-integer(e2) to obtain integer-values 11 and 12 respectively. If 11 and 12 are equal, return <true>; otherwise return <false>.

Chapter 8: The Translator 125
Step 6. Return $\langle \text{true} \rangle$.

4.5.5.11 select-generic-alternative

Operation: \( \text{select-generic-alternative}(ga, arg) \)

where \( ga \) is a \langle generic-attribute \rangle,
arg is an \langle argument-list \rangle.

result: a \langle value-reference \rangle.

Step 1.

Case 1.1. arg is \langle absent \rangle.

\( ga \) must have at least one \langle generic-element \rangle that does not have a \langle generic-description-motion-list \rangle. Let \( rr \) be the \langle reference \rangle immediately contained in the first such \langle generic-element \rangle.

Case 1.2. arg is an \langle argument-list \rangle.

Step 1.2.1. Let \( na \) be the number of \langle argument \rangle\'s in arg. There must be at least one \langle generic-element \rangle in \( ga \). Let \( ge \) be the first such \langle generic-element \rangle.

Step 1.2.2. Let \( ngd \) be the number of \langle generic-description \rangle\'s in \( ge \) that do not declaration-contain MEMBER. If \( ngd \) does not equal \( na \), go to Step 1.2.5.

Step 1.2.3. For \( i = 1, \ldots, na \), perform Steps 1.2.3.1 and 1.2.3.2.

Step 1.2.3.1. Let \( ai \) be the \( i \)'th \langle argument \rangle in arg.

Let \( qd \) be the \( i \)'th \langle generic-description \rangle in \( ge \) that does not declaration-contain MEMBER and let \( gdl \) be \langle generic-description-motion-list \rangle \langle generic-description \rangle.

Case 1.2.3.1.1. There is a \langle generic-description \rangle in \( ge \) that follows \( qd \) and does not declaration-contain MEMBER.

Let \( qdf \) be the leftmost such \langle generic-description \rangle. Append to \( qdl \), in left-to-right order, copies of all \langle generic-description \rangle\'s in \( ge \) between \( qd \) and \( qdf \).

Case 1.2.3.1.2. (Otherwise).

Append to \( qdl \) in left-to-right order, copies of all \langle generic-description \rangle\'s in \( ge \) following \( qd \).

Step 1.2.3.2. Perform test-generic-matching\((ai, gdl)\) to obtain \( tvai \). If \( tvai \) is \langle false \rangle go to Step 1.2.5.

Step 1.2.4. Let \( rr \) be the \langle reference \rangle immediately contained in \( ge \). Go to Step 2.

Step 1.2.5. Let \( ge \) be the next \langle generic-element \rangle of \( ga \) following the current \( ge \). There must be such a \langle generic-element \rangle. Go to Step 1.2.2.

Step 2. Perform find-applicable-declaration\(\langle irr \rangle\) to obtain a \langle declaration \rangle \langle cli \rangle, which must not contain a \langle generic-attribute \rangle.

Step 3. Perform create-reference\(\langle irr, \langle value-reference \rangle \rangle\) to obtain \( vr \). Return \( vr \).
4.4.5.12 Test-generc-matching

Operation: test-generc-matching(a, p,l)

where a, is an <argument>,

p, is a <generic-description-list>.

result: <true> or <false>.

Case 1. p, contains only <generic-description>.

Return <true>.

Case 2. (Otherwise).

Step 2.1. If a, is of the form

<argument>:

<expression>:

<value-reference>:

<variable-reference>, . . .:

then let d, be the <data-description> immediately contained in vr. Otherwise, let d, be the <data-description> immediately contained in a,.

Step 2.2. Perform test-generc-aggregation(p,l,d,) to obtain tvl. Return tvl.

4.4.5.13 Test-generc-aggregation

Operation: test-generc-aggregation(p,l,d,)

where p, is a <generic-description-list>,

d, is a <data-description>.

result: <true> or <false>.

Step 1. Let q, be the first <generic-description> component of p,. If q, declaration-

contains DIMENSION then perform steps 1.1 through 1.3.

Step 1.1. If d, does not immediately contain a <dimensioned-data-description> then

return <false>.

Step 1.2. If the number of * components of the <astern-konnlist> of q, is not equal to the

number of <bound-pair> components of the <bound-pair-list> of d,, then

return <false>.

Step 1.3. Delete the DIMENSION <astern-konnlist> declaration-contained in q,.

Step 2. If d, immediately contains a <dimensioned-data-description> then let d, be a

<data-description>: t, where t, is the immediate component of the <element-

data-description> of d,.

Step 3.

Case 3.1. q, declaration-contains STRUCTURE.

Step 3.1.1. Let lv be the value of the <level> of q,.

Let ni be the number of <level> components of q, with value equal to lv+1 which follow q, without any intervening component of q, whose <level> has value less than or equal to lv.

Step 3.1.2. If d, does not immediately contain a <structure-data-description>, then return <false>. Let ni be the number of <data-description> simple components of d,. If ni does not equal ni then return <false>.
Step 3.1.1. For i = 1, ..., n, perform Steps 3.1.1.1 to 3.1.1.4.

Step 3.1.1.1. Let gdi be a copy of gd. Delete from gdi all the \{generic-description\}s associated with the i'th \{level\} that precedes the \{sequence\} whose \{level\} has value i\#.1.

Step 3.1.1.2. Let \( d_1 \) be the i'th \{data-description\} of \( d_d \).

Step 3.1.1.3. Perform test-generic-aggregation(gdi, tval) to obtain tv.

Step 3.1.1.4. If tv is \{false\} then return \{false\}.

Case 3.2. (Otherwise).

Step 3.2.1. Delete any \{level\} or \{member\} components of gd.

Step 3.2.2. Perform test-generic-description(gd, dd) to obtain tv.

Step 4. Return tv.

4.4.5.14 Test-generic-description

Operation: test-generic-description(gd, dd)

where gd is a \{generic-description\},
\( dd \) is a \{data-description\}.

result: \{true\} or \{false\}.

Step 1.

Case 1.1. \( dd \) immediately contains \{structure-data-description\}.

Return \{false\}.

Case 1.2. \( dd \) immediately contains \{dimensioned-data-description\}.

Return \{false\}.

Case 1.3. \( dd \) immediately contains \{item-data-description\}.

No action.

Step 2. If \( dd \) does not have \{alignment\} then delete any ALIGNED or UNALIGNED declaration-contained in \( gd \).

Step 3. If \( dd \) has neither \{varying\} nor \{constantly\} then delete any VARYING or constant declaration-contained in \( gd \).

Step 4. Let gdal be the \{generic-data-attribute-list\} in gd. For each \{generic-data-attribute\}, gda in gdal whose first immediate component \( atr \) appears in Table 4.1., perform Steps 4.1 to 4.4.

Step 4.1. Perform create-abstract-equivalent-tree(\( atr \)) to obtain abstr.

Step 4.2. If \( dd \) does not simply contain a node whose node type is the same as abstr then return \{false\}.

Step 4.3. If \( atr \) is neither \{entry\} nor \{precision\} then delete gda from gdal.

Step 4.4. If \( atr \) is \{entry\} or \{precision\} and is the sole component of gda, then delete gda from gdal.
Step 5. If gdel contains a \{generic-data-attribute\}, gda which immediately contains a 
\{generic-precision\}, gpec then perform Steps 5.1 to 5.3.

Step 5.1. If dd has \{picture\} or if dd has \{picture\}, then return 
\{false\}.

Step 5.2. Let \texttt{prec} be the \{precision\} of dd. Perform test-generic-
precision(\texttt{prec}, prec) to obtain \texttt{wcl}.

Step 5.3. If \texttt{wcl} is \{false\} then return \{false\}. Otherwise delete gda from gdel.

Step 6. If gdel contains a \{generic-data-attribute\}, gda which immediately contains a 
\{description-comment\}, then perform Steps 6.1 to 6.4.

Step 6.1. If dd does not have a \{parameter-descriptor-list\} then return \{false\}.

Step 6.2. Let \texttt{pdl} be the \{parameter-descriptor-list\} simply contained in dd.

Step 6.3. gdl must not contain any \{identifier\} components. Perform create-data-
description(\texttt{pdl}) to obtain a \{data-description\}, gdd. Perform replace-concrete-designator(\texttt{gdl}). Let \texttt{gd} be the \{parameter-descriptor-list\} in gdd. For each \{parameter-descriptor\}, gpd in gpd, perform validate-
descriptor(\texttt{gdp}).

Step 6.4. If \texttt{gd} is not equal to \texttt{pdl} then return \{false\}. Otherwise delete gda from gdel.

Step 7. If gdel contains a \{generic-data-attribute\}, gda which immediately contains a 
\{returns-descriptor\} then perform Steps 7.1 to 7.4.

Step 7.1. If dd does not have a \{returns-descriptor\} then return \{false\}.

Step 7.2. Let \texttt{rd} be the \{returns-descriptor\} simply contained in dd.

Step 7.3. gdl must not contain any \{identifier\} components. Perform create-data-
description(\texttt{gdl}) to obtain a \{data-description\}, gdd. Perform replace-concrete-designator(\texttt{gdd}). Let \texttt{gd} be the \{returns-descriptor\} in gdd. Perform validate-descriptor(\texttt{gd}).

Step 7.4. If \texttt{gd} is not equal to \texttt{rd} then return \{false\}. Otherwise delete gda from gdel.

Step 8. If gdel contains a \{generic-data-attribute\}, gda which immediately contains 
\{picture\}, p then perform Steps 8.1 to 8.4.

Step 8.1. If dd does not have a \{picture\} then return \{false\}.

Step 8.2. Let \texttt{pd} be the \{picture\} component of dd.

Step 8.3. Perform create-picture(\texttt{pd}) to obtain a \{picture\}, \texttt{sp}.

Step 8.4. If \texttt{sp} is not equal to \texttt{pd} then return \{false\}; otherwise delete gda from gdel.

Step 9. All codes must have been deleted from gdel. Return \{true\}.

Table 5.1. Concrete Terminals of Significance to Test-generic-description:

<table>
<thead>
<tr>
<th>ALIGNED</th>
<th>COMPLEX</th>
<th>FLOAT</th>
<th>POINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>AREA</td>
<td>DECIMAL</td>
<td>FORMAT</td>
<td>PRECISION</td>
</tr>
<tr>
<td>BINARY</td>
<td>ENTRY</td>
<td>LABEL</td>
<td>REAL</td>
</tr>
<tr>
<td>BYTE</td>
<td>FILE</td>
<td>SIGNED</td>
<td>SIGNED</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>FIXED</td>
<td>OFFSET</td>
<td>VARYING</td>
</tr>
</tbody>
</table>

Chapter 4: The Translator 129
### 4.4.5.15 Test-generic-precision

**Operation:** \texttt{test-generic-precision}(\texttt{gprec}, \texttt{prec})

where \texttt{gprec} is a \texttt{(generic-precision)},
\texttt{prec} is a \texttt{(precision)}.
result: \texttt{true} or \texttt{false}.

**Step 1.**

Case 1.1. \texttt{gprec} contains \texttt{(number-of-digits)} \texttt{\$i\$} \texttt{(number-of-digits)}.

Let \texttt{qpio} be the value of the \texttt{(integer)} component of the first \texttt{(number-of-digits)} of \texttt{gprec} and let \texttt{qphi} be the value of the \texttt{(integer)} component of the second \texttt{(number-of-digits)} component of \texttt{gprec}.

Case 1.2. (Otherwise).

Let \texttt{qpio} and \texttt{qphi} both be the value of the \texttt{(integer)} component of the sole \texttt{(number-of-digits)} component of \texttt{gprec}.

**Step 2.** Let \texttt{p} be the value of the \texttt{(integer)} component of the \texttt{(number-of-digits)} of \texttt{prec}.

If it is not the case that \texttt{qpio} ≤ \texttt{p} ≤ \texttt{qphi} then return \texttt{false}.

**Step 3.**

Case 3.1. \texttt{gprec} does not have a \texttt{(scale-factor)} and \texttt{prec} does not have a \texttt{(scale-factor)}.

Return \texttt{true}.

Case 3.2. \texttt{gprec} has a \texttt{(scale-factor)} and \texttt{prec} does not have a \texttt{(scale-factor)}.

Return \texttt{false}.

Case 3.3. \texttt{gprec} has \texttt{(scale-factor)} \texttt{\$i\$} \texttt{(scale-factor)}.

Let \texttt{qpio} be the value of the \texttt{(integer)} component of the first \texttt{(scale-factor)} of \texttt{gprec} and let \texttt{qphi} be the value of the \texttt{(integer)} component of the second \texttt{(scale-factor)} of \texttt{gprec}.

Case 3.4. (Otherwise).

Let \texttt{qpio} and \texttt{qphi} both be the value of the \texttt{(integer)} component of the sole \texttt{(scale-factor)} component of \texttt{gprec}.

**Step 4.** Let \texttt{s} be the value of the \texttt{(signed-integer)} component of the \texttt{(scale-factor)} of \texttt{prec}.

**Step 5.** If it is the case that \texttt{qpio} ≤ \texttt{s} ≤ \texttt{qphi} then return \texttt{true}; otherwise return \texttt{false}.
8.4.6 CREATE-PICTURE

A (picture) may occur as a component of a {declaration} or a {format-item}. In both cases it is translated to a <picture> element. Elements of a (picture) may be repeated by the specification of a {repetition-factor} which is expanded into a sequence of elements first. Then the (picture) is translated to a <picture-character> or <picture-numeric>.

The content of a (picture) is governed by the following syntax:

{picture-content} ::= {picture-item-list} {picture-scale-factor}
{picture-item} ::= {repetition-factor} {picture-element}
{repetition-factor} ::= {integer}
{picture-scale-factor} ::= F {signed-integer}

Operation: create-picture(p)

where p is a (picture).

result: a <picture>.

Step 1. There must be a {string-or-picture-symbol-list}.aspal in p. Let t be a (picture-content) whose concrete-representation is the same as that of aspal. The tree t must exist and be unique.

Step 2. For each component of t which is a

{picture-item}.c;
{repetition-factor}.rf
{picture-element}.pe;

let lv be the decimal value of the {integer} in rf. lv must not be equal to zero. Replace c by lv occurrences of a {picture-item}. pe.

Step 3.

Case 3.1. t contains a {picture-element}: A; or a {picture-element}: X.

if p is a component of a {declaration}.d, a {description}.d, or a {generic-description}.d then d must not declaration-contains REAL or COMPLEX. All terminal nodes of t must be A, X, or D. Return <picture>: <picture-character>: <character-picture-element-list>.opel; where the concrete-representation of opel is the same as that of t.

Case 3.2. (Otherwise),

Perform create-numeric-picture(t) to obtain a <picture-numeric>.pn. If p is a component of a {declaration}, {description}, or {generic-description} which declaration-contains COMPLEX, then replace the <real> component of pn by <complex>. Return <picture>: pn.
S.4.6.1 Create-numeric-picture

The picture-validation syntax is as follows:

\{numeric-picture-specification\} := \{fixed-point-picture\} | \{floating-point-picture\}
\{fixed-point-picture\} := \{non-drifting-field\} | \{drifting-field\}
\{non-drifting-field\} := \{digits\} + \{sign\} + \{\$\} | \{digits\} + \{\$\} | \{credit\} | \{debit\}
\{digits\} := \{pic-digit-list\} V \{pic-digit-list\}
\{pic-digit\} := \$ | \# | R | T | Y
\{sign\} := + | - | \$ | \$
\{drifting-field\} := \{drifting-sign-field\} + \{\$\} | \{drifting-dollar-field\} + \{\$\} | \{drifting-dollar-field\} \{credit\} | \{debit\}
\{credit\} := CR
\{debit\} := DS
\{drifting-sign-field\} := \{sign\} \{scaled-digits-field\}
\{scaled-digits-field\} := \{\$\}-list \{\$\}-list | \{\$\}-list \{\$\}-list
\{plus\} := +
\{minus\} := -
\{digits\} := \{\$\}-list | \{plus\} \{plus-list\} | \{minus\} \{minus-list\}
\{drifting-dollar-field\} := \{\$\}-list \{scaled-digits-field\} | \{\$\}-list \{\$\}-list
\{floating-point-picture\} := \{pic-mantissa\} \{pic-exponent\}
\{pic-mantissa\} := \{sign\} \{digits\} | \{drifting-sign-field\}
\{pic-exponent\} := \{\$\}-list \{\{sign\}\} \{pic-digit-list\} | \{\$\}-list \{\$\}-list \{pic-digit-list\} | \{\$\}-list \{\$\}-list

A digit-position is any occurrence of a \{pic-digit\} or \# or \$, or any occurrence of $, +, -, or $ in an \{\$\}-list, \{plus-list\}, \{minus-list\}, or \{\$\}-list.
Operation: \(\text{create-numeric-picture}\)\(p\)

where \(p\) is of the form \(\langle\text{picture-content}\rangle,\langle\text{picture-item-list}\rangle,pil,\langle\text{picture-scale-factor}\rangle,pil\).

result: a \(\langle\text{picture-numeric}\rangle\).

Step 1. Let pil be a copy of pil. Delete from pilw any \(\langle\text{picture-item}\rangle\) which contains a \(\langle\text{picture-element}\rangle\) containing a

\(\{\text{,} \}\), or

\(\{\text{,} \}\), or

\(\{\text{,} \}\), or

\(\{\text{,} \}\), or

\(\{\text{,} \}\) if that \(\langle\text{picture-item}\rangle\) does not immediately follow a \(\langle\text{picture-item}\rangle\):

\(\langle\text{picture-element}\rangle\) \(p\).

A terminal node so deleted must not have occurred immediately between a \(C\) and an \(R\) nor immediately between a \(D\) and a \(R\).

Step 2. It must be possible to construct a \(\langle\text{numeric-picture-specification}\rangle,nps\) according to the picture-validation syntax (above), such that the concrete-representation of nps is the same as that of pilw.

If \(nps\) exists, then \(nps\) must contain a \(\langle\text{fixed-point-picture}\rangle\).

Step 3. Let \(p\) be a partial tree

\(\langle\text{picture-numeric}\rangle:\)

\(\langle\text{numeric-picture-specification}\rangle,nps\)

\(\langle\text{arithmetic}\rangle:\)

\(\langle\text{node}\rangle:\)

\(\langle\text{real}\rangle:\)

\(\langle\text{base}\rangle:\)

\(\langle\text{decimal}\rangle:\)

\(\langle\text{scale}\rangle:\)

\(\langle\text{precision}\rangle,\text{prec}\)

\(\langle\text{number-of-digits}\rangle:n\)

where \(n\) is the number of digit-positions in \(\langle\text{fixed-point-picture}\rangle\) or in \(\langle\text{numeric-picture}\rangle\), in \(nps\). \(n\) must not be greater than the maximum \(\langle\text{number-of-digits}\rangle\) for \(\langle\text{base}\rangle\), \(\langle\text{decimal}\rangle\), and \(\langle\text{scale}\rangle\) to be set below.

Note: The node \(\langle\text{real}\rangle\) may be replaced by \(\langle\text{complex}\rangle\) at a later stage.

Step 4. If \(nps\) contains a \(\langle\text{floating-point-picture}\rangle\) then attach \(\langle\text{fluct}\rangle\) to \(p\); otherwise attach \(\langle\text{fixed}\rangle\) to \(p\).

Step 5.

Case 5.1. \(nps\) contains a \(\langle\text{fixed-point-picture}\rangle\).

Step 5.1.1. \(nps\) must not contain more than one \(\langle\text{pic-digit}\rangle\) which has a \(T\), \(I\), or \(R\).

\(nps\) must not contain a \(\langle\text{pic-digit}\rangle\) which has a \(T\), \(I\), or \(R\) if it also contains \(E\), \(\times\), \(\pm\), \(\langle\text{credit}\rangle\), or \(\langle\text{debit}\rangle\). Attach to \(nps\) a \(\langle\text{fixed-point-picture}\rangle\) with the same concrete-representation as pilw.

Step 5.1.2. If \(nps\) exists, then let \(V\) be the result of interpreting the \(\langle\text{signed-integer}\rangle,n\), in \(nps\) as a decimal constant, and attach to \(nps\) a \(\langle\text{picture-scale-factor}\rangle\) containing the \(\langle\text{signed-integer}\rangle\) abstract-equivalent of \(n\); otherwise, let \(V\) be \(0\). Let \(\text{prec} = n - \text{scale}\) where \(n\) is the number of digit-positions after the \(V\) in \(nps\) if \(V\) appears, and \(\text{scale}\) otherwise. Attach a \(\langle\text{scale-factor}\rangle\) \(V\) \(\text{prec}\) to \(nps\).

Case 5.2. \(nps\) contains a \(\langle\text{floating-point-picture}\rangle\).

Step 5.2.1. Let \(p\) be a \(\langle\text{numeric-picture-element-list}\rangle\) whose concrete-representation is the same as that of pil up to (but not including) the \(E\) or \(R\). \(p\) must not contain any of \(Y\), \(I\), or \(R\).

Step 5.2.2. Let \(p\) be a \(\langle\text{numeric-picture-element-list}\rangle\) whose concrete-representation is the same as that of pil beginning with the \(E\) or \(R\). \(p\) must not contain any of \(Y\), \(I\), or \(R\).
Step 5.2.3. Attach to each a

  <floating-point-picture>:
  <picture-sign筵yue>:
    pm;
  <picture-exponent>:
    pe.

Step 6. Return ps.

9.4.7 CREATE-CONSTANT

Operation: create-constant(c)

  where c is a <constant>.

  result: a <constant>.

Case 1. c contains a <simple-character-string-constant>.

Step 1.1.

Case 1.1.1. c contains no <string-or-picture-symbol-list>.

  Let csv be a <character-string-value>: <null-character-string>.

  Case 1.1.2. (Otherwise).

    Step 1.1.2.1. Let csv be a <character-string-value> whose symbols (in order)
      have the same concrete-representations as the <string-or-picture-
      symbol-list> in c, except that each <string-or-picture-symbol>: x';
      in c becomes a <symbol>: x'; in csv.

    Step 1.1.2.2. If c contains a <replicated-string-constant>, rec, perform Step
      1.1.2.2.1.

    Step 1.1.2.2.1. Let j be the value obtained by interpreting the <integer> in
      rec as a decimal constant; let i be the number of
      <character-values> in csv.

      If j=0, let v be a <null-character-string>; otherwise, let v
      be a <character-value-list> with i* components, the
      (m+1)*th component equaling the k*th <character-value> of
      csv, for k=1,...,i, and m=0,...,(j-1).

      Replace the <character-value-list> in csv by v.

Step 1.2. Return a <constant>: <basic-value>: csv; dt; where dt is a <data-type>
  containing <character>, <not-warring>, and <maximum-length>: <asterish>.

Case 2. c contains a <simple-bit-string-constant>.

Step 2.1.

Case 2.1.1. c contains no <string-or-picture-symbol-list>.

  Let bsv be a <bit-string-value>: <null-bit-string>.

Case 2.1.2. (Otherwise).

    Step 2.1.2.1. Let m be 1,2,3,4 according to whether <radix-factor> in c has B1,
      B2, B3, or B4. Let still, i=1,...,k, be the <string-or-picture-
      symbol-list> in c.

      Each still must have an entry in Table 4.7 which is valid for the
      value of m. Let bsv be a <bit-string-value> containing m+k <bit-
      values>, such that <bit-values> (i=m+1-n) through (i=m) are
      obtained from Table 4.2 as a function of m and still, i=1,...,k.
Step 2.1.2.2. If \( c \) contains a \{replicated-string-constant\}, \( rsc \), perform step 2.1.2.2.1.

Step 2.1.2.2.1. Let \( j \) be the value obtained by interpreting \{integer\} in \( rsc \) as a decimal constant; let \( i \) be the number of \{bit-values\} in \( bsv \).

If \( j=0 \), let \( v \) be a \{null-bit-string\}; otherwise, let \( v \) be a \{bit-value-list\} with \( j \) components, the \( i \)th component equaling the \( k \)th \{bit-value\} of \( bsv \), for \( k=1, \ldots, j \), and \( n=0, \ldots, (j-1) \).

Replace the \{bit-value-list\} in \( bsv \) by \( v \).

Step 2.2. Return a \{constant\}: \{basic-value\}: \( bsv \): \( dt \): where \( dt \) is a \{data-type\} containing \( \text{bit} \), \{constvar\}, and \{maximum-length\}: \{variable\}.

Case 3. \( c \) contains an \{arithmetic-constant\}, \( ac \).

Step 3.1. Perform evaluate-real-constant\( (rc) \), where \( rc \) is the \{real-constant\} in \( ac \), to obtain a \{value-and-type\}: \( \text{real-value} \)\( v \): \( \text{data-type} \)\( .t \).

Step 3.2. Let \( ds \) be a partial \{declaration\} containing:

\[
\text{CONSTANT}
\]
\[
\text{if } c \text{ contains } P, \text{ then FIXED}
\]
\[
\text{if } c \text{ contains } X, \text{ then FLOAT}
\]
\[
\text{if } c \text{ contains } I, \text{ then COMPLEX}
\]
\[
\text{otherwise, REAL.}
\]

(\( ds \) is partial in that it contains no \{identifier\}).

Step 3.3. If \( c \) does not contain \( P \), then attach a partial \{unit\}, \( u \), \{declaration\}: \( \text{declaration} \)\( .d \) to the \{procedure\} or \{begin-block\} which \{block\} contains \( c \), attach a copy, \( cdu \), of \( ds \) to \( ds \), perform apply-definition\( (cdu) \), let \( ds \) be a copy of \( cdus \), and delete \( u \).

Step 3.4. Let \( dt \) be a partial \{data-type\}: \{computational-type\}: \{arithmetic\}:; and complete it as follows:

Step 3.4.1. For each of the following which is contained in \( ds \), append the abstract-equivalent to \( dt \): \{binary\}, \{decimal\}, \{fixed\}, \{float\}, \{real\}, \{complex\}.

If \( dt \) is still without \{base\} or \{scale\} (or both), copy the \{base\} or \{scale\} (or both) from \( t \).

Step 3.4.2. Let \( cp \) be the converted \{precision\} of \( t \) for the \{base\} and \{scale\} of \( dt \).

Step 3.4.3.

Case 3.4.3.1. \( ds \) contains a \{precision\}: \( p \).

Perform create-abstract-equivalent\( (\text{tree}(p)) \) to obtain a \{precision\}: \( ap \).

If \( dt \) contains \{float\}, \( ap \) must not contain a \{scale-factor\}.

If \( dt \) contains \{fixed\} and \( ap \) contains no \{scale-factor\}, attach \{scale-factor\}: \( 0 \) to \( ap \).

If \( dt \) contains \{fixed\}, the amount by which the \{number-of-digits\} exceeds the \{scale-factor\} in \( ap \) must not be less than that for \( cp \).

Attach \( ap \) to \( dt \).

Case 3.4.3.2. (Otherwise).

Attach \( cp \) to \( dt \).
Step 3.3.

Case 3.3.1. ac immediately contains a \{real-constant\}.

Perform convert(dt, rv) to obtain a \{real-value\}, rv. Let bv be a
\{basic-value\} rv.

Case 3.3.2. ac contains an \{imaginary-constant\}.

Let rd be a \{data-type\} which has \{real\} but in otherwise as dt.
Perform convert(rd, rv) to obtain a \{real-value\}, rv. Let bv be a
\{basic-value\} \{complex-value\} with real part: 0; and imaginary part:
rv.

Step 3.4. Return a \{constant\} bv rd.

Table 8.2. Table of \{bit-values\} as a Function of \{symbol\}s and \{radix-factor\}s for
Create-constant.

<table>
<thead>
<tr>
<th>Contents of l'th {symbol}</th>
<th>Contents of {bit-values} (l<em>m+1) through (l</em>m)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>m=1   m=2   m=3   m=4</td>
</tr>
<tr>
<td>0</td>
<td>000   000   0000</td>
</tr>
<tr>
<td>1</td>
<td>01    01    0010</td>
</tr>
<tr>
<td>2</td>
<td>10    011   0010</td>
</tr>
<tr>
<td>3</td>
<td>11    100   0101</td>
</tr>
<tr>
<td>4</td>
<td>10    110   0010</td>
</tr>
<tr>
<td>5</td>
<td>10    111   0101</td>
</tr>
<tr>
<td>6</td>
<td>10    111   0010</td>
</tr>
<tr>
<td>7</td>
<td>11    101   0101</td>
</tr>
<tr>
<td>8</td>
<td>11    101   0010</td>
</tr>
<tr>
<td>9</td>
<td>10    110   0101</td>
</tr>
<tr>
<td>A</td>
<td>10    110   0010</td>
</tr>
<tr>
<td>B</td>
<td>10    110   0101</td>
</tr>
<tr>
<td>C</td>
<td>10    110   0010</td>
</tr>
<tr>
<td>D</td>
<td>10    110   0101</td>
</tr>
<tr>
<td>E</td>
<td>10    110   0010</td>
</tr>
<tr>
<td>F</td>
<td>10    110   0101</td>
</tr>
<tr>
<td>Other</td>
<td></td>
</tr>
</tbody>
</table>

- indicates that the corresponding \{symbol\} or
\{string-or-picture-symbol\} in invalid for this
value of m
0 indicates \{zero-bit\}
1 indicates \{one-bit\}
4.8 Validation of the Abstract Procedure

An abstract <procedure> has been constructed so far by the Translator, corresponding to the specified concrete <procedure>. This abstract <procedure> is now examined by some final tests before being returned as the result of the translate operation. These tests validate each declaration and apply constraints.

Operation: 
\[ \text{validate-procedure}(ap) \]
where \( ap \) is a <procedure>.

Step 1. Perform \text{apply-constraints}(ap).

Step 2. For each <declaration>, \( ad \), component of \( ap \) perform \text{validate-declaration}(ad).

Step 3. For each <parameter-descriptor>, \( d \), and for each <returns-descriptor>, \( d \), contained in \( ap \) perform \text{validate-descriptor}(d).

Step 4. \( ap \) must satisfy all the Constraints appearing under the heading "Constraint:" in the Abstract Syntax.

Step 5. Each <do-spec> must satisfy the Constraints specified in Section 4.3.8.

4.5.1 VALIDATE-DECLARATION

When all the <declaration>s have been completed, and their contained <expression>s and <value-reference>s properly completed, some additional validation is required to ensure that each <declaration> do represent realistic data entities.

Operation: 
\[ \text{validate-declaration}(ad) \]
where \( ad \) is a <declaration>.

Step 1. If \( ad \) contains \(<\text{automatic}>, <\text{boxed}>, <\text{controlled}>, <\text{defined}>, <\text{parameter}>, \) or \(<\text{static}>\) then perform \text{validate-automatic-declaration}(ad), \text{validate-boxed-declaration}(ad), \text{validate-controlled-declaration}(ad), \text{validate-defined-declaration}(ad), \text{validate-parameter-declaration}(ad), or \text{validate-static-declaration}(ad), respectively.

Step 2. If \( ad \) contains \(<\text{name}>, <\text{constant}>\), then perform \text{validate-static-declaration}(ad).

Step 3. If \( ad \) contains any <data-descriptor>, \( dd \), which simply contains <initial>, then for each such \( dd \) perform Step 3.1.

Step 3.1. The <data-descriptor> immediate component of each <parenthesized-expression> immediate component of an <initial-element> contained in \( dd \) must be proper for assignment to \( dd \).

4.5.2 VALIDATE-AUTOMATIC-DECLARATION

Automatic declarations must satisfy constraints which enable them to be allocated and possibly initialized at the time that the block to which they belong is being activated.

Operation: 
\[ \text{validate-automatic-declaration}(ad) \]
where \( ad \) is a <declaration>.

Step 1. \( ad \) must not contain an <area-size>, \(<\text{asterisk}>\); a <maximum-length>, \(<\text{asterisk}>\); or a <bound-pair>, \(<\text{asterisk}>\); except as subnodes of <entry>.

Step 2. Each <content-expression> in \( ad \) must not contain a <refer-option>.

Step 3. If \( ad \) contains a <variable-reference> \(<\text{declaration-designator}>, <\text{dd}>\), then \( dd \) must not designate a <declaration>, \( d \), containing <automatic> or <defined> if \( d \) is a block-component of the same block as \( ad \).
4.5.3 VALIDATE-BASED-DECLARATION

Based declarations may contain <structure-data-description>s, some of whose <member-description>es refer to other (previous) members of the same structure by means of the <refer-option>.

Operation: validate-based-Declaration(bd)

where bd is a <Declaration>.

Step 1. Each <area-size>, <maximum-length>, or <bound-pair> in bd must not contain <asterisk>, except as subnodes of <entry>.

Step 2. For each <refer-option> ro in bd, perform steps 2.1 to 2.4.

Step 2.1. ro must be simply contained in a <member-description>.

Step 2.2. Let id1[i],...,idn[i] be the components of the <identifier-list> in ro. id1[i] must be equal to the <identifier> immediately contained in bd; bd must simply contain a <structure-data-description>md1[i]; id1[i] md1[i]. For i=1,...,(n-1), id1[i] must have an <identifier>id1[i] equal to id1[i]; for i=n,...,(n-2), the <member-description> in md1[i] corresponding to id1[i] in id1[i] must simply contain a <structure-data-description>md1[i]; id1[i] md1[i]; the <member-description> in md1[i] corresponding to id1[i] in id1[i] must immediately contain a <data-description> <item-data-description> <data-type> <computational-type>; and md1[i] must not be contained in a <dimensioned-data-description>.

Step 2.3. ro must occur to the left of me in bd.

Step 2.4. For every <structure-data-description>md other than md1[i] which contains both me and ro perform step 2.4.1.

Step 2.4.1. For every <item-data-description>,md1[i] which is contained in md and is to the right of me, there must exist at least one <item-data-description>,md2[i] contained in md, which either is simply contained in me or is to the left of me, such that id1[i] and id2[i] match as defined in step 2.4.1.1.

Step 2.4.1.1. Let id1'[i] and id2'[i] be copies of id1[i] and id2[i], modified as follows. Delete any occurrences of <initial>, <maximum-length>, <number-of-digit>, and <local>; delete any subnodes of <offset>, <entry>, and <area>; replace any occurrences of <punctuated> by <string>; <string-type> <character> <puncturing>.

For id1[i] and id2[i] to match, id1'[i] and id2'[i] must be equal, and if id1[i] and id2[i] have <arithmetic>, the <number-of-digits> of id1[i] must be less than or equal to the <number-of-digits> of id2[i].

4.5.4 VALIDATE-CONTROLLED-DECLARATION

Operation: validate-controlled-Declaration(cd)

where cd is a <Declaration>.

Step 1. cd must not contain a <area-size> <asterisk>, a <maximum-length> <asterisk>, or a <bound-pair> <asterisk>, except as subnodes of <entry>.

Step 2. Each <extent-expression> in cd must not contain a <refer-option>.

Step 3. If cd has <external>, then for each <expression>, a simple component of an <extent-expression> of cd, perform step 3.1.

Step 3.1. Perform evaluate-restricted-expression(c) to obtain c. If c is a <constant> having Computational-type, then replace the first immediate component of c by e. (This is preparatory to consistency checking of constant <extent-expression>s in validate-external-declaration. If c is <fail>, this indicates that e is not a restricted-expression, and e remains unchanged.)
4.5.5 VALIDATE-DEFINED-DECLARATION

Operation:  
\texttt{validate-defined-declaration} \( d \) 
where \( d \) is a \texttt{<declaration>}. 

Step 1. Perform \texttt{validate-automatic-declaration} \( d \). 

Step 2. Let \( d \) contain the form \texttt{<defined>}. \texttt{<def>}. \texttt{<base-item>}. \texttt{<variable-reference>}, \texttt{<def>}. IF \( w_r \) contains an \texttt{<item>} then \texttt{<def>} must not contain a \texttt{<position>} and \( \texttt{<def>} \) must not contain a \texttt{<material>}. 

Step 3. Each \texttt{<declaration-designator>} in \( d \) must not designate \( d \). 

Step 4. The \texttt{<data-description>} immediately contained in \( w_r \) and the \texttt{<data-description>} immediately contained in the \texttt{<variables>} in \( d \) must not contain \texttt{<varying>}, other than as a subcode of \texttt{<entry>}. 

4.5.6 VALIDATE-PARAMETER-DECLARATION

Operation:  
\texttt{validate-parameter-declaration} \( p_d \) 
where \( p_d \) is a \texttt{<declaration>}. 

Step 1. Each \texttt{<empty-expression>} in \( p_d \) must not contain a \texttt{<refer-option>}. 

Step 2. For each \texttt{<empty-expression>} \texttt{<expression>} \( e \); in \( p_d \) perform steps 2.1. 

Step 2.1. Perform \texttt{evaluate-restricted-expression} \( e \) to obtain \( e \). \( e \) must be a \texttt{<constant>}. Replace the first immediate component of \( e \) by \( e \). 

4.5.7 VALIDATE-STATIC-DECLARATION

Operation:  
\texttt{validate-static-declaration} \( s_d \) 
where \( s_d \) is a \texttt{<declaration>}. 

Step 1. Perform \texttt{validate-automatic-declaration} \( s_d \). 

Step 2. For each \texttt{<expression>}, single component of \( s_d \) which is not a component of an \texttt{<offset>} perform \texttt{evaluate-restricted-expression} \( e \) to obtain \( e \); which must be a \texttt{<constant>} or a \texttt{<value-reference>}. IF \( e \) is a \texttt{<value-reference>} then it must be a component of an \texttt{<initial-element>}. Replace the first immediate component of \( e \) by \( e \). 

4.5.8 VALIDATE-DESCRIPTOR

Operation:  
\texttt{validate-descriptor} \( d \) 
where \( d \) is a \texttt{<parameter-descriptor>} or a \texttt{<returns-descriptor>}. 

Step 1. For each \texttt{<empty-expression>} \texttt{<expression>} \( e \); in \( d \), perform step 1.1. 

Step 1.1. Perform \texttt{evaluate-restricted-expression} \( e \) to obtain \( e \). \( e \) must be a \texttt{<constant>}. Replace the first immediate component of \( e \) by \( e \). 

Step 2. If \( d \) is a \texttt{<parameter-descriptor>}, any \texttt{<entry>} simply contained in \( d \) must not have any subelements. 

Step 3. \( d \) must not contain any \texttt{<refer-option>}. 

Chapter 3: The Translator 119
4.5.9 EVALUATE-RESTRICTED-EXPRESSION

Operation: evaluate-restricted-expression(e)

where e is an <expression>.
result: a <constant> or a <value-reference> or <fail>.

Step 1. For each <expression> ex simply contained in e, perform Steps 1.1 and 1.2.
Step 1.1. Perform evaluate-restricted-expressions(e) to obtain r. If r is <fail> then return <fail>.
Step 1.2. Replace the first immediate component of ex by r.

Step 2.
Case 2.1. e immediately contains a <value-reference>, vr.

if vr immediately contains either a <named-constant-reference> or a
<multin-function-reference> which has <empty-bif> or <null-bif>, then
return vr; otherwise return <fail>.

Case 2.2. (Otherwise),

Step 2.2.1. If e has an <infix-operator> which does not contain <add>, <subtract>,
<multiply>, <divide>, or <call>, then return <fail>.
Step 2.2.2. Perform evaluate-expression(e) to obtain an <aggregate-value>, sv.
sv must immediately contain an <aggregate-type> which immediately contains
<scalar>. Let v be the <basic-value> in sv.
Step 2.2.3. Let dt be the <data-type> in the <data-description> immediate component
of e.
Step 2.2.4. Return a <constant>: v dt.

4.5.10 APPLY-CONSTRAINTS

In certain contexts of an abstract <procedure> only restricted form of the specified
category are permitted. The restrictions are shown by a constraint-expression enclosed
in parentheses. The definition of a constraint-expression is as follows.

constraint-expression::= multiple-constraint

multiple-constraint::= constraint

| multiple-constraint & constraint

constraint::= ~ constraint | literal | based | character | condition | named-constant |

| defined | computational-type | file | format | controlled | label |

| locator | pointer | scalar | variable

Operation: apply-constraints(p)

where p is a <procedure>.

Step 1. For each subnode, c in p corresponding to a category-name in the Abstract Syntax
with an attached constraint-expression, ca, if c is an <expression-list>, let c be each component of the list in turn, and perform Steps 1.1 and 1.2; otherwise, perform Steps 1.1 and 1.2.
Step 1.1.

Case 1.1.1. c is a \(<variable-reference>\) and ce contains a constraint containing "defined" or "based".

Let d be the \(<declaration>\) designated by the \(<declaration-designator>\) immediately contained in c.

Case 1.1.2. c is a \(<declaration-designator>\).

Let d be the \(<declaration>\) designated by c.

Case 1.1.3. c is a \(<variable-reference>\) (other than as in Case 1.1.1), a \(<value-reference>\), an \(<expression>\), a \(<parenthesized-expression>\), a \(<character-expression>\), or a \(<named-constant-reference>\).

Let d be the \(<data-description>\) immediately contained in c.

Step 1.2. Perform \(test-constraints(d,ce)\) to obtain r.

r must be \(<true>\).

8.5.11 TEST-CONSTRAINTS

Operation: \(test-constraints(d,ce)\)

where d is a \(<declaration>\) or a \(<data-description>\),

ce is a \(<constraint-expression>\), a \(<multiple-constraint>\), or a \(<constraint>\).

result: \(<true>\) or \(<false>\).

Case 1. ce is of the form \(<constraint-expression>\): constraint-expression,con \& multiple-constraint,mc.

Perform \(test-constraints(d,cone)\) to obtain r1; perform \(test-constraints(d,mc)\) to obtain r2. If r1 is \(<true>\), return \(<true>\); otherwise return r2.

Case 2. ce is of the form \(<multiple-constraint>\): multiple-constraint,mc & constraint,ct.

Perform \(test-constraints(d,mc)\) to obtain r1; perform \(test-constraints(d,ct)\) to obtain r2. If r1 is \(<false>\), return \(<false>\); otherwise return r2.

Case 3. ce is of the form constraint: ~ constraint,ct.

Perform \(test-constraints(d,ct)\) to obtain r. If r is \(<false>\), return \(<true>\); otherwise return \(<false>\).

Case 4. ce is a constraint: scalar.

If d is a \(<declaration>\) which has \(<variable>\) immediately containing \(<data-description>\), \(<item-data-description>\), then return \(<true>\). If d is a \(<data-description>\), \(<item-data-description>\), then return \(<true>\). Otherwise return \(<false>\).

Case 5. ce is a constraint: computational-type.

If d contains \(<non-computational-type>\) then return \(<false>\); otherwise return \(<true>\).

Case 6. ce is a constraint other than in Cases 3 and 4.

If d contains a category-name equal to ce other than as a subnode of \(<entity>\), return \(<true>\); otherwise return \(<false>\).

Case 7. (Otherwise).

Let ce be the immediate component of ce. Perform \(test-constraints(d,ce)\) to obtain r. Return r.
4.6 Validate-program

Operation: validate-program

Step 1. For each distinct <identifier> component of the <program> which is an immediate component of a <declaration> which contains <external>, let adl be a <declaration-list> containing copies of all such <declaration>s and perform validate-external-declaration(adl).

4.6.1 VALIDATE-EXTERNAL-DECLARATION

Operation: validate-external-declaration(adl)

where adl is a <declaration-list>.

Step 1. Delete all <identifier-list> components of adl which are immediate components of <structure-data-description>.

Step 2. Delete all <variable-reference>s which are immediate components of an <offset>.

Step 3. For each <declaration>d which has <storage-class>; <static>; and for each <item-data-description> component of d which contains an <initial>, change any <initial-element> which contains an <iteration-factor> and an <initial-element-list> into the equivalent number of <initial-element>s.

Step 4. If any <declaration>d component of adl contains <storage-class>; <controlled>; then delete every <initial> component of d and every <extent-expression> component whose <expression> does not immediately contain <constant>.

Step 5. For each <extent-expression>,ee in adl containing a <constant>c, perform Step 5.1.

Step 5.1. Let e be the <expression> of ee. Perform evaluate-expression-to-integer(e) to obtain an <integer-value>iv. Replace e by iv.

Step 6. For each <initial-element>,ie in adl containing a <constant>c, perform Step 6.1.

Step 6.1. Let tdt be the <data-type> immediately contained in the <item-data-description> containing ie. Let cd be the <data-type> of c, and let cb be the <basic-value> of c. Perform convert(tdt,cd,cb) to obtain a <basic-value>iv. Replace c by (constant); by tdt.

Step 7. Delete any <local> which is an immediate component of a <non-computational-type>.

Step 8. In an implementation-defined fashion, compare <declaration>s of adl which have <environment> components with those which do not have corresponding <environment> components of the <declaration>s of adl. Delete all <environment> components of adl.

Step 9. In an implementation-defined fashion, compare <declaration>s of adl which have <options> components with those which do not have corresponding <options> components, and compare corresponding <options> components of the <declaration>s of adl. Delete all <options> components of adl.

Step 10. All <declaration> components of adl must be equal.
Chapter 5: The PL/I Interpreter

8.0 Introduction

This chapter gives the interpretation-state part of the Machine-state Syntax and also introduces the interpretation phase of the definition. Section 5.1 defines the interpretation-state. Section 5.2 defines some terminology used in the subsequent chapters. Section 5.3 gives the operation interpret and some operations called from it to initialize and terminate the interpretation phase. The subsequent chapters complete the definition of the interpretation phase.

8.1 The Interpretation-state

N0. 
<interpretation-state>::= <program-state> <allocated-storage> {<dataset-list>}
N7. 
<program-state>::= <program-directories>
{<block-state-list>}
{<file-information-list>}

8.1.1 DIRECTORIES

N9. 
<program-directories>::= <static-directories>
<controlled-directories>
<file-directories>
N10. 
<static-directories>::= {<static-directory-entry-list>}
N11. 
<static-directory-entry>::= {<external> | <declaration-designator> }
<identifier> <generation>
N12. 
<controlled-directories>::= {<controlled-directory-entry-list>}
N13. 
<controlled-directory-entry>::= {<external> | <declaration-designator> }
<identifier> {<generation-list>}
N14. 
<file-directories>::= {<file-directory-entry-list>}
N15. 
<file-directory-entry>::= {<external> | <declaration-designator> }
<identifier> {<subscript-value-list>}
<file-information-designator>
M15. 
<subscript-value>::= <integer-value>

8.1.2 BLOCK STATE

N16. 
[block-state>::= <block-directory>
<block-control>
<linkage-part>
{<established-on-unit-list>}
{<block-environments>}
{<condition-bit-value-list>}
{<copy-files>}
N17. 
:block-directory>::= <automatic-directories>
<defined-directories>
{<parameter-directories>}
N18. 
<automatic-directories>::= {<automatic-directory-entry-list>}
N19. 
<automatic-directory-entry>::= <identifier> <generation>
N20. 
<defined-directories>::= {<defined-directory-entry-list>}

Chapter 5: The PL/I Interpreter 143
121.  <Defined-directory-entry>::= <identifier> <evaluated-data-description>
122.  <parameter-directory>::= (parameter-directory-entry-list)
123.  <parameter-directory-entry>::= <identifier>
    {<undefined> | <established-argument>}
124.  <established-argument>::= <generation> ( <empty> | <act-empty> )
125.  <evaluated-entry-reference>::= <entry-value> <established-argument-list>
126.  <block-control>::= <executable-unit-designator> <group-control>
    <statement-control> {string-lo-control} <data-item-control-list> <format-control-list>
    {current-scalar-item-list} <remote-block-state> <current-file-value>
127.  <remote-block-state>::= <block-state-designator>
128.  <current-file-value>::= <file-value>
129.  <group-control>::= <controlled-group-state-list>
130.  <controlled-group-state>::= <spec-designator> <cv-target> <cv-type>
    {dyn-value} <converted-by-type> {to-value} <converted-to-type>
131.  <cv-target>::= <evaluated-target>
132.  <cv-type>::= <data-type>
133.  {dyn-value}: = <real-value> | <complex-value>
134.  <converted-by-type>::= <data-type>
135.  <to-value>::= <real-value>
136.  <converted-to-type>::= <data-type>
137.  <statement-control>::= <operation-list>
138.  <string-lo-control>::= <character-string-value> {string-limit} {first-comma}
139.  <string-limit>::= <integer-value>
140.  {first-comma}: = <opp> | <off>
141.  <data-item-control>::= <data-list-indicator> <data-item-indicator>
142.  <data-list-indicator>::= <designator>
143.  <data-item-indicator>::= <designator> | <undefined>
144.  <format-control>::= <format-specification-list-designator> <format-list-index>
    {format-iteration-value} {format-iteration-index} <statement-designator>
    <remote-block-state>
145.  <format-list-index>::= <integer-value>
146.  <format-iteration-value>::= <integer-value>
147.  <format-iteration-index>::= <integer-value>
148.  <current-scalar-item>::= <basic-value> <data-type> {data-name-field} |
    <evaluated-target>
149.  <data-name-field>::= {symbol-list}
150.  <linkage-parts>::= [entry-point-designator]
    {returned-value} | {returned-source-value} |
    {prologue-flag}
5.1.3 FILE INFORMATION

5.1.3.1 FILE INFORMATION

5.1.3.2FILE INFORMATION

<file-information>::= <open-state> <filename>
<file-description> [<file-opening>]

<open-state>::= <open> | <closed>

<filename>::= <character-string-value>

<file-opening>::= <dataset-designator> <complete-file-description>
<current-position> [ <delete-file> ] [ <allocated-buffer> ] [ <page-number> ] [ <first-column>-]]

<complete-file-description>::= <evaluated-file-description-list>

<evaluated-file-description>::= <stream> [ <record> ] [ <input> ] [ <output> ] [ <update> ] [ <sequential> ] [ <direct> ] [ <print> ] [ <keyed> ] [ <environment> ]
<evaluated-tab-option> | <evaluated-title> | <evaluated-linewidth> | <evaluated-page-size>

<evaluated-tab-option>::= <integer-value-list>

<evaluated-title>::= <character-string-value>

<evaluated-linewidth>::= <integer-value>

<evaluated-page-size>::= <integer-value>

<current-position>::= <designator> | <undefined>

<allocated-buffer>::= <generation> [ <key> ]

<page-number>::= <integer-value>
5.1.4 STORAGE AND VALUES

\[\text{allocated-storage} ::= \{\text{allocation-unit-list}\}\]

\[\text{allocation-unit} ::= \\begin{array}{ll}
\text{basic-value} & \text{real-number} \\
\text{character-string-value} & \text{file-value} \\
\text{entry-value} & \text{offset-value} \\
\text{format-value} & \text{area-value} \\
\text{pointer-value} & \text{undefined} \\
\end{array}\]

\[\text{real-value} ::= \text{real-number}\]

\[\text{complex-value} ::= \text{complex-number}\]

\[\text{complex-number} ::= \text{real-number} \mid \text{undefined} \mid \text{real-number} \mid \text{undefined}\]

\[\text{real-number} ::= \]

The members of the set of real numbers are the alternative choices as immediate and terminal components of \text{real-number}.

\[\text{integer-value} ::= \]

The members of the set of integers are the alternative choices as immediate and terminal components of \text{integer-value}.

\[\text{character-string-value} ::= \text{character-value-list} \mid \text{null-character-string}\]

\[\text{character-value} ::= \text{symbol} \mid \text{undefined}\]

\[\text{bit-string-value} ::= \text{bit-value-list} \mid \text{null-bit-string}\]

\[\text{bit-value} ::= \text{zero-bit} \mid \text{one-bit} \mid \text{undefined}\]

\[\text{entry-value} ::= \text{entry-point-designator} \mid \text{block-state-designator}\]

\[\text{label-value} ::= \text{executable-unit-designator} \mid \text{block-state-designator}\]

\[\text{format-value} ::= \text{format-statement-designator} \mid \text{block-state-designator}\]

\[\text{file-value} ::= \text{file-information-designator}\]

\[\text{pointer-value} ::= \text{pointer-designator} \mid \text{null}\]

\[\text{offset-value} ::= \text{evaluated-data-description} \mid \text{significant-allocation-list} \mid \text{storage-index-list} \mid \text{null}\]

\[\text{area-value} ::= \text{area-allocation-list} \mid \text{significant-allocation-list} \mid \text{empty}\]

\[\text{area-allocation} ::= \text{significant-allocation-list} \mid \text{allocation-unit}\]

\[\text{significant-allocation} ::= \text{evaluated-data-description} \mid \text{occupancy}\]

\[\text{occupancy} ::= \text{allocated} \mid \text{free}\]

\[\text{aggregate-value} ::= \text{aggregate-type} \mid \text{basic-value-list}\]

\[\text{aggregate-type} ::= \text{dimensioned-aggregate-type} \mid \text{structure-aggregate-type} \mid \text{scalar}\]

\[\text{dimensioned-aggregate-type} ::= \text{element-aggregate-type} \mid \text{bounded-pair-list}\]

\[\text{element-aggregate-type} ::= \text{structure-aggregate-type} \mid \text{scalar}\]

\[\text{structure-aggregate-type} ::= \text{member-aggregate-type-list}\]

\[\text{member-aggregate-type} ::= \text{aggregate-type}\]
5.1.5 GENERATIONS, EVALUATED DATA DESCRIPTIONS, AND EVALUATED TARGETS

M107. \(<\text{generation}>::= \text{evaluated-data-description} \\
<\text{allocation-unit-designator} > \\
<\text{storage-index-list} >\)

M108. \(<\text{evaluated-data-description}>::= <\text{data-description} > \\
\text{Note: <external-expression> components of <evaluated-data-descriptions> <data-description> contain only integer-values. (See Section 7.1.)}\)

M109. \(<\text{storage-index}>::= \text{basic-value-index} (<\text{position-index}> )\)

M110. \(<\text{basic-value-index}>::= \text{integer-value} \)

M111. \(<\text{position-index}>::= \text{integer-value}\)

M112. \(<\text{evaluated-target}>::= <\text{generation} > | <\text{evaluated-pseudo-variable-reference} >\)

M113. \(<\text{evaluated-pseudo-variable-reference}>::= <\text{pseudo-variable} >\) \\
\{<\text{generation} > | <\text{aggregate-value} >\} \\
\{<\text{aggregate-value} >\}

5.1.6 DATASET

M114. \(<\text{dataset}>::= <\text{dataset-name} > (<\text{record-dataset} > | <\text{stream-dataset} >)\)

M115. \(<\text{dataset-name}>::= \text{character-string-value}\)

M116. \(<\text{record-dataset}>::= <\text{sequential-dataset} > \\
\{<\text{keyed-dataset} > \} \\
\{<\text{keyed-sequential-dataset} > \}\)

M117. \(<\text{sequential-dataset}>::= <\text{alpha} > (<\text{record-list} > | <\text{open} >)\)

M118. \(<\text{keyed-dataset} >::= <\text{keyed-record-list} >\)

M119. \(<\text{keyed-sequential-dataset} >::= <\text{alpha} > (<\text{keyed-record-list} > | <\text{open} >)\)

M120. \(<\text{keyed-record} >::= <\text{record} > <\text{key} >\)

M121. \(<\text{record}>::= <\text{evaluated-data-description} > <\text{basic-value-list} >\)

M122. \(<\text{key} >::= \text{character-string-value}\)

M123. \(<\text{stream-dataset} >::= <\text{alpha} > (<\text{stream-item-list} > | <\text{open} >)\)

M124. \(<\text{stream-item} >::= <\text{symbol} > | <\text{line-feed} > | <\text{par-indent} > | <\text{par-return} >\)

Chapter 5: The PL/I Interpreter 187
8.2 Terminology and Definitions

The following terms are employed at various places throughout the operations which comprise the interpretation phase.

8.2.1 CURRENT

(1) The last <block-state> member (if any) of the <block-state-list> is termed the <current-block-state>.

(2) Excepting only components of its <controlled-group-state-list> simple component (if any), any component of the current <block-state> is termed current. For example, the <executable-unit-designator> simple component of the current <block-state> is termed the current <executable-unit-designator>.

(3) The last <controlled-group-state> member (if any) of the current <controlled-group-state-list> is termed the current <controlled-group-state>.

(4) Any component of the current <controlled-group-state> is termed current. For example, the <by-value> component of the current <controlled-group-state> is termed the current <by-value>.

(5) The corresponding block (see Section 5.2.2) of the current <block-state> is termed the <current-block>.

(6) Any simple component of the current block is also termed current. For example, the <end-statement> immediate component of the current block is termed the current <end-statement>.

8.2.2 BLOCK

The term block is used to refer to a <simple-block>, a <procedure>, or an <abstract-external-procedure>. Each <block-state> present in the <machine-state> is created to be associated with some particular block. That block is termed the corresponding block of the <block-state> and it may be located since it is that block which has as block-component the <executable-unit-designator> designated by the <executable-unit-designator> simple component of the <block-state> in question.

8.3 The Interpret Operation and the Initialization of the Interpretation State

8.3.1 INTERPRET

First, the <interpretation-state> is initialized, and then the <program> is executed.

Operation: \text{interpret}(\text{di}, \text{ev})

where \text{di} is a <dataset-lists>,
\text{ev} is an <entry-value>.

Step 1. Perform initialize-interpretation-state(di).
Step 2. Let \text{ev} be <evaluated-entry-reference>; \text{ev}. Perform activate-procedure(\text{ev}).
Step 3. Perform program-epilogue.
5.3.2 INITIALIZE-INTERPRETATION-STATE

Initialize-interpretation-state constructs the initial configuration of the <interpretation-state>, including certain portions that are a function of the <program> to be interpreted.

Operation:  initialize-interpretation-state(d1)

where d1 is a <dataset-list>.

Step 1. Append to the <machine-state> the tree

   <interpretation-state>;
    <program-state>;
    program-directory>;
    <static-directory>;
    <controlled-directory>;
    <file-directory>;
    <allocated-storage>
   d1.


Step 3. Perform build-controlled-directory.


5.3.3 BUILD-FILE-DIRECTORY-AND-INFORMATIONS

Operation:  build-file-directory-and-informations

Step 1. For each <declaration> d in the <program>, which has <named-constant> with <file>, selected in any order:

   Case 1.1. d has <external>, and there exists in the <file-directory> a <file-directory-entry> with both <external> and an <identifier> that is equal to the <identifier> in d.

   No action.

   Case 1.2. (Otherwise).

   Case 1.2.1. d does not have a <bound-pair-list>.

   Perform build-idi(d).

   Case 1.2.2. d does have a <bound-pair-list>, bpl.

   Step 1.2.2.1. bpl must not contain <constant> or <refer-option>.

   Step 1.2.2.2. Let n be the number of <bound-pair>s in bpl. For i=1,...,n, let lb(i) and ub(i) be the <lower-bound> and <upper-bound> respectively in the i-th <bound-pair> of bpl.

   Step 1.2.2.3. For i=1,...,n, perform evaluate-expression-to-integer(lb(i)) to obtain an <integer-value>, lb(i), and perform evaluate-expression-to-integer(ub(i)) to obtain an <integer-value>, ub(i).

   Step 1.2.2.4. For each distinct <subscript-value-list>, s, having a <subscript-values> such that the i-th contained <integer-value> lies in the inclusive range defined by lb(i) and ub(i), selected in any order, perform build-idi(s, s).
5.3.4 BUILD-FDI

Operation: `build-fdi(d, subl)`
where `d` is a `<declaration>`,
`subl` is a `<subscript-value-list>`.

Step 1. Let `fn` be a `<filename>` whose component symbols are those of the `<identifier>` in `d`, and are taken in the same order. Let `fd` be the `<file-description>` in `d`. Let `info` be a

- `<file-information>`
- `<open-state>`
- `<closed>`
- `fn`
- `fd`.

Step 2. Append `info` to the `<file-information-list>`. Let `fid` be a `<file-information-designator>` which designates this appended `<file-information>` node.

Step 3. If `d` has `<external>`, then let `sc` be `<external>`; otherwise let `sc` be a `<declaration-designator>` designating `d`. Let `id` be the `<identifier>` in `d`. Let `fde` be

- `<file-directory-entry>`
- `sc`
- `id`
- `fid`.

If `subl` is present, append it to `fde`.

Step 4. Append `fde` to the `<file-directory-entry-list>`.

5.3.5 BUILD-CONTROLLED-DIRECTORY

Operation: `build-controlled-directory`

Step 1. For each `<declaration>` `d` in the `<program>`, selected in any order:

Case 1.1. `d` has `<controlled>`.

Case 1.1.1. `d` has `<external>`, and there exists in the `<controlled-directory>` a `<controlled-directory-entry>` with both `<external>` and an `<identifier>` that is equal to the `<identifier>` in `d`.

No action.

Case 1.1.2. (Otherwise),

Step 1.1.2.1. If `d` has `<external>`, then let `sc` be `<external>`; otherwise let `sc` be a `<declaration-designator>` designating `d`. Let `id` be the `<identifier>` in `d`.

Step 1.1.2.2. Append, to the `<controlled-directory-entry-list>`, the tree `<controlled-directory-entry>`, `sc id`.

Case 1.2. (Otherwise):

No action.
5.3.6 ALLOCATE-STATIC-STORAGE-AND-BUILD-STATIC-DIRECTORY

Operation: \texttt{allocate-static-storage-and-build-static-directory}

Step 1. For each \texttt{<declaration>} \texttt{d} in the \texttt{<program>}, selected in any order:

\textbf{Case 1.1.} \texttt{d} has \texttt{<static>},

\textbf{Case 1.1.1.} \texttt{d} has \texttt{<external>}, and there exists in the \texttt{<static-directory>} a \texttt{<static-directory-entry>} with both \texttt{<external>} and an \texttt{<identifier>} that is equal to the \texttt{<identifier>} in \texttt{d}.

No action.

\textbf{Case 1.1.2.} (Otherwise).

\textbf{Step 1.1.2.1.} Let \texttt{dd} be the \texttt{<data-description>} in \texttt{d}. Perform \texttt{evaluate-data-description-for-allocation(dd)} to obtain an \texttt{<evaluated-data-description>} \texttt{edd}.

\textbf{Step 1.1.2.2.} Perform \texttt{allocate(edd)} to obtain a \texttt{<generation>} \texttt{q}.

\textbf{Step 1.1.2.3.} If \texttt{d} has \texttt{<initial>}, then perform \texttt{initialize-generation(d,q)}.

\textbf{Step 1.1.2.4.} If \texttt{d} has \texttt{<external>}, then let \texttt{sc} be \texttt{<external>}; otherwise let \texttt{sc} be a \texttt{<declaration-designator>} designating \texttt{d}. Let \texttt{id} be the \texttt{<identifier>} in \texttt{d}.

\textbf{Step 1.1.2.5.} Append, to the \texttt{<static-directory-entry-list>}, a \texttt{<static-directory-entry>} \texttt{sc id q}.

\textbf{Case 1.2.} (Otherwise).

No action.

5.3.7 PROGRAM-EPILOGUE

Operation: \texttt{program-epilogue}

Step 1. For each \texttt{<file-information>} \texttt{fi} containing \texttt{<open>}, perform \texttt{close(fi)}, where \texttt{fi} is a \texttt{<file-value>} designating \texttt{fi}. 
Chapter 6: Flow of Control

6.0 Introduction

The definition of the control mechanism of the FL/I Interpreter, introduced in Chapter 5, is completed in this chapter. The definition treats in order the three levels of control, pertaining to the program, the block, and the operations within the block. This is followed by the definition of the control of interrupt operations.

Within the execution of the program, there may be several blocks active at any time, but execution proceeds sequentially only within the most recently activated block while the execution of the other blocks is temporarily suspended.

6.1 Program Activation and Termination

The activation of a program is described in Section 5.1, the initialization of the <program-state> being followed by the performance of the activate-procedure operation. This causes the first <block-state> to be created.

A <program-state> in general contains a list of <block-state>s, one for each block activated within it and not yet terminated. When the execution of a program is terminated, all the contained block activations are also terminated. The deletion of the last remaining <block-state> results in control returning to Step 3 of the interpret operation and performance of the program-epilogue operation (see Section 5.1.7).

6.1.1 PROGRAM TERMINATION

A program may be terminated:

(1) "abnormally", by execution of a <stop-statement>, or

(2) "normally", by execution of an <end-statement> or <return-statement>, in circumstances which lead to the epilogue operation being performed in the original <block-state>. Since the <end-statement> and <return-statement> can also be used for other purposes, their execution will be described in Section 6.1.

6.1.1.1 Execute-stop-statement

Operation: \( \text{execute-stop-statement}(\text{as}) \)

where as is a <stop-statement>.

Step 1. Perform \( \text{raise-condition}(\text{finish-condition}) \).

Step 2. Perform \( \text{stop-program} \).

6.1.1.2 Stop-program

Operation: \( \text{STOP-PROGRAM} \)

Step 1. For each <block-state> contained in the <program-state> except the current <block-state>, replace the <statement-control> by

\(<\text{statement-control}>; <\text{operation-list}>; <\text{operations for epilogue}>\).

Step 2. Perform epilogue.
6.2 Block Activation and Termination

Block activation is described by defining first those actions which are different for
<procedure> and <begin-block>s. The operations prologue and epilogue are the same for
both kinds of block.

6.2.1 ACTIVATE-PROCEDURE

A <procedure> may be activated by execution of a <call-statement>, or by evaluation of a
<value-reference> which is a <procedure-function-reference>. Also an <on-unit> has a
<procedure> which may be activated on the occurrence of an interrupt.

This operation completes when epilogue (see section 6.2.4) is executed and eliminates the
<block-state> and its contained operations.

Operation: activate-procedure(eer, chifs)

where eer is an <evaluated-entry-reference>,
chifs is a <condition-bit-value-list>.

Step 1. Let epd be the <entry-point-designator> of eer, designating an <entry-point>, ep,
which is a simple component of a <procedure>, p.

Step 2. If there exists in the <block-state-list> a <block-state> whose corresponding
block is p, then p must simply contain <reverse>, unless p is the immediate
component of an <on-unit>.

Step 3. Let eod be a designator of the first <executable-unit> after ep in the <entry-
or-executable-unit-list> simply containing ep. If eer contains a <block-state-
designator>, bsd, let bie be <block-environment> bie. Otherwise bie is <absent>.
If bie is present, it must designate an existing <block-state>.

Step 4. Let bs be a

<block-state>:
<block-directory>:
<automatic-directory>
<defined-directory>
<parameter-directory>;
<block-control>:
<executable-unit-designator>:
end;
<query-control>
<statement-control>:
<operation-lists>:
<operation> for instal-arguments(eer);

if bie is a <block-environment> then attach bie to bs. If chifs is a
<condition-bit-value-list> then attach chifs to bs.

Step 5. Append bs to the <block-state-list>.
6.7.1.1 INSTALL-ARGUMENTS

Operation: install-arguments

where e is an <evaluated-entry-reference>.

Step 1. If the <parameter-name-list>, pni of the <entry-point> designated by the <entry-point-designator> of e er exists, then perform step 1.1.

Step 1.1. Let eal be the <established-argument-list> of e. Attach to the current <parameter-directory-entry> a <parameter-directory-entry-pdi>, with the same number of immediate components as pni and whose i'th immediate component is

<parameter-directory-entry>
<identifier> of the i'th immediate component of pni
<established-argument>, the i'th immediate component of eal.

Step 2. For each <identifier>, id immediately contained in a <declaration> containing <parameter> in the current block, and not contained in pni, append to pdi a

<parameter-directory-entry>
id <undefined>.

Step 3. Perform prologue.

6.7.2 ACTIVATE-BEGIN-BLOCK

A <begin-block> is activated when the operation execute-executable-unit is applied to the <executable-unit> immediately containing it.

Operation: activate-begin-block

Step 1. Let eul be a designator of the first <executable-unit> of the <executable-unit>.
<begin-block> <executable-unit-list> designated by the current <executable-unit-designator>. Let bsd be a <block-state-designator> designating the current <block-state>. Append to the <block-state-list> a

<block-state>
<Block-directory>
<automatic-directory>
<defined-directory>
<block-control>
<executable-unit-designator> sub
<group-control>
<statement-control>
<operation-list>
<operation> for prologue
</linkage-part>
<block-environment>
bsd.

Chapter 6: Flow of Control 155
6.2.3 PROLOGUE

This operation is invoked at the beginning of every block activation to establish the <automatic> and <defined> variables local to that block. The <automatic> variables are initialized if their <declaration> specify initialization. Any <expression> evaluated during the prologue, such as in <extent-expression> or <expression> in <initial>, are not allowed to reference other <automatic> or <defined> variables local to this block. The operation find-directory-entry will impose the restriction when it finds a reference to a variable declared in a block for which there exists a <prologue-flag>. The <prologue-flag> is only present while the prologue operation is active.

Operation: prologue

Step 1. Attach a <prologue-flag> to the current <linkage-part>.

Step 2. For each <declaration>, d, of the current block, that contains <automatic> or <defined>, perform Step 2.1.

Step 2.1. Let id be the <identifier> immediately contained in d, and let dd be the <data-description> immediately contained in the <variable> of d. Perform evaluate-data-description-for-allocation(id) to obtain an <evaluated-data-description>, dd.

Case 2.1.1. d contains <automatic>.

Step 2.1.1.1. Perform allocate(dd) to obtain a <generation>, g.

Step 2.1.1.2. Append to the current <automatic-directory-entry-list> an <automatic-directory-entry>; id g.

Step 2.1.1.3. If d contains <initial> then perform initialize-generation(g, d).

Case 2.1.2. d contains <defined>.

Append to the current <defined-directory-entry-list> a <defined-directory-entry>; id dd.

Step 3. Delete the <prologue-flag> of the current <linkage-part>.

Step 4. Replace the current <statement-control> by a

<statement-control>:
  <operation-list>:
    <operation> for advance-execution.

6.2.4 EPilogue

This operation is used to terminate the execution of a block and may be invoked from executing an <end-statement>, a <return-statement>, a <stop-statement>, or a <goto-statement> which causes a transfer of control out of a block. This operation, which deletes the <block-state>, normally causes a return to activate-procedure or activate-begin-block, in the previous <block-state>.

Operation: epilogue

Step 1. For each current <parameter-directory-entry>, pde, which contains <dummy>, let g be the <generation> in pde and perform free(g).

Step 2. For each current <automatic-directory-entry>, ade, let h be the <generation> in ade and perform free(h).

Step 3. If there is a current <resource-value>, then attach its immediate subtree to the <returned-resource-value> of the preceding <block-state> of the <block-state-list>.

Step 4. Delete the current <block-state>.
6.3 Control within a Block

The operation normal-sequence sets the current \(<\text{executable-unit-designator}>\) to designate the next \(<\text{executable-unit}>\). This is normally the last action of each execution of an \(<\text{executable-unit}>\). The special cases of execution of a \(<\text{goto-statement}>\) and \(<\text{call-statement}>\) may set the \(<\text{executable-unit-designator}>\) independently, while the \(<\text{return-statement}>\) and \(<\text{loop-statement}>\) have no further need for it. The \(<\text{executable-unit-designator}>\) is initialized in each activation of a \(<\text{procedure}>\) or \(<\text{block}>\), and may be reset on entering a \(<\text{subprogram}>\) or \(<\text{if-statement}>\).

6.3.1 Normal-sequence

Operation: normal-sequence

Step 1. Let \(e_u\) be the \(<\text{executable-unit}>\) designated by the current \(<\text{executable-unit-designator}>\). Let \(e_u\) be the \(<\text{executable-unit-list}>\) or \(<\text{entry-or-executable-unit-list}>\) which contains \(e_u\), but does not contain any other \(<\text{executable-unit-list}>\) or \(<\text{entry-or-executable-unit-list}>\) which also contains \(e_u\).

Step 2. Let \(e_u\) be that immediate component of \(e_u\) which either contains \(e_u\) or is exactly \(e_u\).

Case 2.1. \(e_u\) is an \(<\text{executable-unit-list}>\).

Let \(e_u\) be that \(<\text{executable-unit}>\) which immediately follows \(e_u\) as an immediate component of \(e_u\).

Case 2.2. \(e_u\) is an \(<\text{entry-or-executable-unit-list}>\).

Step 2.2.1. Let \(e_u\) be that \(<\text{entry-or-executable-unit}>\) which immediately follows \(e_u\) as an immediate component of \(e_u\). If \(e_u\) immediately contains \(<\text{entry-point}>\), then let \(e_u\) be \(e_u\) and go to Step 2.2.1.

Step 2.2.2. Let \(e_u\) be the \(<\text{executable-unit}>\) immediate component of \(e_u\).

Step 3. Set the current \(<\text{executable-unit-designator}>\) to designate \(e_u\).

6.3.1.1 Advance-execution

This operation is the "driver" which initiates execution of each \(<\text{executable-unit}>\) as selected by the current \(<\text{executable-unit-designator}>\).

Operation: advance-execution

Step 1. Perform execute-executable-unit.

Step 2. Go to Step 1.

6.3.2 Execute-executable-unit

The current \(<\text{executable-unit-designator}>\) designates the \(<\text{executable-unit}>\) to be executed. Execution of an \(<\text{executable-unit}>\) consists of performing the appropriate "execute" operation. This operation normally terminates with the current \(<\text{executable-unit-designator}>\) designating some other \(<\text{executable-unit}>\) in the \(<\text{program}>\). Return of control to advance-execution then causes execute-executable-unit to be applied again.

Operation: execute-executable-unit

Step 1. Let \(f\) be the rightmost immediate component of the \(<\text{executable-unit}>\) designated by the current \(<\text{executable-unit-designator}>\).

Step 2. Perform execute-\(\text{xxx}(f)\), where \("\text{xxx}"\) is replaced by the sequence of symbols forming the name of the type of \(f\).
6.3.3 EXECUTE-BEGIN-BLOCK

Operation: \texttt{execute-begin-block}(b)

where \( b \) is a \texttt{begin-block}.

Step 1. Perform \texttt{activate-begin-block}.

Step 2. Perform normal-sequence.

6.3.4 EXECUTE-GROUP

Constraints: In a \texttt{do-spec}, \texttt{dsp}, let \( t_\text{r} \) be the \texttt{target-reference} component. For each \texttt{spec} of \texttt{dsp}, let

\[
\begin{align*}
&s & \text{be the } \texttt{expression} \text{ immediately contained in the } \texttt{spec} , \\
&b & \text{be the } \texttt{expression} \text{ in the } \texttt{by-option} , \\
&tr & \text{be the } \texttt{expression} \text{ in the } \texttt{to-option} , \\
r & \text{be the } \texttt{expression} \text{ in the } \texttt{repeat-option} , \\
\end{align*}
\]

if such options are present. The following constraints must hold for each \texttt{spec}:

1. If \( tr \) is present then \( tr \) and \( e \) must have \texttt{computational-type} and the derived nodes of \( tr, e, b, \) and \( tr_\text{r} \) must all be \texttt{true}.

2. If \( r \) is present then \( e_\text{r} \) and \( r \) must all have:

\( \langle \texttt{computational-type}, \texttt{locator}, \rangle \) or
\( \langle \texttt{non-computational-type}, \texttt{locator}, \rangle \), with immediate subnodes of the \texttt{non-computational-type}s belonging to the same category other than \texttt{locator}.

3. If \( tr \) has \texttt{pointer} and either or both of \( e \) and \( r \) has \texttt{offset}, then each such \texttt{offset} must contain a \texttt{variable-reference}. If \( tr \) has \texttt{offset} and either or both of \( e \) and \( r \) has \texttt{pointer}, then the \texttt{offset} in \( tr \) must contain a \texttt{variable-reference}.

Operation: \texttt{execute-group}(q)

where \( q \) is a \texttt{group}.

Step 1. Let \( feu \) be the first \texttt{executable-unit} simply contained in \( q \).

Case 1.1. \( q \) has a \texttt{non-iterative-group}.

Set the current \texttt{executable-unit-designator} to designate \( feu \).

Case 1.2. \( q \) has a \texttt{while-only-group}.

Let \( exp \) be the \texttt{expression} of the \texttt{while-option} of \( q \). Perform establish-truth-value(s) to obtain \( t \). If \( t \) is \texttt{true}, then set the current \texttt{executable-unit-designator} to designate \( feu \); otherwise, perform normal-sequence.

Case 1.3. \( q \) has a \texttt{controlled-group}.

Let \( dsp \) be the \texttt{do-spec} of \( q \). Perform establish-controlled-group(\texttt{dps}) to obtain \( t \). If \( t \) is \texttt{true}, then set the current \texttt{executable-unit-designator} to designate \( feu \); otherwise, perform normal-sequence.
4.3.4.1 Establish-controlled-group

This operation is used to set up an iteration in the cases of a <controlled-group>, <list-directed-input>, <list-directed-output>, <edit-directed-input>, <edit-directed-output>, and <data-directed-output>.

If the controlling <do-spec> is such as to indicate iteration, then an appropriate <controlled-group-state> is established and <true> is returned. If the controlling <do-spec> indicates no iteration, then no <controlled-group-state> is established and <false> is returned.

Operation: 
```plaintext
establish-controlled-group(dep)
where dep is a <do-spec>.
result: <true> or <false>.
```

Step 1. Let tr be the <target-reference> of dep, and dt be the <data-type> of tr. Perform evaluate-target-reference(tr) to obtain an <evaluated-target>, etc.

Step 2. Let sp be the first <spec> of dep. Append to the current <controlled-group-state-list>, the tree
```plaintext
<controlled-group-state>:
  <spec-designator> a designator designating sp;
  <ev-target> et;
  <ev-type> dt.
```

Step 3. Perform initialize-spec-options.

Step 4. Perform test-spec to obtain tv.

Case 4.1. tv is <true>.
Return <true>.

Case 4.2. tv is <false>.
Perform establish-next-spec to obtain tv2. If tv2 is <true>, then go to Step 4. Otherwise, delete the current <controlled-group-state> and return <false>.

4.3.4.2 Initialize-spec-options

Operation: 
```plaintext
initialize-spec-options
```

Step 1. Let sp be the <spec> designated by the current <spec-designator>. Let t be the <expression> immediate component of sp.

Step 2. Perform Steps 2.1 through 2.3 in any order.

Step 2.1. Perform evaluate-expression(t) to obtain an <aggregate-value>, av.

Step 2.2. If sp contains a <to-option>, t; <expression>, et; then perform Steps 2.2.1 through 2.2.3.

Step 2.2.1. Let edt be the <data-type> of et. Let dt be a
```plaintext
<data-type>:
  <computational-type>;
  <arithmetic>;
  <mode>;
  <real>
```
the derived common <base> of edt and the current <e-v-type> the derived common <scale> of edt and the current <e-v-type> the converted <precision> of edt.
Step 2.2.2. Perform evaluate-expression(st) to obtain an <aggregate-value> x. Let y be the <basic-value> in x. Perform convert0d,ext,y) to obtain a <basic-value> <real-value> y. Attach a <to-value> z to the current <controlled-group-state>.

Step 2.2.3. Let cp be the converted <precision> of the current <cv-type>, where dt is used as the target <data-type> for determining cp. Replace the <precision> tree in dt by cp. Attach a <converted-to-type> dt to the current <controlled-group-state>. (The <converted-to-type> will be used later as the target <data-type> for the addition of the <by-value> and the value of the <cv-target>.)

Step 2.3. If sp contains a <by-option> or a <to-option> then perform steps 2.3.1 through 2.3.5.

Step 2.3.1. Let st be a <data-type> of <expression>, or.

Case 2.3.1.1. sp contains a <by-option>: <expression>, or.

Perform evaluate-expression(st) to obtain an <aggregate-value> x. Let exv be the <basic-value> in x. Let edt be the <data-type> of exv.

Case 2.3.1.2. sp contains a <to-option> but not a <by-option>.

Let exv be a <basic-value> <real-value> l. Let edt be a <data-type> which is integer-type, except that its <base> has <decimal> and its <number-of-digits> has 1.

Step 2.3.2. Let dt be a <data-type> <computational-type> <arithmetic> derived common <code> of edt and current <cv-type> derived common <code> of edt and current <cv-type> converted <precision> of edt.

Perform convert0d,ext,exv,edt) to obtain a <basic-value> x. Let y be the <real-value> or <complex-value> immediately contained in x. Attach a <by-value> y to the current <controlled-group-state>.

Step 2.3.3. Let cp be the converted <precision> of the current <cv-type>, where dt is used as the target <data-type> for determining cp. Let p be the <number-of-digits> of cp, and let r be the <number-of-digits> of dt.

Step 2.3.4.

Case 2.3.4.1. dt has <float>.

Change the value of the <number-of-digits> of dt to max(p,r).

Case 2.3.4.2. dt has <fixed>.

Let q be the <scale-factor> of cp, and let s be the <scale-factor> of dt. Let n=maxl(p-q,r-s)+maxl(q,s)+1), where n is the maximum <number-of-digits> allowed for <fixed> with the <base> of dt. Let n=maxl(q,s). Change the value of the <number-of-digits> in dt to n, and change the value of the <scale-factor> in dt to s.

Step 2.3.5. Attach a <converted-by-type> dt to the current <controlled-group-state>. (The <converted-by-type> will be used later as the result <data-type> for the addition of the <by-value> and the value of the <cv-target>.)

Step 3. Let cvt be the immediate component of the current <cv-targets>. Let dd be the <data-description> immediate component of cvt. Perform assign0d,av,dd).
4.3.4.3 Test-spec

This operation is used to test whether the current controlling <spec> in a <do-spec> indicates continuation (<true> returned) or termination (<false> returned).

Operation: test-spec

result: <true> or <false>.

Step 1. If the immediate component of the current <controlled-group-state> contains a <control><value><y>, perform Steps 1.1 through 1.3.

Step 1.1. Let <cvt> be the immediate component of the current <control><value-target>. Perform value-of-evaluated-target(<cvt>) to obtain an <aggregate-value> containing a <basic-value><x>. Let <x> be the current <control><value-type>.

Step 1.2. Let <dct> be the current <converted-to-type>. Perform convert(<dct>, <x>, <y>) to obtain a <basic-value><cx>

Step 1.3. Let <by> be the current <by-value>. If <by> ≥ 0 and <cx> ≥ <by>, or if <by> < 0 and <cx> < <by>, return <false>.

Step 2. If the <spec> designated by the current <spec-designator> contains a <call-option><expression><e>, then perform Step 2.1.

Step 2.1. Perform establish-truth-value(e) to obtain <tv>. Return <tv>.

Step 3. Return <true>.

4.3.4.4 Establish-next-spec

This operation is used to advance through the list of <spec> in a <do-spec>. If there is a next <spec> available, then conditions are established to use it and <true> is returned. If there is no next <spec> available, then <false> is returned.

Operation: establish-next-spec

result: <true> or <false>.

Step 1. Let <sp> be the <spec> designated by the current <spec-designator>. Let <spl> be the <spec-list> which immediately contains <sp>. If <sp> is the last component of <spl> then return <false>.

Step 2. Replace the immediate component of the current <spec-designator> by a designator of the next <spec> component of <spl>.

Step 3. If the current <controlled-group-state> contains a <by-value> and a <converted-by-type> or a <dso-value> and a <converted-to-type> then delete them.

4.1.4.3 Test-termination-of-controlled-group

This operation is used to test for the termination of an iteration set up by establish-controlled-group. If the controlling <do-spec> is such as to indicate termination, then the current <controlled-group-state> is deleted and <true> is returned. If the controlling <do-spec> indicates continuation, then the appropriate <executable-unit-designator> or <data-ites-indicator> is set to continue and <false> is returned.

Operation: test-termination-of-controlled-group

result: <true> or <false>

Step 1. Let sp be the <spec> designated by the current <spec-designator>. Let svt be the <evaluated-target> of the current <cv-target>.

Case 1.1. sp contains a <repeat-option>, <expression>, re.

Let s be the <data-description> immediate component of re. Perform evaluate-expression(re) to obtain an <aggregate-value>, sv. Perform assign1(sv, svt, s).

Case 1.2. sp contains a <by-option> or a <to-option>.

Case 1.2.1. Let be the <data-type> in the current <converted-by-type>. Let cvt be the <data-type> in the current <cv-type>. Let cvt be a <data-type> that is the same as be except for its <precision>, which is the converted <precision> of cvt, with cvt being the target <data-type> for determining the converted <precision>.

Step 1.2.2. Let st be the <evaluated-target> in the current <cv-target>. Perform value-of-converted-target(st) to obtain an <aggregate-value>, s1. Let s2 be the <basic-value> in s1. Perform convert(cvt1, cvt, s2) to obtain a <basic-value>, s1. Let x be the <real-value> or <complex-value> in x.

Step 1.2.3. Let y be the <real-value> or <complex-value> in the current <by-value>. Perform arithmetic-result(x, y, st) to obtain z, where z is a <real-value> or a <complex-value>.

Step 1.2.4. Let st be a

<data-description>:
  <item-data-description>:
    <data-type>:
      st.

Let w be an

<aggregate-value>:
  <aggregate-type>:
    <data-type>:
      <data-type>:
        w.

Perform assign1(w, svt, s).

Case 1.3. sp does not contain a <repeat-option>, a <to-option>, or a <by-option>.

go to Step 3.

map 2. Perform test-spec to obtain tv. If tv is <true>, return <false> (which indicates that the group does not terminate).

Step 3. Perform establish-next-spec to obtain tv2. If tv2 is <true> then go to Step 2; otherwise delete the current <controlled-group-state> and return <true>.
6.3.5 EXECUTE-IF-STATEMENT

Operation: execute-if-statement(lfs)

where lfs is an <if-statement>.

Step 1. Let $a$ be the <expression> immediate component of the <test> of lfs. Perform establish-truth-value($a$) to obtain tv.

Step 2.

Case 2.1. tv is <true>.

Replace the immediate component of the current <executable-unit-designator> by a designator of the <executable-unit> of the <then-unit> of lfs.

Case 2.2. tv is <false> and lfs simply contains an <else-unit>, etc.

Replace the immediate component of the current <executable-unit-designator> by a designator of the <executable-unit> of each.

Case 2.3. tv is <false> and lfs does not simply contain an <else-unit>.

Perform normal-sequence.

6.3.5.1 Establish-truth-value

Operation: establish-truth-value(exp)

where exp is an <expression>.

result: <true> or <false>.

Step 1. Perform evaluate-expression(exp) to obtain an <aggregate-value>, sv: <basic-value-list>: <basic-value>,sv. Let dt be the <data-type> of exp.

Step 2. Let dt be a

<data-type>:
<comparision-type>:
<string>:
<string-type>:
<boolean>:
<maximum-length>:
<asterisk>.

Perform convert(dt,dtb,sv) to obtain b.

Step 3. If b contains a <bit-value> <one-bit> then return <true>; otherwise return <false>.

6.3.6 EXECUTE-CALL-STATEMENT

Operation: execute-call-statement(ccl)

where ccl is a <call-statement>.

Step 1. Let ar be the <subroutine-reference> component of ccl. Perform evaluate-entry-reference(ar) to obtain an <evaluated-entry-reference>, eer.

Step 2. Perform activate-procedure(eer).

Step 3. Perform normal-sequence.
### 4.3.6.1 Entry-references

An **entry-reference** is either a `<subroutine-reference>` or a `<procedure-function-reference>`. ( `<builtin-function-reference>`s are described in Section 9.4.)

The main difference between a `<subroutine-reference>` and a `<procedure-function-reference>` is that normal termination of a `<procedure>` in the `<subroutine-reference>` case is by a `<return-statement>` not containing an `<expression>`, or by an `<end-statement>`, whereas in the `<procedure-function-reference>` case it is by a `<return-statement>` containing an `<expression>`.

Evaluation of an entry-reference normally takes place just before activation of a `<procedure>`.

#### 4.3.6.1.1 Evaluate-entry-reference

**Operation:** evaluate-entry-reference(?r)

Where ?r is a `<subroutine-reference>` or a `<procedure-function-reference>`.

**result:** an `<evaluated-entry-reference>`

**Step 1.** Let ?r be the `<value-reference>` immediate component of ?r. Perform evaluate-value-reference(?r) to obtain an `<aggregate-value>` ?g. Let sv be the `<entry-values>` in ?g.


Delete from ?de and ?vr all `<variable-reference>`s which are immediate components of `<offset>`. In an implementation-defined fashion, compare corresponding `<options>` components of ?de and ?vr and, if either ?de or ?vr has `<options>` components and the other does not have corresponding `<options>` components, compare ?de and ?vr. Delete all `<options>` components of ?de and ?vr.

For each `<extent-expression>` ?ce in ?de or ?vr containing a `<constant>?c`. Let ?v be an `<expression>`; ?c; and perform evaluate-expression-to-integer(?v) to obtain an `<integer-value>` ?iv and replace on by `<extent-expression>` ?iv. If ?de and ?vr must now be equal. (This checks that the entry point to be invoked agrees with the declaration of the entry value reference, ?vr.)

**Step 3.**

**Case 3.1.** ?r does not contain an `<argument-list>`.


**Case 3.2.** ?r contains an `<argument-list>`, ?al.

Let ?n be the number of `<argument>`s in ?al. Let ?nal be an `<established-argument-list>` with ?n `<established-argument>` immediate components. For ?i=1,...,?n, taken in any order, perform Step 3.2.1.


4.3.6.1.2 establish-argument

Operation: \texttt{establish-argument} \{arg, dd\}

where \texttt{arg} is an \texttt{<argument>},
\texttt{dd} is a \texttt{<data-description>},

returns an \texttt{<established-argument>},

Case 1. \texttt{arg} does not immediately contain \texttt{<dummy>},

Let \texttt{vr} be the \texttt{<variable-reference> simple component of arg}. Perform evaluate-variable-reference(\texttt{vr}) to obtain a \texttt{<generation>}, \texttt{g}. Return an \texttt{<established-argument>} g \texttt{<not-dummy>},

Case 2. \texttt{arg} immediately contains \texttt{<dummy>},

\textbf{Step 2.1.} Let \texttt{e} be the \texttt{<expression>} in \texttt{arg}. Perform evaluate-expression(e) to obtain an \texttt{<aggregate-value>}, \texttt{av}.

\textbf{Step 2.2.} Let \texttt{avdd} be the \texttt{<data-description> immediate component of s}. Perform promote-and-convert(\texttt{av}, \texttt{avdd}) to obtain an \texttt{<aggregate-value>}, \texttt{av}.

\textbf{Step 2.3.} Let \texttt{odd} be a copy of \texttt{dd}. Replace each \texttt{<bound-pair>} in \texttt{odd} that is not a component of \texttt{constr} by the corresponding \texttt{<bound-pair>} in \texttt{av2}. For each \texttt{<area-size>}, \texttt{x}, contained in \texttt{odd}, let \texttt{y} be an \texttt{<integer-value>} determined by an implementation-defined algorithm, and replace \texttt{x} by an \texttt{<area-size>}, \texttt{<extent-expression>} \texttt{y}.

For each \texttt{<data-type>}, \texttt{st} simply contained in \texttt{odd} that simply contains a tree of the form \texttt{<string>}, \texttt{<maximum-length>}, \texttt{<asterisk>}; perform Step 2.3.1.

\textbf{Step 2.3.1.} Let \texttt{a} be the maximum of the lengths of all \texttt{<character-string-values>} or \texttt{<bit-string-values>} in \texttt{av2} that correspond to \texttt{st}. Replace the \texttt{<maximum-length>} in \texttt{st} by a \texttt{<maximum-length>}, \texttt{<extent-expression>}, \texttt{<integer-value>}, \texttt{a}.

\textbf{Step 2.4.} Perform evaluate-data-description-for-allocation(\texttt{odd}) to obtain an \texttt{<evaluated-data-description>}, \texttt{odd}. Perform allocate(\texttt{odd}) to obtain a \texttt{<generation>}, \texttt{g}.

\textbf{Step 2.5.} Let \texttt{svl} be the \texttt{<basic-value-list>} in \texttt{av2}. Perform set-storage(g, \texttt{svl}).

\textbf{Step 2.6.} Return an \texttt{<established-argument>} g \texttt{<dummy>}.

Chapter 4: Flow of Control 165
6.3.7 EXECUTE-GOTO-STATEMENT

Operation: \texttt{execute-goto-statement}(\texttt{gs})

where \texttt{gs} is a \texttt{goto-statement}.

Step 1. Let \(vr\) be the \texttt{value-reference} of \texttt{gs}. Perform \texttt{evaluate-value-reference}() to obtain an \texttt{aggregate-value}, \(ag\). \(ag\) must contain the form

\[
\langle\text{label-value}, lv; \langle\text{executeable-unit-designator}, tp; \langle\text{block-state-designator}, bs; \rangle\rangle\rangle.
\]

Step 2. The \texttt{block-state-list} must contain a \texttt{block-state}, \(bs\), designated by \(bs\). Its corresponding block contains the \texttt{executeable-unit} designated by \(tp\).

Case 2.1. \(bs\) is the current \texttt{block-state}.

Perform local-goto(\texttt{lv}).

Case 2.2. \(bs\) is not the current \texttt{block-state}.

Step 2.2.1. \(vr\) must not immediately contain a \texttt{data-description} whose \texttt{data-type} has \texttt{local}.

Step 2.2.2. The \texttt{statement-control} component of \(bs\) must not contain an \texttt{operation} for \texttt{execute-alocate-statement}, \texttt{execute-locate-statement}, or \texttt{prologue}.

Step 2.2.3. Replace the \texttt{statement-control} component of \(bs\) by:

\[
\langle\text{statement-control}\rangle; \langle\text{operation-list}\rangle;
\]

\[
\langle\text{operation}\rangle \text{ for } \texttt{advance-execution};
\]

\[
\langle\text{operation}\rangle \text{ for } \texttt{trim-to-control};
\]

\[
\langle\text{operation}\rangle \text{ for } \texttt{local-goto(\texttt{lv})} ;
\]

Step 2.2.4. For each \texttt{block-state}, \(b\) that occurs after \(bs\) and before the current \texttt{block-state} in the \texttt{block-state-list} of the \texttt{interpretation-state} perform Steps 2.2.4.1 and 2.2.4.2.

Step 2.2.4.1. The \texttt{statement-control} component of \(b\) must not contain an \texttt{operation} for \texttt{execute-alocate-statement}, \texttt{execute-locate-statement}, or \texttt{prologue}.

Step 2.2.4.2. Replace the \texttt{statement-control} of \(b\) by \texttt{statement-control}:

\[
\langle\text{operation-list}\rangle; \langle\text{operation}\rangle \text{ for epilogue}.
\]

Step 2.2.5. Replace the current \texttt{statement-control} by \texttt{statement-control}:

\[
\langle\text{operation-list}\rangle; \langle\text{operation}\rangle \text{ for epilogue}.
\]

6.3.7.1 Local-goto

Operation: \texttt{local-goto}(\texttt{lv})

where \texttt{lv} is a \texttt{label-value}.

Step 1. Let \(tp\) be the \texttt{executeable-unit-designator} in \texttt{lv}. Let \(su\) be the \texttt{executeable-unit} designated by \(tp\). If there is an \texttt{iterative-group}, \(g\) that contains \(su\) but does not contain an \texttt{iterative-group} or \texttt{begin-block} that contains \(su\), then the current \texttt{executeable-unit-designator} must designate an \texttt{executeable-unit} that is contained in \(g\).

Step 2. Perform trim-group-control(\texttt{tp}).

Step 3. Replace the current \texttt{executeable-unit-designator} by \(tp\).
6.3.7.2 Trim-group-control

Operation: \texttt{trim-group-control}(\texttt{eod})
where \texttt{eod} is a \langle executable-unit-designator \rangle.

Step 1. Let \texttt{e} be the \langle executable-unit \rangle designated by \texttt{eod}. Let \texttt{b} be the \langle begin-block \rangle or \langle procedure \rangle that \texttt{block-contains} \texttt{e}.

Step 2.

Case 2.1. \texttt{b} contains a \langle controlled-group \rangle that \texttt{contains} \texttt{e}.

Let \text{\texttt{n}} be the number of \langle controlled-group \rangles that \texttt{contains} \texttt{e} and are contained in \texttt{b}. If the current \langle controlled-group-state-list \rangle contains more than \text{\texttt{n}} \langle controlled-group-state \rangles, delete those after the \text{\texttt{n}}'th \langle controlled-group-state \rangle.

Case 2.2. (Otherwise).

If there is a current \langle controlled-group-state-list \rangle, delete it.

6.3.8 Execute-null-statement

Operation: \texttt{execute-null-statement}(\texttt{e})
where \texttt{e} is a \langle null-statement \rangle.

Step 1. Perform normal-sequence.

6.3.9 Execute-return-statement

Operation: \texttt{execute-return-statement}(\texttt{rs})
where \texttt{rs} is a \langle return-statement \rangle.

Step 1. Let \texttt{p} be the last \langle block-state \rangle of the \langle block-state-list \rangle whose corresponding \texttt{block} is a \langle procedure \rangle.

Case 1.1. \texttt{rs} does not \texttt{contain} an \langle expression \rangle.

There must be a \langle returns-designator \rangle in the \langle entry-point \rangle designated by the \langle entry-point-designator \rangle of \texttt{p}. If \texttt{p} is the first \langle block-state \rangle in the \langle block-state-list \rangle then perform value-condition (\langle finish-condition \rangle).

Case 1.2. \texttt{rs} \texttt{contains} an \langle expression \rangle, \texttt{e}.

There must be a \langle returns-designator \rangle, \texttt{rd} in the \langle entry-point \rangle designated by the \langle entry-point-designator \rangle of \texttt{p}. Let \texttt{dd} be the \langle data-description \rangle immediate component of \texttt{e}. \texttt{dd} must be proper for assignment to \texttt{rd} (see section 4.4.3).

Step 1.2.1. Perform \texttt{evaluate-expression(e)} to obtain \texttt{ev}.

Step 1.2.2. Perform \texttt{promote-and-converter(d, ev, dd)} to obtain an \langle aggregate-value \rangle, \texttt{ev}. Attach \langle returned-value \rangle, \texttt{ev} to the \langle linkage-part \rangle of the \langle block-state \rangle immediately preceding \texttt{p} in the \langle block-state-list \rangle.

Step 2. For each \langle block-state \rangle (if any) which is or which \texttt{follows} \texttt{p} in the \langle block-state-list \rangle except the current \langle block-state \rangle, replace its \langle statement-control \rangle by a

\langle statement-control \rangle:
\langle operation-list \rangle:
\langle operation \rangle for epilogue.

Step 3. Perform epilogue.
6.3.10 EXECUTE-END-STATEMENT

Operation: \texttt{execute-end-statement(cu)}

\hspace{1cm} where \texttt{cu} is an \texttt{end-statement}.

Step 1. Let \texttt{cu} be that \texttt{executable-unit-list} or \texttt{entry-or-executable-unit-list} which contains \texttt{cu} but does not contain any other \texttt{executable-unit-list} or \texttt{entry-or-executable-unit-list} which also contains \texttt{cu}.

Let \texttt{n} be the node which immediately contains \texttt{cu}.

Case 1.1. \texttt{n} is a \texttt{procedure}.

\hspace{1cm} Step 1.1.1. The \texttt{entry-point} designated by the \texttt{entry-point-designator} component of the current \texttt{linkage-part} must not contain a \texttt{return-descriptor}.

\hspace{1cm} Step 1.1.2. If the \texttt{block-state-list} contains only one \texttt{block-state} then perform \texttt{raise-condition}; \texttt{finish-condition}.

\hspace{1cm} Step 1.1.3. Perform epilogue.

Case 1.2. \texttt{n} is a \texttt{begin-block}.

\hspace{1cm} Perform epilogue.

Case 1.3. \texttt{n} is a \texttt{non-iterative-group}.

\hspace{1cm} Step 1.3.1. Set the current \texttt{executable-unit-designator} to designate the \texttt{executable-unit} which simply contains \texttt{n}.

\hspace{1cm} Step 1.3.2. Perform normal-sequence.

Case 1.4. \texttt{n} is a \texttt{while-only-group}.

\hspace{1cm} Set the current \texttt{executable-unit-designator} to designate the \texttt{executable-unit} which simply contains \texttt{n}.

Case 1.5. \texttt{n} is a \texttt{controlled-group}.

\hspace{1cm} Set the current \texttt{executable-unit-designator} to designate the \texttt{executable-unit} that simply contains \texttt{n}; perform \texttt{test-termination-of-controlled-group} to obtain \texttt{t}.

Case 1.5.1. \texttt{t} is \texttt{true}.

\hspace{1cm} Perform normal-sequence.

Case 1.5.2. \texttt{t} is \texttt{false}.

\hspace{1cm} Set the current \texttt{executable-unit-designator} to designate the first \texttt{executable-unit} of \texttt{cu}.
6.4 Conditions and Interrupts

There are two distinct concepts of "condition" and "interrupt". When a "condition" occurs, e.g. raise-condition((overflow-condition)) is performed, it may lead to an "interrupt", i.e. invocation of the interrupt operation.

The circumstances in which the various "conditions" occur are defined throughout Chapters 6 to 9 at the appropriate points, wherever the operation raise-condition is to be performed. This section defines how the occurrence of a "condition" may also be signalled explicitly, and how the operations raise-condition, interrupt, and system-action, are performed.

6.4.1 CONDITIONS

6.4.1.1 Raise-condition

A condition may be "raised" either implicitly from circumstances defined elsewhere, or explicitly by the execution of a <signal-statement>. In either case, the operation test-enablement is used to determine whether the "condition" is enabled and hence to determine whether the operation interrupt is to be performed.

Operation: raise-condition(c, chfs)

where c is an <evaluated-in-condition>, a <programmer-named-condition>, or a terminal node of <condition-name> apart from the <named-in-condition> or <programmer-named-condition>,

chfs is a <condition-tail-value-list>.

Step 1. There must exist at least one <block-state>.

Step 2. If c is one of the terminal nodes of <computational-condition>, then perform test-enablement() to obtain ch, which must not be <disabled>.

Step 3. Let cc be a copy of chfs. If c is a <conversion-condition> and the current <block-controls> contains a <current-file-value>, <file-value>, fcv, and cc does not contain an <enfile-value>, then attach to cc an <enfile-value>; fc; where fc is the <character-string-value> in the <filename> in fc, where fc is the <file-information> designated by fcv.

Step 4. Perform interrupt(c, cc).

6.4.1.2 Test-enablement

Operation: test-enablement(c)

where c is one of the terminal nodes of <computational-condition>.

result: <enabled> or <disabled>.

Step 1. Let ec be the <executable-unit> designated by the current <executable-unit-designator>.

Case 1.1. The current <linkage-part> does not contain a <prologue-flag> and the current <block-control> does not contain a <remote-block-state>.

Let tp be ec.

Case 1.2. The current <linkage-part> does not contain a <prologue-flag> and the current <block-control> contains a <remote-block-state>, rhs.

Let fc be the last <format-control> of the current <format-control-list> that contains a <remote-block-state>. (This <remote-block-state> equals rhs.) Let tp be the <format-statement> designated by the <format-statement-designator> of fc.

Chapter 5: Flow of Control 169
Case 1.1. The current <linkage-part> contains a <prologue-flag>.

Let tp be the <procedure> or <begin-block> of which ss is a block-component.

Step 2. If tp is a <begin-block> then let tp be the <executable-unit> immediately containing tp.

Step 3. If tp immediately contains a <condition-prefix-list>, cpl, and if cpl contains a
<condition-prefix>, cp, containing a <computational-condition> equal to
<computational-condition>; c; then return the second component of cpl.

Step 4.

Case 4.1. There exists a block, b, which has tp as block-component.

Let tp be b and go to Step 2.

Case 4.2. There is no such block.

If c is <size-condition>, <stringrange-condition>, or <subscriptrange-condition>, then return <disabled>; otherwise return <enabled>.

4.9.1.3 Execute-signal-statement

Operation: execute-signal-statement(ss)

where ss is a <signal-statement>.

Step 1. Let cn be the <condition-name> immediate component of ss and let c be the
<named-to-condition> or <programmer-named-condition> or otherwise the terminal
subnode, of cn. If c is not one of the terminal nodes of <computational-
condition>, then let r be <enabled>. Otherwise perform best-enablement(c) to
obtain r.

Case 1.1. r is <disabled>.

Perform normal-sequence and terminate this operation.

Case 1.2. r is <enabled>.

Case 1.2.1. c is <conversion-condition>.

Let chfs be a

<condition-bit-value-list>
<condition-bit-values>
<onsource-values>
<character-string-values>
<multi-character-string-values>
<condition-bit-values>
<onchar-values>
<integer-values>: 0.

Case 1.2.2. c contains <same-condition>.

Let chfs be a

<condition-bit-value-list>
<condition-bit-values>
<onfield-values>
<character-string-values>
<multi-character-string-values>.
Case 1.2.3. c contains <key-condition>

Let chfs be a

<condition-bif-value-list>
  <condition-bif-value>
  <key-value>
  <character-string-value>
  <null-character-string>

Case 1.2.4. (Otherwise).

Let chfs be <empty>.

Step 2. If c is a <named-io-condition> then perform evaluate-named-io-condition(c) to obtain eo; otherwise let eo be c.

Step 3. Perform interrupt(ec, chfs).

Step 4. Perform normal-sequence.

4.9.1.4 Evaluate-named-io-condition

Operation: evaluate-named-io-condition(nloc)

where nloc is a <named-io-condition>.

result: an <evaluated-io-condition>.

Step 1. Let vr be the <value-reference> immediate component of nloc. Perform evaluate-file-option(vr) to obtain a <file-value>, f.

Step 2. Let loc be the <io-condition> component of nloc. Return <evaluated-io-condition>, loc f.

4.9.2 INTERRUPTS

The <on-statement> and <remove-statement> may be used to influence the action taken on the occurrence of an interrupt operation. First these statements are described, and then the operation interrupt itself is defined.

4.9.2.1 Execute-on-statement

Operation: execute-on-statement(os)

where os is an <on-statement>.

Step 1. For each <condition-name>,on in the <condition-name-list> component of os taken in left-to-right order perform Steps 1.1 through 1.5.

Step 1.1.

Case 1.1.1. on has <named-io-condition>, nic.

Perform evaluate-named-io-condition(nic) to obtain an <evaluated-io-condition>, nic. Let ec be an <evaluated-condition>, nic.

Case 1.1.2. on does not have <named-io-condition>.

Let cal be the immediate subtree of ec. Let ec be an <evaluated-condition>, cal.

Step 1.2. If the current <established-on-unit-list> contains an <established-on-unit>, eso containing ec, then delete eso.

Chapter 4: Flow of Control 171
Step 1.3.

Case 1.3.1. On contains an <on-unit>,On.

Let spd be an <entry-point-designator> designating the <entry-point> of on. Let es be an
<entry-value>:
  spd
<block-state-designator> designating the current <block-state>.

Case 1.3.2. (Otherwise):

Let es be <system-action>.

Step 1.4. Let nee be
<established-on-unit>:
  es:
  on.

Step 1.5. If on contains <map>, attach <map> to nee. Append nee to the current
<established-on-unit-list>.

Step 2. Perform normal-sequence.

6.4.2.2 Execute-revert-statement

Operation: execute-revert-statement(rs)

where rs is a <revert-statement>.

Step 1. Let cal be the <condition-name-list> immediate component of rs. For each
<condition-name>,c in cal taken in left-to-right order perform Steps 1.1 and
1.2.

Step 1.1.

Case 1.1.1. c has <named-io-condition>,etc.

Perform evaluate-named-io-condition(etc) to obtain an <evaluated-io-
condition>,etc. Let ec be an <evaluated-condition>: etc.

Case 1.1.2. c does not have a <named-io-condition>.

Let ec be an <evaluated-condition>: the immediate component of c.

Step 1.2. If the current <established-on-unit-list> contains an <established-on-unit>,
ec containing ec, then delete ec.

Step 2. Perform normal-sequence.

6.4.3 INTERRUPT

Operation: interrupt(s,cf)

where c is an <evaluated-io-condition>, a <programmer-named-condition>, or a
terminal node of <condition-name> apart from the <named-io-
conditions> or <programmer-named-conditions>,

s,cf is a {condition-bit-value-list}.

Step 1. Let be be the current <block-state>:
Step 2. Append to chfs each of the following components whose immediate subtree it does not already possess for let chfs be a <condition-bif-value-list> with these components if chfs is <absent>

- <condition-bif-value>
  - <code-value>
    - <integer-value> an implementation-defined integer;

- <condition-bif-value>
  - <class-value>
    - the <character-string-value> forced by finding the last <block-state> of the <block-state-list> which has an <entry-point-designator>, and taking the <identifier> of the <statement-name> of the <entry-point> designated by it;

and if c is an <evaluated-to-condition> and chfs does not contain an <if-file-value>

- <condition-bif-value>
  - <file-value>
    - the <character-string-value> in the <filename> of the <file-information> designated by the <file-value> of c.

Step 3. Let c be the <established-on-unit-list> of b.

Case 3.1. c contains an <established-on-unit>, e, which contains an <evaluated-condition> with a subtree equal to c, or c is a <programmer-named-condition> such that the following conditions are true:

1) c designates a <declaration> containing <external>, and

2) c contains an <established-on-unit>, e, which contains a <programmer-named-condition> designating a <declaration> equal to e.

Step 3.1.1. If c contains <error>, then output implementation-defined information (e.g., a list of names of currently active blocks) by an implementation-dependent means.

Step 3.1.2.

Case 3.1.2.1. c contains an <entry-value>, ev.

Let ev be an <evaluated-entry-reference>; ev. Perform activate-procedure(eer, chfs).

Case 3.1.2.2. c contains <system-action>.

Perform system-action(c, chfs).

Case 3.2. (Otherwise).

Case 3.2.1. b is not the first <block-state> of the <block-state-list>, b1.

Let b1 be the immediately preceding <block-state> of b1 and go to Step 3.

Case 3.2.2. (Otherwise).

Perform system-action(c, chfs).

Step 4. c must not be <error-condition>, <timed-overflow-condition>, <overflow-condition>, <size-condition>, <string-range-condition>, <subscript-range-condition>, or <zero-division-condition>.
§4.9 SYSTEM-ACTION

For every `<evaluated-condition>`, there is the possibility, as defined in Section §4.3, that an interrupt operation for it will lead to the operation system-action.

Operation: `system-action(c, chfs)`

where `c` is an `<evaluated-loc-condition>`, a `<programmer-named-condition>`, or a terminal node of a `<condition-name>` apart from the `<named-loc-condition>`s or `<programmer-named-condition>`s, `chfs` is a `<condition-hlf-value-list>`.

Case 1. `c` is `<finish-condition>` or `<utrimsize-condition>`.

Terminate this operation.

Case 2. `c` is `<programmer-named-condition>` or `<underflow-condition>`, or contains `<name-condition>`.

Perform comment.

Case 3. `c` contains `<endpage-condition>`.

Let `fv` be the `<file-value>` component of `c`. Perform put-page(fv) (see Section 8.7.2.122).

Case 4. `c` is `<error-condition>` or `<storage-condition>`.

The action is implementation-defined.

Case 5. (Otherwise).

Step 5.1. Perform comment.

Step 5.2. Perform raise-condition(<error-condition>, chfs).

§4.9.1 Comment

Operation: `comment`

Output implementation-defined information by an implementation-dependent means.
Chapter 7: Storage and Assignment

7.0 Introduction

This chapter defines all the operations of the PL/I Machine that change the allocated-storage of the machine-state. The main sections are:

7.1 The Generation
7.2 The Allocation of Storage
7.3 Initialization
7.4 The Freeing of Storage
7.5 Assignment
7.6 Variable-reference
7.7 Reference to Named Constant

The allocated-storage consists of an allocation-unit-list. Elements are appended to this list by the allocate operation; this may be invoked either during the execution of an allocate-statement, read-statement, or locate-statement, or directly by any of the operations allocate-static-storage-and-build-static-directory, establish-argument, or prologue. An allocation-unit is deleted from the list by the free operation; this may be invoked either during the execution of a free-statement, read-statement, or locate-statement, or directly by the epilogue operation.

An area-value also contains an allocation-unit-list; elements may be added to this list by the reallocate operation during the execution of an allocate-statement and may be deleted by the free-based-storage operation during the execution of a free-statement.

Elements of the basic-value-list of an allocation-unit are changed by the set-storage operation; this may be invoked during the execution of the assignment-statement, the group, the read-statement, the get-statement, or during initialization.

7.1 The Generation

The evaluation of a variable-reference yields a generation; a pointer-value is also a generation. A generation describes some or all of the elements of the basic-value-list of the allocation-unit designated by the allocation-unit-designator of the generation. The elements of the storage-index-list component of the generation specify which elements of the basic-value-list of the allocation-unit are being described. Each such element is a scalar-element.

The evaluated-data-description component of a generation contains a data-description where each extent-expression contains only an integer-value.

7.1.1 The number of elements in the storage-index-list of a generation

Let the evaluated-data-description of the generation, q, have the immediate data-description component d.d. Since d.d is contained in an evaluated-data-description, each bound-pair of d.d has two extent-expressions which contain only integer-values; that is, no bound-pair contains an asterisk or expression with a variable-reference. The number of elements of the storage-index-list of q is the number of scalar-elements corresponding to d.d. This number is determined by the operation scalar-elements-of-data-description.
Operation:  scalar-elements-of-data-description(dd)

where dd is a <data-description>,

result: an integer.

Case 1. dd is of the form <data-description>; <item-data-description>.

return the integer 1.

Case 2. dd is of the form <data-description>; <structure-data-description>; <member-data-list>, mdd.

For each tree of the form <member-data-description>; <data-description>, dd(iii), in mdd, i=1,...,m, perform scalar-elements-of-data-description(dd(iii)) to obtain an integer, n(iii). Return the integer

\[ \sum_{i=1}^{m} n(iii). \]

Case 3. dd is of the form <data-description>:

<dimensioned-data-description>
<element-data-description>,edd
<brand-pair-list>, bpl.

Let ic be the immediate component of edd. Let dd be a <data-description>; ic.
Perform scalar-elements-of-data-description(dd) to obtain the integer n. For each tree of the form <brand-pair>, bpl(i) in bpl, i=1,...,n, let sh(i) and lb(i) be the <integer-value> components of the <upper-bound> and <lower-bound> respectively of bpl(i). Return the integer

\[ n \times \prod_{i=1}^{n} (sh(i)-lb(i)+1). \]

7.1.2 CORRESPONDENCE BETWEEN AN ITEM-DATA-DESCRIPTION AND A BASIC-VALUE

There is a correspondence between an element of the <storage-index-list> of a <generation> and a <basic-value> in the <allocated-storage>. There is also a correspondence between a <storage-index> and an <item-data-description> of the <evaluated-data-description> of the <generation>. In general, this is a many-to-one correspondence defined by the operation find-item-data-description which finds the <item-data-description> of a <data-description> that corresponds to a given element of the <storage-index-list>.

Operation:  find-item-data-description(dd, ord)

where dd is a <data-description>,
ord is the ordinal of an element of a <storage-index-list>,

result: an <item-data-description>.

Case 1. The immediate component of dd is an <item-data-description>.

return this <item-data-description>.

Case 2. The immediate component of dd is a <structure-data-description>.

let mdd be this <structure-data-description>. Let sdd be the <member-data-list> of mdd. Let nill be the number of scalar-elements corresponding to the i'th element of sdd, obtained by performing scalar-elements-of-data-description(dd(iii)) where dd(iii) is the <data-description> of the i'th element of sdd. Let sm(i) be 0. Let nill be the sum of the nii for the first i elements of sdd. Let j be such that sm(j)-ord.sm(j) ≥ 0. Perform find-item-data-description(dd(iii), k), where k = ord.sm(j) - 1, to obtain an <item-data-description>, i'dd. Return i'dd.
Case 1. The immediate component of dd is a <dimensioned-data-description>.
   Let dd be this <dimensioned-data-description>.

Case 3.1. The <element-data-description> of dd has an <item-data-description> as the
           immediate component.

           Return a copy of this <item-data-description>.

Case 3.2. (Otherwise).
           Let dd be a <data-description> immediately containing a copy of the
           <structure-data-description> of dd. Perform scalar-elements-of-data-
           description(dd) to obtain an integer n. Perform find-item-data-
           description(dI,k+1), where k is the value of the remainder obtained when
           ord-1 is divided by n, to obtain an <item-data-description>, ida. Return
           ida.

7.1.3 VALUE OF A GENERATION

Operation: value-of-generation(g)
   where g is a <generation>.
   result: an <aggregate-value>

Step 1. The <allocation-unit> designated by the <allocation-unit-designator> of g must
         be contained in <allocated-storage>.

Step 2. Let av be an <aggregate-values>.

Step 3. Let dd be the <data-description> of g, with associated <aggregate-type>, agt.
         Attach agt to av.

Step 4. Let n be the number of elements in the <storage-index-list> of g, and let v be
         the <basic-value-list> of the <allocation-unit> designated by the
         <allocation-unit-designator> of g.

Step 5. For i=1,...,n, perform Steps 5.1 through 5.3.

Step 5.1. Perform find-item-data-description(dd,i) to obtain an <item-data-
          description>, id. Let d be the <data-type> of id.

Step 5.2. Let p be the i'th <storage-index> immediately contained in g.

Step 5.3. Perform value-of-storage-index(p,d,v) to obtain a <basic-value>, bv. Append
          bv to the <basic-value-list> of av.

Step 6. Return av.
7.1.4 Value of Storage Index

**Operation:** value-of-storage-index(p,d,v)

- where p is a <storage-index>
- d is a <data-type>
- v is a <basic-value-list>

**Result:** a <basic-value>

**Step 1.** Let l be the <basic-value-index> of p.

**Step 2.** Let vi will be the immediate component of the l'th <basic-value> of v.

**Case 2.1.** d does not have <string>, <convarying>, or <picture>.

Let e be a copy of vi.

**Case 2.2.** d has <string>, <convarying>, or <picture>.

Let N be the <maximum-length> component of d (if d has <string>) or the associated character-string length of d (if d has <picture>).

**Case 2.2.1.** N = 0.

Let e be the <null-character-string> (if d has <character> or <picture>) or the <null-bit-string> (if d has <bit>).

**Case 2.2.2.** N > 0.

Let M be the <position-index> component of p. Then there are integers j and k such that:

1. \[ i \leq j \leq k \text{ (i=j=k is possible)} \]
2. \[ vi_1, vi_1+1, \ldots, vi_1 \] are either all <bit-string-values> or all <character-string-values>.

\[ \sum_{u=1}^{j-1} \text{LIST}_u \leq M \leq \sum_{u=1}^{j} \text{LIST}_u, \text{ and} \]

\[ N+M-1 \leq \sum_{u=1}^{j} \text{LIST}_u. \]

Here, \( \text{LIST}_u \) is the length of \( vi_1 \). Let e be a <character-string-value> (if vi1 is a <character-string-value>) or a <bit-string-value> (if vi1 is a <bit-string-value>). The length of e is N. The M components of e are the same M successive components of vi1, vi1+1, ..., vi1 beginning at component number \( M-\text{LIST}_1+\text{LIST}_2+\ldots+\text{LIST}_{j-1} \) in vi1.

**Step 3.** Return a <basic-value> containing a copy of e.
DECLARE 1 A AUTOMATIC SEMICYCLED,
2 B CHARACTER(31),
2 C CHARACTER(31),
2 D CHARACTER(31),
X CHARACTER(4) DEFINED & POSITION(23),
Y CHARACTER(6) DEFINED & POSITION(13);
A.B = '123';
A.C = '5878';

After the assignments, the <generations> corresponding to \( A \) looks like

\[
\begin{array}{ccc}
1 & 2 & 3 \\
\hline
v(1) \quad v(2) \quad v(3)
\end{array}
\]

Here \( v(i) \) is the \( i \)th component of the <basic-value-list>, a digit represents the corresponding character, and an empty slot represents <undefined>. Evaluation of a <variable-reference>, \( X \) yields a <scalar> <generation> accessing the <generation> of \( A \) as follows:

\[
\begin{array}{ccc}
1 & 2 & 3 \\
\hline
 & 4 & 5 & 6 & 7 & 8 \\
\hline
X
\end{array}
\]

Hence a <value-reference>, \( X \) yields '2345'. Evaluation of a <variable-reference> \( Y \) yields a <scalar> <generation> accessing the <generation> of \( A \) as follows:

\[
\begin{array}{ccc}
1 & 2 & 3 \\
\hline
 & 4 & 5 & 6 & 7 & 8 \\
\hline
Y
\end{array}
\]

The value of the <generation> corresponding to \( Y \) has <undefined> as its fourth, fifth, and sixth components. Therefore, evaluation of a <value-reference>, \( Y \) at this point would be in error.

Example 7.1. An Example of <generations> and <basic-values> of Defined Variables.
7.2 The Allocation of Storage

7.2.1 EXECUTE-ALLOCATE-STATEMENT

This operation causes the construction of an <allocation-unit> corresponding to the <declaration> designated by the <declaration-designator>. Under certain circumstances, the <storage-condition> or the <area-condition> may be raised.

Operation: execute-allocate-statement (ast)

where ast is an <allocate-statement>.

Step 1. For each <allocation>.ai, in the <allocation-list> of ast, chosen in left-to-right order, perform steps 1.1 through 1.3.

Step 1.1. Let d be the <declaration> designated by the <declaration-designator> of ai.

Step 1.2.

Case 1.2.1. The <storage-class> of d contains <controlled>.

Perform allocate-controlled-storage(ai) to obtain the <generation>, g.

Case 1.2.2. The <storage-class> of d contains <base>.

Perform allocate-based-storage(ai) to obtain the <generation>, g.

Step 1.3. If d has an <initial> component, perform initialize-generation(g, d).

Step 2. Perform normal-sequence.

7.2.2 ALLOCATE-CONTROLLED-STORAGE

This operation causes the allocation of storage for a <controlled> variable and records the allocation in the <controlled-directory>.

Operation: allocate-controlled-storage(ai)

where ai is an <allocation>.

result: a <generation>.

Step 1. Let d be the <declaration> designated by the <declaration-designator>, dp, of ai. Make a copy, dd, of the <data-description> of d.


Step 3. Perform allocate-edd to obtain the <generation>, g.

Step 4. Perform find-directory-entry(dp) to obtain the <controlled-directory-entry>, e, for the declaration, d.

Step 5. Append g to the <generation-list> component of e.

Step 6. Return a copy of g.
7.2.3 ALLOCATE-BASED-STORAGE

This operation causes the allocation of storage for a <base> variable and the assignment of the resulting <generation> to a <locator> variable.

Operation: allocate-based-storage(al)
where al is a <allocation>.
result: a <generation>.

Step 1. Let d be the <declaration> designated by the <declaration-designator> of al.

Step 2.
Case 2.1. al has no <set-option> component.

The <base> component of d must have the component <value-reference> that immediately contains a <variable-reference>. Let vrs be this <variable-reference>.

Case 2.2. al has a <set-option> component.

Let vrs be the <variable-reference> of the <set-option> of al.

Step 3. Perform evaluate-variable-reference(vrs) to obtain a <generation>, $g$.

Step 4. Perform evaluate-in-option(al,vrs). If the allocation is to be made in an area, a <generation> with an <area> component will be obtained; let this be $g_1$; otherwise the value <fail> will be obtained.

Step 5. Make a copy, $d_0$, of the <data-description> immediately contained in the <variable> of $d$. Perform evaluate-data-description-for-allocation($d_0$) to obtain an <evaluated-data-description>, $e_d$.

Step 6.
Case 6.1. A <generation>, $g_1$, was obtained in Step 4.

Make an allocation for $d_0$ in the area <generation>, $g_1$, by performing suballocate($d_0$, $g_1$). The result obtained will be either the <generation>, $g$, or the value <fail>. In the latter case, perform evaluate-condition(<area-condition>); on normal return go to Step 4.

Case 6.2. <fail> was obtained in Step 4.

Make an allocation for $d_0$ in the <allocated-storage> by performing allocate($d_0$) to obtain the <generation>, $g$.

Step 7. Perform steps 7.1 and 7.2 in either order.

Step 7.1. Let av be an

<aggregate-value>
<aggregate-type>
<scalar>
<basic-value-list>
<basic-value>
.pointer-value>

Let ddp be of the form <data-description>, <item-data-description>, <pointer>. Let egs be <evaluated-target>, $g$s. Perform assign(egs,av,ddp).

Step 7.2. Perform initialize-refer-options(g) to carry out the initializations of $g$ specified by each <refer-option> in the <evaluated-data-description> of $g$.

Step 8. Return $g$. 

Chapter 7: Storage and Assignment 181
7.2.4 EVALUATE-IN-OPTION

If an allocation is to be made in an area, this operation yields the area <generation> in which the allocation is to be made.

Operation: \texttt{evaluate-in-option(al, vr)}

where \texttt{al} is an <allocation>, \texttt{vr} is a <variable-reference>.

result: a <generation> or <fail>.

Step 1. Let \texttt{ds} be the <declaration> designated by the <declaration-designator> component of \texttt{vr}. \texttt{ds} is the <declaration> for the locator that will be used to identify the <allocation-unit> that will result from the <allocation>, \texttt{al}.

Step 2. If the component <locator> of \texttt{ds} has <offset> and this has the submode <variable-reference>, then let this <variable-reference> be \texttt{wro}.

Step 3.

Case 3.1. \texttt{al} has an <in-option> component.

Let the <variable-reference> of the <in-option> be \texttt{vri}.

Case 3.2. \texttt{al} has no <in-option> component.

Case 3.2.1. \texttt{ds} has an <offset> component.

\texttt{wro} must have been created in Step 2. That is, the area base for the offset must have been specified. Let \texttt{vri} be the same as \texttt{wro}.

Case 3.2.2. \texttt{ds} has a <pointer> component.

Return the value <fail>, since the <allocation-unit> for \texttt{al} is to be constructed in the <allocated-storage>.

Step 4. Perform \texttt{evaluate-variable-reference(vr)} to obtain the <generation>, \texttt{gia}.

Step 5. If \texttt{wro} has been created in Step 2, then perform \texttt{evaluate-variable-reference(wro)} to obtain the <generation>, \texttt{gib}. \texttt{gib} must be the same as \texttt{gia}.

Step 6. Return \texttt{gia}.

7.2.5 ALLOCATE

This operation adds an <allocation-unit> that is described by an <evaluated-data-description> to the <allocated-storage>. The result of the operation is the <generation> that identifies the new <allocation-unit>.

Operation: \texttt{allocate(edd)}

where \texttt{edd} is an <evaluated-data-description>.

result: a <generation>.

Step 1. Perform either Step 1.1 or Step 1.2.

Step 1.1. Perform \texttt{raise-condition(storage-condition)}). Go to Step 1.

Step 1.2. Perform \texttt{make-allocation-unit(edd)} to obtain an <allocation-unit>, \texttt{as}.

Step 2. Append \texttt{as} to <allocated-storage>.

Step 3. Perform \texttt{scalar-elements-of-data-description(d1, d2, \ldots)} to obtain \texttt{s}, where \texttt{d1} is the <data-description> of \texttt{edd}. Let \texttt{i} be 1. Construct a <storage-index-list>, \texttt{sil}, by performing Steps 3.1 through 3.3 \texttt{i} times.
Step 3.1. Perform \texttt{find-item-data-description(odd, i))} to obtain an \texttt{<item-data-description>.odd}.

Step 3.2.

Case 3.2.1. \texttt{odd} contains either a \texttt{<picture>} component or a \texttt{<string>} component that has \texttt{<nonvarying>}.

Construct a \texttt{<storage-index>.ai}, that contains a \texttt{<basic-value-index>} that has an \texttt{<integer-value>} equal to 1 and also a \texttt{<position-index>} that has an \texttt{<integer-value>} equal to 1.

Case 3.2.2. (Otherwise).

Construct a \texttt{<storage-index>.ai}, that contains a \texttt{<basic-value-index>} that has an \texttt{<integer-value>} equal to 1.

Step 3.3. Append \texttt{ai} to \texttt{a1} and set \texttt{i} to \texttt{i+1}.

Step 4. Return the \texttt{<generation>} constructed from an \texttt{<allocation-unit-designator> designating ax, ay, az, and a1}.

3.2.6 SUBALLOCATE

This operation constructs an \texttt{<area-allocation>} and adds it to the \texttt{<area-allocation-list> of an area <generation>}. The result of this operation in the \texttt{<generation>} that identifies the new \texttt{<area-allocation>}.

Operation: \texttt{suballocate(odd, g)}

where \texttt{odd} is an \texttt{<evaluated-data-description>},

\texttt{g} is a \texttt{<generation>}.

result: a \texttt{<generation>} or \texttt{<fail>}.

Step 1. The value of \texttt{g} must not be \texttt{<undeclared>}.

Step 2. Perform either Step 2.1 or Step 2.2.

Step 2.1. Return the value \texttt{<fail>}.

Step 2.2. Perform \texttt{make-allocation-unit(odd)} to obtain an \texttt{<allocation-unit>.au}.

Step 3. Perform \texttt{value-of-generation(g)} to obtain an \texttt{<aggregate-value>.av} whose \texttt{<aggregate-type>} must immediately contain \texttt{<scalar>} and whose \texttt{<basic-value-list> contains an <area-value>.av}. If \texttt{av} immediately contains \texttt{<empty>} then replace \texttt{av} by

\texttt{<area-value>.av}:
\texttt{<area-allocation-list>}
\texttt{<significant-allocation-list>}

Append a

\texttt{<significant-allocation>}:
\texttt{odd}
\texttt{<occupancy>}
\texttt{<allocated>}:

to the \texttt{<significant-allocation-list> of av}. Append an \texttt{<area-allocation>: sai au} to the \texttt{<area-allocation-list> of av}.
Step 4. Perform scalar-elements-of-data-description-find to obtain a, where d1 is the data-description of odd. Let i be 1. Construct a (storage-index-list),s1i, by performing Step 4.1 through 4.3 n times.

Step 4.1. Perform (find-item-data-description-item-id) to obtain an (item-data-description),id1.

Step 4.2.

Case 4.2.1. id1 contains either a (picture) component or a (string) component that has <containing>.

Construct a (storage-index),si, that contains a (basic-value-index) that has an (integer-value) equal to j and also a (position-index) that has an (integer-value) equal to 1.

Case 4.2.2. (Otherwise).

Construct a (storage-index),si, that contains a (basic-value-index) that has an (integer-value) equal to 1.

Step 4.3. Append si to s1i and set i to i+1.

Step 5. Return the (generation) constructed from an (allocation-unit-designator) that designates an, odd, and s1i.

7.2.7 EVALUATE-DATA-DESCRIPTION-FOR-ALLOCATION

This operation forms an (evaluated-data-description) from a (data-description) by evaluating each (extent-expression).

Operation: evaluate-data-description-for-allocation(dd)

where dd is a (data-description).

result: an (evaluated-data-description).

Step 1. Let odd be a copy of the (data-description),dd. For each (extent-expression) in odd that immediately contains an (expression), perform Step 1.1, with the (extent-expression) chosen in any order.

Step 1.1. Let the chosen (extent-expression) be ee. Evaluate ee by performing evaluate-expression-to-integer(ee) where ee is the (expression) of ee, to obtain the value i. The (expression) of the (extent-expression) is replaced by an (integer-value) with value i.

Step 2. In each (bound-pair) of odd, the value of the (integer-value) of the (upper-bound) must be greater than or equal to the value of the (integer-value) of the (lower-bound).

Step 3. In each (max-length) component of odd, the value of the (integer-value) must be greater than or equal to zero.

Step 4. In each (area-size) component of odd, the value of the (integer-value) must be greater than or equal to zero.

Step 5. For each (refer-option),ro, in odd perform Step 5.1.

Step 5.1. Let ro be the (member-description) in odd that ro references as in Step 2.7 of validate-based-declaration. Let n be an (integer-value) such that the result of (find-item-data-description-item-id) is equal to the (item-data-description) in ro. Replace the (identifier-list) component of ro by n.

Step 6. Return the (evaluated-data-description), odd.
7.2.8 FIND-DIRECTORY-ENTRY

This operation searches the appropriate <machine-state> directory for an entry corresponding to a declaration.

Operation: find-directory-entry(dp)

where dp is a <declaration-designator> designating a <declaration>.d.

result: a <static-directory-entry>, or a <controlled-directory-entry>, or an <automatic-directory-entry>, or a <parameter-directory-entry>, or a <defined-directory-entry>.

Step 1. Let t be the <storage-type> of d.

Step 2.

Case 2.1. t contains <automatic>, <parameter>, or <defined>.

Perform find-block-state-of-declaration(dp) to obtain a <block-state>,n. If at contains <automatic> or <defined> then n must not contain a <prologue-flag> in its <linkage-parts>. Find the directory entry, e, in the <automatic-directory>, <parameter-directory>, or <defined-directory> of n, as appropriate, such that the <identifier> of e is equal to the <identifier> of d.

Case 2.2. t contains <static> or <controlled>.

Case 2.2.1. The <scope> of d has an <external> component.

Search the <static-directory> or the <controlled-directory>, as appropriate, for the directory entry e whose <identifier> is equal to the <identifier> of d and that has an <external> component.

Case 2.2.2. The scope of d has an <internal> component.

Let e be the <static-directory-entry> or <controlled-directory-entry>, as appropriate, which contains a <declaration-designator> equal to dp.

Step 3. Return e.

7.2.9 MAKE-ALLOCATION-UNIT

This operation forms an <allocation-unit> corresponding to an <evaluated-data-description>.

Operation: make-allocation-unit(odd)

where odd is an <evaluated-data-description>.

result: an <allocation-unit>.

Step 1. Let dd be the simply contained <data-description> component of odd. Perform scalar-elements-of-data-description(dd) to obtain a, the number of scalar-elements that correspond to odd. Construct a <basic-value-list>.svl, by performing Steps 1.1 through 1.2 for i=1,...,a.

Step 1.1. Perform find-item-data-description(dd,i) to obtain the <item-data-description>.idd, that corresponds to the i'th scalar-element of odd.

Step 1.2.

Case 1.2.1. The <mode> of idd contains <complex> and idd does not contain <pictered>.

Append to svl a <complex-value> consisting of two <undefined> components.
Let \( m \) be the associated character-string length of the \(<\text{picture}\>\) component of \( \text{idd} \). Append to \( \text{svl} \) a \(<\text{character-value-list}\>\) consisting of \( m \) elements, each of which is \(<\text{undefined}\>\).

Case 1.2.3. \( \text{idd} \) contains \(<\text{string}\>\) containing \(<\text{overflow}\>\).

Let \( m \) be the \(<\text{integer-value}\>\) contained by the \(<\text{maximum-length}\>\) component of \(<\text{string}\>\).

Case 1.2.3.1. The \(<\text{string}\>\) of \( \text{idd} \) contains \(<\text{character}\>\).

If \( m \) is zero, append a \(<\text{character-string-value}\>\) consisting of the \(<\text{null-character-string}\>\) to \( \text{svl} \); otherwise append a \(<\text{character-string-value}\>\) consisting of a \(<\text{character-value-list}\>\) with \( m \) elements containing \(<\text{undefined}\>\).

Case 1.2.3.2. The \(<\text{string}\>\) of \( \text{idd} \) contains \(<\text{bit}\>\).

If \( m \) is zero, append a \(<\text{bit-string-value}\>\) consisting of the \(<\text{null-bit-string}\>\) to \( \text{svl} \); otherwise append a \(<\text{bit-string-value}\>\) consisting of a \(<\text{bit-value-list}\>\) with \( m \) elements containing \(<\text{undefined}\>\).

Case 1.2.4. \( \text{idd} \) contains an \(<\text{area}\>\) component.

Append an \(<\text{area-value}\>\) \(<\text{empty}\>\) to \( \text{svl} \).

Case 1.2.5. (Otherwise).

Append an \(<\text{undefined}\>\) element to \( \text{svl} \).

Step 2. Return an \(<\text{allocation-unit}\>\) containing \( \text{svl} \).

7.2.10 \text{INITIALIZE-REFER-OPTIONS}:

This operation initializes the object of each \(<\text{refer-option}\>\) in a \(<\text{generation}\>\).

Operation: \text{initialize-refer-options}(\text{g})

where \( \text{g} \) is a \(<\text{generation}\>\) whose \(<\text{aggregate-type}\>\) immediately contains
\(<\text{structure-aggregate-type}\>\).

Step 1. Let \( \text{add} \) be the \(<\text{evaluated-data-description}\>\) of \( \text{g} \) and let \( \text{dd} \) be the \(<\text{data-description}\>\) in \( \text{add} \). For each \(<\text{refer-option}\>\), \( \text{ro} \), of \( \text{add} \), chosen in any order, perform Steps 1.1 and 1.2.

Step 1.1. The \(<\text{refer-option}\>\), \( \text{ro} \), has an \(<\text{integer-value}\>\) index, \( i \), constructed by evaluate-data-description-for-allocation; this identifies the element of \( \text{g} \) that is the object of the \(<\text{refer-option}\>\), \( \text{ro} \). Perform find-item-data
description(\( \text{dd}, i \)) to obtain an \(<\text{item-data-description}\>\), \( \text{idd} \), of the element. Let \( \text{oddr} \) be an \(<\text{evaluated-data-description}\>\) with \( \text{idd} \) as immediate component. Construct the \(<\text{generation}\>\), \( \text{g} \), from \( \text{oddr} \), a copy of the \(<\text{allocation-unit-designator}\>\) of \( \text{g} \) and the \(<\text{storage-index-list}\>\) containing a single element, a copy of the \( i \)-th element of the \(<\text{storage-index-list}\>\) of \( \text{g} \).

Step 1.2. \( \text{ro} \) is a component of an \(<\text{extent-expression}\>\) that has an \(<\text{integer-value}\>\), \( \text{iv} \). Let \( \text{ddl} \) be a \(<\text{data-description}\>\) with integer-type (see Section 3.1.7). Let \( \text{av} \) be of the form
\[
\text{aggregate-value};
\text{aggregate-type};
\text{scalar};
\text{basic-value-list};
\text{basic-value};
\text{integer-value}, \text{iv}.
\]

Let \( \text{ct} \) be \(<\text{evaluated-target}\>\), \( \text{g} \). Perform assign-target(\( \text{av}, \text{ddl} \)) to assign the value of the \(<\text{extent-expression}\>\) to the object of the \(<\text{refer-option}\>\).
7.2.11 FIND-BLOCK-STATE-OF-DECLARATION

Operation: \textit{find-block-state-of-\textit{declaration}}(\textit{dp})

where \textit{dp} is a \textit{<declaration-designator>} designating the \textit{<declaration>}.d.

result: a \textit{<block-state>}

Step 1. Let \textit{bs} be the current \textit{<block-state>}. If the \textit{<block-control>} of \textit{bs} contains the form \textit{remote-block-state},Y, then let \textit{bs} be the \textit{<block-state>} designated by \textit{Y}.

Step 2. Let \textit{bb} be the corresponding block of \textit{bs} (see Section 5.2.2). If \textit{bb} contains \textit{d}, then return \textit{bs}.

Step 3. \textit{bs} must have a \textit{<block-environment>}.Yv. Let \textit{bs} be the \textit{<block-state>} designated by \textit{Yv}. Go to Step 2.
7.3 Initialization

7.3.1 initialize-generation

This operation initializes a <generation> according to the specification contained in a <declaration>.

Operation: \texttt{initialize-generation}(q,d)

where \( q \) is a <generation>, \( d \) is a <declaration>.

Case 1. The immediate component of the <data-description> immediately contained in the <variable> of \( d \) is an <item-data-description>, \( \text{id}_d \).

Perform initialize-scalar-element\((q,\text{id}_d)\).

Case 2. The immediate component of the <data-description> immediately contained in the <variable> of \( d \) is a <dimensioned-data-description> whose <element-data-description> is an <item-data-description>.

Perform initialize-array\((q,d)\).

Case 3. The immediate component of the <data-description>, \( \text{dd} \) immediately contained in the <variable> of \( d \) is either a <structure-data-description>, or a <dimensioned-data-description> whose <element-data-description> is a <structure-data-description>.

Let \( \text{odd} \) be the simply contained <structure-data-description> in \( \text{dd} \). For each <member-data-description> immediately contained in the <member-description-list> of \( \text{odd} \), chosen in any order, that has an <initial> component, not necessarily immediate, perform Steps 3.1 through 3.4.

Step 3.1. Let the chosen <member-data-description>, \( \text{md} \) be the \( i \)th immediate component of the <member-description-list>.

Step 3.2. Perform select-qualified-reference\((q,\text{id},d)\); where \( \text{id} \) is an <identifier-list> consisting of the single <identifier>, \( \text{id} \), that is a copy of the \( i \)th immediate component of the <identifier-list> of \( \text{odd} \), to obtain a modified <generation>, \( q_l \).

Step 3.3. Let \( \text{do} \) be a copy of the <declaration>, \( d \), and let \( \text{odd}_c \) be the copy of \( \text{odd} \) contained in \( \text{dc} \).

Case 3.3.1. The immediate component of \( \text{dd} \) is <dimensioned-data-description> and the <data-description> of \( \text{md} \) has a <dimensioned-data-description>, \( \text{dd}_d \).

Append to the <bound-pair-list> of \( \text{dc} \) the immediate components of the <bound-pair-list> of \( \text{dd}_d \). Replace \( \text{odd}_c \) by the immediate sub-tree of the <element-data-description> in \( \text{dd}_d \); this will always be a <structure-data-description> or an <item-data-description>.

Case 3.3.2. (Otherwise).

Replace \( \text{odd}_c \) by the immediate sub-tree of the <data-description> of \( \text{md} \).

Step 3.4. Perform initialize-generation\((q_l,\text{dc})\).
7.3.2 INITIALIZE-SCALAR-ELEMENT

This operation initializes a generation consisting of a single element.

Operation:  \texttt{initialize-scalar-element}(g, idd)

where \( g \) is a \(<\text{generation}>\),

\( \text{idd} \) is an \(<\text{item-data-description}>\).

Step 1. If \( \text{idd} \) does not contain an \(<\text{initial-element}>\) then terminate this operation.

Step 2. If the \(<\text{initial-element}>\) component of \( \text{idd} \) has an \(<\text{asterisk}>\) then terminate this operation.

Step 3. Perform \texttt{evaluate-expression}(e), where \( e \) is the \(<\text{expression}>\) simply contained in the \(<\text{initial-element}>\) of \( \text{idd} \), to obtain an \(<\text{aggregate-value}>\), \( v \).

Step 4. Create an \(<\text{evaluated-target}>\), \( \text{et} \), and attach \( g \) to it.

Step 5. Perform \texttt{assign}(\( \text{et}, v, \text{ddi} \)), where \( \text{ddi} \) is the \(<\text{data-description}>\) immediately contained in \( e \).

7.3.3 INITIALIZE-ARRAY

This operation initializes a \(<\text{generation}>\) that has an array of \(<\text{basic-value}>\)s.

\(<\text{evaluated-initial-element}>\) := \(<\text{asterisk}>\) | \(<\text{parenthesized-expression}>\) (scalar) |
\(<\text{iteration-factor}>\) | \(<\text{evaluated-iteration-factor}>\)
\(<\text{evaluated-initial-element-list}>\) |
\(<\text{evaluated-initial-item}>\)
\(<\text{element}>\)

\(<\text{evaluated-iteration-factor}>\) := \(<\text{integer-value}>\)

\(<\text{evaluated-initial-item}>\) := \(<\text{basic-value}>\) \(<\text{data-description}>\)

Operation:  \texttt{initialize-array}(g, d)

where \( g \) is a \(<\text{generation}>\),

\( d \) is a \(<\text{declaration}>\).

Step 1. Let \( \text{idd} \) be a copy of the \(<\text{evaluated-data-description}>\) of \( g \) in which the simply contained \(<\text{dimensioned-data-description}>\) has been replaced by the subtree of its \(<\text{element-data-description}>\). If \( d \) does not have an \(<\text{initial-element-list}>\) then terminate this operation. Let \( \text{iel} \) be the \(<\text{initial-element-list}>\) of \( d \).

Step 2. Let \( m \) and \( n \) be 1. Let \( m \) be the number of \(<\text{elements}>\) in the \(<\text{storage-index-list}>\) of \( g \). Let \( n \) be the number of \(<\text{elements}>\) in the \(<\text{initial-element-list}>\), \( \text{iel} \).

Step 3. Construct an \(<\text{evaluated-initial-element-list}, \text{iel} >\), by making a copy of \( \text{iel} \) and replacing each \(<\text{initial-element-list}>\), \( \text{node} \) by an \(<\text{evaluated-initial-element-list}>\) and each \(<\text{initial-element-list}>\), \( \text{node} \) by an \(<\text{evaluated-initial-element}>\) in the copy.

Step 4. Perform Steps 4.1 through 4.7 while \( m \leq n \) and \( n \leq m \).

Step 4.1. Let the \( n \)th element of \( \text{iel} \) be \( \text{iel} \).

Step 4.2. If \( \text{iel} \) immediately contains an \(<\text{iteration-factor}>\), \( \text{itf} \), then perform \texttt{evaluate-expression-to-integer}(e), where \( e \) is the \(<\text{expression}>\) of \( \text{itf} \), to obtain the \(<\text{integer-value}>\), \( v \), and replace \( \text{itf} \) by an \(<\text{evaluated-iteration-factor}>\) containing \( v \).
Step 4.3. If e is immediately contained an <evaluated-initial-element-list>, e1, then for each <expression>, e, contained in e1, chosen in any order, perform Step 4.3.1.

Step 4.3.1.

Case 4.3.1.1. e is immediately contained in an <iteration-factor>, itf.

Optionally perform evaluate-expression-to-integer(e) to obtain an <integer-value>, i, and replace itf by an <evaluated-iteration-factor> containing i.

Case 4.3.1.2. e is immediately contained in a <parenthesized-expression>, pe,

optionally perform evaluate-expression(e) to obtain an <aggregate-value> having a <basic-value>, bv, and replace pe by an <evaluated-initial-item> comprising of bv and a copy of the <data-description> immediate component of pe.

Step 4.4. If e is immediately contains an <evaluated-iteration-factor>, eif, then perform Steps 4.4.1 through 4.4.3.

Step 4.4.1. Let eil be the <evaluated-initial-element-list> of e. Let il be the <integer-value> of eif.

Step 4.4.2.

Case 4.4.2.1. il ≤ 0.

Replace eil by <empty>.

Case 4.4.2.2. (Otherwise).

Let k be the number of elements in eil. Replace eil by the ilk elements formed from k replications of the sequence of elements of eil. Let n be n+ilk-1.

Step 4.4.3. Go to Step 4.1.

Step 4.5. If e is neither an <argument> nor an <address> then perform Steps 4.5.1 and 4.5.2.

Step 4.5.1. If e is a <parenthesized-expression>, pe, then perform evaluate-expression(e), where e is the <expression> of pe, to obtain an <aggregate-value> having the <basic-value>, bv, and replace pe by an <evaluated-initial-item> comprising of bv and a copy of the <data-description> immediate component of pe.

Step 4.5.2. e is an <evaluated-initial-item>. Let v be of the form

<aggregate-value>:
<aggregate-type>:
<scalar-type>:
<basic-value-list>:

where bv is the <basic-value> of e. Let g be the <data-description> immediate component of e. Let s be a copy of the m'th element of the <storage-index-list> of g. Perform assign(evaluated-target>g'[1], v, g), where g is the <generation> comprising of g constructed in Step 1, a copy of the <allocation-unit-designator> of g, and a <storage-index-list> consisting of the single element s.

Step 4.6. If e is not <address> then let n be n+1.

Step 4.7. Let n be n+1.
7.4 The Freeing of Storage

The <free-statement> causes storage allocated for specified <based> or <controlled> variables to be freed.

7.4.1 EXECUTE-FREE-STATEMENT

Operation:  \texttt{execute-free-statement}(fs)

where \texttt{fs} is a <free-statement>.

Step 1.  For each \texttt{<freeing>}, \texttt{fr}, in the \texttt{<freeing-list>} of \texttt{fs}, chosen in left-to-right order, perform Steps 1.1 and 1.2.

Step 1.1.  Let \texttt{d} be the <declaration> designated by the <declaration-designator> component of \texttt{fr}.

Step 1.2.

Case 1.2.1.  The <storage-class> of \texttt{d} contains <controlled>.

Perform \texttt{free-controlled-storage}(fr).

Case 1.2.2.  The <storage-class> of \texttt{d} contains <based>.

Perform \texttt{free-based-storage}(fr).

Step 2.  Perform normal-sequence.

7.4.2 FREE-CONTROLLED-STORAGE

This operation frees the most recent allocation of a <controlled> variable.

Operation:  \texttt{free-controlled-storage}(fr)

where \texttt{fr} is a <freeing>.

Step 1.  Let \texttt{d} be the <declaration> designated by the <declaration-designator>, \texttt{dp}, of \texttt{fr}.

Step 2.  Perform \texttt{find-directory-entry}(\texttt{dp}) to obtain a <controlled-directory-entry>, \texttt{e}, corresponding to \texttt{d}.

Step 3.  If \texttt{e} contains a <generation-list>, \texttt{gl}, perform Steps 3.1 to 3.3.

Step 3.1.  Let \texttt{g} be the last <generation> in \texttt{gl}.

Step 3.2.  Perform \texttt{free}(g).

Step 3.3.  Delete \texttt{g} from \texttt{gl}.
7.4.3 FREE-BASED-SPACE

This operation frees a <base>-variable specified in a <freeing>.

**Operation:** free-based-storage(fr)

where fr is a <freeing>.

**Step 1.** fr can consist of three components:

- a <declaration-designator>, dp
- a <locator-qualification>
- and an <in-option>.

Of these, only dp always exists.

**Step 2.** Let d be the <data-description> of the <declaration> designated by dp. If fr contains a <locator-qualification>, dp then let vr be <variable-reference>; 1q dp d. Otherwise let vr be <variable-reference>; dp d. Let n be the number of <bound-pair> in d. If n is not equal to zero, attach a <subscript-list> containing n occurrences of <asterisk> to vr.

**Step 3.** Perform evaluate-variable-reference(vr) to obtain the <generation>, qf to be freed. Let as be the <allocation-unit> designated by the <allocation-unit-designator> of qf.

**Step 3.1.** If the <data-type> components of d either all contain <character>, <nonvarying>, and <unaligned> or all contain <bit>, <nonvarying>, and <unaligned> then the number of elements in each <character-value-list> or <bit-value-list> in as must equal the corresponding <maximum-length> in the <evaluated-data-description> of qf.

**Step 3.2.** If there are n elements in the <basic-value-list> of as then there must be n elements in the <storage-index-list> of qf and the i'th element of the <storage-index-list> must have a <basic-value-index> that contains an <integer-value> equal to i for all values of i from 1 through n.

**Step 4.** Perform deduce-in-option(fri). If an area containing qf can be inferred from fr, a <generation>, qf, will be obtained; otherwise <fail> will be obtained.

**Step 5.**

**Case 5.1.** qf exists.

**Step 5.1.1.** Let s be the <area-value> referred to by qf. The <area-allocation-list> of s must contain as. Let as be the <area-allocation> containing as and let sal be the <significant-allocation-list> of as. Let n be the number of elements of the <significant-allocation-list> of as. Replace the <allocated> component of the n'th element of sal by < freed >.

**Step 5.1.2.** If as is the only <area-allocation> of s then replace s by <area-value>, <empty>. Otherwise, delete as from the <area-allocation-list> of s and perform Step 5.1.2.1.

**Step 5.1.2.1.** If the last element, sal, of sal contains < freed > then delete sal and go to Step 5.1.2.1.

**Case 5.2.** deduce-in-option returned < fail > in Step 4.

The <allocation-unit> as must be an immediate component of the <allocation-unit-list> of <allocated-storage>. The <generation> qf must not:

1. be equal to a component of any of the following:
   1.1 the <controlled-directory>
   1.2 the <static-directory>

2. for any <block-state>, be equal to a component of
7.4.4 DEFUSE-IN-OPTION

This operation infers, if possible, an area (generation) in which a freeing is to be applied.

Operation: defuse-in-option(fr)

where fr is a <freemap>.
esult: a <generation> or <fail>

Case 1. fr contains an <in-option>.

Let io be the <in-option> of fr. Let vr be the <variable-reference> component of io. Perform evaluate-variable-reference(vr) to obtain the <generation>, gs. Return gs.

Case 2. fr contains no <in-option>.

Step 2.1.

Case 2.1.1. There is a <locator-qualifier> as an immediate subnode of fr.

Let vr be the <value-reference> contained in the <locator-qualifiers>.

Case 2.1.2. There is no <locator-qualifier> as an immediate subnode of fr.

The <declaration-designator> of fr designates a <declaration>, d. The <closed> component of d must have a <value-reference>. Let vr be this <value-reference>.

Step 2.2.

Case 2.2.1. vr contains a <variable-reference> with a <declaration-designator> that designates a <declaration> whose <data-type> contains offset.

The <declaration-designator> of vr designates a <declaration>, dvr. The <offset> component of dvr must have a <variable-reference> component, vra. Perform evaluate-variable-reference(vra) to obtain the <generation>, gs. Return gs.

Case 2.2.2. (Otherwise).

Return the value <fail>.

7.4.5 FREE

This operation frees an <allocation-unit>.

Operation: free(g)

where g is a <generation>.

Step 1. Let au be the <allocation-unit> designated by the <allocation-unit-designator> in g.

Step 2. Delete the <allocation-unit>, au, from the <allocation-unit-list> of <allocated-storage>.
7.5 Assignment

Assignment involves changing <basic-value> components of storage (the common case), components in the <file-information-list> (in the case of the <program-pg>), or components in the <condition-bif-values> of the current <block-state> (in the case of the <pseudo-variable> <execute-pg> and <comsource-pg>). The components to be changed are determined by evaluating a <target-reference> (Section 7.5.2), to obtain an <evaluated-targets> (Section 7.5.3). The actual assignment is effected by the operation assign (Section 7.5.3), which, in general, will involve conversion of <basic-values> and promotion of <aggregate-values>. The full generality of assignment is available through the <assignment-statement> (Section 3.5.1), but other constructions cause invocation of the operation assign.

7.5.1 THE ASSIGNMENT STATEMENT

Operation: execute-assignment-statement(last)

where last is an <assignment-statement>.

Step 1. Perform Steps 1.1 and 1.2 in either order.

Step 1.1. Let tr be the leftmost <target-reference> of the <target-reference-list> of last. Perform evaluate-target-reference(tr) to obtain the <evaluated-targets>, et.

Step 1.2. Let e be the <expression> of last. Perform evaluate-expression(e) to obtain an <aggregate-values>, v.

Step 2. Perform assign(let, v, d), where d is the <data-description> immediately contained in e.

Step 3.

Case 3.1. last contains no unevaluated <target-reference>.

Perform normal-sequence.

Case 3.2. last contains one or more unevaluated <target-reference>.

Let tr be the leftmost unevaluated <target-reference>. Perform evaluate-target-reference(tr) to obtain the <evaluated-targets>, et. Go to Step 2.

7.5.2 TARGET REFERENCES

Attributes: The result <data-description> of a <target-reference> that immediately contains a <variable-reference> is the name of the <variable-reference>. For a <target-reference> that has a <pseudo-variable-reference>, the result <data-description> is given in the description of the <pseudo-variable> (Section 7.5.4).

Operation: evaluate-target-reference(tr)

where tr is a <target-reference>.

result: an <evaluated-targets>.

Case 1. The immediate component of tr is a <variable-reference>, vr.

Perform evaluate-variable-reference(vr) to obtain a <generation>, g. Return an <evaluated-targets>, g.
Case 2. The immediate component of tr is a \(<\text{pseudo-variable-reference}\>,\, pvr\).

Perform \texttt{evaluate-f(x\{i\}, \ldots, x\{n\})}, where \(x\) is the immediate component of the \(<\text{pseudo-variable}\>\) contained by pvr, and \(x\{i\}, \ldots, x\{n\}\) are the \(<\text{argument}\>\)s, if any, in the \(<\text{argument-list}\>\). Let \(g\) be the result of \texttt{evaluate-f(x\{i\}, \ldots, x\{n\})}.

(It will be either a \(<\text{generation}\>\) or an \(<\text{evaluated-pseudo-variable-reference}\>\).

Return an \(<\text{evaluated-target}\>\): \(g\).

### 7.5.2.1 Evaluated Targets

**Operation:** \texttt{value-of-evaluated-target(t)}

where \(t\) is an \(<\text{evaluated-target}\>\).

**result:** an \(<\text{aggregate-value}\>\).

**Case 1.** \(t\) immediately contains a \(<\text{generation}\>\): \(g\).

Perform \texttt{value-of-generation(g)} to obtain an \(<\text{aggregate-value}\>\): \(v\), which must not contain \(<\text{undefined}\>\). Return \(v\).

**Case 2.** \(t\) contains \(<\text{imag-py}\>\) or \(<\text{real-py}\>\).

Let \(y\) be the \(<\text{generation}\>\) in \(t\), and perform \texttt{value-of-generation(y)} to obtain an \(<\text{aggregate-value}\>\): \(x\), which must not contain \(<\text{undefined}\>\). Let \(z\) be the \(<\text{scalar-result}\>\) of performing Steps 1 and 2 of \texttt{imag-bif} (see Section 9.4.4.46) or Steps 1 to 3 of \texttt{real-bif} (see Section 9.4.4.65), respectively, taking:

- the \(<\text{scalar-value}\>\) of \(x\) to be the \(<\text{basic-value}\>\) in \(t\);
- the \(<\text{scalar-result-type}\>\) to be the \(<\text{data-type}\>\) in the \(<\text{generation}\>\) of \(t\), with \(<\text{complex}\>\) replaced by \(<\text{real}\>\).

Return an \(<\text{aggregate-value}\>\) containing \(z\).

**Case 3.** \(t\) contains \(<\text{uchar-py}\>\) or \(<\text{short-int}\>\).

Perform \texttt{uchar-bif} (see Section 9.4.4.55) or \texttt{short-int-bif} (see Section 9.4.4.63), respectively, to obtain an \(<\text{aggregate-value}\>\): \(v\). Return \(v\).

**Case 4.** \(t\) contains \(<\text{float-py}\>\).

The \(<\text{file-value}\>\): \(fv\) in \(t\) must obey the constraints of \texttt{pgm-float-bif}. Step 1 (see Section 9.4.4.62). Perform Steps 2 and 3 of \texttt{pgm-float-bif}.

**Case 5.** \(t\) contains \(<\text{substr-py}\>\).

Perform \texttt{value-of-generation(g)} to obtain an \(<\text{aggregate-value}\>\): \(v\), where \(g\) is the \(<\text{generation}\>\) in \(t\), which must not contain \(<\text{undefined}\>\). Let \(st\) be the first \(<\text{aggregate-value}\>\) in \(t\); let \(lt\) be the second \(<\text{aggregate-value}\>\) in \(t\), if present.

Let \(z\) be the \(<\text{scalar-result}\>\) of performing Steps 1 to 4 of \texttt{substr-bif} (see Section 9.4.4.76), taking:

- the \(<\text{scalar-value}\>\) of \(sv\) (or of \(st\) or of \(let\)) to be the single \(<\text{basic-value}\>\) in \(sv\) (or in \(st\) or in \(let\));
- the \(<\text{scalar-result-type}\>\) to be the \(<\text{data-type}\>\) in the \(<\text{generation}\>\) in \(t\).

Return an \(<\text{aggregate-value}\>\) containing \(z\).

**Case 6.** \(t\) contains \(<\text{complex-py}\>\).

Let \(g\) be the \(<\text{generation}\>\) in \(t\). Perform Step 3 of \texttt{complex-bif} (see Section 9.4.4.85).
7.5.3 The Assignment Operation

Operations: \texttt{assign(et,sv,nd)}

where \( et \) is an \texttt{evaluated-target},
\( sv \) is an \texttt{aggregate-value},
\( nd \) is a \texttt{data-description}.

Case 1. \( et \) immediately contains a \texttt{generations}, \( q \).

Step 1.1. Let \( odd \) be the \texttt{evaluated-data-description} of \( q \). Perform \texttt{promote-and-convert(odd,sv,nd)}, to obtain an \texttt{aggregate-value}, \( av \).

Step 1.2. Perform \texttt{set-storeq(q,svl)} where \( svl \) is the \texttt{basic-value-list} of \( sv \).

Case 2. \( et \) immediately contains an \texttt{evaluated-pseudo-variable-reference}, \( oper \).

Let \( f \) be the component of the \texttt{pseudo-variable} in \( oper \). Then \( f \) is the name of the \texttt{pseudo-variable}. Perform \texttt{assign(f,sv,nd)}.

7.5.3.1 Promote-and-convert

Let \( x \) and \( y \) be \texttt{aggregate-type}s. Then \( x \) is \texttt{promotable} to \( y \) if \( x \) and \( y \) are compatible and \( y \) is in the same \texttt{aggregate-type} of \( x \) and \( y \). However, where \( x \) is a \texttt{compound-pair}, which has \texttt{outermost}, the other may have a \texttt{compound-pair} whose \texttt{expression} components contain \texttt{integer-value}s.

Operations: \texttt{promote-and-convert(td,sv,nd)}

where \( td \) is a \texttt{data-description} or \texttt{evaluated-data-description},
\( sv \) is an \texttt{aggregate-value},
\( nd \) is a \texttt{data-description}.

result: an \texttt{aggregate-value}.

Step 1. Let \( ats \) be the \texttt{aggregate-type} of \( sv \).

Case 1.1. \( td \) is an \texttt{evaluated-data-description}.

Let \( td \) be \( td \).

Case 1.2. \( td \) is not an \texttt{evaluated-data-description}.

Perform \texttt{evaluate-data-description-for-placement(ats)} to obtain an \texttt{evaluated-data-description}, \( td \). For each entry of the form \texttt{compound-pair}, \( bp \) \texttt{aggregate} contained in \( td \) that is not a component of an \texttt{entry}, perform Step 1.2.1.

Step 1.2.1. If a \texttt{compound-pair}, \( bp \) corresponding to \( bp \) exists in \( ats \), then replace \( bp \) by \( bp \). Otherwise, replace \( bp \) by a \texttt{compound-pair} whose \texttt{lower-bound} and \texttt{upper-bound} each have an \texttt{integer-value} of 1.

Step 2. Each \texttt{compound-pair-list} in \( ats \) must equal the corresponding \texttt{compound-pair-list} in \( td \).

Step 3. Perform \texttt{scalar-elements-of-data-description(td)} to obtain an \texttt{integer}, \( n \). Let \( sv \) be an \texttt{aggregate-value} whose \texttt{aggregate-type} equals the \texttt{aggregate-type} associated with the \texttt{data-description} of \( td \), and whose \texttt{basic-value-list}, \( svl \), contains a \texttt{basic-values} nodes, each of which has no \texttt{subnode}.

Step 4. For each distinct value of \( j \) between 1 and \( n \), taken in any order, perform Steps 4.1 through 4.3.

Step 4.1. Perform \texttt{find-item-data-description(td,j)} to obtain an \texttt{item-data-description} that describes the \( j \)th scalar-element of \( sv \). Let \( pd \) be the \texttt{data-type} of this \texttt{item-data-description}.
Step 4.2. Let $x$ be the scalar-element of av that corresponds to the $j$'th scalar-element of av. Let $y$ be the <data-type> of the <item-data-description> in av which corresponds to $x$ in av.

Step 4.3.

Case 4.3.1. Both $y$ and $x$ contain <computational-type>, or one contains <offset> and the other contains <pointer>.

Perform convert(y, y, x) to obtain $y[x]$.

Case 4.3.2. Both $y$ and $x$ contain <area>.

Optionally perform raise-condition(area-condition). If there is a normal return from this operation let $y[x]$ be a <basic-value> containing <undefined>. If raise-condition(area-condition) is not performed, then let $y[x]$ be a copy of $x$.

Case 4.3.3. (Otherwise).

Let $y[x]$ be a copy of $x$.

Step 5. Append the $y[x]$ to av in order and return av.

7.5.3.2 The Set-storage Operation

This operation sets the elements of a <basic-value-list> in <allocated-storage> described by a <generation> to have the values contained in a <basic-value-list>.

Operation: set-storage(g, av

where $g$ is a <generation>,

av is a <basic-value-list>.

Step 1. Let $a_1$ be the <allocation-unit> designated by the <allocation-unit-designator> of $g$. all must be contained in <allocated-storage>. Let $a_2$ be the <basic-value-lists> of $a_1$.

Step 2. Let $d_1$ be the <data-description> of the <allocated-data-description> of $g$ and let $d_2$ be the <storage-index-list> of $g$.

Step 3. For each of the elements of all, taken in any order, perform Steps 3.1 through 3.3.

Step 3.1. Let $a_i$ be the chosen element and let $i$ be its ordinal in all. Let $j$ be a copy of the value of the <integer-value> of the <basic-value-index> of $a_i$.

Step 3.2. Let av be the $i$'th element of the <basic-value-lists>, av.

Step 3.3.

Case 1.3.1. $a_i$ contains a <position-index>.

Let $k$ be a copy of the value of the <integer-value> of the <position-index> of $a_i$. $av$ will contain either a <character-string-value> or a <bit-string-value>. Perform find-item-data-description($d_2$, $i$) to obtain the <item-data-descriptions>, $d_3$, of the target scalar-element.

If $d_3$ contains <picted> then let $n$ be its associated string-length; otherwise let $n$ be the value of the <maximum-length> in iod. Let $n$ be 1. Perform Steps 3.3.1.1 through 3.3.1.4 $n$ times.

Step 3.3.1.1. Let $t_1$ be the <character-value-list> or <bit-value-list> of the $j$'th element of $a_i$.

Step 3.3.1.2. Replace the $k$'th element of $t_1$ by a copy of the $m$'th element of $av$. 

Chapter 7: Storage and Assignment 197
Step 3.3.1.3. Let \( k \) be \( k + 1 \). If \( k \) is now greater than the number of elements in \( t \), then let \( k \) be 1 and let \( j \) be \( j + 1 \). Repeat Step 3.3.1.3.1 while the \( j \)-th element of \( t \) contains a \(<\text{null-character-string}>\) or \(<\text{null-bit-string}>\).

Step 3.3.1.3.1. Set \( j \) to \( j + 1 \).

Step 3.1.3.4. Let \( n \) be \( n + 1 \).

Case 3.3.2. \( s \) does not contain a \(<\text{position-index}>\).

Replace the \( j \)-th element of \( t \) by a copy of \( s \).

7.3.4 PSEUDO-VARIABLES

This section presents the definitions of the \(<\text{pseudo-variable}>\)s in alphabetical order. For each \(<\text{pseudo-variable}>\):

The Arguments Section indicates the number of arguments to the \(<\text{pseudo-variable}>\) and supplies names by which the arguments are referenced in the Attributes and Constraints Sections.

The Constraints Section specifies constraints on the arguments.

The Attributes Section defines the \(<\text{data-description}>\) of the \(<\text{pseudo-variable-reference}>\) and of the \(<\text{target-reference}>\) containing it in terms of the \(<\text{data-description}>\)s of the arguments.

One or two operations are defined for each \(<\text{pseudo-variable}>\). This first operation, \(<\text{evaluate}>\), where \( f \) is the name of the \(<\text{pseudo-variable}>\), is used by the operation \(<\text{evaluate-target-reference}>\). This operation returns a \(<\text{generation}>\) or an \(<\text{evaluated-pseudo-variable-reference}>\).

The second operation, \(<\text{assign}>\), where \( f \) is the name of the \(<\text{pseudo-variable}>\), is defined only for those \(<\text{pseudo-variable}>\)s whose \(<\text{evaluate}>\) operation yields an \(<\text{evaluated-pseudo-variable-reference}>\).

7.3.4.1 \textit{imag-py}:

\begin{itemize}
  \item \textbf{Arguments:} \( x \)
  \item \textbf{Constraints:} \( x \) must have the form \(<\text{argument}>\) \(<\text{expression}>\) \(<\text{value-reference}>\) \(<\text{variable-reference}>\).
  \item \textbf{Attributes:} \( x \) must have \(<\text{arithmetic}>\) (including \(<\text{arithmetic}>\) in \(<\text{picted-numeric}>\) \(<\text{mode}>\) \(<\text{complex}>\).
  \item \textbf{Operation:} \texttt{evaluate-imag-py}(x)
\end{itemize}

\textbf{Step 1.} Let \( y \) be the \(<\text{variable-reference}>\) in \( x \). Perform \texttt{evaluate-variable-reference}(y) to obtain a \(<\text{generation}>\) \( y \). Return an \(<\text{evaluated-pseudo-variable-reference}>\) \(<\text{pseudo-variable}>\) \( \textit{imag-py} \) \( y \).
Operation: assign-name-ref(t,sv,od)

where t is an <evaluated-pseudo-variable-reference>,
sv is an <aggregate-value>,
and in a <data-description>.

Step 1. Let dr have the form
<evaluated-pseudo-variable-reference>
<generation>
<evaluated-data-description>,sv
<allocation-unit-designator>,od,

Let dr be the same as dc except that all <rounding> have <real>. Perform promote-
and-convert(dr,sv,od) to obtain vr.

Step 2. For each <storage-index>,p in q, perform steps 2.1 and 2.2.

Step 2.1. Let odc, svr and adr be the scalar-elements of dc, vr and dr, respectively,
    corresponding to p.

Step 2.2.
Case 2.2.1. od has <arithmetic> but not <picture-numeric>.

Let sq be a
<generation>
<evaluated-data-description>
<storage-index-list>
P

Let x1 be the value of sq that contains a single <complex-value>. Let
x2 be a <complex-value> with first component as x1 and second component
equal to the component of svr. Perform set-storaging(sq,bvl) where bvl is
a <basic-value-list> containing x2.

Case 2.2.2. od has <picture-numeric>.

Let sq be a
<generation>
<evaluated-data-description>
<storage-index-list>
P

Let n be the associated character-string length of the <picture-
numeric> in adr. If p contains a <position-index>, increment its value
by n; otherwise, append to p a <position-index> of n+1. Perform set-
storaging(sq,bvl), where bvl is a <basic-value-list> containing svr.

1.3.4.2 Onchar-pv

Arguments: (none)

Attributes: The result has <aggregate-type>: <scalar>. The result <data-type> has
<character>.

Operation: evaluate-onchar-pv

Step 1. Return on <evaluated-pseudo-variable-reference>: <pseudo-variable>;

Chapter 7: Storage and Assignment
Operation: \texttt{assign-overflow}(t, sv, ad)

where \( t \) is an \texttt{<evaluated-pseudo-variable-reference>},
\( sv \) is an \texttt{<aggregate-value>}.
\( ad \) is a \texttt{<data-description>}.

Step 1. In either order, perform \texttt{get-established-overflow(<char-value>)} to obtain \( i \) and
perform \texttt{get-established-overflow(<source-value>)} to obtain \( str \). \( i \) and \( str \) must
not be \texttt{<fail>}.

Step 2. Convert \( sv \) to \texttt{<character> of length } \( i \), using the \texttt{<data-type>} is \( ad \) as the source
\texttt{<data-type>} for the conversion, to obtain \( c \).

Step 3. If \( i \neq 0 \), replace the \( i \)-th \texttt{<character-value>} in \( str \) by the \texttt{<character-value>} in \( c \).

7.5.4.1 \texttt{overflow-py}

Arguments: \( \text{none} \)

Attributes: The result has \texttt{<aggregate-type>}: \texttt{<scalar>}. The result \texttt{<data-type>} has
\texttt{<character>},

Operation: \texttt{evaluate-overflow-py}

Step 1. Return an \texttt{<evaluated-pseudo-variable-reference>}: \texttt{<pseudo-variable>};
\texttt{<source-py>};

Operation: \texttt{assign-overflow}(t, sv, ad)

where \( t \) is an \texttt{<evaluated-pseudo-variable-reference>},
\( sv \) is an \texttt{<aggregate-value>}.
\( ad \) is a \texttt{<data-description>}.

Step 1. Perform \texttt{get-established-overflow(<source-value>)} to obtain \( str \). \( str \) must not
be \texttt{<fail>}. Let \( n \) be its length. Convert \( sv \) to \texttt{<character> of length } \( n \), using
the \texttt{<data-type>} in \( ad \) as the source \texttt{<data-type>} for this conversion, to obtain \( cv \).

Step 2. Replace \( str \) by \( cv \).

7.5.4.9 \texttt{pregmo-py}

Arguments: \( f \in \)

Constraints: \( f \in \text{must have } <aggregate-type>: <scalar>. \) The \texttt{<data-type>} of \( f \) must have
\texttt{<file>}

Attributes: The result has \texttt{<aggregate-type>: <scalar>}. The result \texttt{<data-type>} has
\texttt{integer-type}.

Operation: \texttt{evaluate-pregmo-py}(f)

Step 1. Perform \texttt{evaluate-expression}(f) to obtain \( fv \). Return an \texttt{<evaluated-pseudo-
variable-reference>: <pseudo-variable>}: \texttt{<pregmo-py>}; \( fv \).
Operation: assign-pseudo-prim(t,sv,ed)

where t is an <evaluated-pseudo-variable-reference>,
sv is an <aggregate-value>,
ed is a <data-description>.

Step 1. Let f be the <file-information> designated by the <file-value> in t. If f must have <open>, the <complete-file-description> of f must have <print>.

Step 2. Let y be the <basic-value> in sv. Let dv be a <data-type> which is integer-type (Section 2.5.1.2). Perform convert(dv,ed,yv) to obtain a <real-value>yv. rv must be non-negative.

Step 3. Set the <page-number> component in f to contain an <integer-value> with the same component as rv.

2.5.4.5 real-sv

Arguments: x

Constraints: x must have the form <argument>: <expression>: <value-reference>: <variable-reference>.

All <data-type>s of x must have <arithmetic> (including <arithmetic> is <picture-number>), with <mode>: <complex>.

Attributes: The result <aggregate-type> is the <aggregate-type> of x. Each result <data-type> is the same as the corresponding <data-type> of x except that it has <mode>: <real>.

Operation: evaluate-real-sv(x)

Step 1. Let y be the <variable-reference> in x.


Operation: assign-real-pv(t,sv,ed)

where t is an <evaluated-pseudo-variable-reference>,
sv is an <aggregate-value>,
ed is a <data-description>.

Step 1. Let t have the form
<evaluated-pseudo-variable-reference>:<generation>-q;
<evaluation>-q;
<evaluated-data-description>,dc
<allocation-unit-designator>,and.

Let dr be the same as dc except that all <mode>s have <real>. Perform promote-and-convert(dv,sv,dr) to obtain vr.

Step 2. For each <storage-index>-p in q, perform Steps 2.1 and 2.2.

Step 2.1. Let mdc, svr, and drn be the scalar-elements of dc, vr, and dr, respectively, corresponding to p.

Chapter 7: Storage and Assignment 201
Step 2.2.

Case 2.2.1. $odc$ has $<\text{arithmetic}>$, but not $<\text{picture-numeric}>$.

Let $eg$ be a

$<\text{generation}>,
<\text{evaluated-data-description}>,
odc;
\text{and}
<\text{storage-index-list}>,
p$.

Let $x1$ be the value of $eg$ (it contains a single $<\text{complex-value}>$). Let
$x2$ be a $<\text{complex-value}>$ with first component equal to the component of
$avr$ and second component as $x1$. Perform set-storage($eg, bvl$) where $bvl$ is a $<\text{basic-value-list}>$ containing $x2$.

Case 2.2.2. $odc$ has $<\text{picture-numeric}>$.

Let $eg$ be a

$<\text{generation}>,
<\text{evaluated-data-description}>,
sdr;
\text{and}
<\text{storage-index-list}>,
p$.

Perform set-storage($eg, bvl$) where $bvl$ is a $<\text{basic-value-list}>$ containing
$avr$.

7.5.8.4 String-xy

Arguments: $x$

Constraints: $x$ must have the form $<\text{argument}>,<\text{expression}>,<\text{value-reference}>$; $y$; where
$y$ is a $<\text{variable-reference}>$. Let $ad$ be the $<\text{declaration}>$ designated by the
$<\text{declaration-designator}>$ in $x$. Each $<\text{item-data-description}>$ in $ad$ must have
$<\text{aligned}>$. Further, one of the following two conditions must hold:

1. all $<\text{data-type}>s$ in $ad$ must have $<\text{character}>$, $<\text{nonvarying}>$, or
   $<\text{picture}>$, or
2. all $<\text{data-type}>s$ in $ad$ must have $<\text{bit}>$, $<\text{nonvarying}>$.

Attributes: The result has $<\text{aggregate-type}>,<\text{scalar}>$. The result $<\text{data-type}>$ has the
derived common $<\text{string-type}>$ of the $<\text{data-type}>'s$ of $x$, and $<\text{nonvarying}>$.

Operation: $\text{evaluate-string-xy}(x)$

Step 1. Let $y$ be the $<\text{variable-reference}>$ in $x$. Perform evaluate-variable-referencely
to obtain a $<\text{generation}>,gy$, which must be connected. Let the $<\text{evaluated-data-}
description>\text{'s}$, $<\text{allocation-unit-designator}>\text{'s}$, and $<\text{storage-index-list}>\text{'s}$ components
of $gy$ be $d$, and $n$, and $s$, respectively.

Step 2. If the $<\text{data-type}>$ of the $i$'th scalar-element in the $<\text{generation}>'s$ has $<\text{string}>$
let $kll$ be the value of the corresponding $<\text{maximum-length}>$ component in $d$;
otherwise let $kll$ be the associated character-string length of the $<\text{picture}>'s$
in $d$.

Step 3. Return a $<\text{generation}>,g$, with components as follows. The $<\text{evaluated-data-}
description>\text{'s}$ of $g$ is described under Attributes, above, with $<\text{maximum-length}>
the sum of all the $kll$'. The $<\text{allocation-unit-designator}>$ of $g$ is a copy of
$s2d$. The $<\text{storage-index-list}>$ of $g$ contains a copy of the first component of $s$.202
7.5.6.7 SUBSTR-PV

Arguments: t, st, l, le

Constraints: t must have the form <argument>; <expression>; <value-reference>; <variable-reference>

All the <data-types> of t must have <string>. The <aggregate-types> of the <argument>s of st and le must be promotable to the <aggregate-type> of t. All <data-types> of st and le must have <computational-type>

Attributes: The result <aggregate-type> is the <aggregate-type> of t. Each <data-type> is the same as the corresponding <data-type> of t.

Operation: evaluate-substr-pv(t, st, le)

Step 1. Let ty be the <variable-reference> in t. In any order, perform evaluate-variable-reference(ty) to obtain a <generation>, q, perform evaluate-expression(st) and evaluate-expression(le), if it occurs, to obtain <aggregate-values>, x and y.

Step 2. Corresponding <bound-pair> in the <aggregate-types> of g, a, and y must be equal.

Step 3. Let x' and y' be <aggregate-values> whose <aggregate-types> are the same as those of x and y, respectively, and whose scalar-elements are obtained by converting the corresponding scalar-elements of x and y to integer-type.

Step 4. Return as <evaluated-pseudo-variable-reference>; <pseudo-variable>; <substr-pv>; g x' y'.

Operation: assign-substr-pv(t, sv, sd)

where t is an <evaluated-pseudo-variable-reference> as returned by evaluate-substr-pv, i.e.
<evaluated-pseudo-variable-reference>
<pseudo-variable>
<substr-pv>
<generation>, q;
<evaluated-data-description>, e;
<allocation-unit-designator>, a;
<storage-index-list>, l;
<aggregate-value>, x;
<aggregate-value>, y;
sv is an <aggregate-value>,
sd is a <data-description>.

Step 1. Corresponding <bound-pair> in the <aggregate-types> of sv and g must be equal.

For each <storage-index>-p, in sd, taken in any order, perform Steps 1.1 through 1.3.

Step 1.1. Let pp be the ordinal of p within sd. Perform find-item-data-description(sd, pp) to obtain an <item-data-description>, idd. Let sg be a <generation>
<evaluated-data-description>
<data-description>
<idd>
and
<storage-index-list>
p.

Perform value-of-generation(sg) to obtain an <aggregate-value>, sv. If idd has <varying> then sv must not contain <defined>. Let x be the length of the <character-string-value> or <bit-string-value> in sv (cf. Section 9.1.3.4).

Step 1.2. Let i be the scalar-element in x corresponding to p in sd. If yl exists, let j be the scalar-element in yl corresponding to p in sd; otherwise let j = k - 1.2.
Step 1.3.

Case 1.3.1. \( 0 \leq i-1 \leq j+1 \leq n \).

Step 1.3.1.1. Let \( S_t \) be the \(<\text{string-type}>\) contained in \( y \). Let \( S_v \) be a

\[
\begin{align*}
\text{generation} & : \\
\text{evaluated-data-description} & : \\
\text{data-description} & : \\
\text{item-data-description} & : \\
\text{composite-type} & : \\
\text{compositional-type} & : \\
\text{string} & : \\
\text{maximum-length} & : \\
\text{context-expression} & : \\
\text{integer-value} & : \\
\text{converting} & : \\
\text{storage-index-list} & : \\
\end{align*}
\]

If \( p \) contains a \(<\text{position-index}>\), increment its value by \( i-1 \); otherwise append to \( p \) a \(<\text{position-index}>\) of \( i \).

Step 1.3.1.2. Perform convert\( (S_t, S_v, S_v) \) to contain \( S_v \), where \( S_t \) is the \(<\text{data-type}>\) of \( S_v \), \( s_v \) is the scalar-element in \( s_v \) corresponding to \( p \) in \( S_v \), and \( s_d \) is the \(<\text{data-type}>\) in \( s_d \) corresponding to \( p \) in \( s_d \).

Step 1.3.1.3. Perform set-storage\( (S_t, S_v) \).

Case 1.3.2. (Otherwise).

Perform raise-condition\( (\text{string-range-condition}) \).

7.1.4.8 Unspec-py

Argument: \( x \)

Constraints: \( x \) must have the form \(<\text{argument}>\), \(<\text{expression}>\), \(<\text{value-reference}>\), \(<\text{variable-reference}>\).

\( x \) must have \(<\text{aggregate-type}>\), \(<\text{scalar}>\).

Attributes: The result has \(<\text{aggregate-type}>\), \(<\text{scalar}>\). The result \(<\text{data-type}>\) has \(<\text{bit}>\).

Operation: \( \text{evaluate-unspec-py}(x) \)

Step 1. Let \( y \) be the \(<\text{variable-reference}>\) in \( x \). Perform evaluate-variable-reference\( y \) to obtain a \(<\text{generation}>\), \( g \). Return an \(<\text{evaluated-pseudo-variable-reference}>\), \(<\text{unspec-py}>\), \( g \).

Operation: \( \text{assign-unspec-py}(t, s_v, s_d) \)

where \( t \) is an \(<\text{evaluated-pseudo-variable-reference}>\), \( s_v \) is an \(<\text{aggregate-value}>\), \( s_d \) is a \(<\text{data-description}>\).

Step 1. \( s_v \) must have \(<\text{aggregate-type}>\), \(<\text{scalar}>\). Let \( g \) be the \(<\text{generation}>\) in \( t \). Convert \( s_v \) to \(<\text{bit}>\) of length \( n \), where \( n \) depends on \( g \) in an implementation-defined fashion.

Step 2. In an implementation-defined fashion construct from the converted value of \( s_v \) a \(<\text{basic-value}>\), \( v \), depending on properties of \( g \). This value may contain \(<\text{undefined}>\). Perform set-storage\( (g, v) \).
7.6 Variable-reference

7.6.1 EVALUATE-VARIABLE-REFERENCE

The result of this operation is the <generation> referenced in the <variable-reference>.

Operation: 
`evaluate-variable-reference(vr)`

where `vr` is a `<variable-reference>`.

results: a `<generation>`.

Step 1. Let `d` be the `<declaration>` designated by the `<declaration-designator>, dp`, or `vr`.

Step 2.

Case 2.1. The `<storage-type>` of `d` has `<based>`.

Perform `select-based-generation(vr)` to obtain the `<generation>, q`.

Case 2.2. The `<storage-type>` of `d` has `<defined>`.

Perform `evaluate-defined-reference(vr)` to obtain the `<generation>, q`. Go to Step 6.

Case 2.3. The `<storage-type>` of `d` has `<parameter>`.

Perform `find-block-state-of-declaration(dp)` to obtain the `<block-state>, bs` of `d`. Find the `<parameter-directory-entry>, pde`, in the `<parameter-directories>` of `bs` whose `<identifier>` component is the same as the `<identifier>` immediate component of `d`. `pde` must not immediately contain the `<undefined>` component. Let `g` be a copy of the `<generation>` of `pde`. The `<allocation-unit>` designated by the `<allocation-unit-designator>` of `g` must be contained in the `<allocation-unit-list>`, possibly as a component of an `<area-list>`.

Case 2.4. The `<storage-type>` of `d` has `<automatic>`, `<controlled>`, or `<static>`.

Perform `find-directory-entry(dp)` to obtain a directory entry. If the `<storage-type>` of `d` has `<controlled>`, the entry must have a `<generation-list>`; let `g` be a copy of the `<generation>` most recently added to the `<generation-list>`. Otherwise, let `g` be a copy of the `<generation>` component of the directory entry.

Step 3. If `vr` immediately contains an `<identifier-list>, il`, then perform `select-qualified-reference(g, il, d)` to obtain a `<generation>, q`.

Step 4. If `vr` has a `<by-name-parts-list>` perform `evaluate-by-name-parts-list(y, vr, d)` to obtain a `<generation>, q`.

Step 5. If `vr` has a `<subscript-list>`, then perform `select-subscripted-reference(g, il)`, where `il` is the `<subscript-list>`, to obtain a `<generation>, q`.

Step 6. In the `<evaluated-data-description>` of `g`, replace each `<offset>` component by a copy of the corresponding `<offset>` component of the `<data-description>` immediate component of the `<variable-reference>, vr`.

Note: This is a definitional artifact to make the proper target `<offset>`s available for conversion in the operation `assign`.

Step 7. Return `g`. 

Chapter 7: Storage and Assignment 205
7.6.1.1 Connected Generations

A <generation>, $g$, is connected unless it is found to be unconnected by the following steps.

Step 1. Let $g$ be

- <generation>,
  - <evaluated-data-description>,edt,
  - <data-description>,dd,
  - <allocation-unit-designator>,and,
  - <storage-index-list>,nil.

Let $n$ be the number of elements in nil.

Step 2. Let bwill be the $i$th element of the <basic-value-list> of the <allocation-unit> designated by and.

Step 3. Let i be the <integer-value> contained in the <basic-value-index> of the first element of nil. Let j be the <integer-value> contained in the <position-index> of the first element of nil, if this <position-index> exists, and 1 otherwise. For $k=1,\ldots,n$ perform Steps 3.1 and 3.2.

Step 3.1. Perform find-item-data-description(dd,k) to obtain an <item-data-description>,idd.

Step 3.2.

Case 3.2.1. idd contains both <string> and <converting>.

Step 3.2.1.1. Let the $k$th element of nil be

- <storage-index>,
  - <basic-value-index>,bvi
  - <position-index>,poi.

Step 3.2.1.2. If the value contained in the <integer-value> of bvi is not equal to 1, $g$ is unconnected.

Step 3.2.1.3. If the value contained in the <integer-value> of poi is not equal to j, $g$ is unconnected.

Step 3.2.1.4. Let m be the <maximum-length> of idd. Let j be j+1.

Step 3.2.1.5. Let n-nil be the number of elements in the <character-value-list> or <bit-value-list> of bvi. Repeat Step 3.2.1.5.1 while j is greater than n-nil.

Step 3.2.1.5.1. Let j be j-n-nil. Let i be i+1.

Case 3.2.2. (Otherwise).

Step 3.2.2.1. Let the $k$th element of nil be

- <storage-index>,
  - <basic-value-index>,bvi.

Step 3.2.2.2. If the value contained by the <integer-value> of bvi is not equal to 1, $g$ is unconnected.

Step 3.2.2.3. If j is not 1, $g$ is unconnected.

Step 3.2.2.4. Let i be i+1.
7.6.2 SELECT-BASED-GENERATION

The result of this operation is the generation corresponding to the base variable referenced in a variable-reference before any name-qualification or subscripting has been performed.

Operation: select-based-generation(vr)

where vr is a variable-reference.

result: a generation.

Step 1.

Case 1.1. vr has a locator-qualifier, lq.

Let vair be the value-reference of lq.

Case 1.2. vr has no locator-qualifier.

The base component of the declaration designated by the declaration-designator of vr has a value-reference. Let this be vair.

Step 2. Perform evaluate-value-reference(vair) to obtain an aggregate-value with a basic-value,v.

Step 3. Let dt be the data-type of vair. If dt has offset, perform convert(pointer,dt,v) to obtain a pointer-value,v.

Step 4. v must not contain null. Let g be the generation in v.

Step 5. The allocation-unit designated by the allocation-unit-designator component of g must be contained in the allocation-unit-list of allocated-storage, possibly as a component of an area-value.

Step 6. Perform check-based-reference(g,vr) to obtain a generation,g1.

Step 7. Return g1.

7.6.3 CHECK-BASED-REFERENCE

This operation checks that the attributes of the variable being referenced agree with those of the generation being referenced.

Operation: check-based-reference(g,vr)

where g is a generation,

vr is a variable-reference.

result: a generation.

Step 1. Let d be the declaration designated by the declaration-designator of vr.

Let dd be the data-description immediately contained in the variable of d.

Step 2. Let g comprise the allocation-unit-designator,add, the evaluated-data-description,add, and the storage-index-list,add.

Step 3.

Case 3.1. The immediate component of dd is a structure-data-description,add.

Step 3.1.1. add contains an identifier-list,mil and a member-description-list,mmil.

Step 3.1.2. If vr contains an identifier-list,mil, then let m be the position of the element of mil that is identical to the first element of il. Otherwise, let m be the number of elements in mil.

Chapter 7: Storage and Assignment 207
Step 3.1.3. Let d2l be

\[
\begin{align*}
\langle \text{data-description} \rangle: \\
\langle \text{structure-data-description} \rangle: \\
\langle \text{identifier-list} \rangle, lli \\
\langle \text{member-description-list} \rangle, ddl;
\end{align*}
\]

where lli contains a copy of the first n elements of ll, and ddl contains a copy of the first n elements of dll.

Case 3.2. (Otherwise).

Let ddl be a copy of dd.

Step 4. Perform evaluate-data-description-for-reference(ddl, g) to obtain an \langle evaluated-data-description \rangle, ddl.

Step 5.

Case 5.1. ddl has no \langle refer-option \rangle node and both ddl and dd are such that each \langle data-type \rangle component contains \langle smallmem \rangle and

either (1) each \langle data-type \rangle component contains \langle convarying \rangle and \langle bit \rangle,

or (2) each \langle data-type \rangle component contains either \langle pictured \rangle or \langle convarying \rangle and \langle character \rangle.

Perform overlay-strings(ddl, g, lli) to obtain a \langle storage-index-list \rangle, sll.

Case 5.2. (Otherwise).

Let cdd2 be a copy of dd. Delete from cdd2 and ddl2, a copy of ddl2, all occurrences of \langle refer-option \rangle, \langle initial \rangle, \langle local \rangle, and all \langle variable-reference \rangle components of \langle offset \rangle. Each subnode of cdd2 must be equal to the corresponding subnode of cdd except that if cdd2 is an

\[
\begin{align*}
\langle \text{evaluated-data-description} \rangle: \\
\langle \text{data-description} \rangle: \\
\langle \text{structure-data-description} \rangle: \\
\langle \text{member-description-list} \rangle, sll;
\end{align*}
\]

then the \langle member-description-list \rangle in cdd2 corresponding to ddl2 may have more components than sll, and the excess components are ignored.

Let d2lie be the \langle data-description \rangle node of d2l. Perform scalar-elements-of-data-description(d2lie) to obtain n, the number of scalar-elements corresponding to d2lie. Let sll2 be a \langle storage-index-list \rangle containing a copy of the first n elements of sll.

Step 6. Return a \langle generation \rangle and ddl sll2.

7.6.4 OVERLAY-STRINGS

This operation constructs a \langle storage-index-list \rangle that reflects the fact that is a string-overlay defined reference or a reference to a based variable, the resulting \langle generation \rangle may describe strings that start inside a \langle character-string-value \rangle or \langle bit-string-value \rangle. The \langle position-index \rangle is used to define the starting point of the strings.

Operation: overlay-strings(d2l, g, index)

where d2l is an \langle evaluated-data-description \rangle, g is a \langle generation \rangle, index is an integer.

result: a \langle storage-index-list \rangle.

Step 1. Let d2l be the \langle data-description \rangle simply contained in d2l. Perform scalar-elements-of-data-description(d2l) to obtain the number of scalar-elements, n, described by d2l. Perform Step 1.1 for i = 1, ..., n.
Step 1.1. Perform find-item-data-description(oid,1) to obtain the <item-data-descriptions>, oid of the 1st scalar-element described by oid. If oid contains <string> then let m1 be the value of the <maximum-length> of oid; otherwise, let m1 be the associated character-string length of the <data-type> of oid.

Step 2. Let d1 be the <data-description> simply contained in q. Perform scalar-elements-of-data-description(oid,q) to obtain the number of scalar-elements corresponding to q, nq. Perform Step 2.1 for i=1,...,nq.

Step 2.1. Perform find-item-data-description(oid,q,i) to obtain the <item-data-descriptions>, oid, of the i' th scalar-element corresponding to q. If oid contains <string> then let m1 be the value of the <maximum-length> of oid; otherwise, let m1 be the associated character-string length of the <data-type> of oid.

Step 3. Let index be a copy of index. Let m1 be the value of the sum m1(1) + ... + m1(q). Let nil be a <storage-index-list> with no elements. Let i be 1. Perform Steps 3.1 through 3.7 n times.

Step 3.1. Find the maximum integer j such that j<eq> and such that the value of the sum m1(1) + ... + m1(j) is less than index. Let the value of this sum be nilj. Note that j and nil may be zero.

Step 3.2. Let j be j+1.

Step 3.3. Let nil be the <basic-value-index> of the j' th element of the <storage-index-list> of q, and let p be the <position-index> of the same element.

Step 3.4. Let px be the <integer-value> containing the value index-nilj-1+p.

Step 3.5. Append

   <storage-index>:
   nil
   <position-index>:
   px;
   to nil.

Step 3.6. Let index be index-nilj. The value of index must not be greater than nil+1.

Step 3.7. Let i be i+1.

Step 4. Return nil.

7.3.5 EVALUATE-DATA-DESCRIPTION-FOR-REFERENCE

This operation takes a <data-description> and a <generation> and constructs an <evaluated-data-description>, using the <generation> to evaluate any <refer-option>.

Operation: evaluate-data-description-for-reference(oid,q)

where oid in a <data-description>,
q is a <generation>.
result: an <evaluated-data-description>.

Step 1. Let o11 be a copy of oid. For each <extent-expression>, ee, of o11, taken in left-to-right order, perform Step 1.1.

Step 1.1.

Case 1.1.1. ee has a <refer-option>, ee.

Step 1.1.1.1. Let id1 be the <identifier-list> of ee. Let n be the number of elements in id1. Let o11 be a copy of o11 and let o11 designate the <data-description>, o11. Let i = 1. Perform Steps 1.1.1.1.1 through 1.1.1.1.3 n times.
Step 1.1.1.1. Let id be the i'th element of idl. Let add be the structure-data-description that is the immediate component of the <data-description> designated by dd. Let id31 be the <identifier-list> of add. Let the n'th element of idl be identical with id.

Step 1.1.1.2. Let mll be the <member-decription-list> of add. Delete all elements following the n'th element in both idl and mll. Set ddp to designate the <data-description> immediately contained in the n'th element of mll.

Step 1.1.1.3. Set i to i+1.

Step 1.1.1.2. Perform scalar-elements-of-data-description(cdd) to obtain j, the number of scalar elements corresponding to cdd.

Step 1.1.1.3. Let bv be the j'th element of the <basic-value-lists> designated by v. Replace the <expression> of vv by the <integer-value> obtained by converting bv to integer-type. The source type for this conversion is the <data-type> in the <data-description> designated by ddp.

Case 1.1.2. (Otherwise).
Perform eval-expressions-to-integer(vv), where vv is the <expression> of vv, to obtain the <integer-value>. Replace vv by vv.

Step 2. Return the <evaluated-data-description> containing cdd, as modified, as its immediate component.

7.6.4 SELECT-QUALIFIED-REFERENCE

This operation selects the part of a <generation> that corresponds to an <identifier-list>.

Operation: select-qualified-reference(q, idl, d)
where q is a <generation>,
idl is an <identifier-list>,
d is a <declaration>.
result: a <generation>.

Step 1. Let tdd be the immediate component of the <data-description> immediately contained in the <variable> of d. Let cdd be a copy of the <storage-index-list> of q and cdd be a copy of the <evaluated-data-descriptions> of q. Let red be the immediate component of the <data-description> of cdd.

Step 2. Let account be 1. Let n be the number of elements in idl. For j=1,...,n, perform Steps 2.1 through 2.9.

Step 2.1. If tdd is a <dimensioned-data-description> then perform Steps 2.1.1 through 2.1.2.

Step 2.1.1. Let bpl be the <bound-pair-list> of tdd. Let until and il[i][i] be the <integer-value> components of the <upper-bound> and the <lower-bound> of the i'th <bound-pair> of bpl respectively. Let account be the value of the product

$$
\prod_{i=1}^{n} \text{until[i][i][i][i]}
$$

where k is the number of elements of bpl.

Step 2.1.2. Let red and red be the immediate components of the <element-data-description> of tdd and red respectively.

218
Step 2.2. \( \text{tdl} \) and \( \text{tdll} \) will both be a \text{structure-data-description}. Let \( \text{ndl} \) and \( \text{ndll} \) be, respectively, the \text{member-description-list} and the \text{identifier-list} of \( \text{tdl} \). Let \( \text{ndll} \) be the \text{member-description-list} of \( \text{tdll} \).

Step 2.3. Let the \( n \)th element of \( \text{ndl} \) be the one that is identical with the \( j \)th element of \( \text{tdl} \).

Step 2.4. Let the number of elements of \( \text{ndl} \) be \( n \). Let \( \text{ndlv} \) be the \text{data-description} of the \( k \)th element of \( \text{ndl} \). Perform scalar-elements-of-data-description(\( \text{ndlv} \)) to obtain the number of scalar-elements \( \text{ndlvi} \) corresponding to \( \text{ndlv} \), for \( k \) as in.

Step 2.5. Let \( ii \) be the value of the sum \( \text{ndlvi} + \ldots + \text{ndlvin} \), \( 12 \) be \( \text{ndlvi} \), and \( \text{nl} \) be the value of the sum \( \text{ndlvi} + \ldots + \text{ndlvin} \).

Step 2.6. Let \( \text{ind} \) be \( 1 + \text{ind} + \text{ind} + 1 \). Perform Steps 2.6.1 through 2.6.3 \text{count} times.

Step 2.6.1. Delete \( ii \) successive elements of \( \text{ndll} \) starting with the element whose index is \( \text{ind} + 12 \).

Step 2.6.2. Delete \( ii \) successive elements of \( \text{ndll} \) starting with the element whose index is \( \text{ind} \).

Step 2.6.3. Let \( \text{ind} \) be \( \text{ind} + \text{ind} + \text{ind} + 13 \).  

Step 2.7. Let \( \text{tdl} \) be the immediate component of the \text{data-description} of the \( n \)th element of \( \text{ndl} \) and let \( \text{tdll} \) be the immediate component of the \text{data-description} of the \( n \)th element of \( \text{ndll} \).

Step 2.8.

Case 2.8.1. \( \text{tdl} \) is a \text{dimensioned-data-description} and \( \text{tdll} \) is a \text{dimensioned-data-description}.

Append the elements of the \text{bound-pair-list} of \( \text{tdll} \) to the \text{bound-pair-list} of \( \text{tdl} \). Replace the \text{element-data-description} of \( \text{tdll} \) by the \text{element-data-description} of \( \text{tdl} \).

Case 2.8.2. \( \text{tdl} \) is a \text{dimensioned-data-description} and \( \text{tdll} \) is a \text{structure-data-description}.

Replace the \text{element-data-description} of \( \text{tdll} \) by \( \text{tdl} \).

Case 2.8.3. (Otherwise).

Replace \( \text{tdll} \) by \( \text{tdl} \).

Step 2.9. Let \( \text{tdll} \) be \( \text{tdl} \).

Step 3. Return the \text{generation} comprising a copy of the \text{allocation-unit-designator} of 
\( g \), \( \text{call} \), and \( \text{call} \).
7.6.7 SELECT-SUBSCRIPTED-REFERENCE

This operation selects the part of a \textit{generation} that corresponds to a given \textit{subscript-list}.

\begin{quote}
Operation: \texttt{select-subscripted-reference}(g, sll) \\
where \texttt{g} is a \textit{generation}, \\
\texttt{sll} is a \textit{subscript-list}. \\
result: a \textit{generation}.
\end{quote}

\textbf{Step 1.} Let \(cg\) be a copy of \texttt{g}. \(cg\) will have the form
\begin{quote}
\texttt{<generation>} \\
\texttt{<evaluated-data-description>} \\
\texttt{<data-description>} \\
\texttt{<dimensioned-data-description>} \\
\texttt{<element-data-description>}, \texttt{e10} \\
\texttt{<bound-pair-list>}, \texttt{sll}; \\
\texttt{<allocation-unit-denominator>} \\
\texttt{<storage-index-list>}, \texttt{nil}.
\end{quote}

\textbf{Step 2.} Let \texttt{cpil} be a copy of \texttt{sll}.

\textbf{Step 3.} Let \texttt{dd} be a \textit{data-description} that immediately contains a copy of the immediate sub-tree of \texttt{e10}. Perform \texttt{scalar-elements-of-data-description}(\texttt{dd}) to obtain the integer \(n\).

\textbf{Step 4.} Let \(n\) be the number of elements of \texttt{sll}. Let \texttt{ubill} be the \(1\)\textsuperscript{st} element of \texttt{sll}. Let \texttt{cpil}\(_{i}\) be the \(i\)\textsuperscript{th} element of \texttt{cpil} and let \texttt{ubill}\(_{i}\) and \texttt{lbill}\(_{i}\), respectively, be the \textit{integer-value} contained in the \texttt{<upper-bound>} and \texttt{<lower-bound>} of \texttt{cpil}\(_{i}\). For \(i = 1, \ldots, n\), where the values are chosen in any order, perform \textbf{Steps 4.1} and 4.2.

\textbf{Step 4.1.}

\textbf{Case 4.1.1.} \texttt{sll} has \texttt{<asterisk>}

Let \texttt{ubp} be a copy of \texttt{cpil}\(_{1}\).

\textbf{Case 4.1.2.} (Otherwise).

\textbf{Step 4.1.2.1.} If \texttt{ubill} immediately contains an \textit{expression}, \(v\), then perform \texttt{evaluate-expression-to-integer}(\textit{expression}) to obtain the \textit{integer-value} \(v\). Otherwise let \(v\) be the \textit{integer-value} contained in \texttt{ubill}.

\textbf{Step 4.1.2.2.} If \(v\) is less than \texttt{ubill}\(_{1}\) or \(v\) is greater than \texttt{ubill}\(_{n}\), then perform \texttt{raise-condition}(\textit{<subscript-range-condition>}). Let \texttt{ubp} be
\begin{quote}
\texttt{<bound-pair>} \\
\texttt{<upper-bound>}; \\
\texttt{<element-expression>}; \\
\texttt{v}; \\
\texttt{<lower-bound>}; \\
\texttt{<element-expression>}; \\
\texttt{v}.
\end{quote}

\textbf{Step 4.2.} Replace \texttt{cpil}\(_{1}\) by \texttt{ubp}.

\textbf{Step 5.} Perform \texttt{extract-slice-of-array}(\texttt{ipl}, \texttt{ubp}, \texttt{sll}) to obtain a \texttt{<storage-index-list>}, \texttt{nil}. Replace \texttt{nil} by \texttt{nil}.

\textbf{Step 6.} Let \texttt{apil} be a \textit{<bound-pair-list>} with no components. For \(i = 1, \ldots, n\), if \texttt{ubill}\(_{i}\) has \texttt{<asterisk>} then append a copy of the \(i\)\textsuperscript{th} component of \texttt{apil} to \texttt{apil}. If \texttt{ubill}\(_{i}\) has no components then replace the \textit{data-description} of \texttt{cg} by \texttt{g}; otherwise replace \texttt{apil} by \texttt{apil}.

\textbf{Step 7.} Return \texttt{cg}.
7.6.8 EVALUATE-BY-NAME-PARTS-LIST

This operation takes a <generation> and a <by-name-parts-list> and constructs a new <generation> containing all the parts specified by the <by-name-parts-list> in the order of that list.

Operation: evaluate-by-name-parts-list(g,vr,d)

where & is a <generation>,
vr is a <variable-reference>,
d is a <declaration>.

result: a <generation>.

Step 1. Let &k be the allocation-unit-designator of &. Let &e be:
<generation>
<evaluated-data-descriptions>
<Data-description>
<structure-data-description>
<Member-description-list>,&d1,...
<storage-index-list>,&ll.

Step 2. Let n be the number of elements in the <by-name-parts-list>,&npl, of vr. For i=1,...,n perform steps 2.1 through 2.3.

Step 2.1. Let &m be the i'th element of &npl. Let &i be a copy of the <identifier-list> of &v, if one exists; otherwise let &i be an <identifier-list> with no elements. Append the elements of the <identifier-list> of &m to &i.

Step 2.2. Perform select-qualified-reference(&i, &d1, &d1) to obtain a <generation>, &eg.

Step 2.3. Let &d be the <data-description> of &eg. Let &d be a <member-description>; &d. Append &d to &ll. Append the elements of the <storage-index-list> of &eg to &ll.

Step 3. Return &e.

7.6.9 EVALUATE-DEFINED-REFERENCE

This operation evaluates a <variable-reference> for a <variable> whose <storage-type> is defined and yields a <generation>.

Operation: evaluate-define-reference(vr)

where vr is a <variable-reference>.

result: a <generation>.

Step 1. Let &p be the <declaration-designator> of the <variable-reference>,&vr, let &d be the <declaration> designated by &p, and let &d be the <defined> component of &d. Perform find-block-state-of-declaration(&p) to obtain the <block-state>,&b.

Step 2. Let &d be the <defined-directory-entry> in &b whose <identifier> is equal to that of &d. Let &d be the <evaluated-data-description> of &d.

Step 3. &d has a <base-item> with a <variable-reference>,&vr1. Let &vr2 be a copy of &vr1. If &vr2 contains a <module-list> with <list> components then delete the <module-list> in &vr2.

Step 4. Perform evaluate-variable-reference(&vr2) to obtain the <generation>, &g.

Step 5. Case 5.1. &d has a <position> component.

Perform evaluate-string-overlay-defined-reference(&vr2, &d, &g) to obtain the
<generation>, g2.

Case 5.2. dc has a <subscript-list> with an <index> component.

Perform evaluate-qual-defined-reference(vr, edd, g2) to obtain the
<generation>, g6.

Case 5.3. (Otherwise).

Perform check-simply-defined-reference(vr). If the value obtained is
<empty>, then perform evaluate-simply-defined-reference(vr, edd, g) to obtain
the <generation>, g6. Otherwise perform evaluate-string-overlay-defined-
reference(vr, edd, g) to obtain the <generation>, g6.

Step 6. Return g6.

7.6.10 EVALUATE-SIMPLY-DEFINED-REFERENCE

This operation takes a simply-defined <variable-reference>, the <evaluated-data-
description> associated with the variable and the <generation> being referenced and
constructs the <generation> that is the result of the reference.

Operation: evaluate-simply-defined-reference(vr, edd, g)

where vr is a <variable-reference>,

edd is an <evaluated-data-description>,

q is a <generation>.

result: a <generation>.

Step 1. Let eddy be the <evaluated-data-description> of g.

Step 2. For every <extent-expression>e contained in eddy that is not contained in a
<parameter-descriptor> or <returns-descriptor>, let v be the <integer-value>
contained in e.

Case 2.1. e is contained in a <member-description> or an <area-size>.

v must be equal to the corresponding <integer-value> of eddy.

Case 2.2. e is contained in a <lower-bound> and is not contained in a <member-
description>.

v must be greater than or equal to the corresponding <integer-value> of eddy.

Case 2.3. (Otherwise).

v must be less than or equal to the corresponding <integer-value> of eddy.

Step 3. If eddy contains any <bound-pair>, perform adjust-bound-pairs(e, g1) to yield a
new <storage-index-list>s1, corresponding to the <bound-pair> of eddy. Let g2 be the <generation> constructed from a copy of the <allocation-unit-designator>
of q, edd, and s1.

Step 4. If vr has an <identifier-list>,id1, then perform select-qualified-
reference(q1, id1, g2), where q is the <declaration> designated by the
<declaration-designator> of vr, to obtain a <generation>. Replace q1 with this
<generation>.

Step 5. If vr has a <by-name-parts-list>, perform evaluate-by-name-parts-list(g2, vr, g1) to
obtain a <generation>. Replace g1 with this <generation>.

Step 6. If vr has a <subscript-list>,sed, perform select-subscripted-reference(q1, sed) to
obtain a <generation>. Replace q1 with this <generation>.

Step 7. Return g1.
7.6.11 ADJUST-BOUND-PAIRS

This operation takes a <generation> and modifies the <storage-index-list> to reflect the <bound-pair>s contained in an <evaluated-data-description>.

Operation: \texttt{adjust-bound-pair}(\texttt{g}, \texttt{edd})

where \texttt{g} is a <generation>,
\texttt{edd} is an <evaluated-data-description>,
result: a <storage-index-list>

Step 1. Let \texttt{edd} be the <evaluated-data-description> of \texttt{g} and let \texttt{cell} be a copy of the <storage-index-list> of \texttt{g}.

Step 2. If the <data-description> of \texttt{edd} immediately contains a <dimensioned-data-description>, \texttt{edd}, then perform Steps 2.1 through 2.4.

Step 2.1. Let \texttt{edd} be the simply contained <dimensioned-data-description> of \texttt{edd}. Let \texttt{bpl} and \texttt{bpl} be, respectively, the <bound-pair-list>s of \texttt{edd} and \texttt{edd}.

Step 2.2. Let \texttt{a} be the number of elements in \texttt{bpl}. For \texttt{i}=1,\ldots,\texttt{a}, the values being chosen in any order, perform Steps 2.2.1 through 2.2.5.

Step 2.2.1. Let \texttt{bpl} and \texttt{bpl} be the \texttt{i}th element of, respectively, \texttt{bpl} and \texttt{bpl}.

Step 2.2.2. Let \texttt{ubpl} and \texttt{ubpl} be the <integer-value> contained in the <upper-bound> of, respectively, \texttt{bpl} and \texttt{bpl}.

Step 2.2.3. Let \texttt{lbpl} and \texttt{lbpl} be the <integer-value> contained in the <lower-bound> of, respectively, \texttt{bpl} and \texttt{bpl}.

Step 2.2.4. If \texttt{ubpl} is greater than \texttt{lbpl} then perform \texttt{raise-condition}<\texttt{subscript-range-condition}>

Step 2.2.5. If \texttt{lbpl} is less than \texttt{lbpl} then perform \texttt{raise-condition}<\texttt{subscript-range-condition}>

Step 2.3. Let \texttt{add} be the <data-description> of the <element-data-description> of \texttt{add}. Perform scalar-elements-of-data-description(\texttt{add}) to obtain the integer \texttt{n}.

Step 2.4. Perform extract-slice-of-array(\texttt{bpl}, \texttt{bpl}, \texttt{a}, \texttt{cell}) to obtain a <storage-index-list>, \texttt{cell}.

Step 3. Return \texttt{cell}.

7.6.12 EVALUATE-LOOP-DEFINED-REFERENCE

This operation takes an loop-defined <variable-reference>, the <evaluated-data-descriptions> associated with the <variable> and the <generation> being referenced, and constructs the <generation> that is the result of the reference.

Operation: \texttt{evaluate-loop-defined-reference}(\texttt{vr}, \texttt{edd}, \texttt{g})

where \texttt{vr} is a <variable-reference>,
\texttt{edd} is an <evaluated-data-description>,
\texttt{g} is a <generation>.
result: a <generation>.

Step 1. Let \texttt{edd} and \texttt{bpl} be, respectively, the immediately contained <dimensioned-data-description> and its <bound-pair-list> in the <data-description> of \texttt{edd}. Let \texttt{n} be the number of elements in \texttt{bpl}.

Step 2. Let \texttt{dp} and \texttt{edd} be, respectively, the immediately contained <declaration-designator> and <subclass-list> of \texttt{vr}. Let \texttt{add} be a <subclass-list> containing a copy of the first \texttt{n} elements of \texttt{edd} and let \texttt{add} be a <subclass-list> containing a copy of the remaining elements, if any, of \texttt{edd}.

Chapter 7: Storage and Assignment 213
Step 3. Choose, in any order, each element of sbls that does not immediately contain
<asterisk> and perform Steps 3.1 through 3.3.

Step 3.1. Let c be the chosen element of sbls and let i be its ordinal in sbls.

Step 3.2. Perform evaluate-expression(c) and convert the result obtained to integer-
type of value v. Replace the <expression> component of the i'th element of
sbls by v.

Step 3.3. Let lb and ub be, respectively, the <lower-bound> and the <upper-bound> of
the i'th <bound-pair> of bpl. If v is less than the <integer-value> contained in lb or
greater than the <integer-value> contained in ub, perform
raise-condition(<subscript-range-condition>).

Step 4. Construct a <subscript-list-list>, nil, whose single element is a copy of sbls.
If sbls contains any <asterisk> component, then expand sbls by performing
expand-list-of-subscript-lists(sbls, eddl) to obtain a new <subscript-list-list>,
and replace sbls with this.

Step 5. Let d be the <declaration> designated by dp. For each element, eddl, of eddl
perform transform-subscript-lists(eddl, d) to obtain a <subscript-list>, and
replace eddl in sbls with this. This transforms each <subscript-list> of sbls
from a <subscript-list> that applies to the <data-description> of d into one
that applies to the <data-description> of the <declaration> designated by
the <declaration-designator> of the <variable-reference> of the <defined>
component of d.

Step 6. Construct a new <storage-index-list>, nil, by performing Steps 6.1 and 6.2 for
each element of sbls chosen in left-to-right order.

Step 6.1. Let sbli be the chosen element of sbls. Perform select-subscripted-
reference(sbli, sbli) to obtain a <generation>, gl.

Step 6.2. Append the elements of the <storage-index-list> of gl to nil.

Step 7. Let sbpl be a copy of the <bound-pair-list>, bpl. Delete each element of sbls
that is not <asterisk> and also delete the element of sbpl with the same
ordinal. Let vrc be a copy of vrc. Replace the copy of sbli in vrc by a
<subscript-list> constructed by appending the elements of a copy of sbli
to a copy of sbls.

Step 8. Let eddc be a copy of the <evaluated-data-description> of e. The <data-
description> of eddc has an immediate <dimensioned-data-description> componen:
let this be ddd2. Let eddc be a copy of eddc and let its constant copy of ddd1
be deddc.

Case 8.1. As a result of Step 7, sbpl no longer exists.

Step 8.1.1. Replace ddd1 in eddc by the subtree of its <element-data-description>.

Case 8.1.2. Replace ddd2 in eddc by the subtree of its <element-data-description>.

Case 8.2. (Otherwise).

Replace the <bound-pair-list> of eddc by sbpl.

Step 9. Let gl be a <generation> constructed from a copy of the <allocation-unit-
designator> of g, eddc, and nil. Perform check-simply-defined-reference(eddc).
The value obtained must be <true>. Perform evaluate-simply-defined-
reference(vrc, eddc, gl) to obtain a <generation>, gl.

Step 10. Return gl.
7.6.13 EXPAND-LIST-OF-SUBSCRIPT-LISTS

This operation expands a \langle\text{subscript-list-list}\rangle so that each element that is an \langle\text{asterisk}\rangle, meaning a cross-section, causes the generation of an appropriate number of elements in the list with one element for each element of the cross-section.

**Operation:** \text{expand-list-of-subscript-lists}(\text{subli}, \text{edd})

where \text{subli} is a \langle\text{subscript-list-list}\rangle,
\text{edd} is an \langle\text{evaluated-data-description}\rangle.

\text{result:} a \langle\text{subscript-list-list}\rangle.

**Step 1.** Let \text{subli} be a copy of \text{subli}. Let \text{edd} be the \langle\text{bound-pair-list}\rangle of the \langle\text{dimensioned-data-description}\rangle that is simply contained by \text{edd}. Perform Steps 1.1 through 1.3 until no \langle\text{asterisk}\rangle remains in \text{subli}.

**Step 1.1:** Choose an element, \text{subli} of \text{subli}, such that one of the elements of \text{subli} is \langle\text{asterisk}\rangle. Let an \langle\text{asterisk}\rangle be the \text{i}th element of \text{subli}.

**Step 1.2:** Let \text{ll}p be the \text{i}th element of \text{llp}. \text{llp} has the two components \langle\text{upper-bound}\rangle, \text{ll}, and \langle\text{lower-bound}\rangle, \text{lb}.

**Step 1.3:** Replace \text{subli} by (\text{ub}-\text{lb}+1) copies of \text{subli} modified such that the \text{j}th copy has its \langle\text{asterisk}\rangle component in the \text{i}th position replaced by an \langle\text{expression}\rangle containing the \langle\text{integer-value}\rangle, \text{ub}-(\text{lb}+\text{j})).

**Step 2.** Return \text{subli}.

7.6.14 TRANSFORM-SUBSCRIPT-LIST

This operation takes a \langle\text{subscript-list}\rangle and performs the transformation specified by \text{lsub} that defines to generate a new \langle\text{subscript-list}\rangle.

**Operation:** \text{transform-subscript-list}(\text{subli}, \text{d})

where \text{subli} is a \langle\text{subscript-list}\rangle,
\text{d} is a \langle\text{declaration}\rangle.

\text{result:} a \langle\text{subscript-list}\rangle.

**Step 1.** In the \langle\text{variable-reference}\rangle of the \langle\text{defined}\rangle component of \text{d} there is a \langle\text{subscript-list}\rangle. Let \text{csbl} be a copy of this \langle\text{subscript-list}\rangle.

**Step 2.** For each \langle\text{lsub}, \text{ic}\rangle of \text{csbl} perform Step 2.1.

**Step 2.1:** Let \text{i} be the value of the \langle\text{integer}\rangle component of \text{ic}. \text{i} must be greater than zero and less than or equal to the number of elements in \text{subli}. Replace \text{ic} in \text{csbl} by the \langle\text{integer-value}\rangle of the \text{i}th element of \text{subli}.

**Step 3.** For each \langle\text{expression}\rangle, \text{e} of \text{csbl} perform Step 3.1.

**Step 3.1:** Perform \text{evaluate-expression-to-integer} to obtain an \langle\text{integer-value}\rangle, \text{iv}, and replace \text{e} in \text{csbl} by \text{iv}.

**Step 4.** Return \text{csbl}.

Chapter 7: Storage and Assignment
7.6.15 EVALUATE-STRING-OVERLAY-DEFINED-REFERENCE

This operation takes a <variable-reference> for a string-overlay-defined <variable>, the evaluated-data-descriptions associated with the <variable>, and the base <generation> being referenced and constructs the <generation> that is the result of the reference.

**Operation:** 
`evaluate-string-overlay-defined-reference(vr, odd, g)`

where `vr` is a <variable-reference>,
`odd` is an <evaluated-data-description>,
`g` is a <generation>.

**Step 1.** `g` must be a connected <generation>. The scalar-elements of `g` and the scalar-elements described by `odd` must each contain <aligned> and either (1) each contain <assembling> and <bit>,

or (2) each contain either <picture>, or <assembling> and <character>.

**Step 2.** Let `d` be the <declaration> designated by the <declaration-designator> of `vr`. If the <defined> component of `d` contains <position>, perform evaluate-expression(e), where `e` is the <expression> of <position>, and convert the result to integer-type of value `y`. Otherwise, let `y` be 1. The value of `y` must be greater than or equal to 1.

**Step 3.** Perform overlay-string ngại<odd, g, p> to obtain a <storage-index-list>, nil. Let `gl` be the <generation> comprising the allocation-unit-designator of `g`, a copy of `odd`, and nil.

**Step 4.** If `vr` immediately contains an <identifier-list>, then perform select-qualified-reference<gl, id1, d> to obtain a <generation>,`gl`.

**Step 5.** If `vr` immediately contains a <by-name-parts-list>, then perform evaluate-by-name-parts-list(g, vr, d) to obtain a <generation>,`gl`.

**Step 6.** If `vr` immediately contains a <subscript-list>, then perform select-subscripted-reference<gl, sb1> to obtain a <generation>,`gl`.

**Step 7.** Return `gl`.

7.6.16 CHECK-SIMPLE-DEFINED-REFERENCE

This operation checks that the relationship between the <declaration> of a <variable-reference> to a variable that is <defined> and the <declaration> referenced in the <base-item> is suitable for evaluation as a simply-defined reference.

**Operation:** check-simply-defined-reference(vr)

where `vr` is a <variable-reference>.

**result:** <true> or <false>.

**Step 1.** Let `d` be the <declaration> designated by the <declaration-designator> of `vr`. Let `dd` be a copy of the <data-descriptions> immediately contained in the <variable> of `d`. Replace each <bound-pair> in `dd` by <bound-pair>: <asterisk>, each <max-length> by <max-length>: <asterisk>, and each <alias-size> by <size>: <asterisk>. Let `pd` be

<parameter-descriptor>:

`dd`.

**Step 2.** Let `bvr` be the <variable-reference> of the <defined> component of `dd`.

**Step 3.** Perform text-matching(bvr, pd). If the value obtained is <true> then return <true>; otherwise return <false>.
7.6.17 EXTRACT-SLICE-OF-ARRAY

This operation selects the part of a `storage-index-list` or `basic-value-list` according to a `bound-pair-list`.

Operation: `extract-slice-of-array(obpl, sbpl, n, nil)`

where `obpl` is a `bound-pair-list`,
`sbpl` is a `bound-pair-list`,
`n` is an integer,
`nil` is a `storage-index-list` or `basic-value-list`.

result: a `storage-index-list` or `basic-value-list`.

Step 1. For each `<bound-pair>, obpl[i], i=1,...,m, in obpl let `obibpli` and `obithpli` be the `<integer-value>` components of the `<upper-bound>` and `<lower-bound>`, respectively, of `obpl[i].`

Step 2. For each `<bound-pair>, sbpl[i], i=1,...,m, in sbpl let `sibpli` and `sibpli` be the `<integer-value>` components of the `<upper-bound>` and `<lower-bound>`, respectively, of `sbpl[i].`

Step 3. Let `n2` be

\[ n = \prod_{i=1}^{m} (sibpli - obibpli + 1). \]

If `nil` is a `storage-index-list` then let `nil` be a `storage-index-list` with all `storage-index` elements. Otherwise, let `nil` be a `basic-value-list` with `n2` elements of the form `basic-value<undefined>`. 

Step 4. For each set of integers `a[i], i=1,...,m,` such that `a[i] ≤ nil ≤ a[i]+1`, perform steps 4.1 and 4.2.

Step 4.1. Let

\[ k = \frac{1}{a[1]} \sum_{i=1}^{m} (sibpli - obibpli) \prod_{j=1}^{i-1} (a[j] - a[j+1] + 1). \]

and let

\[ k2 = \frac{1}{a[1]} \sum_{i=1}^{m} (sibpli - obibpli) \prod_{j=1}^{i-1} (n - (a[j] - a[j+1] + 1)). \]

Step 4.2. Replace `nil[i]` by `nil[i]*k` for `i=0,...,m-1`.

Step 5. Return `nil`.
7.7 Reference to Named Constant

7.7.1 EVALUATE-NAMED-CONSTANT-REFERENCE

This operation obtains the value of a <named-constant-reference>.

**Operation:**

```
evaluate-named-constant-reference(ncr)
```

where ncr is a <named-constant-reference>.

_result:_ an <aggregate-value>.

**Step 1.** Let d be the <declaration> designated by the <Declaration-Designator> of ncr and let nc be a copy of the <named-constant> component of d.

**Step 2.**

**Case 2.1.** nc contains a <bound-pair-list>, bpl.

**Step 2.1.1.** For each <expression>, e, of bpl, chosen in any order, perform evaluate-expression-to-integer() to obtain the <integer-values>, i and replace e by i.

**Step 2.1.2.** For each <bound-pair>, bpl_i in bpl, let ubl_i and lbl_i be the integer components of the <upper-bound> and <lower-bound>, respectively, of bpl_i.

**Step 2.1.3.** Let m be the integer

```
\prod_{i=1}^{m} (ubl_i-lbl_i+1)
```

and let svl be a <basic-value-list> consisting of m <undefined> elements.

**Case 2.2.** (Otherwise).

**Step 2.2.1.** Let m be 1, and let svl be a <basic-value-list> consisting of one <undefined> element.

**Step 3.**

**Case 3.1.** nc has an <entry> component and d has <external>.

Let ep be the <entry-point> that is an element of the <entry-or-executable-unit-list> that is immediately contained in a <procedure> of the <abstract-external-procedure-list> and that has a <statement-name> with an <identifier> that is equal to the <identifier> of d. Let epd be an <entry-point-designator> designating ep and replace the <undefined> component of svl by epd.

**Case 3.2.** nc has an <entry> component and d has <internal>.

**Step 3.2.1.** Perform find-block-state-of-declaration(d) to obtain the <block-state>, bs. Let pl be the <procedure-list> of the <Begin-block> or <procedure> that simply contains d.

**Step 3.2.2.** For each <entry-point>, ep that is simply contained in pl and whose <statement-name> has an <identifier> that is equal to the <identifier> of d perform Steps 3.2.2.1 and 3.2.2.2.

**Step 3.2.2.1.** Let epd be an <entry-point-designator> designating ep and let bod be a <block-state-designator> designating bs. Let svd be an <entry-value> epd bod.
Step 3.2.2.2:

Case 3.2.2.2.1. \( \text{ep} \) has a <signed-integer-list>, nil.

Let \( \text{nil} \) be the \( i \)’th element of \( \text{si} \). Let \( m \) be the integer

\[
\sum_{i=1}^{n-1} (\text{nil}_{i+1} - \text{nil}_{i}) \times \text{nil}_{i}
\]

where

\[
m = \prod_{j=1}^{m} (\text{nil}_{j} - \text{nil}_{j-1})
\]

Replace the <undefined> component of the \( n \)’th element of
\( \text{svi} \) by \( ev \).

Case 3.2.2.2.2. (Otherwise).

Replace the <undefined> component of \( \text{svi} \) by \( ev \).

Case 3.3. \( \text{nc} \) has a <file> component.

Perform search-file-directory(\( \text{acr}, \text{svi} \)) to obtain a <basic-value-list>, \( \text{bvi} \).

Replace \( \text{svi} \) by \( \text{bvi} \).

Case 3.4. \( \text{nc} \) has a <format> component.

Step 3.4.1. Perform find-block-state-of-declaration(\( d \)) to obtain the <block-state>, \( f_a \). Let \( f_a \) be the <format-statement-list> component of the <begin-block> or <procedure> that simply contains \( d \).

Step 3.4.2. For each <format-statement>, \( f_s \), element of \( f_a \) that has a <statement-name> whose <identifier> is equal to the <identifier> of \( d \), perform Steps 3.4.2.1 and 3.4.2.2.

Step 3.4.2.1. Let \( f_d \) be a <format-statement-designator> designating \( f_s \) and let \( f_b \) be a <block-state-designator> designating \( f_a \). Let \( f_v \) be a <format-value>; find \( f_b \).

Step 3.4.2.2.

Case 3.4.2.2.1. The <statement-name> of \( f_s \) has a <signed-integer-list>, nil.

Let \( \text{nil} \) be the \( i \)’th element of \( \text{si} \). Let \( m \) be the integer

\[
\sum_{i=1}^{n-1} (\text{nil}_{i+1} - \text{nil}_{i}) \times \text{nil}_{i}
\]

where

\[
m = \prod_{j=1}^{m} (\text{nil}_{j} - \text{nil}_{j-1})
\]

Replace the <undefined> component of the \( n \)’th element of
\( \text{svi} \) by \( fv \).

Case 3.4.2.2.2. (Otherwise).

Replace the <undefined> component of \( \text{svi} \) by \( fv \).
Case 3.5. No has a <label> component.

Step 3.5.1. Perform find-block-state-of-declaration(d) to obtain the <block-state>, bs. Let env be the <executable-unit-list> or <entry-or-executable-unit-list> of the <begin-block> or <procedure>, respectively, that simply contains d.

Step 3.5.2. For each <executable-unit>, eu, component of env that has a <statement-name> whose <identifier> is equal to the <identifier> of d, perform Steps 3.5.2.1 and 3.5.2.2.

Step 3.5.2.1. Let end be an <executable-unit-designator> designating eu, and let be a <block-state-designator> designating bs. Let itv be a <label-value>; end end.

Step 3.5.2.2.

Case 3.5.2.2.1. The <statement-name> of eu has a <signed-integer-list>, al.

Let all be the i'th element of al. Let m be the integer

\[ \sum_{i=1}^{m-1} \frac{1}{(a[i]!b[i]!)} \]

where

\[ m = \prod_{j=1}^{n} (w[j]!z[j]!+1). \]

Replace the <undefined> component of the m'th element of env by itv.

Case 3.5.2.2.2. (Otherwise).

Replace the <undefined> component of env by itv.

Step 4.

Case 4.1. No contains <bound-pair-list>, bpl.

Let agt be

- <aggregate-type>,
- <dimensioned-aggregate-type>,
- <element-aggregate-type>,
- <scalar>.

bpl.

Case 4.2. (Otherwise).

Let agt be <aggregate-type> <scalar>.

Step 5. Let av be <aggregate-values> agt env.

Step 6. If ncr has a <subscript-list> then perform Steps 6.1 through 6.4.

Step 6.1. Let a, b[i], and n, i=1,...,n, be as determined in Step 2. Let bpl2 be a <bound-pair-list> with m elements of the form

- <bound-pair>:
  - <upper-bound>, n[i+1]
  - <lower-bound>, b[i+1];

for i=1,...,m. Perform Step 6.6.1 for i=1,...,m, taken in any order.
Step 6.1.1.

Case 6.1.1.1. The 1\textsuperscript{st} element of the \texttt{<subscript-list> in} \texttt{av} has a \texttt{<a3terior> -}

Let \texttt{lb}\texttt{ll} and \texttt{ub}\texttt{ll} be, respectively, \texttt{lb}\texttt{l1} and \texttt{ub}\texttt{l1}.

Case 6.1.1.2. The 1\textsuperscript{st} element of the \texttt{<subscript-list>} in \texttt{av} has an \texttt{<expression> -}

Perform evaluate-\texttt{expression-to-integer(\texttt{exp})} to obtain the \texttt{<integer-value>} \texttt{v and let \texttt{lb}\texttt{ll} and \texttt{ub}\texttt{ll} be the integer in \texttt{v. If \texttt{v in less than \texttt{lb}\texttt{l1 or greater than \texttt{ub}\texttt{l1 perform raise-condition〈subscribe-range-condition〉-}

Step 6.2. Perform extract-slice-of-array(\texttt{av\texttt{p1}}, \texttt{av\texttt{p2}}, \texttt{av\texttt{p3}}) to obtain a \texttt{<basic-value-list> \texttt{av\texttt{p2}}.}

Step 6.3. Replace the \texttt{<basic-value-list> in \texttt{av} by a copy of \texttt{av\texttt{p2}}.}

Step 6.4.

Case 6.4.1. Each element of the \texttt{<subscript-list>} in \texttt{av} has \texttt{<expression> -}

Replace the \texttt{<aggregate-type>} of \texttt{av} by \texttt{<aggregate-type>}: \texttt{<scalar> -}

Case 6.4.2. (Otherwise -)

Delete from the \texttt{<bound-pair-list>} of \texttt{av} the elements which correspond to elements of the \texttt{<subscript-list>} of \texttt{av} which have \texttt{<expression> -}

Step 7. The \texttt{<aggregate-value>} \texttt{av must not contain \texttt{<undefined>}. Return \texttt{av.}

3.7.2 SEARCH-FILE-DIRECTORY

This operation searches the \texttt{<file-directory>} for the entry or entries that correspond to the \texttt{<declaration>} referred to in a \texttt{<named-constant-reference> -}

Operation: \texttt{search-file-directory(\texttt{acv, avl})}

where \texttt{acv} is a \texttt{<named-constant-reference> -}

\texttt{avl} is a \texttt{<basic-value-list> -}

result: a \texttt{<basic-value-list> -}

Step 1. Let \texttt{avl} be a copy of \texttt{avl}. Let \texttt{d} be the \texttt{<declaration>} designated by the \texttt{<declaration-designator>} of \texttt{acv}. Perform Steps 1.1 through 1.2 until \texttt{avl} contains no \texttt{<undefined> elements.}

Step 1.1. Search the \texttt{<file-directory>} for a \texttt{<file-directory-entry>, e, whose \texttt{<identifier>} is identical with the \texttt{<identifier>} of \texttt{d and which has the component \texttt{<external> - If \texttt{d}} has the component \texttt{<external> it has a \texttt{<declaration-designator> component that designates \texttt{d.}

Step 1.2. Let \texttt{fid} be the \texttt{<file-information-designator> in \texttt{e and let \texttt{bv} be a \texttt{<basic-value> <file-value> \texttt{fid}. If \texttt{e has a <subscribe-value-list>, replace the element of \texttt{bv} designated by this <subscribe-value-list> by \texttt{bv. Otherwise, replace the single element of \texttt{bv by \texttt{bv.}

Step 2. Return \texttt{avl.}
Chapter 8: Input/Output

8.0 Introduction

This Chapter describes the abstract structure of a <dataset> and the transmission of data between a <dataset> and the <allocated-storage> of the <machine-state> directed by FL/I programs as introduced in Chapter 5. The main Sections are concerned with the following:

8.1 Datasets and the interface between them and the program
8.2 Files
8.3 Conditions applicable to I/O operations
8.4 Evaluation of a <file-option>
8.5 File opening and closing
8.6 Statements performing record transmission
8.7 Statements performing stream transmission, and a description of how data may be organized in the data stream

8.1 Datasets

A <dataset> is an abstract model of a physical dataset. Its properties and structure are those which are necessary for a correct interpretation of a FL/I program. The concrete-representation of a <dataset> is implementation-defined. <alpha> and <omega> are end-markers for <datasets> that have a sequence.

8.1.1 RECORD DATASETS

A <record-data> may contain discrete <records>; <record-data> without any <records> are permitted.

The "size" (see Section 8.6.6.11) is an implementation-defined function of the <evaluated-data-description> of a <record>. It is checked whenever the <records> is transmitted, and under implementation-defined circumstances any cause the raising of the <record-condition>.

<keys> are a means of identifying particular <records>. Within a <dataset>, <keys> are unique.

8.1.2 STREAM DATASETS

A <stream-item> is either a <symbol> or a control item which indicates (in an implementation-defined way) a line or page break or that the following <symbols> are to be sent to the same line as the preceding ones. The <stream-items> <page-mark> and <carriage-return> are not allowed in a <stream-item-list> associated with a file open for stream input and may only appear in a <stream-item-list> associated with a file open for print stream output.
8.2 Files

Whenever a P6/I program requires to access a <dataset>, it does so by naming a <file-option>. This <file-option> is evaluated yielding a <file-value>, fv, which designates a <file-information>, fi. In order successfully to access a <dataset>, fi must contain <open>, in which case it also contains a <file-opening> with a <dataset-designator>, ddesignates a <dataset>, which is thus accessed by naming the original <file-option>.

8.2.1 RECORD FILES

The <current-position> of a <file-opening> containing <record> contains either the <designator> of a component of a <record-dataset> or <undefined>. The designated node may be a <record>, a <keyed-record>, <alpha>, or <numeric>. One of the actions in executing a statement may be to update the <current-position>.

The <delete-flag>, which may be present in a <file-opening> when <record> appears, is an indication that certain actions must not be performed. (See, for example, the operation delete.)

The <allocated-buffer> in a <file-opening> contains a <generated> allocated by a <read-statement> or a <locate-statement> with the <pointer-set-option>. In the case of the <locate-statement> it may also contain a <key> to be associated with the <record> to be associated with the <allocated-buffer>.

8.2.2 STREAM FILES

The <current-position> of a <file-opening> containing <stream> contains either the <designator> of a component of a <stream-dataset> or <undefined>. The designated node may be a <stream-item>, <alpha>, or <numeric>. One of the actions in executing a statement may be to update the <current-position>.

The <page-number> component of a <file-opening> is applicable only when the <file-opening> also contains <stream> and <print>.

The <first-comma> component of a <file-opening> is applicable only when the <file-opening> also contains <stream> and <print>.

8.3 I/O Conditions

In describing the execution of each input/output statement, the circumstances under which any <io-condition>, except <transmit-condition>, may be raised, are indicated.

8.3.1 RAISE-IO-CONDITION

Operation: raise-io-condition(cod, fv, str, iat)

where cod in <endfile-condition>, <endpage-condition>, <key-condition>, <same-condition>, <record-condition>, <transmit-condition>, <undefinedfile-condition>, or <conversion-condition>.

fv is a <file-value>,

str is a [character-string-values],

iat is an [integer-values].

Step 1. If fv is a <file-value> then let fi be the <file-information> designated by fv.
Step 2.

Case 2.1. cond is not <conversion-condition>.

If fv is <absent> then perform raise-condition(error-condition).

Let chifs be an "evaluated-io-condition":
<io-condition>:
   cond:
   fv.

Case 2.2. cond is <conversion-condition>.

Let chifs be <conversion-condition>.

Step 3.

Case 3.1. cond is <name-condition>.

Let chifs be a <condition-bif-value-list>:
<condition-bif-value>:
<value-value>:
str.

Case 3.2. cond is <conversion-condition>.

Let chifs be a <condition-bif-value-list> simply containing <source-value>:
<str> and <char-value>:
int. If fv is a <file-value> then let fn be the <character-string-value> in the <filename> in fi and attach an <ofile-value>:
fn to chifs.

Case 3.3. cond is <transmit-condition>, <record-condition> or <key-condition>, and str is present.

Let chifs be a <condition-bif-value-list>:
<condition-bif-value>:
<key-value>:
str.

Case 3.4. (otherwise).

Let chifs be <absent>.


Step 5. If cond is <key-condition> or <endfile-condition> then perform exit-from-io(fv).

8.4 Evaluate-file-option

The evaluation of a <file-option> may be performed during input/output statements. This operation is also used to evaluate <copy-option>s and options of <con-statement>s, <equal-statement>s, and <assert-statement>s containing chained-io-conditions.

Operation: evaluate-file-option(vr)

   where vr is a <value-reference>.

   result: a <file-value>.

Step 1. Perform evaluate-value-reference(vr) to obtain an <aggregate-value>, av. Return the <file-value> in av.
8.5 File Opening and Closing

Opening a file causes a <file-opening> to be attached to the <file-information> and a <dataset> to be associated with the file. Closing a file removes the <file-opening> and dissociates the <dataset> from the file.

### 8.5.1 THE OPEN STATEMENT

#### 8.5.1.1 Execute-open-statement

**Operation:**

```plaintext
execute-open-statement(on)
where on is an <open-statement>.
```

**Step 1.** For each <single-opening>, set each open in on, in order, perform execute-single-opening<sep>.

**Step 2.** Perform normal-sequence.

#### 8.5.1.2 Execute-single-opening

**Operation:**

```plaintext
execute-single-opening<sep>
where ago is a <single-opening>.
```

**Step 1.** Let sfdi be an <evaluated-file-description-list> with no components.

**Step 2.** Perform Steps 2.1 to 2.5 in any order.

**Step 2.1.** Let opt be the <value-reference> in the <file-option> in ago. Perform evaluate-file-option<sto> to obtain a <file-value>, fv.

**Step 2.2.** If ago contains a <title-option>, tso, then perform evaluate-title-option<sto> to obtain an <evaluated-title-option>, et, and append <evaluated-file-descriptions>; et; to sfdi.

**Step 2.3.** If ago contains a <tab-option>, tso, then perform evaluate-tab-option<sto> to obtain an <evaluated-tab-option>, eto, and append <evaluated-file-descriptions>; eto; to sfdi.

**Step 2.4.** If ago contains a <linesize-option>, then let lso be its <expression>, perform evaluate-expression-to-integer<iso> to obtain an <integer-value>, ls, which must be greater than zero, and append <evaluated-file-descriptions>; <evaluated-linesize>; ls; to sfdi.

**Step 2.5.** If ago contains a <pagesize-option>, then let psd be its <expression>, perform evaluate-expression-to-integer<psd> to obtain an <integer-value>, ps, which must be greater than zero, and append <evaluated-file-descriptions>; <evaluated-pagesize>; ps; to sfdi.

**Step 3.** Let fi be the <file-information> designated by fv. If fi contains <open> then terminate this operation.

**Step 4.** For each immediate component, i.e., of ago which is a terminal node, append <evaluated-file-descriptions>; ia; to sfdi.

**Step 5.** Perform open(fi, sfdi) to obtain res.

**Step 6.** If res is <fail> then perform raise-in-condition(undefinedfile-condition), fv.
8.5.1.3 open

Operation: \( \text{open}(\text{fi}, \text{efdi}) \)

where \( \text{fi} \) is a \( \text{file-value} \),
\( \text{efdi} \) is an \( \text{evaluated-file-description-list} \).

result: \( \text{success} \) or \( \text{failure} \).

Step 1. Let \( \text{fi} \) be the \( \text{file-information} \) designated by \( \text{fi} \). For each terminal node, \( \text{tn} \), of the \( \text{file-description} \) of \( \text{fi} \), append \( \text{evaluated-file-description} \): \( \text{tn} \) to \( \text{efdi} \).

Step 2. Augment \( \text{efdi} \) with implied attributes as follows: for each terminal node in \( \text{efdi} \) which occurs under "Attribute" in the table below, append to \( \text{efdi} \) (tree) containing the corresponding "Implied Attributes" categories.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Implied Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;direct&gt;</td>
<td>&lt;record&gt; &lt;output&gt;</td>
</tr>
<tr>
<td>&lt;record&gt;</td>
<td>&lt;direct&gt; &lt;output&gt;</td>
</tr>
<tr>
<td>&lt;input&gt;</td>
<td>&lt;output&gt; &lt;update&gt;</td>
</tr>
<tr>
<td>&lt;stream&gt;</td>
<td>&lt;output&gt; &lt;update&gt;</td>
</tr>
<tr>
<td>&lt;sequential&gt;</td>
<td>&lt;record&gt;</td>
</tr>
</tbody>
</table>

Step 3. Augment \( \text{efdi} \) with default attributes as follows: when \( \text{efdi} \) does not contain any of the "Alternative Attributes" in a line of the table below, append to \( \text{efdi} \) a tree containing the corresponding "Default" category.

<table>
<thead>
<tr>
<th>Alternative Attributes</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;input&gt;</td>
<td>&lt;record&gt;  &lt;output&gt;</td>
</tr>
</tbody>
</table>

Step 4. If \( \text{efdi} \) contains \( <record> \) but does not contain either \( <sequential> \) or \( <direct> \), append \( <sequential> \) to \( \text{efdi} \).

Step 5. If the \( \text{filename} \) in \( \text{fi} \) contains in order precisely the symbols of the word \( \text{STEPWISE} \), and the \( \text{file-directory-entry} \) whose \( \text{file-information-designator} \) designates \( \text{fi} \) has \( <external> \), and \( \text{efdi} \) contains \( <stream> \) and \( <output> \), then append \( \text{evaluated-file-description} \): \( <print> \) to \( \text{efdi} \).

Step 6. If both \( <stream> \) and \( <record> \), or any two of \( <input> \), \( <output> \), and \( <update> \), or both \( <direct> \) and \( <sequential> \) are contained in \( \text{efdi} \), then return \( \text{failure} \).

Step 7. If \( \text{efdi} \) contains an \( <\text{evaluated-line-size}> \) then it must contain \( <\text{stream}> \) and \( <\text{output}> \). If \( \text{efdi} \) contains \( <\text{stream}> \) and \( <\text{output}> \) but not an \( <\text{evaluated-line-size}> \) then append \( \text{evaluated-file-description} \): \( <\text{evaluated-line-size}> \) an implementation-defined \( <\text{integer-value}> \); to \( \text{efdi} \).

If \( \text{efdi} \) contains an \( <\text{evaluated-page-size}> \) then it must contain \( <\text{print}> \). If \( \text{efdi} \) contains \( <\text{print}> \) but not an \( <\text{evaluated-page-size}> \) then append \( \text{evaluated-file-description} \): \( <\text{evaluated-page-size}> \) an implementation-defined \( <\text{integer-value}> \); to \( \text{efdi} \).

If \( \text{efdi} \) contains an \( <\text{evaluated-tab-option}> \) then it must contain \( <\text{print}> \). If \( \text{efdi} \) contains \( <\text{print}> \) but not an \( <\text{evaluated-tab-option}> \) then append \( \text{evaluated-file-description} \): \( <\text{evaluated-tab-option}> \) an implementation-defined \( <\text{integer-value-list}> \); to \( \text{efdi} \).

If \( \text{efdi} \) does not contain an \( <\text{evaluated-title}> \) then let \( \text{fn} \) be the \( \text{filename} \) in \( \text{fi} \) and perform evaluate-filemame(str)[fn] to obtain an \( <\text{evaluated-title}>[\text{t}] \), and append \( \text{evaluated-file-description} \): \( \text{t} \) to \( \text{efdi} \).
Step 8. Attempt to find, among the &dataset& of the &machine-state& (if any), a &dataset& whose &character-string-value& of the &dataset-name& of which matches the &character-string-value& of the &evaluated-title& in &efdi& in an implementation-defined manner. If the attempt fails, then return &fail&.

If &efdi& contains &record& then &ds& must contain a &record-dataset&.
If &efdi& contains &stream& then &ds& must contain a &stream-dataset&.
If &efdi& contains &sequential& and &keyed& then &ds& must contain a &keyed-sequential-dataset&.
If &efdi& contains &sequential& and &record& but not &keyed& then &ds& must contain a &sequential-dataset&.
If &efdi& contains &direct& then &ds& must contain a &keyed-dataset&.
If &efdi& contains &stream& and &input& then &ds& must not contain any &comment& or &carriage-return&.
If &efdi& contains &record& and &keyed& then &ds& must not contain two distinct &keyed-records& whose &keys& are equal.

Step 9. Attach to &fi& a

&file-opening& for:

&dataset-designator&
&complete-file-description&
&efdi&.

The &dataset-designator& designates &ds& which has been associated with &fi& in &Step 8&. If &efdi& contains &record& then attach a &delete-flag& to &fo&. If &efdi& contains &print& then attach a &page-number& with &i& to &fo&. If &efdi& contains &stream& and &input& then attach &first-comma& &one& to &fo&.

Step 10.

Case 10.1. &efdi& contains &direct&.

Attach to &fo& a &current-position& &undefined&.

Case 10.2. &efdi& contains &named& and not &direct& and &ds& contains a &record-list& &ri& or a &keyed-record-list& &ri& or a &stream-item-list& &ri&.

Attach to &fo& a &current-position& designating the last immediate component of &ri&.

Case 10.3. Otherwise.

Attach to &fo& a &current-position& designating the &alpha& in &ds&.

Step 11. Set the &open-state& in &fi& to contain &open&.

Step 12. Return &success&.

8.5.1.4 Evaluate-tab-option

Operation: evaluate-tab-option(tbo)

where &tbo& is a &tab-option&.

result: an &evaluated-tab-options&.

Step 4. For each &expression& in &tbo& in any order, perform evaluate-expression-to-integer to obtain an &integer-value&. Let &lit& be an &integer-values-list& containing these &integer-values& in the same order as their original &expression& in &tbo&.
Step 2. The <integer-values> in 11 must each be greater than zero, and the list must be in ascending order.


8.5.1.5 Evaluate-title-option

Operation: evaluate-title-option(t)

   where t is a <title-option>.

   result: an <evaluated-title>.

Step 1. Let e be the <expression> in t. Perform evaluate-expression to obtain an <aggregate-value>, av. Let d be the <data-description> immediately contained in e.

Step 2. Let dt be the <data-type> in d and let td be a <data-type> containing <character>, <converting> and an implementation-defined <maximum-length>. Let dv be the <basic-value> in av. Perform convert(d, td, dv) to obtain a <character-string-value>, csv.


8.5.1.6 Evaluate-filename

Operation: evaluate-filename(fn)

   where fn is a <filename>.

   result: an <evaluated-title>.

Step 1. Let s be the <character-string-value> in fn and let m be the number of <character-values> in s. Let n be the implementation-defined maximum length of the <evaluated-title>.

Step 2.

Case 2.1. $m = 0$.

   Let csv be <character-string-value>: <null-character-string>.

Case 2.2. $m > 0$.

   Let csv be <character-string-value>: <character-value-list> where the length of <character-value-list> is $m$ and where the $i$'th <character-value> is the same as the $i$'th <character-value> in s, $i=1,...,m$.

Case 2.3. $m < n$.

   Let csv be <character-string-value>: <character-value-list> where the length of <character-value-list> is $n$ and where the $i$'th <character-value> is the same as the $i$'th <character-value> in s for $i=1,...,m$, and where the remaining <character-values> have $\#$.

8.5.2 THE CLOSE STATEMENT

8.5.2.1 Execute-close-statement

Operation: execute-close-statement(cs)

where cs is a <close-statement>.

Step 1. For each <single-closing>, sc in cs, in order, perform execute-single-closing(sc).

Step 2. Perform normal-sequence.

8.5.2.2 Execute-single-closing

Operation: execute-single-closing(sc)

where sc is a <single-closing>.

Step 1. Let fo be the <value-reference> in the <file-option> in sc. Perform evaluate-file-option(fo) to obtain a <file-value>, fv.

Step 2. If the <file-information> designated by fv contains <open> then perform close(fv).

8.5.2.3 Close

Operation: close(fv)

where fv is a <file-value>.

Step 1. Let fi be the <file-information> designated by fv, and let ds be the <dataset-designator> designated by the <dataset-designator> in fi.

Step 2. If fi contains <output> and an <allocated-buffer>, abuf containing the <generation>, q then perform Steps 2.1 through 2.3.

Step 2.1. If abuf contains a <key>, then let k be a copy of that <key>; otherwise k is <absent>. Perform construct-record(q,k) to obtain r.

Step 2.2. If k is a <key> then if there is a <key> in ds equal to k, or if k is unacceptable to the implementation, then:

Case 2.2.1. This operation is preceded in its operation-list by an operation for execute-single-closing.

Perform raise-lo-condition(key-condition),fv, osv, where osv is the <character-string-value> in k.

Case 2.2.2. This operation is preceded in its operation-list by an operation for program-epilogue.

Perform some implementation-defined action and go to Step 4.

Step 2.3. Perform insert-record(r,fv).

Step 3. If abuf is present then perform free(fo) and delete abuf.

Step 4. Delete the <file-opening> in fi, and set the <open-state> in fi to contain <closed>. 
8.6 The Record I/O Statements

The record I/O statements perform data transmission to and from <record-dataset>s. Several of the record I/O statements use common operations. These are described in Section 8.6.6. Several local variables are used in Section 8.6 in a consistent manner:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abf</td>
<td>for allocated-buffer</td>
</tr>
<tr>
<td>d</td>
<td>for declaration</td>
</tr>
<tr>
<td>ds</td>
<td>for record-dataset</td>
</tr>
<tr>
<td>edd</td>
<td>for evaluated-data-description</td>
</tr>
<tr>
<td>elo</td>
<td>for evaluated-into-option</td>
</tr>
<tr>
<td>ekp</td>
<td>for evaluated-keyoption</td>
</tr>
<tr>
<td>epsao</td>
<td>for evaluated-pointer-set-option</td>
</tr>
<tr>
<td>efddl</td>
<td>for evaluated-file-description-list</td>
</tr>
<tr>
<td>fl</td>
<td>for file-information</td>
</tr>
<tr>
<td>fn</td>
<td>for filename</td>
</tr>
<tr>
<td>fvr</td>
<td>for file-value</td>
</tr>
<tr>
<td>q</td>
<td>for generations</td>
</tr>
<tr>
<td>int</td>
<td>for integer-value</td>
</tr>
<tr>
<td>k</td>
<td>for key</td>
</tr>
<tr>
<td>kr</td>
<td>for key-record or record</td>
</tr>
<tr>
<td>pos</td>
<td>for current-position</td>
</tr>
<tr>
<td>f</td>
<td>for record</td>
</tr>
</tbody>
</table>

8.6.1 THE READ STATEMENT

Purpose: The <read-statement> causes a <record> to be transmitted from a <record-dataset> to a target <generation> or an <allocated-buffer>.

8.6.1.1 Execute-read-statement

<evaluated-read-statement> ::= <file-value>

- <evaluated-into-option> |
- <evaluated-pointer-set-option> |
- <evaluated-ignore-option> 
- <key> |
- <evaluated-key-option>

Operation: execute-read-statement(rs)

where rs is a <read-statement>.

Step 1. Let err be an <evaluated-read-statement> without subnodes.

Step 2. Perform steps 2.1 through 2.6 in any order.

Step 2.1. Let f be the immediate component of the <file-option> in rs. Perform evaluate-file-option(f) to obtain a <file-value>,fv. Attach fv to err.

Step 2.2. If rs contains an <into-option>, i.e., perform evaluate-into-option(terminal) to obtain an <evaluated-into-option>,eito and attach eito to err.

Step 2.3. If rs contains a <pointer-set-option>,eps, perform evaluate-pointer-set-option(pso) to obtain an <evaluated-pointer-set-option>,epso and attach epso to err.

Step 2.4. If rs contains an <ignore-option>,igo, perform evaluate-ignore-option(igo) to obtain an <evaluated-ignore-option>,eigo and attach eigo to err.

Step 2.5. If rs contains a <key-option>,kso, perform evaluate-key-option(ksol) to obtain a <key>,k and attach k to err.

Step 2.6. If rs contains a <key-option>,kto, perform evaluate-keyto-option(ktos) to obtain an <evaluated-keyto-option>,ekto and attach ekto to err.

Step 3. If the <file-information> f1, designated by fv contains <open> then go to step 5.
Step 8.

Step 8.1. Let efdl be an «evaluated-file-description-list» containing <record>.

Step 8.2. If fi does not contain <update> then attach «evaluated-file-description» <input> to efdl.

Step 8.3. Perform open(FV, efdl) to obtain rf. If rf is «fail» then perform raise-condition «io-condition» «errdefn» «io-condition» «errval». If, on normal return, rf contains «closed» then perform raise-condition «errort-condition» «errval».

Step 5. fi must contain:

<record>;
<input> or <update>;
if ers contains an «evaluated-ignore-option» or if ers
does not contain a «key» then «sequential»;
if ers contains a «key» or as «evaluated-keyto-option», then «header».

Step 6. Perform readest.

Step 7. Perform normal-sequence.

8.6.1.2 Read

Operation: readest

where ers is an «evaluated-read-statement».

Step 1. Let fi be the «file-information» designated by the «file-value» FV, in ers. fi must contain «open» and «allocated-buffer» ADRF. Let k be the «key» and k1 be the immediate component of k. Otherwise let k and k1 be «absent».

Step 2. If fi contains an «allocated-buffer» ADRF, then perform frees(FR), where FR is the
<generation> in ADRF, and delete ADRF from fi.

Step 3. Perform position-fileterm().

Step 4. If ers does not contain an «evaluated-keyto-option» EKO, then go to Step 5. Otherwise let c be the number of «symbol» subnodes of the «character-string-value» CVRK in the «key» in the «keyed-record» designated by the «current-position» in fi. Let c be a «data-description» containing «character».

Case 5.1. ers contains an «evaluated-into-option» EKO.

Let r be the «record» or «keyed-record» designated by the «current-position» in fa.

Step 5.1.1. Let e1 be the «evaluated-data-description» in r and e2 be the «evaluated-data-description» in ers. Perform evaluatesize(addl) to obtain an «integer-value» intl and perform evaluate-sized(adr) to obtain an «integer-value» int2.

Case 5.1.2.1. intl is equal to int2.

Perform Steps 5.1.2.1.1 and 5.1.2.1.2 in either order.

Step 5.1.2.1.1. addl and add2 must be equal. Perform set-storage(v, vi) where
v is the «basic-value-list» in r.

Step 5.1.2.1.2. If ers contains an «evaluated-keyto-option» then perform
assignkeyserset(add2, ADL).
Case 5.1.2.2. (Otherwise).

Step 5.1.2.2.1. Let nsi be the number of <storage-indexes> in g. Let undef be a <basic-value-list> containing <basic-values>: <undefined>; nsi times. Perform Steps 5.1.2.2.1.1 and 5.1.2.2.1.2 in either order.

Step 5.1.2.2.1.1. Perform set-storage(g, undef).

Step 5.1.2.2.1.2. If err contains an <evaluated-keyo-option> then perform assign(ehov, avk, ddi).

Step 5.1.2.2.2. Perform raise-lo-condition<record-condition>, fV, kli.

Case 5.2. err has an <evaluated-pointer-set-options>, epso.

Perform Steps 5.2.1 through 5.2.5 in any order such that Step 5.2.1 precedes Step 5.2.2, Step 5.2.3, and Step 5.2.4.

Step 5.2.1. Let r be the <record> or <keyed-record> designated by the <current-position> in fl. Let e be the <evaluated-data-description> in r. Perform allocate(fed) to obtain a <generation>, e.

Step 5.2.2. Let abf be an <allocated-buffer> g. Attach abf to the <file-opening> in fl.

Step 5.2.3. Let v be the <basic-value-list> in r. Perform set-storage(g, v).

Step 5.2.4. Let pd be a <data-description> simply containing <pointer> and no other terminal subnodes. Let epso be an <evaluated-target> containing the <generation> contained in epos. Let avg be an <aggregate-value> containing the <pointer-value> g. Perform assign(ehov, avk, pd).

Step 5.2.5. If err contains an <evaluated-keyo-option> then perform assign(ehov, avk, ddi).

Case 5.3. (Otherwise).

If err contains an <evaluated-keyo-option> then perform assign(ehov, avk, ddi).

8.6.2 The Write Statement

Purpose: The <write-statement> causes a <record> or <keyed-record> to be transmitted from a <generation> or an <allocated-buffer> to a <record-dataset>.

8.6.2.1 Execute-write-statement

<execute-write-statement>::= <file-value> <evaluated-from-option> 
<evaluated-keyfrom-option>

Operation: execute-write-statement(ws)

where ws is a <write-statement>.

Step 1. Let ew be a <evaluated-write-statement> without subnodes.

Step 2. Perform Steps 2.1 through 2.3 in any order.

Step 2.1. Let f be the immediate component of the <file-option> in ws. Perform evaluate-file-option(f) to obtain a <file-value>, fV. Attach fV to ew.

Step 2.2. Let fV be the <from-option> in ws. Perform evaluate-from-option(fV) to obtain an <evaluated-from-option>, ofo and attach ofo to ew.

Step 2.3. If we contains a <keyfrom-option>, ko, then perform evaluate-keyfrom-option(ko) to obtain an <evaluated-keyfrom-option>, ekfo and attach ekfo to ew.
Step 3. If the <file-information>, fi, designated by fv contains <open> then go to Step 5.

Step 4.

Step 4.1. Let efdl be an <evaluated-file-description-list> containing <record>.

Step 4.2. If fi does not contain <update> then attach <evaluated-file-description>: <content> to efdl.

Step 4.3. Perform open(fv, efdl) to obtain af. If af is <fail> then perform raise-ionic-condition(<underdefined-file-condition>, fv). If on normal return fi contains <closed> then perform raise-ionic-condition(<error-condition>).

Step 5. fi must contain:

<record>;<content>, or <update> and <direct>:
if and only if ewv contains an <evaluated-keyfrom-option>, then <keyed>.

Step 6. Perform write(fw).

Step 7. Perform normal-sequence.

8.4.1.2 Write

Operation: write(fw)

where ewv is an <evaluated-write-statement>.

Step 1. Let fi be the <file-information> designated by the <file-value>, fv, in ewv. fi must contain <open>. If fi contains an <evaluated-keyfrom-option>; <character-string-value>, cvv; then let k be a <key>; cvv. Otherwise let k be <absent>. Let ds be the <record-dataset> designated by the <dataset-designator> in fi.

Step 2.

Case 2.1. fi does not contain an <allocated-buffer> or does not contain <output>.

Go to Step 3.

Case 2.2. fi contains an <allocated-buffer>, abuf, and fi contains <output>.

Step 2.2.1. Let q be the <generation> immediately contained in abuf. If abuf contains a <key>, let kbe this <key> and let cwp be the immediate component of abuf; otherwise let k be <absent> and cwp be <absent>. Perform construct-record, kbe, to obtain kr.

Step 2.2.2. If fi contains <keyed> and if kwa is equal to any <key> in the <dataset>, ds, or if kwa is unacceptable to the implementation then perform raise-ionic-condition(<key-condition>, fv, cvv).

Step 2.2.3. Perform insert-record(kr,fw) to obtain a <designator>, pos.

Step 2.2.4. Replace the immediate component of the <current-position> in fi with pos.

Step 3. If fi contains an <allocated-buffer>, abuf, containing a <generation>, q, then perform free(q) and delete abuf from fi.

Step 4. Let q be the <generation> in the <evaluated-from-option> in ewv. Perform construct-record, kw, to obtain kr.

Step 5. If fi contains <keyed> then if k is equal to any <key> in ds or if k is unacceptable to the implementation then perform raise-ionic-condition(<key-condition>, fv, cvv).

Step 6. Perform insert-record(kr,fv) to obtain a <designator>, pos.

Step 7. Replace the immediate component of the <current-position> in fi by pos.

236
S.6.3 THE LOCATE STATEMENT

Purpose: The <locate-statement> causes allocation of the specified base variable in an allocated-buffer; it may also cause transmission of a base variable previously allocated in an allocated-buffer.

S.6.3.1 Execute-locate-statement

<evaluation-locate-statement>::= <declaration-designator> <file-value>
                             |<evaluation-pointer-set-option>
                             |<evaluation-keyfrom-option>

Operation: evaluate-locate-statement(is)
            where is is a <locate-statement>.

Step 1. Let els be an <evaluation-locate-statement> without subscript.

Step 2. Perform Steps 2.1 through 2.4 in any order.

Step 2.1. Let f be the immediate component of the <file-option> in is. Perform evaluate-file-option(f) to obtain a <file-value>,f. Attach f[v] to els.

Step 2.2. If is contains a <pointer-set-option>,pso, then perform evaluate-pointer-set-option(pso) to obtain an <evaluation-pointer-set-option>,epto and attach epto to els.

Step 2.3. If in contains a <keyfrom-option>,kfo, then perform evaluate-keyfrom-option(kfo) to obtain an <evaluation-keyfrom-option>,ekf0 and attach ekf0 to els. If fi contains a <keyed> then let chs be a copy of the immediate component of ekf0; otherwise let chs be an absent. If fi contains a <keyed> then let xk be the key; chs.

Step 2.4. Let cd[ch] be a copy of the <declaration-designator> immediately contained in ib. Attach cd[ch] to els.

Step 3. If the <file-information>,fi, designated by f[v] contains a <open> then go to Step 5.

Step 4.

Step 4.1. Let eddl be an <evaluation-file-description-list> containing <record> and <output>.

Step 4.2. Perform open(f[v],eddl) to obtain sf. If sf is equal to <fail>, then perform raise-condition(undeffile-condition),(f[v]). If, on normal return, fi contains <closed> then perform raise-condition(error-condition).

Step 5. Fi must contain:

<record>
<output>
if and only if els contains an <evaluation-keyfrom-option>, then <keyed>.

Step 6. If els does not contain an <evaluation-pointer-set-option> then the <declaration-designator> contained in els must designate a <declaration>,d, of the form

<base>
  <value-reference>,v
  <variable-reference>,p,
  <declaration-designator>,p;

p must designate a <declaration> containing <pointer>. Perform evaluate-variable-reference(v) to obtain q. Let epto be an <evaluation-pointer-set-option>; g; and attach epto to els.

Step 7.

Step 7.1. If f[v] does not contain an <allocated-buffer>,abuf, then go to Step 8.
Step 7.2. Let q be the <generation> immediately contained in abuf. If abuf contains a <key>, let k be this <key> and let cwb be its immediate component; otherwise let k and cwb be <absent>. Perform construct-record(q,k) to obtain kr.

Step 7.3. If fi contains <keyed> then if k is equal to any <key> in the <record-dataset> designated by the <dataset-designator> in fi, or if k is unacceptable to the implementation, then perform raise-lo-condition(<key-condition>,fv,cwb).

Step 7.4. Perform insert-record(kr,fv) to obtain pos.

Step 7.5. Replace the immediate component of the <current-position> in fi with pos.

Step 7.6. Perform free(q) and delete abuf from fi.

Step 8. Let dd be the <data-description> immediately contained in the <variable> of the <declaration> designated by cdq. Perform evaluate-data-description-for-allocation(dd) to obtain edd.

Step 9. Perform evaluate-size(dd) to obtain an <integer-value>,int. If int is unacceptable to the implementation then perform raise-lo-condition(<record-condition>,f,v,ch) and optionally perform exit-from-fi.

Step 10. Perform allocate(dd) to obtain g.

Step 11. Let dsc be a <data-description> simply containing <pointer> without other terminal subnodes. Let epseq be an <evaluated-target> containing the <generation> in the <evaluated-pointer-set-option> in els. Let aeq be an <aggregate-value> containing <pointer-value>: g. Perform assign(epseq,apeq,dsc).

Step 12. Let d be the <declaration> designated by the <declaration-designator> in els.

Step 12.1. If the <aggregate-type> of g contains <structure-aggregate-type> then perform initialize-refer-options(g).

Step 12.2. Perform initialize-generation(q,d).

Step 13. Let abuf be an <allocated-buffer>: <generation>,q. If fi contains <keyed> then attach k to abuf. Attach abuf to the <file-opening> in fi.

Step 14. Perform normal-sequence.

8.6.4 THE REWRITE STATEMENT

Purpose: The <rewrite-statement> causes replacement of an existing <record> or <keyed-record> in a <record-dataset>.

8.6.4.1 Execute-rewrite-statement

<rewrited-rewrite-statement> ::= <file-value>
                 {<key> <rewrited-from-option>}

Operation: execute-rewrite-statement(rws)

where rws is a <rewrite-statement>.

Step 1. Let rws be an <rewrited-rewrite-statement> without subnodes.

Step 2. Perform Steps 2.1 through 2.3 in any order.

Step 2.1. Let f be the immediate component of the <file-option> in rws. Perform evaluate-file-option(f) to obtain a <file-value>,fv. Attach fv to rws.

Step 2.2. If rws contains a <rewrited-from-option>,fr, then perform evaluate-from-option(fr) to obtain an <rewrited-from-option>,sfo and attach sfo to rws.
Step 2.3. If the string contains a <key-option>, k, then perform evaluate-key-option(k) to obtain a <key>, k, and attach k to errors.

Step 3. If the <file-information> fi, designated by fi, contains <open> then go to Step 5.

Step 4.

Step 4.1. Let sf1 be an <evaluated-file-description-list> containing <record> and <update>.

Step 4.2. Perform open(fi, sf1) to obtain sf. If sf is <fail> then perform raise-condition(<undefined-file-condition>, fi). If an error occurs, return sf contains <closed> then perform raise-condition(<error-condition>).

Step 5. fi must contain:

<record>
<update>
if errors does not contain a <key>, then <sequential>
if errors contains a <key>, then <keyed>.

If fi contains <direct> then errors must contain a <key>. If fi contains <sequential> then errors may contain a <key> and an <evaluated-from-option>.


Step 7. Perform normal-connection.

6.6.4.1 Rewrite

Operation: rewrite-term

where errors is an <evaluated-rewrite-statement>.

Step 1. Let fi be the <file-information> designated by the <file-value>, fi in errors. fi must contain <open>. If errors contains a <key>, then let k be this <key>; otherwise let k be <absent>. If k is a <key>, then let cv be its immediate component; otherwise let cv be <absent>.

Step 2.

Case 2.1. k is <absent>.

fi must not contain a <delete-flag> and the <current-position> in fi must not contain <undefined>. If the <current-position> in fi designates a <keyed-record>, kr then let k be a copy of the <key> in kr and let cv be the immediate component of k.

Case 2.2. k is a <key>.

Perform position-file-term.

Step 3.

Case 3.1. errors contains an <evaluated-from-option>, afo.

Let g be the <generation> in afo.

Case 3.2. Otherwise.

fi must contain an <allocated-buffer>, af. Let g be the <generation> in afbuf.

Step 4. Perform construct-record(g, k) to obtain r.

Step 5. Let addl be the evaluated-data-description in g and addl the <evaluated-data-description> in the <record> designated by the <current-position> in fi. Perform evaluate-size(addl2) to obtain an <integer-value, int1> and evaluate-size(addl2) to obtain an <integer-value, int2>.

Chapter 5: Input/Output 239
Step 6. Let rd be the <record> or <keyed-record> designated by the <current-position> in fi.

Case 6.1. int1 and int2 are equal.
   Replace rd by r.

Case 6.2. int1 and int2 are not equal.
   If rd is a <record>, replace rd by an implementation-defined <record>; otherwise replace rd by an implementation-defined <keyed-record> with an equal <key>. Perform raise-lo-condition(<record-condition>, fv, cv), and optionally perform exit-from-io.

Step 7. If fi contains an <allocated-buffer>, abort, then let g be the <generation> in abort, perform free(g), and delete abort from fi.

8.6.5 THE DELETE STATEMENT

Purpose: The <delete-statement> deletes a <record> or <keyed-record> from a <record-dataset>.

8.6.5.1 Execute-delete-statement

<evaluated-delete-statement> := <file-value> [key]

Operation: execute-delete-statement(dis)
   where dis is a <delete-statement>.

Step 1. Let eds be an <evaluated-delete-statement> without subnodes.

Step 2. Perform Steps 2.1 and 3.2 in either order.

Step 2.1. Let f be the immediate component of the <file-option> in dis. Perform evaluate-file-option(f) to obtain a <file-value>, fv. Attach fv to eds.

Step 2.2. If dis contains a <key-option>, ko, then perform evaluate-key-option(ko) to obtain a <key>, k and attach k to eds.

Step 3. If the <file-information>.fi, designated by fv contains <open> then go to Step 5.

Step 6.

Step 4.1. Let edf1 be an <evaluated-file-description-list> containing <update> and <record>.

Step 4.2. Perform open(fv, edf1) to obtain sf. If sf is <fail> then perform raise-lo-condition(<undeclared-file-condition>, fv). If on normal return fi contains <closed> then perform raise-condition(<error-condition>).

Step 5. fi must contain:

<record>
<update>
   if eds contains a <key>, then <keyed>;
   if eds does not contain a <key>, then <sequential>.

   If fi contains <direct> then eds must contain a <key>. If fi contains <sequential> then eds may contain a <key>.

Step 6. Perform delete(eds).

Step 7. Perform normal-sequence.
8.6.5.2 DELl TE

Operation: \textit{delete\{eds\}}

where \textit{eds\}} is an \textit{<evaluated-delete-statement>\}}.

Step 1. Let \textit{fi} be the \textit{<file-information>\}} designated by the \textit{<file-value>\}}, \textit{fv}, in \textit{eds\}}. \textit{fi} must contain \textit{<open>\}}. If \textit{eds\}} contains a \textit{<key>\}}, then let \textit{k} be this \textit{<key>\}} and let \textit{csv} be the immediate component of \textit{k}; otherwise let \textit{k} and \textit{csv} be \textit{<absent>\}}.

Step 2.

Case 2.1. \textit{k} is \textit{<absent>\}}.

\textit{fi} must not contain a \textit{<delete-flag>\}}. The \textit{<current-position>\}} in \textit{fi} must not contain \textit{<undefined>\}}.

Case 2.2. \textit{k} is a \textit{<key>\}}.

Perform position-\textit{file\{eds\}}.

Step 3. Let \textit{ds} be the immediate component of the \textit{<record-dataset>\}} designated by the \textit{<dataset-designator>\}} in \textit{fi}. Let \textit{kr} be the node designated by the \textit{<current-position>\}} in \textit{fi} (\textit{kr} is a \textit{<record>\}} or a \textit{<keyed-record>\}}). Delete \textit{kr} from \textit{ds}.

Step 4.

Case 4.1. \textit{fi} contains \textit{<direct>\}}.

Replace the immediate subnode of the \textit{<current-position>\}} in \textit{fi} by \textit{<undefined>\}}.

Case 4.2. \textit{fi} contains \textit{<essential>\}}.

Replace the immediate subnode of the \textit{<current-position>\}} in \textit{fi} by a \textit{<designator>\}} designating the predecessor of \textit{kr} in \textit{ds} (this may be \textit{<alpha>\}} a \textit{<record>\}} or a \textit{<keyed-record>\}}).

Step 5. Attach a \textit{<delete-flag>\}} to the \textit{<file-opening>\}} in \textit{fi}.

Step 6. If \textit{fi} contains an \textit{<allocated-buffer>\}} of \textit{abuf}, then let \textit{g} be the \textit{<generation>\}} in \textit{abuf}, perform free\{g\}, and delete \textit{abuf} from \textit{fi}.

8.6.6 OPERATIONS APPLICABLE TO RECORD I/O

8.6.6.1 Evaluate-from-option

\textit{<evaluated-from-option>\} := \textit{<generation>\}}

Operation: \textit{evaluate-from-option\{fr\}\}}

\textit{result\} as \textit{<evaluated-from-option>\}}.

Step 1. Let \textit{v} be the \textit{<variable-reference>\}} immediately contained in \textit{fr}. Perform evaluate-variable-reference\{v\} to obtain a \textit{<generation>\}} \textit{g}, which must be connected.

Step 2. Return as \textit{<evaluated-from-option>\} g.\}}

Chapter 8: Input/Output 243
8.6.6.2 Evaluate-into-option

<evaluated-into-option>::= <generation>

Operation: evaluate-into-option(ito)

where ito is a <into-option>.

result: an <evaluated-into-option>.

Step 1. Let v be the <variable-reference> immediately contained in ito. Perform evaluate-variable-reference(v) to obtain a <generation>, q, which must be connected.

Step 2. Return an <evaluated-into-option> q.

8.6.6.3 Evaluate-pointer-set-option

<evaluated-pointer-set-option>::= <generation>

Operation: evaluate-pointer-set-option(pso)

where pso is a <pointer-set-option>.

result: an <evaluated-pointer-set-option>.

Step 1. Let v be the <variable-reference> immediately contained in pso. Perform evaluate-variable-reference(v) to obtain a <generation>, q.

Step 2. Return an <evaluated-pointer-set-option> q.

8.6.6.4 Evaluate-key-option

Operation: evaluate-key-option(ko)

where ko is a <key-option>.

result: a <key>.

Step 1. Let e be the <expression> immediately contained in ko. Perform evaluate-expression(e) to obtain res, containing a <basic-value>, bv.

Step 2. Let tt be a <data-type> containing <character>, <hexadecimal>, and <maximum-length>: <asterisk>. Let rt be the <data-type> of e. Perform convert(ttt, rt, bv) to obtain a <character-string-value>, cha.

Step 3. Return a <key> cha.

8.6.6.5 Evaluate-keyfrom-option

<evaluated-keyfrom-option>::= <character-string-value>

Operation: evaluate-keyfrom-option(kfo)

where kfo is a <keyfrom-option>.

result: an <evaluated-keyfrom-option>.

Step 1. Let e be the <expression> immediately contained in kfo. Perform evaluate-expression(e) to obtain res, containing a <basic-value>, bv.

Step 2. Let tt be a <data-type> containing <character>, <hexadecimal>, and <maximum-length>: <asterisk>. Let rt be the <data-type> of e. Perform convert(ttt, rt, bv) to obtain a <character-string-value>, cha.
Step 3. Return an `<evaluated-keyfrom-option>`.  

8.6.6.6 Evaluate-ignore-option

<evaluated-ignore-option>::=<integer-value>

Operation: evaluate-ignore-option(igo)

   where igo is an `<ignore-option>`.
   result: an `<evaluated-ignore-option>`.

Step 1. Let u be the `<expression>` immediately contained in igo. Perform evaluate-expression-to-integer(u) to obtain an `<integer-value>`, int, which must not be negative.

Step 2. Return an `<evaluated-ignore-option>`: int.

8.6.6.7 Evaluate-keyto-option

<evaluated-keyto-option>::=<evaluated-target>

Operation: evaluate-keyto-option(kto)

   where kto is a `<keyto-option>`.
   result: an `<evaluated-keyto-option>`.

Step 1. Let kto be the `<target-reference>` in kto. Perform evaluate-target-reference(kto) to obtain an `<evaluated-target>`, et.

Step 2. Return an `<evaluated-keyto-option>`: et.

8.6.6.8 Construct-record

Operation: construct-record(q,k)

   where q is a connected `<generation>`,
   k is a `<key>`.
   result: a `<record>` or a `<keyed-record>`.

Step 1. Let avg be an `<aggregate-value>` which is the value of q. (See Section 7.1.3.) Let bvl be the `<evaluated-data-description>` in q. Let bvl be the `<basic-value-list>` in avg. Let r be a `<record>`; add bvl.

Step 2.

Case 2.1. k is `<absent>`

   Return r.

Case 2.2. k is a `<key>`

   Return `<keyed-record>`: r k.
8.6.9 Insert-record

Operation: \texttt{insert-record} \langle x, y, w \rangle

where \texttt{x} is a \texttt{record} or a \texttt{keyed-record},
\texttt{y} is a \texttt{file-value},
\texttt{z} is a \texttt{designator}.

result: a \texttt{designator}.

Step 1. If \texttt{x} is a \texttt{keyed-record} then let \texttt{k} be the \texttt{key} in \texttt{x} and let \texttt{cv} be the
immediate component of \texttt{k}; otherwise let \texttt{k} and \texttt{cv} be \texttt{absent}. Let \texttt{f} be the
\texttt{file-information} designated by \texttt{f}. Let \texttt{i} be the \texttt{record-list} or \texttt{keyed-
record-list} in the \texttt{dataset} designated by the \texttt{dataset-designator} in \texttt{f}. Let \texttt{a} be the
\texttt{file-data-description} in \texttt{x}. Perform evaluate-designator to
obtain an \texttt{integer-value-list}. If \texttt{i} is not acceptable to the implementation
then perform Step 1.1.

Step 1.1. If \texttt{x} is a \texttt{record}, let \texttt{k} be a new implementation-defined \texttt{record};
otherwise let \texttt{k} be an implementation-defined \texttt{keyed-record} with an \equal
\texttt{key}. Optionally perform Step 2. If this operation is preceded in its
immediately containing \texttt{operation-list} by an \texttt{operation} for program-
epilogue then perform some implementation-defined action; otherwise perform
raise-io-condition(\texttt{record-condition}, \texttt{f}, \texttt{cv}). Optionally perform exit-
from-\texttt{io}.

Step 2.

Case 2.1. \texttt{i} contains \texttt{direct}.

Attach \texttt{x} to \texttt{i} in a position chosen in an implementation-defined way.

Case 2.2. \texttt{i} contains \texttt{keyed} and \texttt{sequential}.

Attach \texttt{x} to \texttt{i} in a position defined which may depend on
the \texttt{key} in \texttt{x}, on the \texttt{record} in \texttt{x} and the \texttt{current-position} in \texttt{i}.

Case 2.3. \texttt{i} contains \texttt{sequential} and not \texttt{keyed}.

Append \texttt{x} to \texttt{i} in the position immediately following the immediate
component of \texttt{i} designated by the \texttt{current-position} in \texttt{i}.

Step 3. Return a \texttt{designator} designating \texttt{x} in \texttt{i}.

8.6.10 Position-file

Operation: \texttt{position-file} \langle \texttt{event} \rangle

where \texttt{event} is an \texttt{evaluated-read-statement}, an \texttt{evaluated-rewrite-
statement} or an \texttt{evaluated-delete-statement}.

Step 1. If \texttt{event} contains a \texttt{key} then let \texttt{k} be this \texttt{key} and let \texttt{cv} be its immediate
component; otherwise let \texttt{k} and \texttt{cv} both be \texttt{absent}. Let \texttt{f} be the \texttt{file-value} in \texttt{event}, let \texttt{i} be the
\texttt{file-information} designated by \texttt{f}, let \texttt{d} be the
\texttt{record-dataset} in the \texttt{dataset} designated by the \texttt{dataset-designator} in \texttt{f},
and let \texttt{pos} be the \texttt{current-position} in \texttt{i}.

Step 2.

Step 2.1. If \texttt{k} is a \texttt{key} and \texttt{k} is unacceptable to the implementation then perform
raise-io-condition(\texttt{key-condition}, \texttt{f}, \texttt{cv}).

Step 2.2.

Case 2.2.1. \texttt{k} is \texttt{absent}.

Go to Step 3.
Case 2.2.2. k is a <key> and k is not equal to any <key> in ds.

Step 2.2.2.1. If fl does not contain a <delete-flag>, then attach a <delete-flag> to the <file-opening> in fl.

Step 2.2.2.2. Replace the immediate component of pos by <undefined>.

Step 2.2.2.3. Perform raise-io-condition(chaev-condition, fl, cv).

Case 2.2.3. k is equal to a <key> in a <keyed-record>, k, in ds.

Step 2.2.3.1. If fl contains a <delete-flag>, del, then delete del from fl.

Step 2.2.3.2. Replace the immediate component of pos by a <designator> designating kr.

Step 3.

Case 3.1. exist is an <evaluated-read-statement> containing an <evaluated-ignore-option>, sigo.

Let int be the <integer-value> in sigo.

Case 3.2. exist is an <evaluated-read-statement> not containing a <key> or an <evaluated-ignore-option>.

Let int be an <integer-value> with value 1.

Case 3.3. (Otherwise).

Terminate this operation.

Step 4. pos must contain a <designator>, rdes.

Step 4.1. If int is 0, then terminate this operation.

Step 4.2.

Case 4.2.1. rdes designates <omega>.

Perform raise-io-condition(omega-condition, fl).

Case 4.2.2. rdes designates the last element of the <keyed-record-list> or <record-list> in ds.

Replace rdes by a <designator> designating the <omega> in ds.

Case 4.2.3. (Otherwise).

Replace rdes by a <designator> designating the next <keyed-record> or <record> in ds.

Step 4.3. If rdes designates <omega> then perform raise-io-condition(omega-condition, fl).

Step 4.4. If fl contains a <delete-flag> then delete it from fl.

Step 4.5. Decrement int by 1. Go to Step 3.

8.4.4.11 Evaluate-nize

Operation: evaluate-nize(odd)

where odd is an <evaluated-data-description>,

result: an <integer-value>.

Step 1. Return an implementation-defined <integer-value>, depending on odd.
8.4.4.12 Exit-from-io

Operation: \texttt{exit-from-io}(fv)

where \(fv\) is a \texttt{<file-value>}

Step 1. Let \(fi\) be the \texttt{<file-information>} designated by \(fv\). If \(fi\) contains an
\texttt{<allocated-buffer>}, abuf, containing a \texttt{<generation>}, g, then perform free(g) and
delete abuf from \(fi\).

Step 2. Perform \texttt{trim-io-control}.

Step 3. Let \(x\) be the current \texttt{<executable-unit-designator>}. Perform \texttt{trim-group-control}(x).

Step 4. Replace the immediate component of the current \texttt{<statement-control>} by

\texttt{<operation-list>}

\texttt{<operation>} for \texttt{advance-execution}

\texttt{<operation>} for \texttt{normal-sequence}

8.4.4.13 Trim-io-control

Operation: \texttt{trim-io-control}

Step 1. Let \(bc\) be the current \texttt{<block-control>}

Step 2. If \(bc\) contains a \texttt{<data-item-control-list>}, dcl, then delete dcl from \(bc\).

Step 3. If \(bc\) contains a \texttt{<current-scalar-item-list>}, call, then delete call from \(bc\).

Step 4. If \(bc\) contains a \texttt{<string-io-control>}, sio, then delete sio from \(bc\).

Step 5. If \(bc\) contains a \texttt{<format-control-list>}, fcl, then delete fcl from \(bc\).

Step 6. If \(bc\) contains a \texttt{<create-block-state>}, cbs, then delete cbs from \(bc\).

Step 7. If \(bc\) contains a \texttt{<current-file-value>}, cfv, then delete cfv from \(bc\).
8.7 The Stream I/O Statements

8.7.1 The Get Statement

8.7.1.1 Execute-get-statement

Operation: \texttt{execute-get-statement}(s)

where \( s \) is a \texttt{<get-statement>}. \\

Step 1.

Case 1.1. \( s \) has a \texttt{<get-file>}, \( qf \).

Perform \texttt{execute-get-file}(qf).

Case 1.2. \( s \) has a \texttt{<get-string>}, \( qstr \).

Perform \texttt{execute-get-string}(qstr).

Step 2. Perform trim-to-control.

Step 3. Perform normal-sequence.

8.7.1.2 Execute-get-file

Operation: \texttt{execute-get-file}(qf)

where \( qf \) is a \texttt{<get-file>}. \\

Step 1. Perform Steps 1.1, 1.2, and 1.3 in any order.

Step 1.1. Let \( f \) be the \texttt{<value-reference>} in the \texttt{<file-option>} in \( qf \). Perform \texttt{evaluate-file-option}(f) to obtain \texttt{<file-value>}, \( fv \).

Step 1.2. If \( qf \) contains a \texttt{<skip-option>}, \( sko \), then let \( s \) be the \texttt{<expression>} in \( sko \), and perform \texttt{evaluate-expression-to-integer}(s) to obtain an \texttt{<integer-value>}, \( sk \).

Step 1.3. If \( qf \) contains a \texttt{<copy-option>}, \( co \), then let \( cv \) be the \texttt{<value-reference>} in \( co \), and perform \texttt{evaluate-file-option}(cv) to obtain \texttt{<file-value>}, \( cf \).

Step 2. Let \( fi \) be the \texttt{<file-information>} designated by \( fv \), and, if a \texttt{<copy-option>} is present in \( qf \), let \( cfi \) be the \texttt{<file-information>} designated by \( cf \). If \( fi \) contains \texttt{<open>} then go to Step 4.

Step 3. Let \( efdl \) be an \texttt{<evaluated-file-description-list>} containing \texttt{<stream>} and \texttt{<input>}. Perform \texttt{open} \( efdl \) to obtain \( rf \). If \( rf \) is \texttt{<fail>} then perform \texttt{raise-lo-condition}(\texttt{<undefinedfile-condition>}, \( fvl \)). If an normal return \( fi \) contains \texttt{<closed>} then perform \texttt{raise-condition}(\texttt{<error-condition>}).

Step 4. \( fi \) must contain \texttt{<stream>} and \texttt{<input>}. 

Step 5. If \( qf \) does not have a \texttt{<copy-option>} then go to Step 9. If \( cfi \) contains \texttt{<open>} then go to Step 7.

Step 6. Let \( efdl \) be a \texttt{<evaluated-file-description-list>} containing \texttt{<stream>} and \texttt{<output>}. Perform \texttt{open} \( efdl \) to obtain \( rcf \). If \( rcf \) is \texttt{<fail>} then perform \texttt{raise-lo-condition}(\texttt{<undefinedfile-condition>}, \( cfi \)). If an normal return \( fi \) contains \texttt{<closed>} then perform \texttt{raise-condition}(\texttt{<error-condition>}).

Step 7. \( cfi \) must contain \texttt{<stream>} and \texttt{<output>}. 

Step 8. Attach a \texttt{<copy-file>}, \( cf \), to the current \texttt{<block-state>}. 

Step 9. If \( qf \) has a \texttt{<skip-option>} then perform \texttt{skip}(s), \( tv \).
Step 10. If qf has an <input-specification> then:

Case 10.1. qf has a <list-directed-input>, idi.
Perform get-list(idi, fvi).

Case 10.2. qf has a <data-directed-input>, ddi.
Perform get-data(ddi, fvi).

Case 10.3. qf has an <edit-directed-input>, edi.
Perform get-edited(edi, fvi).

Step 11. If qf has a <copy-option> then delete the current <copy-file>.

Step 12. If qf has a <list-directed-input> then terminate this operation. Otherwise set the <first-comma> in fi to contain qf.

8.7.1.3 Execute-get-string

Operation: execute-get-string(qstr)

where qstr is a <get-string>.

Step 1. Perform Steps 1.1 and 1.2 in either order.

Step 1.1. Let e be the <expression> immediately contained in qstr. Perform evaluate-expression(e), to obtain an <aggregate-value>, av. Let dt be the <data-type> in the <data-description> immediate component of e. Let sv be the <basic-value> in av. Let tdt be a <data-type> simply containing <character>, <varying>, and maximum-length; <string>. Perform convert(tdt, idt, sv) to obtain a <character-string-values>, cvv. Attach to the current <block-control> a <string-ic-control>; cvv <first-comma>; <typ>.

Step 1.2. If qstr has a <copy-option>, co, then perform Steps 1.2.1 through 1.2.4.

Step 1.2.1. Let cv be the <value-reference> in co. Perform evaluate-file-option(cv) to obtain a <file-values>, cf. Let cf be the <file-information> designated by cf. If cf contains <open> then go to Step 1.2.3.

Step 1.2.2. Let sdf be an <evaluated-file-description-list> containing <stream> and <output>. Perform open(sdf, cv) to obtain a result rcf. If rcf is <fail> then perform raise-io-condition(<undefined-file-condition>, cf). If on normal return rcf contains <closed> then perform raise-condition(<error-condition>).

Step 1.2.3. cf must contain <stream> and <output>.

Step 1.2.4. Attach a <copy-file>: cf; to the current <block-state>.

Step 2.

Case 2.1. qstr has a <list-directed-input>, idi.
Perform get-list(idi).

Case 2.2. qstr has a <data-directed-input>, ddi.
Perform get-data(ddi).

Case 2.3. qstr has an <edit-directed-input>, edi.
Perform get-edited(edi).

Step 3. If qstr has a <copy-option> then delete the current <copy-file>.
§7.1.3 get-list

Operation:  get-list(1di, fv)

where 1di is a <list-directed-input>,
       fv is a <file-value>.

Step 1. Attach to the current <block-control> a

   <data-item-control-list>:
   <data-item-control>: the <data-list-indicator> designating the <input-target-list> of 1di
   <data-item-indicator>:
   <undefined>.

Step 2. Perform establish-seux-data-item to obtain a <current-scalar-item>, mdi or
         <error>, mdi. If mdi is <error> then terminate this operation.

Step 3. Perform parse-list-input(fv) to obtain a <character-string-values>, csv.

Step 4. If fv is a <file-value> then attach to the current <block-control> a <current-
        file-values>, fv.

Case 4.1. csv contains just one terminal which is a $, $, or a <null-character-string>.

   Go to Step 6.

Case 4.2. The terminal nodes of csv can be parsed as "<non-blank-comma-quotes> <non-
           blank-comma-list>".

   Let v be csv. Let st be <character>.

Case 4.3. The terminal nodes of csv can be parsed as "<simple-character-string-
           constant>".

   Perform basic-character-values(csv) to obtain a <character-string-value>, v.
   Let st be <character>.

Case 4.4. The terminal nodes of csv can be parsed as "<simple-bit-string-constant>".

   Perform basic-bit-values(csv) to obtain a <bit-string-value>, v. Let st be <bit>.

Case 4.5. (Otherwise).

   Let intg be the smallest integer such that the <character-string-value> of
   length intg containing the first intg <character-values> of csv does not
   have a continuation conforming, and does not itself conform, to either of
   the syntaxes: <simple-character-string-constant> or <simple-bit-string-
   constant>.

   Perform raise-lo-condition(<conversion-condition>, fv, csv, intg). On normal
   return let csv be the immediate component of the current <returned-source-
   values>; csv must not contain any blanks: go to Step 4.

Step 5. Let et be the <evaluated-target> in mdi. Let ssv be
Let d1 be

\[
\text{let } d1 = \text{data-description}.
\]

Step 5: If fv is a \text{file-value} then delete the current \text{current-file-value}. Go to Step 4.

8.2.1.3.2 Parse-list-input

Operation: \text{parse-list-input}(fv)

where fv is a \text{file-value}.

result: a \text{character-string-value}.

Step 1.

Case 1.1. \text{fv} is a \text{file-value}.

Let fi be the \text{file-information} designated by fv, and let ds be the \text{dataset-designator} designated by the \text{dataset-designator} in fi. Let cp be the \text{current-position} in fi.

Case 1.1.1. There is no \text{stream-item-list} in ds or cp designates the last \text{stream-item} in the \text{stream-item-list} in ds.

Perform \text{raise-io-condition}(\text{dead-file-condition}, fv).

Case 1.1.2. \text{Otherwise}.

Let sl be a \text{stream-item-list} containing the \text{stream-items} in the \text{stream-item-list} in ds, following the \text{stream-item} designated by cp.

Case 1.2. \text{fv} is \text{current}.

Let sl be the \text{character-string-value} in the current \text{string-io-control}. If sl contains no \text{symbol} then perform \text{raise-condition}(\text{error-condition}).

Step 2.

Case 2.1. sl can be parsed as "[\text{leading-delimiter-list}] . \text{[stream-item-list]}".

Let \text{fd} be that sequence in sl which satisfies "[\text{leading-delimiter-list}] . " in this parse, and let its number of terminal nodes be \( n \). Perform input: \text{stream-item}(fv) \( n \) times. If the current \text{first-comma} exists and contains \text{cov} or the \text{first-comma} in the \text{file-information}, \text{fi} exists and contains \text{cov} then return \text{character-string-value}: \text{character-value-list}; \text{character-value}: \text{symbol}: \text{[value]}: \text{[value]}: \text{[value]}. \text{Otherwise} set the current \text{first-comma} (respectively, the \text{first-comma} in \text{fi}) to contain \text{cov}, perform \text{parse-list-input}(fv) to obtain \text{cov}, and return \text{cov}.

250
Case 2.2. \(s_l\) can be parsed as "(leading-delimiter-list)".

Let \(l\) be the number of terminal nodes in \(s_l\). Perform \(\text{input-stream-item}(f_v)\) \(l\) times. If the current \(<\text{first-comma}>\) exists and contains \(<og\), or the \(<\text{first-comma}>\) in \(f_l\) exists and contains \(<og\) then return \(<\text{character-string-value}>\) \(<\text{null-character-string}>\). Otherwise go to Step 1.

Case 2.3. \(s_l\) can be parsed as "(leading-delimiter-list) + (string-symbol-or-linemark-list)".

Let \(l\) be the number of terminal nodes in \(s_l\). Perform \(\text{input-stream-item}(f_v)\) \(l\) times. Perform \(\text{raise-condition}(\text{error-condition})\).

Case 2.4. \(s_l\) can be parsed as "(leading-delimiter-list) (putative-list-constant) [(R,) ] (stream-item-list)".

Let \(l\) be the number of terminal nodes in \(s_l\) preceding that part which satisfies "(stream-item-list)" in this parse. Perform \(\text{input-stream-item}(f_v)\) \(l\) times. Let \(c_v\) be a \(<\text{character-string-value}>\) whose terminal nodes are those of the part of \(s_l\) which satisfies "(putative-list-constant) [(R,) ]" except those terminal nodes that are \(<\text{linemark}>\). Let \(c_f\) be the current \(<\text{first-comma}>\), if that exists; otherwise let \(c_f\) be the \(<\text{first-comma}>\) in \(f_l\). If the last \(<\text{symbol}>\) of \(c_v\) is a \(,\) then set \(c_f\) to contain \(<og>\); otherwise set \(c_f\) to contain \(<\text{off}>\). Delete from \(c_v\) the last \(<\text{symbol}>\), if that \(<\text{symbol}>\) is a \(,\) or a \(\{\). Return \(c_v\).

8.7.3.4.2 Parsing Categories for List Directed Input

Some of the categories used in parsing input streams for list directed input are categories of the Concrete Syntax or the Machine-state Syntax. Others are defined as follows:

\[
\text{leading-delimiter} ::= \# | \text{linemark}
\]

\[
\text{putative-list-constant} ::= \text{(string-symbol-or-linemark-list)}\]

\[
\text{data-symbol} ::= \text{(letter | digit | _ | + | - | [ ] | = | ; | , | & | % | \{| | \} | \# | < | < | > | / | * | | \$ | \$ | \text{pseudographical-character})}
\]

Note: The subnodes of \(\text{data-symbol}\) are those of \(\text{symbol}\) except for \(\#\), \(\{\}, \(\{\}, \(\}\), and \(\\})\.

\[
\text{non-blank-comma-quote} ::= \text{data-symbol} | | = | \text{linemark}
\]

\[
\text{non-blank-comma} ::= \text{non-blank-comma-quote} | \text{'}
\]

\[
\text{string-symbol-or-linemark} ::= \text{string-or-picture-symbol | linemark}
\]

Chapter 8: Input/Output 251
§ 7.1.3 Get-data

\{data-basic-reference\} ::=
\{data-structure-reference\}
\{data-structure-reference\} ::=
\{data-basic-reference\}.
\{data-structure-reference\} ::=
\{data-structure-reference\} \{data-subscript\}.
\{data-subscript\} ::=
\{data-subscript-chainlist\} \{data-list\}.
\{data-subscript\} ::=
\{data-list\} \{\text{-}\} \{integer\} \{data-list\}.
\{scalar-facts\} ::=
\{identifier-list\} \{declaration-designator\}
\{data-description\} \{integer-value\}.

Operation: \texttt{get-data(idl, fv)}

where \texttt{idl} is a \{data-directed-input\},
\texttt{fv} is a \{file-value\}.

Step 1. Perform \texttt{parse-data-input-name(fv)} to obtain a \{character-string-value\}, \textsf{vf}.

Step 2.

Case 2.1. \{data\} contains a \{null-character-string\} or has just one \{symbol\} which contains a \{\textsf{\textbackslash i}\}.

Go to Step 1.

Case 2.2. \{data\} has just one \{symbol\} which contains a \{\textsf{\textbackslash i}\}.

Terminate this operation.

Case 2.3. The last \{symbol\} of \{data\} contains an \{\textsf{\textbackslash i}\}.

Let \textsf{nsf} be \{symbol-list\} containing, in order, all the \{symbol\}s of \{data\} except its terminal \{symbol\}, \{\textsf{\textbackslash i}\}. If \textsf{nsf} is an empty list then let \textsf{nsf} be a \{null-character-string\}.

Case 2.4. (Otherwise).

Let \textsf{vf} be \{character-string-value\} \{null-character-string\}. If the last \{symbol\} of \{data\} contains a \{\textsf{\textbackslash i}\} then let \textsf{li} := 1; otherwise let \textsf{li} := 0. Go to Step 9.

Step 3. Perform \texttt{parse-data-input-value(fv)} to obtain a \{character-string-value\}, \textsf{vf}. If \textsf{vf} contains a \{symbol\} and its last \{symbol\} contains a \{\textsf{\textbackslash i}\} then let \textsf{li} := 1; otherwise let \textsf{li} := 0.

Step 4.

Step 4.1. If \textsf{nsf} conforms to the syntax for \{data-basic-reference\} then let \textsf{dbr} be a \{data-basic-reference\} whose terminal nodes are pairwise equal to the terminal nodes of \textsf{nsf}, both sets being taken in order. Otherwise go to Step 9.

Step 4.2. Let \textsf{wr} be a \{variable-reference\} without subnodes.

Step 4.2.1. Let \textsf{id1} be an \{identifier-list\} containing, in order, the \{identifier\}s in \textsf{dbr}. Let \textsf{id12} be an \{identifier-list\} without subnodes. For each \{identifier\}, \textsf{id}, in \textsf{id1} let \textsf{id} be the corresponding \{identifier\} and append \textsf{id} to \textsf{id12}. If \textsf{id12} contains more than one element, then let \textsf{id1} be a copy of \textsf{id12}, delete the first element of \textsf{id13}, and attach \textsf{id13} to \textsf{wr}.
Step 8.2.2. If $db$ does not contain any $\text{data-subscript}$s then let $n$ be $\text{absent}$ and go to Step 4.2.3.

Let $dl$ be a $\text{data-subscript-list}$ containing all the $\text{data-subscript}$s in all the $\text{data-subscript-list}$s in $db$, in order. Let $sdl$ be a $\text{subscript-list}$ without subscripts.

For each $\text{data-subscript}$, $x$, in $sdl$, in order, perform Step 8.2.2.1. Attach $x$ to $vr$.

Step 8.2.2.1. Let $cs$ be the $\text{character-string-value}$ corresponding to $x$. Perform basic-numeric-value-of-code to obtain a $\text{value-and-type}$ $rv$ $\text{data-type}$, $dt$. Let $sv$ be

$$<\text{expression}>$$
$$<\text{constant}>$$
$$<\text{basic-numeric-value}>$$
$$rv$$
$$dt$$
$$<\text{data-description}>$$
$$<\text{item-data-description}>$$

Step 8.2.3. Let $bio$ be the $\text{begin-block}$ or $\text{procedure}$ that simply contains the $\text{executable-unit}$ designated by the current $\text{executable-unit-designator}$.

Step 8.2.3.1. If $bio$ contains a $\text{declaration-list}$, $dl$, then go to Step 8.2.3.2.

Step 8.2.3.2. If $bio$ is not contained in a $\text{procedure}$ then go to Step 9. Let $bio$ be a $\text{begin-block}$ or $\text{procedure}$ containing $bio$ which contains no other $\text{begin-block}$ or $\text{procedure}$ containing $bio$. Go to Step 8.2.3.3.

Step 8.2.3.3. Let $sv$ be a $\text{scalar-facts-list}$ containing all of the distinct $\text{scalar-facts}$ which can be obtained by performing Steps 8.2.3.3.1 and 8.2.3.3.2.

Step 8.2.3.3.1. Let $d$ be a $\text{declaration}$ in $dl$. Let $sfx$ be a $\text{scalar-facts}$ containing a $\text{declaration-designator}$ designating $d$, an $\text{identifier-list}$, $id$; $id$; where $id$ is the $\text{identifier}$ immediately contained in $d$, and an $\text{integer-value}$; $id$. If $d$ contains a $\text{variable}$ which immediately contains a $\text{data-description}$, $dd$, then perform Steps 8.2.3.3.1.1 and 8.2.3.3.1.2.

Step 8.2.3.3.1.1. If $dd$ immediately contains a $\text{dimensioned-data-description}$, $dd$, then let $n$ be the number of $\text{bound-pair}$s in the $\text{bound-pair-list}$ of $dd$, add $n$ to the value of the $\text{integer-value}$ in $sfx$, replace that $\text{integer-value}$ with the sum so obtained, and let $dd$ be the $\text{element-data-description}$ in $dd$.

Step 8.2.3.3.1.2. If $dd$ contains a $\text{structure-data-description}$, $sdx$, which immediately contains an $\text{identifier-list}$, $id$, then choose an $\text{identifier}$, $id$, from $id$, append $id$ to $id$, let $dd$ be the $\text{data-description}$ in the $\text{member-description-list}$ corresponding to $id$ in the $\text{member-description-list}$ of $dd$, and go to Step 8.2.3.3.1.1.

Step 8.2.3.3.2. If $d$ contains a $\text{variable}$ then let $dd$ be the $\text{item-data-description}$ simply contained in $d$ and attach a $\text{data-description}$ $\text{item-data-description}$ to $sfx$. 

Chapter 8: Input/Output 253
Step 4.2.3.4. In this step a list will be said to be an ordered sublist of another list if the two lists are equal or if a list equal to the first list can be obtained by deleting one or more elements from the second.

If sfl2 is not an ordered sublist of the <identifier-list> of any <array-definition> in sfl then go to Step 4.2.3.2. If sfl2 is an ordered sublist of the <identifier-list> of more than one <scalar-facts> in sfl but is not equal to one of them, then go to Step 9. If sfl2 is equal to the <identifier-list> of a <scalar-facts>, sfs, then let idfl2 be that <identifier-list>. If idfl2 is an ordered sublist of the <identifier-list>, idfl, of exactly one <scalar-facts>, sfs in sfl, then idfl2 must equal idfl.

If sfl is a <substring-list> then let nsl be the number of <substring> in sfl. Otherwise let nsl be 0. If nsl is not equal to the value of the <integer-value> in sfs then go to Step 9.

Let d be the <declaration> designated by the <declaration-designator> in sfs. d must contain <declaration-type>; <variable>; and must not contain <base> without a subnode. The <data-description>.d. in sfs must contain an <integer-data-description> containing a <data-type>; <computational-type>.

Attach the <declaration-designator> in sfs to vr. Attach the <data-description> in sfs to vr.

Step 4.2.3.5. In this step a list is said to be an initial sublist of another list if the two lists are equal or if a list equal to the first can be obtained by deleting the last element of the second list one or more times.

If sfl does not contain a <data-target-list> then go to Step 4.1. Let vr1 be a <variable-reference-list> consisting of each <variable-reference> in a <data-target-list> of sfl whose <declaration-designator> equals that of vr and whose <identifier-list>, if present, has fewer elements than idfl. If vr1 is empty then go to Step 9. If any <variable-reference> in vr1 is without an <identifier-list> then go to Step 4.3. Otherwise, go to Step 9 unless some <variable-reference> in vr1 has an <identifier-list> which is an initial sublist of the <identifier-list> obtained by deleting the first <identifier> from idfl.

Step 4.3. Perform evaluate <variable-reference>(vr1) to obtain a <generation>.q.

Step 5. If vr is a <file-value> then attach <current-file-value>(vr) to the current <block-controls>. If vr contains a <null-character-string> then go to Step 8. If vr contains a single <symbol> ; then go to Step 6. If vr contains the single <symbol> ; then go to Step 8. If vr terminates in the <symbol> ; then remove this <symbol>.

Step 6. Case 6.1. The terminal nodes of vr can be parsed as "{non-blank-con-paren-quote}: {non-blank-con-paren-quote}.

Let v be vr. Let st be <character>.

Case 6.2. The terminal nodes of vr can be parsed as "{simple-character-string-con}: {simple-character-string-con}.

Perform basic-character-value(vr1) to obtain a <character-string-value>.v. Let st be <character>.

Case 6.3. The terminal nodes of vr can be parsed as "{simple-bit-string-constant}.

Perform basic-bit-value(vr1) to obtain a <bit-string-value>.v. Let st be <bit>.
Case 6.4.  (Otherwise).

Let \( i \) be the smallest integer such that the \(<\text{character-string-value}\>\) of length \( i \) containing the first \( i \) \(<\text{character-values}\>\) of \( \text{src} \) does not have a continuation conforming, and does not itself conform, to either of the syntaxes: \(<\text{simple-character-string-constant}\>) or \(<\text{simple-bit-string-constant}\>).

Perform \(<\text{raise-io-condition}\>(\text{conversion-condition}, \text{fv}, \text{v}, \text{intq})\). On normal return let \( \text{vf} \) be the immediate component of the current \(<\text{returned-resource-value}\>\); \( \text{vf} \) must not contain only blanks; go to Step 6.

Step 7.  Let \( \text{eq} \) be \(<\text{evaluated-target}\>\); \( \text{g} \). Let \( \text{agv} \) be

\[
\begin{align*}
\text{aggregate-value} :& \text{aggregate-type} : \text{scalar} : \\
\text{basic-value-list} :& \text{basic-value} : \text{v}.
\end{align*}
\]

Let \( \text{dd} \) be

\[
\begin{align*}
\text{data-description} :& \text{item-data-description} : \\
\text{data-type} :& \text{non-computational-type} : \\
\text{string} :& \text{string-type} : \text{at} ; \\
\text{maximum-length} :& \text{material} : \\
\text{counter-value} :& \text{counter-value} : \\
\end{align*}
\]

Perform \(<\text{assign}\>(\text{eq}, \text{agv}, \text{dd})\).

Step 8.  If \( \text{fv} \) is a \(<\text{file-value}\>\) then remove the current \(<\text{current-file-value}\>\). If \( \text{li} = 1 \) then terminate this operation; otherwise go to Step 1.

Step 9.  Perform \(<\text{raise-io-condition}\>(\text{case-condition}, \text{fv}, \text{str})\), where \( \text{str} \) is a \(<\text{character-string-value}\>\) containing, in order, the \(<\text{symbols}\>\) of \( \text{nf} \) and the \(<\text{symbols}\>\) of \( \text{vf} \) (excluding a final \(<\text{space}\>)\); \( \text{li} \); \( \text{if} \); \( \text{agv} \)). On normal return if \( \text{li} = 0 \) then go to Step 1; otherwise terminate this operation.

8.7.1.5.1 \text{Parse-data-input-name}

Operation:  \text{parse-data-input-name}(\text{fv})

where \( \text{fv} \) is a \(<\text{file-value}\>\).

results: a \(<\text{character-string-value}\>\).

Step 1.

Case 1.1. \( \text{fv} \) is a \(<\text{file-value}\>\).

Let \( \text{fi} \) be the \(<\text{file-information}\>\) designated by \( \text{fv} \), and let \( \text{do} \) be the \(<\text{dataset}\>\) designated by the \(<\text{dataset-designator}\>\) in \( \text{fi} \). Let \( \text{cp} \) be the \(<\text{current-position}\>\) in \( \text{fi} \).

Case 1.1.1.  There is no \(<\text{stream-item-list}\>\) in \( \text{do} \), or \( \text{cp} \) designates the last \(<\text{stream-item}\>\) in the \(<\text{stream-item-list}\>\) in \( \text{do} \).

Perform \(<\text{raise-io-condition}\>(\text{endfile-condition}, \text{fv})\).

Case 1.1.2.  (Otherwise).

Let \( \text{ali} \) be a \(<\text{stream-item-list}\>\) containing, in the same order, \(<\text{stream-items}\>\) equal to the \(<\text{stream-items}\>\) in the \(<\text{stream-item-list}\>\) in \( \text{do} \) which follow the \(<\text{stream-item}\>\) designated by \( \text{cp} \).

Chapter 8: Input/Output 255
Case 1.2. \( fv \) is \(<\text{absent}>\).

Let \( ai \) be the \(<\text{character-string-value}>\) in the current \(<\text{string-io-control}>\).
If \( ai \) contains no \(<\text{symbol}>\) then perform raise-condition\(<\text{error-condition}>\).

Step 2.

Case 2.1. \( ai \) can be parsed as \("[\text{leading-delimiter-list}]\{,;|\} \text{[stream-item-list]}\"\).

Let \( ld \) be that sequence in \( ai \) which satisfies \("[\text{leading-delimiter-list}]\{,;|\} \text{=}\"\) in this parse, and let the number of its terminal nodes be \( n \). Perform input-stream-items\(i\) times. Let \( endld \) be the last terminal node in \( ld \). Return \(<\text{character-string-value}>\); \(<\text{character-value-list}>\); \(<\text{character-value}>\); \(<\text{symbol}>\); \text{endld}.

Case 2.2. \( ai \) can be parsed as \("[\text{leading-delimiter-list}][\text{putative-name-field}]\{,;|\} \text{[stream-item-list]}\text{]}\"\).

Let \( ln \) be the number of terminal symbols in \( ai \). Perform input-stream-items\(f\) times. Return \(<\text{character-string-value}>\); \(<\text{null-character-string}>\).

Case 2.3. \( ai \) can be parsed as \("[\text{leading-delimiter-list}][\text{putative-name-field}]\{,;|\} \text{[stream-item-list]}\text{]}\"\).

Let \( ln \) be the number of terminal nodes in \( ai \) before that part which satisfies \("[\text{stream-item-list}]\"\) in this parse. Perform input-stream-items\(f\) times. Let \( csv \) be a \(<\text{character-string-value}>\) whose terminal nodes are those preceding the part of \( ai \) which satisfies \("[\text{stream-item-list}]\"\) in this parse, excluding the part of \( ai \) satisfying \("[\text{leading-delimiter-list}]\"\) and those terminals that are \(<\text{linebreak}>\). Return \( csv \).

8.7.1.2 Parse-data-input-value

Operation: \text{parse-data-input-value}\(fv\)

where \( fv \) is a \(<\text{file-value}>\).

result: a \(<\text{character-string-value}>\).

Step 1.

Case 1.1. \( fv \) is a \(<\text{file-value}>\).

Let \( fl \) be the \(<\text{file-information}>\) designated by \( fv \), and let \( ds \) be the \(<\text{dataset}>\) designated by the \(<\text{Dataset-Designator}>\) in \( fl \). Let \( cp \) be the \(<\text{current-position}>\) in \( fl \).

Case 1.1.1. There is no \(<\text{stream-item-list}>\) in \( ds \), or \( cp \) designates the last \(<\text{stream-item}>\) in the \(<\text{stream-item-list}>\) in \( ds \).

Perform raise-io-condition\(<\text{endfile-condition}>\); \(fv\).

Case 1.1.2. (Otherwise),

Let \( ai \) be a \(<\text{stream-item-list}>\) containing, in the same order, \(<\text{stream-item}>\) equal to the \(<\text{stream-item}>\) in the \(<\text{stream-item-list}>\) in \( ds \) which follow the \(<\text{stream-item}>\) designated by \( cp \).

Case 1.2. \( fv \) is \(<\text{absent}>\).

Let \( ai \) be the \(<\text{character-string-value}>\) in the current \(<\text{string-io-control}>\).
If \( ai \) contains no \(<\text{symbol}>\) then perform raise-condition\(<\text{error-condition}>\).
Step 2.

Case 2.1. sl can be parsed as "({leading-delimiter-list}) (\{j\}) (\{stream-item-list\})".

Let \( n \) be the sequence in sl which satisfies "({leading-delimiter-list}) (\{j\})" in this parse, and let the number of its terminal nodes be \( n \).

Perform input-stream-item(fv) \( n \) times. Return a <character-string-values> containing the last symbol in \( n \).

Case 2.2. sl can be parsed as "({leading-delimiter-list})".

Let \( n \) be the number of terminal nodes in sl. Perform input-stream-item(fv) \( n \) times. Return <character-string-values> <null-character-string>.

Case 2.3. sl can be parsed as "({leading-delimiter-list}) <(string-or-picture-symbol-list)>".

Let \( n \) be the number of terminal nodes in sl. Perform input-stream-item(fv) \( n \) times. Perform raise-condition<error-condition>.

Case 2.4. sl can be parsed as "({leading-delimiter-list}) <(putative-data-constant)>

Let \( n \) be the number of terminal nodes in sl preceding that part which satisfies "({stream-item-list})" in this parse. Perform input-stream-item(fv) \( n \) times. Let \( \text{cur} \) be a <character-string-values> whose terminal nodes are those of the part of sl which satisfies "(<putative-data-constant>)" except those terminals that are <linebreak>. Delete from \( \text{cur} \) the last symbol, if that symbol is a \# or a \?, \( \text{cur} \) is <character-string-values>.

8.7.1.5.3 Parsing Categories for data-directed input

Some of the categories used in parsing input streams for data-directed input are categories of the Concrete Syntax or Machine state syntax. Some of the categories are defined in Section 8.7.1.4.2 "Parsing categories for list-directed input". The remainder are defined as follows:

\[
\begin{align*}
\text{(putative-name-field)} &::= \text{(data-symbol) | '!' \{field-element-1-list\}} \\
\text{(field-element-1)} &::= \text{(data-symbol)}'!', ']'<\text{linebreak}> \\
\text{(putative-data-constant)} &::= \text{(simple-character-string-constant)} \\
&\quad \{\text{(data-symbol)} | '<\text{linebreak}> | '=' | \{\text{field-element-2-list}\} \} \\
&\quad \{\text{(data-symbol)} | '=' | \{\text{field-element-2-list}\} \} \\
\text{(field-element-2)} &::= \text{(data-symbol)}'<\text{linebreak}>|'='
\end{align*}
\]

8.7.1.6 Get-edit

Operation: \text{get-edit}(edi, fv)

where edi is an <edit-directed-input>,
fv is a <file-value>.

Step 1. Attach to the current <block-control> a

\[
\begin{align*}
\text{(data-item-control-list)}: \\
\text{(data-item-control)}: \\
\text{(data-list-indicator)}: \text{designating the first <input-target-list> of edi} \\
\text{<data-item-indicator>}: \text{undefined}.
\end{align*}
\]

Chapter 8: Input/Output 257
Step 2. Attach to the current <block-control> a

<format-control-list>
  <format-control>
    <format-specification-list-designator> designating the first <format-specification-list> of edi
    <format-list-index>
    <integer-value>
  ;

Step 3. Perform establish-next-data-item to obtain a <current-scalar-item>, adi;
<evaluated-target>, et; or <none>, adi. If adi is <none> then go to Step 6.

Step 4. Perform establish-next-format-item to obtain a <format-item>, sli.

Step 5.

Case 5.1. sli contains a <control-format>, ccf.
  Perform execute-input-control-format(ccf, fiv). Go to Step 4.

Case 5.2. sli contains a <data-format>, df.
  Perform execute-input-data-format(df, fiv). Go to Step 3.

Step 6. Let dp be the first <data-item-control> of the current <data-item-control-list>.

Case 6.1. The <data-list-indicator> of dp designates the last <input-target-list> of edi.
  Terminate this operation.

Case 6.2. (Otherwise).
  Replace dp by
    <data-item-control>
    <data-list-indicator> designating the next <input-target-list> of edi
    <data-item-indicator>
    <undefined>.

  Replace the current <format-control-list> by
    <format-control-list>
    <format-control>
    <format-specification-list-designator> designating the next
    <format-specification-list-designator> of edi
    <format-list-index>
    <integer-value>
  ;

  go to Step 3.

8.7.1.6.1 Execute-input-control-format

Operation: execute-input-control-format(ccf, fiv)

where ccf is a <control-format>,
            fiv is a [<file-values>].

Case 1. fiv is a [<file-values>].

Step 1.1. Let fi be the <file-information> designated by fiv. ccf must not have a
        <block>, <line-format>, or <col-format>. Let di be the <dataset-designator> designated
        by the <dataset-designator> in fi.
Step 1.2.

Case 1.2.1. ecf has a \(<space-format>\); \(<integer-value>\), w.

w must not be negative. If w=0 then terminate this operation; otherwise perform input-stream-item-for-edit(fv, l), for l=1,...,w.

Case 1.2.2. ecf has a \(<skip-format>\); \(<integer-value>\), w.

w must be greater than zero. Perform skip(w, fv).

Case 1.2.3. ecf has a \(<column-format>\); \(<integer-value>\), w.

Step 1.2.3.1. w must not be negative. Perform evaluate-current-column(w) to obtain cc. If w = 0 then let be 1. Let n be the number of symbols in \(d_{\text{i}}\) which follow the \(<\text{stream-item}>\) designated by the \(<\text{current-position}>\) in \(f_{\text{i}}\), up to the next \(<\text{stream-item}>\) which has a \(<\text{line-mark}>\) or \(<\text{space}>\).

Step 1.2.3.2.

Case 1.2.3.2.1. cc < (w-1) and n \(\leq\) cc-1.

Perform input-stream-item(fv) cc-1 times.

Case 1.2.3.2.2. cc = w-1.

Terminate this operation.

Case 1.2.3.2.3. cc > (w-1), or cc < (w-1) and n < (w-1).

Perform skip(integer-value, cc, fv). Let n be the number of symbols in \(d_{\text{v}}\) which follow the \(<\text{stream-item}>\) designated by the \(<\text{current-position}>\) in \(f_{\text{v}}\), up to the next \(<\text{stream-item}>\) which has a \(<\text{line-mark}>\) or \(<\text{space}>\). If n < (w-1) then perform input-stream-item(fv), w-1 times.

Case 2. fv is \(<absent>\).

ecf must have a \(<space-format>\); \(<integer-value>\), w. w must not be negative. If w=0 then terminate this operation; otherwise perform input-stream-item-for-edit(fv, l), for l=1,...,w.

E.7.3.4.2 Execute-input-data-format

Operation: execute-input-data-format(ec, edf, fv)

where ec is an \(<\text{evaluated-target}>\), edf is a \(<\text{data-format}>\),
fv is a \(<\text{file-value}>\).

Step 1.

Case 1.1. edf immediately contains a \(<\text{real-format}>\) or a \(<\text{string-format}>\).

Let w be the first \(<\text{integer-value}>\) in edf; w must be present.

Case 1.2. edf immediately contains a \(<\text{picture-format}>\), pf.

Let w be the associated character-string length of pf. (See Section 9.5.2.)

Case 1.3. edf immediately contains a \(<\text{complex-format}>\), ecf.

Step 1.3.1. If the first immediate component of ecf is a \(<\text{real-format}>\) then let w be the first \(<\text{integer-value}>\) in ecf; otherwise let w be the associated character-string length of the first component of ecf.
Step 1.3.2. If there is a second immediate component, sc, of scf and sc is a <real-format> then let w2 be the first <integer-value> in sc. If there is an sc and sc is a <picture-format> then let w2 be the associated character-string-length of sc. If there is no second immediate component of scf then let w2 be w1. Both w1 and w2 must be non-negative.

Step 1.3.3. Let w = w1+w2.

Step 2. w must not be negative. If w=0 then let n be a <basic-value>: <character-string-value>: <null-character-string>: and go to Step 6. Otherwise perform input-stream-item-for-edit(FV1), for i=1,...,w, to obtain w <stream-items> n[i] each containing a $Symbol$. If FV1 is a <file-value> then attach to the current <block-control> a <current-file-value> FV. Let cv1 be a <character-string-value> containing, in order, the w <character-values> n[i]. i=1,...,w, where cv1[i] contains the $Symbol$ n[i] for i=1,...,w.

Step 3. If ecf has a <complex-format>, ecf, then perform steps 3.1 through 3.7.

Step 3.1. If w=0 then let cv1 be a <character-string-value>: <null-character-string>. Otherwise let cv1 be a <character-string-value> obtained by deleting the last w <character-values> from a copy of cv1. If w2=0 then let cv2 be a <character-string-value>: <null-character-string>. Otherwise let cv2 be a <character-string-value> obtained by deleting the first w <character-values> from a copy of cv2.

Step 3.2. Let c1 be the first component of ecf; if c1 is a <real-format> then let d1 be its immediate component; otherwise let d1 be c1.

Step 3.3. If ecf has no second component then let d1 be d1. Otherwise let c2 be the second component of ecf; if c2 is a <real-format> then let d2 be its immediate component; otherwise let d2 be c2.

Step 3.4. Perform validate-input-format($d1,cv1) to obtain a <value-and-type>, r1 or <invalid>, r1. If r1 is <invalid> then let intg be its immediate component and perform raise-to-condition(conversion-condition), FV, cv1, intg; on normal return let cv1 be the immediate component of the current <returned-source-value> and go to Step 3.4.

Step 3.5. Perform validate-input-format($d2,cv2) to obtain a <value-and-type>, r2 or <invalid>, r2. If r2 is <invalid> then let intg be its immediate component and perform raise-to-condition(conversion-condition), FV, cv2, intg; on normal return let cv2 be the immediate component of the current <returned-source-value> and go to Step 3.5.

Step 3.6. Let v1 and v2 be the first components of r1 and r2, respectively. Let d1 and d2 be the second components of r1 and r2, respectively. Let adt be a <data-type> containing <real> and <decimal>, and <scale> and <precision> defined as follows:

Case 3.6.1. d1 has <fixed>, and <precision>; r = 0; and d2 has <fixed>, and <precision>; t = 0.

adt has <scale>; <fixed>; <precision>; p = q; where:

\[ p = \min(\max(r-3, 0), \max(a, u)) \]

q = max(a, u)

N = maximum <number-of-digits> for <fixed> and <decimal>.

Case 3.6.2. (Otherwise).

Let r and t be the <number-of-digits> of d1 and d2, respectively. adt has <float>; <number-of-digits>; N = maximum <number-of-digits> for <float> and <decimal>.

Step 3.7. Perform convert(adt, d1, v1) to obtain a <real-value> containing the <real-number>, cv1, and perform convert(adt, d2, v2) to obtain a <real-value> containing the <real-number>, cv2. Let be be a <basic-value> containing a <complex-value>: cv1 cv2. Let add be an <evaluated-data-description> containing an <item-data-description> containing a <data-type> with cmode: <complex> but otherwise as adt. Go to Step 7.
Step 6. If df immediately contains a <real-format>, f, or a <string-format>, f, then perform Steps 4.1 through 4.3.

Step 4.1. Let df be the immediate component of f.

Step 4.2. Perform validate-input-format(df, csv) to obtain a <value-and-type>, v, or a <character-string-value>, v, or <invalid>, v. If v is <invalid> then let intg by its immediate component and perform raise-to-condition (<conversion-condition>, f, csv, intg); on normal return let csv be the immediate component of the current <returned-quantity-value> and go to Step 4.2.

Step 4.3. Go to Step 6.

Step 5. If df immediately contains a <picture-format>, pf, then perform Step 5.1.

Step 5.1. Perform validate-input-format(pf, csv) to obtain a <value-and-type>, v, or a <character-string-value>, v, or <invalid>, v. If v is <invalid> then let intg be its immediate component and perform raise-to-condition (<conversion-condition>, f, csv, intg); on normal return let csv be the immediate component of the current <returned-quantity-value> and go to Step 5.1.

Step 6. If v is a <character-string-value> or a <bit-string-value>.

Case 6.1. v is a <character-string-value>.

Let odd be an <evaluated-data-description> containing an <item-data-description> or <character> or <bit> (respectively) and <maximum-length>: *atend*). Let bv be a <basic-value>: v.

Case 6.2. v is a <value-and-type>: val adt.

Let odd be an <evaluated-data-description> containing an <item-data-description> containing adt. Let bv be a <basic-value>: val.

Step 7. Let aay be an

<aggregate-value>:
<aggregate-type>:
<scalar>:
<basic-value-list>:
 bv.

Let ddf be the <data-description> immediate component of odd. Perform validate(input, aay, ddf). If fv is a <file-value> then delete the current <current-file-value>.

8.7.1.4.2.1 Validate-input-format

<invalid>: <integer-value>

Operation: validate-input-format(df, csv)

where df is an immediate component of a <real-format> or a <string-format>, or is a <picture-format>,

csv is a <character-string-value>.

result: <invalid> or <character-string-value> or <bit-string-value> or <value-and-type>.

Case 1. df is a <picture-format>.

Case 1.1. df contains <picture-numeric>.

Perform validate-numeric-pictured-value(df, csv) to obtain a <picture-value>: pv. If pv has <picture-invalid>: val; then return <invalid>: val. If pv has <picture-valid>: val; then return a <value-and-type>: val adt; where adt is the associated arithmetic data-type of df.
Case 1.2. `df` contains `<picture-character>.

Perform `validate-character-pictured-value(df, cv)` to obtain a `<picture-validity>, pv`. If `pv` has `<picture-invalid>` val; then return `<invalid>` val. Otherwise return `cv`.

Case 2. `df` is a `<character-format>`.

Return `cv`.

Case 3. `df` is a `<bit-format>`.

Case 3.1. `cv` conforms to the syntax `"[E-list] [e-list] [M-list]"` where `e` is one of the `[symbols] in the first column of Table 4.2 corresponding to the `<radix-factor>` in `df`.

Let `bc` be that part of `cv` which satisfies `"[e-list]"`. Let `bc` be the `<character-string-value>` obtained by concatenating `pc` with a `" \" on the left, and, on the right, with a `\'81`, `\'82`, `\'83` or `\'84`, according to the `<radix-factor>` of `df`. Perform `basic-bit-value(bv)` to obtain a `<bit-string-value>bv. Return `bv`.

Case 3.2. (Otherwise).

Let `intg` be the smallest value such that the first `intg` `[symbols] in `cv` do not have a valid continuation according to this syntax. Return `<invalid>`, `intg`.

Case 4. `df` is a `<fixed-point-format>`.

Step 4.1. Let `w`, `d`, and `s` be respectively the `<integer-values>` in `df`. If either `d` or `s` (or both) is `<absent>` then 0 is assumed. `d` must not be negative.

Step 4.2.

Case 4.2.1. `cv` conforms to the syntax `"[E-list]"` or is a `<character-string-value>`:

- `<null-character-string>`.

Return `<value-and-type>`: `<real-value>`: 0; `adr`; where `adr` is a `<data-type>` containing `<real>`, `<fixed>`, `<decimal>`, `<number-of-digits>`: 1; and `<scale-factor>`: 0.

Case 4.2.2. `cv` conforms to the syntax `"[E-list]\ [s]\ [decimal-number]\ [M-list]"`.

Perform `basic-numeric-values(v)` to obtain a `<value-and-type>`: `v dt`. If `cv` contains a `s` then let `w` = `s`; otherwise, let `w` = `d`. Return a `<value-and-type>`: `<real-values>`: `(v\*10^w)”; `adr`; where `adr` is a `<data-type>` equal to `dt` except that `<scale-factor>` is decremented by `s`.

Case 4.2.3. (Otherwise).

Let `intg` be the smallest value such that the first `intg` `[symbols] in `cv` do not have a valid continuation according to the syntax in Case 4.2.2. Return `<invalid>`: `intg`.

Case 5. `df` is a `<floating-point-format>`.

Step 5.1. Let `w`, `d`, and `s` be respectively the `<integer-values>` in `df`. If `s` is present, it is ignored. If `d` is `<absent>` then let `d` be 0. `d` must not be negative.

Step 5.2.

Case 5.2.1. `cv` conforms to the syntax `"[E-list]"` or is a `<character-string-value>`:

- `<null-character-string>`.

Return `<value-and-type>`: `<real-value>`: 0; `adr`; where `adr` is a `<data-type>` containing `<real>`, `<fixed>`, `<decimal>`, `<number-of-digits>`: 1; and `<scale-factor>`: 0.
Case 5.2.2. csv conforms to the syntax "\{f-\list\} \ [+\ [-] \ \{decimal-number\} \ [[f][i][l][e][\+][\-][\+][\+][\-]3] \ \{integer\} \ \{f-\list\} ",

If csv contains a sign, not set to \$. after a substring conforming to \{decimal-number\}, then replace that substring with the concatenation of that substring with \$. Perform \{scale-factor\} to obtain a \{value-and-type\}: \v, \dt. If csv contains a \{\} then let \d'=\dt; otherwise let \d'=\dt. Return a \{value-and-type\}: \{real-value\}: \{f-\list\}: \{\} \ dt, where \adc \ is a \{data-type\} equal to dt except that its \{scale-factor\} (if any) is incremented by \d'.

Case 5.2.3. (Otherwise).

Let \INT be the smallest value such that the first \{symbol\} in csv do not have a valid continuation according to the syntax in Case 5.2.2.
Return \{invalid\}: Intg.

§7.1.7 Input-stream-item

Operation: \text{input-stream-item}(\text{fv})
where \text{fv} is a \{file-value\}.
result: a \{stream-item\} or \{omega\}.

Step 1.
Case 1.1. \text{fv} is a \{file-value\}.

Step 1.1.1. Let \text{fi} be the \{file-information\} designated by \text{fv}; let \text{cp} be the \{current-position\} in \text{fi}, and let \text{ds} be the \{dataset\} designated by the \{dataset-designator\} in \text{fi}.

Step 1.1.2.
Case 1.1.2.1. \text{cp} designates \{alpha\}.
Replace the immediate component of \text{cp} by a \{designator\} to the first \{stream-item\} in \text{ds}, or to the \{omega\} in \text{ds} if there are no \{stream-item\}s in \text{ds}.

Case 1.1.2.2. \text{cp} designates a \{stream-item\}.
Replace the immediate component of \text{cp} by a \{designator\} to the next \{stream-item\} in \text{ds}, or to the \{omega\} in \text{ds} if there are no further \{stream-item\}s in \text{ds}.

Case 1.1.2.3. \text{cp} designates \{omega\}.
go to Step 1.1.3.

Step 1.1.3. Let \text{si} be the node designated by \text{cp}.
Case 1.2. \text{fv} is \{absent\}.

Let \text{sc} be the current \{string-to-control\}. Let csv be the \{character-string-value\} in misc. If csv contains a \{null-character-string\} then perform \{raise-condition\} \{error-condition\}. Otherwise let csv be a \{stream-items\}: \{\}, where \v is the first \{symbol\} in csv. If csv contains only one \{symbol\} then replace csv by \{character-string-value\} \{null-character-string\}. Otherwise delete the first \{character-value\} in csv.

Step 2. If there is a \{copy-file\} in the current \{block-state\} then let \text{cf} be its immediate component; otherwise go to Step 3. If csv is not \{omega\} or \{line-mark\} then perform \{output-string-item\}: \{symbol\}: \{cf\}, where \acs is a \{stream-items\} containing the \{symbol\} in \text{si}. If csv is \{line-mark\} then perform \{skip\}: \{integer-value\}: \{cf\}.

Step 3. Return \text{si}.
8.7.1.8 Basic-character-value

Operation: \[\text{basic-character-value}(\text{cvin})\]

where cvin is a <character-string-value>.

result: a <character-string-value>.

Step 1. The symbols of cvin must conform to the syntax of <simple-character-string-constant>.

Step 2. Let cvo be a copy of cvin. Remove the first and last <character-values> of cvo. If no <character-values> remain then return <character-string-value>: <null-character-string>. Replace all adjacent pairs of <character-values>: \{symbol: `\"', \"`\}" originally in cvo with a single <character-values>: \{symbol: `\"`\}".

Step 3. Return cvo.

8.7.1.9 Basic-bit-value

Operation: \[\text{basic-bit-value}(\text{cvin})\]

where cvin is a <character-string-value>.

result: a <bit-string-value>.

Step 1. cvin must conform to the syntax of <simple-bit-string-constant>.

Step 2. Let m be 1, 2, 3, or 4 according as the \{radix-factor\} in cvin is B, B1, B2, B3, or B4. Remove the portion of cvin corresponding to the syntax for \{radix-factor\} and remove the first and last of the remaining <character-values>. Let n be the number of <character-values> remaining, if any. If m=0 then return <bit-string-value>: <null-bit-string>.

Step 3. Let \(sy{i}\) for \(i=1, \ldots, n\), be the \{symbol\} in order, of the remaining <character-values> in cvin. Each \(sy{i}\) must have an entry in Table 4.2 which is valid for the value of m. Let bvo be a <bit-string-values>, containing \(n\) <bit-values>, such that <bit-values> \(i=1, \ldots, n\) are obtained from Table 4.2 as a function of \(m\) and \(sy{i}\) for \(i=1, \ldots, n\).

Step 4. Return bvo.

8.7.1.10 Input-stream-item-for-edit

Operation: \[\text{input-stream-item-for-edit}(\text{fv}, i)\]

where fv is a <file-values>, i is an <integer-values>.

result: a <stream-item>.

Step 1. Perform input-stream-item(fv) to obtain a <stream-item>, si or an <empty>, si.

Step 2.

Case 2.1. si is <empty>.

If \(i=1\) then perform raise-error-condition(<file-condition>, fv); otherwise perform raise-condition(<error-condition>).

Case 2.2. (Otherwise).

If si contains <linemark> then go to Step 1. Otherwise return si.
6.7.2 The Put Statement

6.7.2.1 Execute-put-statement

Operation: execute-put-statement(ps)

where ps is a <put-statement>.

Step 1.

Case 1.1. ps has a <put-file>, pf:

Perform execute-put-file(pf).

Case 1.2. ps has a <put-string>, str:

Perform execute-put-string(str).

Step 2. Perform trim-to-control.

Step 3. Perform normal-sequence.

6.7.2.2 Execute-put-file

Operation: execute-put-file(pf)

where pf is a <put-file>.

Step 1. Perform Steps 1.1, 1.2, and 1.3 in any order.

Step 1.1. Let fo be the <value-reference> of the <file-option> in pf. Perform evaluate-file-option(fo) to obtain a <file-value>, fv.

Step 1.2. If pf has a <skip-option>, sko, then let e be the <expression> in sko, and perform evaluate-expression-to-integer(e), to obtain an <integer-value>, ik.

Step 1.3. If pf has a <line-option>, lopt, then let e be the <expression> in lopt, and perform evaluate-expression-to-integer(e), to obtain an <integer-value>, ik.

Step 2. Let fi be the <file-information> designated by fv. If fi contains <open> then go to Step 4; otherwise go to Step 3.

Step 3. Let eff be an <evaluated-file-description-list> containing <stream> and <output>. Perform open(fi, eff) to obtain a result sf. If sf is <fail> then perform raise-condition(not-defined-file-condition), fv. If on normal return fi contains <closed> then perform raise-condition(error-condition).

Step 4. fi must contain <stream> and <output>.

Step 5. If pf immediately contains <page> then perform put-page(fv).

Step 6. If pf has a <line-option> then perform put-line(i, fv).

Step 7. If pf has a <skip-option> then perform skip(fv).

Step 8. If pf has an <output-specification> then:

Case 8.1. pf has a <list-directed-output>, lod:

Perform put-list(lod, fv).

Case 8.2. pf has a <data-directed-output>, dod:

Perform put-data(dod, fv).

Case 8.3. pf has a <edit-directed-output>, edo:

Perform put-edit(edo, fv).

Chapter 8: Input/Output 265
8.7.2.3 Execute-get-string

Operation: \texttt{execute-get-string}(ptr)

where \texttt{ptr} is a \texttt{<get-string>},

Step 1. Let \texttt{tr} be the \texttt{<target-reference>} in \texttt{ptr}. Perform \texttt{evaluate-target-reference}(tr) to obtain an \texttt{<evaluated-target>, et}.

Case 1.1. \texttt{et} has a \texttt{<generation>},

Let \texttt{n} be the \texttt{<maximum-length> contained in et}.

Case 1.2. \texttt{et} has an \texttt{<evaluated-pseudo-variable-reference>},

Case 1.2.1. \texttt{et} contains \texttt{<onchar-pr>},

There must be a current \texttt{<onchar-value>}. Let \texttt{m=1},

Case 1.2.2. \texttt{et} contains \texttt{<onsource-pr>},

There must be a current \texttt{<onsource-value>}. Let \texttt{n be the length of the current \texttt{<onsource-value>},}

Case 1.2.3. \texttt{et} contains \texttt{<ondata-pr>},

If \texttt{et} contains four elements, let \texttt{n} be the fourth of its elements. Otherwise let \texttt{n be log-st-1}, where \texttt{log is the length of the value of the \texttt{<generation> contained in et and st is the third element of et}}.

Step 2. Attach to the current \texttt{<block-control>} a

\texttt{<string-io-control>},

\texttt{<character-string-value>},

\texttt{<null-character-string>},

\texttt{<string-list>},

\texttt{<integer-value>},

\texttt{n}.

Step 3.

Case 3.1. \texttt{ptr} has a \texttt{<list-directed-output>}, \texttt{ldo}.

Perform \texttt{put-list(ldo)}.

Case 3.2. \texttt{ptr} has a \texttt{<data-directed-output>}, \texttt{ddo}.

Perform \texttt{put-data(ddo)}.

Case 3.3. \texttt{ptr} has a \texttt{<edit-directed-output>}, \texttt{edo}.

Perform \texttt{put-edit(edo)}.

Step 4. Let \texttt{k} be the \texttt{<character-string-value>} of the current \texttt{<string-io-control>}. Let \texttt{n} be the number of \texttt{<symbol>}s in \texttt{k}. Perform \texttt{assign(temp, Aviv, dd)} where \texttt{Aviv} is an \texttt{<aggregate-value> containing \texttt{on}}, and \texttt{dd} is a \texttt{<data-description> containing \texttt{<character>}, <converting>}, and \texttt{<maximum-length> with n}}.

Step 5. Delete the \texttt{<string-io-control>} from the current \texttt{<block-control>}. 
8.7.2.1 Put-list

Operation: \texttt{put-list}(ldo, fv)

where ldo is a \texttt{list-directed-output},
fv is a \texttt{file-value}.

Step 1. Attach to the current \texttt{block-control} a

\texttt{data-item-control-list}:
\texttt{data-item-control}:
\texttt{data-list-indicator} designating the \texttt{output-source-list} of ldo
\texttt{data-item-indicator}:
\texttt{undefined}.

Step 2. If fv is a \texttt{file-value} then let fi be the \texttt{file-information} designated by fv
Step 3. Let t be an \texttt{integer-value}.

Case 3.1. fv is a \texttt{file-value} and fi contains \texttt{print}.

Step 3.1.1. Perform \texttt{evaluate-current-column}(fv) to obtain cc.

Step 3.1.2. If the \texttt{evaluated-tab-option} of fi contains an \texttt{integer-value}, iv, such
that iv-1, then let t = iv; otherwise let t = 0.

Case 3.2. (Otherwise).

Let t = 0.

Step 4. Perform \texttt{establish-next-data-item} to obtain a \texttt{current-scalar-item}, ni, of
\texttt{none}. If ni is \texttt{none} then perform \texttt{output-string-item}(\texttt{stream-item}:\texttt{symbol}:a:::t, fv) and terminate this operation.

Step 5. Let \texttt{dt} be the \texttt{data-type} in \texttt{di}, and let \texttt{dt} be a \texttt{data-type} simple
containing \texttt{character}, \texttt{numbering}, and \texttt{maximum-length}\texttt{c}, \texttt{asterisk}. Perform
\texttt{convert}(\texttt{id}, \texttt{dt}, \texttt{tv}), where \texttt{tv} is the \texttt{basic-value} in \texttt{ni1}, to obtain
\texttt{character-string-value}, cv.

Step 6.

Case 6.1. \texttt{dt} contains \texttt{bit}.

Let cv be a \texttt{stream-item-list} containing, in order, the \texttt{symbol}s

\texttt{'}\texttt{'}
the \texttt{symbol}s of cv, if any,
\texttt{'}\texttt{'}
\texttt{'}

Case 6.2. \texttt{dt} contains \texttt{character} or \texttt{picted-character}, and either \texttt{fv} is \texttt{absent},
or \texttt{fv} is a \texttt{file-value} and \texttt{fi} does not contain \texttt{print}.

Let cv be a \texttt{stream-item-list} containing, in order, the \texttt{symbol}s

\texttt{'}\texttt{'}
the \texttt{symbol}s of cv, if any, not with each \{symbol\}: \{"\}; replaced
by two occurrences of \{symbol\}: \{"\}
\texttt{'}\texttt{'}

Case 6.3. (Otherwise).

Let cv be a \texttt{stream-item-list} containing, in order, the \texttt{symbol}s of cv.

Step 7.

Case 7.1. \texttt{fv} is a \texttt{file-value} and \texttt{fi} contains \texttt{print}.

Perform \texttt{table}(t, fv).
Case 7.2. (otherwise).

If \( t = 1 \) then perform output-string\( <\text{stream-item-list}> \) \( <\text{stream-item}> \) \{symbol\} \( \{\text{fi}\} \), \( \{\text{fv}\} \).

Step 8. Let \( t = 1 \).

Step 9.

Case 9.1. \( \text{fv} \) is a \{file-value\}.

Perform evaluate-current-column\( \{\text{fv}\} \) to obtain \( \text{co} \). Let \( \text{lo} \) be the \{evaluated-column\} in \( \text{fi} \). Let \( \text{lev} \) be the number of \{symbol\} in \( \text{cv} \). If \( \text{lev} > \{\text{lex-column}\} \) and \( \text{co} = 0 \) then perform skip\{integer-value\}; \( \{\text{fi}\} \), \( \{\text{fv}\} \).

Perform output-string\( \{\text{cv}\}, \{\text{fv}\} \).

Case 9.2. \( \text{fv} \) is \{absent\}.

Perform output-string\( \{\text{cv}\}, \{\text{fv}\} \).


8.7.2.5 \text{Put-data}

Operation: \text{put-data}\( \{\text{ddo}\}, \{\text{fv}\} \)

where \( \{\text{ddo}\} \) is a \{data-directed-object\},

\( \{\text{fv}\} \) is a \{file-value\}.

Step 1.

Case 1.1. \( \{\text{ddo}\} \) contains a \{data-source-list\}, \( \{\text{dal}\} \).

Go to Step 2.

Case 1.2. \( \{\text{ddo}\} \) does not contain a \{data-source-list\}.

Let \( \{\text{dal}\} \) be a \{data-source-list\} which contains an implementation-defined list of trees of the form

\[
\{\text{data-source}\}, t:
\{\text{variable-reference}\};
\{\text{declaration-designator}\}, \{\text{dd}\}
\{\text{identifier-list}\}, \{\text{data-description}\}, \{\text{ddesc}\};
\]

where \( \{\text{dd}\} \) designates a \{declaration\} which contains \{variable\} but does not contain \{based\} without a \{nodesc\} and \( \{\text{ddesc}\} \) does not contain a \{non-computational-type\}.

Step 2. Attach to the current \{block-control\} a

\[
\{\text{data-item-control-list}\};
\{\text{data-item-control}\};
\{\text{data-list-indicator}\} \text{designating} \{\text{ddi}\}
\{\text{data-item-indices}\}:
\{\text{undefined}\};
\]

Step 3. If \( \{\text{fv}\} \) is a \{file-value\} then let \( \{\text{fi}\} \) be the \{file-information\} designated by \( \{\text{fv}\} \).

Step 4. Let \( t \) be an \{integer-value\}.

Case 4.1. \( \{\text{fv}\} \) is a \{file-value\} and \( \{\text{fi}\} \) contains \{print\}.

Step 4.1.1. Perform evaluate-current-column\( \{\text{fv}\} \) to obtain \( \text{co} \).

Step 4.1.2. If the \{evaluated-tab-optional\} of \( \{\text{fi}\} \) contains an \{integer-value\}, \( \{\text{iv}\} \), such that \( \text{co} \neq \{\text{iv}\} \) then let \( t = 0 \); otherwise let \( t = 1 \).
Case 6.2. (Otherwise). 

Let cv be.

Step 5. Perform `establish-next-data-item` to obtain a `<current-scalar-item>, ndl or `<none>, ndl`. If ndl is `<none>` then perform `output-string-item(oc, fv)` where oc is a `<stream-item>; `symbol`; '{'}; and terminate this operation. Otherwise ndl is

```
<current-scalar-item>;
<basic-value>, bv
<data-type>, sd;
<data-name-field>, sf;
`symbol-list`, sl;
```

`sd` must contain `<computational-type>`.

Step 6. Let n be a `<stream-item-list>` containing, in order, the `symbol`s of the `<data-name-field>` in ndl.

Step 7. Let tdt be a `<data-type>` simply containing `<character>, <converting>,` and `<numeric-length>; `asterisk>`. Perform `convert(tdt, ndl, fv)` to obtain a `<character-string-values>, cvv`.

Step 8.

Case 8.1. ndl contains `<ch>`,

Let cv be a `<stream-item-list>` containing, in order, the `symbol`s

```
'c'
the `symbol`s of cvv, if any,
'
```

Case 8.2. ndl contains `<character>`, `<pictorial-character>`.

Let cv be a `<stream-item-list>` containing, in order, the `symbol`s

```
'c'
the `symbol`s of cvv, if any, not with each `symbol`; '{' replaced by two occurrences of `symbol`; '{'};
'
```

Case 8.3. (Otherwise).

Let cv be a `<stream-item-list>` containing, in order, the `symbol`s of cvv.

Step 9.

Case 9.1. fv is a `<file-value>` and fi contains `<print>`.

Perform `tabit, fv`.

Case 9.2. (Otherwise).

```
If t=1, then perform `output-string-itemstream-item-list>; `<stream-item>; `symbol`; `n1`, fv).
```

Step 10. Let t=1.

Step 11.

Case 11.1. fv is a `<file-value`.

Step 11.1.1. Perform `evaluate-current-column(fv)` to obtain `ocv`. Let len be the `evaluated-length` in fi. Let ln be the number of `symbol`s in cvv, and let lcn be the number of `symbol`s in ln.

Step 11.1.2. If (ln+1) > (ln-ocv) and ocv # 0 then perform `skip(integer-values); l2, fv)`.
Step 11.1.3. Perform `output-string(a, fv)`.
Step 11.1.4. Perform `output-string-item(<stream-item>; <symbol>; "f": ..., fv)`.
Step 11.1.5. Perform `evaluate-current-column(fv)` to obtain `cool`. If `low > (last-cool)` and `cool = 0` then perform `skip(<integer-value>; 1); fv`.

Case 11.1.2. fv is `Absent`.

Perform `output-string(in, fv)`; perform `output-string-item(<stream-item>; <symbol>; "f": ..., fv)`; perform `output-string(cm, fv)`.

Step 12. Go to Step 5.

8.1.2.6 Put-edit

Operation: `put-edit(edo, fv)`

where `edo` is an `<edit-directed-output>`, `fv` is a `<file-value>`.

Step 1. Attach to the current `<block-control>` a

- `<data-item-control-list>`;
- `<data-item-control>`;
  - `<data-list-indicator>` designating the first `<output-source-list>` of `edo`;
  - `<data-item-indicator>`;
  - `<undefined>`.

Step 2. Attach to the current `<block-control>` a

- `<format-control-list>`;
- `<format-control>`;
  - `<format-specification-list-designator>` designating the first `<format-specification-list>` of `edo`;
  - `<format-list-index>`;
  - `<integer-value>`;
  - `<undefined>`.

Step 3. Perform `establish-next-data-item` to obtain a `<current-scalar-item>`, `ndi`, or `<none>`, `ndi`. If `ndi` is `<none>` then go to Step 4.

Step 4. Perform `establish-next-format-item` to obtain a `<format-item>`, `efi`.

Step 5.

Case 5.1. `efi` contains a `<control-format>`, `efc`.

Perform `execute-output-control-format(efc, fv)`. Go to Step 6.

Case 5.2. `efi` contains a `<data-format>`, `efd`.

Perform `execute-output-data-format(ndi, edf, fv)`. Go to Step 1.

Step 6. Let `dpp` be the first `<data-item-control>` of the current `<data-item-control-list>`.

Case 6.1. The `<data-list-indicator>` of `dpp` designates the last `<output-source-list>` of `edo`.

Terminate this operation.
Case 6.2. (Otherwise).
Replace dpp by

<data-item-control>
  <data-list-indicator> designating the next compute-source-list> of
  edo
  <data-item-indicator>
  <undefined>
</data-item-control>

Replace the current <format-control-list> by

<format-control-list>
  <format-control>
    <format-specification-list-designator> designating the next
    <format-specification-list> of edo;
    <format-list-indexes>
      <integer-value>
      0.
  </format-control>
</format-control-list>

Go to Step 3.

8.7.2.6.1 Execute-output-control-format

Operation: execute-output-control-format(ecf,fv)
where ecf is a <control-format>,
fv is a <file-value>.
Step 1. If fv is a <file-value> then let fi be the <file-information> designated by fv.
If fv is <defined> then ecf must have a <space-format>.
Step 2.
Case 2.1. ecf has a <space-format>: <integer-value>,w.
   w must not be negative. If w=0 then terminate this operation. Perform
   output-stringlist[fw] where str is a <stream-item-list> whose w terminal
   nodes are all &q.
Case 2.2. ecf has a <skip-format>: <integer-value>,w.
   w must not be negative. Perform skip(w,fv).
Case 2.3. ecf has a <line-format>: <integer-value>,w.
   Perform put-line(fw,fv).
Case 2.4. ecf has <page>.
   Perform put-page(fv).
Case 2.5. ecf has a <tab-format>: <integer-value>,w.
   w must not be negative. Perform tab(w,fv).
Case 2.6. ecf has a <column-format>: <integer-value>,w.
   Step 2.6.1. Perform evaluate-current-column(fw) to obtain cc. w must not be
   negative. If w=0 or w exceeds the <evaluated-linesize> in fi then let
   w be 1.
   Step 2.6.2.
   Case 2.6.2.1. cc < w-1.
   Perform output-stringlist[fw], where str is a <stream-item-list>
   containing (w-cc-1) terminal nodes, each of which is a &q.
Case 2.6.2.3. \( n = w-1 \).

Terminate this operation.

Case 2.6.2.3. \( cc > w-1 \).

Perform \( \text{skip}(\text{integer-value}, 1, \text{fv}) \). If \( w > 1 \) then perform \( \text{output-string}(\text{str}, \text{fv}) \), where \( \text{str} \) is a \( \text{stream-item-list} \) containing \( (w-1) \) terminal nodes, each of which is a \( \text{x} \).

8.7.2.4.2 Execute-output-data-format

Operation: \( \text{execute-output-data-format}(\text{edi}, \text{edf}, \text{fv}) \)

where \( \text{edi} \) is a \( \text{current-scalar-item} \),
\( \text{edf} \) is a \( \text{data-format} \),
\( \text{fv} \) is a \( \text{file-value} \).

Case 1. \( \text{edf} \) contains a \( \text{bit-format} \), \( \text{edt} \).

Let \( n \) be the value of the \( \text{radix-factor} \) in \( \text{edt} \). Perform \( \text{convert}(\text{dt}, \text{edf}, \text{dv}) \) to obtain a \( \text{bit-string-values}, \text{bsv} \) where \( \text{dt} \) is a \( \text{data-type} \) containing \( \text{bit} \), \( \text{negerating} \), and \( \text{maximum-length} \): \( \text{asterisk} \), \( \text{dt} \) is the \( \text{data-type} \) of \( \text{edi} \), and \( \text{dv} \) is the \( \text{basic-value} \) or immediate component of \( \text{edi} \). Let \( n \) be the number of immediate components of \( \text{bsv} \). Let \( m = n \cdot n \) if \( n \) is a multiple of \( m \), then let \( m \) be the next higher multiple of \( n \) and insert \( \text{n} \) \( \text{bit-values} \), \( \text{zero-bit} \) at the end of the \( \text{bit-value-list} \) of \( \text{bsv} \). Let \( m \) be the number of \( \text{bit-values} \) in \( \text{bsv} \). If \( \text{edf} \) contains an \( \text{integer-values}, \text{w} \), and \( w > m \) then perform \( \text{value-condition}(\text{string-space-condition}) \) and let \( w = 0 \). If \( \text{edf} \) has no \( \text{integer-values} \) then let \( w = 1 \). If \( w = 0 \) then terminate this operation. Let \( \text{csv} \) be a \( \text{stream-item-list} \) containing \( w \) \( \text{symbol} \) as follows:

The first \( k \) contain, in order, the \( \text{symbol} \) in Table 4.2 corresponding to successive groups of \( m \) \( \text{bit-values} \) in \( \text{bsv} \).
The remaining \( w-k \) contain \( \text{he} \).

Perform \( \text{output-string}(\text{csv}, \text{fv}) \).

Case 2. \( \text{edf} \) contains a \( \text{character-format} \), \( \text{ect} \).

Let \( \text{dt} \) be a \( \text{data-type} \) containing \( \text{character} \) and \( \text{maximum-length} \); the \( \text{integer-values} \) in \( \text{ect} \) or \( \text{asterisk} \) if there is no \( \text{integer-values} \) in \( \text{ect} \); let \( \text{dt} \) be the \( \text{data-type} \) and \( v \) the \( \text{basic-values} \) of \( \text{edi} \). Perform \( \text{convert}(\text{dt}, \text{edf}, \text{dv}) \) to obtain a \( \text{character-string-values}, \text{csv} \). Let \( \text{csv} \) be a \( \text{stream-item-list} \) containing, in order, the \( \text{symbol} \) of \( \text{csv} \). Perform \( \text{output-string}(\text{csv}, \text{fv}) \).

Case 3. \( \text{edf} \) immediately contains a \( \text{picture-format} \) with a \( \text{picted} \) node, \( \text{pic} \).

Let \( \text{dt} \) be the \( \text{data-type} \) and \( \text{dv} \) the \( \text{basic-values} \) of \( \text{edi} \). Perform \( \text{convert}(\text{dt}, \text{edf}, \text{dv}) \), where \( \text{dt} \) is a \( \text{data-type} \) containing \( \text{pic} \). Perform \( \text{convert}(\text{dt}, \text{edf}, \text{dv}) \), where \( \text{dt} \) is a \( \text{data-type} \) containing \( \text{pic} \). Let \( \text{csv} \) be a \( \text{stream-item-list} \) containing, in order, the \( \text{symbol} \) of \( \text{csv} \). Perform \( \text{output-string}(\text{csv}, \text{fv}) \).

Case 4. \( \text{edf} \) contains a \( \text{complex-format} \), \( \text{ect} \).

Step 4.1. Let \( \text{dt} \) be the \( \text{data-type} \) of \( \text{edi} \). Let \( m \) be the \( \text{basic-values} \) of \( \text{edi} \).

Case 4.1.1. \( \text{dt} \) contains \( \text{character} \), \( \text{picted-character} \), or \( \text{bit} \).

Perform \( \text{convert-to-arithmetic}(\text{dt}, \text{dv}) \) to obtain a \( \text{value-and-type} \) either with components \( \text{dp} \) and \( \text{rdt} \), or with immediate components \( \text{rp} \), \( \text{dp} \), \( \text{ip} \), and \( \text{idt} \). If \( \text{ip} \) and \( \text{idt} \) do not exist, let \( \text{ip} \) be a \( \text{real-values} \), \( 0 \); and let \( \text{idt} \) be \( \text{rdt} \).
Case 4.1.2. edt contains <real> (including <picture-numeric>).

Let rp be sv. Let rd and idt be edt. Perform convert(idt,dt,v) to obtain ip, where v is a <real-value> 0; and dt is a <data-type> containing <real>, <fixed>, <decimal>, <number-of-digits>: 1; and <scale-factor>: 0.

Case 4.1.3. edt contains <complex> (including <picture-numeric>).

Let rd and idt be <data-type>s containing <real> but otherwise as edt. If edt does not contain <picture-numeric>, let rp and ip be <basic-values> which respectively contain the two <real-number>s in c11. If edt has <picture-numeric>, let rp and ip be <character-string-value>s containing, respectively, the first n and last a <character-value>s in c11, where n is the associated character-string length of rd and idt.

Step 4.2. Let c1 be the immediate component of the first immediate component of c11. If c11 has a second immediate component, so, then let c2 be the immediate component of c11; otherwise let c2 be c1.

Step 4.3. If c1 is <picture>, then perform convert(pl,rd,rp) to obtain a <character-string-value>,osv1, where pl is a <data-type> containing c1. Otherwise perform edit-numeric-output(ip,rd,cl) to obtain osv1.

Step 4.4. If c2 is <picture>, then perform convert(p2,rd,rp) to obtain a <character-string-value>,osv2, where p2 is a <data-type> containing c2. Otherwise perform edit-numeric-output(ip,rd,cl) to obtain osv2.

Step 4.5. Let s1 be a <stream-item-list> containing, in order, the $\text{symbol}_i$s of osv1 and osv2. If s1 contains one or more $\text{symbol}_i$s then perform output-stream(s1,sv).

Case 5. Otherwise.

Let sf be the <fixed-point-format> or <floating-point-format> in edt. Perform edit-numeric-output(bv,dt,fpf) to obtain a <character-string-value>,osv, where bv is the <basic-value> in c11 and dt is the <data-type> in c11. Let s1 be a <stream-item-list> containing, in order, the $\text{symbol}_i$s of osv. If s1 contains one or more $\text{symbol}_i$s then perform output-string(s1,sv).

8.7.2.6.3 Edit-numeric-output

Operation: edit-numeric-output(bv,dt,fpf)

where bv is a <basic-value>,
    dt is a <data-type>,
    fpf is a <fixed-point-format> or a <floating-point-format>.

result: a <character-string-value>.

Step 1.

Case 1.1. bv has a <real-value>,cv.

Let edt be edt.

Case 1.2. bv has a <complex-value>.

Let cv be a <real-value> containing the real part or bv; let cvt be a <data-type> containing <real> but otherwise as edt.

Case 1.3. bv has a <character-string-value> or a <bit-string-value>.

Perform convert-to-arithmetic(dt,v) to obtain a <value-and-type> with first two components cvt and dt, where v in the immediate component of bv. Let cvt be a <data-type> containing dt.

Step 2. Let w, d, and e be (respectively) the <integer-values> in fpf, if they exist.
Step 3.

Case 3.1. $fpf$ is a <fixed-point-format>.

Step 3.1.1. If the second <integer-value> is missing from $fpf$, let $d=0$; if the third is missing, let $a=0$. $d$ must not be negative.

Step 3.1.2. Let $dt$ be a <data-type> containing <real>, <fixed>, <decimal>, <scale-factor>$d$. $a$, used below, is implementation-defined.

Case 3.1.2.1. $d=0$ and $cv \geq 0$.

$w$ must satisfy $w=0$. Let $nw=1$, $pmin(a,0)$, $i=w-p$, and $j=p$. Let pic be "$$118(3j)129$".

Case 3.1.2.2. $d=0$ and $cv < 0$.

$w$ must satisfy $w=1$. Let $nw=1$, $pmin(a,0)$, $i=w-1$, and $j=p$. Let pic be "$$118(3j)9$".

Case 3.1.2.3. $d>0$ and $cv \geq 0$.

$w$ must satisfy $w=d$. Let $nw=1$, $pmin(a,0)$, $i=w-p$, and $j=p-1$. Let pic be "$$112(3j)(9).0d9$".

Case 3.1.2.4. $d>0$ and $cv < 0$.

$w$ must satisfy $w=d+2$. Let $nw=1$, $pmin(a,0)$, $i=w-2$, and $j=p-2$. Let pic be "$$112(3j)(9).9d0$".

Step 3.1.3. Attach <number-of-digits>: $p$ to $dt$. Remove from pic any repetition factor equal to 0 and the character following it. Let $dt$ be the <data-type> corresponding to the <picture> with concrete-representation pic, defined above. (For a fuller treatment, see Section 6.4.4.) Let $x$ be the <real-value> calculated as $x = 10^i + 0.5 \times \text{sign} \times 10^d$. Perform convert($dt$, $cvt$, $x$) to obtain $cv$ and then perform edit-numeric-picture($cv$, $pic$) to obtain a <character-string-value>, $cvv$. Return $cvv$.

Case 3.2. $fpf$ is a <floating-point-format>.

Step 3.2.1. If the second and third <integer-values> are missing from $fpf$, let $s$ be the converted <number-of-digits> of $dt$ for a target <data-type> of <real>, <float>, <decimal>, and let $d=a-1$. If only the third <integer-value> is missing, let $s=d+1$. $a$ must not be less than 0 and $a$ must not be less than $d$.

Step 3.2.2. Let $dt$ be a <data-type> which has <real>, <float>, <decimal>, <number-of-digits>: $a$. Perform convert-to-floating-decimal($dt$, $cvt$, $cv$) to obtain a <real-value>, $cvv$.

Step 3.2.3. There exists a unique representation of $cvv$ in the form $w \times 10^{ve}$, where $w$ is an integer and either $ve=0$ or $10^{ve-2} \times \text{sign} = 10^{ve-2}$.

Step 3.2.4. If $w<0$, let $i=0$; otherwise, let $i=1$. Let $j=\max(s-d,1)$. If $d=0$, let $k=0$; otherwise let $k=1$. Let $n$ be the implementation-defined size of the exponent field for <floating-point-format>. Let $nw=1-j-k-2-n$. Informally, $i$ indicates the need for a sign position in the mantissa field; $j$ is the number of digit positions before the decimal point; $k$ indicates the need for a decimal point in the mantissa field; $n$ is the excess of the field-size over the requirements of the specification for the particular value. If $w<0$ or abs($w$)$\geq 10^n$, perform raise-condition <size-condition>.
Step 3.2.5. Let pix be the picture with concrete-representation "null-\(\{GFV(k), k\}\)". Remove from pix any repetition factor whose value is zero and the character following it. Set pix be the data-type corresponding to pix. Perform edit-numeric-picture(pix, pix) to obtain a character-string-value, cvm, where cvm is a character-value containing ve. Let pix be the data-type corresponding to the picture with concrete-representation "null". Perform edit-numeric-picture(pix, pix) to obtain a character-string-value, cvm, where cvm is a character-value containing ve. Return a character-string-value containing the concatenation of cvm, the character-value |symbol|: x1, and cvm.

8.7.2.7 Output-string

Operation: output-string(sil, fv)

where sil is a stream-item-list,
fv is a file-value.

Step 1. For each stream-item, si, in sil in order, perform output-string-item(si, fv).

8.7.2.8 Output-string-item

Operation: output-string-item(si, fv)

where si is a stream-item,
fv is a file-value.

Step 1. If fv is a file-value then let fi be the file-information designated by fv.
Step 2. If fv is a file-value then perform steps 2.1 and 2.2.

Step 2.1. Perform evaluate-current-column(fv) to obtain an integer-value, cc, and perform evaluate-current-line(fv) to obtain an integer-value, cl. Let lsn be the evaluated-linesize in fi, and put be the evaluated-pagesize in fi, where they exist.

Step 2.2.

Case 2.2.1. fi contains "print", cc = 1ls and cl = pages.

Perform output-stream-item(stream-item: lsn, fi, tv). Perform value-to-condition(value-condition, tv), and on normal return go to Step 2.1.

Case 2.2.2. cc = 1ls, but Case 2.2.1 does not apply.

Perform output-stream-item(stream-item: lsn, fi, tv).

Case 2.2.3. (otherwise).

go to Step 3.

Step 3. Perform output-stream-item(sil, tv).
8.7.2.9 Output-stream-item

Operations: \texttt{output-stream-item}(si,fv)

where \( si \) is a \texttt{stream-item},
\( fv \) is a \texttt{file-value}.

Case 1. \( fv \) is a \texttt{file-value}.

Let \( fi \) be the \texttt{file-information} designated by \( fv \). \( fi \) must contain \texttt{stream}.
Let \( da \) be the \texttt{dataset} designated by the \texttt{dataset-designator} in \( fi \), and let \( cp \) be the \texttt{current-position} in \( da \).

If \( si \) has \texttt{pagemark} or \texttt{carriage-return} then \( fi \) must contain \texttt{print}. Append \( si \) to the \texttt{stream-item-list} in \( da \). Replace the immediate component of \( cp \) by a \texttt{designator} to the last \texttt{stream-item} in \( da \). If \( fi \) contains \texttt{print} and \( si \) is a \texttt{pagemark} then add 1 to the \texttt{integer-value} in the \texttt{page-number} in \( fi \).

Case 2. \( fv \) is a \texttt{scheme}.

\( si \) must contain a \texttt{symbol}. Let \( cvv \) be the \texttt{character-string-value} in the current \texttt{string-to-control} and let \( n \) be the \texttt{integer-value} of the \texttt{string-limit} of the current \texttt{string-to-control}. If \( cvv \) contains a \texttt{symbol} or if \( n \neq 0 \) then perform \texttt{raise-condition} (\texttt{error-condition}).

Let \( n \) be the \texttt{symbol} in \( si \). If \( cvv \) contains \texttt{null-character-string} then replace the immediate component of \( cvv \) with \texttt{character-value-list}: \texttt{character-values}: \( n \). Otherwise, append \texttt{character-value-list}: \( n \) to the \texttt{character-value-list} of \( cvv \).

8.7.2.10 Tab

Operation: \texttt{tab}(w,fv)

where \( w \) is an \texttt{integer-value},
\( fv \) is a \texttt{file-value}.

Step 1. Let \( eto \) be the \texttt{evaluated-tab-option} in the \texttt{file-information}, \( fi \) designated by \( fv \). \( w \) must be non-negative. If \( w = 0 \) then terminate this operation.

Step 2. Perform \texttt{evaluate-current-column}(fv) to obtain an \texttt{integer-value}, \( ec \).

Step 3.

Case 3.1. There are at least \( w \) \texttt{integer-values}, \( t \) in \( eto \) satisfying \( ec < t \cdot 1 < \text{lin} \), where \( \text{lin} \) is the \texttt{evaluated-linesize} in \( fi \).

Perform \texttt{output-tab}(fv) \( w \) times.

Case 3.2. (Otherwise).

Step 3.2.1. Perform \texttt{skip}(\texttt{integer-values}:1,fv).

Step 3.2.2. If the first \texttt{integer-value} in \( eto \) is not \( 1 \) then perform \texttt{output-tab}(fv).
8.7.2.10.1 Output-tab

Operation: \texttt{output-tab}(\texttt{fv})

where \texttt{fv} is a \texttt{file-value}.

Step 1. Perform \texttt{evaluate-current-column}(\texttt{fv}) to obtain an \texttt{integer-value}, \texttt{cc}.

Step 2. Let \texttt{tv} be the smallest \texttt{integer-value} in the \texttt{evaluated-tab-options} of the \texttt{file-information} designated by \texttt{fv}, which is greater than \texttt{cc}.

Step 3. Let \texttt{tv} be a \texttt{stream-item-list}, the \texttt{tv-cc-1} \texttt{stream-items} of which all contain \texttt{fv}. Perform \texttt{output-stringitem}(\texttt{tv}).

8.7.2.11 Put-line

Operation: \texttt{put-line}(\texttt{w}, \texttt{fv})

where \texttt{w} is an \texttt{integer-value},
\texttt{fv} is a \texttt{file-value}.

Step 1. Perform \texttt{evaluate-current-line}(\texttt{fv}) to obtain an \texttt{integer-value}, \texttt{cl}. Let \texttt{cp} be the \texttt{evaluated-page-size} and let \texttt{cp} be the \texttt{current-position} in the \texttt{file-information} designated by \texttt{fv}.

Step 2.

Case 2.1. \texttt{w} = \texttt{cl} and the node designated by \texttt{cp} contains a \texttt{pageblank}, a \texttt{lineblank} or a \texttt{carriage-return}.

Terminate this operation.

Case 2.2. \texttt{cl} < \texttt{w} ≤ \texttt{cp}.

Perform \texttt{skip}(\texttt{integer-value};(\texttt{w}-\texttt{cl});\texttt{fv}).

Case 2.3. \texttt{cp} ≥ \texttt{cl} ≥ \texttt{w} and Case 2.1 does not apply, or \texttt{w} > \texttt{cp} ≥ \texttt{cl}.

Perform \texttt{output-stream-item}(\texttt{stream-item};\texttt{lineblank};\texttt{fv}) \texttt{(w-cl)} times. Perform \texttt{raise-to-condition}(\texttt{endpage-condition};\texttt{fv}).

Case 2.4. (Otherwise).

Perform \texttt{put-page}(\texttt{fv}).

8.7.2.12 Put-page

Operation: \texttt{put-page}(\texttt{fv})

where \texttt{fv} is a \texttt{file-value}.

Step 1. Perform \texttt{output-stream-item}(\texttt{stream-item};\texttt{pageblank};\texttt{fv}).
8.7.3 Operations Applicable to Stream I/O

8.7.3.1 Skip

Operation: \texttt{skip}(w,fv)

where \(w\) is an \texttt{integer-value} which must not be negative,
\(fv\) is a \texttt{file-value}.

Step 1. Let \(fi\) be the \texttt{file-information} designated by \(fv\).

Step 2.

Case 2.1. The value of \(w\) is zero.
Perform output-stream-item(stream-item: \texttt{carriage-return},fv).

Case 2.2. \(fi\) contains \texttt{input}.
Perform repeatedly input-stream-item(fv) until either \(w\) \texttt{lines} have been returned or an \texttt{exec} is returned. In the latter case perform raise-lo-condition\texttt{end-file-condition}(fv).

Case 2.3. \(fi\) contains \texttt{output} but not \texttt{print}.
Perform output-stream-item(stream-item: \texttt{lines},fv) \(w\) times.

Case 2.4. \(fi\) contains \texttt{print}, and \(w \neq 0\).
Perform evaluate-current-line(fv) to obtain an \texttt{integer-value}, \(cl\). Let \(ep\) be the \texttt{evaluated-pagenumber} in the \texttt{file-information} designated by \(fv\). If \(cl > ep\) or if \(ep \geq cl\) then perform output-stream-item(stream-item: \texttt{lines},fv) \(w\) times. Otherwise perform output-stream-item(stream-item: \texttt{lines},fv) \(ep-cl\) times and then perform raise-lo-condition\texttt{end-page-condition}(fv).

8.7.3.2 Evaluate-current-column

Operation: \texttt{evaluate-current-column}(fv)

where \(fv\) is a \texttt{file-value}.

result: an \texttt{integer-value}.

Step 1. Let \(fi\) be the \texttt{file-information} designated by \(fv\), and do the \texttt{dataset} designated by the \texttt{dataset-designator} in \(fi\). Let \(cp\) be the node in \(ds\) designated by the \texttt{current-position} in \(fi\).

Step 2.

Case 2.1. \(cp\) contains an \texttt{alpha}, a \texttt{lines}, a \texttt{pagenumber}, or a \texttt{carriage-return}.
Return \texttt{integer-value}: 0.

Case 2.2. \(cp\) contains a \texttt{symbol} or an \texttt{exec}.

Let \(p\) be the \texttt{previous} of \(cp\) such that any \texttt{stream-items} in \(ds\) between \(p\) and \(cp\) contain \texttt{symbols}. Let \(n\) be the number of \texttt{symbols} between \(p\) and \(cp\), inclusive. Return \texttt{integer-value}: \(n\).
8.7.3.3 Evaluate-current-line

Operation: evaluate-current-line(fv)

where fv is a «file-value».
result: an «integer-value».

Step 1. Let fi be the «file-information» designated by fv, and dc the «data-set» designated by the «dataset-designator» in fi. Let cp be the node in dc designated by the «current-position» in fi.

Step 2.

Case 2.1. cp contains an «alpha» or a «page-mark».
Return «integer-value»: 1.

Case 2.2. (Otherwise).

Let p be the last «alpha» or «page-mark» in dc preceding cp such that all «statement-items» in dc between p and cp contain «symbolic», «line-markers», or «carriage-returns». Let n be the number of «line-markers» between p and cp, inclusive. Return «integer-value»: n+1.

8.7.3.4 Establish-next-data-item

Operation: establish-next-data-item

result: a «current-scalar-item» or «none».

Step 1. In the current «block-control» let cld be the last «data-list-indicator», and let cld be the last «data-item-indicator».

Step 2.

Case 2.1. cld is «undefined».

Set cld to designate the first component of the list designated by cld.

Case 2.2. cld designates an immediate component of the list designated by cld, but not the last component.

Set cld to designate the next immediate component of the list designated by cld.

Case 2.3. cld designates the last immediate component of the list designated by cld.

If cld designates a «current-scalar-item» then delete the current «current-scalar-items-list» and go to Step 4. If the current «data-item-control-list» contains precisely one «data-item-control» then return «none», otherwise perform test-termination-of-controlled-group to obtain t. If t is «type» then go to Step 4. Otherwise, let cld designate the first immediate component of the list designated by cld.
Case 3.1. The node designated by code immediately contains an \texttt{expression}, \texttt{e}.

Perform \texttt{evaluate-expression(e)} to obtain an \texttt{aggregate-value}, \texttt{av}, having a \texttt{basic-values}. Let \texttt{call} be a \texttt{current-scalar-item-list} having \texttt{n} elements whose \texttt{i}th element is a \texttt{current-scalar-item} containing the \texttt{i}th \texttt{basic-value} in \texttt{av}, \texttt{avl}, and the \texttt{data-type} corresponding to \texttt{avl} in the \texttt{data-description} immediately contained in \texttt{e}. Attach \texttt{call} to the current \texttt{block-control}. Append to the current \texttt{block-control}:

\texttt{<data-item-control>}
\texttt{<data-list-indicator>} designating \texttt{call}
\texttt{<data-item-indicator>} designating the first element of \texttt{call}.

Return the first element of \texttt{call}.

Case 3.2. The node designated by code immediately contains a \texttt{target-reference}, \texttt{tr}.

Perform \texttt{evaluate-target-reference(tr)} to obtain an \texttt{evaluated-target}, \texttt{et}.

Step 3.2.1.

Case 3.2.1.1. \texttt{et} immediately contains a \texttt{generated-\texttt{q}}.

Perform \texttt{expand-generated(\texttt{q})} to obtain a \texttt{generation-list}, \texttt{gl}, having \texttt{n} elements. Let \texttt{call} be a \texttt{current-scalar-item-list} having \texttt{n} elements whose \texttt{i}th element is a \texttt{current-scalar-item} containing the \texttt{i}th \texttt{evaluated-target}, \texttt{et[i]}, where \texttt{et[i]} is the \texttt{i}th element of \texttt{et}.

Case 3.2.1.2. \texttt{et} immediately contains an \texttt{evaluated-pseudo-variable-reference}, \texttt{epvr}.

Case 3.2.1.2.1. \texttt{epvr} contains an \texttt{<num-\texttt{py}>}, \texttt{<num-source-\texttt{py}>}, \texttt{<estr-\texttt{py}>}, \texttt{<error-\texttt{py}>}, \texttt{<attrlist-\texttt{py}>}, or \texttt{<empty-\texttt{py}>}.

Let \texttt{call} be a \texttt{current-scalar-item-list} containing \texttt{n} \texttt{current-scalar-items}.

Case 3.2.1.2.2. \texttt{epvr} contains an \texttt{<list-\texttt{py}>} or \texttt{<real-\texttt{py}>}.

Let \texttt{q} be the \texttt{generation} in \texttt{epvr}. Perform \texttt{expand-generated(\texttt{q})} to obtain a \texttt{generation-list}, \texttt{gl}. Let \texttt{n} be the number of \texttt{generations} in \texttt{gl}. Let \texttt{call} be a \texttt{current-scalar-item-list} containing \texttt{n} \texttt{current-scalar-items}.

Case 3.2.1.2.3. \texttt{epvr} contains a \texttt{<symet-\texttt{py}>}.

Let \texttt{q} be the \texttt{generation} in \texttt{epvr}. Perform \texttt{expand-generated(\texttt{q})} to obtain a \texttt{generation-list}, \texttt{gl}. Let \texttt{n} be the number of elements in \texttt{gl} and let \texttt{ql} denote the \texttt{i}th element in \texttt{gl}. Let \texttt{avl} be the \texttt{i}th \texttt{basic-value} in \texttt{avl} and let \texttt{avl[i]} be the element of the \texttt{basic-value-list} in \texttt{avl} which corresponds to \texttt{avl[i]}. If \texttt{epvr} has two \texttt{aggregate-values}, then let \texttt{avl} be the second and let \texttt{avl[i]} be the element of the \texttt{basic-value-list} in \texttt{avl} which corresponds to \texttt{avl[i]}. Let \texttt{call} be a \texttt{current-scalar-item-list} containing \texttt{n} \texttt{current-scalar-items}.

For \texttt{i=1,...,n}, replace the \texttt{numeral} of the \texttt{i}th \texttt{evaluated-pseudo-variable-reference} in \texttt{call} as follows:

- replace the \texttt{generation} by \texttt{avl[i]}
- replace the first \texttt{aggregate-value} by an \texttt{aggregate-value} containing \texttt{avl[i]}
- and, if a second \texttt{aggregate-value} is present, replace it by an \texttt{aggregate-value} containing \texttt{avl[i]}.
Step 3.2.2. Attach call to the current <block-state>. Append to the <data-item-control-list> in the current <block-control> a

<data-item-control>
  <data-list-indicator> designating call
  <data-item-indicator> designating the first element of call.

Return the first element of call.

Case 3.3. The node designated by odd immediately contains a <variable-reference>, vr.

Step 3.3.1. Perform evaluate-variable-reference (vr) to obtain a <generation>, g. Perform make-name-and-subscript-list (g) to obtain a <name-and-subscript-list>, mask. Let odd be the <evaluated-data-description> in q. Perform expand-eddi to obtain an <evaluated-data-description-list>, eddl having n elements. Let call be a <current-scalar-item-list> having n elements whose i'th element contains:

   the i'th basic-value in the <allocation-unit> designated by the <allocation-unit-designator> in g, where j is the i'th <storage-index> in q;

   the <data-type> from the i'th <evaluated-data-description> of eddl;

   a <data-name-field> containing a <symbol-list>, sdsf obtained from mask, the i'th <name-and-subscript> of mask, by performing Step 3.3.1.1.

Step 3.3.1.1. Let ddfs be the <symbol-list> obtained from the <dot-name-list> of mask by concatenating the <symbol-list>s of the <dot-names> in their natural order. Delete the final <symbol>; $j$ from ddfs.

If mask has a <comma-subscript-list>, csl, then replace ddfs by the <symbol-list> obtained by concatenating dip, <symbol-list>; symbol; (s); ..., and the <symbol-list> in the <comma-subscript-list> of call in their natural order, and replacing the final <symbol>; $j$ by <symbol>; (s).

Step 3.3.2. Attach call to the current <block-control>. Append to the current <data-item-control-list> a

<data-item-control>
  <data-list-indicator> designating call
  <data-item-indicator> designating the first element of call.

Return the first element of call.

Case 3.4. The node designated by odd immediately contains a tree of the form <do-spec>, dsp.

Let ddd1 be the first immediate component of the node designated by odd. Let ddd1 be the first immediate component of odd. Append to the current <data-item-control-list> a

<data-item-control>
  <data-list-indicator>
  <designator> designating ddd1;
  <data-item-indicator>
  <designator> designating ddd1.

Let odd be the current <data-item-indicator>. Perform establish-controlled-precipitation to obtain t. If t is <tree> then go to Step 3.

Case 3.5. The node designated by odd is a <current-scalar-item>, cal.

Return cal.

Step 4. If the current <data-item-control-list> contains more than one <data-item-control> then delete the last of them and go to Step 1. Otherwise return <tree>.

Chapter 8: Input/Output 281
§7.3.4.1 Expand-odd

Operation: \texttt{expand-odd}(odd)

where \texttt{odd} is an \langle evaluated-data-description \rangle.

result: an \langle evaluated-data-description-list \rangle.

Case 1. \texttt{odd} immediately contains a \langle data-description \rangle immediately containing a \langle dimensioned-data-description \rangle, odd.

Let \texttt{bpl} be the \langle bound-pair-list \rangle in \texttt{odd}, let \texttt{lbl1} and \texttt{ubl1} be respectively the values of the 1st \langle lower-bound \rangle and \langle upper-bound \rangle in \texttt{bpl}, and let \texttt{n} be the number of elements in \texttt{bpl}. Let \texttt{n} be the integer

\[ \prod_{i=1}^{n} (\texttt{ubl1} - \texttt{lb1} + 1) . \]

Let \texttt{odd1} be an \langle evaluated-data-description \rangle, \langle data-description \rangle: the immediate subtree of the \langle element-data-description \rangle in \texttt{odd}. Perform \texttt{expand-odd(odd1)} to obtain an \langle evaluated-data-description-list \rangle, \texttt{odd1}. Let \texttt{odd1} be an \langle evaluated-data-description-list \rangle with, in order, \texttt{n} replications of the subtree of \texttt{odd1}. Return \texttt{odd1}.

Case 2. \texttt{odd} immediately contains a \langle data-description \rangle immediately containing a \langle structure-data-description \rangle, odd.

For each \langle data-description \rangle, \texttt{d1} in \texttt{odd}, perform \texttt{expand-odd(odd-data-description\langle data-description \rangle: \texttt{d1})} to obtain an \langle evaluated-data-description-list \rangle, \texttt{odd1}. Let \texttt{odd1} be an \langle evaluated-data-description-list \rangle with, in order, the subtrees of these \texttt{odd1}. Return \texttt{odd1}.

Case 3. \texttt{odd} immediately contains a \langle data-description \rangle immediately containing an \langle item-data-description \rangle.

Return \langle evaluated-data-description-list \rangle: \texttt{odd}.

§7.3.4.2 Expand-generation

Operation: \texttt{expand-generation}(\texttt{g})

where \texttt{g} is a \langle generation \rangle.

result: a \langle generation-list \rangle.

Step 1. Let \texttt{a} and \texttt{b} be the \langle allocation-unit-designator \rangle in \texttt{g}. Let \texttt{odd} be the \langle evaluated-data-description \rangle in \texttt{g}. Let \texttt{all} be the \langle storage-index-list \rangle in \texttt{g}. Let \texttt{n} be the number of elements in \texttt{all}. Perform \texttt{expand-odd(odd)} to obtain an \langle evaluated-data-description-list \rangle, \texttt{odd1}. Let \texttt{qq} be a \langle generation-list \rangle without subnodes. For \texttt{i=1,\ldots,n}, let \texttt{odd1} be a copy of the \texttt{i}-th element of \texttt{all}, let \texttt{odd1} be a copy of the \texttt{i}-th element of \texttt{odd1}, and attach to \texttt{qq} a \langle generation \rangle: \texttt{odd1} and \langle storage-index-list \rangle: \texttt{all}.
8.7.3.4.3 Make-name-and-subscript-list

\[
\begin{align*}
\text{<name-and-subscript-list>} & : = \text{<dot-name-list> | <comma-subscript-list>} \\
\text{<dot-name>} & : = \text{<symbol-list>} \\
\text{<comma-subscript>} & : = \text{<symbol-list>}
\end{align*}
\]

Operation: \text{make-name-and-subscript-list}(\text{<vname>}, \text{<q>})

where \text{<vname>} is a \text{<variable-reference>},
\text{<q>} is a \text{<generation>},
result: a \text{<name-and-subscript-list>}

Step 1. Let \text{id} be the \text{<identifier>} in the \text{<declaration>} designated by the \text{<declaration-designator>} of \text{<vname>}. Perform identifier-to-dotname(\text{id}) to obtain a \text{<dot-name>,<dn>}. Let \text{<dnl>} be a \text{<dot-name-list>}. \text{<dn>}. \text{<dn>} has an \text{<identifier>} \text{xid} in \text{id}, in order, perform identifier-to-dotname(\text{xid}) to obtain a \text{<dot-name>,<xdn> and append <xdn> to <dnl>}. If \text{<vname>} has a \text{<comma-subscript-list>,<scdl>}, then create a \text{<comma-subscript-list>},\text{<scdl> from <xdl> by performing Step 1.1.

Step 1.1. Let \text{<icdl>} be a \text{<comma-subscript-list>} without supernodes. Let \text{<sn>} be the number of immediate subtrees of \text{xdl}, and let \text{<sn>}, i=1,\ldots,\text{<sn>}, be these subtrees. For i=1,\ldots,\text{<sn>}, append to \text{<icdl>} a \text{<symbol-list>,<sl>} obtained from \text{<xdl>} as follows: if \text{<sl>}, i is a \text{<subscript>,<sl>}, then \text{<sl>}, i is \text{<symbol-list>}, \{\text{<symbol>}, \{\}\}. Otherwise, let \text{<ex> be the <expression> in <xdl>}, perform evaluate-expression-to-integer(\text{<ex>}) to obtain an \text{<integer-value>,<iv>}, perform subscript-to-comma-subscript(\text{<iv>}) to obtain a \text{<symbol-list>,<vsl>}

Step 2. \text{<q>} contains an \text{<evaluated-data-description>,<data-description>,<d>}. Let \text{<xdl>} be the immediate subtrees of \text{id}. Perform expand-name-and-subscript(\text{id},\text{<xdl>}) to obtain a \text{<name-and-subscript-list>,<xans>}

Step 3. Perform Step 3.1 for each \text{<name-and-subscript>,<xans> of <xans>.

Step 3.1. If \text{xans} contains a \text{<dot-name-list>,<xdl>}, then replace \text{xdl} by the result of concatenating a copy of \text{xdl} with \text{xdl}. Otherwise attach a copy of \text{xdl} to \text{xans}.

Step 4. If \text{<vname>} has a \text{<comma-subscript-list>,<scdl>}, then perform Step 4.1 for each \text{<name-and-subscript>,<xans> of <xans>.

Step 4.1. Let \text{<cicdl>} be a copy of \text{<icdl>}. Let \text{<nast>} be the number of trees \text{<symbol-list>},\{\text{<symbol>}, \{\}\} in \text{<cicdl>}

Case 4.1.1. \text{<nast> = 0}.

If \text{xans} contains a \text{<comma-subscript-list>,<xcl>}, then replace \text{xcl} by the result of concatenating \text{cicdl} with \text{xcl}; otherwise attach \text{cicdl} to \text{xans}.

Case 4.1.2. (Otherwise).

For i=1,\ldots,\text{<nast>}, replace the \text{i}'th \text{<comma-subscript>,<symbol-list>},\{\} in \text{<cicdl> by a copy of the \text{i}'th \text{<comma-subscript> in <xans>}. Delete the first \text{nast} \text{<comma-subscript-list> from the <comma-subscript-list>,<scdl> of <xans>. If <xcl> is now without supernodes then replace it with <cicdl>; otherwise replace <xcl> by the result of concatenating <cicdl> and <xcl>.

Step 5. Return \text{<xans>}.
4.1.3.4.4 Expand-name-and-subscript

Operation: expand-name-and-subscript(ddi)

where ddi is an <item-data-description>, or a <dimensioned-data-description>,
or a <structure-data-description>.

result: a <name-and-subscript-list>.

Case 1. ddi is an <item-data-description>.

Let xmx be a <name-and-subscript> without subnodes. Return <name-and-subscript-list>: xmx.

Case 2. ddi is a <dimensioned-data-description>.


Step 2.2. For each (bound-pair),bp in the (bound-pair-list) of ddi, taken in order from left-to-right perform Steps 2.2.1 and 2.2.2.

Step 2.2.1. Let lb and ub be <integer-values> equal to the <integer-values> in the <lower-bound> and <upper-bound> of bp, respectively. Let n be ub-lb+1. Let nmsl[i], i=1,...,n, be n copies of nmsl. For each i, i=1,...,n, let nmsl[i] be nmsl[i-1]+1, let xmx be an <integer-values>: lb+1-1, perform subscript-to-comma-subscript(xmx) to obtain a <comma-subscript>, xcs, and perform Step 2.2.1.1.

Step 2.2.1.1. For each <name-and-subscript>,xmx in nmsl perform Step 2.2.1.1.1.

Step 2.2.1.1.1. If xmx has a <comma-subscript-list>, xcl, then insert xcs before the first component of xcl. Otherwise attach to xmx a <comma-subscript-list>: xcs.

Step 2.2.2. Let nmsl be the <name-and-subscript-list> obtained by concatenating the n nmsl[i] in the order i=1,...,n.

Step 2.3. Return nmsl.

Case 3. ddi is a <structure-data-description>.

Step 3.1. Let n be the number of <identifier>s in the <identifier-list>, idl in ddi. Let nmsl[i], i=1,...,n, be a <name-and-subscript-list> without subnodes. For i=1,...,n, let nmsl[i] be nmsl[i-1]. Let idl be the i'th <identifier> in idl, let ddi be the <data-description> in the i'th <member-description> in ddi, let xmx be the immediate subtree of ddi, and perform Step 3.1.1.

Step 3.1.1. Perform identifier-to-dotsame(idl) to obtain a <dot-name>, dnl. Perform expand-name-and-subscript(xdl) to obtain a <name-and-subscript-list>, nmsl2. Perform Step 3.1.1.1 for each <name-and-subscript>,xmx in nmsl2.

Step 3.1.1.1. If xmx has a <dot-name-list>, xdl, then insert dnl before the first subtree of xdl. Otherwise attach to xmx a <dot-name-list>: dln.

Step 3.2. Let nmsl be the <name-and-subscript-list> obtained by concatenating the n nmsl[i], in the order i=1,...,n. Return nmsl.
8.7.3.4.5 Subscript-to-comma-subscript

Operation: \texttt{subscript-to-comma-subscript}\(\texttt{e}\)

where \(e\) is an \texttt{<integer-value>},

result: a \texttt{<comma-subscript>},

Step 1. If \(e=0\) then let \(n\) be 1 and let \texttt{mod} be the \{symbol\}: \texttt{02}. Otherwise let \(n\) be the number of digits required for the decimal representation of the absolute value of \(e\) and let \texttt{mod} be \{symbol\}: \texttt{1\ldots n}, be the \{symbol\} representing the digits of that representation.

Step 2. Let \texttt{call} be a \{symbol-list\} containing, in order, \texttt{mod}, \texttt{1\ldots n+1}, \{symbol\}: \texttt{\ldots e\}}, \{symbol\}: \texttt{\ldots e\}}. If \(e>0\) then replace \texttt{call} by the result of concatenating a \{symbol-list\}: \{symbol\}: \texttt{\ldots e\}} with \texttt{call}.

Step 3. Return \texttt{<comma-subscript> call}.

8.7.3.4.6 Identifier-to-dotname

Operation: \texttt{identifier-to-dotname(id)}

where \(id\) is an \texttt{<identifier>},

result: a \texttt{<dot-name>},

Step 1. Let \texttt{call} be a \{symbol-list\} obtained by concatenating the \{symbol-list\} of \(id\) with a \{symbol-list\}: \{symbol\}: \texttt{1}.

Step 2. Return \texttt{<dot-name> call}.

8.7.3.5 Establish-next-format-item

Operation: \texttt{establish-next-format-item}

result: a \texttt{<format-item>},

Step 1. Let \(fc\) be the list \texttt{<format-control> \ldots \texttt{<format-control-list>}}, \(flp\) be the \texttt{<format-specification-list-designator>}, \(fl\) be the \texttt{<format-list-index>}, \(fli\) be the \texttt{<format-iteration-value>}, \(fli\) be the immediate subnode of the \texttt{<format-list-index>}, and \(fli\) be the immediate subnode of the \texttt{<format-iteration-value>}. Let \(fli\) be the \texttt{<format-specification-list> designated by \texttt{flp}\}}, and let \(n\) be the number of immediate components of \(fli\).

Step 2.

\begin{enumerate}
  \item \texttt{Case 2.1.} \(fli\) is less than \(n\).
    \begin{enumerate}
      \item Add 1 to \(fli\), giving a value, \(m\).
    \end{enumerate}
  \item \texttt{Case 2.1.1.} The \(m\)'th element of \(fli\) is a \texttt{<format-item>}. \(fli\).
    \begin{enumerate}
      \item Let \(fli\) be a copy of \(fli\). Perform evaluate-format-item(\(fli\)) to obtain a \texttt{<format-item>}. \(fli\). Return \(fli\).
    \end{enumerate}
\end{enumerate}
Case 2.1.2. The m'th element of fli is a <format-iteration>-fli.

Let e be the <expression> in the <format-iteration-factor> immediately contained in fli. Perform evaluate-format-expression(e) to obtain an <integer-value>-intg. intg must not be negative. If intg > 0 then append to the current <format-control-list> a

<format-control>:
  <format-specification-list-designator> designating the
  <format-specification-list> in the m'th element of the list designated by fli:
  <format-list-index>;
  <integer-values>:
  intg;
  <format-iteration-values>:
  intg;
  <format-iteration-index>:
  intg;
  ...

Go to Step 1.

Case 2.2. fli equals n, and fci is the only <format-control> in the current <format-control-list>.

Set fli to 0 and go to Step 1.

Case 2.3. fli equals n, and there are at least two <format-controls> in the current <format-control-list>.

Case 2.3.1. fli is less than fiv.

Add 1 to fli, set fli to 0 and go to Step 1.

Case 2.3.2. fli equals fiv.

Delete fci, and go to Step 1.

8.7.3.6 Evaluate-Format-Item

When a <format-item> contains a <data-format> or a <control-format>, all <expression>s contained in it are evaluated, and a tree having the same structure as the <format-item> that with <integer-values> replacing the <expression>s is returned. When a <format-item> contains a <remote-format>, this <remote-format> specifies a <format-statement>; designators of the <format-specification-list> of the statement, and of the <format-statement> itself, are placed in the current <format-control-list> (the latter designator allows the correct environment to be established for the evaluation of <expression>s.) The operation establish-next-format-item is then re-invoked to scan the <format-specification-list>.

Operation: evaluate-format-item(fli)

where fli is a <format-item>.

result: a <format-item>.

Case 1. fli contains a <control-format> or a <data-format>.

Let cfi be a copy of fli. For each <expression>-e, in cfi, chosen in any order, perform evaluate-format-expression(e) to obtain an <integer-value>-iv and replace e by iv. Return cfi.

Case 2. fli contains a <remote-format>-rf.

Step 2.1. If there is in the current <format-control-list> a <remote-block-state>, then let rhs be the last such <remote-block-state>. Attach a copy of rhs to the current <block-control>.
Case 2.1.1. If contains a <variable-reference>, vr.

Perform evaluate-variable-reference(vr) to obtain a <generate> whose value (see Section 7.1.1) contains a single <format-value>, fv.

Case 2.1.2. If contains a <named-constant-reference>, ncr.

Perform evaluate-named-constant-reference(ncr) to obtain an aggregate-value containing a single <format-value>, fv.

Step 2.2. If there exists a <remote-block-state>, rbs in the current <block-control> then delete rbs. If contains a <block-state-designator>, bsd and a <format-statement-designator>, fsd. The current <format-control-list> must not contain a <format-control> with a <format-statement-designator> equal to fsd. The current <block-state-list> must contain a <block-state> designated by bsd. If rbs immediately contains a <variable-reference> whose <data-type> has <local> then rbs must designate the current <block-state>.

Step 2.3. Let ful designate the <format-specification-list> in the node designated by ful. Append to the current <format-control-list> a

<format-control>:
<format-specification-list-designator>: ful;
<format-list-index>:
  <integer-value>:
    0;
  ful
  <remote-block-state>:
    bsd.

Step 2.4. Perform establish-next-format-item to obtain a <format-item>, fli. Return fli.

8.7.3.6.1 Evaluate-format-expression

An <expression> which appears in a <format-specification-list> is evaluated and converted to an <integer-value>. However, when such an <expression> occurs in a <format-statement> (and is accessed via a <remote-format>), the necessary environment must be set up for its evaluation. This is achieved by temporarily placing a <remote-block-state> in the current <block-control>.

Operation: evaluate-format-expression(e)

where e is an <expression>.

result: an <integer-value>.

Case 1. There exists in the current <format-control-list> a <remote-block-state>.

Let rbs be the last such <remote-block-state>. Attach a copy of rbs to the current <block-control>. Perform evaluate-expression-to-integer(e) to obtain an <integer-value>, int. Delete the <remote-block-state> from the current <block-control>. Return inty.

Case 2. (Otherwise).

Perform evaluate-expression-to-integer(e) to obtain an <integer-value>, int. Return int.
Chapter 9: Expressions and Conversion

9.0 Introduction

This chapter defines operations that are involved in the evaluation of expressions. Included are definitions for built-in functions and the rules for data conversion. The main sections are:

9.1 Aggregate Expressions
9.2 Prefix Operators
9.3 Infix Operators
9.4 Built-in Functions
9.5 Conversion

9.1 Aggregate Expressions

9.1.1 Scalar and Aggregate Types

9.1.1.1 Aggregate Type of a Data Description

An «aggregate-type» is an alternative way of specifying the part of a «data-description» which describes the structuring of elements into an aggregate. For each «data-description» there is an associated «aggregate-type» determined as follows:

- Ignoring all «identifier-list» immediate components of «structure-data-description» and all components of «item-data-description», the shapes of the trees for the «data-description» and the «aggregate-type» are identical, and categories correspond as follows:
  - «data-description»
  - «element-data-description»
  - «bound-pair-list» and all its components
  - «structure-data-description»
  - «member-description»
  - «item-data-description»

- «aggregate-type»
  - «dimensioned-aggregate-type»
  - «element-aggregate-type»
  - «bound-pair-list» and equal components if «integer-value»; otherwise «external» components
  - «structure-aggregate-type»
  - «member-aggregate-type»
  - «scalar»

In many contexts, properties of «data-description» are stated in terms of properties of their associated «aggregate-types» without explicitly building the corresponding tree or even using the term "associated".

For example: "The result has aggregate-type: «scalar»." is equivalent to "the result «data-description» immediately contains an «item-data-description»."

9.1.1.2 Scalar Elements

The scalar-elements of an «aggregate-value» are the components of its «basic-value-list»; the scalar-elements of a «generation» are the components of its «storage-index-list».

The number-of-scalar-elements is defined for a «data-description» by the operation scalar-elements-of-data-description (Section 7.1.1), and applies equally to a «data-description>, a «generation» containing that «data-description>, the associated «aggregate-type» of that «data-description>, and an «aggregate-value» containing that «aggregate-type>.
4.1.1.2 Treatment of Scalars

To facilitate the systematic treatment of expression evaluation and assignment, operations such as evaluate-expression and evaluate-assignment-reference are defined so that they return or accept aggregate-values in all cases. There are, of course, many contexts in which a value's aggregate-type is known a priori, to immediately contain scalar, but these are treated as degenerate cases of the general mechanism.

4.1.1.4 Compatibility

In general, the aggregate-values involved in operations, such as infix-oidl, are not required to have equal aggregate-types but only to have aggregate-types that are compatible. The result of such an operation has an aggregate-type that is the common aggregate-type of the operand aggregate-types. These terms are defined as follows:

Let t1, ..., tn be a set of aggregate-types. If n=1, the set is compatible, and its common aggregate-type is t1. If n>2, the set is compatible if and only if t1, ..., tn are compatible and t1 is compatible with the common aggregate-type of t2, ..., tn. In this case, the common aggregate-type of t1, ..., tn equals the common aggregate-type of t1 and ct. Finally, if n=2, compatibility and the common aggregate-type are determined as follows:

Case 1. At least one of t1, t2 immediately contains scalar.

Assume t1 immediately contains scalar. Then the set (t1, t2) is compatible, and its common aggregate-type is t1.

Case 2. Both t1 and t2 immediately contain structure-aggregate-type.

The set (t1, t2) is compatible if and only if

11 t1 and t2 have the same number of components n, in their member-aggregate-type-lists;

12 for i ≤ n, the i'th member-aggregate-type of t1 is compatible with the i'th member-aggregate-type of t2.

When t1 and t2 are compatible, their common aggregate-type immediately contains a structure-aggregate-type with n member-aggregate-types, the i'th having the common aggregate-type of the i'th member-aggregate-type of t1 and t2.

Case 3. One of t1, t2 immediately contains a dimensioned-aggregate-type and the other does not.

Assume t1 immediately contains a dimensioned-aggregate-type. The set (t1, t2) is compatible if and only if the element-aggregate-type of t1 is compatible with t2.

In this case, the common aggregate-type of t1 and t2 immediately contains a dimensioned-aggregate-type with a bound-pair-list equal to that of t1 and element-aggregate-type containing the same immediate component as the common aggregate-type of t2 and the element-aggregate-type of t1.

Case 4. Both t1 and t2 have dimensioned-aggregate-type.

The set (t1, t2) is compatible if and only if

11 the bound-pair-list of t1 and t2 are equal except that where a bound-pair's lower-bound occurs in one, the other may have a bound-pair containing a lower-bound and an upper-bound;

12 the element-aggregate-types of t1 and t2 are compatible.

When t1 and t2 are compatible, their common aggregate-type, if exists, immediately contains a dimensioned-aggregate-type whose element-aggregate-type contains
the same immediate components as the common <aggregate-type> of the <element-aggregate-types> of t[i] and t[j]. The <bound-pair-list> of t[i] equals that of t[j] except that, when a <bound-pair> in t[i] has <asterisk>, the corresponding <bound-pair> of t[j] is used.

9.1.1.5 correspondence

The following two sections extend the notion of node correspondence (see Section 9.1.1.1) to apply to scalar-elements of <aggregate-values> and <generations> whose associated <aggregate-types> are compatible, and <data-types> in <data-descriptions> whose associated <aggregate-types> are compatible. The extended notion of correspondence is used extensively in defining aggregate operations and in defining the result <data-descriptions> of <expression>.

9.1.1.5.1 Correspondence of Scalar Elements

Let ax and ay be compatible <aggregate-types>, with number-of-scalar-elements na and ny, respectively. Let x[i], ..., x[nx] and y[i], ..., y[ny] be sequences of scalar elements. (For example, x[i], ..., x[nx] might be the scalar-elements of an <aggregate-value> whose <aggregate-type> is ax.) The <aggregate-types> ax and ay determine a correspondence between the x[i], ..., x[nx] and the y[i], ..., y[ny] as follows:

Case 1. At least one of ax and ay immediately contains <scalar>.

Assume ax immediately contains <scalar>. Then nx=1, and x[i] corresponds to all of the y[i], y[i+1], ..., y[ny].

Case 2. Both ax and ay immediately contain <structure-aggregate-type>.

Let p be the number of <member-aggregate-types> in ax (also ay), and let mx[i] (respectively, my[i], ly[i]) be the number of scalar-elements of the i-th <member-aggregate-type> in ax (respectively, ay). Let mxi and myi be 0, when i=1. Let mx[i] be the sum of mxi[j], li≤j≤p, where 2mxi and let mxi be the sum of mxi[j], li≤j≤p. Then the correspondence between the x[i], ..., x[nx] and the y[i], ..., y[ny] is such that the x[i+n], ..., x[nx] correspond to the y[i+n], ..., y[ny]. (Informally, the correspondence matches elements of the i-th member of x with those of the i-th member of y.) Between these groups the correspondence is determined by the <aggregate-types> of the i-th <member-aggregate-types> of ax and ay.

Case 3. One of ax and ay immediately contains a <dimensioned-aggregate-type> and the other does not.

Assume ax has the <dimensioned-aggregate-type>. Let n be the number of scalar-elements of the <element-aggregate-type> of ax. Then the correspondence between the x[i], ..., x[nx] and the y[i], ..., y[ny] is such that the x[i+n], ..., x[nx] correspond to the y[i], y[i+1], ..., y[ny]. (Informally, the correspondence matches the scalar-elements of y with each of the elements of the array x.) Between these groups, the correspondence is determined by the <element-aggregate-type> of ax and by ay.

Case 4. Both ax and ay have <dimensioned-aggregate-types>.

Let mx (respectively, my) be the number of scalar-elements of ax (respectively, ay). The correspondence between the x[i], ..., x[nx] and the y[i], ..., y[ny] is such that the x[i+n], ..., x[nx] correspond to the y[i+n], ..., y[ny], where i=0, ..., n-1. (Informally, the i-th element of the array x corresponds to the i-th element of the array y.) Between these groups, the correspondence is determined by the <element-aggregate-type> of ax and by ay.
3.1.1.5.2 Correspondence of Data Types

Let \( x \) be an \(<aggregate-value\> or a \(<generation\>\), and let \( d \) be a \(<data-description\> whose associated \(<aggregate-type\> equals the \(<aggregate-type\) of \( x \). Let \( x_{i1}, x_{i2}, \ldots \) be the scalar-elements of \( x \). Then the \(<data-type\>s in \( d \) correspond to the \( x_{i1}, x_{i2}, \ldots \) as follows:

Case 1. \( d \) immediately contains an \(<item-data-description\>.

The single \(<data-type\> simply contained in \( d \) corresponds to the single scalar-element \( x_{i1} \).

Case 2. \( d \) immediately contains a \(<structure-data-description\>.

Let the \(<member-data-description\>s simply contained in \( d \) be \( d_{i1}, \ldots, d_{i1+p} \). Let \( n_{i1}, i_{12}, \ldots, \) be the number of scalar-elements of \( d_{i1} \). Let \( n_{i2} \) be 0 when \( i = 1 \). Let \( n_{i1} \) be the sum of \( n_{i1}, i_{12}, \ldots, i_{12+p} \). The \(<data-type\>s in \( d_{i1} \) correspond to the scalar-elements \( x_{i1+j1}, i_{12}, \ldots, i_{12+p} \). The correspondence being determined by the \(<aggregate-type\> of \( d_{i1} \).

Case 3. \( d \) immediately contains a \(<dimensioned-data-description\>.

Let \( e \) be the \(<element-data-description\> simply contained in \( d \), and let \( a \) be the number of scalar-elements of the \(<aggregate-type\> of \( e \). Then for \( i = 0, 1, \ldots \), the \(<data-type\>s in \( e \) correspond to the scalar-elements \( x_{i1+1}, i_{12}, \ldots, i_{12+p} \). The correspondence being determined by the \(<aggregate-type\> of \( e \).

Now the correspondence is easily extended to cover cases where the \(<aggregate-type\>s of \( d \) and \( x \) are merely compatible. Suppose that all \(<bound-pair\>s in \( d \) contain \(<integer-values\>\), and let \( y \) be any \(<aggregate-value\> whose \(<aggregate-type\> is the same as that of \( d \). Then a \(<data-type\> in \( d \) corresponds to a scalar-element in \( x \) if and only if they both correspond to some scalar-element in \( y \). If some \(<bound-pair\>s in \( d \) have \(<asterisk\>\), consider instead a \(<data-description\> that is equal to \( d \) except for \(<bound-pair\>s and whose \(<aggregate-type\> is the same as that of \( x \).

Finally the correspondence can be established between \(<data-type\>s of two \(<data-description\>s, \( d_1 \) and \( d_2 \), as follows. Let \( x \) be an \(<aggregate-value\> whose \(<aggregate-type\> equals the common \(<aggregate-type\> of \( d_1 \) and \( d_2 \) except that all \(<bound-pair\>s have \(<lower-bound\> equal to 1 and \(<upper-bound\> equal to 1. Then a \(<data-type\> in \( d_1 \) corresponds to a \(<data-type\> in \( d_2 \) if and only if both correspond to some scalar-element in \( x \).
9.1.1.6 Generate-aggregate-result

Most operations that return an aggregate-value work in the same general way:

The operands are evaluated, results of any arrays are checked for compatibility, and the aggregate-type of the result is determined. Then each scalar-element of the result is determined by performing a sequence of steps involving only scalar-elements of the operand values.

The various aggregate operations differ only in the details of this sequence of scalar steps. To simplify the description of these aggregate operations a special macro operation, generate-aggregate-result, is used.

Macro Operation: generate-aggregate-result

A reference to this macro operation always occurs as the first statement in the body of an operation, f. The remainder of the body will be a sequence of steps numbered 1,...,nstep, where nstep is the number of steps needed. To perform f, substitute these steps into the following macro body immediately after Step 3.2, rename them and their corresponding Steps and Cases as Step 3.2.1, Step 3.2.2, .... Step 3.2.nstep, and, in Step 3.2, replace "nstep" by its actual value. Rename references to these Steps and Cases correspondingly. Then interpret the expanded macro body as an operation body in the normal way.

Macro Body:

Step 1. For each operand, op1j1, j=1,...,(number of operands)-1, taken in any order, perform evaluate-expression(op1j1) to obtain an aggregate-value, val1j.

Step 2. If two or more of the val1j have aggregate-type containing corresponding bound-pair-lists, then corresponding bound-pair-lists must be equal.

Step 3. Let atp be the common aggregate-type of the val1j. Let nbe be the number-of-scalar-elements of atp. For i=1,...,nbe, taken in any order, perform Steps 3.1 and 3.2.

Step 3.1. Let scalar-result-type be that data-type of nd that corresponds to scalar-element, i, in an aggregate-value whose aggregate-type is atp. For each operand, op1j1, let the scalar data-type of op1j1 be the corresponding data-type of op1j1, and let the scalar-value of op1j1 be the corresponding scalar-element in val1j.

Step 3.2. Perform Steps 3.2.1 through 3.2.nstep to obtain scalar-result. Let availi equal scalar-result.

(Replace this line with the steps following the macro reference, renumbered as above.)

Step 4. Return an aggregate-value whose aggregate-type is atp, and whose basic-value-list contains avail1i, avail2i, ....,availnbei.

Note: (1) An operation referencing generate-aggregate-result always has a first operand, nd, which is a data-description. Its other operands will be expressions or arguments. To refer to the scalar-elements of the operands, the result, and their respective data-type, it uses the four special terms defined in Step 3, namely: scalar-value, scalar data-type, scalar-result, and scalar-result-type. In the expanded macro body, scalar-values, scalar-result, and scalar-result-type are local variables, together with nd, atp, avail1i, and val1i.

(2) The concepts of scalar-value, scalar data-type, scalar-result, and scalar-result-type are also used in an extended sense in the section on "Attributes" and "Constraints", where they refer to the parts of a data-description being constructed or checked by the Translator, rather than to parts of nd.
DECLARE A(10),
2X FLOAT SIGNED,
2Y CHAR(32),
B(10) FIXED SIGNED,
C(10) FLOAT DECIMAL,
1 D, 2 X, 2 Y, 2 Z;
A + B + C; A = B+C; C = B+C(10);

Consider the (infix-expression) "A+1+2" in the first assignment-statement. The aggregate-type of "A+1" immediately contains a structure-aggregate-type whose member-aggregate-type-list has two components, both of which have aggregate-type <scalar>. The aggregate-type of B also immediately contains a structure-aggregate-type, but its member-aggregate-type-list has three components. Thus the two aggregate-types are not compatible (section 9.1.1.8, case B). Because the two aggregate-types are not compatible, "A+1+2" does not satisfy the constraints for (infix-expressions). So data-description is defined for "A+1+2", and the program is rejected by the Translator.

Consider next the (infix-expression) "B+C" in the second assignment-statement. B and C have the same aggregate-type, i.e., this immediately contains a dimensioned-aggregate-type whose bound-pair-list contains a single entity, and with element-aggregate-type <scalar>. Thus B and C have compatible aggregate-types, and the aggregate-type of "B+C" is their common aggregate-type, i.e., C. Once the aggregate-type of "B+C" is known, determination of the result of the assignment-description, C, reduces to the determination of the data-type components of C (i.e., the result-type) from the corresponding components of the data-descriptions of B and C. In this rule there is only a single result-type. It has (arithmetic), with the common derived mode of B and C, which has (real), the common derived class of B and C, which has (binary), etc.

Evaluation of "B+C" is by the general procedure described in section 9.1.1.6. First the expressions "B" and "C" are evaluated. Next the aggregate-types of B and C are checked for compatibility. Because of the general properties of expressions, the aggregate-types, t(B) and t(C), of the values of B and C will be the same as the aggregate-types of the expressions themselves, except that in t(B) and t(C) any (bound-pair) will have a pair of integer-values as its components instead of an entity. Thus it is really only necessary to check that bound-pairs match. In this case, the (bound-pair) in t(B) has components 1 and 10, while that in t(C) has components 1 and 9. Thus the aggregate-types are not compatible, and the program fails to satisfy the test in step 2 of generate-aggregate-result.

Now consider the (assignment-statement) "A=B+C(1)"; which is actually correct! Evaluation of B yields an aggregate-value whose aggregate-type is t(B) (see above), and which has 10 scalar-elements, b(1), ..., b(10). The value of "C(1)" is an aggregate-value with aggregate-type <scalar> and which has a single scalar-element, c. The aggregate-value of "B+C(1)" is generated by the general procedure in section 9.1.1.6. The 1'th scalar-element of the result is generated by "addition" of b(1) and c. Here the precise result of the "addition" depends on the result-type and the data-types of B and C as well as on the value of b(1) and c.

Evaluation of the (variable-reference) "A" yields a generation, qa, whose aggregate-type immediately contains a dimensioned-aggregate-type, t(A), containing a single (bound-pair) with components 1 and 10. The aggregate-type of t(A) has a structure-aggregate-type with two member-aggregate-types, both with aggregate-type <scalar>, hence qa has 20 scalar-elements, a(1) ..., a(20). The aggregate-type of the value of "B+C(1)" is t(A), which is promotable to the aggregate-type, t(A) (section 7.5.3.11). Therefore the assignment can be carried out. The assignment involves corresponding scalar-elements. The first element in the value of B+C(1) is assigned to the components of the (bound-pair) 1 follows by def. and a(12). The second element is assigned to a(1) and a(1), etc. The details of the assignment depend on the corresponding data-type in the data-description of qa. Assignment of a(1), a(1), ..., is controlled by the data-type of A, which has (float) and (binary); assignment to a(1), a(1), ..., is controlled by the data-type of A, which has (character).
9.1.2 INTEGER TYPE

The term integer-type means a <data-type> which has <arithmetic> containing <real>, <fixed>, <binary> <number-of-digits> m and <scale-factor> s. m is implementation-defined and depends on the particular context in which the term is used.

9.1.2.1 Evaluate-expression-to-integer

Operation: evaluate-expression-to-integer(s)

where s is an <expression>.

result: an <integer-value>.

Step 1. Perform evaluate-expression(s) to obtain an <aggregate-value>, av. Let tv be the <basic-value> in av. Let dt be the <data-type> of s.

Step 2. Let dtX be a <data-type> which is integer-type (Section 9.1.2). Perform convert(dtX, av, tv) to obtain a <real-value>, rv.

Step 3. Return as <integer-value> with the same component as rv.

9.1.3 DERIVED DATA TYPES

In general, a scalar <data-type> of an expression or a target <data-type> for a conversion operation is derived from other <data-type>s. Special terminology is introduced for the most common cases.

9.1.3.1 Derived Base, Scale, and Mode

Let S be a set of one or more <data-type>s which have <computational-type>.

The derived <base> of S has <binary> if any element of S has <arithmetic> with <base>; <binary>; or if any element of S has <string> with <string-type>; <bit>. Otherwise the derived <base> of S has <decimal>.

The derived <scale> of S has <float> if any element of S has <arithmetic> (including <arithmetic> in <computed-numeric> with <scale>; <float>). Otherwise the derived <scale> has <fixed>.

The derived <mode> of S has <complex> if any element of S has <arithmetic> (including <arithmetic> in <computed-numeric> with <mode>; <complex>). Otherwise the derived <mode> has <real>.

When the set S contains at least two <data-type>s, the terminology derived common <base>, derived common <scale>, and derived common <mode> will generally be used.
### 9.1.2.3 Converted Precision

Given a source \(<\text{data-type}>\) which has \(<\text{computational-type}>\), a target \(<\text{base}>\), and a target \(<\text{scale}>\), the source \(<\text{data-type}>\) has associated with it a converted \(<\text{precision}>\), with a converted \(<\text{number-of-digits}>\) and a converted \(<\text{scale-factor}>\). For a source \(<\text{data-type}>\) which has \(<\text{arithmetic}>\), including \(<\text{arithmetic}>\) in \(<\text{picture-numeric}>\), the converted \(<\text{precision}>\), converted \(<\text{number-of-digits}>\) and converted \(<\text{scale-factor}>\) are defined in Table 9.1.

For a source \(<\text{data-type}>\) which has \(<\text{string}>\) with \(<\text{string-type}>\): \(<\text{big}>\); the converted \(<\text{precision}>\) is the same as for a source \(<\text{data-type}>\) which has \(<\text{arithmetic}>\) with \(<\text{base}>\): \(<\text{binary}>\); \(<\text{scale}>\): \(<\text{fixed}>\); \(<\text{scale-factor}>\): \(<\text{0}>\); and \(<\text{number-of-digits}>\): \(<\text{N}>\); where \(<\text{N}>\) is the maximum \(<\text{number-of-digits}>\) allowed for \(<\text{binary}>\) and \(<\text{fixed}>\).

For a source \(<\text{data-type}>\) which has \(<\text{string}>\) with \(<\text{string-type}>\): \(<\text{character}>\); or has \(<\text{picture-numeric}>\); the converted \(<\text{precision}>\) is the same as for a source \(<\text{data-type}>\) which has \(<\text{arithmetic}>\) with \(<\text{base}>\): \(<\text{decimal}>\); \(<\text{scale}>\): \(<\text{fixed}>\); \(<\text{scale-factor}>\): \(<\text{0}>\); and \(<\text{number-of-digits}>\): \(<\text{N}>\); where \(<\text{N}>\) is the maximum \(<\text{number-of-digits}>\) allowed for \(<\text{decimal}>\) and \(<\text{fixed}>\).

#### Table 9.1. Table of Converted Precisions as a Function of Target and Source Attributes.

<table>
<thead>
<tr>
<th>Target &lt;base&gt; and &lt;scale&gt;</th>
<th>Source &lt;base&gt; and &lt;scale&gt;</th>
<th>Binary Fixed</th>
<th>Decimal Fixed</th>
<th>Binary Float</th>
<th>Decimal Float</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary Fixed</td>
<td>p&lt;sup&gt;*&lt;/sup&gt;&lt;sub&gt;q&lt;/sub&gt;</td>
<td>p&lt;sup&gt;*&lt;/sup&gt;min(ceil&lt;sub&gt;(p&lt;sub&gt;3.12&lt;/sub&gt;)&lt;/sub&gt; +1,0)</td>
<td>q&lt;sup&gt;*&lt;/sup&gt;ceil&lt;sub&gt;(q&lt;sub&gt;3.12&lt;/sub&gt;)&lt;/sub&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decimal Fixed</td>
<td>p&lt;sup&gt;*&lt;/sup&gt;min(ceil&lt;sub&gt;(p&lt;sub&gt;3.12&lt;/sub&gt;)&lt;/sub&gt; +1,0)</td>
<td>p&lt;sup&gt;*&lt;/sup&gt;&lt;sub&gt;p&lt;/sub&gt;</td>
<td>q&lt;sup&gt;*&lt;/sup&gt;&lt;sub&gt;q&lt;/sub&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary Float</td>
<td>p&lt;sup&gt;*&lt;/sup&gt;&lt;sub&gt;p&lt;/sub&gt;&lt;sub&gt;min(p,0)&lt;/sub&gt;</td>
<td>p&lt;sup&gt;*&lt;/sup&gt;min(ceil&lt;sub&gt;(p&lt;sub&gt;3.12&lt;/sub&gt;)&lt;/sub&gt;,0)</td>
<td>p&lt;sup&gt;*&lt;/sup&gt;&lt;sub&gt;p&lt;/sub&gt;</td>
<td>p&lt;sup&gt;*&lt;/sup&gt;min(ceil&lt;sub&gt;(p&lt;sub&gt;3.12&lt;/sub&gt;)&lt;/sub&gt;,0)</td>
<td></td>
</tr>
<tr>
<td>Decimal Float</td>
<td>p&lt;sup&gt;*&lt;/sup&gt;min(ceil&lt;sub&gt;(p&lt;sub&gt;3.12&lt;/sub&gt;)&lt;/sub&gt;,0)</td>
<td>p&lt;sup&gt;*&lt;/sup&gt;&lt;sub&gt;p&lt;/sub&gt;&lt;sub&gt;min(p,0)&lt;/sub&gt;</td>
<td>p&lt;sup&gt;*&lt;/sup&gt;&lt;sub&gt;p&lt;/sub&gt;&lt;sub&gt;min(p,0)&lt;/sub&gt;</td>
<td>p&lt;sup&gt;*&lt;/sup&gt;&lt;sub&gt;p&lt;/sub&gt;&lt;sub&gt;min(p,0)&lt;/sub&gt;</td>
<td></td>
</tr>
</tbody>
</table>

Each table entry shows the \(<\text{number-of-digits}>\), \(<\text{p}>\), of the converted \(<\text{precision}>\), and the \(<\text{scale-factor}>\), \(<\text{q}>\), of the converted \(<\text{precision}>\) as functions of the \(<\text{number-of-digits}>\), \(<\text{p}>\), and \(<\text{scale-factor}>\), \(<\text{q}>\), of the source \(<\text{precision}>\). These table entries are for cases which never arise in the language definition. \(<\text{N}>\) is the maximum \(<\text{number-of-digits}>\) allowed for the target \(<\text{base}>\) and \(<\text{scale}>\).

In many cases, evaluation of an expression requires conversion of the values of its subexpressions to target \(<\text{data-type}>\) which have \(<\text{arithmetic}>\) and whose \(<\text{base}>\) and \(<\text{scale}>\) are all the same, but whose \(<\text{precision}>\) are determined individually using the rules for converted \(<\text{precision}>\).
Consider the \texttt{infix-expression} \texttt{"1.000 + '10'8"}. The rules for evaluating \texttt{infix-expression}s specify that, before the addition is performed, the values of the two expressions \texttt{1.000} and \texttt{'10'8} are to be converted to the derived common \texttt{class}, \texttt{scale}, and \texttt{mode} of the expressions. According to Section 9.1.3.1, this means conversion to a target \texttt{data-type} that has \texttt{binary}, \texttt{float}, and \texttt{real}. The \texttt{precision} of each target \texttt{data-type} is the converted \texttt{precision}.

For the \texttt{constant} \texttt{"1.000"}, the source \texttt{class} has \texttt{decimal}, and the source \texttt{number-of-digits} is 4. Therefore, according to Table 9.1, the converted \texttt{number-of-digits} is 7. For the \texttt{digit constant} \texttt{'10'8}, the source \texttt{class} has \texttt{binary}, the source \texttt{scale} has \texttt{fixed}, and the \texttt{precision} \texttt{p} in Table 9.1 is taken to be the maximum \texttt{number-of-digits}. Therefore, the converted \texttt{number-of-digits} is \texttt{max(4,7)}, where 7 is the maximum \texttt{number-of-digits} allowed for \texttt{float} and \texttt{binary}.

\textbf{Example 9.2.} An Example of Converted Precision.

\subsection*{9.1.3.3 Derived String Type}

Let \texttt{S} be a set of one or more \texttt{data-type}s which have \texttt{computational-type}s.

The \texttt{derived} \texttt{string-type} of \texttt{S} has \texttt{bit} if all the types in \texttt{S} have \texttt{string-type} \texttt{bit}. Otherwise the \texttt{derived} \texttt{string-type} has \texttt{character}.

When the set \texttt{S} contains at least two \texttt{data-type}s, the terminology \texttt{derived common string-type} will generally be used.

\subsection*{9.1.3.4 Further Definitions for Character and Bit Strings}

In the definitions of operations whose result \texttt{data-type} is either \texttt{character} or \texttt{bit}, the following definitions are used:

\begin{enumerate}
\item The \texttt{length} of a \texttt{character-string-value} or of a \texttt{bit-string-value} is zero if it contains a \texttt{null-character-string} or a \texttt{null-bit-string}; otherwise it is equal to the number of components in its \texttt{character-value-list} or \texttt{bit-value-list}.
\item A \texttt{null-string} is a \texttt{character-string-value} containing a \texttt{null-character-string} when the result \texttt{string-type} has \texttt{character}, or a \texttt{bit-string-value} containing a \texttt{null-bit-string} when the result \texttt{string-type} has \texttt{bit}.
\item A \texttt{string} is a \texttt{character-string-value} when the result \texttt{string-type} has \texttt{character}, or a \texttt{bit-string-value} when the result \texttt{string-type} has \texttt{bit}.
\item A \texttt{substring}, \texttt{st}, of a \texttt{string}, \texttt{st}, is a \texttt{string} of the same type as \texttt{st}. It either contains a null-\texttt{string}, or it contains a contiguous set of \texttt{character-value}s or \texttt{bit-value}s from \texttt{st} in the same order as in \texttt{st}.
\end{enumerate}
9.1.4 ARITHMETIC RESULTS

In the definitions of the evaluation for many infix-expression's and builtin-function-references the final step in determining a scalar-result is to take an arithmetic value computed in some natural way (e.g., by addition of two numbers) and adjust it in a way that represents the special properties of arithmetic in PL/I. This adjustment is made by the operation arithmetic-result, which is defined below. This operation may cause certain conditions to be raised. Note, however, that there are other points in a program execution where the same conditions can be raised.

Operation: arithmetic-result(v,rt)

where v is a <real-value> or <complex-value>,
rt is a <data-type> that has <arithmetic>.
result: a <real-value> or a <complex-value> <as v>.

Let b be the numerical-base of rt, n be the <number-of-digits> of rt, s be the <scale-factor> of rt (if rt has <fixed>), and b be the maximum <number-of-digits> allowed for the <base> and <scale> of rt. If v is a <complex-value>, let x and y be its real and imaginary parts, respectively.

Case 1. rt has <fixed> and <real>.

Determine a <real-value>, v' as follows:

v' = (b^n-n)floor(b^tn), if v ≥ 0,
v' = (b^n-n)ceil(b^tn), if v < 0.

If abs(v') < b^n0, then optionally perform raise-condition(<overflow-condition>); otherwise, return v'.

Case 2. rt has <fixed> and <complex>.

Determine values x' and y' as follows:

x' = (b^n-n)floor(b^n+x), if x ≥ 0,
x' = (b^n-n)ceil(b^n+x), if x < 0,
y' = (b^n-n)floor(b^n+y), if y ≥ 0,
y' = (b^n-n)ceil(b^n+y), if y < 0.

If abs(x') < b^n0 or if abs(y') < b^n0, then optionally perform raise-condition(<overflow-condition>); otherwise, return a <complex-value> x'+iy'.

Case 3. rt has <float> and <real>.

Step 3.1. If this operation was invoked by infix-add, infix-subtract, infix-multiply or infix-divide then perform Step 3.1.1.

Step 3.1.1. Let op1 and op2 be the <real-value> denoted by x' and y' in Step 1 of the operation that invoked this operation. If op1, op2 and v are all integers, and if abs(op1), abs(op2) and abs(v) are less than b^n, then return v.

Step 3.2. Optionally perform raise-condition(<underflow-condition>) and return a <real-value>: 0.

Step 3.3. Optionally perform raise-condition(<overflow-condition>).

Step 3.4. Return an implementation-dependent approximation to v.

Case 4. rt has <float> and <complex>.

Step 4.1. Optionally perform raise-condition(<underflow-condition>), let x' = 0, and go to Step 4.4.

Step 4.2. Optionally perform raise-condition(<overflow-condition>).

Step 4.3. Let x' be an implementation-dependent approximation to x.
Step 4.4. Optionally perform raise-condition(underflow-condition), let $y' = 0$, and go to Step 4.7.

Step 4.5. Optionally perform raise-condition(overflow-condition).

Step 4.6. Let $y'$ be an implementation-dependent approximation to $y$.

Step 4.7. Return a complex-value: $x' + i y'$.

9.1.1 Conditions in Expressions

Certain properties are common to a number of <expression>s that yield arithmetic results. If a condition may occur at the point where a result is normally obtained, then the operation arithmetic-result is invoked. If a condition may occur during the course of a more complicated evaluation, then the operation conditions-in-arithmetic-expression is invoked.

Operation: conditions-in-arithmetic-expression

where $v$ is a <data-type>.

Case 1. $v$ contains <float>.

Step 1.1. Optionally perform raise-condition(underflow-condition).

Step 1.2. Optionally perform raise-condition(overflow-condition).

Case 2. $v$ contains <fixed>.

Optionaliy perform raise-condition(fixed-overflow-condition).

9.1.5 Expressions

Operation: evaluate-expression(e)

where $e$ is an <expression> or <argument>.

result: an <aggregate-value>.

Step 1. If $e$ is an <argument>, let $x$ be the first immediate component of the <expression> immediately contained in $e$; otherwise, let $x$ be the first immediate component of $e$. Let $f$ be the type of $x$ (e.g. prefix-expression). Perform evaluate-f(x) to obtain $v$. Return $v$. (See Section 9.1.6 for the case where $x$ is <float>.)

9.1.6 Value References

Operation: evaluate-value-reference(vr)

where $vr$ is a <value-reference>.

result: an <aggregate-value>.

Case 1. $vr$ immediately contains a <builtin-function-reference>, $x$.

Perform evaluate-builtin-function-reference(x) to obtain $v$. Return $v$.

Case 2. $vr$ immediately contains a <named-constant-reference>, $x$.

Perform evaluate-named-constant-reference(x) to obtain $v$. Return $v$.
Case 3. wr immediately contains a <variable-reference>, x.
   Perform evaluate-variable-reference(x) to obtain a "generation", y. Perform
   value-of-generation() to obtain an <aggregate-value>, v. v must not contain
   undefined). Return v.

Case 4. wr immediately contains a <procedure-function-reference>, x.
   Step 4.1. Perform evaluate-entry-reference(x) to obtain an <evaluated-entry-
             reference>, eer.
   Step 4.2. Perform activate-procedure(eer).
   Step 4.3. Let v be a copy of the <aggregate-value> in the <returned-value> of the
             current <block-state>. Return v.

9.1.7 CONSTANTS

Operation: evaluate-constant(c)

   where c is a <constant>.
   result: an <aggregate-value>.

Step 1. Return an <aggregate-value> containing the <basic-value> in c.

9.1.8 INSUB

Note: There is no operation evaluate-insub. Before an <expression> containing an
      <insub> is evaluated, the <insub> is replaced by an <expression> without <insubs>. See Section 7.6.12.

9.1.9 PARENTHESIZED EXPRESSIONS

Operation: evaluate-parenthesized-expression(e)

   where e is a <parenthesized-expression>.
   result: an <aggregate-value>.

Step 1. Let x be the <expression> immediately contained in e. Perform evaluate-
        expression(x) to obtain v. Return v.

9.1.10 ARGUMENTS

Note: There is no operation evaluate-argument. See evaluate-expression in Section
      9.1.5.
9.2 Prefix Operators

This section describes the prefix operators available in PL/I. Section 9.2.1 gives the general rules for evaluating a <prefix-expression>, and Section 9.2.2 presents the details for each <prefix-operator> in alphabetical order.

9.2.1 PREFIX EXPRESSIONS

Operation: evaluate-prefix-expression

where e is a <prefix-expression>.

result: an <aggregate-value>.

Step 1. Let x be the <expression> immediately contained in e. Let rdd be the <data-description> immediately contained in e. Perform prefix-find(rdd,x) to obtain v, where t is the name of the <prefix-operator> immediately contained in e. Return v.

9.2.2 DEFINITION OF THE PREFIX OPERATORS

The descriptions of the <prefix-operator>s are given in the following sections in alphabetical order.

Under Constraints and Attributes, x denotes the <expression> immediately contained in the <prefix-expression>. Note that all the operations use the macro operation generate-aggregate-result.

9.2.2.1 Prefix-minus

Constraints: Each scalar <data-type> of x must have <computational-type>.

Attributes: Each scalar-result-type has <arithmetic> with the derived class>, <scale>, and <mode> and the converted <precision> of the corresponding scalar <data-type> of x.

The result <aggregate-type> is the same as the <aggregate-type> of e.

Operation: prefix-minus(rdd,x)

Perform generate-aggregate-result.

Step 1. Convert the scalar-value of x to the scalar-result-type to obtain y. The scalar-result is -y.
5.2.2.2 Prefix-not

Constraints: Each scalar <data-type> of x must have <computational-type>.

Attributes: Each scalar-result-type has <bit>.

The result <aggregate-type> is the same as the <aggregate-type> of x.

Operation: `prefix-not(x)`

Perform generate-aggregate-result.

Step 1. Convert the scalar-value of x to <bit> to obtain y.

Step 2. If y contains the <null-bit-string>, then the scalar-result is <null-bit-string-value>. Otherwise the scalar-result is a <bit-string-value> whose length is the same as the length of y. The i'th <bit-value> of the result has <zero-bit> if the i'th <bit-value> of y has <one-bit>; it has <one-bit> if the i'th <bit-value> of y has <zero-bit>.

5.2.2.3 Prefix-plus

Constraints: Each scalar <data-type> of x must have <computational-type>.

Attributes: Each scalar-result-type has <arithmetic> with the derived <base>, <scale>, and <mode> and the converted (precision) of the corresponding scalar <data-type> of x.

The result <aggregate-type> is the same as the <aggregate-type> of x.

Operation: `prefix-plus(x)`

Perform generate-aggregate-result.

Step 1. Convert the scalar-value of x to the scalar-result-type to obtain the scalar-result.
9.3 Infix Operators

This section describes the `<infix-operator>`s available in PL/1. Section 9.3.1 gives the general rules for evaluating an `<infix-expression>`, and Section 9.3.2 presents the details for each `<infix-operator>` in alphabetical order.

9.3.1 INFIX EXPRESSIONS

Operation: \[ \text{evaluate-infix-expression(}e\text{)} \]

where \( e \) is an `<infix-expression>`.

result: an `<aggregate-value>`.

Step 5: Let \( x \) and \( y \) be, in order, the `<expression>`s immediately contained in \( e \). Let \( \text{add}, \text{subtract}, \text{multiply}, \text{divide}, \text{and power} \) be the `<data-description>`s immediately contained in \( e \). Perform `<infix-function(x,y)` to obtain \( v \), where \( f \) is the name of the `<infix-operator>` immediately contained in \( e \). Return \( v \).

9.3.2 DEFINITION OF THE INFIX OPERATORS

The descriptions of the `<infix-operator>`s are given in the following sections in alphabetical order.

Under Constraints and Attributes, \( x \) and \( y \) denote, in order, the `<expression>`s immediately contained in the `<infix-expression>`. Further, for the `<infix-operator>`s `<add>, <subtract>, <multiply>, <divide>, and <power>`, certain local variables are used with special meanings as follows. The local variables \( n, q, \) and \( s \) denote the `<number-of-digits>` components of, respectively, the scalar-result-type, the converted `<precision>` of the corresponding `<data-type>` of \( x \), and the converted `<precision>` of the corresponding scalar `<data-type>` of \( y \). The local variables \( n, q, \) and \( s \) denote the `<scale-factor>` components of, respectively, the scalar-result-type, the converted `<precision>` of the corresponding scalar `<data-type>` of \( x \), and the converted `<precision>` of the corresponding scalar `<data-type>` of \( y \). Unless otherwise stated, the target `<base>` and `<scale>` for determining the `converted <precision>` are the `<base>` and `<scale>` of the scalar-result-type. The letter \( b \) denotes the maximum `<number-of-digits>` allowed for data of the `<base>` and `<scale>` of the scalar-result-type. The local variable \( b \) denotes the numerical-base of the scalar-result-type.

Note that all operations use the macro operation `generate-aggregate-result` (Section 9.1.1.6).
9.3.2.1 Infix-add

Constraints: All scalar <data-type>s of x and y must have <computational-type>s.

The <aggregate-type>s of x and y must be compatible.

Attributes: Each scalar-result-type has <arithmetic> with the derived common <base>,
<scale>, and <mode> of the corresponding scalar <data-type>s of x and y. If the result <scale> has <floats>, then

\[ m = \min(s, \max(p-r, s-n) + \max(q, s) + 1) \]

\[ n = \max(q, s) \]

The result <aggregate-type> is the common <aggregate-type> of x and y.

Operation: `infix-add(x, y)`

Perform generate-aggregate-result.

Step 1. Convert the scalar-values of x and y to target scalar <data-type>s which have
<arithmetic> with the derived common <base>, <scale>, and <mode> of the scalar
<data-type>s of x and y. The target <precision> for the conversion of the scalar-
value of x (respectively, y) is the converted <precision> of the scalar
<data-type> of x (respectively, y). Let x' and y' be the converted values.

Step 2. Perform arithmetic-result(x"+"y, scalar-result-type), to obtain the scalar-
result.

9.3.2.2 Infix-and

Constraints: All scalar <data-type>s of x and y must have <computational-type>s.

The <aggregate-type>s of x and y must be compatible.

Attributes: Each scalar-result-type has <bit>.

The result <aggregate-type> is the common <aggregate-type> of x and y.

Operation: `infix-and(x, y)`

Perform generate-aggregate-result.

Step 1. Convert the scalar-values of x and y to <bit>. Let n be the maximum of the
lengths of the converted values.

Step 2. If the length of one converted value is less than n, convert it to <bit> of
specified length n. Let x' and y' be the final converted values.

Step 3. If n = 0, the scalar-result is a <bit-string-value>: <null-bit-string>
Otherwise the scalar-result is a <bit-string-value> of length n. The i’th <bit-
value> of the result is <one-bit> if the i’th <bit-values> of both x' and y' are
<one-bit>; otherwise the i’th <bit-values> of the result is <zero-bit>.
9.3.2.3 Infix-\text{cat}

Constraints: All scalar \textlangle\text{data-type}\rangle s of \(x\) and \(y\) must have \textlangle\text{computational-type}\rangle.

The \textlangle\text{aggregate-type}\rangle s of \(x\) and \(y\) must be compatible.

Attributes: Each scalar-result-type has \textlangle\text{string-type}\rangle of the derived common \textlangle\text{string-type}\rangle of the corresponding scalar \textlangle\text{data-type}\rangle s of \(x\) and \(y\).

The result \textlangle\text{aggregate-type}\rangle is the common \textlangle\text{aggregate-type}\rangle of \(x\) and \(y\).

Operation: \texttt{infix-\text{cat}(x,y)}

Perform \text{generate-aggregate-result}.

Step 1. Convert the scalar-values of \(x\) and \(y\) to the derived common \textlangle\text{string-type}\rangle of the scalar \textlangle\text{data-type}\rangle s of \(x\) and \(y\). Let the converted values be \(x'\) and \(y'\).

Step 2. Perform \text{concatenate}(x',y') to obtain the scalar-result.

9.3.2.3.1 Concatenation of String Values

Operation: \texttt{concatenate(s1,s2)}

where \(s1\) and \(s2\) have the same \textlangle\text{string-type}\rangle.

Result: a string of the same \textlangle\text{string-type}\rangle as \(s1\) and \(s2\).

Case 1. \(s1\) and \(s2\) are both null-strings.

Return a null-string.

Case 2. (Otherwise).

Return a string containing the \textlangle\text{character-values}\rangle or \textlangle\text{bit-values}\rangle of \(s1\), if any, in order, followed by the \textlangle\text{character-values}\rangle or \textlangle\text{bit-values}\rangle of \(s2\), if any, in order.

9.3.2.4 Infix-\text{divide}

Constraints: All scalar \textlangle\text{data-type}\rangle s of \(x\) and \(y\) must have \textlangle\text{computational-type}\rangle.

The \textlangle\text{aggregate-type}\rangle s of \(x\) and \(y\) must be compatible.

Attributes: Each scalar-result-type has \textlangle\text{arithmetic}\rangle with the derived common \textlangle\text{base}\rangle, \textlangle\text{scale}\rangle, and \textlangle\text{mode}\rangle of the corresponding scalar \textlangle\text{data-type}\rangle s of \(x\) and \(y\). If the result \textlangle\text{scale}\rangle has \textlangle\text{fixed}\rangle, then \(n = \text{max}(p,q)\). If the result \textlangle\text{scale}\rangle has \textlangle\text{floating}\rangle, then:

\[
\begin{align*}
  n &= \text{max}(p,q), \\
  n &= \text{max}(p,q).
\end{align*}
\]

The result \textlangle\text{aggregate-type}\rangle is the common \textlangle\text{aggregate-type}\rangle of \(x\) and \(y\).

Operation: \texttt{infix-\text{divide}(x,y)}

Perform \text{generate-aggregate-result}.

Step 1. Convert the scalar-values of \(x\) and \(y\) to target scalar \textlangle\text{data-type}\rangle s which have \textlangle\text{arithmetic}\rangle with the derived common \textlangle\text{base}\rangle, \textlangle\text{scale}\rangle, and \textlangle\text{mode}\rangle of the scalar \textlangle\text{data-type}\rangle s of \(x\) and \(y\). The target \textlangle\text{precision}\rangle for the conversion of the scalar-value of \(x\) (respectively, \(y\)) is the converted \textlangle\text{precision}\rangle of the scalar \textlangle\text{data-type}\rangle of \(x\) (respectively, \(y\)). Let \(x'\) and \(y'\) be the converted values.

Step 2. If \(y' = 0\), perform \text{raise-condition}(\textlangle\text{zero-division-condition}\rangle); otherwise perform \text{arithmetic-result}(x',y',\textlangle\text{scalar-result}\rangle), to obtain the scalar-result.
4.3.2.5 Infix-eq

Constraints: Corresponding scalar <data-type> of x and y must:

1. Both have <computational-type>, or
2. Both have <locator>, or
3. Have <non-computational-type>, with the immediate subnodes of the <non-computational-type> belonging to the same category other than <locator> or <area>.

Further, if one scalar <data-type> has <offset> and the other has <pointer>, then the <offset> must contain a <variable-reference>.

The <aggregate-types> of x and y must be compatible.

Attributes:
Each scalar-result-type has <bit>.

The result <aggregate-type> is the common <aggregate-type> of x and y.

Operation: \text{infix-eq}(x, y)

Perform generate-aggregate-result.

Step 1. Perform compare\((x', y', t[1], t[2])\) to obtain comp, where \(x'\) is the scalar-value of x, \(y'\) is the scalar-value of y, t[1] is the scalar <data-type> of x, and t[2] is the scalar <data-type> of y.

Step 2. If comp is <true>, the scalar-result is a <bit-string-value> containing <true>. Otherwise the scalar-result is a <bit-string-value> containing <false>.

4.3.2.3.1 Compare

<comparison-results>::=<equal> | <not-equal>

<not-equal>::=<less-than> | <greater-than> |

Operation: \text{compare}(v[1], v[2], t[1], t[2])

where \(v[1]\) is a <basic-value>,
\(v[2]\) is a <basic-value>,
t[1] is a scalar <data-type>,
t[2] is a scalar <data-type>.

result: a subtree of <comparison-results>.

Case 1. At least one of t[1] and t[2] has <arithmetic> or <picture-number>.

Convert \(v[1]\) and \(v[2]\) to target scalar <data-type> which have <arithmetic> with the derived common <base>, <scale>, and <node> of t[1] and t[2]. The <precision> of the target type for \(v[1]\) is the converted <precision> of t[1]. Let \(v[1]'\) and \(v[2]'\) be the converted values.

Case 1.1. \(v[1]' = v[2]'\).
Return <equal>.

Case 1.2. The derived common <node> is <complex>, and \(v[1]' \neq v[2]'\).
Return <not-equal> without any subnode.

Case 1.3. The derived common <node> is <real>, and \(v[1]' < v[2]'\).
Return <not-equal>; <less-than>.
Case 1.4. The derived common <code>good</code> is <code>real</code>, and \( v11' > v12' \).

Return <code>not-equal</code>: <code>greater-than</code>.

Case 2. Each of \( v11 \) and \( v12 \) has <code>string</code> or <code>picture-character</code>.

Convert \( v11 \) and \( v12 \) to the derived common <code>string-type</code> of \( v11 \) and \( v12 \). Let \( n \) be the maximum of the lengths of the converted values. If the length of one converted value is less than \( n \), convert it to the derived common <code>string-type</code> with specified length \( n \). Let \( v11'' \) and \( v12'' \) be the final converted values.

Case 2.1. \( v11'' = v12'' \).

Return <code>equal</code>.

Case 2.2. \( v11'' \) and \( v12'' \) are <code>bit-string-values</code>, and \( i \) is the smallest integer such that the \( i \)th <code>bit-value</code> of \( v11'' \) and \( v12'' \) differ.

If the \( i \)th <code>bit-value</code> of \( v11'' \) is <code>zero-bit</code> while the \( i \)th <code>bit-value</code> of \( v12'' \) is <code>one-bit</code>, return <code>not-equal</code>: <code>less-than</code>. Otherwise return <code>not-equal</code>: <code>greater-than</code>.

Case 2.3. \( v11'' \) and \( v12'' \) are <code>character-string-values</code>, and \( i \) is the smallest integer such that the \( i \)th <code>character-value</code> of \( v11'' \) and \( v12'' \) differ.

If the <code>symbol</code> in the \( i \)th <code>character-value</code> of \( v11'' \) precedes the <code>symbol</code> in the \( i \)th <code>character-value</code> of \( v12'' \) in the result of performing <code>collate-bif</code>, return <code>not-equal</code>: <code>less-than</code>. Otherwise return <code>not-equal</code>: <code>greater-than</code>.

Case 3. \( v11 \) and \( v12 \) both have <code>one-computational-type</code>.

If \( v11 \) has <code>type-string</code> and \( v12 \) has <code>offset</code>, perform convert\( (v11, v12) \) to obtain \( x \), and let \( v12' \) be a <code>basic-value</code>; \( x \), and let \( v11'' = v11 \). Otherwise let \( v11'' = v11 \) and \( v12'' = v12 \).

Case 3.1. \( v11'' \) and \( v12'' \) do not contain <code>pointer-values</code> or <code>offset-values</code>.

If \( v11'' \) and \( v12'' \) are identical, return <code>equal</code>; otherwise return <code>not-equal</code> with no subcase.

Case 3.2. \( v11'' \) and \( v12'' \) contain <code>pointer-values</code>.

Step 3.2.1. Any <code>allocation-unit-designator</code> contained in \( v11'' \) or \( v12'' \) must designate an <code>allocation-unit</code> that exists.

Step 3.2.2. Let \( ed1 \) and \( ed2 \) be the <code>evaluated-data-descriptions</code> of \( v11'' \) and \( v12'' \), respectively.

Step 3.2.3.

Case 3.2.3.1. \( v11'' \) and \( v12'' \) are equal.

Return <code>equal</code>.

Case 3.2.3.2. Either \( v11'' \) or \( v12'' \), but not both, has <code>null</code>.

Return <code>not-equal</code>.

Case 3.2.3.3. The <code>allocation-unit-designator</code>s of \( v11'' \) and \( v12'' \) are different and \( ed1 \) and \( ed2 \) both contain a <code>data-type</code> that has neither a <code>maximal-length</code> with \( 0 \) nor an <code>area-size</code> with \( 0 \).

Return <code>not-equal</code>.
Case 3.2.3.6. (Otherwise).

The <allocation-unit-designator> of \(v111'\) and \(v121'\) must be equal. The first elements, \(s111\) and \(s121\), respectively, of the <storage-index-lists> of \(v111'\) and \(v121'\) must be different. Suppose \(s111 < s121\). Let \(k\) be such that \(s11(k) = s121\). If such a value exists, otherwise let \(k = m1\), where \(m\) is the number of elements in \(s11\). There must exist an \(n\), \(1 \leq k\), such that the <item-data-description>, \(i1d\), obtained by performing find-item-data-description\(d1d, a\), where \(d1d\) is the <data-description> of \(d1d\), has a <data-type> that contains neither a <maximum-length> with 0 nor an <area-size> with 0.

Return <not-equal>.

Case 3.3. \(v111'\) and \(v121'\) contain different-values.

Case 3.3.1. \(v111'\) and \(v121'\) are equal, or differ only in their <occupancy> components.

Return <equal>.

Case 3.3.2. \(v111'\) or \(v121'\), but not both, not <null>.

Return <not-equal>.

Case 3.3.3. (Otherwise).

The <significant-allocation-lists> in \(v111'\) and \(v121'\) each have \(n1\) and \(n2\) components, respectively, and \(n1\) must not equal \(n2\) (say \(n1 < n2\)). Let \(odd111, \ l1, \ldots, \ l11\) and \(odd211, \ l2, \ldots, \ l21\) be respectively the <evaluated-data-descriptions> in the <significant-allocation-lists> of \(v111'\) and \(v121'\). \(odd111\) must equal \(odd211\) for \(i = 1, \ldots, n1\), and for some \(j\), \(n1 < j < n2\), \(odd211\) must have a <data-type> which does not contain a <maximum-length> with 0 nor an <area-size> with 0.

Return <not-equal>.

3.3.2.6 Infix-ge

Constraints: All scalar <data-type>s of \(x\) and \(y\) must have <computational-type>s. Further, each such <data-type> must not have <arithmetic> or <picture-numeric> with <mode>: <complex>.

The <aggregate-type>s of \(x\) and \(y\) must be compatible.

Attributes: Each scalar-result-type has <bit>.

The result <aggregate-type> is the common <aggregate-type> of \(x\) and \(y\).

Operation: \texttt{infix-ge}(x,y)

Perform generate-aggregate-result.

Step 1. Perform \texttt{compare}(x',y',t(1),t(2)) to obtain \(comp\), where \(x'\) is the scalar-value of \(x\), \(y'\) is the scalar-value of \(y\), \(t1\) is the scalar <data-type> of \(x\), and \(t2\) is the scalar <data-type> of \(y\).

Step 2. If \(comp\) is <not-equal>, <greater-than>, or <equal>, the scalar-result is a <bit-string-value> containing <zero-bit>; otherwise the scalar-result is a <bit-string-value> containing <zero-bit>.
6.3.2.7 Infix-ut

Constraints: All scalar <data-type>s of x and y must have <computational-type>. Further, each such <data-type> must not have <arithmetic> or <printed-numeric> with <mode>: <complex>.

The <aggregate-type>s of x and y must be compatible.

Attributes: Each scalar-result-type has <bit>.

The result <aggregate-type> is the common <aggregate-type> of x and y.

Operation: \texttt{infix-ut}(x,y)

Perform generate-aggregate-result.

Step 1. Perform compare(x',y',t[1],t[2]) to obtain comp, where x' is the scalar-value of x, y' is the scalar-value of y, t[1] is the scalar <data-type> of x, and t[2] is the scalar <data-type> of y.

Step 2. If comp is <not-equal>, <greater-than> then the scalar-result is a <bit-string-value> containing <one-bit>; otherwise the scalar-result is a <bit-string-value> containing <zero-bit>.

6.3.2.8 Infix-lt

Constraints: All scalar <data-type>s of x and y must have <computational-type>. Further, each such <data-type> must not have <arithmetic> or <printed-numeric> with <mode>: <complex>.

The <aggregate-type>s of x and y must be compatible.

Attributes: Each scalar-result-type has <bit>.

The result <aggregate-type> is the common <aggregate-type> of x and y.

Operation: \texttt{infix-lt}(x,y)

Perform generate-aggregate-result.

Step 1. Perform compare(x',y',t[1],t[2]) to obtain comp, where x' is the scalar-value of x, y' is the scalar-value of y, t[1] is the scalar <data-type> of x, and t[2] is the scalar <data-type> of y.

Step 2. If comp is <not-equal>, <less-than>, or <equal> then the scalar-result is a <bit-string-value> containing <one-bit>; otherwise the scalar-result is a <bit-string-value> containing <zero-bit>.
3.3.2.9 Infix-\(=\)

Constraints: All scalar \(<\text{data-type}>\)s of \(x\) and \(y\) must have \(<\text{computational-type}>'\). Further, each such \(<\text{data-type}>\) must not have \(<\text{arithmetic}>\) or \(<\text{pictured-numeric}>\) with \(<\text{mode}>\): \(<\text{complex}>\).

The \(<\text{aggregate-type}>\)s of \(x\) and \(y\) must be compatible.

Attributes: Each scalar-result-type has \(<\text{bit}>\).

The result \(<\text{aggregate-type}>\) is the common \(<\text{aggregate-type}>\) of \(x\) and \(y\).

Operation: infix-\(=\)(\(\text{ld}, x, y\))

Perform generate-aggregate-result.

Step 1. Perform compare\(x', y', \text{til1}, \text{til2}\) to obtain \(\text{comp}\), where \(x'\) is the scalar-value of \(x\), \(y'\) is the scalar-value of \(y\), \(\text{til1}\) is the scalar \(<\text{data-type}>\) of \(x\), and \(\text{til2}\) is the scalar \(<\text{data-type}>\) of \(y\).

Step 2. If \(\text{comp}\) is \(<\text{not-equal}>\): \(<\text{less-than}>\); then the scalar-result is a \(<\text{bit-string-value}>\) containing \(<\text{zero-bit}>\); otherwise the scalar-result is a \(<\text{bit-string-value}>\) containing \(<\text{one-bit}>\).

3.3.2.10 Infix-\(+\)

Constraints: All scalar \(<\text{data-type}>\)s of \(x\) and \(y\) must have \(<\text{computational-type}>\).

The \(<\text{aggregate-type}>\)s of \(x\) and \(y\) must be compatible.

Attributes: Each scalar-result-type has \(<\text{arithmetic}>\) with the derived common \(<\text{name}>\), \(<\text{scale}>\), and \(<\text{mode}>\) of the corresponding scalar \(<\text{data-type}>\)s of \(x\) and \(y\). If the result \(<\text{scale}>\) has \(<\text{float}>\), then \(x = \max(p, q)\). If the result \(<\text{scale}>\) has \(<\text{fixed}>\), then

\[
\begin{align*}
    n &= \min(N, p+q) \\
    n' &= a.
\end{align*}
\]

The result \(<\text{aggregate-type}>\) is the common \(<\text{aggregate-type}>\) of \(x\) and \(y\).

Operation: infix-\(+\)(\(\text{ld}, x, y\))

Perform generate-aggregate-result.

Step 1. Convert the scalar-values of \(x\) and \(y\) to target scalar \(<\text{data-type}>\)s which have \(<\text{arithmetic}>\) with the derived common \(<\text{name}>\), \(<\text{scale}>\), and \(<\text{mode}>\) of the scalar \(<\text{data-type}>\)s of \(x\) and \(y\). The target \(<\text{precision}>\) for the conversion of the scalar-value of \(x\) (respectively, \(y\)) is the converted \(<\text{precision}>\) of the scalar \(<\text{data-type}>\) of \(x\) (respectively, \(y\)). Let \(x'\) and \(y'\) be the converted values.

Step 2. Perform \(<\text{arithmetic-result}>\)(\(x' \text{+} y', \text{scalar-result-type}\)), to obtain the scalar-result.
9.3.2.11 Infix-ne

Constraints: Corresponding scalar <data-type> of x and y must:

1. both have `<computational-type>`, or
2. both have `<locator>`, or
3. have `<non-computational-type>`, with the immediate subcodes of the `<non-computational-type>`s belonging to the same category other than `<locator>` or `<area>`.

Further, if one scalar <data-type> has `<offset>` and the other has `<pointer>`, then the `<offset>` must contain a `<variable-reference>`.

The `<aggregate-type>`s of x and y must be compatible.

Attributes: Each scalar-result-type has `<hit>`.

The result `<aggregate-type>` is the common `<aggregate-type>` of x and y.

Operation: `infix-ne(x,y)`

Perform generate-aggregate-result.

Step 1. Perform `compare(x,y,t1(t2))` to obtain comp, where x' is the scalar-value of x, y' is the scalar-value of y, t1 is the scalar <data-type> of x, and t2 is the scalar <data-type> of y.

Step 2. If comp is `<not-equal>`, the scalar-result is a `<bit-string-value>` containing `<one-bit>`; otherwise the scalar-result is a `<bit-string-value>` containing `<zero-bit>`.

9.3.2.12 Infix-or

Constraints: All scalar <data-type> of x and y must have `<computational-type>`.

The `<aggregate-type>`s of x and y must be compatible.

Attributes: Each scalar-result-type has `<hit>`.

The result `<aggregate-type>` is the common `<aggregate-type>` of x and y.

Operation: `infix-or(x,y)`

Perform generate-aggregate-result.

Step 1. Convert the scalar-values of x and y to `<hit>`. Let n be the maximum of the lengths of the converted values.

Step 2. If the length of one converted value is less than n, convert it to `<hit>` of specified length n. Let x' and y' be the final converted values.

Step 3. If n = 0, the scalar-result is a `<bit-string-value>`: `<null-bit-string>`. Otherwise the scalar-result is a `<bit-string-value>` of length n. The i'th `<bit-value>` of the result is `<zero-bit>` if the i'th `<bit-values>` of both x' and y' are `<zero-bit>`; otherwise the i'th `<bit-value>` of the result is `<one-bit>`.
9.3.2.13 Infix-power

Constraints: All scalar <data-type>s of \( x \) and \( y \) must have <computational-type>.

The <aggregate-type>s of \( x \) and \( y \) must be compatible.

Attributes: Each scalar-result-type has <arithmetic>, <base>, <mode>, and <precision> are determined from the corresponding scalar <data-type>s of \( x \) and \( y \) as follows:

The result <aggregate-type> is the common <aggregate-type> of \( x \) and \( y \).

Case 1. The derived <scale> of \( x \) has <fixed>, \( y \) is a <constant> with <mode> <real>, <scale-factor> \( s \); and whose value, \( y' \), is positive: \( |p+1|y' - 1 \) does not exceed the maximum <number-of-digits> for the derived <base> and <scale> of \( x \).

The result <scale> has <fixed>, the result <base> and <mode> has the derived <base> and <mode> of \( x \), and

\[
\begin{align*}
  p &= p + \lceil y' \rceil - 1, \\
  q &= q^{y'}. 
\end{align*}
\]

Case 2. The derived <scale> of \( y \) has <fixed>, the derived <mode> of \( y \) has <real>, \( s \neq 0 \), but Case 1 does not hold.

The result <scale> has <fixed>, the result <base> and <mode> have the derived <base> and <mode> of \( x \), and \( s \neq 0 \).

Case 3. (Otherwise).

The result <scale> has <float>, the result <base> and <mode> have the derived common <base> and <mode> of \( x \) and \( y \), and \( s = \max(p, q) \).

Operation: \texttt{infix-power}(\texttt{pow}, x, y)

Perform generate-aggregate-result.

Step 1. Determine values \( u \) and \( v \) as follows:

Case 1.1. Conditions the same as for Attributes, Case 1.

Let \( u \) and \( v \) be the scalar-values of \( x \) and \( y \), respectively.

Case 1.2. Conditions the same as for Attributes, Case 2.

Convert the scalar-value of \( x \) to the scalar-result-type, and let \( u \) be the converted value. Let \( v \) be the scalar-value of \( y \).

Case 1.3. Conditions the same as for Attributes, Case 3.

Convert the scalar-values of \( x \) and \( y \) to target scalar <data-type>s which have <arithmetic> with <scale> <float>, which have the derived common <base> and <mode> of the scalar <data-type>s of \( x \) and \( y \), and whose <precision>s are the converted <precision>s of the scalar <data-type>s of \( x \) and \( y \), respectively. Let \( u \) and \( v \) be the converted scalar-values of \( x \) and \( y \), respectively.

Step 2. Determine a value \( z \) as follows:

Case 2.1. The result <mode> is <real> and \( u < 0 \).

If the conditions of Attributes, Case 1 or Case 2, hold, then \( z = u \times v \); otherwise, perform raise-condition<error-condition>.

Case 2.2. The result <mode> is <real>, and \( u = 0 \).

If \( v \neq 0 \) then perform raise-condition<error-condition>; otherwise let \( z = 0 \).
Case 2.3. The result <code>\text{mode}</code> is \text{〈real〉}, and \( u > 0 \).

Then:
\[
\begin{align*}
  z &= 1/(\text{int-}v), \text{ if } v < 0, \\
  z &= 1, \text{ if } v = 0, \\
  z &= u/v, \text{ if } v > 0.
\end{align*}
\]

Case 2.4. The result <code>\text{mode}</code> has \text{〈complex〉}.

Interpret both \( u \) and \( v \) as \text{〈complex-number〉}.

Case 2.4.1. \( u = 0 \), the real part of \( v \) is greater than zero, and the imaginary part of \( v \) is zero.

\( z = 0 \).

Case 2.4.2. \( u = 0 \), but the conditions of Case 2.4.1 do not hold.

Perform \text{raise-condition〈error-condition〉}.

Case 2.4.3. \( u \neq 0 \).

\[
  z = u \times (v+\text{log}(w)), \text{ where } \text{log}(w) \text{ is that value of the complex logarithm function whose imaginary part } w \text{ is in the interval } -\pi < w \leq \pi.
\]

Step 3. Perform \text{arithmetic-result〈scalar-result-type〉} to obtain the scalar-result.

9.1.2.13 \text{Infix-subtract}

Constraints: All scalar \text{〈data-type〉}s of \( x \) and \( y \) must have \text{〈computational-type〉}.

The \text{〈aggregate-type〉}s of \( x \) and \( y \) must be compatible.

Attributes: Each scalar-result-type has \text{〈arithmetic〉} with the derived common \text{〈base〉}, \text{〈scale〉}, and \text{〈mode〉} of the corresponding scalar \text{〈data-type〉}s of \( x \) and \( y \). If the result \text{〈scale〉} has \text{〈float〉}, then \( m = \text{max}(p,q) \). If the result \text{〈scale〉} has \text{〈fixed〉}, then \( m = \text{max}(p,q) \).

The result \text{〈aggregate-type〉} is the common \text{〈aggregate-type〉} of \( x \) and \( y \).

Operation: \text{infix-subtract} \text{〈infix〉} \((\text{〈binary〉}, x, y)\)

Perform \text{generate-aggregate-result}.

Step 1. Convert the scalar-values of \( x \) and \( y \) to target scalar \text{〈data-type〉}s which have \text{〈arithmetic〉} with the derived common \text{〈base〉}, \text{〈scale〉}, and \text{〈mode〉} of the scalar \text{〈data-type〉}s of \( x \) and \( y \). The target \text{〈precision〉} for the conversion of the scalar-value of \( x \) (respectively, \( y \)) is the converted \text{〈precision〉} of the scalar \text{〈data-type〉} of \( x \) (respectively, \( y \)). Let \( x' \) and \( y' \) be the converted values.

Step 2. Perform \text{arithmetic-result〈x’–y’〈scalar-result-type〉〉}, to obtain the scalar-result.
9.4 Builtin-Functions

This section describes the <builtin-function> available in PL/I. Section 9.4.1 gives the general rules for evaluating a <builtin-function-reference>, and Section 9.4.2 presents the details for each <builtin-function> in alphabetical order.

In this section an additional heading "Arguments" appears in the description of each <builtin-function>. The number of <argument>-s required by the <builtin-function> and the <argument>-names used in the description are given under this heading. The <argument>-names are separated by commas and the names of optional <argument>-s are enclosed in brackets. An <argument-list> of indeterminate length, e.g. the <argument-list> of the <mku-bif>, is indicated by ellipsis.

Arguments: x, y1, p, q

The letters x, y, p, and q will be used to refer to the <argument>-s in the description of the <builtin-function>. The <argument>-s p and q are optional, i.e. neither need be specified; but if q is specified, then p must also be specified.

Example 9.3. An Example of Optional Arguments.

9.4.1 BUILTIN-FUNCTION REFERENCE

Operation: evaluate-builtin-function-reference(bfr)

where bfr is a <builtin-function-reference>.

result: an <aggregate-value>.

Step 1. Let bfr be the <builtin-function> immediately contained in bfr. Let x11, x12, ... , x15 be the <argument>-s, if any, simply contained in bfr.

Step 2.

Case 2.1. bfr is <collate-bif>, <date-bif>, <empty-bif>, <null-bif>, <aschar-bif>, <asnode-bif>, <asfield-bif>, <asfile-bif>, <askey-bif>, <asnode-bif>, or <time-bif>.

Perform f, where f is the operation whose name is the same as that of bfr, to obtain an <aggregate-value> v. Return v.

Case 2.2. Otherwise.

Let v2 be the <data-description> immediately contained in bfr. Perform f(c00, x[1], x[2], ..., x[15]), where f is the operation whose name is the same as that of bfr, to obtain an <aggregate-value> v. Return v.

9.4.2 SPECIAL TERMS DEFINED FOR BUILTIN-FUNCTIONS

9.4.2.1 Definition of N

Under the heading "Attributes", N is used in describing the <precision> of the result <data-type>. It denotes the maximum <number-of-digits> allowed by the implementation for the result <base> and <scale>.
9.4.7.2 The Arguments p and q

Constraints on p and q

When p and q are used under the heading "Arguments", certain special rules apply. Both p and q must be of the form <argument> <expression> <constant> <data-type> d<sub>type</sub>, where dt has <fixed> and <decimal> and has <scale-factor> equal to zero. The value of p must be greater than zero and less than or equal to 0, i.e. the maximum number of digits for the result <base> and <scale>. q must not occur unless each scalar-result-type has <fixed>.

Attributes of Result Determined by p and q

Case 1. p and q are both specified.

The <number-of-digits> of the result-type is the value of p. The <scale-factor> of the result-type is the value of q.

Case 2. p is specified, q is not specified.

The <number-of-digits> of the result-type is the value of p. If the <scale> of the result-type has <fix>e, then the <scale-factor> of the result-type has 0.

Case 3. Neither p nor q is specified.

The <precision> of the result-type is the converted common <precision> of the <argument>s if there is more than one, and the converted <precision> of the <argument> otherwise.

9.4.3 OPERATIONS USED IN MULTIEX-FUNCTION DEFINITIONS

9.4.3.1 Get-established-onvalue

Operation: get-established-onvalue(tv)

where tv is one of <onchar-value>, <oncode-value>, <onfield-value>, <onfile-value>, <onkey-value>, <onloc-value>, or <onresource-value>.

results: a <character-string-value>, an <integer-value>, or <fail>.

Step 1. Let tv be the current <block-state>.

Step 2.

Case 2.1. tv contains a <condition-bif-value>, cvv, with an immediate component of the type indicated by tv. Let tv be the <integer-value> or <character-string-value> component of cvv.

Return tv.

Case 2.2. (Otherwise).

Case 2.2.1. There is a <block-state> immediately preceding tv in the <block-state-list>.

Let tv be this <block-state>. Go to Step 2.

Case 2.2.2. (Otherwise).

Return <fail>.
9.4.4 Definition of the Builtin-Functions

The descriptions of the builtin-functions are given in the following sections in alphabetical order.

9.4.4.1 abs-bif

Arguments: x

Constraints: All <data-type>s of x must have <computational-type>.

Attributes: The result <aggregate-type> is the <aggregate-type> of x. Each scalar-result-type has <real>. Its <base> is the derived <base> of the corresponding <data-type> of x. The scalar-result-type <scale> and <precision> are determined as follows:

- Case 1. The derived <node> of the corresponding <data-type> of x is <real>.
  - The <scale> and <precision> of the scalar-result-type are the derived <scale> and <precision> of the <data-type>.

- Case 2. The derived <node> of the corresponding <data-type> of x is <complex>.
  - Let r be the converted <number-of-digits> of the corresponding <data-type>, and let s be its converted <scale-factor>. Then the scalar-result-type <number-of-digits> is min(r,s+1), and the scalar-result-type <scale-factor> is r.

Operation: abs-bif(x)

Perform generate-aggregate-result.

Step 1.

- Case 1.1. The derived <node> of the <data-type> of x has <real>.
  - Convert the scalar-value of x to the scalar-result-type. Let y = |x| where x is the converted value.

- Case 1.2. The derived <node> of the <data-type> of x has <complex>.
  - Convert the scalar-value of x to a target type which is the same as the scalar-result-type except that the target <node> is <complex>. Perform condition-in-arithmetic-expression(rt), where rt is the result-type.
  - Determine y, which is the positive square root of (u^2 + v^2), where u and v are, respectively, the real and imaginary parts of the converted value.

Step 2. Perform arithmetic-result(y,rt), where rt is the scalar-result-type, to obtain the scalar-result.
9.4.5.2 Acos-hif

Arguments: x

Constraints: All <data-type>s of x must have <computational-type>. The derived <mode> of the <data-type>s of x must have <real>.

Attributes: The result <aggregate-type> is the <aggregate-type> of x. The <scale> of the scalar-result-type has <float>. The <base>, <mode>, and <precision> of the scalar-result-type are the derived <base>, <mode>, and converted <precision> of the corresponding <data-type> of x.

Operation: acos-hif(rd8,x)

Perform generate-aggregate-result.

Step 1. Convert the scalar-value of x to the scalar-result-type to obtain y.

Step 2. The <data-type> of y has <real>. The value of y must be between -1 and 1, inclusive. Perform conditions-in-arithmetic-expression(x), where x is the scalar-result-type. Let w be the arc cosine of y, in radians, such that 0 ≤ w ≤ π.

Step 3. Perform arithmetic-result(w,r), where r is the scalar-result-type, to obtain the scalar-result.

9.4.5.1 Add-hif

Arguments: x,y,p,q

Constraints: The <aggregate-type>s of x and y must be compatible. All <data-type>s of x and y must have <computational-type>. Constraints on p and q are described in Section 9.4.2.2.

Attributes: The result <aggregate-type> is the common <aggregate-type> of x and y. The <base>, <scale>, and <mode> of the scalar-result-type are the derived common <base>, <scale>, and <mode> of the corresponding <data-type>s of x and y. The <precision> of the scalar-result-type is determined as defined in Section 9.4.2.2.

Operation: add-hif(rd8,x,y,p,q)

Perform generate-aggregate-result.

Step 1. Perform Steps 1.1 and 1.2 in either order.

Step 1.1. Convert the scalar-value of x to the derived common <base>, <scale>, and <mode> of the corresponding <data-type>s of x and y, and the converted <precision> of the <data-type> of x.

Step 1.2. Convert the scalar-value of y to the derived common <base>, <scale>, and <mode> of the corresponding <data-type>s of x and y, and the converted <precision> of the <data-type> of y.

Step 2. Let z be the sum of the converted scalar-value of x and the converted scalar-value of y.

Step 3. Perform arithmetic-result(z,rt), where rt is the scalar-result-type, to obtain the scalar-result.
9.4.4.4 Addr-bif

Arguments: x

Constraints: x must be of the form <argument> | <expression> | <value-reference> | <variable-reference> | y.

Attributes: The result <aggregate-type> immediately contains <scalar>. The result <data-type> has <pointer>.

Operation: addr-bif(x, y)

Step 1. Let dp be the <declaration-designator> immediately contained in y.

Step 2. If the <declaration>-x, designated by dp contains <controlled>, perform find-directory-entry(dp) to obtain the corresponding <controlled>-directory-entry, cde.

Case 2.1. d contains <controlled> and cde does not contain a <generation-list>.

Retain an <aggregate-value> containing <pointer-value>: <null>.

Case 2.2. (Otherwise).

Perform evaluate-variable-reference() to obtain a <generation>-y, which must be connected. Retain an <aggregate-value> containing <pointer-value>: y.

9.4.4.5 After-bif

Arguments: as, cs

Constraints: The <aggregate-type> of as and cs must be compatible. All <data-type>s of as and cs must have <computational-type>.

Attributes: The result <aggregate-type> is the common <aggregate-type> of as and cs. Each scalar-result-type has <string>. The <string-type> is the derived common <string-type> of corresponding <data-type>s of as and cs.

Operation: after-bif(as, cs)

Perform generate-aggregate-result.

Step 1. In either order, convert the scalar-value of as to the scalar-result-type to obtain sb and convert the scalar-value of cs to the scalar-result-type to obtain cb.

Step 2.

Case 2.1. The string sb is a null-string.

The scalar-result is a null-string.

Case 2.2. The string sb is a null-string.

The scalar-result is the string cb.

Case 2.3. The string cb is not a null-string, and cb is not a substring of sb.

The scalar-result is a null-string.

Case 2.4. The string cb is a substring of sb.

Let i denote the position of the last <bit-value> or <character-values> of the leftmost occurrence of cb in sb. Let j denote the position of the last <bit-value> or <character-values> in sb. If i=j, then the scalar-result is a null-string. If i>j then the scalar-result is a string of length j-i whose k'th <bit-value> or <character-values> is the <bit-value> or <character-values> in position (i+k) in the string sb.
3.5.3.6 Allocation-bif

Arguments:  x

Constraints:  x must be of the form <argument> : <expression> : <value-reference> ; <variable-reference> ; y; z.  y must not contain an <identifier-list> or <export-list>.  The <declaration-designator> of y must designate a <declaration> which contains <controlled>.

Attributes:  The result <aggregate-type> immediately contains <scalar>.  The result <data-type> is integer-type.

Operation:  allocation-bif(fd, x)

Step 1.  Let y be the <variable-reference> simply contained in x.  Let dp be the <declaration-designator> immediately contained in y.

Step 2.  Perform find-directory-entry(dp) to obtain the corresponding <controlled-directory-entry>, cde.

Case 2.1.  cde immediately contains a <generation-list>, qe.

Let n be the number of <generation> immediately contained in qe.

Case 2.2. (otherwise).

Let n be 0.

Step 3.  Return an <aggregate-value> containing <real-value> : x.

3.5.3.7 Asin-bif

Arguments:  x

Constraints:  All <data-type>s of x must have <computational-type>.  The derived <mode> of the <data-type>s of x must have <real>.

Attributes:  The result <aggregate-type> is the <aggregate-type> of x.  The <scalar> of the scalar-result-type has <float>.  The <base> <mode>, and <precision> of the scalar-result-type are the derived <base>, <mode>, and converted <precision> of the corresponding <data-type> of x.

Operation: asin-bif(fd, x)

Perform generate-aggregate-result.

Step 1.  Convert the scalar-value of x to the scalar-result-type to obtain y.

Step 2.  The <data-type> of y has <real>.  The value of y must be between -1 and 1, inclusive.  Perform conditions-in-arithmetic-expression(r), where rt is the scalar-result-type.  Let w be the arc sine of y, in radians, such that

-π/2 ≤ w ≤ π/2.

Step 3.  Perform arithmetic-result(w, rt), where rt is the scalar-result-type, to obtain the scalar-result.
The `<aggregate-type>` of `y` and `x` must be compatible. All `<data-type>`s of `y` and `x` must have `<computational-type>`. If the derived `<mode>` of the `<data-type>` of `y` has `<real>` and `x` occurs, the `<data-type>` of `x` must also have `<real>`. If the derived `<mode>` of the `<data-type>` of `y` has `<complex>`, `x` must not be specified.

Attributes: The result `<aggregate-type>` is the common `<aggregate-type>` of `y` and `x`. The scalar-result-type has `<scale>`: `<float>`. The `<base>` and `<mode>` of the scalar-result-type are the derived common `<base>` and `<mode>` of corresponding `<data-type>`s in `y` and `x`. The `<precision>` of the scalar-result-type is the greater of the converted `<precision>` of the corresponding `<data-type>`s of `y` and `x`.

Operation: `atan-bif(rdd, y, x)`

Perform `generate-aggregate-result`.

Step 1.

Case 1.1. `x` is not `<absent>`.

In either order, convert the scalar-value of `y` to the scalar-result-type to obtain `n` and convert the scalar-value of `x` to the scalar-result-type to obtain `v`. The values of `x` and `n` must not both be 0.

Case 1.2. `x` is `<absent>`.

Convert the scalar-value of `y` to the scalar-result-type to obtain `n`. If the scalar-result-type has `<complex>`, `x` must not be 0.

Step 2. Perform `conditions-in-arithmetic-expression(rt)`, where `rt` is the scalar-result-type.

Case 2.1. `x` is not `<absent>`.

Let `v` be the value, in radians, of `arctangent(n/x)` such that

- if `x > 0` then `0 ≤ v ≤ π/2`, and
- if `x < 0` then `−π/2 < v < 0`.

Case 2.2. `x` is `<absent>` and `rt` has `<real>`.

Let `v` be the arc tangent of `n`, such that

- `−π/2 < v < π/2`.

Case 2.3. `rt` has `<complex>`.

Let `v` be the arc tangent of `n`, where the real part of the result, `w`, satisfies

- `−π/2 < w < π/2`.

Step 3. Perform `arithmetic-result(v, rt)` to obtain the scalar-result.
3.6.4.9 Atan-biff

Arguments: \( y, x \)

Constraints: The aggregate-type of \( y \) and \( x \) must be compatible. The derived \( \text{mode} \) of the \( \text{data-type} \) of \( y \) and \( x \), if specified, must have \( \text{real} \).

Attributes: The result aggregate-type is the common aggregate-type of \( y \) and \( x \). The \( \text{mode} \) of the scalar-result-type has \( \text{real} \). The \( \text{scale} \) of the scalar-result-type has \( \text{float} \). The \( \text{base} \) of the scalar-result-type is the derived common \( \text{base} \) of the \( \text{data-type} \) of \( y \) and \( x \). The \( \text{precision} \) of the scalar-result-type is the greater of the converted \( \text{precision} \) of the corresponding \( \text{data-type} \) of \( y \) and \( x \).

Operation: \( \text{atan-biff}(\text{real}, y, x) \)

Perform generate-aggregate-result.

Step 1. Perform \( \text{atan-biff}(y, x) \) to obtain a value, \( v \).

Step 2. Let \( w \) be the value of \( v \) multiplied by 180/\( \pi \).

Step 3. Perform arithmetic-result\((w, rt)\), where \( rt \) is the scalar-result-type, to obtain the scalar-result.

3.6.4.10 Atanh-biff

Arguments: \( x \)

Constraints: All \( \text{data-type} \) of \( x \) must have computational type.

Attributes: The result aggregate-type is the aggregate-type of \( x \). The \( \text{scale} \) of the scalar-result-type has \( \text{float} \). The \( \text{base} \), \( \text{mode} \), and \( \text{precision} \) of the scalar-result-type are the derived \( \text{base} \), \( \text{mode} \), and converted \( \text{precision} \) of the corresponding \( \text{data-type} \) of \( x \).

Operation: \( \text{atanh-biff}(\text{real}, x) \)

Perform generate-aggregate-result.

Step 1. Convert the scalar-value of \( x \) to the scalar-result-type to obtain \( y \). If the scalar-result-type has \( \text{real} \), \( v \) must be less than 1 in absolute value. If the scalar-result-type has \( \text{complex} \), \( y \) must not be 1 or -1.

Step 2. Perform conditions-in-arithmetic-expression\((rt)\), where \( rt \) is the scalar-result-type. Let \( w \) be the arc-hyperbolic tangent of \( y \).

Step 3. Perform arithmetic-result\((w, xt)\), where \( rt \) is the scalar-result-type, to obtain the scalar-result.
3.4.4.11 Before-bif

Arguments: na, ca

Constraints: The <aggregate-type> of na and ca must be compatible. All <data-type>s of na and ca must have <computational-type>.

Attributes: The result <aggregate-type> is the common <aggregate-type> of na and ca. The scalar-result-type has <string>, having the derived common <string-type> of the corresponding <data-type>s of na and ca.

Operation: before-bif(frd, na, ca)

Perform generate-aggregate-result.

Step 1. In either order, convert the scalar-value of na to the scalar-result-type to obtain ab and convert the scalar-value of ca to the scalar-result-type to obtain cb.

Step 2.

Case 2.1. The string ab is a null-string.

The scalar-result is a null-string.

Case 2.2. The string cb is a null-string.

The scalar-result is a null-string.

Case 2.3. The string cb is not a null-string, and it is not a substring of ab.

The scalar-result is the string cb.

Case 2.4. The string cb is a substring of ab.

Let i denote the position of the first <bit-value> or <character-value> of the leftmost occurrence of cb in ab. If i = 1, then the scalar-result is a null-string. If 1 > i, then the scalar-result is a string of length (i-1) whose j'th <bit-value> or <character-value> (1 ≤ j ≤ i-1) is the j'th <bit-value> or <character-value> in the string cb.

3.4.4.12 Binary-bif

Arguments: xl, pf, q11

Constraints: All <data-type>s of x must have <computational-type>. Constraints on p and q are described in Section 9.4.2.2.

Attributes: The result <aggregate-type> is the <aggregate-type> of x. The <base> of the scalar-result-type has <binary>. The <mode> and <scale> of the scalar-result-type are the derived <mode> and <scale> of the corresponding <data-type> of x. The <precision> of the scalar-result-type is determined as defined in Section 9.4.2.2.

Operation: binary-bif(frd, xl, pf, q11)

Perform generate-aggregate-result.

Step 1. Convert the scalar-value of x to the scalar-result-type to obtain v.

Step 2. Perform arithmetic-result(v, rt), where rt is the scalar-result-type, to obtain the scalar-result.
9.4.4.13 Bit-bif

Arguments: x, l, e

Constraints: All <data-type>e of x and l must have <compositional-type>. l must have <aggregate-type> <scalar).

Attributes: The result <aggregate-type> is the <aggregate-type> of x. The scalar-result-type has <bit>.

Operation: bit-bif(r85, x, l, e)

Perform generate-aggregate-result.

Step 1.

Case 1.1. l is <absent>.

Convert the scalar-value of x to <bit> to obtain the scalar-result.

Case 1.2. l is not <absent>.

Convert l to integer-type. The result, n, must not be negative. Convert the scalar-value of x to <bit> of length n to obtain the scalar-result.

9.4.4.14 Bool-bif

Arguments: x, y, c, s

Constraints: The <aggregate-type> of x and y must be compatible. All <data-type>s of x, y, and c must have <compositional-type>. The <aggregate-type> of s must have <scalar>.

Attributes: The result <aggregate-type> is the common <aggregate-type> of x and y. The scalar-result-type has <bit>.

Operation: bool-bif(s85, x, y, c, s)

Perform generate-aggregate-result.

Step 1. In either order, convert the scalar-value of x to <bit> to obtain ra and convert the scalar-value of y to <bit> to obtain rb. Let l be the greatest of the lengths of ra and rb. In either order, convert ra to <bit> of length l to obtain sa, and convert rb to <bit> of length l to obtain sb.

Step 2. Convert s to <bit> of length 8 to obtain d. Let the <bit-values>s within d be named d(0), d(1), d(2), d(3), d(4), from left-to-right, respectively.

Step 3.

Case 3.1. l = 0.

The scalar-result is a null-string.

Case 3.2. l > 0.

The i'th <bit-value> of the scalar-result is set to one of the values d(1), d(2), d(3), d(4) depending on the i'th <bit-values>s of sa and sb as shown in Table 9.7.
Table 9.2: Table of scalar-results as a function of <bit-values> for Bool-bif.

<table>
<thead>
<tr>
<th>i'th &lt;bit-values&gt; of</th>
<th>sa</th>
<th>sb</th>
<th>scalar-result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>d[3]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1</td>
<td>d[3]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0</td>
<td>d[3]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1</td>
<td>d[4]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In this example BOOL is used to perform an exclusive-or operation on the first two arguments. The value "1011"B is the result of the evaluation of:

`BOOL('1100"B,'0101"B,'0110"B)`

Example 9.4: An example of the BOOL bif.

9.3.3.15 Cell-bif

Arguments: x

Constraints: The derived <mode> of the <data-type> of x must have <real>.

Attributes: The result <aggregate-type> is the <aggregate-type> of x. The scalar-result-type has <real>. The scalar-result-type has the derived <base> and <scale> of the corresponding <data-type> of x.

Case 1. The scalar-result-type has <scale>: <fixed>.

The <precision> of the scalar-result-type is the converted <precision> of the <data-type> of x.

Case 2. The scalar-result-type has <scale>: <fixed>.

Let r denote the <number-of-digits> and s denote the <scale-factor> of the converted value of x. The <precision> of the scalar-result-type has:

- <number-of-digits>: min(3, max(r+s-1, 1));
- <scale-factor>: 3.

Operation: cell-bif(real.x)

Perform generate-aggregate-result.

Step 1. Convert the scalar-value of x to its derived <mode>, <base>, <scale>, and converted <precision>.

Step 2. Perform conditions-in-arithmetic-expression(r1), where r1 is the scalar-result-type. Let v be the smallest integer that is greater than or equal to the converted scalar-value of x.

Step 3. Perform arithmetic-result(v,r1), where r1 is the scalar-result-type, to obtain the scalar-result.
3.9.8.16 Character-bif

Arguments: any

Constraints: All <data-type> of sa must have <computational-type>, is must have <aggregate-type>, <scalar>.

Attributes: The result <aggregate-type> is the <aggregate-type> of sa. The scalar-result-type has <character>.

Operation: character-bif(sa) {0, sa}

Perform generate-aggregate-result.

Case 1. Is is <absent>.

Convert the scalar-value of sa to <character> to obtain the scalar-result.

Case 2. Is is not <absent>.

Convert is to integer-type. The result, n, must not be negative. Convert the scalar-value of sa to <character> of length n to obtain the scalar-result.

3.9.8.17 Collate-bif

Arguments: (none)

Attributes: The result <aggregate-type> immediately contains <scalar>. The result <data-type> has <character>.

Operation: collate-bif

Step 1. Let v be a <character-string-value> containing all the terminal nodes of the category symbol, then in an implementation-defined order known as the collating sequence. (See Section 5.1.2.3.1 for use of this sequence in the comparison of <character-string-values>.)

Step 2. Return an <aggregate-value> containing v.
9.4.3.18 Complex-bif

Arguments: x, y

Constraints: The aggregate-type of x and y must be compatible. The derived common mode of the data-type of x and y must have real.

Attributes: The result aggregate-type is the common aggregate-type of x and y. The scalar-result-type has complex. The base and scale of the scalar-result-type are the derived common base and scale of the corresponding data-type of x and y. Let p and q denote the converted number-of-digits and scale-factor of the data-type of x, let r and t denote the converted number-of-digits and scale-factor of the corresponding data-type of y, and let p and q denote the number-of-digits and scale-factor of the scalar-result-type.

Case 1. The scalar-result-type has fixed.

\[ p = \min(p, \max(t - s, t - u) + \max(s, u)) \]
\[ q = \max(s, u) \]

Case 2. The scalar-result-type has float.

\[ p = \max(r, t) \]

Operation: complex-bif(cdd, x, y)

Perform generate-aggregate-result.

Step 1. Let v be a complex-value whose real part is the scalar-value of x converted to the base, scale, and precision of the scalar-result-type and whose imaginary part is the scalar-value of y similarly converted.

Step 2. Perform arithmetic-result(v, rt), where rt is the scalar-result-type, to obtain the scalar-result.

9.4.4.19 Conjugate-bif

Arguments: x

Constraints: All data-type of x must have computational-type.

Attributes: The result aggregate-type is the aggregate-type of x. The scalar-result-type has complex. The scalar-result-type has the derived base, scale, and converted precision of the corresponding data-type of x.

Operation: conjugate-bif(cdd, x)

Perform generate-aggregate-result.

Step 1. Convert the scalar-value of x to the scalar-result-type. Let v be the complex conjugate of the converted scalar-value.

Step 2. Perform arithmetic-result(v, rt), where rt is the scalar-result-type, to obtain the scalar-result.
9.3.5.20 Copy-bif

Arguments: x, n

Constraints: All <data-type>s of x and n must have <computational-type>. x must have <aggregate-type> <scalar>.

Attributes: The result <aggregate-type> in the <aggregate-type> of n. The scalar-result-type has the derived <string-type> of the corresponding <data-type> of n.

Operation: copy-bif(x, n, 1)

Perform generate-aggregate-result.

Step 1. Perform Steps 1.1. and 1.2. in either order.

Step 1.1. Convert n to integer-type. The result, n, must not be negative.

Step 1.2. Convert the scalar-value of x to the scalar-result-type, to obtain sv.

Step 2.

Case 2.1. n > 0.

Let v be a null-string. Perform Step 2.1.1 n times. The scalar-result is v.

Step 2.1.1. Perform concatenation(v, sv) to obtain v.

Case 2.2. n = 0.

The scalar-result is a null-string.

An invocation of

COPY("12", 3)

will return the <character-string> "121212".

Example 9.5. An Example of the COPY bif.

9.3.4.21 Cos-bif

Arguments: x

Constraints: All <data-type>s of x must have <computational-type>.

Attributes: The result <aggregate-type> is the <aggregate-type> of x. The <scale> of the scalar-result-type has <float>. The <base>, <mode>, and <precision> of the scalar-result-type have the derived <base>, <mode>, and converted <precision> of the corresponding <data-type> of x.

Operation: cos-bif(x, x)

Perform generate-aggregate-result.

Step 1. Convert the scalar-value of x to the scalar-result-type. The value of x is assumed to be given in radians.

Step 2. Perform conditions-is-arithmetic-expression(rt), where rt is the scalar-result-type. Let v be the cosine of the converted scalar-value of x.

Step 3. Perform arithmetic-result(v, rt), to obtain the scalar-result.
### 3.4.3.22. cosd-bif

**Arguments:** `x`

**Constraints:** All `<data-type>`s of `x` must have `<computational-type>`. The derived `<mode>` of the `<data-type>` of `x` must have `<real>`.

**Attributes:** The result `<aggregate-type>` is the `<aggregate-type>` of `x`. The `<scale>` of the scalar-result-type has `<float>`. The `<mode>` of the scalar-result-type has `<real>`.

The `<base>` of the scalar-result-type is the derived `<base>` of the corresponding `<data-type>` of `x`. The `<precision>` of the scalar-result-type is the converted `<precision>` of the corresponding `<data-type>` of `x`.

**Operation:** `cosd-bif(r0d, x)`

Perform generate-aggregate-result.

**Step 1:** Convert the scalar-value of `x` to the scalar-result-type. The value of `x` is assumed to be given in degrees.

**Step 2:** Perform `conditions-in-arithmetic-expression(rt3, where rt is the scalar-result-type, Let v be the cosine of the converted scalar-value of x.`

**Step 3:** Perform `arithmetic-result(v, rt)`, to obtain the scalar-result.

### 3.4.3.22. cosh-bif

**Arguments:** `x`

**Constraints:** All `<data-type>`s of `x` must have `<computational-type>`. The `<mode>` of the scalar-result-type has `<float>`. The `<base>`, `<mode>`, and `<precision>` of the scalar-result-type are the derived `<base>`, `<mode>`, and converted `<precision>` of the corresponding `<data-type>` of `x`.

**Operation:** `cosh-bif(r0d, x)`

Perform generate-aggregate-result.

**Step 1:** Convert the scalar-value of `x` to the scalar-result-type to obtain `w`.

**Step 2:** Perform `conditions-in-arithmetic-expression(rt3, where rt is the scalar-result-type, Let v be the hyperbolic cosine of `w`.

**Step 3:** Perform `arithmetic-result(v, rt)` to obtain the scalar-result.

### 3.4.3.28. date-bif

**Arguments:** `none`

**Attributes:** The result `<aggregate-type>` immediately contains `<scalar>`. The result `<data-type>` has `<character>`.

**Operation:** `date-bif`

The result has length 8. It represents the date in the form "yyyy-mm-dd" where:

- each letter represents one `<character-value>` in the result,
- each `<character-value>` is a digit.
- `yy`, `mm`, and `dd` are respectively in the ranges 00:99, 01:12, and 01:31, representing year, month, and day.
### Decat-bif

**Arguments:** sa, ca, pa

**Constraints:** All data-types of sa, ca, and pa must have <computational-type>. pa must have <aggregate-type>: <scalar>.

**Attributes:** The result <aggregate-type> is the common <aggregate-type> of sa and ca. The scalar-result-type has the derived common <string-type> of sa and ca.

**Operation:** `decat-bif(rdd, sa, ca, pa)`

Perform generate-aggregate-result.

**Step 1.** Perform steps 1.1 through 1.3 in any order.

**Step 1.1.** Convert the value of pa to <obj> of length 3 to obtain pb.

**Step 1.2.** Convert the scalar-value of sa to the scalar-result-type to obtain sb.

**Step 1.3.** Convert the scalar-value of ca to the scalar-result-type to obtain sc.

**Step 2.** Divide sb into 3 strings, sb[1], sb[2], sb[3]:

- **Case 2.1.** sb is a null-string.
  - sb[1], sb[2], and sb[3] are null-strings.

- **Case 2.2.** sb is not a null-string and cb is a null-string.

- **Case 2.3.** sb and cb are not null-strings and cb is a substring of sb.
  - sb[1] is the substring of sb before the leftmost occurrence of cb in sb.
  - sb[2] is the string cb and sb[3] is the substring of sb after the leftmost occurrence of cb in sb.

- **Case 2.4.** (Otherwise).

**Step 3.** For i=1,2,3, let tv[i] be sb[i] if the i-th <bit-value> of pb has <open-bits>, otherwise let tv[i] be a null-string. Perform concatenate(tv[1], tv[2]) to obtain v. Perform concatenate(v, tv[3]) to obtain the scalar-result.

---

The value of \text{DECAT}(\text{'XY'}, \text{'Y'}, \text{'E1'\text{\&B}}) is the <character-string-value> "E". Note that this result is the same as the value of \text{AFTER}(\text{'XY'}, \text{'Y'}). The value of \text{DECAT}(\text{'XY'}, \text{'Y'}, \text{'E1'\text{\&B}}) is the <character-string-value> "YE".

---

**Example 9.4.** An example of the \text{DECAT} BIF.
9.8.4.26 Decimal-bif

Arguments: \( x, p, q \)

Constraints: All \(<\text{data-type}>\)s of \( x \) must have \(<\text{computational-type}>\). Constraints on \( p \) and \( q \)
are described in Section 9.8.2.2.

Attributes: The result \(<\text{aggregate-type}>\) is the \(<\text{aggregate-type}>\) of \( x \). The \(<\text{base}>\) of the
scalar-result-type has \(<\text{decimal}>\). The \(<\text{mode}>\) and \(<\text{scale}>\) of the scalar-result-type are the derived \(<\text{mode}>\) and \(<\text{scale}>\) of the corresponding \(<\text{data-type}>\)
of \( x \). The \(<\text{precision}>\) of the scalar-result-type is determined as
defined in Section 9.8.2.2.

Operation: \( \text{Decimal-bif}(\text{rdd},x,p,q) \)
Perform generate-aggregate-result.

Step 1. Convert the scalar-value of \( x \) to the scalar-result-type to obtain \( v \).

Step 2. Perform arithmetic-result\( (v, r) \), where \( r \) is the scalar-result-type, to obtain
the scalar-result.

9.8.4.27 Dimension-bif

Arguments: \( x, n \)

Constraints: The \(<\text{data-type}>\) of \( n \) must have \(<\text{computational-type}>\). \( n \) must have
\(<\text{aggregate-type}>\); \(<\text{scalar}>\); \( x \) must have the form \(<\text{argument}>\); \(<\text{expression}>\); \(<\text{value-reference}>\); \(<\text{variable-reference}>\); \(<\text{data-description}>\); \(<\text{dimensioned-data-description}>\).

Attributes: The result \(<\text{aggregate-type}>\) immediately contains \(<\text{scalar}>\). The result
\(<\text{data-type}>\) is integer-type.

Operation: \( \text{Dimension-bif}(\text{rd}, x, n) \)
Step 1. Let \( y \) be the \(<\text{variable-reference}>\) in \( x \). Let \( dp \) be the \(<\text{declaration-designator}>\) immediately contained in \( y \).

Step 2. Perform evaluate-expression-to-integer\( (y) \) to obtain \( j \). The value of \( j \) must be
positive and not greater than the number of \(<\text{bound-pair}>\)s simply contained in the
\(<\text{data-description}>\) immediate component of \( y \).

Step 3. Perform evaluate-variable-reference\( (y) \) to obtain a \(<\text{variable-reference}>\). Let \( x \) and \( i \)
be the \(<\text{lower-bound}>\) and \(<\text{upper-bound}>\) respectively of the \( j \)'th \(<\text{bound-pair}>\) in
the \(<\text{bound-pair-list}>\) of \( y \)'s \(<\text{evaluated-data-description}>\). Return an
\(<\text{aggregate-value}>\) containing \(<\text{real-value}>\); \((i-k+1)\).

-----------------------------------------------------------------------------------------------------

DECLARE A(10:2:01) CONTROLLED;
ALLOCATE A;
N = DIM(A,2);
-----------------------------------------------------------------------------------------------------

The value of \( S \) is 5 after the execution of the statements shown.

-----------------------------------------------------------------------------------------------------

Example 9.7. An Example of the DIM bif.
3.4.8.20 \texttt{divide-bif}

Arguments: \( x, y, p_1, q_1 \)

Constraints: The \texttt{aggregate-type}s of the \texttt{argument}s \( x \) and \( y \) must be compatible. All \texttt{data-type}s of \( x \) and \( y \) must have \texttt{computational-type}. Constraints for \( p \) and \( q \) are described in Section 3.4.2.2.

Attributes: The result \texttt{aggregate-type} is the common \texttt{aggregate-type} of \( x \) and \( y \). The \texttt{base}, \texttt{scale}, and \texttt{mode} of the scalar-result-type are the derived common \texttt{base}, \texttt{scale}, and \texttt{mode} of the corresponding \texttt{data-type}s in \( x \) and \( y \). The \texttt{precision} of the scalar-result-type is determined as defined in Section 3.4.2.2.

Operation: \texttt{divide-bif(x,y,p_1,q_1)}

Perform \texttt{generate-aggregate-result}.

Step 1. Perform Steps 1.1 and 1.2 in either order.

Step 1.1. Convert the scalar-value of \( x \) to the derived common \texttt{base}, \texttt{scale}, and \texttt{mode} of the corresponding \texttt{data-type}s of \( x \) and \( y \), and the converted \texttt{precision} of the \texttt{data-type} of \( x \).

Step 1.2. Convert the scalar-value of \( y \) to the derived common \texttt{base}, \texttt{scale}, and \texttt{mode} of the corresponding \texttt{data-type}s of \( x \) and \( y \), and the converted \texttt{precision} of the \texttt{data-type} of \( y \).

Step 2. If the converted scalar-value of \( y \) is zero, perform \texttt{raise-condition(zerodivide-condition)}.

Step 3.

Case 3.1. The derived \texttt{mode} of the \texttt{data-type}s of \( x \) and \( y \) has \texttt{real}.

Let \( v \) be the converted scalar-value of \( x \) divided by the converted scalar-value of \( y \).

Case 3.2. At least one of the \texttt{mode}s of the \texttt{data-type}s of \( x \) and \( y \) has \texttt{complex}.

Perform \texttt{conditions-in-arithmetic-expression(r)}, where \( r \) is the scalar-result-type. Let \( v \) be the converted scalar-value of \( x \) divided by the converted scalar-value of \( y \).

Step 6. Perform \texttt{arithmetic-result(r,t)}, where \( r \) is the scalar-result-type, to obtain the scalar-result.
9.4.1.2 Dot-bif

Arguments: x,y1,p1,q1

Constraints: x and y must both be of the form <argument>; <expression>; <data-description>; <dimensioned-data-description>; <element-data-description>; <bound-pair-list>, bpli; ... where bpl contains one component. All <data-type>s of x and y must have <computational-type>. The constraints on p and q are described in section 9.4.2.2. If p is not specified, then the common derived <scale> of x and y must have <float>.

Attributes: The result <aggregate-type> immediately contains <scalar>. The result <base>, <scale>, and <mode> are the derived common <base>, <scale>, and <mode> of the <data-type>s of x and y. If p is not specified they are the derived <scale>; <float>; and the <precision> of the converted <precision>s of x and y. Otherwise the <scale> and <precision> of the result-type are determined as defined in section 9.4.2.2.

Operation: dot-bif(rdd,x,y1,p1,q1)

Step 1. In either order, perform evaluate-expression(x) and evaluate-expression(y) to obtain <aggregate-values>, u and v, each of which has a single <bound-pair> in its <aggregate-type>. The aggregates u and v must have the same number-of-scalar-elements, n.

Step 2. Let u1 and v1 be the <data-type>s with the derived common <base>, <scale>, and <mode> of x and y, and the converted <precision>s of x and y, respectively. In any order, convert the scalar-elements of u to v1 and the scalar-elements of v to v1 to obtain u1,...,u1,...,u1, v1,...,v1,...,v1. Let

\[ w = \sum_{i=1}^{n} u1_i * v1_i. \]

Step 3. Perform conditions-in-arithmetic-expression(r1), where r1 is the <data-type> component of rdd.

Step 4. Perform arithmetic-result(u1,r1), where r1 is the <data-type> component of rdd, to obtain x. Return an <aggregate-value> containing x.

9.4.1.3 Empty-bif

Arguments: (none)

Attributes: The result <aggregate-type> immediately contains <scalar>. The result <data-type> has <area>.

Operation: empty-bif

Step 1. Return the <area-value>: <empty>.
9.4.5.16 Brf-bif

Arguments: x

Constraints: All <data-type>s of x must have <computational-type>. The derived <node> of the <data-type>s of x must have <real>.

Attributes: The result <aggregate-type> is the <aggregate-type> of x. The <scale> of the scalar-result-type has <float>. The <node> of the scalar-result-type is the derived <base> of the corresponding <data-type> of x. The <precision> of the scalar-result-type is the converted <precision> of the corresponding <data-type> of x.

Operation: \text{brf-bif}(\text{r0},x)

Perform \text{generate-aggregate-result}.

Step 1. Convert the scalar-value of x to the scalar-result-type to obtain y.

Step 2. Perform conditions-in-arithmetic-expression(\text{rt}), where \text{rt} is the scalar-result-type. Let w be

\[ \mathcal{F}(\text{2}/\text{pi}) \times \int_{0}^{\infty} e^{-\text{rt} t} \, dt. \]

Step 3. Perform arithmetic-result(w,\text{rt}), where \text{rt} is the scalar-result-type, to obtain the scalar-result.

9.4.5.32 Brfc-bif

Arguments: x

Constraints: All <data-type>s of x must have <computational-type>. The <node> of the <data-type>s of x must have <real>.

Attributes: The result <aggregate-type> is the <aggregate-type> of x. The <scale> of the scalar-result-type has <float>. The <node> of the scalar-result-type has <real>. The <base> of the scalar-result-type is the derived <base> of the corresponding <data-type> of x. The <precision> of the scalar-result-type is the converted <precision> of the corresponding <data-type> of x.

Operation: \text{brf-bif}(\text{r0},x)

Perform \text{generate-aggregate-result}.

Step 1. Convert the scalar-value of x to the scalar-result-type to obtain y.

Step 2. Perform conditions-in-arithmetic-expression(\text{rt}), where \text{rt} is the scalar-result-type. Let w be

\[ 1-(\text{2}/\text{pi}) \times \int_{0}^{\infty} e^{-\text{rt} t} \, dt. \]

Step 3. Perform arithmetic-result(w,\text{rt}), where \text{rt} is the scalar-result-type, to obtain the scalar-result.
9.4.4.3 Every-bif

Arguments: x

Constraints: All <data-type$m of x must have <computational-type>n.

Attributes: The result <aggregate-type>d immediately contains <scalar>. The result <data-type> has <bit>.

Operation: every-bif(\texttt{rd}, x)

Step 1. Perform evaluate-expression(x) to obtain an <aggregate-value>.\texttt{u}.

Step 2. In any order, convert each scalar-element of \texttt{u} to <bit>, to obtain \texttt{v}.

Step 3.

Case 3.1. Every scalar-element of \texttt{v} that does not contain <null-bit-string> has a <bit-string-value> with every <bit-value> containing <one-bit>.

Let \texttt{r} be <one-bit>.

Case 3.2. (Otherwise).

Let \texttt{r} be <zero-bit>.

Step 4. Return an <aggregate-value> containing a <bit-string-value> containing \texttt{r}.

9.4.8.15 Exp-bif

Arguments: x

Constraints: All <data-type>$ of x must have <computational-type>.

Attributes: The result <aggregate-type>d is the <aggregate-type> of x. The <scale> of the scalar-result-type has <float>. The <base>, <mode>, and <precision> of the scalar-result-type are the derived <base>, <mode>, and converted <precision> of the corresponding <data-type> of x.

Operation: exp-bif(\texttt{rd}, x)

Perform generate-aggregate-result.

Step 1. Convert the scalar-value of x to the scalar-result-type to obtain \texttt{w}.

Step 2. Perform conditions-in-arithmetic-expression(rt), where rt is the scalar-result-type. Let v be \texttt{g}, where g is the base of the natural logarithm system.

Step 3. Perform arithmetic-result(v, rt) to obtain the scalar-result.
2.4.4.35 Fixed-bif

Arguments: x, p, q

Constraints: All <data-type> of x must have <computational-type>. Constraints on p and q are given in Section 9.4.3.2.

Attributes: The result <aggregate-type> is the <aggregate-type> of x. The <scale> of the scalar-result-type has <fixed>. The <mode> and <base> of the scalar-result-type are the derived <mode> and <base> of the corresponding <data-type> of x. The precision of the scalar-result-type is determined as defined in Section 9.4.2.2.

Operation: \texttt{fixed-bif(rdd,x,p,q)}

Perform generate-aggregate-result.

Step 1. Convert the scalar-value of x to the scalar-result-type to obtain v.

Step 2. Perform arithmetic-result(v,rt), where rt is the scalar-result-type, to obtain the scalar-result.

2.4.4.36 Float-bif

Arguments: x, p

Constraints: All <data-type> of x must have <computational-type>. Constraints on p are described in Section 9.4.2.2.

Attributes: The result <aggregate-type> is the <aggregate-type> of x. The <scale> of the scalar-result-type has <float>. The <mode> and <base> of the scalar-result-type are the derived <mode> and <base> of the corresponding <data-type> of x. The precision of the scalar-result-type is determined as defined in Section 9.4.2.2.

Operation: \texttt{float-bif(rdd,x,p)}

Perform generate-aggregate-result.

Step 1. Convert the scalar-value of x to the scalar-result-type to obtain v.

Step 2. Perform arithmetic-result(v,rt), where rt is the scalar-result-type, to obtain the scalar-result.
5.4.4.37 floor-bif

Arguments: x

Constraints: The derived <node> of the <data-type>n of x must have <real>.

Attributes: The result <aggregate-type> is the <aggregate-type> of x. The scalar-result-type has <real>. The scalar-result-type has the derived <base> and <scale> of the corresponding <data-type> of x.

Case 1. The scalar-result-type has <scale>: <float>.

The <precision> of the scalar-result-type is the converted <precision> of the <data-type> of x.

Case 2. The scalar-result-type has <scale>: <fixed>.

Let r denote the <number-of-digits> and s denote the <scale-factor> of the converted scalar-value of x. The <precision> of the scalar-result-type is given by:

- <number-of-digits> = \text{min}(r, \text{max}(s+1, 1))
- <scale-factor> = 0

Operation: floor-bif(rdd, x)

Perform generate-aggregate-result.

Step 1. Convert the scalar-value of x to its derived <node>, <base>, <scale>, and converted <precision>.

Step 2. Perform conditions-in-arithmetic-expression, where r is the scalar-result-type. Set v be the greatest integer less than or equal to the converted scalar-value of x.

Step 3. Perform arithmetic-result(v, r), where r is the scalar-result-type, to obtain the scalar-result.

5.4.4.38 round-bif

Arguments: x, n

Constraints: The <data-type> of n must have <compositional-type> n must have <aggregate-type>; <scalar>. x must have the form <argument>; <expression>; <value-reference>; <variable-reference>; <data-description>; <dimensioned-data-description>.

Attributes: The result <aggregate-type> immediately contains <scalar>. The result <data-type> is integer-type.

Operation: round-bif(rdd, x, n)

Step 1. Let y be the <variable-reference> simply contained in x. Let dp be the <declaration-designator> immediately contained in y.

Step 2. Perform evaluate-expression-to-integer to obtain j. The value of j must be positive and not greater than the number of <bound-pair> simply contained in the <data-description> immediate component of y.

Step 3. Perform evaluate-variable-reference(y) to obtain a <generation>, g. Let k be the <upper-bound> of the j"th <bound-pair> in the <bound-pair-list> of g's <evaluated-data-description>. Return an <aggregate-value> containing <real-value>: k.
3.3.4.39 High-bif

Arguments: 1e

Constraints: The <data-type> of 1e must have <computational-type>. 1e must have <aggregate-type> <scalar>.

Attributes: The result <aggregate-type> immediately contains <scalar>. The result <data-type> has <character>.

Operation: high-bif(1e)

Step 1. Perform evaluate-expression-to-integer(1e) to obtain s. The value of s must not be negative.

Step 2.

Case 2.1. s = 0.

Let v be a <character-string-value> of length n containing n occurrences of the last <character-value> in the result of performing collapse-bif.

Case 2.2. s > 0.

Let v be a <character-string-value> <null-character-string>.

Step 3. Return an <aggregate-value> containing v.

3.3.4.40 Imag-bif

Arguments: x

Constraints: All <data-type>s of x must have <computational-type>.

Attributes: The result <aggregate-type> is the <aggregate-type> of x. The scalar-result-type has <real>. The scalar-result-type has the derived <base>, <scale>, and converted <precision> of the corresponding <data-type> of x.

Operation: imag-bif(x)

Perform generate-aggregate-result.

Step 1. Convert the scalar-value of x to <complex> with the <base>, <scale>, and <precision> of the scalar-result-type to obtain w. Let v be the imaginary part of w.

Step 2. Perform arithmetic-result-wrti, where it is the scalar-result-type, to obtain the scalar-result.
3.4.3.41 INDEX-BIF

Arguments: na, ca

Constraints: The <aggregate-type>s of na and ca must be compatible. All <data-type>s of na and ca must have <computational-type>.

Attributes: The result <aggregate-type> is the common <aggregate-type> of na and ca. The scalar-result-type is integer-type.

Operation: INDEX-BIF(rbb, na, ca)

Perform generate-aggregate-result.

Step 1. In <either order>, convert the scalar-values of na and ca to their derived common <string-type>. Let nb be the converted scalar-value of na, and cb be the converted scalar-value of ca.

Step 2.

Case 2.1. cb is not a substring of nb or the length of either nb or cb is 0.

The scalar-result is 0.

Case 2.2. cb is a substring of nb.

Let j denote the position of the first <bit-value> or <character-value> of the leftmost occurrence of cb in nb. The scalar-result is j.

*******************************************************************************

The integer j is the result of evaluating: INDEX("ABCABC", "CD")

*******************************************************************************

Example 9-8 An example of the INDEX-bif.

9.4.4.82 LBOUND-BIF

Arguments: x, i

Constraints: The <data-type> of x must have <computational-type>. i must have <aggregate-type> <scalar>. x must have the form <argument> <expression> <value-reference> <variable-reference> <data-description> <dimensioned-data-description>.

Attributes: The result <aggregate-type> immediately contains <scalar>. The result <data-type> is integer-type.

Operation: LBOUND-BIF(rbb, x, i)

Step 1. Let y be the <variable-reference> simply contained in x. Let dp be the <declaration-designator> immediately contained in y.

Step 2. Perform evaluate-expression-to-integer-ind to obtain j. The value of j must be positive and not greater than the number of <bound-pair>s simply contained in the <data-description> immediate component of y.

Step 3. Perform evaluate-variable-reference(y) to obtain a <generator>, q. Let k be the <lower-bound> of the j'th <bound-pair> in the <bound-pair-list> of q's <evaluated-data-description>. Return an <aggregate-value> containing <real-value> k.
9.4.4.3 `length-hif`

Arguments: `sa`

Constraints: All `<data-type>`s of `sa` must have `<computational-type>`.

Attributes: The result `<aggregate-type>` is the `<aggregate-type>` of `sa`. The scalar-result-type is `<integer-type>`.

Operation: `length-hif(cdd,sa)`

Perform generate-aggregate-result.

Step 1. Convert the scalar-value of `sa` to the derived `<string-type>` of `sa` to obtain `sa`.

Step 2. The scalar-result is the length of `sa`.

9.4.4.4 `lineio-hif`

Arguments: `f`.

Constraints: `f` must have `<aggregate-type>`: `<scalar>`. The `<data-type>` of `f` must have `<file-type>`.

Attributes: The result `<aggregate-type>` immediately contains `<scalar>`. The result `<data-type>` is `<integer-type>`.

Operation: `lineio-hif(cdd,f)`

Step 1. Perform `evaluate-expression(f)` to obtain a `<file-value>`, `fv`. The `<file-information>`, `fi`, designated by `fv` must contain `<open>`. The `<evaluated-file-descriptions>` of `fi` must contain `<print>`.

Step 2. Perform `evaluate-current-line(fv)` to obtain an `<integer-value>`, `iv`. Return an `<aggregate-value>` containing `iv`.

9.4.4.5 `log-hif`

Arguments: `x`

Constraints: All `<data-type>`s of `x` must have `<computational-type>`.

Attributes: The result `<aggregate-type>` is the `<aggregate-type>` of `x`. The `<scale>` of the scalar-result-type has `<float>`-<base>, `<mode>`, and `<precision>` of the scalar-result-type are the derived `<base>`, `<mode>`, and converted `<precision>` of the corresponding `<data-type>` of `x`.

Operation: `log-hif(cdd,x)`

Perform generate-aggregate-result.

Step 1. Convert the scalar-value of `x` to the scalar-result-type to obtain `x`. If the scalar-result-type has `<mode>`: `<real>`; then the value of `x` must be greater than 0. If the scalar-result-type has `<mode>`: `<complex>`; then the value of `x` must not equal 0.

Step 2. Perform `condition-is-arithmetic-expression(rt)`, where `rt` is the scalar-result-type. Let `v` be the natural logarithm of `x`, such that if the scalar-result-type has `<mode>`: `<complex>`, then:

\[-\pi < \text{imaginary part of } v \leq \pi\]

Step 3. Perform `arithmetic-result(v,rt)`, where `rt` is the scalar-result-type, to obtain the scalar-result.
1.4.40 \log_{10}-bif

Arguments: \( x \)

Constraints: The derived <mode> of the <data-type>s of \( x \) must have <real>.

Attributes: The result <aggregate-type> is the <aggregate-type> of \( x \). The scalar-result-
type has <real> and <float>. The <base> and <precision> of the scalar-
result-type are the derived <base> and converted <precision> of the corresponding <data-type> of \( x \).

Operation: \log_{10}-bif(\mathfrak{r}08,x)

Perform generate-aggregate-result.

Step 1. Convert the scalar-value of \( x \) to the scalar-result-type to obtain \( u \). The value of \( u \) must be greater than 0.

Step 2. Perform conditions-in-arithmetic-expression(\mathfrak{r}1), where \( \mathfrak{r}1 \) is the scalar-result-
type. Let \( v \) be the common logarithm, i.e. base 10, of \( u \).

Step 3. Perform arithmetic-result(\mathfrak{r}1, \mathfrak{r}1), where \( \mathfrak{r}1 \) is the scalar-result-type, to obtain the scalar-result.

1.4.47 \log_{2}-bif

Arguments: \( x \)

Constraints: The derived <mode> of the <data-type>s of \( x \) must have <real>.

Attributes: The result <aggregate-type> is the <aggregate-type> of \( x \). The scalar-result-
type has <real> and <float>. The <base> and <precision> of the scalar-
result-type are the derived <base> and converted <precision> of the corresponding <data-type> of \( x \).

Operation: \log_{2}-bif(\mathfrak{r}08,x)

Perform generate-aggregate-result.

Step 1. Convert the scalar-value of \( x \) to the scalar-result-type to obtain \( u \). The value of \( u \) must be greater than 0.

Step 2. Perform conditions-in-arithmetic-expression(\mathfrak{r}1), where \( \mathfrak{r}1 \) is the scalar-result-
type. Let \( v \) be the base 2 logarithm of \( u \).

Step 3. Perform arithmetic-result(\mathfrak{r}1, \mathfrak{r}1), where \( \mathfrak{r}1 \) is the scalar-result-type, to obtain the scalar-result.
9.4.4.48 Low-bif

Arguments: \( x \)

Constraints: The \(<\text{data-type}>\) of \( x \) must have \(<\text{computational-type}>\). \( x \) must have \(<\text{aggregate-type}>\) \(<\text{scalar}>\).

Attributes: The result \(<\text{aggregate-type}>\) immediately contains \(<\text{scalar}>\). The result \(<\text{data-type}>\) has \(<\text{character}>\).

Operation: \( \text{low-bif}(x; \text{int}) \)

Step 1. Perform \( \text{evaluate-expression-to-integer}(x) \) to obtain \( n \). The value of \( n \) must not be negative.

Step 2.

Case 2.1. \( n \neq 0 \).

Let \( v \) be a \(<\text{character-string-value}>\) of length \( n \) containing \( n \) occurrences of the first \(<\text{character-value}>\) in the result of performing \( \text{collate-bif} \).

Case 2.2. \( n = 0 \).

Let \( v \) be a \(<\text{character-string-value}>\): \( \text{null} \).\(<\text{character-string}>\).

Step 3. Return an \(<\text{aggregate-value}>\) containing \( v \).

9.4.4.49 Max-bif

Arguments: \( x[1], x[2], \ldots, x[n] \)

Constraints: \( n \), the number of arguments, must be at least 1. The \(<\text{aggregate-type}>\)s of the \(<\text{argument}>\)s must be compatible. The derived common \(<\text{mode}>\) of the \(<\text{data-type}>\)s of the \(<\text{argument}>\)s must have \(<\text{real}>\).

Attributes: The result \(<\text{aggregate-type}>\) is the common \(<\text{aggregate-type}>\) of the \(<\text{argument}>\)s. The \(<\text{base}>\) and \(<\text{scale}>\) of the scalar-result-type are the derived common \(<\text{base}>\) and \(<\text{scale}>\) of the corresponding \(<\text{data-type}>\)s of the \(<\text{argument}>\)s, and its \(<\text{mode}>\) is \(<\text{real}>\).

Case 1. The derived common \(<\text{scale}>\) of the corresponding \(<\text{data-type}>\)s of the \(<\text{argument}>\)s has \(<\text{fixed}>\).

The \(<\text{precision}>\) of the scalar-result-type is \( \max(p[1], p[2], \ldots, p[n]) \), where \( p[1], p[2], \ldots, p[n] \) are the converted \(<\text{number-of-digits}>\) of the \(<\text{data-type}>\)s of \( x[1], x[2], \ldots, x[n] \).

Case 2. The derived common \(<\text{scale}>\) of the corresponding \(<\text{data-type}>\)s of the \(<\text{argument}>\)s has \(<\text{float}>\).

Let \( s[p[1], q[1]], p[2], q[2], \ldots, p[n], q[n] \) be the converted \(<\text{number-of-digits}>\) and \(<\text{scale-factor}>\) of the \(<\text{data-type}>\)s of \( x[1], x[2], \ldots, x[n] \) respectively. Then the \(<\text{precision}>\) of the scalar-result-type is given by:

\[
\begin{align*}
\text{<number-of-digits> : } & \min(0, \max(p[1]-q[1], p[2]-q[2], \ldots, p[n]-q[n]) + \max(q[1], q[2], \ldots, q[n]) ) \nonumber \\
\text{<scale-factor> : } & \max(q[1], q[2], \ldots, q[n]) \nonumber 
\end{align*}
\]

Operation: \( \text{max-bif}(r, x[1], x[2], \ldots, x[n]) \)

Perform \( \text{generate-aggregate-result} \).

Step 1. In any order, convert the scalar-values of \( x[1], \ldots, x[n] \) to the scalar-result-type. Let the converted scalar-values be \( s[1], s[2], \ldots, s[n] \).

Step 2. Let \( v \) be \( \max(s[1], s[2], \ldots, s[n]) \).

Step 3. Perform \( \text{arithmetic-result}(v, r) \), where \( r \) is the scalar-result-type, to obtain the scalar-result.
Arguments: x11, x2, ..., xin

Constraints: n, the number of arguments, must be at least 1. The aggregate-type of the argument must be compatible. The derived common type of the argument must have <real>.

Attributes: The result aggregate-type is the common aggregate-type of the argument. The <base> and <scale> of the scalar-result-type are the derived common <base> and <scale> of the corresponding data-type of the argument, and its <mode> is <real>.

Case 1. The derived common <scale> of corresponding data-type of the argument is <float>.

The <precision> of the scalar-result-type is max(p11, p2, ..., pni), where p11, p2, ..., pni are the converted number-of-digits of the data-type of x11, x2, ..., xni.

Case 2. The derived common <scale> of corresponding data-type of the argument is <fixed>.

Let q11, q12, ..., q1ni be the converted number-of-digits of <scale-factor> of the data-type of x11, x2, ..., xni respectively. Then the <precision> of the scalar-result-type is given by:

(number-of-digits): min(q11, max(p11, q12, p13, ..., pni), ..., q1ni)
(scale-factor): max(q11, q12, ..., q1ni)

Operation: min-bif(c0, x11, x2, ..., xni)

Perform generate-aggregate-result.

Step 1. In any order, convert the scalar-values of x11, ..., xni to the scalar-result-type. Let the converted scalar-values be u11, u2, ..., un1.

Step 2. Let v be sum(u11, u2, ..., un1).

Step 3. Perform arithmetic-result(v, rt), where rt is the scalar-result-type, to obtain the scalar-result.
Arguments: \( x, y \)

Constraints: The \(<\text{aggregate-type}>\) of \( x \) and \( y \) must be compatible. All derived \(<\text{data-type}>>\) of \( x \) and \( y \) must have \(<\text{real}>>\).

Attributes: The result \(<\text{aggregate-type}>>\) is the common \(<\text{aggregate-type}>>\) of \( x \) and \( y \); the \(<\text{base}>>\) and \(<\text{scale}>>\) of the scalar-result-type are the derived common \(<\text{base}>>\) and \(<\text{scale}>>\) of the corresponding \(<\text{data-type}>>\) of \( x \) and \( y \); and its \(<\text{mode}>>\) is \(<\text{real}>>\).

Case 1. The \(<\text{scale}>>\) of the scalar-result-type has \(<\text{float}>>\).

The \(<\text{precision}>>\) of the scalar-result-type is the greater of the converted \(<\text{precision}>>\) of the corresponding \(<\text{data-type}>>\) of \( x \) and \( y \).

Case 2. The \(<\text{scale}>>\) of the scalar-result-type has \(<\text{fixed}>>\).

Let \((p[11],q[11])\) and \((p[21],q[21])\) be the respective converted \(<\text{number-of-digits}>>\) and \(<\text{scale-factor}>>\) of the corresponding \(<\text{data-type}>>\) of \( x \) and \( y \). Then the \(<\text{precision}>>\) of the scalar-result-type is given by:

\[
\begin{align*}
&\text{<number-of-digits>}: \min(p[11]-q[11]+\max(q[11],q[21])); \\
&\text{<scale-factor>}: \max(q[11],q[21]).
\end{align*}
\]

Operation: \( \text{mod}\cdot\text{biff}(r\cdot\text{op},x,y) \)

Perform \text{generate-aggregate-result}.

Step 1. Perform Steps 1.1 and 1.2 in either order.

Step 1.1. Convert the scalar-value of \( x \) to the derived common \(<\text{base}>, <\text{scale}>, \text{and <mode> of the corresponding <data-type> of x and y, and the converted <precision> of the <data-type> of x.}\)

Step 1.2. Convert the scalar-value of \( y \) to the derived common \(<\text{base}>, <\text{scale}>, \text{and <mode> of the corresponding <data-type> of x and y, and the converted <precision> of the <data-type> of y.}\)

Step 2. Let the converted scalar-value of \( x \) be \( u \) and the converted scalar-value of \( y \) be \( w \).

Case 2.1. \( w \neq 0 \).

Perform \text{conditions-in-arithmetic-expression}(r\cdot\text{op}), where \( r\cdot\text{op} \) is the scalar-result-type. Let \( v \) be given by:

\[
\begin{align*}
v & = u\cdot\text{floor}(v/w).
\end{align*}
\]

Case 2.2. \( w = 0 \).

Let \( v \cdot w \).

Step 3. Perform \text{arithmetic-result}(v, r\cdot\text{op}), where \( r\cdot\text{op} \) is the scalar-result-type, to obtain the scalar-result.
9.4.52 Multiply-Real

Arguments: \( x, y, p[f, q] \)

Constraints: The \( \langle\text{aggregate-type}\rangle \) of \( x \) and \( y \) must be \( \langle\text{computational-type}\rangle \). All \( \langle\text{data-type}\rangle s \) of \( x \) and \( y \) must have \( \langle\text{computational-type}\rangle \). Constraints on \( p \) and \( q \) are given in Section 9.4.2.2.

Attributes: The result \( \langle\text{aggregate-type}\rangle \) is the common \( \langle\text{aggregate-type}\rangle \) of \( x \) and \( y \). The \( \langle\text{base}\rangle \), \( \langle\text{scale}\rangle \), and \( \langle\text{mode}\rangle \) of the scalar-result-type are the derived common \( \langle\text{base}\rangle \), \( \langle\text{scale}\rangle \), and \( \langle\text{mode}\rangle \) of the corresponding \( \langle\text{data-type}\rangle s \) of \( x \) and \( y \). The \( \langle\text{precision}\rangle \) of the scalar-result-type is determined as defined in Section 9.4.2.2.

Operation: \( \text{multiply-real}(x, y, p[f, q]) \)

Perform \( \text{generate-aggregate-result} \).

Step 1. Perform Steps 1.1 and 1.2 in either order.

Step 1.1. Convert the scalar-value of \( x \) to the derived common \( \langle\text{base}\rangle \), \( \langle\text{scale}\rangle \), and \( \langle\text{mode}\rangle \) of the corresponding \( \langle\text{data-type}\rangle s \) of \( x \) and \( y \), and the converted \( \langle\text{precision}\rangle \) of the \( \langle\text{data-type}\rangle \) of \( x \).

Step 1.2. Convert the scalar-value of \( y \) to the derived common \( \langle\text{base}\rangle \), \( \langle\text{scale}\rangle \), and \( \langle\text{mode}\rangle \) of the corresponding \( \langle\text{data-type}\rangle s \) of \( x \) and \( y \), and the converted \( \langle\text{precision}\rangle \) of the \( \langle\text{data-type}\rangle \) of \( y \).

Step 2.

Case 2.1. The scalar-result-type has \( \langle\text{real}\rangle \).

Let \( v \) be the product of the converted scalar-value of \( x \) and the converted scalar-value of \( y \).

Case 2.2. The scalar-result-type has \( \langle\text{complex}\rangle \).

Perform \( \text{condition-in-arithmetic-expression}(r[, t]) \), where \( r \) is the scalar-result-type. Let \( v \) be the product of the converted scalar-value of \( x \) and the converted scalar-value of \( y \).

Step 3. Perform \( \text{arithmetic-result}(v, r[, t]) \), where \( r \) is the scalar-result-type, to obtain the scalar-result.
9.4.4.53 `null-bif`

**Arguments:** (none)

**Attributes:** The result `<aggregate-type>` immediately contains `<scalar>`. The result `<data-type>` has `<pointer>`.

**Operation:** `null-bif`

Step 1. Return an `<aggregate-value>` containing `<pointer-value>`: `null`.

---

9.4.4.54 `offset-bif`

**Arguments:** `pt`, `ar`.

**Constraints:** All `<data-type>`s of `pt` must have `<pointer>`; and must have `<aggregate-type>` `<scalar>`. The `<data-type>` of `ar` must have `<array>`, or must have the form `<argument>`, `<expression>`, `<value-reference>`, `<variable-reference>`.

**Attributes:** The result `<aggregate-type>` is the `<aggregate-type>` of `pt`. The scalar result-type has `offset` with no subtype.

**Operation:** `offset-bif(odd, pt, ar)`

Perform `generate-aggregate-result`.

Step 1. Let `odd` be a `<data-type>` containing `<offset>`: `vr`, where `vr` is the `<variable-reference>` in `ar`.

Step 2. Convert the scalar-value of `pt` to `<data-type>`, `odd` to obtain the scalar-result.

---

9.4.4.55 `uchar-bif`

**Arguments:** (none)

**Attributes:** The result `<aggregate-type>` immediately contains `<scalar>`. The result `<data-type>` has `<character>`.

**Operation:** `uchar-bif`

Step 1. Perform `get-establishing-source-value(uchar-value)` to obtain `j`.

Case 1.1. `j` is `<fail>` or `j=0`.

Let `v` be a `<character-value>` containing `N`.

Case 1.2. `j` > 0.

Perform `get-establishing-source-value(source-value)` to obtain a `<character-string-value>`, `N`. Let `v` be the `j`th `<character-value>` in `N`.

Step 2. Return an `<aggregate-value>` containing `<character-string-value>`, `<character-value-list>`: `v`.

---

Chapter 9: Expressions and Conversion 345
8.4.8 Oncode-bif

Arguments: (none)

Attributes: The result <aggregate-type> immediately contains <scalar>. The result <data-type> is integer-type.

Operation: oncode-bif

Step 1. Perform get-established-name(value(oncode-value)) to obtain j.
   Case 1.1. j is <fail>
      Let v be 0.
   Case 1.2. (Otherwise)
      Let v be j, an implementation-defined value.

Step 2. Return an <aggregate-value> containing <real-value>; v.

8.4.9.57 Onfield-bif

Arguments: (none)

Attributes: The result <aggregate-type> immediately contains <scalar>. The result <data-type> has <character>.

Operation: onfield-bif

Step 1. Perform get-established-name(value(onfield-value)) to obtain xa.
   Case 1.1. xa is <fail>
      Let v be a <character-string-value>; <null-character-string>
   Case 1.2. (Otherwise)
      Let v be xa.

Step 2. Return an <aggregate-value> containing v.

8.4.9.58 Onfile-bif

Arguments: (none)

Attributes: The result <aggregate-type> immediately contains <scalar>. The result <data-type> has <character>.

Operation: onfile-bif

Step 1. Perform get-established-name(value(onfile-value)) to obtain xa.
   Case 1.1. xa is <fail>
      Let v be a <character-string-value>; <null-character-string>
   Case 1.2. (Otherwise)
      Let v be xa.

Step 2. Return an <aggregate-value> containing v.
2.4.4.19 onerror-bif

Arguments: (none)

Attributes: The result <aggregate-type> immediately contains <scalar>. The result <data-type> has <character>.

Operation: onerror-bif

Step 1. Perform get-established-onvalue(onkey-value) to obtain na.
   Case 1.1. na is <fail>.
       Let v be a <character-string-value>: <null-character-string>
   Case 1.2. (Otherwise).
       Let v be na.

Step 2. Return an <aggregate-value> containing v.

2.4.4.20 onloc-bif

Arguments: (none)

Attributes: The result <aggregate-type> immediately contains <scalar>. The result <data-type> has <character>.

Operation: onloc-bif

Step 1. Perform get-established-onvalue(onloc-value) to obtain na.
   Case 1.1. na is <fail>.
       Let v be a <character-string-value>: <null-character-string>
   Case 1.2. (Otherwise).
       Let v be na.

Step 2. Return an <aggregate-value> containing v.

2.4.4.61 onsource-bif

Arguments: (none)

Attributes: The result <aggregate-type> immediately contains <scalar>. The result <data-type> has <character>.

Operation: onsource-bif

Step 1. Perform get-established-onvalue-onsource-value) to obtain na.
   Case 1.1. na is <fail>.
       Let v be a <character-string-value>: <null-character-string>
   Case 1.2. (Otherwise).
       Let v be na.

Step 2. Return an <aggregate-value> containing v.
9.4.4.62 Pageno-bif

Arguments: fn

Constraints: fn must have <aggregate-type>: <scalar>. The <data-type> of fn must have <file>.

Attributes: The result <aggregate-type> immediately contains <scalar>. The result <data-type> is integer-type.

Operation: pageno-bif(rdd, fn)

Step 1. Perform evaluate-expression(f) to obtain a <file-value>, fv. The <file-information>, fi, designated by fv must contain <open>. The <evaluated-file-descriptions> of fi must contain <print>.

Step 2. Let n be the <integer-value> in <page-number> in fi.

Step 3. Return an <aggregate-value> containing <real-value>: n.

9.4.4.63 Pointer-bif

Arguments: ove, ar

Constraints: All <data-type>s of ove must have <offset>, ar must have <aggregate-type>: <scalar>. The <data-type> of ar must have <area>. ar must have the form <argument>: <expression>;<value-reference>;<variable-reference>.

Attributes: The result <aggregate-type> is the <aggregate-type> of ove. The scalar-result-type has <pointer>.

Operation: pointer-bif(rdd, ove, ar)

Perform generate-aggregate-result.

Step 1. Perform Steps 1.1 and 1.2 in either order.

Step 1.1. Perform evaluate-expression(ove) to obtain ove.

Step 1.2. Let ovd be a <data-type> containing <offset>: vr, where vr is the <variable-reference> in ar.

Step 2. Perform convert(pdt, ovd, ove) to obtain the scalar-result, where pdt is a <data-type> containing <pointer>.

9.4.4.66 Precision-bif

Arguments: x, pi, qi

Constraints: Each <data-type> of x must have <computational-type>, Constraints on p and q are given in Section 9.4.2.2.

Attributes: The result <aggregate-type> in the <aggregate-type> of x. The <base>, <scale>, and <mode> of the scalar-result-type are the derived <name>, <scale>, and <mode> of the corresponding <data-type> of x. The <precision> of the scalar-result-type is determined as defined in Section 9.4.7.2.

Operation: precision-bif(rdd, x, pi, qi)

Perform generate-aggregate-result.

Step 1. Convert the scalar-value of x to the scalar-result-type to obtain v.

Step 2. Perform arithmetic-result(v, rt), where rt is the scalar-result-type, to obtain the scalar-result.
Arguments: x

Constraints: x must be of the form <argument>; <expression>; <data-description>; <dimensioned-data-description>; <element-data-description>; <item-data-description>. The <data-type> of x must have <computational-type>.

Attributes: The result <aggregates-type> immediately contains <scalar>. The <base> and <mode> of the result <data-type> are the derived <base> and <mode> of the <data-type> of x.

Case 1. The derived <scale> of the <data-type> of x has <fixed> and the converted <scale-factor> has 0.

  The result <data-type> has <fixed> and <number-of-digits> = n. The result <data-type> has <scale-factor> = 0.

Case 2. (Otherwise).

  The result <data-type> has <float> and its <number-of-digits> is the converted <number-of-digits> of the <data-type> of x.

Operations: prod-bif(rød,x)

Step 1. Perform evaluate-expression(x) to obtain an <aggregate-value>, v.

Step 2. Convert the scalar-elements of v to the result <data-type> in any order. Let w be the product of the converted values.

Step 3. Perform condition-in-arithmetic-expression(w), where w is the <data-type> component of rød.

Step 4. Perform arithmetic-result(v,w), where w is the <data-type> of rød, to obtain x. Return an <aggregate-value> containing x.
9.4.4.66 Real-bif

Arguments: x

Constraints: Each <data-type> of x must have <computational-type>.

Attributes: The result <aggregate-type> is the <aggregate-type> of x. The scalar-result-type has <real>. The <base> and <scale> of the scalar-result-type are the derived <base> and <scale> of the corresponding <data-type> of x. The <precision> of the scalar-result-type is the converted <precision> of the corresponding <data-type> of x.

Operation: real-bif(x)

Perform generate-aggregate-result.

Step 1. Convert the scalar-value of x to <complex> with the <base>, <scale>, and <precision> of the scalar-result-type.

Step 2. Let v be the real part of the converted scalar-value of x.

Step 3. Perform arithmetic-result(v, rt), where rt is the scalar-result-type, to obtain the scalar-result.

9.4.4.67 Reverse-bif

Arguments: as

Constraints: Each <data-type> of as must have <computational-type>.

Attributes: The result <aggregate-type> is the <aggregate-type> of as. The scalar-result-type has the derived <string-type> of the corresponding <data-type> of as.

Operation: reverse-bif(as)

Perform generate-aggregate-result.

Step 1. Convert the scalar-value of as to the scalar-result-type to obtain sb.

Step 2.

Case 2.1. sb is a null-string.

The scalar-result is a null-string.

Case 2.2. sb is not a null-string.

Let m be the length of sb. The scalar-result is a string of length m whose i'th <bit-value> or <character-value> is the (m-i+i)'th <bit-value> or <character-value> in sb.
3.4.4.8 Round-hlf

Arguments: x, n

Constraints: All <data-type> of x must have <computational-type> and integer-type. If the derived <scale> of any <data-type> of x has <float>, the value of n must be greater than zero.

Attributes: The result <aggregate-type> is the <aggregate-type> of x. The <base>, <scale>, and <mode> of the scalar-result-type are the derived <base>, <scale>, and <mode> of the corresponding <data-type> of x.

Case 1. The derived <scale> and <mode> of the <data-type> of x have <real> and <fixed>.
Let y and a be the converted <number-of-digits> of the corresponding <data-type> of x.

The precision of the scalar-result-type is given by:
\<number-of-digits> = \text{max}(\text{min}(y + a, n), 0))
\<scale-factor> = n.

Case 2. The derived <scale> and <mode> of the <data-type> of x have <real> and <float>.
The <number-of-digits> in the scalar result-type is given by max(y, n).

Case 3. The derived <mode> of the <data-type> of x has <complex>.
The precision of the scalar-result-type is determined from the converted <precision> of the real part of the scalar-value of x by either Case 1 or Case 2.

Operation: \text{round-hlf}(\text{rd}, x, n)

Perform generate-aggregate-result.

Step 1. Convert the scalar-value of x to its derived <mode>, <base>, <scale>, and converted <precision>. Let y be the converted value.

Step 2. Let b be the numerical base of the scalar-result.

Case 2.1. The <scale> and <mode> of the <data-type> of y have <real> and <fixed>.
Perform conditions-in-arithmetic-expression(), where ft is the scalar-result-type. Let w be given by:
\(w = \text{sign}(y) \times (b^n) \times \text{floor}(\text{abs}(y) + (b\#n) + 1/2)).

Case 2.2. The <scale> and <mode> of the <data-type> of y have <real> and <float>.
Let c be the unique integer such that
\(\#(c-1) \leq \text{abs}(y) < \#(c).
Perform conditions-in-arithmetic-expression(), where rt is the scalar-result-type. Let w be an implementation-defined value that satisfies the inequality:
\(\text{abs}(w) \leq \text{abs}(y) + 1/2).

Case 2.3. The <mode> of the <data-type> of y has <complex>.
Let u be the real part of v and z be the imaginary part of v. Obtain w by applying Case 2.1 to u and then to z, if the <scale> of v has <fixed>; otherwise obtain the values w by applying Case 2.2 to u and then to z.

Step 3. Perform arithmetic-result(w, rt), where rt is the scalar-result-type, to obtain the scalar-result.
2.4.8.49 \texttt{sign-bit}

Arguments: \( x \)

Constraints: Each <data-type> of \( x \) must have <computational-type>; the derived <mode> must not have \texttt{complex}.

Attributes: The result <aggregate-type> is the <aggregate-type> of \( x \). The scalar-result-type is integer-type.

Operation: \texttt{sign-bit}(\texttt{rdd}, x)

Step 1. Convert the scalar-value of \( x \) to its derived <mode>, <scale>, <base>, and converted <precision> to obtain \( v \).

Step 2.

Case 2.1. \( v > 0 \).

The scalar-result is <real-value>: 1.

Case 2.2. \( v = 0 \).

The scalar-result is <real-value>: 0.

Case 2.3. \( v < 0 \).

The scalar-result is <real-value>: -1.

2.4.8.70 \texttt{sin-bit}

Arguments: \( x \)

Constraints: All <data-type>x of \( x \) must have <computational-type>.

Attributes: The result <aggregate-type> is the <aggregate-type> of \( x \). The <scale> of the scalar-result-type has <float>. The <base>, <mode>, and <precision> of the scalar-result-type are the derived <base>, <mode>, and converted <precision> of the corresponding <data-type> of \( x \).

Operation: \texttt{sin-bit}(\texttt{rdd}, x)

Step 1. Convert the scalar-value of \( x \) to the scalar-result-type. The scalar-value of \( x \) is assumed to be given in radians.

Step 2. Perform \texttt{conditionals-is-arithmetic-expression}(\texttt{rt}), where \( rt \) is the scalar-result-type. Let \( v \) be the sine of the converted scalar-value of \( x \).

Step 3. Perform \texttt{arithmetic-result}(v, rt), where \( rt \) is the scalar-result-type, to obtain the scalar-result.
9.4.4.71 sinh-bif

Arguments: x

Constraints: Each <data-type> of x must have <computational-type>; the derived <mode> must not have <complex>.

Attributes: The result <aggregate-type> is the <aggregate-type> of x. The <scale> of the scalar-result-type has <fixed>. The <mode> of the scalar-result-type has <real>. The <base> and <precision> of the scalar-result-type are the derived <base> and converted <precision> of the corresponding <data-type> of x.

Operation: sinh-bif(real,x)

Perform generate-aggregate-result.

Step 1. Convert the scalar-value of x to the scalar-result-type. The value of x is assumed to be given in degrees.

Step 2. Perform conditions-in-arithmetic-expression(r), where r is the scalar-result-type. Let v be the sine of the converted scalar-value of x.

Step 3. Perform arithmetic-result(v,r), where r is the scalar-result-type, to obtain the scalar-result.

9.4.4.72 sinh-bif

Arguments: x

Constraints: All <data-type>s of x must have <computational-type>.

Attributes: The result <aggregate-type> is the <aggregate-type> of x. The <scale> of the scalar-result-type has <fixed>. The <base>, <mode>, and <precision> of the scalar-result-type are the derived <base>, <mode>, and converted <precision> of the corresponding <data-type> of x.

Operation: sinh-bif(real,x)

Perform generate-aggregate-result.

Step 1. Convert the scalar-value of x to the scalar-result-type to obtain w.

Step 2. Perform conditions-in-arithmetic-expression(r), where r is the scalar-result-type. Let v be the hyperbolic sine of w.

Step 3. Perform arithmetic-result(v,r) to obtain the scalar-result.
9.4.6.31 Some-bit

Arguments: x

Constraints: All <data-type> of x must have <computational-type>.

Attributes: The result <aggregate-type> immediately contains <scalar>. The result <data-type> has <bit>.

Operation: some-bit(rdd, x)

Step 1. Perform evaluate-expression(rdd) to obtain an <aggregate-value>, y.

Step 2. In any order, convert each scalar-element of y to <bit>.

Step 3.

Case 3.1. At least one <bit-value> in some converted scalar-element has <one-bit>.

Let r be <one-bit>.

Case 3.2. (Otherwise).

Let r be <zero-bit>.

Step 4. Return an <aggregate-value> containing a <bit-string-value> containing r.

9.4.6.32 Sort-bit

Arguments: x

Constraints: All <data-type> of x must have <computational-type>.

Attributes: The result <aggregate-type> is the <aggregate-type> of x. The <scale> of the scalar-result-type has <float>. The <base>, <mode>, and <precision> of the scalar-result-type are the derived <base>, <mode>, and converted <precision> of the corresponding <data-type> of x.

Operation: sort-bit(rdd, x)

Perform generate-aggregate-result.

Step 1. Convert the scalar-value of x to the scalar-result-type to obtain y.

Step 2.

Case 2.1. The <data-type> of y has <real>.

The value of y must not be negative. Perform conditions-in-arithmetic-expression(s), where s is the scalar-result-type. Let w be the positive square root of y.

Case 2.2. The <data-type> of y has <complex>.

Perform conditions-in-arithmetic-expression(s), where s is the scalar-result-type. Let w be that square root of y satisfying the following: If the real part of w is u and its imaginary part is v, then either u is greater than zero, or w is zero and v is non-negative.

Step 3. Perform arithmetic-result(w, rt), where rt is the scalar-result-type, to obtain the scalar-result.
9.2.4.75 String-bif

Arguments: ss

Constraints: All <data-type>s of ss must have <computational-type>.

Attributes: The result <aggregate-type> immediately contains <scalar>. The result <data-type> has the derived common <string-type> of the <data-type>s of ss.

Operation: string-bif(rs, ss)

Step 1. Perform eval-funct-expression(s) to obtain an <aggregate-value>, v.

Step 2. In any order, convert all the scalar-elements of v to the derived common <string-type> of all the <data-type>s of ss. Let the converted values be cvill in the order of the scalar-elements of v.

Step 3. let r be a null-string. For i=1 to number-of-scalar-elements of v, perform step 3.1.

Step 3.1. Perform concat-str(cvill) to obtain r.

Step 4. Return an <aggregate-value> containing r.

9.2.4.76 Substr-bif

Arguments: ss, stt, lel

Constraints: All <data-type>s of ss, st and le must have <computational-type>. The <aggregate-type>s of ss, st and le must be compatible.

Attributes: The result <aggregate-type> is the common <aggregate-type> of ss, st and le. Each scalar-result-type has <string> with the derived <string-type> of the corresponding <data-type> of ss.

Operation: substr-bif(rs, ss, stt, lel)

Perform generate-aggregate-result.

Step 1. Perform steps 1.1 to 1.3 in any order.

Step 1.1. Convert the scalar-value of ss to the scalar-result-type to obtain sb. Let the length of sb be k.

Step 1.2. Convert the scalar-value of st to integer-type to obtain l.

Step 1.3. If le is not <scalar>, convert its scalar-value to integer-type to obtain j.

Step 2. If le is <absent>, let j=k=l+1.

Step 3. Test the inequalities:

\[ 1 \leq i \leq k + 1 \]
\[ 0 \leq j \leq k - 1 + 1. \]

If these inequalities are not satisfied, perform raise-condition("string-range-condition").

Step 4.

Case 4.1. j = 0.

The scalar-result is a null-string.

Case 4.2. j > 0.

The scalar-result is a string of length j whose n'th <bit-value> or <character-value> (lsb(j)) is the (i+j-1)th <bit-value> or <character-value> in the string sb, n=1,...,j.
3.5.4.78 Subtract-bif

Arguments: x, y, p, q

Constraints: The aggregate-types of x and y must be compatible. Each data-type of x and y must have a computational-type. Constraints on p and q are described in Section 9.4.2.2.

Attributes: The result aggregate-type is the common aggregate-type of x and y. The base, scale, and mode of the scalar-result-type are the derived common base, scale, and mode of the corresponding data-type of x and y. The precision of the scalar-result-type is determined as defined in Section 9.4.2.2.

Operation: subtract-bif(rdd, x, y, p, q)

Perform generate-aggregate-result.

Step 1. Perform Steps 1.1 and 1.2 in either order.

Step 1.1. Convert the value of x to the derived common base, scale, and mode of the corresponding data-type of x and y, and the converted precision of the data-type of x.

Step 1.2. Convert the value of y to the derived common base, scale, and mode of the corresponding data-type of x and y, and the converted precision of the data-type of y.

Step 2. Let v be the difference between the converted scalar-value of x and the converted scalar-value of y.

Step 3. Perform arithmetic-result(v, rt), where rt is the scalar-result-type, to obtain the scalar-result.

3.5.4.78 Sum-bif

Arguments: x

Constraints: x must be of the form argument; expression; data-description; dimensioned-data-description; element-data-description; item-data-description. Its data-type must have a computational-type.

Attributes: The result aggregate-type immediately contains scalar. The base, scale, and mode of the result data-type are the derived base, scale, and mode of the data-type of x.

Case 1. The derived scale of the data-type of x has fixed.

Let r be the converted scale-factor of the data-type of x. Then the precision of the result data-type has number-of-digits \( r \) and scale-factor \( r \).

Case 2. The derived scale of the data-type of x has float.

The precision of the result data-type has the converted precision of the data-type of x.

Operation: sum-bif(rdd, x)

Step 1. Perform evaluate-expression to obtain an aggregate-values, v.

Step 2. In any order, convert the scalar-elements of v to the result data-type. Let w be the sum of the converted values.

Step 3. Perform conditions-in-arithmetic-expression(rt), where rt is the data-type component of rdd.

Step 4. Perform arithmetic-result(w, rt), where rt is the data-type component of rdd, to obtain rt. Return an aggregate-values containing rt.
2.4.3.79 Tan-bif

Arguments: x

Constraints: All <data-type> of x must have <computational-type>.

Attributes: The result <aggregate-type> is the <aggregate-type> of x. The <scale> of the scalar-result-type has <float>. The <base>, <mode>, and <precision> of the scalar-result-type are the derived <base>, <mode>, and converted <precision> of the corresponding <data-type> of x.

Operation: tan-bif(x, x)

Perform generate-aggregate-result.

Step 1. Convert the scalar-value of x to the scalar-result-type. Its value is assumed to be given in radians, and must not be an odd multiple of pi/2.

Step 2. Perform conditions-in-arithmetic-expression(ri), where ri is the scalar-result-type. Let v be the tangent of the converted scalar-value of x.

Step 3. Perform arithmetic-result(v, rt), where rt is the scalar-result-type, to obtain the scalar-result.

2.4.3.80 Tand-bif

Arguments: x

Constraints: Each <data-type> of x must have <computational-type>; the derived <mode> must not have <complex>.

Attributes: The result <aggregate-type> is the <aggregate-type> of x. The <scale> of the scalar-result-type has <float>. The <mode> of the scalar-result-type has <real>. The <base> and <precision> of the scalar-result-type are the derived <base> and converted <precision> of the corresponding <data-type> of x.

Operation: tand-bif(x, x)

Perform generate-aggregate-result.

Step 1. Convert the scalar-value of x to the scalar-result-type. The converted scalar-value is assumed to be given in degrees, and must not be an odd multiple of 90 degrees.

Step 2. Perform conditions-in-arithmetic-expression(ri), where ri is the scalar-result-type. Let v be the tangent of the converted scalar-value of x.

Step 3. Perform arithmetic-result(v, rt), where rt is the scalar-result-type, to obtain the scalar-result.
9.4.4.51 tanh-bif

Arguments: x

Constraints: All <data-type>s of x must have <computational-type>.

Attributes: The result <aggregate-type> is the <aggregate-type> of x. The <scale> of the scalar-result-type has <float>. The <case>, <mode>, and <precision> of the scalar-result-type are the derived <case>, <mode>, and converted <precision> of the corresponding <data-type> of x.

Operation: tanh-bif(x04,x)

Perform generate-aggregate-result.

Step 1. Convert the scalar-value of x to the scalar-result-type to obtain w.

Step 2. Perform conditions-in-arithmetic-expression(rtl), where r is the scalar-result-type. Let v be the hyperbolic tangent of w.

Step 3. Perform arithmetic-result(rtl) to obtain the scalar-result.

9.4.4.52 time-bif

Arguments: (none)

Attributes: The result <aggregate-type> immediately contains <scalar>. The result <data-type> has <character>.

Operation: time-bif

Step 1. Let v be a <character-string-value> of length 6+4, where t is an implementation-defined non-negative integer. v represents the time-of-day on a 24-hour scale, in the form "HH:mm:ss", where each letter represents one <character-value> and each contained (symbol) is a digit. HH, mm, ss are respectively in the ranges 00:23, 00:59, and 00:59, representing hours, minutes, and seconds. d..., represents decimal fractions of a second.

Step 2. Return an <aggregate-value> containing v.
9.3.1.1 Translate-bif

Arguments: \( sa, ra, pa \)

Constraints: The \(<aggregate-type>\) of \( sa, ra \) and \( pa \) must be compatible. All \(<data-type>\)'s of \( sa, ra \) and \( pa \) must have \(<computational-type>\).

Attributes: The result \(<aggregate-type>\) is the common \(<aggregate-type>\) of \( sa, ra \) and \( pa \). The scalar-result-type has \(<character>\).

Operation: \( \text{translate-bif}(sa, ra, pa) \)

Perform \( \text{generate-aggregate-result} \).

Step 1. Perform Steps 1.1 and 1.2 in either order.

Step 1.1. Convert the scalar-value of \( sa \) to \(<character>\) to obtain \( sb \). Let \( n \) be the length of \( sb \).

Step 1.2.

Case 1.2.1. \( pa \) is not \(<absent>\).

Convert the scalar-value of \( pa \) to \(<character>\) to obtain \( pb \).

Case 1.2.2. \( pa \) is \(<absent>\).

Perform \( \text{collate-bif} \) to obtain \( pb \).

Step 2. Convert the scalar-value of \( ra \) to \(<character>\). If the length of this converted scalar-value is less than the length of \( pb \), then convert it to a \(<character>\) string of the length of \( pb \). Let \( rb \) denote the converted scalar-value of \( ra \).

Step 3.

Case 3.1. The string \( sb \) is a null-string.

The scalar-result is a null-string.

Case 3.2. The string \( sb \) is not a null-string.

For each \( i \), \( 1 \leq i \leq n \), let \( sli \) be the \( i \)'th \(<character-value>\) in \( sb \) and let \( rji \) be the \( j \)'th \(<character-value>\) in the scalar-result. The string \( pb \) is searched for the leftmost occurrence of \( sli \). If \( sli \) is not found in \( pb \), then let \( j \) be \( n + 1 \). If \( sli \) occurs in \( pb \), let \( j \) be the ordinal of the occurrence in \( pb \) and let \( rji \) be the \( j \)'th \(<character-value>\) in the string \( rb \). The scalar-result is \( rj \).

The result of evaluating:

\[ \text{TRANSLATE}("1 2", "0", ") \]

is the \(<character-string-value>\) '102'.

Example 9.9. An Example of the TRANSLATE bif.
9.4.4.84 Trunc-bif

Arguments: x

Constraints: The derived <mode> of the <data-type> of x must have <real>.

Attributes: The result <aggregate-type> is the <aggregate-type> of x. The <base> and <scale> of the scalar-result-type are the derived <base> and <scale> of the corresponding <data-type> of x.

Case 1. The derived <scale> of the corresponding <data-type> of x has <fixed>.

Let r be the converted <number-of-digits> and s be the converted <scale-factor> of the <data-type> of x. Then the <precision> of the scalar-result-type is given by:

<number-of-digits> = \min(8, \max(r+1, 1));
<scale-factor> = 0.

Case 2. The derived <scale> of the corresponding <data-type> of x has <float>.

The <precision> of the scalar-result-type is the converted <precision> of the corresponding <data-type> of x.

Operation: trunc-bif(rdd, x)

Perform generate-aggregate-result.

Step 1. Convert the scalar-value of x to the scalar-result-type to obtain v. Perform
conditions-in-arithmetic-expression(r), where r is the scalar-result-type.

Step 2.

Case 2.1. v < 0.

Let w = ceil(v).

Case 2.2. v = 0.

Let w = 0.

Case 2.3. v > 0.

Let w = floor(v).

Step 3. Perform arithmetic-result(w, r), where r is the scalar-result-type, to obtain
the scalar-result.

9.4.4.85 Unspec-bif

Arguments: x


Attributes: The result <aggregate-type> immediately contains <scalar>. The result <data-type> has <bit>.

Operation: unspec-bif(rdd, x)

Step 1. Let y be the <variable-reference> simply contained in x.

Step 2. Perform evaluate-variable-reference(y) to obtain a <generation>, q.

Step 3. Return an <aggregate-value> containing a <bit-string-value>, v, which depends on the properties of q in an implementation-defined manner. This value may be <undefined>.
9.4.4.66 Valid-bif

Arguments: aa

Constraints: All <data-type>s of aa must have <picture>.

Attributes: The result <aggregate-type> is the <aggregate-type> of aa. The scalar-result-type has <bit>.

Operation: valid-bif(aa)

Perform generate-aggregate-result.

Step 1. Let sv be the scalar-value of aa; let dt be the scalar <data-type> corresponding to aa; Perform validate-numeric-picture-value(dt,sv) or validate-character-picture-value(dt,sv) according as dt has <numeric-picture> or <character-picture>, to obtain a <picture-validity>, pv.

Step 2.
Case 2.1. pv has <picture-valid>.

The scalar-result is a <bit-string-value> <bit-value-list> <bit-value> <zero-bit>.

Case 2.2. pv has <picture-invalid>.

The scalar-result is a <bit-string-value> <bit-value-list> <bit-value> <zero-bit>.

9.4.4.67 Verify-bif

Arguments: ss, ca

Constraints: The <aggregate-type>s of ss and ca must be compatible. All <data-type>s of ss and ca must have <computational-type>.

Attributes: The result <aggregate-type> is the common <aggregate-type> of ss and ca. The scalar-result-type is integer-type.

Operation: verify-bif(ca, ss, ca)

Perform generate-aggregate-result.

Step 1. In either order, convert the scalar-value of ss to <character> to obtain sb and convert the scalar-value of ca to <character> to obtain cb. Let n be the length of sb.

Step 2. Each <character-value> of sb, sb[i], (0 ≤ i < n), is compared in turn with the <character-values> of cb.

Case 2.1. The string sb has a <null-character-string>.

The scalar-result is 0.

Case 2.2. The value of sb[i] occurs in cb for all values of i.

The scalar-result is 0.

Case 2.3. (Otherwise).

The scalar-result is j, where the j'th <character-value> of sb is the leftmost <character-value> of sb that does not occur in cb.
VERIFY('297b', '0123456789')

will return the value 4, i.e., the position of the first non-numeric character in the string '297b'.

Example 9.10. An Example of the VERIFY bif.
9.5 Conversion

9.5.1 Conversion of Scalar Values

The operation convert is used to convert a basic-value, sv, into a new basic-value consistent with the target <data-type>, tt. The operand, sv, is a <data-type> associated with sv; for example, it may be the <data-type> of an <expression> whose evaluation yielded sv.

One of the following three relationships always holds for the operands:

1. The <data-type>, tt and st, have <computational-type> and sv is a <real-value>, <complex-value>, <character-string-value>, or <bit-string-value> that does not contain <undefined>.

2. The <data-type>, tt has <offset>, st has <pointer>, and sv is a <pointer-value>.

3. The <data-type>, tt has <pointer>, st has <offset>, and sv is an <offset-value>.

In (2) and (3), the <offset> contains a <variable-reference>.

Note that convert may be invoked informally (see Section 9.3.1.1).

Operation: \( convert(tt, st, sv) \)

where tt is a <data-type>,

st is a <data-type>,

sv is a <real-value>, sv,

or a <complex-value>, sv,

or a <character-string-value>, sv,

or a <bit-string-value>, sv,

or a <pointer-value>, sv,

or a <offset-value>, sv,

or a <basic-value> containing an immediate component, sv,

or an <aggregate-value> containing a <basic-value> with

such an immediate component, sv,

result: a <real-value>, or a <complex-value>, or a <character-string-value>,

or a <bit-string-value>, or a <pointer-value>, or an <offset-value>.

Case 1. tt has <real> and <float> but not <pictured-numeric>, and sv is a <real-value>.

Perform convert-to-float(tt, st, sv) to obtain cv. Return cv.

Case 2. tt has <real> and <float> not <pictured-numeric>, and sv is a <real-value>.

Perform convert-to-float(tt, st, sv) to obtain cv. Return cv.

Case 3. tt has <complex> but not <pictured-numeric>, and sv is a <real-value>.

Let rt be a <data-type> with <mode> <real> but which is otherwise the same as tt. Perform convert(tt, st, sv) to obtain x. Return a <complex-value> whose real part is x and whose imaginary part is 0.

Case 4. tt has <complex> but not <pictured-numeric> and sv is a <complex-value>.

Let rt and rtt be <data-type> whose <mode>s have <real> but which are otherwise the same as tt and st. Let x and y be the real and imaginary parts of sv. In any order perform convert(rtt, rtt, sv) to obtain x' and convert(rtt, rtt, y) to obtain y'.

Return a <complex-value> whose real part is x' and whose imaginary part is y'.

Case 5. tt has <real> but not <pictured-numeric> and sv is a <complex-value>.

Let x be the real part of sv; let rtt be a <data-type> with <mode> <real> but otherwise the same as st. Perform convert(tt, rtt, x) to obtain cv. Return cv.
Case 6. tt is <arithmetic> but not <pictured-numeric>, and sv is a <character-string-value> or a <bit-string-value>.

Step 6.1. Perform convert-to-arithmeticlist(sv) to obtain a <value-and-type>: rv rdt; or a <value-and-type>: rv rdt lv ld t.

Step 6.2.

Case 6.2.1. tt has <real>.

Perform convert(rtt, rdt, rv) to obtain v. Return v.

Case 6.2.2. tt has <complex>.

If lv and ld t do not exist, let lv be a <real-value>: 0; and ld t be rdt.
Let rtt be a <data-type> which has <real> but is otherwise as tt.
Perform convert(rtt, rdt, rv) to obtain v1; perform convert(rtt, ld t, lv) to obtain v2. Return a <complex-value> whose real part is v1 and imaginary part is v2.

Case 7. tt is <pictured-numeric> and sv is a <real-value>, <complex-value>, <bit-string-value>, or <character-string-value>.

Step 7.1. If tt and st are identical, optionally return sv.

Step 7.2. Let att be the associated arithmetic data-type (see Section 9.5.23 of tt).

Step 7.3.

Case 7.3.1. att has <fixed>.

Perform convert(att, st, sv) to obtain v.

Case 7.3.2. att has <float> and <real>.

Step 7.3.2.1.

Case 7.3.2.1.1. sv is a <character-string-value> or a <bit-string-value>.

Perform convert-to-arithmeticlist(sv) to obtain a <value-and-type> whose first two components are sv and st.

Case 7.3.2.1.2. sv is a <complex-value>.

Let cv be the real part of sv; let ct be a <data-type> which has <real> but otherwise is as st.

Case 7.3.2.1.3. sv is a <real-value>.

Let cv be sv; let ct be st.

Step 7.3.2.2. Perform convert-to-float-decimal(att, st, cv) to obtain v.

Case 7.3.3. att has <float> and <complex>.

Step 7.3.3.1.

Case 7.3.3.1.1. sv is a <character-string-value> or a <bit-string-value>.

Perform convert-to-arithmeticlist(sv) to obtain a <value-and-type>: sv rdt; or a <value-and-type>: sv rdt lv ld t. If lv and ld t do not exist, let lv be a <real-values>: 0; and let ld t be rdt.

Case 7.3.3.1.2. sv is a <complex-value>.

Let rv and lv be the real and imaginary parts of sv; let rdt and ld t be <data-types> which have <real> but are otherwise as st.
Case 7.3.3.1.3. sv is a <real-value>.

Let pv be sv; let iv be a <real-value>; 0; let rdt and ldt
be st.

Step 7.3.3.2. Let rtt be a <data-type> that has <real> but is otherwise as
tt. Perform convert-to-<float--decimal>(rtt, rdt, rvl) to obtain
rvl; perform convert-to-<float--decimal>(rtt, ldt, lvl) to obtain lvl.
Let v be a <complex-value> with real part: rvl; and imaginary
part: lvl.

Step 7.4. Perform edit-numeric-picture(r, tt) to obtain a <character-string-value>.

Return cvv.

Case 8. tt has <string>, and sv is a <real-value>, <complex-value>, <bit-string-value>,
or <character-string-value>.

Step 8.1. If tt has <bit>, perform convert-to-bitlist(sv) to obtain s; otherwise, perform
convert-to-characterstring(sv) to obtain s.

Step 8.2. Let n be the (maximum-length) component of tt. If n has <asterisk>, return
s.

Step 8.3. Let n be the length of n. If n is then perform raise-condition(<stringizing-
condition>).

Step 8.4.

Case 8.4.1. n = 0.

Return a null-string.

Case 8.4.2. n > 0.

Return a string whose length is n and whose i'th <bit-value> or
<character-value> is the same as the i'th <bit-value> or <character-
value> in s, i=1,...,n.

Case 8.4.3. n < m.

If tt has <varying>, then return s; otherwise return a string whose
length is n, whose i'th <bit-value> or <character-value> is the same as
the i'th <bit-value> or <character-value> in s for i=1,..,m, and whose
remaining <bit-values> or <character-values> have, respectively,
<zero-bit>s or Rs.

Case 9. tt has <pictures--character>, and sv is a <real-value>, <complex-value>, <bit-
string-value>, or <character-string-value>.

Step 9.1. Let ct be a <data-type> that has <character> and <convert--varying> and whose
(maximum-length) contains the associated character-string length of st.
Perform convert(st, ct, svl) to obtain s.

Step 9.2. Perform validate-character-picture-valued(tt, s) to obtain a <picture-
validity>. If pv has <picture--valid>, return s.

Step 9.3. Let chfls be a <condition-bit-value-list> containing <source-value>, u;
and <char-value>, v, where u is the <integer-value> in pv. Perform
raise-condition(<conversion-condition>, chfls). Let e be the [immediate
component of the current <returned-source-value>], and go to Step 9.3.

Case 10. tt has <pointer>, st has <offset>; <variable-reference>, sv; and sv is an
<offset-value>.

Step 10.1. Perform Steps 10.1.1 and 10.1.2 in either order.

Step 10.1.1. If sv contains <null>, then return <pointer-value>: <null>.

Step 10.1.2. Perform evaluate-variable-reference(sv) to obtain a <generation>, q.
Step 10.2. Let $e$, $a$, and $i$ be, respectively, the «evaluated-data-description», «significant-allocation-list», and «storage-index-list» components of $sv$. The value of the «generation» of $sv$ has an «area-value». $sv$ must not immediately contain «empty» and there must be an immediate component, $aa$, of the «area-allocation-list» of $sv$ such that the «significant-allocation-list» of $aa$ is identical with $ai$. Let $a0$ be an «allocation-unit-designator» that designates $aa$'s «allocation-unit». Return a «pointer-values» «generation» $e0$ and $aii$.

Case II. It has «offset» «variable-reference», $x$, $st$ has «pointer» and $sv$ is a «pointer-values».

Step 11.1. Perform Steps 11.1.1 and 11.1.2 in either order.

Step 11.1.1. If $sv$ contains «null», then return «offset-values» «null».

Step 11.1.2. Perform evaluate-variable-reference($x$) to obtain a «generation» $g$.

Step 11.2. Let $e0$, $a0$, and $i0$ be, respectively, the «evaluated-data-description», «allocation-unit-designator», and «storage-index-list» components of $sv$. The value of the «generation» of $sv$ (see Section 7.1.3) has an «area-value», $sv$. $sv$ must not immediately contain «empty» and must designate an «allocation-unit» that is an immediate component of one of the elements, $aa$, of the «area-allocation-list» of $sv$. Let $ai$ be the «significant-allocation-list» of $aa$. Return as «offset-values» $e0$ $ai0$.

3.5.1.1 Informal Invocation of Convert

The operation convert may be invoked informally by text of the form "convert source-value to target-type", where "source-value" is text describing the third operand, $sv$, and "target-type" is text describing the first operand, $tt$. The result returned by convert may be referred to as "the converted value".

The second operand, $st$, may be specified explicitly in following text that describes the source-type for the conversion. Alternatively, if the source-type is not explicitly specified, and $sv$ was obtained by performing evaluate-expression), the source-type is by implication the scalar «data-type» of $x$ that corresponds to the «basic-values», $sv$.

The description of the operand $tt$ need not be complete. The following conventions are applied to complete the description of the operand $tt$:

1) When $tt$ has «string»), its «maximum-length» has «asterisk» unless the contrary is explicitly specified by a phrase such as "of specified length n", and it has «varying» unless «converting» is explicitly specified.

2) When «base», «scale», and «mode» components are specified, then $tt$ has «arithmetic» and not «pictureable» unless «pictureable» is explicitly specified.

3.5.1.2 Convert-to-fixed

Operation: convert-to-fixed($tt$, $st$, $sv$)

where $tt$ is a «data-type» that has «real» and «fixed», $st$ is in a «data-type», $sv$ is a «real-value».

result: a «real-value».

Step 1. Let $b=2$ or $b=10$, according as $tt$ has «binary» or «decimal». Let $p$ and $q$ be the «number-of-digits» and «scale-factor» of $tt$.

Step 2. Let $cv = (b-1) * \text{sign}(sv) * \text{floor}((b+1) * \text{abs}(sv))$. 366
Step 5.

Case 5.1. at has <fixed> and <scale-factor> 0, or at has <fixed> and the same <name> as tt.

Let v = cv.

Case 5.2. (Otherwise).

Let v be an implementation-dependent approximation to cv such that v = w * p - q
for some integer w.

Step 6.

Case 6.1. abs(v) ≥ b^p-q.

Perform raise-condition(<raise-condition>).

Case 6.2. (Otherwise).

Return a <real-value>; v.

9.5.1.1 Convert-to-float

Operation: convert-to-float(tt, st, sv)

where tt is a <data-type> that has <real> and <float>,

st is a <data-type>,

sv is a <real-value>.

result: a <real-value>.

Step 1. Let p = 2 or p = 16, according as tt has <binary> or <decimal>. Let p be the <number-of-digits> of tt.

Step 2.

Case 2.1. sv is an integer such that abs(sv) < b^p.

Return sv.

Case 2.2. (Otherwise).

Perform Step 2.2.1 or Step 2.2.2 or Step 2.2.3.

Step 2.2.1. Let e be the unique integer such that b^e < |abs(sv)| < b^(e+1). Let v be a value such that abs(v) = |sv| * b^(-e). Return a <real-value> containing an implementation-dependent approximation to v.

Step 2.2.2. Perform raise-condition(<underflow-condition>). Return a <real-value>; 0.

Step 2.2.3. Perform raise-condition(<overflow-condition>).
9.5.1.4 Convert-to-bit

Operation:  convert-to-bit(st, sv)

where st is a <data-type>,
   sv is a <real-value>, <complex-value>, <bit-string-value>, or
   <character-string-value>.

result: a <bit-string-value>.

Case 1.  st has <arithmetic> (including <arithmetic> in <picture-numeric>).

Step 1.1.

Case 1.1.1.  sv is a <real-value>.

Let v = abs(sv).  Let ct be st.

Case 1.1.2.  sv is a <complex-value>.

Let w = abs(sv), where x is the real part of sv.  Let ct be a <data-type>
   with <code>: <real>; but otherwise as st.

Case 1.1.3.  st has <picture-numeric>, and sv is a <character-string-value>.

Perform validate-numeric-picture-value(st, sv) to obtain <picture-
   validity>, w.  w must have <picture-valid>.  The <picture-valid>
   component of w then immediately contains a <real-value>-x, or a
   <complex-value> whose real part is x.  Let v = abs(|x|).  Let ct be the
   associated arithmetic data-type of st.

Step 1.2.  Let r be the <number-of-digits> of ct, and let s be the 
   <scale-factor> of st.  According to the <base> and <scale> of ct,
   determine p as follows:

   <binary> <fixed> : p = min(S, max(r-s,0))
   <decimal> <fixed> : p = min(S, max(ceil(3.32*|r-s|),0))
   <binary> <float>   : p = min(S, r)
   <decimal> <float> : p = min(S, ceil(3.32*r)).

S is the maximum <precision> for <binary> and <fixed>.

Step 1.3.  If p=0, return a <bit-string-value>: <null-bit-string>.  Otherwise let tt be
   a <data-type> that has <real>, <fixed>, and <binary> with <number-of-
   digits>: p; and <scale-factor>: s.  Perform convert(tt,ct,v) to obtain a
   non-negative integer, n.

Step 1.4.  If n>2^p, then perform raise-condition(<raise-condition>); otherwise n can be
   exactly represented as the sum of ℓ12+...+ℓi12 for i=1 to p where each ℓi12
   is 0 or 1.

   Return a <bit-string-value> of length p, whose i'th <bit-value> has a <zero-
   bit> or <one-bit> according as ℓi12 is 0 or 1.

Case 2.  st has <bit>, and sv is a <bit-string-value>.

Return sv.

Case 3.  st has <character> or <picture-character>, and sv is a <character-string-
   value>.

Step 3.1.  If sv contains a <character-value> that has neither $\text{12}$ nor $\text{13}$, go to Step
   3.3.

Step 3.2.  If sv has a <null-character-string>, return the <bit-string-value>:
   <null-bit-string>.  Otherwise return a <bit-string-value>, whose length
   equals the length of sv, and whose i'th <bit-value> has a <zero-bit> or
   <one-bit> according as the i'th <character-value> in sv has $\text{12}$ or $\text{13}$. 

368
Step 3.3. Let \( c \) be a \(<\text{condition-expression-value}>>\) containing \(<\text{resource-value}>>\) \( n \); and \(<\text{char-value}>>\) \( s \); where \( n \) is the position of the first \(<\text{character-value}>>\) in \( s \) that has neither \( \{\} \) nor \( \{\} \). Perform \( \text{value-to-condition-condition}(c, s) \). Let \( s' \) be the immediate component of the current \(<\text{return-value}>>\), and go to Step 3.1.

### 9.5.1.5 convert-to-character

In this operation the notation \( \{n} \text{picture-element} \{n} \) is used to indicate \( n \) (\( n \)) consecutive occurrences of that \( \{ \text{picture-element} \}. \)

**Operation:**  \( \text{convert-to-character}(st, sv) \)

where \( st \) is a \(<\text{data-type}>>\), \( sv \) is a \(<\text{real-value}>>\), \(<\text{complex-value}>>\), \(<\text{bit-string-value}>>\), or \(<\text{character-string-value}>>\).

**result:** a \(<\text{character-string-value}>>\).

**Case 1.** \( st \) has \(<\text{real}>>\) and \(<\text{fixed}>>\) but not \(<\text{picted}>>\).

**Step 1.1.** If \( st \) has \(<\text{binary}>>\), let \( \text{d} \) be a \(<\text{data-type}>>\) which has \(<\text{real}>>\) \(<\text{fixed}>>\) \(<\text{decimal}>>\) with the converted \(<\text{precision}>>\) of \( st \) for that target; otherwise, let \( \text{d} \) be \( st \). Perform \( \text{convert-to-value}(\text{d}, sv) \) to obtain \( cv \).

Let \( p \) and \( q \) be the \(<\text{number-of-digits}>>\) and \(<\text{scale-factor}>>\) of \( \text{d} \).

**Step 1.2.**

**Case 1.2.1.** \( p < q \) or \( q \geq 0 \).

Let \( \text{pic} \) be the \(<\text{data-type}>>\) corresponding to the \( \{ \text{picture} \} \) with concrete-representation as follows:

- if \( q > 5 \) \( \{12\}8\{p\}-q \)
- if \( p > 0 \) \( \{-9.4\}q \)
- if \( p = 0 \) \( \{0m\}-9.4\{q\}8 \) where \( m = q \).

Perform \( \text{edit-numeric-picture}(cv, \text{pic}) \) to obtain a \(<\text{character-string-value}>>\).

**Return \( s' \).**

**Case 1.2.2.** \( p \geq q \) or \( q < 0 \).

Let \( \text{pic}_1 \) be the \(<\text{data-type}>>\) corresponding to the \( \{ \text{picture} \} \) with concrete-representation \( \{1\}8\).

Perform \( \text{edit-numeric-picture}(\text{d}, \text{pic}_1) \) to obtain a \(<\text{character-string-value}>>\), \( v_1 \), where \( z \) is a \(<\text{real-value}>>\) \( cv\times10^q \).

Let \( k \) be the unique integer such that \( 10^k \in \{1\}8 \cdot \text{sign}\{1\}8 \cdot 10^k \). Let \( \text{pic}_2 \) be the \(<\text{data-type}>>\) corresponding to the \( \{ \text{picture} \} \) with concrete-representation \( \{1\}k \). Perform \( \text{edit-numeric-picture}(q, \text{pic}_2) \) to obtain a \(<\text{character-string-value}>>\), \( v_2 \).

Let \( f \) be a \(<\text{character-string-value}>>\) containing the single \( \{ \text{symbol} \} \) \( \{\} \). Perform \( \text{concatenate}(v_1, f) \) to obtain \( s_1 \); perform \( \text{concatenate}(s_1, v_2) \) to obtain \( s_2 \).

**Return \( s_2 \).**

**Case 2.** \( st \) has \(<\text{real}>>\) and \(<\text{floating}>>\) but not \(<\text{picted}>>\).

**Step 2.1.** If \( st \) has \(<\text{binary}>>\), let \( \text{d} \) be a \(<\text{data-type}>>\) which has \(<\text{real}>>\) \(<\text{fixed}>>\) \(<\text{decimal}>>\) with the converted \(<\text{precision}>>\) of \( st \) for that target; otherwise, let \( \text{d} \) be \( st \). Perform \( \text{convert-to-float-decimal}(\text{d}, sv) \) to obtain \( cv \).

Let \( p \) be the \(<\text{number-of-digits}>>\) of \( \text{d} \).
Step 2.3: Let k be the implementation-defined size of the exponent field for
floating-point-format. Let pic be the <data-type> corresponding to the
<picture> with concrete-representation "~SV.ni925399", where n=1.
Perform edit-numeric-picture(cv,pic) to obtain a <character-string-value>,
v.
Return v.

Case 3. st has <complex> but not <parsed-numeric>, and sv is a <complex-value>.

Step 3.1. Let cv be a <data-type> that has <real> but is otherwise the same as st.
Let x and y be the real and imaginary parts of sv. In either order, perform
convert-to-character[int,k] to obtain x' and convert-to-character[int,y] to
obtain y'.

Step 3.2. Let n be the length of the <character-string-value>,x' (and necessarily,
also of y'). Let k be the number of leading <character-values> in y' which
have Bs. Let y'[j], j=1,...,k, be the <character-values> in y'.

Step 3.3.

Case 3.3.1. y'[i+1] has $\#$.
Let y'i be a <character-string-values> of length n+1 containing, in order,
y'[i+1] through y'[i], \{symbol\}: \{t\}; and y'[i+1] through y'[i].

Case 3.3.2. (Otherwise).

Let y'i be a <character-string-values> of length n+1 containing, in order,
\{symbol\}: \{t\}; y'[i+1] through y'[i], \{symbol\}: \{i\}; and y'[i+1] through
y'[i].

Step 3.4. Perform concatenate(x',y'i) to obtain a <character-string-value>,v, of length
t^2+n+1.
Return v.

Case 4. st has <bit> and sv is a <bit-string-value>.

If sv has the <null-bit-string>, return <character-string-value>:
<null-character-string>. Otherwise return a <character-string-value> whose
length equals the length of sv, and whose i'th <character-value> has \{0\} or \{1\}
according to whether the i'th <bit-value> of sv has \{zero-bit\} or \{one-bit\}.

Case 5. st has <character>, <parsed-character>, or <parsed-numeric>, and sv is a
<character-string-value>.

Return sv.

2.5.2.6 Conversion to Float Decimal

This operation returns a <real-value> which is exactly representable in floating notation
in a given number of significant digits. It is used in the conversion of numeric values to character representations.

Operation: convert-to-float-decimal(st,sv)

where st is a <data-type> which has <real>, <float>, and <decimal>,
sv is a <real-value>.

result: a <real-value>.

Step 1. If sv is 0, return a <real-value>: 0.

Step 2. Let p be the number-of-digits in st. Let q be the unique integer such that
10^{p-1} <abs(sv) < 10^p.

Let cv = 10^q+p*abs(cv)*floor(10^p-q*abs(cv)+0.5).

170
Step 3.

Case 3.1. st has <fixed> and <decimal>.

Return a <real-value>; cv.

Case 3.2. (Otherwise).

Return a <real-value> containing an implementation-dependent approximation to cv, having the form \( m \times d \), where \( m \) and \( d \) are integers and abs(d) <= 1.

3.5.1.7 Conversion from String or Picture to Arithmetic

Operation: convert-to-arithmetic(st, cv)

where st is a <data-type>,

sv is a <character-string-value> or a <bit-string-value>.

result: a <value-and-type> (see Section 9.5.1.8).

Case 1. st has <picture-numeric>.

Step 1.1. Perform validate-numeric-picture-st, cv to obtain a <picture-validity>, pv, which must have <picture-value>.

Step 1.2.

Case 1.2.1. st has <real>.

Let rv be the <real-value> in pv. Let stt be the associated arithmetic data-type of st. Return a <value-and-type>; rv stt.

Case 1.2.2. st has <complex>.

Let rv and iv be the real and imaginary parts of the <complex-value> in pv. Let stt be a <data-type> which has <real> but is otherwise as the associated arithmetic data-type of st. Return a <value-and-type>; rv stt iv.

Case 2. st has <bit>.

Step 2.1.

Case 2.1.1. sv has <all-bit-string>.

Let cv=0 and n=1.

Case 2.1.2. (Otherwise).

Let n be the length of sv. Let xill be 0 or 1 according as the i'th <bit-value> of sv is <zero-bit> or <one-bit>; let cv be the sum of xill*2^(n-i), for i=1.

Step 2.2. Let ct be a <data-type> containing <real>, <fixed>, <binary>, <number-of-digits>: n; and <scale-factor>: 0.

Return a <value-and-type>; <real-value>; cv; ct.

Case 3. st has <character> or <picture-character>.

Step 3.1.

Case 3.1.1. sv conforms to the syntax for <numeric-string> (see Section 9.5.1.8) but does not contain \( \$ \).

Perform basic-numeric-value(sv) to obtain a <value-and-type>; vt. Return vt.
Case 3.1.2. (Otherwise).

Let \( \text{cblks} \) be a <condition-hift-value-list> containing <cons-constant>; \( sv \); and <char-constant>; \( n \), where \( n \) is the smallest integer such that the <character-string-value> of length \( n \) of \( sv \) does not have a continuation conforming to the syntax of <numeric-string> without \( \text{cblks} \). (If the whole of \( sv \) has such a continuation, let \( n \) be the length of \( sv \).)

Perform <raise-condition><conversion-condition>, cblks. Let \( sv \) be the immediate component of the current <returned-numeric-value>, and go to Step 3.1.

### 9.5.1.8 Basic Numeric Value of a String

\[
\text{<value-and-type> ::= <real-value> <data-type> | <real-value> <data-type>}
\]

The optional components occur when a <value-and-type> represents the two parts of a <complex-value>.

\[
\text{<numeric-string> ::= <blanks> | <blanks> [+] - <arithmetic-constant> | <complex-expression>} <blanks>}
\]

<blanks>::= {[E-list]}

<complex-expression>::= <sign> <real-constant>

<sign>::= [+] -

<sign>-::= +-

Operation: \( \text{basic-numeric-value}(\text{str}) \)

where \( \text{str} \) is a <character-string-value> whose terminal components are permitted terminal components of <numeric-string>.

result: a <value-and-type>.

Step 1. Let \( \text{str} \) be the <numeric-string> whose components are the elements of \( \text{str} \), taken in order.

Step 2.

Case 2.1. \( \text{str} \) contains only <blanks>.

Returns a <value-and-type> with components <real-value>: 0; and <data-type> containing <arithmetic> with <real>, <fixed>, <decimal>, <number-of-digits>: 1; and <scale-factor>: 0.

Case 2.2. \( \text{str} \) immediately contains <arithmetic-constant>: <real-constant>,<.

Perform evaluate-real-constant, to obtain a <value-and-type>, \( \text{vt} \). If \( \text{str} \) immediately contains \( \text{,} \), return a <value-and-type> equal to \( \text{vt} \) except that the sign of the <real-value> is negative; otherwise, return \( \text{vt} \).

Case 2.3. \( \text{str} \) immediately contains <arithmetic-constant>: <real-constant>,<.

Perform evaluate-real-constant, to obtain a <value-and-type>, \( \text{vt} \). If \( \text{str} \) immediately contains \( \text{,} \), let \( \text{vt} \) = \( \text{yr} \); otherwise, let \( \text{vt} \) = \( \text{yl} \). Return a <value-and-type>: <real-value>: 0; \( \text{yr} \) <real-value>: \( \text{yr} \) <real-value>: \( \text{yr} \).

Case 2.4. \( \text{str} \) contains <complex-expression>,<.

Step 2.4.1. Perform evaluate-real-constant, where \( \text{yr} \) is the <real-constant> immediately contained in \( \text{yr} \), to obtain a <value-and-type>: <real-value>,<. Perform evaluate-real-constant, where \( \text{yr} \) is the <real-constant> in the <complex-constant> in \( \text{yr} \), to obtain a <value-and-type>: <im-constant>,<.
Step 2.4.2. If `<sign-r>` has `$-$`, let `rv1=rv`; otherwise, let `rv1=rv`. If `<sign-l>` has `$+$`, let `lv1=lv`; otherwise let `lv1=lv`.

Step 2.4.3. Return a `<value-and-type>`: `rv1 rv2 lv1 lv2`.

9.5.1.2 Evaluate-real-constant

Operation: `evaluate-real-constant(rc)`

where `rc` is a `<real-constant>`.

result: a `<value-and-type>` (see Section 9.5.1.3).

Step 1.

Case 1.1. `rc` contains a `<decimal-constant>`.

Let `v` be the `<real-value>` obtained by interpreting the `<decimal-number>` in `rc` as a decimal constant.

Let `b = 10`.

Case 1.2. `rc` contains a `<binary-constant>`.

Let `v` be the `<real-value>` obtained by interpreting the `<binary-number>` in `rc` as a binary constant.

Let `b = 2`.

Step 2. If `<exponent>` exists in `rc`, let `x` be the `<integer-value>` obtained by interpreting the component of `<exponent>` as a decimal constant; otherwise, let `x=0`.

Step 3. Let `ar` be a `<float-type>` containing `<arithmetic>` containing `<mode>`: `<real`; are `<scale>`, `<base>`, and `<precision>` as defined below:

Step 3.1. `<base>` has `<decimal>` or `<binary>` as `$10$` or `$2$`.

Step 3.2. `<number-of-digits>` is the total number of `<digits>` or `<binary-digits>` in `<decimal-number>` or `<binary-number>`.

Step 3.3.

Case 3.3.1. `rc` contains `<scale-type>`: `<\textbf{3}>`.

- `<scale>` has `<fixed>`.

Case 3.3.2. (Otherwise):

- `<scale>` has `<fixed>`. `<scale-factor>` is `$q\cdot\text{ sign }\cdot \langle\text{digits}\rangle$ or `$\langle\text{binary-digits}\rangle$ following `$+$` (if any) in `$\langle\text{decimal-number}\rangle$ or `$\langle\text{binary-number}\rangle$`.

Step 4. The `<number-of-digits>` in `ar` must not be greater than the maximum `<number-of-digits>` allowed for the `<base>` and `<scale>` of `ar`. Return a `<value-and-type>` `<real-value>` `<ar>`.
### 9.5.2 Numeric Pictures

Informally, `<pictorial-numeric>` is a way of holding a numeric value in a `<character-string-value>`. For a `<pictorial-numeric>` in a `<data-type>`, the `<character-string-value>` will be a value in the machine state and represents a real or complex numeric value; for a `<pictorial-numeric>` in a `<format-item>`, the value will be a string of characters transmitted to or from a `<stream-dataset>`, an `<expression>` in a `<got-string>`, or a `<target-reference>` in a `<got-string>`, and represents a real numeric value.

The `<numeric-picture-specification>` in `<pictorial-numeric>` specifies:

1. In conjunction with the optional `<picture-scale-factor>`, the `<precision>` and `<scale>` of the decimal numeric values which may be held. Together with `<base>`, `<decimal>`, and a `<code>`, which is declared or defined for always `<real>` for a `<format-item>`, these make up the `<arithmetic>` subnode of `<pictorial-numeric>` established in Section 9.4.4.1. The `<data-type>` which contains `<arithmetic>` with the same subnodes is known as the associated arithmetic `<data-type>` of the `<pictorial-numeric>` `<data-type>` or `<format-item>`.

2. The constant length of the `<character-string-value>` representations of numeric values.

The associated `<character-string-length>` of a `<numeric-picture-specification>` or `<numeric-picture-element-list>` is equal to the number of its terminal nodes, with the exceptions of `[@]` and `[@]`.

The associated `<character-string-length>` of a `<data-type>` or `<format-item>` containing `<pictorial-numeric>` and `<real>` is equal to that of its `<numeric-picture-specification>`.

The associated `<character-string-length>` of a `<data-type>` containing `<pictorial-numeric>` and `<complex>` is twice that of its `<numeric-picture-specification>`.

3. Exactly how the `<real-value>` or `<complex-value>` is to be edited into or extracted from the `<character-string-value>`.

Section 9.5.2.1 defines the editing of the `<real-value>` or `<complex-value>` into the `<character-string-value>` under the control of the `<pictorial-numeric>` specification; Section 9.5.2.2 is a sub-operation of Section 9.5.2.1 which edits a single field (see below) of a picture.

Section 9.5.2.3 defines the reverse process: checking the validity of a `<character-string-value>` against a `<pictorial-numeric>` specification, and extracting the associated `<real-value>` or `<complex-value>` from a valid `<character-string-value>`; Section 9.5.2.4 is a sub-operation of Section 9.5.2.3 which checks the validity of (and extracts the value from) a single field (see below) of a picture.

A field of a `<numeric-picture-specification>` (or of a `<character-string-value>`) is the subtree of `<numeric-picture-specification>` (or the substring of the value) which corresponds to one of the following:

- `<fixed-point-picture>`
- `<picture-scientific>`
- `<picture-exponent>`.
9.5.2.1 Editing Numeric Pictures

Operation: edit-numeric-picture(vc,pic)

where vc is a <real-value> or a <complex-value>,
      pic is a <data-type> containing <picture-numeric> with associated
      arithmetic data-type, adt.

results: a <character-string-value>.

Case 1. adt has <scale>: <fixed>; and <mode>: <real>.

Step 1.1. Let p and q be the <number-of-digits> and <scale-factor> of adt. There is a
unique representation of vc in terms of integers dij:

\[ \text{sgn} \times \sum_{j=1}^{p} \left( d_{ij} \times 10^{(p-q-j)} \right) \]

where \( 0 \leq d_{ij} \leq 9 \), \( j=1 \) to \( p \)

\[ \text{sgn} = +1 \quad \text{if vc} \geq 0 \]
\[ \text{sgn} = -1 \quad \text{if vc} < 0. \]

Perform edit-numeric-picture-field(spec,d,sgn) to obtain a <character-string-value>, cm, where:

spec is the <numeric-picture-element-list> of pic,

\( d \) is the <character-string-value> of length \( p \) whose \( j \)'th <character-value>

has the \{symbol\} which is the \{digit\} that represents \( d_{ij} \).

Step 1.2. Return cm.

Case 2. adt has <scale>: <float>; and <mode>: <real>.

Step 2.1. Let \( p \) be the <number-of-digits> in adt; let \( q \) be the number of digit-
positions (as in Section 9.4.6.1) following the \{FP\} <numeric-picture-
element> in <picture-mantissa> of pic \( q = 0 \) if there is no \{FP\}; let \( px \) be the
number of digit-positions in <picture-exponent> of pic.

Then there is a unique representation of vc as \( \text{ve} + 10^{px} \), where \( \text{ve} \) is a
signed integer such that:

\[ 10^{-(p-q-1)} \leq \text{abs}(\text{ve}) < 10^{(p-q)} \quad \text{if} \ vc \neq 0 \]
\[ \text{ve} = 0 \quad \text{or} \quad \text{ve} = \text{ve} = 0 \quad \text{if} \ vc = 0. \]

Step 2.2. There is a unique representation of vc in terms of integers dij:

\[ \text{sgn} \times \sum_{j=1}^{p} \left( d_{ij} \times 10^{(p-q-j)} \right) \]

where \( 0 \leq d_{ij} \leq 9 \), \( j=1 \) to \( p \)

\[ \text{sgn} = +1 \quad \text{if vc} \geq 0 \]
\[ \text{sgn} = -1 \quad \text{if vc} < 0. \]

Perform edit-numeric-picture-field(spec,d,sgn) to obtain a <character-string-value>, cm, where:

spec is the <numeric-picture-element-list> of <picture-mantissa> of pic,

\( d \) is the <character-string-value> of length \( p \) whose \( j \)'th <character-value>

has the \{symbol\} which is the \{digit\} that represents \( d_{ij} \).
Step 2.3. If abs(vn) ≥ 10 * px, perform raise-condition(sizc-condition); otherwise, there is a unique representation of vn in terms of integers diji:

\[ \text{sgn} \sum_{j=1}^{px} (\text{dij}_j + 10 \times (\text{px} - j)) \]

where \( 0 \leq \text{dij}_j \leq 9 \), \( j \) from 1 to px.
\( \text{sgn} = +1 \) if \( \text{vn} > 0 \)
\( \text{sgn} = -1 \) if \( \text{vn} < 0 \).

Perform edit-numeric-picture-field(spec,d,sgn) to obtain a character-string-value, cr, where:

spec is the numeric-picture-element-list of the picture-exponent of pic,
\( d \) is the character-string-value of length px whose \( j \)th character-value has the symbol which is the digit that represents diji.

Step 2.4. Perform concatenate(sc,cr) to obtain c. Return c.

Case 3. \( \text{ad} \) has <mode> <complex>.

Step 3.1. Let vcr, wci be the real and imaginary parts of \( \text{vn} \); let picr be the <data-type> which is the same as pic except that it has <mode> <real>.

Perform edit-numeric-picture-field(vcr,picr) to obtain cr; perform edit-numeric-picture-field(wci,picr) to obtain c1.

Step 3.2. Perform concatenate(cr,c1) to obtain c. Return c.

9.5.2.2 Editing a Numeric Picture Field

<pic-status>=<supression> {<supression-type>}

<supression>:=<on> | <off>

<supression-type>:= {# | # | # | # | # | # | # | #}

Operation: edit-numeric-picture-field(pic,d,sgn)

where pic is a numeric-picture-element-list containing an numeric-picture-element,
\( d \) is a character-string-value containing p <symbols>, where p is the number of digit-positions in pic and each <symbol> corresponds to a digit,
\( \text{sgn} \) is the value 1 or -1.

result: a character-string-value of length nc, where nc is the associated character-string length of pic.

Step 1. If \( \text{sgn} = -1 \), perform raise-condition(sizc-condition) unless pic contains at least one numeric-picture-element which immediately contains #52, #42, #32, #22, #12, #02, <credit>, or <debit>.

Step 2. If pic contains no numeric-picture-element which immediately contains #52, #42, #32, #22, or #12, and if all <symbols> in \( d \) have #0, then return a character-string-value of length nc all of whose character-values contain #* or & according to whether pic contains an #* or not.
Step 3. Let still, i=1, ..., n, be the element in pic.
Let dij, j=1, ..., p, be the character-value in d.
Let cikl, k=1, ..., n, be the character-value; the remainder of this operation completes the trees cikl as a function of the still and dij.
Let patat be a <pic-status>:
  <suppression>, sup:
  <off>.
Let each of i, j, k be initially 1.

Step 4. Select the appropriate case depending on still.

Case 4.1. still immediately contains $\text{&&}$, $\text{!!}$, $\text{??}$, $\text{!!}$, or $\text{!!}$.

Step 4.1.1.

Case 4.1.1.1. sup is <suppression>; $	ext{&&}$.
Replace sup by <suppression>; <off>. If <suppression-type> has $\text{&&}$, $\text{!!}$, $\text{??}$, or $\text{!!}$, replace the $\text{symbol}$ contained in cikl by the $\text{symbol}$ obtained from Table 9.3 as a function of <suppression-type> and (possibly) sup.

Case 4.1.1.2. sup is <suppression>; <off>.
No action.

Step 4.1.2. Attach to cikl the $\text{symbol}$ obtained from Table 9.3 as a function of still, dij, and (possibly) sup.

Step 4.1.3. i=i+1; j=j+1; k=k+1.

Case 4.2. still has $\text{??}$ or $\text{!!}$.

Step 4.2.1. If patat does not have <suppression-types>, replace sup by <suppression>; $\text{&&}$; and append <suppression-type>; pc; to patat, where pc is the component of still.

Step 4.2.2.

Case 4.2.2.1. sup is <suppression>; $\text{&&}$; and dij has $\text{??}$.
If still has $\text{&&}$, attach # to cikl; if still has $\text{??}$, attach $\text{&&}$ to cikl.

Case 4.2.2.2. sup is <suppression>; <off>; or dij does not have $\text{??}$.
Attach the immediate component of dij to cikl. Replace sup by <suppression>; <off>.

Step 4.2.3. i=i+1; j=j+1; k=k+1.

Case 4.3. still has $\text{??}$, $\text{!!}$, $\text{??}$, or $\text{!!}$, and there is no other n such that still = still.

Step 4.3.1. Attach to cikl the $\text{symbol}$ obtained from Table 9.3 as a function of still and (possibly) sup.

Step 4.3.2. i=i+1; k=k+1.

Case 4.4. still has $\text{!!}$, $\text{!!}$, $\text{!!}$, or $\text{!!}$, and still = still for some n greater than i, but for no n less than i.

Step 4.4.1. Replace sup by <suppression>; $\text{??}$; append <suppression-type>; pc; to patat, where pc is the component of still.

Step 4.4.2. Attach # to cikl.
Step 4.4.3. i=i+1; k=k+1.

Chapter 9: Expressions and Conversion 377
Case 4.5. i still has $\{T\}, \{P\}, \{+\}, \{\_\}$, or $\{\_\}$, and $\text{nil} = \text{nil}$ for some $n$ less than $i$.

Step 4.5.1.

Case 4.5.1.1. sep is $\langle$suppression$\rangle$: $\{og\}$ and $dijl$ has $\{og\}$.
Attach $\text{nil}$ to $cikl$.

Case 4.5.1.2. sep is $\langle$suppression$\rangle$: $\{og\}$ and $dijl$ does not have $\{og\}$.
Replace the immediate component of $cikl$ by the $\{og\}$ symbol obtained from Table 9.1, as a function of $dijl$ and (possibly) $\text{syn}$. Attach the immediate component of $dijl$ to $cikl$. Replace sep by $\langle$suppression$\rangle$: $\{og\}$.

Case 4.5.1.3. sep is $\langle$suppression$\rangle$: $\{off\}$.
Attach the immediate component of $dijl$ to $cikl$.

Step 4.5.2. $i=i+1$; $j=j+1$; $k=k+1$.

Case 4.6. still has $\{V\}$.


Step 4.6.2. If $\text{patat}$ does not have $\langle$suppression-type$\rangle$, append $\langle$suppression-type$\rangle$ with no submodel to $\text{patat}$.

Note: This handles the case of pictures with all digit-positions suppressible and after $\{V\}$, for non-zero values.

Step 4.6.3. $i=i+1$.

Case 4.7. still has $\langle$insertion-character$\rangle$.

Step 4.7.1.

Case 4.7.1.1. sep is $\langle$suppression$\rangle$: $\{off\}$.
If still has $\{w\}$, attach $\text{nil}$ to $cikl$; otherwise attach to cikl the terminal component of still.

Case 4.7.1.2. sep is $\langle$suppression$\rangle$: $\{og\}$.
If $\langle$suppression-type$\rangle$ has $\{+\}$, attach $\{+\}$ to $cikl$, otherwise, attach $\text{nil}$ to $cikl$.

Step 4.7.2. $i=i+1$; $k=k+1$.

Case 4.8. still has $\langle$credit$\rangle$ or $\langle$debit$\rangle$.

Step 4.8.1.

Case 4.8.1.1. $\text{syn}=1$.
Attach $\text{nil}$ to each of $cikl$ and $cikl+1$.

Case 4.8.1.2. $\text{syn}=1$.
Attach to $cikl$ and $cikl+1$ $\{C\}$ and $\{P\}$ (respectively) if still has $\langle$credit$\rangle$, or $\{O\}$ and $\{D\}$ (respectively) if still has $\langle$debit$\rangle$.

Step 4.8.2. $i=i+1$; $k=k+2$.

Case 4.9. still has $\{E\}$.
Attach $\{E\}$ to $cikl$.

$i=i+1$; $k=k+1$. 378
Case 4.10 still has \( \mathcal{E} \).

\[ i = i + 1. \]

Step 5. If \( i \) is odd then go to Step 4.

Step 4. Return a character-string-value containing \( c(i) \), \( k = 1, \ldots, m \), in order.

### Table 4.3. Table of symbols as a function of suppression-type for Edit-numeric-picture-field.

<table>
<thead>
<tr>
<th>sign</th>
<th>sill or suppression-type</th>
</tr>
</thead>
<tbody>
<tr>
<td>( +1 )</td>
<td>( # )</td>
</tr>
<tr>
<td>( -1 )</td>
<td>( # )</td>
</tr>
</tbody>
</table>

The positions indicated by \( \# \) represent a symbol which is implementation-defined. This symbol represents a currency symbol.

### Table 4.4. Table of symbols as a function of numeric-picture-element\\-s and character-values for Edit-numeric-picture-field.

<table>
<thead>
<tr>
<th>still</th>
<th>sign</th>
<th>( d(i) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( {2} )</td>
<td>( # )</td>
<td>( {2} )</td>
</tr>
<tr>
<td>( {2} )</td>
<td>( # )</td>
<td>( {2} )</td>
</tr>
<tr>
<td>( {2} )</td>
<td>( # )</td>
<td>( {2} )</td>
</tr>
<tr>
<td>( {2} )</td>
<td>( # )</td>
<td>( {2} )</td>
</tr>
<tr>
<td>( {2} )</td>
<td>( # )</td>
<td>( {2} )</td>
</tr>
<tr>
<td>( {2} )</td>
<td>( # )</td>
<td>( {2} )</td>
</tr>
</tbody>
</table>

The 20 positions indicated by \( * \) represent 20 symbols which are implementation-defined. These symbols represent a digit and a sign in one symbol.
9.5.2.1 Validity of a Numeric Pictured Value

\[
\text{\texttt{picture-validity}} = \text{\texttt{picture-valid}} \mathbin{\text{\lor}} \text{\texttt{picture-invalid}}
\]

\[
\text{\texttt{picture-valid}} = (\text{\texttt{real-value}} \mathbin{\text{\lor}} \text{\texttt{complex-value}})
\]

\[
\text{\texttt{picture-invalid}} = \text{\texttt{integer-value}}
\]

The subcode of \text{\texttt{picture-valid}} is the associated numeric value of \texttt{v} with respect to \texttt{pic}. The subcode of \text{\texttt{picture-invalid}} is the ordinal of the first \texttt{character-value} in \texttt{v} which is invalid with respect to \texttt{pic}.

This operation is invoked by the operation \text{\texttt{valid-date-pict-val}}(pic,\text{\texttt{pv}}), by the operation \text{\texttt{convert}} for \text{\texttt{pict-format-source-type}}, by the operation \text{\texttt{valid-date-input-format}} for a \text{\texttt{picture-format}}.

Operation: \text{\texttt{validate-pict-val}}(pic,\text{\texttt{pv}})

where \texttt{pic} is a \text{\texttt{data-type}} or \text{\texttt{picture-format}} containing \text{\texttt{pict-numeric}}.
\texttt{v} is a \text{\texttt{character-string-value}} of length equal to the associated \text{\texttt{character-string length}} of \texttt{pic}.

result: a \text{\texttt{picture-validity}}.

Case 1. \texttt{pic} has associated arithmetic data-type with \text{\texttt{mode}}: \text{\texttt{real}}; and \text{\texttt{scale}}: \text{\texttt{fixed}}.

Step 1.1. Perform \text{\texttt{validate-field-of-pict-val}}(pic,\text{\texttt{pv}}) to obtain a \text{\texttt{picture-validity}}.\text{\texttt{pv}}, where \texttt{plot} is the \text{\texttt{numeric-picture-element-list}} in \texttt{pic}.

Step 1.2.

Case 1.2.1. \texttt{pv} has \text{\texttt{picture-valid}}.

Let \texttt{rv} be the \text{\texttt{real-value}} in \texttt{pv}. Return a \text{\texttt{picture-validity}}: \text{\texttt{picture-valid}}: \text{\texttt{real-value}}: \text{\texttt{rv} \cdot 10^{\text{\texttt{scale-factor}}}}\ldots, where \texttt{q} is the \texttt{scale-factor} in the associated arithmetic data-type of \texttt{pic}.

Case 1.2.2. \texttt{pv} has \text{\texttt{picture-invalid}}.

Return \texttt{pv}.

Case 2. \texttt{pic} has associated arithmetic data-type with \text{\texttt{mode}}: \text{\texttt{real}}; and \text{\texttt{scale}}: \text{\texttt{fixed}}.

Step 2.1. Let \texttt{ps}, \texttt{px} be the \texttt{numeric-picture-element-lists} of \text{\texttt{picture-significance}} and \text{\texttt{picture-exponent}} in \texttt{pic}. Let \texttt{i} be the associated \texttt{character-string length} of \texttt{ps} and \texttt{px}. Let \texttt{vs}, \texttt{vx} be \texttt{character-string-values} containing the first \texttt{i} and last \texttt{i} \texttt{character-values} of \texttt{v}, respectively.

Step 2.2. Perform \text{\texttt{validate-field-of-pict-val}}(\texttt{ps},\text{\texttt{vs}}) to obtain \texttt{psv}; perform \text{\texttt{validate-field-of-pict-val}}(\texttt{px},\text{\texttt{vx}}) to obtain \texttt{pxv}.

Step 2.3.

Case 2.3.1. \texttt{psv} has \text{\texttt{picture-invalid}}.

Return \texttt{psv}.

Case 2.3.2. \texttt{psv} has \text{\texttt{picture-valid}}, and \texttt{pxv} has \text{\texttt{picture-invalid}}.

Return a \text{\texttt{picture-validity}}: \text{\texttt{picture-invalid}}: \text{\texttt{integer-value}}: \text{\texttt{psv}}, where \texttt{n} is the value in \texttt{psv}.

Case 2.3.3. (Otherwise).

Return a \text{\texttt{picture-validity}}: \text{\texttt{picture-valid}}: \text{\texttt{real-value}}: \text{\texttt{psv} \cdot 10^{\text{\texttt{psv-scale}}}}\ldots, where \texttt{n} is the value in \texttt{psv}, \texttt{x} is the value in \texttt{pxv}, and \texttt{q} is the number of digit-positions following \{\texttt{v}\} in \texttt{ps} (\texttt{q}=0 if there is no \{\texttt{v}\} in \texttt{ps}).
Case 3. pic has associated arithmetic data-type with <mode> <complex>.

Step 3.1. Let picr be the <data-type> which is the same as pic except that it has <mode> <real>. Let ir be the associated character-string length of pic, that of picr is therefore 2*ir. Let vr, vi be <character-string-values> containing the first ir and last ir <character-values> of v, respectively.

Step 3.2. Perform validate-numeric-pictured-value(picr, vr) to obtain pvr, and perform validate-numeric-pictured-value(picr, vi) to obtain pvii.

Step 3.3.

Case 3.3.1. pvr has <picture-invalid>.

Return pvr.

Case 3.3.2. pvr has <picture-valid>, and pvii has <picture-invalid>.

Return a <picture-validity> <picture-invalid> <integer-values> n; where n is the sum of ir and the value in pvii.

Case 3.3.3. pvr and pvii both have <picture-valid>.

Return a <picture-validity> <picture-valid> cv; where cv is a <complex-values> with real and imaginary parts equal to the values in pvr and pvii, respectively.

9.5.2.9 Validity of a Field of a Numeric Pictured Value

Operation: validate-field-of-pictured-value(plist, c)

where plist is a numeric-pictured-element-list of a <fixed-point-picture>, <picture-mentioned>, or <picture-exponent>.

c is a <character-string-value> of length equal to the associated character-string length ofplist.

result: a <picture-validity> (Section 9.5.2.3).

Case 1. v is one of the values obtainable by normal return from performing edit-numeric-pictured-field(plist, d, n) where d is the <character-string-values> containing as many digits as there are digit-positions in plist, each digit independently takes each value 0 through 9 in turn, and n takes values +1 and -1 in turn.

Let ed and epg be the unique values of d and npg which edited to v. Let wx be the <real-values> which in the integer containing the same digits as v0.

Return a <picture-validity> <picture-valid> <real-values> wx*epgp.

Case 2. (Otherwise).

Let n be the lowest integer such that the first n <character-values> of v are different from the first n <character-values> of all the values obtainable from the editing operations in the predicate of Case 1.

Return a <picture-validity> <picture-invalid> <integer-values> n.
9.5.3 CHARACTER PICTURES

The operation validate-character-pictured-value defines the checking of a ⟨character-string-values⟩ for validity with respect to a ⟨data-type⟩ or ⟨format-item⟩ which contains ⟨picture-character⟩. This operation is invoked by operation valid-bif; by operation convert for a ⟨picture-character⟩ target-type, and by operation validate-input-format for a ⟨picture-format⟩.

The associated character-string length of a ⟨data-type⟩ or ⟨format-item⟩ containing ⟨picture-character⟩, or of a ⟨character-picture-element-list⟩ in the number of ⟨character-picture-element-list⟩ in its ⟨character-picture-element-list⟩.

Operation: validate-character-pictured-value⟨pic,v⟩

where pic is a ⟨data-type⟩ or a ⟨picture-format⟩ that has ⟨picture-character⟩,

v is a ⟨character-string-values⟩ whose length, n, is the associated character-string length of pic.

result: a ⟨picture-validity⟩ (Section 9.5.2).

Step 1. For i=1,...,n, in order, perform Step 1.1.

Step 1.1. Let pc be the i'th ⟨character-picture-element⟩ in pic. Let c be the i'th ⟨character-value⟩ in v. Perform test-char-pic-char⟨pc,c⟩ to obtain x. If x is ⟨false⟩, return a ⟨picture-validity⟩: ⟨picture-invalid⟩: ⟨integer-value⟩: i.

Step 2. Return a ⟨picture-validity⟩: ⟨picture-valid⟩.

9.5.3.1 Test-char-pic-char

Operation: test-char-pic-char⟨pc,c⟩

where pc is a ⟨character-picture-element⟩,

c is a ⟨character-value⟩.

result: ⟨true⟩ or ⟨false⟩

Case 1: pc has $\mathbb{D}$.

If c has a digit or a $\#$, return ⟨true⟩; otherwise, return ⟨false⟩.

Case 2: pc has $\mathbb{A}$.

If c has a letter or a $\#$, then return ⟨true⟩; otherwise, return ⟨false⟩.

Case 3: pc has $\mathbb{X}$.

Return ⟨true⟩.
Index

Entries in this index consist of operation names, category names, and other specially defined terms. Operation names and category names are cross-referenced, with the number of occurrences other than 1 on a page being indicated in parentheses. For operations and non-terminal category names, the number indicates the page of definition. For other specially defined terms, the numbers indicate where a definition may be found. Category names ending in "-list", "-consmalist", and "-designator" are indexed with the unsuffixed category.
The text in the image appears to be a page from a document, possibly a textbook or a technical manual, with dense formatting and numbers. Due to the complexity and density of the text, it is challenging to extract meaningful information without specialized OCR technology. It seems to include various sections and subsections, possibly discussing technical terms or instructions. Without clearer visibility or additional context, it is difficult to provide a coherent translation or summary of the content.