ECMA

EUROPEAN COMPUTER MANUFACTURERS ASSOCIATION

STANDARD ECMA - 50

PROGRAMMING LANGUAGE PL/1

December 1976

BRIEF HISTORY

In 1965 ECMA set up a new technical committee TC10 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/1.

In 1970 ANSI too set up a technical committee X3J1 for PL/1. It was decided that the two committees will work in common on a Joint PL/1 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/1 and to the maintenance of the Standard.

This Standard ECMA-50 has been accepted by the General Assembly of December 16, 1976.

FOREWORD

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

Tables

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 of this document 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 %INCLUDE construct may be used to include text from other sources during program translation.

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

Structural:	PROCEDURE ENTRY BEGIN DO END
Declarative:	DECLARE DEFAULT FORMAT
Flow of Control:	CALL RETURN IF GO TO Null Statement STOP
	ON, REVERT, SIGNAL } Interrupt Handling
Storage:	ALLOCATE FREE
	Assignment Statement

Input/Output:

- OPEN
- CLOSE
- GET, PUT — Stream I/O
- READ, WRITE, LOCATE, REWRITE, DELETE — 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 — or PICTURE
- BIT
- AREA
- ENTRY
- FILE
- FORMAT
- LABEL
- OFFSET
- POINTER

Values may be computed by expressions written using a specific set of operators and builtin 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.

1.1.2 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.

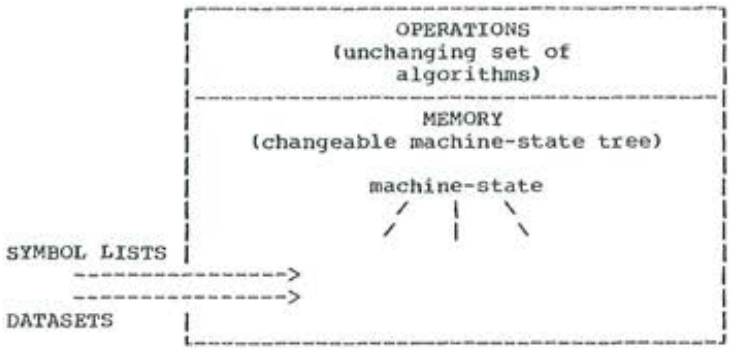
Sequences of symbols which are intended to represent PL/I external procedures (i.e. procedures not contained in any other procedure) are processed by a Translator. This processing involves systematically analyzing, transforming, and validating each external procedure. The analysis uses a grammar known 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 described in Section 1.2.

The method of definition may be seen in outline as follows:

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



The inputs from outside the machine occur at initialization of the Translator 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 7. STORAGE AND ASSIGNMENT 8. INPUT/OUTPUT	Concerned with the three parts of the interpretation-state
9. EXPRESSIONS	Common Subroutines for Chapters 5-8

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

The operations of the Translator 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 as 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 M,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 CM110, 111, 119, 122, and 123 reveals that an extra pair of parentheses is required around the form {input-target-commalist} 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),N);
```

or

```
GET LIST ((A(I,J) DO I = 1 TO M,N));
```

depending on whether the last input value is intended to be assigned to N or to A(N,J).

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 more subtle distinctions. The rule A117 for <keyto-option> shows " <target-reference> (scalar & character) ", and A179 shows that this permits a <pseudo-variable-reference>, e.g. the SUBSTR 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 PL/I 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> (A124), 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-value> 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 1.3.3.4 for the definition of "must".)

Additionally, in Step 2 of operation interpret (Section 5.3.1), an <evaluated-entry-reference> containing only an <entry-value> and not an <established-argument-list> (see production rule M25 in Section 5.1.2) 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 6 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 6.4.2.2) deletes a member from the current ‹established-on-unit-list› if an appropriate one exists, otherwise it performs normal-sequence to pass on to the next statement. Since the action of execute-on-statement is merely to append ‹established-on-unit›s 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 A(5) 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 4 while n≤nt. Since nt in this case is 1 (see Step 2) and n is initially 1 and is incremented in Step 4.7, the iteration is performed once only, thus initializing the first element A(1) only.

Chapter 8 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 RECORD; OPEN FILE(F) PRINT; valid?

ANSWER
Although most of the declarative structure 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 <print> implies <stream> in Step 2, and that <stream> 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 UNDEFINEDFILE condition when an attempt is made to open the file (see Step 6 of execute-single-opening, Section 8.5.1.2).

Chapter 9 describes the evaluation of expressions, also conversions between data types, which can take place in expression evaluation, and the builtin 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 9.1.1.4 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 X 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 "contain" 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 trees, namely, to test them for the presence or type of their subtrees, to select subtrees from them, and to construct or alter them. However, construction can be laborious if phrased as "root-node <a> with two immediate subtrees, the first being <b> 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

```
<a>:
  <b>:
    <c>
    <d>;
  x;
```

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.

1.1.4.2 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

<p>::= <q>|{<r-list>|<s>}•<t>[<u>]

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 <p> 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 single operands, and the first stage in using a rule to construct a tree is to partition the syntax expression into subexpressions separated by "|", 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

<t>[<u>]{<r-list>|<s>}

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

```
<p>:
  <t>
  <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> indicates 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.

1.1.4.3 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 metabrackets "<" and ">" are used in the PL/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.m, 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>s specify initialization. Any
<expression>s evaluated during the prologue, such as in <extent-expression>s or
<expression>s 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 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 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(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.

          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 edd.

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 prologue 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 latter. The word "current" has been defined precisely with respect to the particular ‹machine-state› used in defining PL/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 ",d". On subsequent uses of d, it is dereferenced, 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 let statement is merely a more explicit syntax for this kind of definition of a local variable, and Step 2.1 contains an instance of this. If d contains ‹automatic›, the predicate of Case 2.1.1 is satisfied and Steps 2.1.1.1 to 2.1.1.3 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. edd is passed as an argument to this operation, whose definition will name a parameter to correspond to it. 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 append 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 the usual implied dereferencing of the variable name, meaning "replace the tree designated by the strict designator value of tv by ‹true›".

Other statement types 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 have now reached a point at which it is necessary to indicate a mechanization of the metalanguage which will suffice to bolster intuition in its weaker moments. Step 4 provides the cue for this, since it constructs a tree which is to be placed in the ‹machine-state› at exactly the point at which it tails off into informality. Of the ‹operation-list›s 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 PL/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 PL/I machine are one or more {symbol-list}s representing PL/I external procedures, an <entry-value>, and a <dataset-list>. This combination will be referred to as a program-run.

The standard definition of PL/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-list}s, <entry-value>, and {dataset-list}) in the implementation's operating environment. With this information available, the conceptual PL/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 PL/I program-runs. These semantics are constituted solely by the sequence of changes in the <data-set>s of the <machine-state>, and an implementation is free to achieve them 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 "must" 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 PL/I machine permit most of the flexibility necessary for efficient implementation of PL/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 syntaxes allow 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 operations "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>, <value-reference>, <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 number 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, fixedoverflow, zerodivide, size, stringsize, stringrange, subscriptrange, storage, area, error.

1.2.1.6 Input/Output

(1) The <environment> attribute and option are provided for the specification of implementation-defined information concerned with the manipulation of the ‹file-value›s 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-value›s 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-io-condition(<undefinedfile-condition>,fv) operation performed by the execute-single-opening operation. In particular, an implementation may defer the raise-io-condition operation(s) 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-io-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 a part of file closing during 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>.

1.2.1.7 On-units

In order to avoid defining certain "side-effects" of <on-unit>s, and to avoid overly defining the state of the machine upon entry to <on-unit>s, a program must satisfy the following constraints not explicitly enforced by the PL/I machine:

(1) An <on-unit>, or any of its dynamic descendants, entered for the underflow, conversion, or stringsize condition must not allocate, free, or assign a value to any <allocation-unit> used in the block of interrupt, unless the <on-unit> terminates by executing a <goto-statement>.

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

(2) Let B be a <block-state>; let E be an <executable-unit> executed while B is the current <block-state>. During the execution of E, various <generation>s are returned by invocations of evaluate-target-reference. Let T be the set of <allocation-unit>s designated by these <generation>s, i.e., the targets of a given statement execution.

During the execution of E whenever the "interrupt" operation is invoked, all <basic-value>s contained in any member of T are set <undefined>. If control reaches Step 4 of the "interrupt" operation, the original <basic-value>s are restored, unless their containing <allocation-unit> was assigned one or more <basic-value>s by the <on-unit> or its dynamic descendants. In the case of such assignment, the <basic-value>s of the <allocation-unit> are set <undefined>.

1.2.2 IMPLEMENTATION-DEFINED FEATURES

The PL/I language features listed below are termed 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 TRANSMIT condition is raised (1.2.1.6).

(2) Actions performed, instead of raising the RECORD, KEY, or 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 (1.2.1.6, 8.5.2.3, and 8.6.6.9).

(3) Determination of the <dataset-list> passed to the "interpret" operation (1.4.3.3).

(4) Determination of the <entry-value> passed to the "interpret" operation (1.4.3.3).

(5) ENVIRONMENT attribute and option syntax (2.4.4.4) and semantics (Chapter 8).

(6) OPTIONS attribute and option syntax (2.4.4.4) and semantics.

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

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

(9) Collating sequence, hardware representations, graphic representations, and symbol names of an implementation's character set (2.6 and 9.4.4.17).

(10) Default precisions of arithmetic data (4.3.6.3).

(11) Default AREA size (4.3.6.3).

(12) Consistency requirements for ENVIRONMENT and OPTIONS attributes in EXTERNAL declarations (4.6.1), and for OPTIONS attributes in ENTRY references (6.3.6.1.1).

(13) Size of an <area-value> passed as a dummy argument (6.3.6.1.2).

(14) Information output when SNAP is specified in ON 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 UNSPEC pseudo-variable (7.5.4.8).

(23) Concrete representation of a <dataset> (8.1).

(24) The "size" of a <record> (8.1.1).

(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 LINESIZE 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 KEY condition is raised (8.5.2.3, 8.6.2.2, 8.6.3.1, and 8.6.6.10).

(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 (8.6.6.9).

(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 <base> and <scale> (4.4.3.5, 9.1.3.2, 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 (9.4.4.68).

(42) Length of string returned by TIME builtin function (9.4.4.82).

(43) Result of UNSPEC builtin function (9.4.4.85).

(44) Results of numeric conversions (9.5.1.2, 9.5.1.3, and 9.5.1.6).

(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 syntaxes 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) X is a subtree of Z, 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 usually elided. 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, "A simply contains B" may be expressed as "A with a B", "A has B", "the B in the A", or even just "B A". These abbreviated forms may also be compounded in a single 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 X follows Z and Z follows Y.

(3) There is a tree Z such that Z contains X and Z follows Y.

(4) There is a tree Z such that Z contains Y and X follows Z.

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 subnodes 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 is 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 thus 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 ordinals. 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 member 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 (or 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 denotations 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) "␣" 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 "LIKE" 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 ␣. Any possible subnodes of the category {extralingual-character} 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 subnode 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 comma and a name 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 trees which are 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 <absent>,

and a semicolon.

For example, the tree which consists of a <data-type> which immediately contains a <non-computational-type> which immediately contains <format> and <local>, can be written as: <data-type>,dt: <non-computational-type>: <format> <local>;;. Indentation is often used as a visual aid, e.g.

```
<data-type>,dt:
     <non-computational-type>:
          <format>
          <local>;;.
```

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

Use of an enumerated-tree in the metalanguage 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 set to designate the nodes there created.

A frequently used abbreviation for a particular class of enumerated-trees is to enclose a potential PL/I concrete-representation in double quotes. This is an abbreviation for that tree rooted in {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 metalanguage 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 metalanguage is always preceded by the word 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 1.3.1.4.1 are to be set to designate the nodes corresponding to them 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 1.3.2.3) or, in a few instances, an English Language phrase. Such a 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 "-commalist", and it substitutes for the explicit appearance of a defining production-rule for such category-names (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", "-commalist", 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.

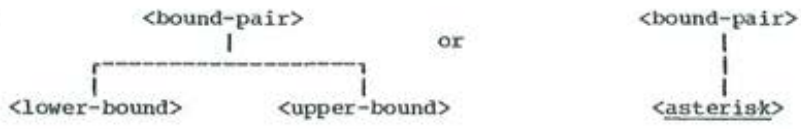
The production-rule

A17. <bound-pair>::= <lower-bound> <upper-bound> | <asterisk>

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

(a)
```
<bound-pair>:                          <bound-pair>:
    <lower-bound>          or              <asterisk>;
    <upper-bound>;
```

(b)



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

```
<identifier-list>:  or  <identifier-list>:  or  <identifier-list>:
    <identifier>;           <identifier>            <identifier>
                            <identifier>;           <identifier>
                                                    <identifier>;
```

and so on.

A node whose type is {parameter-commalist} may have any non-zero number of {parameter} immediate subnodes, but with nodes whose type is the PL/I character {,} interspersed between adjacent ones, i.e.

```
{parameter-commalist}:   or  {parameter-commalist}:   or  {parameter-commalist}:
    {parameter};                 {parameter}                  {parameter}
                                 {,}                          {,}
                                 {parameter};                 {parameter}
                                                              {,}
                                                              {parameter};
```

and so on.

The production-rule

A67. <repeat-option>::= <expression> (scalar)

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

Example 1.2. Examples of Syntax.

1.3.2.2 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 1.3.2.4, 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.

1.3.2.3 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 "|" or 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 "[" and "]", or

a syntactic-expression enclosed in the braces "(" and ")".

1.3.2.4 Application of the Production Rules

Given a syntax and a category-name, the algorithm shown just below obtains a (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 "-list".

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 "-list" from the denotation of the given category-name.

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

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 "-commalist" from the denotation of the given category-name, and which

(2) contains n-1 instances of the category-name {,}, 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 "-designator".

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

Case 4. There is in the given syntax a defining production-rule for the given category-name.

An interpretation is an interpretation of the syntactic-expression written following the "::=" in the defining production-rule.

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 as at least one 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 arranging 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 (a 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/I 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 type(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 <absent>.

(3) For an operation which returns a result, a final part of the heading is of the form "result:" followed by a description of the type(s) of tree which may be returned.

```
Operation:    evaluate-in-option(al,vr)

              where al is an <allocation>,
                    vr is a <variable-reference>.

              result: a <generation>.
```

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

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

Steps are normally performed sequentially.

Each 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 whenever 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 Steps and Cases may be modified by explicit instructions using such terms as "go to" or "perform" which are defined in Section 1.3.3.4. 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:    every-bif(rdd,x)

Step 1.   Perform evaluate-expression(x) to obtain an <aggregate-value>,u.

Step 2.   In any order, convert each scalar-element of u to <bit>, to obtain v.

Step 3.

  Case 3.1.   Every scalar-element of v that does not contain <null-bit-string> has
              a <bit-string-value> with every <bit-value> containing <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.
```

Example 1.4. An Example of an Operation Ccntaining a Step-list and a Case-set.

1.3.3.4 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 designated 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 use 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 uniquely specified, may be non-existent. This latter case causes the construction of the (single) 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:

*	denotes multiplication;
/	denotes division;
↑	denotes exponentiation;
$\sum$	denotes iterated addition (summation); the result is taken as zero if the iteration range is empty;
$\prod$	denotes iterated multiplication (product); the result is taken as one if the iteration range is empty;
[]	denotes subscripting in the metalanguage as a notational convenience used with local names;
pi	denotes the mathematical constant of that name;
e	denotes the mathematical constant of that name;
i	denotes the square root of -1;
ceil(x)	denotes the smallest integer larger than or equal to x;
floor(x)	denotes the largest integer smaller than or equal to x;
min(x,y)	denotes the value of x if x≤y, otherwise the value of y;
max(x,y)	denotes the value of x if x≥y, otherwise the value of y;
log(x)	denotes the natural logarithm of x;
|x|	denotes the absolute value of x.

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

```
<basic-value>:
    <complex-value>:
        <real-number>,x
        <real-number>,y;;
```

then the arithmetic expression "z+1" denotes the complex number (x+1)+iy.

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 algorithm which comprises the PL/I 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-control⟩s 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.)

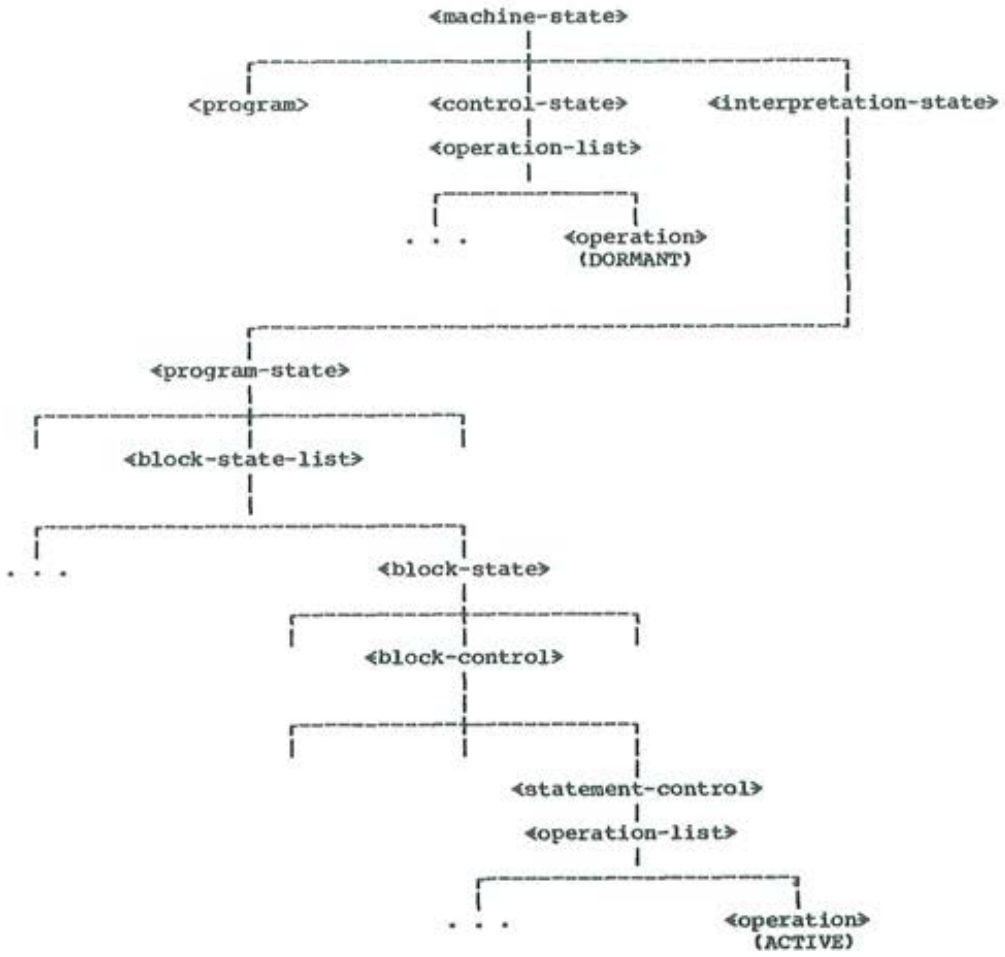
The processor carries out actions as follows:

> 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⟩ in 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-tree 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 <operation>s 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-list>s of a <machine-state>.

1.4 Initialization of the Machine-state

1.4.1 THE MACHINE-STATE

M1. ⟨machine-state⟩::= ⟨program⟩
⟨control-state⟩
[⟨translation-state⟩ | ⟨interpretation-state⟩]

M2. ⟨control-state⟩::= ⟨operation-list⟩

M3. ⟨translation-state⟩::= [{concrete-external-procedure}]

M4. {concrete-external-procedure}::= [{declaration-commalist}] {procedure}

M5. ⟨operation⟩ ::=

The exact structure of ⟨operation⟩ is left unformalized 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/I definition process begins by creating an initial ⟨machine-state⟩ tree, consisting of:

⟨machine-state⟩:
⟨program⟩
⟨control-state⟩:
⟨operation-list⟩:
⟨operation⟩ for 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.)

1.4.3.2 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 composing a putative PL/I external procedure, constructed in the form of a {symbol-list},sl.

Step 2.2. Perform translate(sl) to obtain an <abstract-external-procedure>,aep. Append aep 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›.

1.4.3.3 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›,dl.

(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(dl,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 4), 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 {procedure}s. 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 3).

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

```
CH1.   {procedure}::= {prefix-list} {procedure-statement} [{unit-list}] {ending}
```

2.3.2 UNIT

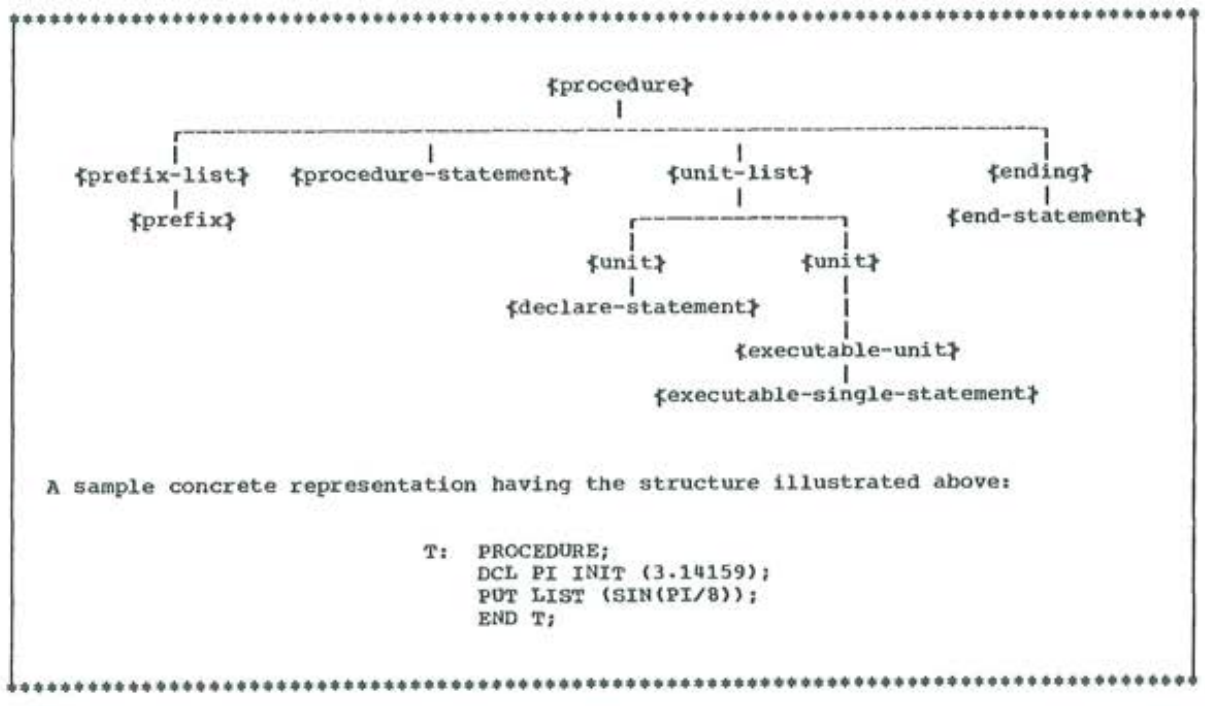
```
CH2.   {unit}::= [{statement-name-list}]  [{declare-statement} | {default-statement}] |
                 {statement-name-list} {entry-statement} |
                 {prefix-list} {format-statement} |
                 {procedure} |
                 {executable-unit}
```

2.3.3 EXECUTABLE UNITS

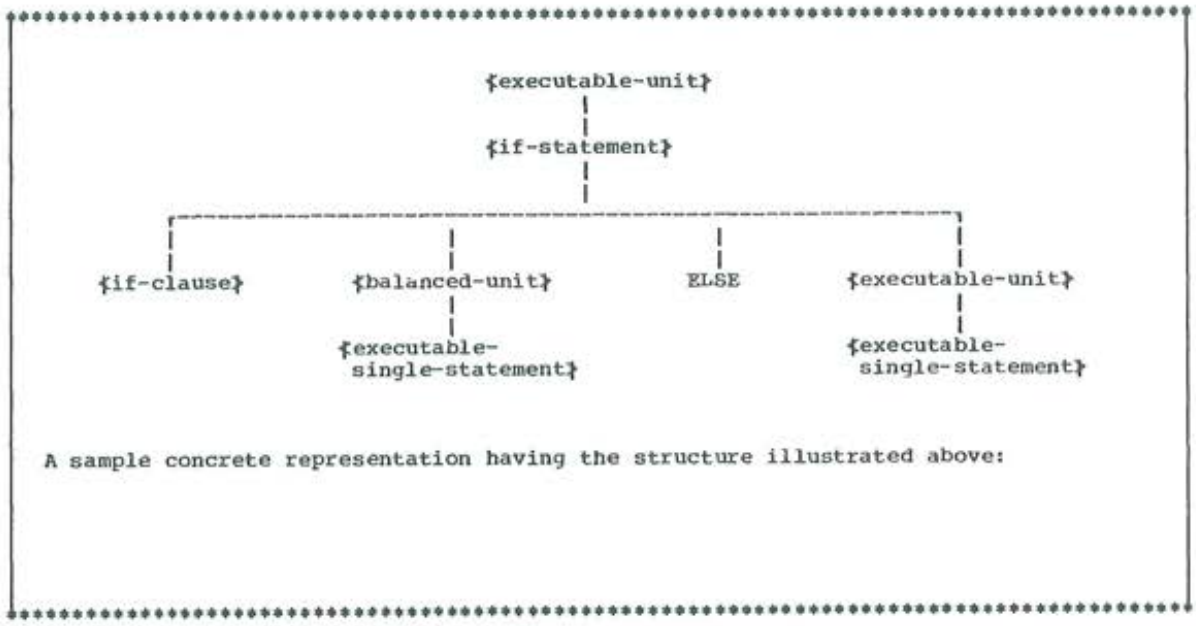
```
CH3.   {executable-unit}::= [{prefix-list}]
                             ({group} | {begin-block} |
                              {on-statement} | {if-statement} |
                              {executable-single-statement})

CH4.   {if-statement}::= {if-clause} ({executable-unit} |
                                      {balanced-unit} ELSE {executable-unit})

CH5.   {balanced-unit}::= [{prefix-list}]
                           ({executable-single-statement} | {group} | {begin-block} |
                            {on-statement} |
                            {if-clause} {balanced-unit} ELSE {balanced-unit})

CH6.   {group}::= {do-statement} [{unit-list}] {ending}

CH7.   {begin-block}::= {begin-statement} [{unit-list}] {ending}

CH8.   {on-statement}::= ON {condition-name-commalist} [SNAP] ({on-unit} | SYSTEM;)

CH9.   {on-unit}::= [({condition-prefix-commalist}):]
                    {executable-single-statement} | {begin-block})

CH10.  {ending}::= [{statement-name-list}] {end-statement}
```

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}.



Example 2.1. The High-level Structure of a Simple {procedure}.



Example 2.2. The High-level Structure of an {if-statement}.

2.4 The Middle-level Syntax of PL/I

2.4.1 SENTENCE

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

CM1. {sentence}::= [{prefixed-clause-list}] {single-statement} | {else-part}

CM2. {else-part}::= ELSE [{prefixed-clause-list}] {single-statement}

CM3. {prefixed-clause}::= [{prefix-list}]
({if-clause} |
ON {condition-name-commalist} [SNAP])

CM4. {prefix}::= ({condition-prefix-commalist}): | {statement-name}

CM5. {if-clause}::= IF {expression} THEN

2.4.2 STATEMENT

```
CM6.   {single-statement}::= [{statement-name-list}]
                             ({declare-statement} | {default-statement} |
                                                        {end-statement} ) |
                             {statement-name-list} {entry-statement} |
                             [{prefix-list}] ({executable-single-statement}|
                             {begin-statement} [{unmatched}] |
                             {do-statement} [{unmatched}] | {format-statement} |
                             SYSTEM;  | {procedure-statement} [{unmatched}])
```

Note: {unmatched} is used only by the operation high-level-parse.

```
CM7.   {executable-single-statement}::= {allocate-statement}    | {open-statement}    |
                                        {assignment-statement}  | {put-statement}     |
                                        {call-statement}        | {read-statement}    |
                                        {close-statement}       | {return-statement}  |
                                        {delete-statement}      | {revert-statement}  |
                                        {free-statement}        | {rewrite-statement} |
                                        {get-statement}         | {signal-statement}  |
                                        {goto-statement}        | {stop-statement}    |
                                        {locate-statement}      | {write-statement}   |
                                        {null-statement}
```

2.4.3 PREFIXES

2.4.3.1 Condition Prefixes

```
CM8.   {condition-prefix}::= {computational-condition} |
                             {disabled-computational-condition}

CM9.   {computational-condition}::= CONVERSION | FIXEDOVERFLOW | OVERFLOW | SIZE |
                                    STRINGRANGE | STRINGSIZE | SUBSCRIPTRANGE |
                                    UNDERFLOW | ZERODIVIDE

CM10.  {disabled-computational-condition}::= NOCONVERSION | NOFIXEDOVERFLOW |
                                             NOOVERFLOW | NOSIZE | NOSTRINGRANGE |
                                             NOSTRINGSIZE | NOSUBSCRIPTRANGE |
                                             NOUNDERFLOW | NOZERODIVIDE
```

2.4.3.2 Statement Name Prefixes

```
CM11.  {statement-name}::= {identifier} [({signed-integer-commalist})]:

CM12.  {signed-integer}::= [+ | -] {integer}
```

2.4.4 DATA DECLARATION

```
CM13.  {declare-statement}::= DECLARE {declaration-commalist};

CM14.  {declaration}::= [{level}]   [{identifier} | ({declaration-commalist})]
                        [{dimension-suffix}] [{attribute-list}]

CM15.  {level}::= {integer}
```

2.4.4.1 Dimension Attribute and Dimension Suffix

CM16. {dimension-attribute}::= DIMENSION [{dimension-suffix}]

CM17. {dimension-suffix}::= ({bound-pair-commalist})

CM18. {bound-pair}::= [{lower-bound}:] {upper-bound} | *

CM19. {lower-bound}::= {extent-expression}

CM20. {upper-bound}::= {extent-expression}

CM21. {extent-expression}::= {expression} [{refer-option}]

CM22. {refer-option}::= REFER ({unsubscripted-reference})

2.4.4.2 Attributes

CM23. {attribute}::=

{data-attribute}	KEYED
AUTOMATIC	LIKE {unsubscripted-reference}
BASED [({reference})]	LOCAL
BUILTIN	{options}
CONDITION	OUTPUT
CONSTANT	PARAMETER
CONTROLLED	POSITION [({expression})]
DEFINED ({reference} | ({reference}))	PRINT
DIRECT	RECORD
{environment}	SEQUENTIAL
EXTERNAL	STATIC
{generic-attribute}	STREAM
{initial}	UPDATE
INPUT	VARIABLE
INTERNAL	

2.4.4.3 Data Attributes

CM24. {data-attribute}::=

ALIGNED	LABEL
AREA [({area-size})]	MEMBER
BINARY [{precision}]	NONVARYING
BIT [({maximum-length})]	OFFSET [({reference})]
CHARACTER [({maximum-length})]	PICTURE [{picture}]
COMPLEX [{precision}]	POINTER
DECIMAL [{precision}]	PRECISION [{precision}]
{dimension-attribute}	REAL [{precision}]
ENTRY [([{description-commalist}])]	{returns-descriptor}
FILE	STRUCTURE
FIXED [{precision}]	UNALIGNED
FLOAT [({number-of-digits})]	VARYING
FORMAT	

CM25. {area-size}::= {extent-expression} | *

CM26. {precision}::= ({number-of-digits} [,{scale-factor}])

CM27. {number-of-digits}::= {integer}

CM28. {scale-factor}::= {signed-integer}

CM29. {maximum-length}::= {extent-expression} | *

CM30. {description}::= [{level}] [{dimension-suffix}] [{data-attribute-list}]

Constraint: At least one subnode must be present.

CM31. {picture}::= {simple-character-string-constant}

2.4.4.4 Environment and Options

```
CM32.  {environment}::= ENVIRONMENT ({environment-specification})

CM33.  {environment-specification}::=

CM34.  {options}::= OPTIONS ({options-specification})

CM35.  {options-specification}::=
```

2.4.4.5 Generic

```
CM36.  {generic-attribute}::= GENERIC [({generic-element-commalist})]

CM37.  {generic-element}::= {reference} WHEN (({generic-description-commalist}))

CM38.  {generic-description}::= [{level}] [{asterisk-bounds}]
                                [{generic-data-attribute-list}] | *

       Constraint:  At least one subnode must be present.

CM39.  {generic-data-attribute}::= ALIGNED                          | LABEL                  |
                                   AREA                             | MEMBER                 |
                                   BINARY [{generic-precision}]     | NONVARYING             |
                                   BIT                              | OFFSET                 |
                                   CHARACTER                        | PICTURE {picture}      |
                                   COMPLEX [{generic-precision}]    | POINTER                |
                                   DECIMAL [{generic-precision}]    | PRECISION {generic-
                                                                                precision} |
                                   DIMENSION {asterisk-bounds}      | REAL [{generic-
                                                                                precision}] |
                                   ENTRY [([{description-
                                                        commalist}])] | {returns-descriptor} |
                                   FILE                             | STRUCTURE              |
                                   FIXED [{generic-precision}]      | UNALIGNED              |
                                   FLOAT [{generic-precision}]      | VARYING                |
                                   FORMAT

CM40.  {asterisk-bounds}::= ({*-commalist})

CM41.  {generic-precision}::= ({number-of-digits} [:{number-of-digits}]
                              [,{scale-factor} [:{scale-factor}]])
```

2.4.4.6 Initial

```
CM42.  {initial}::= INITIAL [({initial-element-commalist})]

CM43.  {initial-element}::= * | {parenthesized-expression} |
                           ({iteration-factor}) [{initial-constant-two} | * |
                                                 ({initial-element-commalist})) |
                           {initial-constant-one}

CM44.  {initial-constant-one}::= [{prefix-operator}] {simple-string-constant} |
                                 {initial-constant-two}

CM45.  {initial-constant-two}::= [{prefix-operator}]
                                 {reference} | {replicated-string-constant} |
                                  {arithmetic-constant}} |
                                 [+ | -] {real-constant} [+ | -] {imaginary-constant}

CM46.  {iteration-factor}::= {expression}
```

2.4.4.7 The Default Statement

CM47. {default-statement}::= DEFAULT ({default-specification} | NONE | SYSTEM);

CM48. {default-specification}::= ({predicate-expression})
{ERROR | {default-attributes-commalist}}

CM49. {predicate-expression}::= {predicate-expression-three} |
{predicate-expression} {|} {predicate-expression-three}

CM50. {predicate-expression-three}::= {predicate-expression-two} |
{predicate-expression-three} &
{predicate-expression-two}

CM51. {predicate-expression-two}::= {predicate-expression-one} |
¬ {predicate-expression-two}

CM52. {predicate-expression-one}::= ({predicate-expression}) | {attribute-keyword} |
{range-specification}

CM53. {range-specification}::= RANGE (({identifier} | {letter} : {letter} | *))

CM54. {attribute-keyword}::=

ALIGNED	DEFINED	INTERNAL	PRECISION
AREA	DIMENSION	KEYED	PRINT
AUTOMATIC	DIRECT	LABEL	REAL
BASED	ENTRY	LOCAL	RECORD
BINARY	ENVIRONMENT	MEMBER	RETURNS
BIT	EXTERNAL	NONVARYING	SEQUENTIAL
BUILTIN	FILE	OFFSET	STATIC
CHARACTER	FIXED	OPTIONS	STREAM
COMPLEX	FLOAT	OUTPUT	STRUCTURE
CONDITION	FORMAT	PARAMETER	UNALIGNED
CONSTANT	GENERIC	PICTURE	UPDATE
CONTROLLED	INITIAL	POINTER	VARIABLE
DECIMAL	INPUT	POSITION	VARYING

CM55. {default-attributes}::= {attribute-list}

2.4.5 THE PROCEDURE STATEMENT

CM56. {procedure-statement}::= PROCEDURE [{entry-information}];

CM57. {entry-information}::= [({parameter-name-commalist})]
([{returns-descriptor}] • [{options}] • [RECURSIVE])

CM58. {parameter-name}::= {identifier}

CM59. {returns-descriptor}::= RETURNS [({description-commalist})]

2.4.6 THE ENTRY STATEMENT

CM60. {entry-statement}::= ENTRY [{entry-information}];

2.4.7 THE BEGIN STATEMENT

CM61. {begin-statement}::= BEGIN [{options}];

2.4.8 THE DO STATEMENT

CM62. {do-statement}::= DO; | DO {while-option}; | DO {do-spec};

CM63. {do-spec}::= {reference} = {spec-commalist}

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}::= [GOTO | 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-commalist};

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

CM82. {assignment-statement}::= {reference-commalist} = {expression} [,BY NAME];

CM83. {allocate-statement}::= ALLOCATE {allocation-commalist};

CM84. {allocation}::= {identifier} [({set-option}) • [{in-option}]]

CM85. {set-option}::= SET ({reference})

CM86. {in-option}::= IN ({reference})

CM87. {free-statement}::= FREE {freeing-commalist};

CM88. {freeing}::= [{locator-qualifier}] {identifier} [{in-option}]

2.4.12 INPUT/OUTPUT STATEMENTS

2.4.12.1 The Open and Close Statements

CM89. {open-statement}::= OPEN {single-opening-commalist};

CM90. {single-opening}::= {file-option} • [{tab-option}] • [{title-option}] •
[{linesize-option}] • [{pagesize-option}] •
[STREAM] • [RECORD] • [INPUT] • [OUTPUT] • [UPDATE] •
[SEQUENTIAL] • [DIRECT] • [PRINT] • [KEYED] •
[{environment}]

CM91. {file-option}::= FILE ({reference})

CM92. {tab-option}::= TAB ({expression-commalist})

CM93. {title-option}::= TITLE ({expression})

CM94. {linesize-option}::= LINESIZE ({expression})

CM95. {pagesize-option}::= PAGESIZE ({expression})

CM96. {close-statement}::= CLOSE {single-closing-commalist};

CM97. {single-closing}::= {file-option} • [{environment}]

2.4.12.2 Record I/O

CM98. {delete-statement}::= DELETE [{file-option} • [{key-option}]];

CM99. {locate-statement}::= LOCATE {identifier} [{file-option} • [{pointer-set-option}]•
[{keyfrom-option}]];

CM100. {pointer-set-option}::= SET ({reference})

CM101. {read-statement}::= READ [{file-option} • [{into-option} | {pointer-set-option} |
{ignore-option}] •
[{key-option} | {keyto-option}]];

CM102. {into-option}::= INTO ({reference})

CM103. {ignore-option}::= IGNORE ({expression})

CM104. {key-option}::= KEY ({expression})

CM105. {keyto-option}::= KEYTO ({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}::= KEYFROM ({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}] • [PAGE] •
[{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-commalist})]

CM121. {data-target}::= {unsubscripted-reference}

CM122. {list-directed-input}::= LIST ({input-target-commalist})

CM123. {input-target}::= {reference} | ({input-target-commalist} DO {do-spec})

CM124. {edit-directed-input}::= EDIT {edit-input-pair-list}

CM125. {edit-input-pair}::= ({input-target-commalist}) ({format-specification-commalist})

2.4.12.3.2 Stream Output Specification

```
CM126. {output-specification}::= {data-directed-output} | {list-directed-output} |
                                 {edit-directed-output}

CM127. {data-directed-output}::= DATA [({data-source-commalist})]

CM128. {data-source}::= {reference} |
                        ({data-source-commalist} DO {do-spec})

CM129. {list-directed-output}::= LIST ({output-source-commalist})

CM130. {output-source}::= {expression} | ({output-source-commalist} DO {do-spec})

CM131. {edit-directed-output}::= EDIT {edit-output-pair-list}

CM132. {edit-output-pair}::= ({output-source-commalist})
                             ({format-specification-commalist})
```

2.4.12.3.3 Format Specification Lists and the Format Statement

```
CM133. {format-specification}::= {format-item} | {format-iteration}

CM134. {format-iteration}::= {format-iteration-factor}
                             ({format-item} | ({format-specification-commalist}))

CM135. {format-iteration-factor}::= {integer} | ({expression})

CM136. {format-item}::= {data-format} | {control-format} | {remote-format}

CM137. {data-format}::= {real-format} | {complex-format} | {picture-format} |
                        {string-format}

CM138. {real-format}::= {fixed-point-format} | {floating-point-format}

CM139. {fixed-point-format}::= F ({expression} [,{expression} [,{expression}]])

CM140. {floating-point-format}::= E ({expression} [,{expression} [,{expression}]])

CM141. {complex-format}::= C (({real-format} | {picture-format})
                              [,{real-format} | ,{picture-format}])

CM142. {picture-format}::= P {picture}

CM143. {string-format}::= {character-format} | {bit-format}

CM144. {character-format}::= A [({expression})]

CM145. {bit-format}::= {radix-factor} [({expression})]

CM146. {control-format}::= {tab-format} | {line-format} | {space-format} |
                           {skip-format} | {column-format} | PAGE

CM147. {tab-format}::= TAB [({expression})]

CM148. {line-format}::= LINE ({expression})

CM149. {space-format}::= X ({expression})

CM150. {skip-format}::= SKIP [({expression})]

CM151. {column-format}::= COLUMN ({expression})

CM152. {remote-format}::= R ({reference})

CM153. {format-statement}::= FORMAT ({format-specification-commalist});
```

2.4.13 EXPRESSIONS

CM154. {expression}::= {expression-seven} | {expression} {|} {expression-seven}

CM155. {expression-seven}::= {expression-six} | {expression-seven} & {expression-six}

CM156. {expression-six}::= {expression-five} |
{expression-six} {comparison-operator} {expression-five}

CM157. {comparison-operator}::= > | >= | = | < | <= | ¬> | ¬= | ¬<

CM158. {expression-five}::= {expression-four} | {expression-five} {||} {expression-four}

CM159. {expression-four}::= {expression-three} |
{expression-four} (+ | -) {expression-three}

CM160. {expression-three}::= {expression-two} |
{expression-three} (* | /) {expression-two}

CM161. {expression-two}::= {primitive-expression} | {prefix-expression} |
{parenthesized-expression} | {expression-one}

CM162. {expression-one}::= ({primitive-expression} | {parenthesized-expression})
** {expression-two}

CM163. {prefix-expression}::= {prefix-operator} {expression-two}

CM164. {prefix-operator}::= + | - | ¬

CM165. {parenthesized-expression}::= ({expression})

CM166. {primitive-expression}::= {reference} | {constant} | {isub}

CM167. {reference}::= [{locator-qualifier}] {basic-reference} [{arguments-list}]

CM168. {locator-qualifier}::= {reference} ->

CM169. {arguments}::= ([{subscript-commalist}])

CM170. {basic-reference}::= [{structure-qualification}] {identifier}

CM171. {structure-qualification}::= {basic-reference} [{arguments}].

CM172. {subscript}::= {expression} | *

CM173. {unsubscripted-reference}::= [{unsubscripted-reference} .] {identifier}

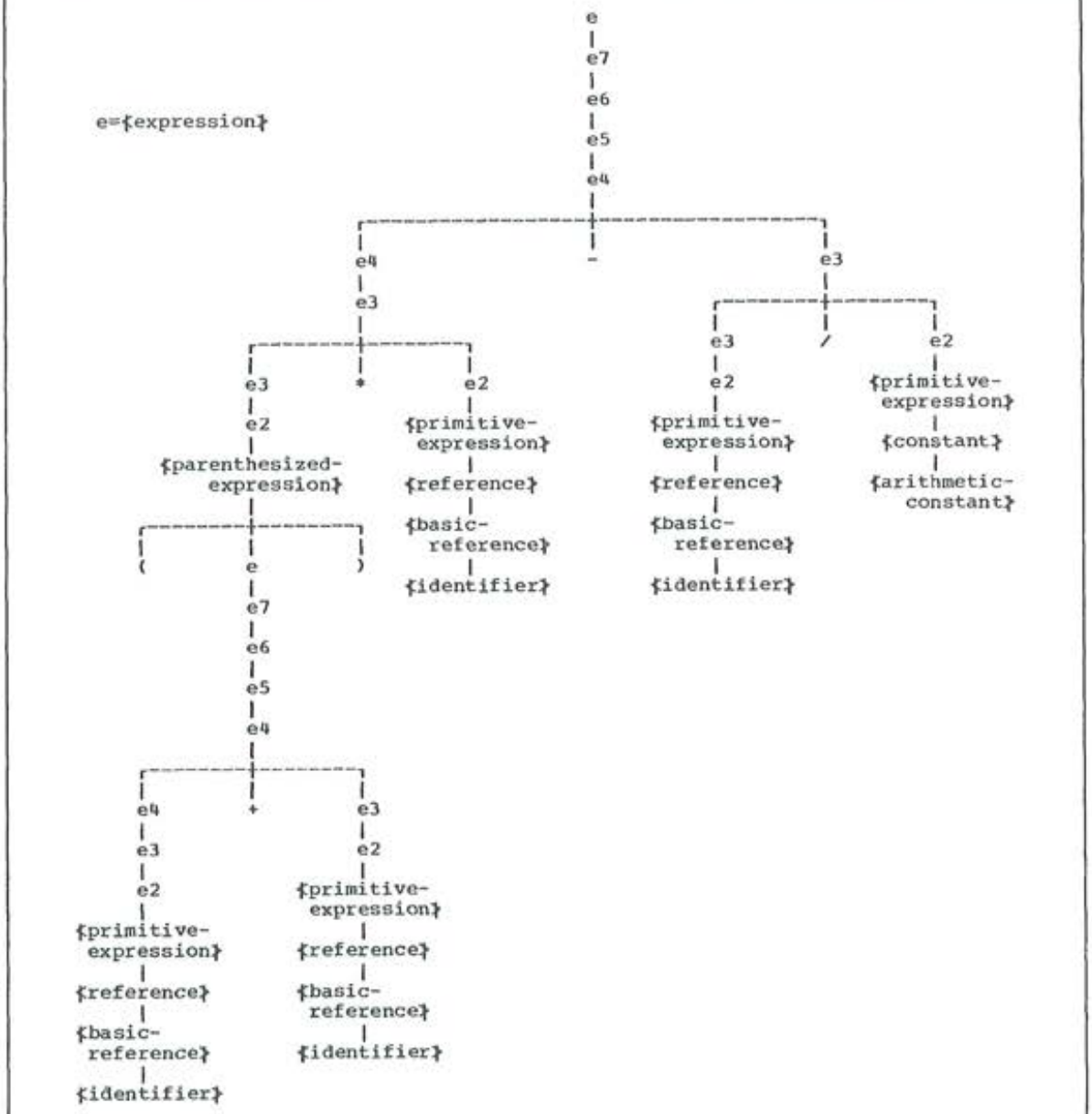
CM174. {constant}::= {arithmetic-constant} | {string-constant}

CM175. {string-constant}::= {simple-string-constant} | {replicated-string-constant}

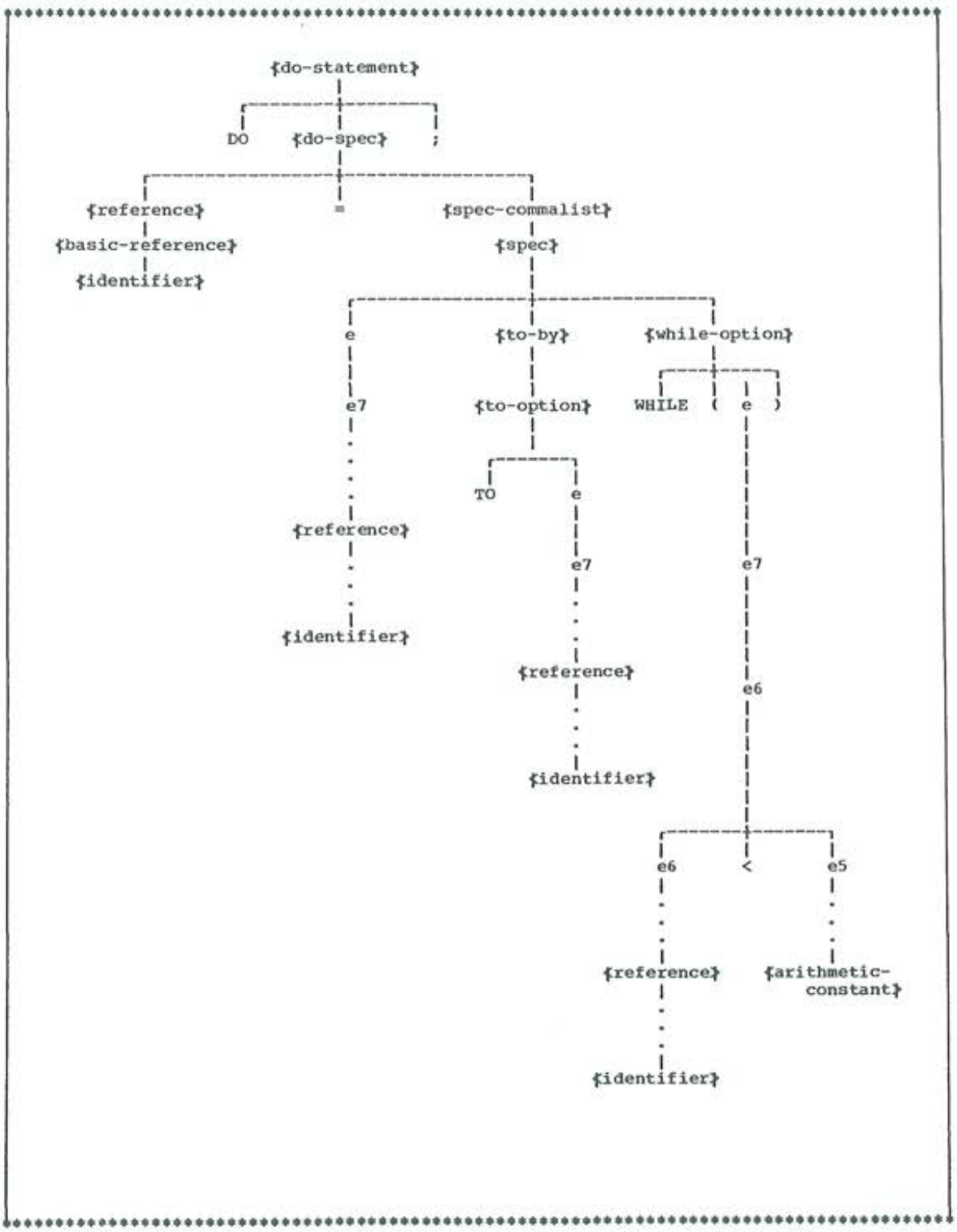
CM176. {simple-string-constant}::= {simple-character-string-constant} |
{simple-bit-string-constant}

CM177. {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⟩.



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}::= + | - | * | / | ** | > | < | = | >= | <= | ¬> | ¬< | ¬= | ¬ |
& | {|} | {||} | (|) | . | , | ; | : | -> | % | {comment} |
{include}

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

2.5.2 COMMENT

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

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

CL7. {comment-character}::= {letter} | {digit} | . | % | ' | = | + | - | (|) | , |
_ | $ | ; | : | & | {|} | ¬ | > | < | % |
{extralingual-character}

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 IDENTIFIER

CL8. {identifier}::= {letter} | {identifier} [{letter} | {digit} | _]

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}::= [+ | -] {integer}

CL18. {binary-constant}::= {binary-number} [{scale-type} {exponent}] B [P]

CL19. {binary-number}::= {binary-digit-list} [.[{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}::= '[{string-or-picture-symbol-list}]' {radix-factor}

CL23. {radix-factor}::= B|B1|B2|B3|B4

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 ISUB

CL27. {isub}::= {integer} SUB

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}s 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 |
. | ¤ | ' | = | + | - | * | / | (|) | , | _ | $ | ; | : | & | {|} |
¬ | < | > | % | {extralingual-character}

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.

2.6.1.1 Letters and Digits

Name	Graphic
Letter A	A
Letter B	B
Letter C	C
Letter D	D
Letter E	E
Letter F	F
Letter G	G
Letter H	H
Letter I	I
Letter J	J
Letter K	K
Letter L	L
Letter M	M
Letter N	N
Letter O	O
Letter P	P
Letter Q	Q
Letter R	R
Letter S	S
Letter T	T
Letter U	U
Letter V	V
Letter W	W
Letter X	X
Letter Y	Y
Letter Z	Z

Name	Graphic
Digit 0	0
Digit 1	1
Digit 2	2
Digit 3	3
Digit 4	4
Digit 5	5
Digit 6	6
Digit 7	7
Digit 8	8
Digit 9	9

2.6.1.2 Special Characters

Name	Graphic
Plus	+
Minus	-
Asterisk	*
Slash	/
Greater than	>
Less than	<
Equal	=
Not	¬
And	&
Or	|
Percent	%

Name	Graphic
Period	.
Comma	,
Semicolon	;
Colon	:
Blank	␣
Single quote	'
Left parenthesis	(
Right parenthesis	)
Break	
Dollar	$

2.6.2 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}s.

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 name scopes. The abbreviations are shown to the right of the full denotations in the following list.

ALLOCATE	ALLOC	NOOVERFLOW	NOOFL
ALLOCATION	ALLOCN	NOSTRINGRANGE	NOSTRG
AUTOMATIC	AUTO	NOSTRINGSIZE	NOSTRZ
BINARY	BIN	NOSUBSCRIPTRANGE	NOSUBRG
CHARACTER	CHAR	NOUNDERFLOW	NOUFL
COLUMN	COL	NOZERODIVIDE	NOZDIV
COMPLEX	CPLX	OVERFLOW	OFL
CONDITION	COND	PARAMETER	PARM
CONTROLLED	CTL	PICTURE	PIC
CONVERSION	CONV	POINTER	PTR
DECIMAL	DEC	POSITION	POS
DECLARE	DCL	PRECISION	PREC
DEFAULT	DFT	PROCEDURE	PROC
DEFINED	DEF	SEQUENTIAL	SEQL
DIMENSION	DIM	STRINGRANGE	STRG
ENVIRONMENT	ENV	STRINGSIZE	STRZ
EXTERNAL	EXT	SUBSCRIPTRANGE	SUBRG
FIXEDOVERFLOW	FOFL	UNALIGNED	UNAL
INITIAL	INIT	UNDERFLOW	UFL
INTERNAL	INT	UNDEFINEDFILE	UNDF
NOCONVERSION	NOCONV	VARYING	VAR
NOFIXEDOVERFLOW	NOFOFL	ZERODIVIDE	ZDIV
NONVARYING	NONVAR		

Chapter 3: Abstract Syntax

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>s, 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>::= [<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> | <builtin> | <condition>

3.1.4 VARIABLE

A7. <variable>::= <storage-type> <data-description>

A8. <storage-type>::= <storage-class> | <defined> | <parameter>

A9. <storage-class>::= <automatic> | <based> | <controlled> | <static>

A10. <based>::= [<value-reference> (scalar & locator) | {reference-designator}]

A11. <defined>::= <base-item> [<position>]

A12. <base-item>::= <variable-reference> (¬defined & ¬based) |
{reference-designator}

A13. <position>::= <expression> (scalar & computational-type) | {expression-designator}

3.1.5 DATA-DESCRIPTION

A14. <data-description>::= <dimensioned-data-description> |
<structure-data-description> | <item-data-description>

A15. <dimensioned-data-description>::= <element-data-description> <bound-pair-list>

A16. <element-data-description>::= <structure-data-description> |
<item-data-description>

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> | <pictured>

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-factor>]

A37. <number-of-digits>::= <integer>

Constraint: The <integer> must not be zero.

A38. <scale-factor>::= <signed-integer>

A39. <string>::= <string-type> <maximum-length> [<varying> | <nonvarying>]

A40. <string-type>::= <character> | <bit>

A41. <maximum-length>::= <extent-expression> | <asterisk>

A42. <pictured>::= <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>::= [<stream>] [<record>] [<input>] [<output>] [<update>]
[<sequential>] [<direct>] [<print>] [<keyed>] [<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>

A56. <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>]
[<begin-block> | <group> |
<allocate-statement> | <null-statement> |
<assignment-statement> | <on-statement> |
<call-statement> | <open-statement> |
<close-statement> | <put-statement> |
<delete-statement> | <read-statement> |
<end-statement> | <return-statement> |
<free-statement> | <revert-statement> |
<get-statement> | <rewrite-statement> |
<goto-statement> | <signal-statement> |
<if-statement> | <stop-statement> |
<locate-statement> | <write-statement>]

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> [<snap>] [<on-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-io-condition> |
<programmer-named-condition> | <area-condition> |
<error-condition> | <finish-condition> | <storage-condition>

A85. <condition-prefix>::= <computational-condition> [<enabled> | <disabled>]

A86. <computational-condition>::= <conversion-condition> | <fixedoverflow-condition> |
<overflow-condition> | <size-condition> |
<stringrange-condition> | <stringsize-condition> |
<subscriptrange-condition> | <underflow-condition> |
<zerodivide-condition>

A87. <named-io-condition>::= <io-condition> <value-reference> (scalar & file)

A88. <io-condition>::= <endfile-condition> | <endpage-condition> | <key-condition> |
<name-condition> | <record-condition> | <transmit-condition> |
<undefinedfile-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)

Constraint: 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>]

3.1.15 I/O STATEMENTS

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

A98. <single-opening>::= <file-option> [<tab-option>] [<title-option>]
[<linesize-option>] [<pagesize-option>] [<stream>] [<record>]
[<input>] [<output>] [<update>] [<sequential>] [<direct>]
[<print>] [<keyed>] [<environment>]

A99. <file-option>::= <value-reference> (scalar & file)

A100. <tab-option>::= <expression-list> (scalar & computational-type)

A101. <title-option>::= <expression> (scalar & computational-type)

A102. <linesize-option>::= <expression> (scalar & computational-type)

A103. <pagesize-option>::= <expression> (scalar & computational-type)

A104. <close-statement>::= <single-closing-list>

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

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 & pointer)

A109. <read-statement>::= <file-option>
[<into-option> | <pointer-set-option> | <ignore-option>]
[<key-option> | <keyto-option>]

A110. <into-option>::= <variable-reference>

A111. <ignore-option>::= <expression> (scalar & computational-type)

A112. <rewrite-statement>::= <file-option>
[[<key-option>] <from-option>]

A113. <write-statement>::= <file-option> <from-option>
[<keyfrom-option>]

A114. <from-option>::= <variable-reference>

A115. <key-option>::= <expression> (scalar & computational-type)

A116. <keyfrom-option>::= <expression> (scalar & computational-type)

A117. <keyto-option>::= <target-reference> (scalar & character)

3.1.17 STREAM I/O STATEMENTS

A118. <get-statement>::= <get-file> | <get-string>

A119. <get-file>::= <file-option> [<copy-option>] [<skip-option>] [<input-specification>]

Constraint: At least one of the last two options must be present.

A120. <skip-option>::= <expression> (scalar & computational-type)

A121. <copy-option>::= <value-reference> (scalar & file)

A122. <get-string>::= <expression> (scalar & computational-type)
<input-specification> [<copy-option>]

A123. <put-statement>::= <put-file> | <put-string>

A124. <put-file>::= <file-option> [<skip-option>] [<line-option>] [<page>]
[<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 <page>.

A125. <line-option>::= <expression> (scalar & computational-type)

A126. <put-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-parts-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>)]]]

A148. <floating-point-format>::= [<expression> (scalar & computational-type) |
                                 <integer-value>]
                                 [[<expression> (scalar & computational-type) |
                                 <integer-value>]
                                 [[<expression> (scalar & computational-type) |
                                 <integer-value>]]]

A149. <complex-format>::= [<real-format> | <picture-format>]
                          [<real-format> | <picture-format>]

Constraint: A <complex-format> must not contain <pictured-character>.

A150. <picture-format>::= <pictured>

A151. <string-format>::= <character-format> | <bit-format>

A152. <character-format>::= [[<expression> (scalar & computational-type) |
                              <integer-value>]]

A153. <bit-format>::= <radix-factor> [[<expression> (scalar & computational-type) |
                      <integer-value>]]

A154. <radix-factor>::= 1 | 2 | 3 | 4

A155. <control-format>::= <tab-format> | <line-format> | <space-format> | <skip-format> |
                          <column-format> | <page>

A156. <tab-format>::= [<expression> (scalar & computational-type) | <integer-value>]

A157. <line-format>::= [<expression> (scalar & computational-type) | <integer-value>]

A158. <space-format>::= [<expression> (scalar & computational-type) | <integer-value>]

A159. <skip-format>::= [<expression> (scalar & computational-type) | <integer-value>]

A160. <column-format>::= [<expression> (scalar & computational-type) | <integer-value>]

A161. <remote-format>::= <variable-reference> (scalar & format) |
                         <named-constant-reference> (scalar & format)

A162. <format-statement>::= [<condition-prefix-list>]
                            <statement-name-list> <format-specification-list>

3.1.18 EXPRESSION

A163. <expression>::= [<value-reference> | <constant> | <isub> |
                       <infix-expression> | <prefix-expression> |
                       <parenthesized-expression>] <data-description>

A164. <infix-expression>::= <expression> <infix-operator> <expression> <data-description>

A165. <infix-operator>::= <or> | <and> | <gt> | <ge> | <eq> | <le> | <lt> | <ne> |
                          <cat> | <add> | <subtract> | <multiply> | <divide> | <power>

A166. <prefix-expression>::= <prefix-operator> <expression> <data-description>

A167. <prefix-operator>::= <plus> | <minus> | <not>

A168. <parenthesized-expression>::= <expression> <data-description>

3.1.19 TYPES OF REFERENCE

A169. <value-reference>::= {<variable-reference> | <procedure-function-reference> |
<builtin-function-reference> | <named-constant-reference>}
<data-description>

A170. <variable-reference>::= [<locator-qualifier>] <declaration-designator> (variable)
[<identifier-list>] [<subscript-list>]
[<by-name-parts-list>] <data-description>

A171. <by-name-parts>::= <identifier-list>

A172. <locator-qualifier>::= <value-reference> (scalar & locator)

A173. <subscript>::= <expression> (scalar & computational-type) | <asterisk> |
<integer-value>

A174. <procedure-function-reference>::= <value-reference> [<argument-list>]
<data-description>

A175. <argument>::= <expression> [<dummy>] <data-description>

A176. <builtin-function-reference>::= <builtin-function> [<argument-list>]
<data-description>

A177. <builtin-function>::=

<abs-bif>	<empty-bif>	<onkey-bif>
<acos-bif>	<erf-bif>	<onloc-bif>
<add-bif>	<erfc-bif>	<onsource-bif>
<addr-bif>	<every-bif>	<pageno-bif>
<after-bif>	<exp-bif>	<pointer-bif>
<allocation-bif>	<fixed-bif>	<precision-bif>
<asin-bif>	<float-bif>	<prod-bif>
<atan-bif>	<floor-bif>	<real-bif>
<atand-bif>	<hbound-bif>	<reverse-bif>
<atanh-bif>	<high-bif>	<round-bif>
<before-bif>	<imag-bif>	<sign-bif>
<binary-bif>	<index-bif>	<sin-bif>
<bit-bif>	<lbound-bif>	<sind-bif>
<bool-bif>	<length-bif>	<sinh-bif>
<ceil-bif>	<lineno-bif>	<some-bif>
<character-bif>	<log-bif>	<sqrt-bif>
<collate-bif>	<log10-bif>	<string-bif>
<complex-bif>	<log2-bif>	<substr-bif>
<conjg-bif>	<low-bif>	<subtract-bif>
<copy-bif>	<max-bif>	<sum-bif>
<cos-bif>	<min-bif>	<tan-bif>
<cosd-bif>	<mod-bif>	<tand-bif>
<cosh-bif>	<multiply-bif>	<tanh-bif>
<date-bif>	<null-bif>	<time-bif>
<decat-bif>	<offset-bif>	<translate-bif>
<decimal-bif>	<onchar-bif>	<trunc-bif>
<dimension-bif>	<oncode-bif>	<unspec-bif>
<divide-bif>	<onfield-bif>	<valid-bif>
<dot-bif>	<onfile-bif>	<verify-bif>

A178. <named-constant-reference>::= <declaration-designator> (named-constant)
[<subscript-list>] <data-description>

A179. <target-reference>::= {<variable-reference> | <pseudo-variable-reference>}
<data-description>

A180. <pseudo-variable-reference>::= <pseudo-variable> [<argument-list>]
<data-description>

A181. <pseudo-variable>::= <imag-pv> | <onchar-pv> | <onsource-pv> | <pageno-pv> |
<real-pv> | <string-pv> | <substr-pv> | <unspec-pv>

A182. <subroutine-reference>::= <value-reference> [<argument-list>]

3.1.20 CONSTANT AND ISUB

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

A184. <isub>::= <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 CL8 and CL15 in Chapter 2.

3.1.22 TYPES OF PICTURE

A188. <pictured-character>::= <character-picture-element-list>

A189. <character-picture-element>::= A | X | 9

A190. <pictured-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>::= E | I | K | R | S | T | V | Y | Z | $ | 9 | + | - |
* | <insertion-character> | <credit> | <debit>

A198. <insertion-character>::= B | / | . | ,

A199. <credit>::= CR

A200. <debit>::= DB

The abstract text corresponding to the concrete representation

```
GO TO P -> S.LV;
```

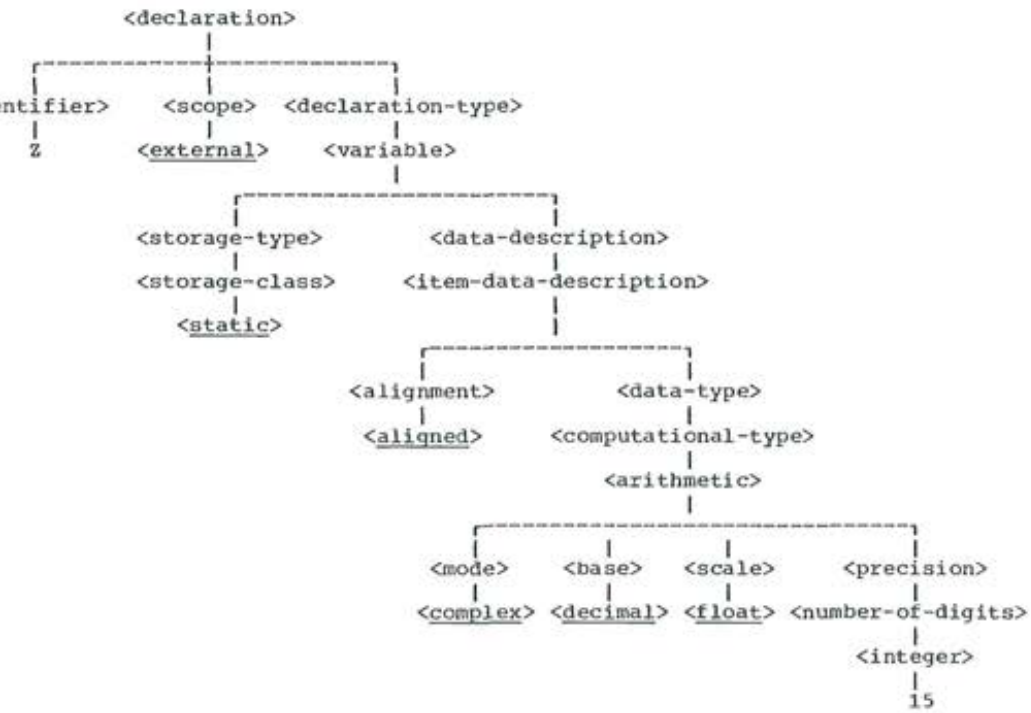
is as follows:

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

The abstract text corresponding to the concrete-representation

```
DECLARE Z EXTERNAL STATIC ALIGNED COMPLEX DECIMAL FLOAT(15);
```

is as follows:



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-procedure>s 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>.

x 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 n, if d contains n and d does not contain a {description} or {generic-description} that also contains n.

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

Operation: translate(t)

where t is a {symbol-list}.

result: an <abstract-external-procedure>.

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

Step 2. Perform complete-concrete-procedure.

Step 3. Let aep be an <abstract-external-procedure>.

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

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

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

Step 8. Perform validate-procedure(aep).

Step 9. Return aep.

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},snl. Perform high-level-parse(snl,n) to obtain nt.

Return nt.

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

Perform parse(sl,{pli-text}) to obtain a {pli-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 {non-delimiter}, 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.

{delimiter-or-non-delimiter}::= {delimiter} | {non-delimiter}

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

where pt is a {pli-text},
cn is a tree with a single node, whose type is a non-terminal of the middle-level syntax.

result: 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 cn.

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

Step 2. Repeat Steps 2.1 through 2.5 as long as there is a {delimiter-or-non-delimiter},d: {delimiter}: {include};; in t.

Step 2.1. Let s be {symbol-list}: {symbol}: ⊬.

Step 2.2. Append to s any {symbol}s obtained in an implementation-defined way from the {text-name} in d. Append {symbol}: ⊬; to s.

Step 2.3. Perform low-level-parse(s,{pli-text}) to obtain a {pli-text},tx.

Step 2.4. Let t1 be a {delimiter-or-non-delimiter-list} which contains a copy of the {delimiter} and {non-delimiter} components of tx in the same order.

Step 2.5. Replace d by the immediate components of t1 in the same order.

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

Step 4. Let nt[i], i=1,...,n, be the ordered list of nodes which are the immediate components of the {delimiter}s and {non-delimiter}s in t.

Step 5. There must exist one and only one tree, mt, which is a complete tree with respect to the middle-level syntax for the root-node cn and satisfies any additional constraints specified with that syntax, and which is such that mt contains m terminal nodes nmt[j], j=1,...,m and there is a one-to-one correspondence between the nt[i], i=1,...,n and nmt[j], 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 nmt[j] is an {environment-specification} or an {options-specification} then it corresponds to k nodes nt[l], l=i,...,i+k-1 such that

(1) no nt[l], l=i,...,i+k-1 is a {;};

(2) all nodes nt[l], l=i,...,i+k-1 which are either a {(} or a {)} must be matched in the normal way for balancing parentheses.

Case 5.1. nmt[j] is a keyword.

nt[i] must be an {identifier} containing the same terminals as the characters appearing either in the denotation of nmt[j] or in the abbreviation for nmt[j]. (See Section 2.7.)

Case 5.2. nmt[j] is a non-bracketed category-name other than a keyword.

nmt[j] and nt[i] must be equal.

Case 5.3. nmt[j] is a {radix-factor}.

nt[i] must be an {identifier} such that the ordered sequence, seq, of its terminals can be a denotation of an immediate component of a {radix-factor}.

Replace nmt[j] by a {radix-factor}: seq.

Case 5.4. nmt[j] is an {imaginary-constant}, a {real-constant}, or an {integer}.

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

Replace nmt[j] by the tree with root-node lc.

Case 5.5. nmt[j] is one of the following:

{identifier}
{arithmetic-constant}
{simple-bit-string-constant}
{simple-character-string-constant}
{isub}
{letter}

nt[i] and nmt[j] must be of the same type.

Replace nmt[j] by nt[i].

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

Step 7. Return mt.

4.2.3 HIGH-LEVEL-PARSE

Operation: high-level-parse(sl,cn)

where sl is a {sentence-list},
cn 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 cn.

Step 1. For each {single-statement},s, component of sl, in left-to-right order perform Steps 1.1 through 1.3.

Step 1.1. If s contains a {begin-statement}, {do-statement}, or {procedure-statement}, then attach {unmatched} to s.

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

Step 1.3. If s contains an {end-statement} containing an {identifier},id then perform Steps 1.3.1 and 1.3.2.

Step 1.3.1. Let rpu be the rightmost preceding {single-statement} containing {unmatched} such that rpu contains also a tree of the form {statement-name},sn: {identifier},idsn; and idsn is equal to id. sn must not contain a {signed-integer-commalist}. rpu must exist.

Step 1.3.2. Let k be the number of {unmatched} components of sl following rpu and preceding s. Let es be a

{sentence}:
{single-statement}:
{end-statement}:
END{;}.

Attach k copies of es to sl immediately preceding s. Delete the k+1 instances of {unmatched} which immediately precede s.

Step 2. Let nt[i], i=1,...,n be the ordered sequence of components of sl which are such that for each component, c, the following conditions are satisfied:

(1) the category-name of c is terminal with respect to the middle-level syntax, and

(2) c is not contained in any component of sl whose category-name is terminal with respect to the middle-level syntax.

Step 3. There must exist one and only one tree, ht, with root-node of the same type as cn such that the following conditions are true:

(1) ht 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) ht contains n terminal nodes htn[j], j=1,...,n, such that for every i, i=1,...,n, the type of the node nt[i] is the same as the type of htn[i].

Step 4. For each i, i=1,...,n, replace htn[i] by nt[i]. Return ht.

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 4. Perform construct-contextual-declarations.

Step 5. Perform construct-implicit-declarations.

Step 6. Perform complete-declarations.

Step 7. Perform validate-concrete-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-commalist},ds immediate component of a {declare-statement} component of the {concrete-external-procedure} perform defactor-declarations(ds) to obtain a {declaration-commalist},dds and replace ds by dds.

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},gs component of the {concrete-external-procedure} perform Steps 1.1 and 1.2.

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

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

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("FILE(SYSPRINT)",{file-option}) to obtain a {file-option},fo, 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(1)",{tab-format}) to obtain a {tab-format},ntf, and replace tf by ntf.

Step 4. For each {skip-option},sk component of the {concrete-external-procedure}, if sk does not contain an {expression} then perform parse("SKIP(1)",{skip-option}) to obtain nsk, and replace sk by nsk.

Step 5. For each {skip-format},sf component of the {concrete-external-procedure}, if sf does not contain an {expression} then perform parse("SKIP(1)",{skip-format}) to obtain nsf, and replace sf by nsf.

Step 6. For each {radix-factor},rf component of the {concrete-external-procedure}, if rf contains only B then replace rf by a {radix-factor}: B1.

4.3.1.2 Modify-statement-names

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

Operation: modify-statement-names

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

Step 1.1. Let pl be a {prefix-list}. For each {statement-name},sn of snl perform Step 1.1.1.

Step 1.1.1. Append {prefix}: snc; to pl, where snc is a copy of sn.

Step 1.2. Let un be a

```
{unit}:
  {executable-unit}:
       pl
       {executable-single-statement}:
            {null-statement}:
                 {;}.
```

Attach un to ul immediately preceding u. Delete snl.

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

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

Step 2.1.1. Let snc be a copy of sn. Let un1 be a

```
{unit}:
     {statement-name-list}:
          snc;
     {entry-statement},es:
          ENTRY
          {;}.
```

Step 2.1.2. If the {procedure-statement} immediate component of pc contains an {entry-information},ei then let eic be a copy of ei, delete any RECURSIVE subnode of eic, and attach eic to es.

Step 2.1.3. Attach un1 to pc as the first component of the {unit-list} immediate component of pc.

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

Step 3. For each {unit},u component of the {concrete-external-procedure}, where u immediately contains a {statement-name-list},snl and an {entry-statement},es and snl contains more than one {statement-name}, perform Steps 3.1 and 3.2.

Step 3.1. For each {statement-name},sn component of snl after the first, perform Steps 3.1.1 and 3.1.2.

Step 3.1.1. Let snc be a copy of sn. Let esc be a copy of es. Let unl be a

```
{unit}:
     {statement-name-list}:
          snc;
     esc.
```

Step 3.1.2. Attach unl to the {unit-list} which immediately contains u so that unl immediately precedes u.

Step 3.2. Delete every {statement-name} from snl except the first.

4.3.1.3 Complete-attribute-implications

The {dimension-attribute}, {precision}, and the {data-attribute}: FIXED; can be implied without the use of the keywords DIMENSION, PRECISION, and FIXED. These implications are replaced by explicit declarations of these attributes.

Operation: complete-attribute-implications

Step 1. For each {dimension-suffix},ds component of the {concrete-external-procedure} such that ds is not a component of a {dimension-attribute} append a

```
{data-attribute}:
     {dimension-attribute}:
          DIMENSION
          dsc;;
```

where dsc is a copy of ds, to the node immediately containing ds, and delete ds.

Step 2. For each {generic-description},gda component of the {concrete-external-procedure}, if gda immediately contains an {asterisk-bounds},ab then append a {generic-data-attribute}: DIMENSION ab; to the {generic-data-attribute-list} component of gda and delete ab.

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

```
{data-attribute}:
    PRECISION
    p1;
```

where p1 is a copy of p, and delete p.

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

```
{data-attribute}:
    FIXED.
```

4.3.1.4 Defactor-declarations

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

Operation: defactor-declarations(dc)

where dc is a {declaration-commalist}.

result: a {declaration-commalist}.

Step 1. Let ds be a copy of dc.

Step 2. For each {declaration},d immediate component of ds, if d immediately contains a {declaration-commalist},dcl then perform defactor-declarations(dcl) to obtain a {declaration-commalist},dcl1 and replace dcl by dcl1.

Step 3. For each {declaration},d immediate component of ds, if d immediately contains a {declaration-commalist},dl then perform Steps 3.1 through 3.3.

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

Step 3.2. If d immediately contains an {attribute-list},al then append a copy of each {attribute} of al to each {declaration} immediate component of dl.

Step 3.3. Replace d by the immediate components of dl in the same order.

Step 4. Return ds.

4.3.2 CONSTRUCT-EXPLICIT-DECLARATIONS

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

{declared-statement-names}::= [{name-list}] ({procedure} | {begin-block})

{name}::= {statement-name} [{entry-information}]

Operation: construct-explicit-declarations

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

Step 2. Let p be the {procedure} immediate component of the {concrete-external-procedure}. Perform declare-statement-names(p) to obtain a {declared-statement-names},dsn. Replace p by the {procedure} of dsn. The {name-list},nl of dsn must not contain a {signed-integer-commalist}.

Step 3. Perform construct-statement-name-declarations(nl,eattr) where eattr is an

```
{attribute}:
   {data-attribute}:
      ENTRY();;
```

to obtain a {unit} containing a {declaration-commalist},dc. Replace all occurrences of {attribute}: INTERNAL; in dc by {attribute}: EXTERNAL.

Step 4. Attach dc to the {concrete-external-procedure}.

4.3.2.1 Declare-parameters

Each {identifier} occurring in a {parameter-name-commalist} 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: declare-parameters(p)

where p is a {procedure}.

Step 1. For each {parameter-name-commalist},pl block-component of p perform Step 1.1.

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

Step 1.1.1. Perform find-applicable-declaration(id) to obtain d.

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

```
{unit}:
  {declare-statement}:
       DECLARE
       {declaration-commalist}:
            {declaration},d:
                 id;;
       {;};;
```

and append d to the {unit-list} of p.

Step 1.1.3. If d does not contain {attribute}: PARAMETER; then attach {attribute}: PARAMETER; to d.

4.3.2.2 Declare-statement-names

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

Operation: declare-statement-names(p)

where p is a {procedure} or {begin-block}.

result: a {declared-statement-names}.

Step 1. Let pc be a copy of p and let lnl, pnl, fnl, and enl each be a {name-list} with no components.

Step 2. For each {statement-name},sn 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 {executable-unit},eu.

Append sn to lnl.

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

Append sn to fnl.

Case 2.1.3. sn is contained either in a {prefix-list} immediate component of pc where pc has {procedure-statement},ep, or in a {unit} with an {entry-statement},ep as an immediate component.

Case 2.1.3.1. ep has an {entry-information},ei.

If ep is an {entry-statement}, then ei must not contain RECURSIVE. Append {name}: sn ei; to enl.

Case 2.1.3.2. ep has no {entry-information}.

Append {name}: sn; to enl.

Step 3. For each {procedure} or {begin-block}, pn concrete-block-component of pc perform Step 3.1.

Step 3.1. Perform declare-statement-names(pn) to obtain a {declared-statement-names},dsn. Append the elements of the {name-list} of dsn to pnl. Replace pn by the {procedure} or {begin-block} of dsn. If pn is a {begin-block}, then the {name-list} component of dsn must be empty.

Step 4. If lnl is not empty then perform construct-statement-name-declarations(lnl,lattr), where lattr is an

```
{attribute}:
   {data-attribute}:
      LABEL;;
```

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

Step 5. If fnl is not empty then perform construct-statement-name-declarations(fnl,fattr), where fattr is an

```
{attribute}:
   {data-attribute}:
      FORMAT;;
```

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

Step 6. If pnl is not empty then perform construct-statement-name-declarations(pnl,pattr), where pattr is an

```
{attribute}:
   {data-attribute}:
      ENTRY();;
```

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

Step 7. Return the {declared-statement-names} consisting of enl and pc.

4.3.2.3 Construct-statement-name-declarations

This operation takes a {name-list} containing {statement-name}s and possibly {entry-information}s and constructs a {declare-statement} for them. The type of the {statement-name}s is given by a supplied {attribute}. Any type of {statement-name} may contain one or more {integer}s, 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: construct-statement-name-declarations(nl,att)

where nl is a {name-list},
att is an {attribute}.

result: a {unit}.

Step 1. Let nlc be a copy of the {name-list},nl. Let un be a

```
{unit}:
  {declare-statement}:
       DECLARE
       {declaration-commalist},dcl
       {;}.
```

Step 2. While nlc contains any element, perform Steps 2.1 through 2.7.

Step 2.1. Let id be the {identifier} component of the first element of nlc.

Step 2.2. Let tnl be a {name-list} containing a copy of all elements of nlc whose {statement-name} immediately contains an {identifier} equal to id.

Step 2.3. Let d be a

```
{declaration}:
   id
   {attribute-list},al:
      {attribute}:
         INTERNAL;
      {attribute}:
         CONSTANT;
      att.
```

Step 2.4. If tnl contains more than one element or if tnl contains only one element and this element has a {signed-integer-commalist}, then perform Steps 2.4.1 through 2.4.6.

Step 2.4.1. Let m be the number of elements of tnl. Each element of tnl must contain a {signed-integer-commalist} with the same number of elements, n. tnl must be such that no two {signed-integer-commalist} components of tnl represent the same ordered sequence of numerical values.

Step 2.4.2. Let si[i,j] be the j'th {signed-integer} of the {signed-integer-commalist} of the i'th element of tnl.

Step 2.4.3. Let ub[k] and lb[k], k=1,...,n be, respectively, the largest and least value of si[1,k],...,si[m,k].

Step 2.4.4. Let bpl be the {bound-pair-list} containing n {bound-pair} elements such that the j'th element is a

```
{bound-pair}:
   {lower-bound}:
      e1;
   {:}
   {upper-bound}:
      e2;;
```

where e1 and e2 are {extent-expression}s representing lb[j] and ub[j] respectively.

Step 2.4.5. Let bpcl be the {bound-pair-commalist} produced by inserting commas as appropriate in bpl.

Step 2.4.6. Append an

```
{attribute}:
  {data-attribute}:
   {dimension-attribute}:
      DIMENSION
      {dimension-suffix}:
         (
         bpcl
          );;;;
```

to al, the {attribute-list} of d.

Step 2.5. If al contains ENTRY then perform Steps 2.5.1 through 2.5.2.

Step 2.5.1. If any {entry-information} of tnl 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 tnl must contain a {returns-descriptor}.

Note: a check for consistency is made in copy-descriptors.

Step 2.5.1.2. Append to al an

{attribute}:
{data-attribute}:
rd.

Step 2.5.2. If any {entry-information} of tnl contains an {options},op then perform Steps 2.5.2.1 and 2.5.2.2.

Step 2.5.2.1. Each element of tnl must contain an {options}.

Note: a check for consistency is made in copy-descriptors.

Step 2.5.2.2. Append to al an

{attribute}:
op.

Step 2.6. If dcl has a subcomponent, append {,} to dcl. Append d to dcl.

Step 2.7. Delete from nlc all those elements of which there is a copy in tnl.

Step 3. Return un.

4.3.3 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-commalist}, {description-commalist}, or {generic-description-commalist}, dl component of the {concrete-external-procedure} perform determine-structure(dl) to obtain a {declaration-commalist}, {description-commalist}, or {generic-description-commalist}, dlm. Replace dl by dlm.

Step 2. Let d[j] be the j'th {declaration} component of the {concrete-external-procedure} that declaration-contains LIKE. For each d[j] perform Step 2.1.

Step 2.1. Perform expand-like-attribute(d[j]) to obtain a {declaration-commalist}, dcml[j].

Step 3. For each d[j] perform Step 3.1.

Step 3.1. Let dl be the {declaration-commalist} that immediately contains d[j]. Attach a {,} followed by the elements of dcml[j] in sequence so that they immediately follow d[j].

Step 4. Delete all {attribute}: LIKE {unsubscripted-reference}; components of {declaration}s in the {concrete-external-procedure}.

Step 5. For each {declaration-commalist}, {description-commalist}, or {generic-description-commalist}, dl component of the {concrete-external-procedure} that declaration-contains STRUCTURE, perform convert-to-logical-levels(dl) to obtain dl1, a node of the same type as dl. Perform propagate-alignment(dl1) to obtain dl2, a node of the same type as dl1. Replace dl by dl2.

4.3.3.1 Determine-structure

Operation: determine-structure(cml)

where cml is a {declaration-commalist}, {description-commalist}, or {generic-description-commalist}.

result: a {declaration-commalist}, a {description-commalist}, or {generic-description-commalist}.

Step 1. Let cmlc be a copy of cml. Let e[j] be the j'th immediate component of cmlc that is not {,}. Let n be the number of such components.

Step 2. For each e[j] that immediately contains a {level},lv, perform Steps 2.1 through 2.4.

Step 2.1. lv must not be 0.

Step 2.2. If j is less than n and e[j+1] immediately contains a {level} whose numeric value is greater than that of lv, then attach STRUCTURE to e[j].

Step 2.3. If e[j] declaration-contains LIKE then attach STRUCTURE to e[j]. In this case, e[j] must not contain more than one instance of LIKE and cml must be a {declaration-commalist}.

Step 2.4. If the numeric value of lv is greater than one, attach MEMBER to e[j].

Step 3. For each e[j], all of the following must be false:

(1) e[j] immediately contains {level} and does not declaration-contain STRUCTURE or MEMBER.

(2) e[j] immediately contains {level} whose value is 1 and e[j] declaration-contains MEMBER.

(3) e[j] declaration-contains STRUCTURE or MEMBER and does not immediately contain {level}.

(4) e[j] declaration-contains STRUCTURE, does not declaration-contain LIKE and either j is equal to n or e[j+1] does not declaration-contain a {level} whose value is greater than the {level} declaration-component of e[j].

(5) e[j] declaration-contains MEMBER and either j is equal to one or e[j-1] does not declaration-contain MEMBER or STRUCTURE.

(6) e[j] declaration-contains STRUCTURE and LIKE, j is less than n, and e[j+1] declaration-contains MEMBER and a {level} whose numeric value is greater than the numeric value of the {level} declaration-contained in e[j].

(7) e[j] declaration-contains LIKE but not a {level}.

Step 4. Return cmlc.

4.3.3.2 Expand-like-attribute

Operation: expand-like-attribute(d)

where d is a {declaration}.

result: a {declaration-commalist}.

Step 1. d declaration-contains an

{attribute}:
LIKE
{unsubscripted-reference},r.

Step 2. Perform find-applicable-declaration(r) to obtain ld. ld must be a {declaration} that declaration-contains STRUCTURE and must not declaration-contain LIKE.

Step 3. Let ld be immediately contained in the {declaration-commalist},dcml. Let e[j] be the j'th element of dcml that is not {,}. Let k be such that e[k] is identical to ld and let n be the number of elements of dcml that are not {,}.

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 e[k] and let cf be the numeric value (m-lv).

Step 5. Let cl be a {declaration-commalist} 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, e[t+1] does not declaration-contain MEMBER, or e[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 e[i]. Let the numeric value of the immediately contained {level} of ec be lvc. Replace this {level} by one whose numeric value is lvc+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 LIKE.

Step 7. Return cl.

4.3.3.3 Convert-to-logical-levels

Operation: convert-to-logical-levels(cml)

where cml is a {declaration-commalist}, {description-commalist}, or {generic-description-commalist}.

result: a {declaration-commalist}, {description-commalist}, or {generic-description-commalist}.

Step 1. Let cmlc be a copy of cml. Let e[j] be the j'th component of cmlc that is not {,} and let m be the number of such components. Let l[j] be the {level} declaration-contained in e[j], if it exists, and let n[j] be the numeric value of l[j].

Step 2.

Case 2.1. There exists an integer t less than m such that e[t] and e[t+1] each declaration-contains a {level}, and n[t+1] is greater than n[t]+1.

Let j be the least such t.

Case 2.2. (Otherwise).

Return cmlc.

Step 3. Let j1 be the least integer greater than j such that either j1 is equal to m, or e[j1+1] does not declaration-contain MEMBER, or n[j1+1] is less than or equal to n[j]+1. For i=j+1,...,j1, perform Step 3.1.

Step 3.1. Replace l[i] by a {level} whose numeric value is n[i]-1.

Step 4. Go to Step 2.

4.3.3.4 Propagate-alignment

Operation: propagate-alignment(cml)

where cml is a {declaration-commalist}, {description-commalist}, or {generic-description-commalist}.

result: a {declaration-commalist}, {description-commalist}, or {generic-description-commalist}.

Step 1. Let cmlc be a copy of cml. Let e[j] be the j'th component of cmlc that is not {,} and let m be the number of such components. Let l[j] be the {level} declaration-contained in e[j], if it exists, and let n[j] be the numeric value of l[j].

Step 2. For i=1,...,m, perform Steps 2.1 and 2.2.

Step 2.1. e[i] must not declaration-contain both ALIGNED and UNALIGNED.

Step 2.2.

Case 2.2.1. e[i] declaration-contains MEMBER, and e[i] declaration-contains neither ALIGNED nor UNALIGNED.

Let k be the greatest integer such that k is less than i and n[k] is equal to n[i]-1. If e[k] declaration-contains ALIGNED or UNALIGNED, then attach ALIGNED or UNALIGNED, respectively, to e[i].

Case 2.2.2. (Otherwise).

No action.

Step 3. For i=1,...,m, perform Step 3.1.

Step 3.1. If e[i] contains STRUCTURE and ALIGNED, or STRUCTURE and UNALIGNED, delete ALIGNED or UNALIGNED, respectively, from e[i].

Step 4. Return cmlc.

4.3.3.5 Find-applicable-declaration

Operation: find-applicable-declaration(r)

where r is an {identifier}, {unsubscripted-reference}, {basic-reference}, or {reference}.

result: a {declaration} or <absent>.

Step 1.

Case 1.1. r is an {identifier}.

Let idl be an {identifier-list}: r.

Case 1.2. r is an {unsubscripted-reference}. Append, in order, a copy of each {identifier} component of r to an {identifier-list},idl.

Case 1.3. r is a {basic-reference}.

Append, in order, a copy of each {identifier} component of r which is not a component of an {arguments}, in order, to an {identifier-list},idl.

Case 1.4. r is a {reference}.

Let br be the {basic-reference} immediate component of r. Perform find-applicable-declaration(br) to obtain d. Return d.

Step 2. Let b be the concrete-block that block-contains r. If r is contained in a {statement-name},sn then perform Step 2.1.

Step 2.1. If sn is contained in a {unit} which simply contains {entry-statement} or sn is contained in a {prefix-list} immediate component of a {procedure} then let b be the concrete-block which block-contains b.

Step 3. Let id be the rightmost {identifier} component of idl.

Step 4. Let dl be a {declaration-designator-list} each component of which designates a {declaration} that is a concrete-block-component of b and that immediately contains an {identifier} that is equal to id.

Step 5. Delete from dl any component that designates a {declaration},d such that find-fully-qualified-name(d) returns an {identifier-list},didl such that idl does not contain an ordered sublist of the {identifier}s contained in didl.

Step 6.

Case 6.1. dl is empty.

Let b be the concrete-block that block-contains b. If there is no such block then return <absent>; otherwise, go to Step 4.

Case 6.2. dl contains a single component.

Let d be the {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 {declaration},d such that find-fully-qualified-name(d) returns an {identifier-list} equal to idl. Return d.

4.3.3.6 Find-fully-qualified-name

Each {declaration} has a fully qualified name associated with it. If it is an array or item declaration then this is a single {identifier}. If the {declaration} is a member of a structure then it may have as many {identifier}s as its logical level-number indicates its depth of embedding to be.

Operation: find-fully-qualified-name(d)

where d is a {declaration}.

result: an {identifier-list}.

Case 1. d declaration-contains MEMBER.

Step 1.1. Let dcml be the {declaration-commalist} that immediately contains d. Let dl be the rightmost preceding {declaration} of dcml that declaration-contains a {level} whose numeric value is less than the numeric value of the {level} of d.

Step 1.2. Perform find-fully-qualified-name(dl) to obtain an {identifier-list},idl.

Step 1.3. Append the {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 {identifier} immediate component of d.

Step 2.2. Return {identifier-list}: id.

4.3.4 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},dcml
     {;};;
```

where dcml contains no elements.

Step 2. For each {identifier},id in cep perform Steps 2.1 through 2.3.

Step 2.1. Let attrs be <absent>.

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 an

```
{attribute-list}:
  {attribute}:
    {data-attribute}:
       FILE;;
  {attribute}:
    CONSTANT.
```

Case 2.2.1.1.4. r is an immediate component of a {set-option} or a {locator-qualifier}, or r is an immediate component of an {attribute} that contains BASED.

Let attrs be an

```
{attribute-list}:
  {attribute}:
    {data-attribute}:
      POINTER;;
  {attribute}:
    VARIABLE.
```

Case 2.2.1.1.5. (Otherwise).

No action.

Case 2.2.2. id is an immediate component of a {programmer-named-condition}.

Let attrs be an

```
{attribute-list}:
  {attribute}:
    CONDITION.
```

Case 2.2.3. id is the only component of the {basic-reference} of a

```
{reference}:
  {basic-reference}:
    {identifier},id;
  {arguments-list};
```

where the {reference} has no other immediate components.

Let attrs be an

```
{attribute-list}:
  {attribute}:
    BUILTIN.
```

Case 2.2.4. (Otherwise).

No action.

Step 2.3. If attrs is not <absent> then perform find-applicable-declaration(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

```
{declaration}:
  id
  attrs.
```

Step 2.3.2. If an element of dcml immediately contains an {identifier} equal to id then the {attribute-list} of this {declaration} must equal attrs.

Step 2.3.3. If no element of dcml immediately contains an {identifier} equal to id then append d to dcml.

Step 3. If dcml contains any elements, append u to the {unit-list} of the {procedure} immediately contained in cep.

4.3.5 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.

Operation: construct-implicit-declarations

Step 1. For each tree of the form

```
{reference}:
  {basic-reference}:
    {identifier},id;;
```

where the {reference} has no other components, or that is the only {identifier} of an {unsubscripted-reference}, or that is an immediate component of an {allocation}, {freeing}, or {locate-statement}, perform Step 1.1.

Step 1.1. Perform find-applicable-declaration(id) to obtain d. If d is <absent> then perform Step 1.1.1.

Step 1.1.1. Let u be a

```
{unit}:
  {declare-statement}:
    DECLARE
    {declaration-commalist}:
      {declaration}:
        id;;
    {;}.
```

Append u to the {unit-list} of the {procedure} immediately contained in the {concrete-external-procedure}.

4.3.6 COMPLETE-DECLARATIONS

Operation: complete-declarations

Step 1. For each {default-attributes},das component of the {concrete-external-procedure}, perform Steps 1.1 and 1.2.

Step 1.1. Let d be a {declaration}: x;, where x is a copy of the {attribute-list} of das (d is a partial {declaration}).

Step 1.2. Perform test-attribute-consistency(d,<absent>) to obtain tv. tv must not be <false>.

Step 2. For each {declaration}, {description}, or {generic-description},d component ot the {concrete-external-procedure}, perform Step 2.1.

Step 2.1. Perform test-attribute-consistency(d) to obtain tv. tv must not be <false>.

Step 3. Perform append-system-defaults.

Step 4. Each {default-attributes} component of the {concrete-external-procedure} must not declaration-contain LIKE, MEMBER, STRUCTURE, or PARAMETER.

Step 5. For each {declaration},d component of the {concrete-external-procedure} perform Steps 5.1 and 5.2.

Step 5.1. Perform apply-defaults(d).

Step 5.2. Perform check-attribute-completeness-and-delete-attributes(d).

Step 6. For each {description},d component of the {concrete-external-procedure} that satisfies the following conditions, perform Steps 6.1 and 6.2.

(1) d is contained in a {declaration} but not in a {generic-attribute}.

(2) d was not produced by Step 6.1.

Step 6.1. Perform apply-defaults(d).

Step 6.2. Perform check-attribute-completeness-and-delete-attributes(d).

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(d).

Step 9. For each tree of the form

```
{procedure}:
  {prefix-list}:
    {prefix}:
      {statement-name},sn;;
  {procedure-statement}
  [{unit-list}]
  {ending};
```

or of the form

```
{unit}:
  {statement-name-list}:
    {statement-name},sn;
  {entry-statement};
```

perform Steps 9.1 through 9.3.

Step 9.1. Let id be the {identifier} immediately contained in sn.

Step 9.2. Perform find-applicable-declaration(id) to obtain d. d must be a {declaration}.

Step 9.3. Perform copy-descriptors(d).

4.3.6.1 Test-attribute-consistency

Operation: test-attribute-consistency(d,das)

where d is a {declaration}, {description}, or {generic-description},
das is a [{default-attributes}].

result: <true> or <false>.

Step 1. If das is <absent> then :

Case 1.1. d is a {declaration}.

For each pair of {attribute}s, 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}s, 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. d is a {generic-description}.

For each pair of {generic-data-attribute}s, a1 and a2, which are declaration-components of d, perform test-invalid-duplicates(a1,a2) to obtain tv. tv must not be <true>.

Step 2.

Case 2.1. das is <absent>.

Let akl be an {attribute-keyword-list} consisting of a copy of each keyword declaration-component of d.

Case 2.2. das is not <absent>.

Step 2.2.1. Let al be an {attribute-list} consisting of copies of each {attribute} declaration-component of das, 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 al, a1 and a2, perform test-invalid-duplicates(a1,a2) to obtain tv. If tv is <true>, return <false>.

Step 2.2.3. Let akl be an {attribute-keyword-list} consisting of a copy of each keyword declaration-component of al.

Step 3. Replace all multiple occurrences of a given keyword in akl by a single occurrence of that keyword.

Step 4. If akl contains GENERIC or MEMBER and also contains EXTERNAL, return <false>.

Step 5. If akl contains STRUCTURE and ALIGNED, or contains STRUCTURE and UNALIGNED, return <false>.

Step 6.

Case 6.1. d is a {description} or {generic-description}.

If the set of all keywords in akl is not a subset of the keywords that form the concrete-representation of some tree whose root-node is {consistent-description}, return <false>.

Case 6.2. d is a partial {declaration} produced by the operation create-constant.

If the set of all keywords in akl is not a subset of the keywords that form the concrete-representation of some tree whose root-node is {consistent-literal-constant}, return <false>.

Case 6.3. d is a {declaration}.

If the set of all keywords in akl is not a subset of the keywords that form the concrete-representation of some tree whose root-node is {consistent-declaration}, return <false>.

Step 7. Return <true>.

{consistent-declaration}::= {scope} {declaration-type}

{scope}::= EXTERNAL | INTERNAL

{declaration-type}::= {variable} | {named-constant} | BUILTIN | CONDITION | GENERIC

{variable}::= VARIABLE {storage-type} {data-description}

{storage-type}::= {storage-class} | DEFINED [POSITION] | PARAMETER | MEMBER

{storage-class}::= AUTOMATIC | BASED | CONTROLLED | STATIC

{data-description}::= [DIMENSION] {alignment} ({data-type} [INITIAL] | STRUCTURE)

{alignment}::= ALIGNED | UNALIGNED

{data-type}::= {computational-type} | {non-computational-type}

{computational-type}::= {arithmetic} | {string} | {pictured}

{non-computational-type}::= AREA | {entry} | FILE | FORMAT [LOCAL] | LABEL [LOCAL] | {locator}

{arithmetic}::= [REAL | COMPLEX] [BINARY | DECIMAL] [FIXED | FLOAT] PRECISION

{string}::= [CHARACTER | BIT] [VARYING | NONVARYING]

{pictured}::= PICTURE [REAL | COMPLEX]

{entry}::= ENTRY [RETURNS] [OPTIONS]

{locator}::= POINTER | OFFSET

{named-constant}::= CONSTANT [ENTRY | FILE {file-description-set} | FORMAT | LABEL] [DIMENSION]

{file-description-set}::= [ENVIRONMENT] [{stream-set} | {record-set}]

{stream-set}::= STREAM [INPUT | OUTPUT [PRINT]]

{record-set}::= RECORD [INPUT | OUTPUT | UPDATE] [{sequential-set} | {direct-set}]

{sequential-set}::= SEQUENTIAL [KEYED]

{direct-set}::= DIRECT KEYED

{consistent-literal-constant}::= CONSTANT [{arithmetic} | BIT | CHARACTER]

{consistent-description}::= {data-description} [MEMBER]

4.3.6.2 Test-invalid-duplicates

Operation: test-invalid-duplicates(a1,a2)

where a1 and a2 are {attribute}s or {data-attribute}s or are both {generic-data-attribute}s.

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>.

4.3.6.3 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-statement} that contains SYSTEM, append a

```
{unit}:
  {default-statement}:
       DEFAULT SYSTEM {;}+;
```

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 ld.

```
"/* ENTRY DEFAULTS */

 DEFAULT (RETURNS) ENTRY;


 /* FILE DEFAULTS */

 DEFAULT (DIRECT|INPUT|KEYED|
     OUTPUT|PRINT|RECORD|SEQUENTIAL|STREAM|UPDATE)  FILE;


 /* ARITHMETIC DEFAULTS */

 DEFAULT (¬CONSTANT & ¬PICTURE)  FIXED, BINARY, REAL;
 DEFAULT (FIXED & BINARY & ¬CONSTANT)  PRECISION(d1,0);
 DEFAULT (FIXED & DECIMAL & ¬CONSTANT) PRECISION(d2,0);
 DEFAULT (FLOAT & BINARY & ¬CONSTANT)  PRECISION(d3);
 DEFAULT (FLOAT & DECIMAL & ¬CONSTANT) PRECISION(d4);


 /* STRING AND AREA DEFAULTS */

 DEFAULT (CHARACTER) CHARACTER(1), NONVARYING;
 DEFAULT (BIT) BIT(1), NONVARYING;
 DEFAULT (AREA) AREA(d5);
 DEFAULT (POSITION) POSITION(1);


 /* SCOPE AND STORAGE CLASS DEFAULTS */

 DEFAULT ((ENTRY|FILE) &
           (AUTOMATIC|BASED|DEFINED|PARAMETER|STATIC|CONTROLLED|MEMBER|
            ALIGNED|UNALIGNED|INITIAL)) VARIABLE;
 DEFAULT ((ENTRY|FILE) & RANGE(*)) CONSTANT;
 DEFAULT (RANGE(*) & ¬CONSTANT) VARIABLE;
 DEFAULT (CONDITION|((FILE|ENTRY) & CONSTANT)) EXTERNAL;
 DEFAULT (RANGE(*)) INTERNAL;
 DEFAULT (VARIABLE & EXTERNAL) STATIC;
 DEFAULT (VARIABLE) AUTOMATIC;


 /* ALIGNMENT DEFAULTS */

 DEFAULT ((CHARACTER|BIT|PICTURE) & ¬CONSTANT) UNALIGNED;
 DEFAULT (¬CONSTANT) ALIGNED;"
```

Step 3. For each {unit},u component of the {concrete-external-procedure} where u immediately contains a {default-statement} that contains SYSTEM, perform Steps 3.1 and 3.2.

Step 3.1. Perform parse(ld,{unit-list}) to obtain ul.

Step 3.2. Replace the single {unit},u by the {unit} immediate components of ul such that all of the immediate components of ul are effectively inserted, in order, in place of u.

4.3.6.4 Apply-defaults

Operation: apply-defaults(d)

where d is a {declaration}, or a {description}, or a {returns-descriptor} with no {description} component.

Step 1. If d is a {returns-descriptor}, then attach to d a {description},decl with no components, and the surrounding parentheses as required; otherwise let decl be d.

Step 2. For each {default-statement},dft component of the {concrete-external-procedure} taken in left-to-right order perform Steps 2.1 through 2.3.

Step 2.1. dft must be a block-component of the {procedure} immediate component of the {concrete-external-procedure}.

Step 2.2. If dft immediately contains NONE then go to Step 3.

Step 2.3. Let dpe be the {predicate-expression} component of dft. Perform test-default-applicability(dpe,decl) to obtain tv. If tv is <true> perform Steps 2.3.1 and 2.3.2.

Step 2.3.1. dft must not contain ERROR.

Step 2.3.2. For each {default-attributes},das component of dft in left-to-right order perform Steps 2.3.2.1 through 2.3.2.3.

Step 2.3.2.1. Perform test-attribute-consistency(decl,das) to obtain tv.

Step 2.3.2.2. If tv is <true> and decl is a {declaration}, then append to the {attribute-list} in decl copies of the {attribute} simple components of das.

Step 2.3.2.3. If tv is <true> and decl is a {description}, then append to the {data-attribute-list} in decl copies of the {data-attribute} simple components of das.

Step 3. If d is a {returns-descriptor}, then decl must contain at least one subnode.

4.3.6.5 Test-default-applicability

Operation: test-default-applicability(dpe,decl)

where dpe is a {predicate-expression}, {predicate-expression-three}, {predicate-expression-two}, {predicate-expression-one}, {range-specification}, or {attribute-keyword},
decl is a {declaration}, or a {description}.

result: <true> or <false>.

Case 1. dpe immediately contains a {|}.

Let pe be the {predicate-expression} simple component of dpe, and p3 be the {predicate-expression-three} simple component ot dpe. If either, or both, test-default-applicability(pe,decl) or test-default-applicability(p3,decl) yield <true> then return <true>; otherwise return <false>.

Case 2. dpe immediately contains an &.

Let p3 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(p3,decl) yield <true>, then return <true>; otherwise return <false>.

Case 3. dpe immediately contains a ¬.

Let pe be the other simple component of dpe. Perform test-default-applicability(pe,decl) to obtain tv. If tv is <true> then return <false>; otherwise return <true>.

Case 4. dpe is a {range-specification},rs.

Case 4.1. rs contains an *.

If decl is a {declaration} which immediately contains an {identifier} return <true>, otherwise return <false>.

Case 4.2. rs 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 idd be the {identifier} immediate component of decl. Compare the terminal nodes of id and idd, 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. rs contains {letter},l1 {:} {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} simple component of decl. If dl is, or is after l1 and is, 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},atr or a {data-attribute},atr such that the leftmost keyword, k of atr is equal to ak then return <true>; otherwise return <false>.

Case 6. (Otherwise).

Let pe be the {predicate-expression} component of dpe. Perform test-default-applicability(pe,decl) to obtain tr. Return tr.

4.3.6.6 Copy-descriptors

Operation: copy-descriptors(d)

where d is a {declaration}.

Step 1. Let e be the {data-attribute} declaration-component of d that immediately contains ENTRY.

Step 2.

Case 2.1. d declaration-contains a {dimension-attribute}.

Let ep [i], i=1,...,n, be the n occurrences of

```
{statement-name}:
   {identifier},id
   ({signed-integer-commalist}) {:};
```

such that d is obtained by performing find-applicable-declaration(id).

Case 2.2. (Otherwise).

Let ep[1] be the single occurrence of a

```
{statement-name}:
   {identifier},id {:};
```

such that d is obtained by performing find-applicable-declaration(id). Let n be 1.

Step 3. For each ep[i], i=1,...,n, perform Steps 3.1 through 3.4.

Step 3.1.

Case 3.1.1. ep[i] is contained in a {prefix-list} immediate component of a {procedure},p.

Let es be the immediately contained {procedure-statement} of p.

Case 3.1.2. ep[i] is contained in a {statement-name-list} immediate component of a {unit},u.

Let es be the immediately contained {entry-statement} of u.

Step 3.2.

Case 3.2.1. es contains a {parameter-name-commalist},pl.

Step 3.2.1.1. Let pn[j] be the {identifier} of the j'th immediately contained {parameter-name} component of pl. Let m be the number of such components.

For each pn[j], j=1,...,m, taken in order, perform Steps 3.2.1.1.1 through 3.2.1.1.4.

Step 3.2.1.1.1. Perform find-applicable-declaration(pn[j]) to obtain pd. pd must be a {declaration}.

Step 3.2.1.1.2. Let dcml be the {declaration-commalist} that immediately contains pd. Let dcm[k] be the k'th {declaration} component of dcml. Let there be kmx such components and let k be such that dcm[k] is identical to pd.

Step 3.2.1.1.3. Let knd be the minimum integer greater than or equal to k such that one of the following is true:

(1) knd is equal to kmx;

(2) dcm[knd+1] does not contain {level};

(3) dcm[knd+1] contains a {level} whose {integer} has the value 1.

Step 3.2.1.1.4. For each dcm[inx], inx=k,...,knd, 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 pnd be a copy of dcm[inx]. Delete from pnd any occurrence of an

```
{attribute}:
    PARAMETER.
```

Step 3.2.1.1.4.2. If dcm[inx] contains an

```
{attribute}:
  {data-attribute},da:
    OFFSET
    (
    {reference},r
    );;
```

then perform test-offset-in-description(da) to obtain tv. If tv is <trim>, then replace the {data-attribute} in pnd 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 pnd contains a

```
{data-attribute},da:
    ENTRY
    (
    {description-commalist}
    );
```

then replace da with a

```
{data-attribute}:
    ENTRY.
```

Step 3.2.1.1.4.4. If pnd 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-commalist},des[i], a {description} simply containing copies of the {level} and {data-attribute} simple components of pnd. pnd must not contain any {extent-expression}s containing an {identifier}.

Case 3.2.2. es does not contain a {parameter-name-commalist}.

Let des[i] be <absent>.

Step 3.3. If es contains a <returns-descriptor>, let rd[i] be that {returns-descriptor}; otherwise, let rd[i] be <absent>.

Step 3.4. If es contains an {options} then let ops[i] be a copy of that {options}; otherwise let ops[i] be <absent>.

Step 4.

Step 4.1. Let x[i] be des[i] for i=1,...,n. Perform Steps 4.5 and 4.6.

Step 4.2. Let x[i] be rd[i] for i=1,...,n. Perform Steps 4.5 and 4.6.

Step 4.3. Let x[i] be ops[i] 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 cx[i] of all the x[i] in which the ordering of the immediate components of {data-attribute-list}s and {options-specification}s has been altered such that all the cx[i] are equal, except for the subnodes of any {extent-expression}s.

Step 4.6. For each pair of corresponding {extent-expression}s, ext1 and ext2, in each pair of members of the set cx[i], perform test-descriptor-extent-expressions(ext1,ext2) to obtain tv, which must be <true>.

Step 5. If des[1] is not <absent>, replace e with a

```
{data-attribute}:
    ENTRY
    (
    des[1]
    ).
```

Step 6. If d declaration-contains a {returns-descriptor},drd, then for i=1,...,n, replace rd[i] (which must not be <absent>) by drd.

Case 6.1. d is not block-contained in the {concrete-external-procedure}.

For each {data-attribute},da: OFFSET({reference}); in rd[i], perform Step 6.1.1.

Step 6.1.1. Perform test-offset-in-description(da) to obtain tv, which must be <notrim>.

Case 6.2. (Otherwise).

For each {data-attribute}: OFFSET({reference}); in drd, delete the ({reference}).

Step 7. If d declaration-contains an {options},dops, then for i=1,...,n, replace ops[i] (which must not be <absent>) by dops.

4.3.6.7 Test-offset-in-description

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-in-description(da)

where da is a {data-attribute}: OFFSET({reference}).

result: <trim> or <notrim>.

Step 1. For each {reference},r 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 <trim>.

Step 2. Return <notrim>.

4.3.6.8 Test-descriptor-extent-expressions

Operation: test-descriptor-extent-expressions(ext1,ext2)

where ext1 and ext2 are {extent-expression}s.

result: <true> or <false>.

Step 1. If ext1 or ext2 contains {identifier}, return <false>.

Step 2. Let ce1 and ce2 be the {expression} immediate components of ext1 and ext2.

Perform create-expression(ce1) to obtain an <expression>,e1, and create-expression(ce2) 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>s i1 and i2 respectively. If i1 and i2 are equal, return <true>; otherwise return <false>.

4.3.7 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}s 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 {declaration}s, 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).

(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-declaration(r) yields <absent>.

(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 d was created by the operation construct-statement-name-declarations.

4.3.7.1 Check-attribute-completeness-and-delete-attributes

Operation: check-attribute-completeness-and-delete-attributes(d)

where d is a {declaration} or a {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 {attribute},att or a {data-attribute},att declaration-component of d, such that att declaration-contains k, but not as its sole terminal component, then delete all {attribute} or {data-attribute} declaration-components of d which declaration-contain k, except for att; otherwise, delete all but one of the {attribute} or {data-attribute} declaration-components of d which declaration-contain k.

Step 2. d must not declaration-contain an {attribute} or {data-attribute} with AREA, BIT, CHARACTER, DIMENSION, GENERIC, INITIAL, PICTURE, POSITION, PRECISION, or RETURNS as its sole terminal node.

Step 3. If d declaration-contains EXTERNAL, it must not declaration-contain AUTOMATIC, BASED, DEFINED, PARAMETER, or BUILTIN.

Step 4. If d declaration-contains EXTERNAL and CONSTANT, it must not declaration-contain FORMAT or LABEL.

Step 5. If d declaration-contains EXTERNAL and CONSTANT and ENTRY, it must not declaration-contain DIMENSION.

Step 6. If d is a {description}, d must not declaration-contain INITIAL.

Step 7. If d is a {declaration} which declaration-contains DEFINED or PARAMETER (or declaration-contains MEMBER and the rightmost preceding {declaration} whose {level} has the value one declaration-contains DEFINED or PARAMETER), then d must not declaration-contain INITIAL.

Step 8. If d is a {declaration} which declaration-contains STATIC and either ENTRY, FORMAT, or LABEL (or declaration-contains MEMBER and either ENTRY, FORMAT, or LABEL, and the rightmost preceding {declaration} whose {level} has the value one declaration-contains STATIC), then d must not declaration-contain INITIAL.

Step 9. Let akl be an {attribute-keyword-list} consisting of copies of the keyword declaration-components of d.

Step 10. Delete from akl any keyword that can be contained in a tree whose root-node is {file-description-set}.

Step 11.

Case 11.1. d is a {description}.

There must exist a tree whose root-node is {consistent-description} and whose concrete-representation consists of the same set of keywords as are contained in akl.

Case 11.2. d is a {declaration}.

There must exist a tree whose root-node is {consistent-declaration} and whose concrete representation consists of the same set of keywords as are contained in akl.

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 terminals 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: create-abstract-equivalent-tree(ct)

where 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 <none>.

Case 1. ct is an {allocation}, {assignment-statement}, {begin-block}, {constant}, {expression}, {format-iteration}, {freeing}, {identifier}, {initial-element}, {if-statement}, {group}, {locate-statement}, {on-statement}, or {picture}.

Perform create-f(ct) to obtain abt, where "{f}" is the name of the category of ct. Return abt.

Case 2. ct belongs to a terminal category.

Case 2.1. ct is a keyword and there is a terminal category whose name is of the form "<cn>" where cn is the lowercase equivalent of category name of ct (e.g., if ct is FIXED, cn is "fixed").

Return a <cn>.

Case 2.2. ct is an *.

Return an <asterisk>.

Case 2.3. (Otherwise).

Return <none>.

Case 3. The category-name of ct is of the form "{cn-list}" or "{cn-commalist}", where "cn" is some name.

Let x be a <cn-list>. For each {cn},y, in ct, taken in order, perform create-abstract-equivalent-tree(y) to obtain z, and append z to x. Return x.

Case 4. ct is a {radix-factor}.

Let d be the digit in the denotation of the immediate component of ct. Return <radix-factor>: d.

Case 5. ct is a {condition-name}.

Perform create-condition(ct) to obtain abt. Return abt.

Case 6. (Otherwise).

Step 6.1. Let "cn" be the name such that "{cn}" is the category-name of ct. Let x be a <cn>. If <cn> is a terminal category, return x.

Step 6.2. For each immediate subnode, y, of ct, taken in order, perform Step 6.2.1.

Step 6.2.1.

Case 6.2.1.1. y is a {reference}.

Let ref be a <variable-reference>, or a <target-reference>, or a <subroutine-reference>, choosing the alternative that permits ref to be attached to x as an immediate subnode. Perform create-reference(y,ref) to obtain z. Attach z to x.

Case 6.2.1.2. y is an {unsubscripted-reference}.

Perform create-reference(y,<variable-reference>) to obtain z. Attach z to x.

Case 6.2.1.3. (Otherwise).

Perform create-abstract-equivalent-tree(y) to obtain z. If z is not equal to <none>, attach z to x.

Step 6.3. Return x.

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},cps. Perform create-entry-point(cps) to obtain an <entry-point>,ep. Let ap be a

```
<procedure>:
    <entry-or-executable-unit-list>,eeul:
      <entry-or-executable-unit>: ep.
```

Step 2. Perform create-block(cp,ap).

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 <absent>, attach cpl to ap.

Step 4. If cps simply contains RECURSIVE, then attach <recursive> to ap.

Step 5. Return ap.

4.4.1.2 Create-begin-block

A <begin-block> is constructed in much 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 <entry-point>,ep, contained in abb, abb must contain a <procedure> that contains ep.

Step 4. Return abb.

4.4.1.3 Create-block

Operation: create-block(cb,ab)

where cb is a {procedure} or {begin-block},
ab is a <procedure> or <begin-block>.

Step 1. For each {declaration},d that is a block-component of cb and that does not declaration-contain MEMBER or GENERIC, perform create-declaration(d) to obtain a <declaration>,ad, and append ad to the <declaration-list> immediately contained in ab.

Step 2. For each <declaration>,ad component of ab such that ad contains at least one {expression-designator}, {reference-designator}, or {initial-designator} perform replace-concrete-designators(ad).

Step 3. For each {procedure},p that is a block-component of cb perform create-procedure(p) to obtain a <procedure>,ap and append ap to the <procedure-list> in ab.

Step 4. For each {format-statement},fs that is a block-component of cb, perform create-format-statement(fs) to obtain a <format-statement>,afs and append afs to the <format-statement-list> in ab.

Step 5. If cb immediately contains a {unit-list}, let ul be that {unit-list}. Otherwise let ul be <absent>.

Case 5.1. cb is a {procedure}.

Perform create-entry-or-executable-unit-list(ul) to obtain an <entry-or-executable-unit-list>,eul and attach eul to ab.

Case 5.2. cb is a {begin-block}.

Perform create-executable-unit-list(ul) to obtain an <executable-unit-list>,eul and attach eul to ab.

4.4.1.4 Replace-concrete-designators

Operation: replace-concrete-designators(ad)

where ad is a <declaration> or a <data-description>.

Step 1. For each {expression-designator},ed component of ad, perform Steps 1.1 and 1.2.

Step 1.1. Let e be the {expression} designated by ed.

Step 1.2. Perform create-abstract-equivalent-tree(e) to obtain an <expression>,ae and replace ed by ae.

Step 2. For each {reference-designator},rd component of ad, perform Steps 2.1 and 2.2.

Step 2.1. Let r be the {reference} designated by rd.

Case 2.1.1. rd is an immediate component of <based>.

Perform create-reference(r,<value-reference>) to obtain a <value-reference>,vr.

Case 2.1.2. rd is an immediate component of <base-item> or <offset>.

Perform create-reference(r,<variable-reference>) to obtain a <variable-reference>,vr.

Step 2.2. Replace rd by vr.

Step 3. For each {initial-designator},ides component of ad, perform Step 3.1.

Step 3.1. Let int be the {initial} designated by ides. Perform create-abstract-equivalent-tree(int) to obtain an <initial>,i and replace int by i.

4.4.1.5 Create-group

Operation: create-group(g)

where g is a {group}.

result: a <group>.

Step 1. Let ds be the {do-statement} in g. If g immediately contains a {unit-list}, let ul be that {unit-list}. Otherwise let ul be <absent>.

Case 1.1. ds immediately contains a {do-spec},dsp.

Perform create-abstract-equivalent-tree(dsp) to obtain a <do-spec>,adsp. Perform create-executable-unit-list(ul) 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>: adsp eul.

Case 1.2. ds immediately contains a {while-option},wo.

Perform create-abstract-equivalent-tree(wo) to obtain a <while-option>,awo. Perform create-executable-unit-list(ul) 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(ul) 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(ul)

where ul is a [{unit-list}].

result: an <entry-or-executable-unit-list>.

Step 1. Let eul be an <entry-or-executable-unit-list>. If ul is not <absent> then for each {unit},u in ul, taken in order, perform Step 1.1.

Step 1.1. If u immediately contains an {entry-statement},es then perform create-entry-point(es) to obtain an <entry-point>,ep, and append an <entry-or-executable-unit>: ep; to eul; otherwise, if u immediately contains an {executable-unit},eu then perform create-executable-unit(eu) to obtain an <executable-unit>,aeu and append an <entry-or-executable-unit>: aeu; 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(ul)

where ul is a [{unit-list}].

result: an <executable-unit-list>.

Step 1. Let eul be an <executable-unit-list>. If ul is not <absent> then for each {unit},u in ul, taken in order, perform Step 1.1.

Step 1.1. u must not immediately contain an {entry-statement}. If u immediately contains an {executable-unit},eu, perform create-executable-unit(eu) to obtain an <executable-unit>,aeu and append aeu to eul.

Step 2. Append an <executable-unit>: <end-statement>; to eul, and return eul.

4.4.1.8 Create-executable-unit

Operation: create-executable-unit(eu)

where eu is {executable-unit}.

result: an <executable-unit>.

Step 1. If eu immediately contains an {executable-single-statement},es, let st be the immediate component of es; otherwise let st be the immediate component of eu that is not a {prefix-list}.

Step 2. Perform create-abstract-equivalent-tree(st) to obtain ast. Let aeu be an <executable-unit>: ast.

Step 3. If eu immediately contains a {prefix-list},pl, perform Steps 3.1 and 3.2.

Step 3.1. Perform create-condition-prefix-list(pl) to obtain a <condition-prefix-list>,cpl or <absent>,cpl. If cpl is not <absent> then attach cpl to aeu.

Step 3.2. Perform create-statement-name-list(pl) to obtain a <statement-name-list>,snl or <absent>,snl. If snl is not <absent>, then attach snl to aeu.

Step 4. Return aeu.

4.4.1.9 Create-entry-point

An <entry-point> may be the primary <entry-point> of a <procedure>, or a secondary <entry-point> specified in the concrete {procedure} by an {entry-statement}. Each <entry-point> has its own information for entry and return which beccmes the <entry-information> and <returns-descriptor> respectively.

Operation: create-entry-point(es)

where es is an {entry-statement} or a {procedure-statement}.

result: an <entry-point>.

Step 1. Let ep be an <entry-point>: <entry-information>,ei.

Case 1.1. es is an {entry-statement}.

Let pl be the {statement-name-list} of the {unit} immediately containing es.

Case 1.2. es is a {procedure-statement}.

Let pl be the {prefix-list} of the {procedure} immediately containing es.

Step 2. Perform create-statement-name-list(pl) to obtain a <statement-name-list>,snl. Let asn be a copy of the <statement-name> in snl, and attach asn to ep.

Step 3. If es has a {parameter-name-commalist},pnl, perform create-abstract-equivalent-tree(pnl) to obtain a <parameter-name-list>,apnl and attach apnl to ei.

Step 4. If es has a {returns-descriptor},rd, then perform Step 4.1.

Step 4.1. Let d be the first {description} in rd. Perform create-data-description(d) to obtain a <data-description>,dd. Perform replace-concrete-designators(dd). Let rdd be a <returns-descriptor>: dd; and attach rdd to ei.

Step 5. If es has an {options}, attach an implementation-dependent tree of type <options> to ei.

Step 6. Return ep.

4.4.1.10 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 snl be a <statement-name-list>. For each {statement-name},sn in pl, perform Steps 1.1 and 1.2.

Step 1.1. Let id be the {identifier} in sn. Perform create-identifier(id) to obtain an <identifier>,ad. Let asn be a <statement-name>: ad. For each {signed-integer},si in sn, taken in order, perform Step 1.1.1.

Step 1.1.1. Let asi be a <signed-integer> whose concrete-representation is the same as that of si. Append asi to the <signed-integer-list> in asn.

Step 1.2. Append asn to snl.

Step 2. If snl has any subnodes, return snl; otherwise return <absent>.

4.4.1.11 Create-condition-prefix-list

Operation: create-condition-prefix-list(pl)

where pl is a {prefix-list} or a {condition-prefix-commalist}.

result: a <condition-prefix-list>.

Step 1. Let cpl be a <condition-prefix-list>.

Step 2. For each {computational-condition},cc in pl, perform Step 2.1.

Step 2.1. Perform create-condition(cc) to obtain a <computational-condition>,acc. Append a <condition-prefix>: acc <enabled>; to cpl.

Step 3. For each {disabled-computational-condition},dcc, in pl, perform Step 3.1.

Step 3.1. Perform create-condition(dcc) to obtain a <computational-condition>,acc. cpl must not contain <condition-prefix>: acc <enabled>. Append a <condition-prefix>: acc <disabled>; to cpl.

Step 4. If cpl has any subnodes then return cpl; otherwise return <absent>.

4.4.1.12 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},cc.

Perform create-condition(cc) to obtain a <computational-condition>,acc, and return a <condition-name>: acc.

Case 6. c is a {condition-name}: {named-io-condition}: {io-condition},ioc ({reference},ref).

Perform create-condition(ioc) to obtain an <io-condition>,aioc, and perform create-reference(ref,<value-reference>) to obtain a <value-reference>,vr. Return a <condition-name>: <named-io-condition>: aioc 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.

4.4.2 CREATION OF STATEMENTS

4.4.2.1 Create-assignment-statement

Operation: create-assignment-statement(ast)

where ast is an {assignment-statement}.

result: an <assignment-statement>.

Step 1.

Case 1.1. ast immediately contains {,}, BY, and NAME.

Perform create-by-name-assignment(ast) to obtain an <assignment-statement>,aast or a <null-statement>,aast. If aast is a <null-statement> then return aast.

Case 1.2. (Otherwise).

Let aast be an <assignment-statement>: <target-reference-list>,trl. For each {reference},r, immediately contained in the {reference-commalist} in ast, taken in left-to-right order, perform create-reference(r,<target-reference>) to obtain a <target-reference>,tr, 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 aast.

Step 2. The <data-description> of ae must be proper for assignment to the <data-description> of each <target-reference> in aast.

Step 3. Return aast.

4.4.2.2 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>,bnpl or <absent>,bnpl. If bnpl is <absent>, return a <null-statement>.

Step 2. Let aast be an <assignment-statement>: <target-reference-list>,trl. For each {reference},r, immediately contained in the {reference-commalist} in ast, taken in left-to-right order, perform Step 2.1.

Step 2.1. Perform create-reference(r,<target-reference>,bnpl) to obtain a <target-reference>,tr. Append tr to trl.

Step 3. Let e be the {expression} immediately contained in ast. Perform create-expression(e,bnpl) to obtain an <expression>,ae and attach ae to aast.

Step 4. Each <builtin-function-reference> or <procedure-function-reference> simple component of aast, which is not contained in a <locator-qualifier>, in a <subscript>, or in a <builtin-function-reference> whose result has <aggregate-type>: <scalar>; (see "Attributes" for each builtin function in Section 9.4.4), must immediately contain a <data-description> which simply contains an <item-data-description>.

Step 5. Return aast.

4.4.2.3 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 subnodes of the <non-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>.

4.4.2.4 Create-by-name-parts-list

Operation: create-by-name-parts-list(asr)

where asr is an {assignment-statement}
or an {arguments}.

result: a <by-name-parts-list> or <absent>.

Step 1. Let bnpl be a <by-name-parts-list-list> with no components.

Step 2. For each {reference},lr immediately contained in the {reference-commalist} of asr perform Step 2.1.

Step 2.1. Perform find-applicable-declaration(lr) to obtain a {declaration},cd, which must declaration-contain STRUCTURE. Perform find-by-name-parts(cd) to obtain a <by-name-parts-list>,bnp and append bnp to bnpl.

Step 3. For each {reference},r contained in the {expression} immediate component of asr, but not contained in a {locator-qualifier} or {arguments}, perform Steps 3.1 and 3.2.

Step 3.1. Perform find-applicable-declaration(r) to obtain a {declaration},cd.

Step 3.2. If cd declaration-contains STRUCTURE then perform find-by-name-parts(cd) to obtain a <by-name-parts-list>,bnp and append bnp to bnpl.

Step 4. Let rbnpl be a <by-name-parts-list> consisting of those <by-name-parts> which are common to every <by-name-parts-list> of bnpl.

Step 5. If rbnpl is not empty then return rbnpl; otherwise return <absent>.

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 dl be the {declaration-commalist} node that immediately contains d. Let e[j] be the j'th immediate component of dl that is not a {,}. Let n be the number of such components and let k be such that e[k] is d. Let ld be the numeric value of the {level} of d. Let i be k+1.

Step 2. Let bnpl 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 e[i] does not declaration-contain MEMBER then go to Step 4.

Step 3.2. Let le be the numeric value of the {level} of e[i]. If le≤ld, go to Step 4.

Step 3.3. If le = ld+1 then perform Steps 3.3.1 and 3.3.2.

Step 3.3.1. Let cid be the {identifier} of e[i]. Perform create-identifier(cid) to obtain an <identifier>,id.

Step 3.3.2.

Case 3.3.2.1. e[i] declaration-contains STRUCTURE.

Perform find-by-name-parts(e[i]) to obtain a <by-name-parts-list>,rbnp. Attach a copy of id to each <identifier-list> component of rbnp as the initial element. Append a copy of each immediate component of rbnp to bnpl.

Case 3.3.2.2. (Otherwise).

Append a <by-name-parts> containing id to bnpl.

Step 3.4. Let i = i+1.

Step 4. Return bnpl.

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},cdcl. cdcl must not declaration-contain MEMBER.

Step 2. Let adcl be the <declaration> whose {declaration-designator} designates cdcl. If adcl contains <controlled>, then al must not contain a {set-option} or an {in-option}. If adcl contains a <based>,b and al does not contain a {set-option}, then b must immediately contain a <value-reference> that immediately contains a <variable-reference>.

Step 3. Let des be a <declaration-designator> designating adcl. Let aal be an <allocation>: des. If al contains a {set-option},sp, then perform create-abstract-equivalent-tree(sp) to obtain a <set-option>,asp, and attach asp to aal. If al contains an {in-option},io, then perform create-abstract-equivalent-tree(io) to obtain an <in-option>,aio, and attach aio to aal.

Step 4. Return aal.

4.4.2.7 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-commalist} component of fs. Perform create-abstract-equivalent-tree(fsl) to obtain a <format-specification-list>,afsl.

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>,snl, which must not be <absent>. Let afs be a <format-statement>: snl afsl.

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.

4.4.2.8 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-expression(e) to obtain an <expression>,ae and attach a <format-iteration-factor>: ae; to afi.

Step 3.

Case 3.1. fi immediately contains a {format-specification-commalist},fsc.

Perform create-abstract-equivalent-tree(fsc) to obtain a <format-specification-list>,fl, and attach fl to afi.

Case 3.2. fi immediately contains a {format-item},ft.

Perform create-abstract-equivalent-tree(ft) to obtain a <format-item>,ft and attach a <format-specification-list>: <format-specification>: ft;; to afi.

Step 4. Return afi.

4.4.2.9 Create-freeing

Operation: create-freeing(fr)

where fr is a {freeing}.

result: a <freeing>.

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 afr be a <freeing>: des. If fr immediately contains a {locator-qualifier} with {reference},r, perform create-reference(r,<value-reference>) to obtain a <value-reference>,vr and attach a <locator-qualifier>: vr; to afr. If afr contains an {in-option},io, perform create-abstract-equivalent-tree(io) to obtain an <in-option>,aio, and attach aio to afr. Return afr.

4.4.2.10 Create-if-statement

Operation: create-if-statement(ifs)

where ifs is an {if-statement}.

result: an <if-statement>.

Step 1. ifs immediately contains an {if-clause} which contains an {expression},e. Perform create-expression(e) to obtain an <expression>,ae.

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>,aeu. Return an

```
<if-statement>:
     <test>: ae;
     <then-unit>: aeu.
```

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>,eu1. Perform create-executable-unit(eu) to obtain an <executable-unit>,eu2. Return an

```
<if-statement>:
     <test>: ae;
     <then-unit>: eu1;
     <else-unit>: eu2.
```

4.4.2.11 Create-balanced-unit

Operation: create-balanced-unit(bu)

where bu is a {balanced-unit}.

result: an <executable-unit>.

Step 1.

Case 1.1. bu immediately contains an {executable-single-statement},ess.

Let s be the immediate component of ess. Perform create-abstract-equivalent-tree(s) to obtain stat.

Case 1.2. bu immediately contains a {group}, {begin-block}, or {on-statement}.

Let s be that {group}, {begin-block}, or {on-statement}. Perform create-abstract-equivalent-tree(s) to obtain stat.

Case 1.3. (Otherwise).

Let e be the {expression} immediately contained in the {if-clause} immediately contained in bu. Perform create-expression(e) to obtain an <expression>,ae. Let bu1 and bu2 be, in order, the {balanced-unit}s immediately contained in bu. Perform create-balanced-unit(bu1) to obtain eu1 and create-balanced-unit(bu2) to obtain eu2. Let stat be an

```
<if-statement>:
     <test>: ae;
     <then-unit>: eu1;
     <else-unit>: eu2.
```

Step 2. Let eu be an <executable-unit>: stat. If bu immediately contains a {prefix-list},pl, perform Steps 2.1 and 2.2.

Step 2.1. Perform create-statement-name-list(pl) to obtain a <statement-name-list>,snl or <absent>,snl. If snl is not <absent>, attach snl to eu.

Step 2.2. Perform create-condition-prefix-list(pl) to obtain a <condition-prefix-list>,cpl or <absent>,cpl. If cpl is not <absent>, attach cpl to eu.

Step 3. Return eu.

4.4.2.12 Create-locate-statement

Operation: create-locate-statement(ls)

where ls is a {locate-statement}.

result: a <locate-statement>.

Step 1. Let id be the {identifier} immediately contained in ls. Perform find-applicable-declaration(id) to obtain a {declaration},cdcl. cdcl must not declaration-contain MEMBER.

Step 2. Let des be a <declaration-designator> designating the <declaration>,ad, whose {declaration-designator} designates cdcl. ad must contain <based>. Let als be a <locate-statement>: des. For each immediate subnode, x, of ls other than id or LOCATE, perform create-abstract-equivalent-tree(x) to obtain a <file-option>,y, <pointer-set-option>,y or a <keyfrom-option>,y, and attach y to als.

Step 3. Return als.

4.4.2.13 Create-on-statement

Operation: create-on-statement(os)

where os is an {on-statement}.

result: an <on-statement>.

Step 1.

Case 1.1. os immediately contains SYSTEM.

Let p be a <system-action>.

Case 1.2. os immediate contains an {on-unit},ou.

Step 1.2.1.

Case 1.2.1.1. ou immediately contains an {executable-single-statement},ess.

Let s be the immediate component of ess.

Case 1.2.1.2. ou immediately contains a {begin-block}.

Let s be that {begin-block}.

Step 1.2.2. Perform create-abstract-equivalent-tree(s) to obtain sa. Let ep be an <entry-point> whose only subnode is <entry-information>. Let eul be an

```
<entry-or-executable-unit-list>:
     <entry-or-executable-unit>: ep;
     <entry-or-executable-unit>:
          <executable-unit>,eu: sa;;
     <entry-or-executable-unit>:
          <executable-unit>:
               <end-statement>.
```

Step 1.2.3. If ou immediately contains a {condition-prefix-commalist},cpl, perform create-condition-prefix-list(cpl) to obtain acpl, and attach acpl to eu.

Step 1.2.4. Let p be an <on-unit>: <procedure>: eul.

Step 2. Let aos be an <on-statement>: p. If os immediately contains SNAP attach <snap> to aos. For each {condition-name},cn in the {condition-name-commalist} in os, perform create-condition(cn) to obtain a <condition-name>,acn, and append acn to the <condition-name-list> in aos. Return aos.

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 sc 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},da then perform create-bound-pair-list(da) to obtain a <bound-pair-list>,bpl and attach bpl to nc.

Step 3.

Case 3.1. d declaration-contains ENTRY.

Perform create-entry(d) to obtain an <entry>,ae. Attach ae 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>, <update>, <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 4. Return nc.

4.4.3.2 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 4.4.5).

Operation: create-variable(d)

where d is a {declaration}.

result: 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 an {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-statement}s.

Case 1.5. d contains DEFINED ({reference},r) 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-statement}s.

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>,dd.

Step 3. Return a <variable>: st dd.

4.4.3.3 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-commalist} 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 {1}.

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 abp be a

<bound-pair>:
<lower-bound>,alb
<upper-bound>,aup.

Attach to alb an {expression-designator} designating the {expression} simply contained in the {lower-bound} component of bp. Attach to aup 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},ro then perform create-refer-option(ro) to obtain a <refer-option>,aro and attach aro to alb.

Step 3.1.2.3. If the {upper-bound} component of bp contains a {refer-option},ro perform create-refer-option(ro) to obtain a <refer-option>,aro and attach aro to aup.

Step 3.1.2.4. Append abp 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(da) to obtain a <bound-pair-list>,abpl.

Step 2.2. Attach <dimensioned-data-description>: <element-data-description> abpl; to dd.

Step 3. If d declaration-contains STRUCTURE then perform Steps 3.1 through 3.4.

Step 3.1. Let dl be the node that immediately contains d. Let e[j] be the j'th immediate component of dl that is not a {,}. 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 ld be the numeric value of the {level} of d.

Step 3.2. Let idl be an <identifier-list> and let mdl be a <member-description-list>.

Step 3.3. While i≤n perform Steps 3.3.1 through 3.3.4.

Step 3.3.1. If e[i] does not declaration-contain MEMBER then go to Step 3.4.

Step 3.3.2. Let le be the numeric value of the {level} of e[i]. If le≤ld then go to Step 3.4.

Step 3.3.3. If le=ld+1 then perform Steps 3.3.3.1 through 3.3.3.3.

Step 3.3.3.1. Perform create-data-description(e[i]) to obtain a <data-description>,add.

Step 3.3.3.2. Append a <member-description>: add; to mdl.

Step 3.3.3.3. If d is a {declaration} then let cid be the {identifier} of e[i], 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 sdd be a <structure-data-description>: mdl; otherwise let sdd be a <structure-data-description>: idl mdl. Attach sdd to dd.

Step 4. If d does not declaration-contain 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 al be <alignment>: <aligned>.

Case 4.1.2. d declaration-contains UNALIGNED.

Let al be <alignment>: <unaligned>.

Step 4.2. Perform create-data-type(d) to obtain a <data-type>,dt. Let idd be an <item-data-description>: al dt.

Step 4.3. If d declaration-contains {initial},int then let intd be an {initial-designator} that designates int and attach intd to idd.

Step 4.4. Attach idd to dd.

Step 5. Return dd.

4.4.3.5 Create-data-type

Operation: create-data-type(d)

where d is a {declaration}, a {description}, or a {generic-description} that is restricted to those forms that are equivalent to {description}s.

result: a <data-type>.

One and only one of the following cases must apply:

Case 1. d declaration-contains AREA({area-size},asz).

Step 1.1. Let ar be an <area>. If asz immediately contains an {*} then let ed be an <asterisk>; otherwise let ed be an {expression-designator} that designates the {expression} simple component of asz. Attach ed to ar.

Step 1.2. If asz contains a {refer-option},ro, perform create-refer-option(ro) to obtain a <refer-option>,aro and attach aro to ar.

Step 1.3. Return <data-type>: <non-computational-type>: ar.

Case 2. d declaration-contains ENTRY.

Perform create-entry(d) to obtain an <entry>,e and return a <data-type>: <non-computational-type>: e.

Case 3. d declaration-contains FILE.

Return a <data-type>: <non-computational-type>: <file>.

Case 4. d declaration-contains FORMAT or LABEL.

Attach <format> or <label> respectively to <data-type>,dt: <non-computational-type>. If d declaration-contains LOCAL then attach <local> to dt. Return dt.

Case 5. d declaration-contains POINTER or OFFSET[({reference},r)].

Let dt be a <data-type>: <non-computational-type>: <locator>,loc. Attach <pointer> or <offset>,ofs, respectively to loc. If r exists then attach a {reference-designator} that designates r to ofs.

Case 6. d declaration-contains PICTURE {pictur },pa.

Perform create-picture(pa) to obtain a <pictured>,p and return a <data-type>: <computational-type>: p.

Case 7. d declaration-contains a {data-attribute},sa which declaration-contains CHARACTER or BIT.

Step 7.1. If sa contains CHARACTER then let st be a <string-type>: <character>. Otherwise, let st be a <string-type>: <bit>.

Step 7.2. Let ml be the {maximum-length} component of sa. Let aml be a <maximum-length>. If ml immediately contains an {*} then let ed be an <asterisk>; otherwise let ed be an {expression-designator} that designates the {expression} simple component of ml. Attach ed to aml.

Step 7.3. If ml contains a {refer-option},ro then perform create-refer-option(re) to obtain a <refer-option>,aro and attach aro to aml.

Step 7.4. If d declaration-contains VARYING let v be <varying>; otherwise let v be <nonvarying>.

Step 7.5. Return a <data-type>: <computational-type>: <string>: st aml v.

Case 8. (Otherwise).

Step 8.1. Let dt be a

```
<data-type>:
  <computational-type>:
       <arithmetic>:
            <mode>,m
            <base>,b
            <scale>,s
            <precision>,p.
```

Step 8.2. If d declaration-contains REAL then attach <real> to m; otherwise, attach <complex>.

Step 8.3. If d declaration-contains BINARY then attach <binary> to b; otherwise, attach <decimal>.

Step 8.4. If d declaration-contains FIXED then attach <fixed> to s; otherwise, attach <float>.

Step 8.5. Let cp be the {precision} declaration-component of d. Perform create-abstract-equivalent-tree(cp) to obtain a <precision>,ap. The <number-of-digits> in ap must not be greater than the maximum <number-of-digits> allowed for the <base> and <scale> of dt. Replace p by ap.

Step 8.6. Return dt.

4.4.3.6 Create-entry

Operation: create-entry(d)

where d is a {declaration}, a {description}, or a {generic-description} that declaration-contains an ENTRY [(({description-commalist},dlo))].

result: an <entry>.

Step 1. Let ent be an <entry> with no subnodes.

Step 2. If dlo exists, then for each {description},pd in dlo that does not declaration-contain MEMBER, perform Steps 2.1 and 2.2.

Step 2.1. Perform create-data-description(pd) to obtain a <data-description>,dd.

Step 2.2. Append a <parameter-descriptor>: dd; to the <parameter-descriptor-list> in ent.

Step 3. If d declaration-contains RETURNS ({description-commalist},dc), perform Steps 3.1 through 3.3.

Step 3.1. Let rd be the {description} immediate component of dc that does not declaration-contain MEMBER. There must be exactly one such {description}.

Step 3.2. Perform create-data-description(rd) to obtain a <data-description>,dd.

Step 3.3. Attach a <returns-descriptor>: dd; to ent.

Step 4. If d contains OPTIONS, attach <options> with some implementation-defined subnodes, to ent.

Step 5. Return ent.

4.4.3.7 Create-refer-option

Operation: create-refer-option(cro)

where cro is a {refer-option}.

result: a <refer-option>.

Step 1. Let ur be the {unsubscripted-reference} of cro. Perform find-applicable-declaration(ur) to obtain d.

Step 2. Perform find-fully-qualified-name(d) to obtain an {identifier-list},idl.

Step 3. Perform create-abstract-equivalent-tree(idl) to obtain an <identifier-list>,aidl.

Step 4. Return a <refer-option>: aidl.

4.4.3.8 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.

4.4.3.9 Create-initial-element

Operation: create-initial-element(ine)

where ine is an {initial-element}.

result: an <initial-element>.

Step 1. Let aine be an <initial-element>.

Case 1.1. ine immediately contains * as its only component.

Attach an <asterisk> to aine.

Case 1.2. ine immediately contains a {parenthesized-expression},cpe.

Perform create-abstract-equivalent-tree(cpe) to obtain a <parenthesized-expression>,pe and attach pe to aine.

Case 1.3. ine 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>,ae, and attach a <parenthesized-expression>: ae; to aine.

Case 1.4. ine immediately contains an {iteration-factor},itf.

Perform create-abstract-equivalent-tree(itf) to obtain aif and attach aif to aine.

Case 1.4.1. ine immediately contains an *.

Attach an <initial-element-list>: <initial-element>: <asterisk>;; to aine.

Case 1.4.2. ine immediately contains an {initial-constant-two},ict.

Let e be the {expression} whose concrete-representation is the same as the concrete-representation of ict. Perform create-expression(e) to obtain an <expression>,ae. Attach an <initial-element-list>: <initial-element>: <parenthesized-expression>: ae;;; to aine.

Case 1.4.3. ine immediately contains an {initial-element-commalist},iec.

Perform create-abstract-equivalent-tree(iec) to obtain aiec, and attach aiec to aine.

Step 2. Return aine.

4.4.4 CREATE-EXPRESSION

Operation: create-expression(e,bnpl)

where e is an {expression}, {expression-seven}, {expression-six}, {expression-five}, {expression-four}, {expression-three}, {expression-two}, {expression-one}, {primitive-expression}, {prefix-expression}, or {parenthesized-expression},
bnpl is a [<by-name-parts-list>].

result: an <expression>.

Case 1. e is an {expression}, {expression-seven}, {expression-six}, {expression-five}, {expression-four}, {expression-three}, or an {expression-two} and e has only one component, ec.

Perform create-expression(ec,bnpl) to obtain an <expression>,aec. Return aec.

Case 2. e is an {expression}, {expression-seven}, {expression-six}, {expression-five}, {expression-four}, {expression-three}, or an {expression-one} and e has three components, e1, op, and e2.

Step 2.1. Perform create-expression(e1,bnpl) to obtain an <expression>,ae1. Perform create-expression(e2,bnpl) to obtain an <expression>,ae2.

Step 2.2. If op is or has |, &, >, >=, =, <=, <, ¬>, ¬=, ¬<, ||, +, -, *, /, ** then let aop be <or>, <and>, <gt>, <ge>, <eq>, <le>, <lt>, <le>, <ne>, <ge>, <cat>, <add>, <subtract>, <multiply>, <divide>, or <power> respectively.

Step 2.3. Let dd1 and dd2 be the <data-description>s immediately contained in ae1 and ae2, respectively.

The associated <aggregate-type>s of dd1 and dd2 must be compatible. Individual <data-type>s in dd1 and dd2 and corresponding <data-type>s in dd1 and dd2 must satisfy any constraints specified in the "Constraints" paragraphs of the section of Chapter 9 for the <infix-operator>,aop.

Let dd be a <data-description> whose associated <aggregate-type> is the common <aggregate-type> of dd1 and dd2 and whose <data-type>s are defined as "scalar-result-types" in the "Attributes" paragraphs of the section of Chapter 9 for the <infix-operator>,aop in terms of the corresponding <data-type>s in dd1 and dd2.

Step 2.4. Return an

```
<expression>:
   <infix-expression>:
      ae1
      <infix-operator>:
         aop;
      ae2
      dd;
   dd.
```

Case 3. e is a {primitive-expression}.

Let ec be the immediate component of e.

Case 3.1. ec is a {reference}.

Perform create-reference(ec,<value-reference>,bnpl) to obtain a <value-reference>,vr. Let dd be the <data-description> immediately contained in vr. Return an <expression>: vr dd.

Case 3.2. ec is a {constant}.

Perform create-constant(ec) to obtain a <constant>,c. Let dt be the <data-type> immediately contained in c. Return an

<expression>:
c
<data-description>:
<item-data-description>: dt.

Case 3.3. ec is an {isub}.

ec must be contained in an {attribute} that immediately contains DEFINED. Let i be an <integer> whose concrete-representation is the same as that of the {integer} in ec. Let dt be a <data-type> that is an integer-type. Return an

<expression>:
<isub>: i;
<data-description>:
<item-data-description>: dt.

Case 4. e is a {prefix-expression}: op e1.

Step 4.1. Perform create-expression(e1,bnpl) to obtain ae1.

Step 4.2. If op has ¬, +, - then let aop be <not>, <plus>, or <minus> respectively.

Step 4.3. Let dd1 be the <data-description> immediately contained in ae1.

The <data-type>s in dd1 must satisfy any constraints in the "Constraints" paragraphs of the section of Chapter 9 for the <prefix-operator>,aop.

Let dd be a <data-description> whose associated <aggregate-type> is the same as that of dd1 and whose <data-type>s are defined as "scalar-result-types" in the "Attributes" paragraphs of the section of Chapter 9 for the <prefix-operator>,aop in terms of the corresponding <data-type> in dd1.

Step 4.4. Return an

<expression>:
<prefix-expression>:
<prefix-operator>: aop;
ae1
dd;
dd.

Case 5. e is a {parenthesized-expression}: ({expression},e1).

Step 5.1. Perform create-expression(e1,bnpl) to obtain an <expression>,ae. Let dd be the <data-description> immediate component of ae.

Step 5.2. Return an

<expression>:
<parenthesized-expression>:
ae
dd;
dd.

4.4.5 CREATE-REFERENCE

Operation: create-reference(cr,targ,bnpl)

where cr is a {reference} or an {unsubscripted-reference},
targ is a <variable-reference>, <value-reference>, <subroutine-reference>, or <target-reference>,
bnpl is a [<by-name-parts-list>].

result: a tree of the same type as targ.

Step 1. If cr immediately contains an {arguments-list},ca, let al be a copy of ca. 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 <absent>.

Case 1.2. cd declaration-contains MEMBER.

Let rcd be the rightmost preceding {declaration} that does not declaration-contain MEMBER. Let dcl be the <declaration> whose {declaration-designator} designates rcd.

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 des be a <declaration-designator> designating dcl. Let dd be a copy of the <data-description> in dcl. Let ref be a <variable-reference>: des dd.

Step 2.1.2. Perform find-fully-qualified-name(cd) to obtain an {identifier-list},il. Perform create-abstract-equivalent-tree(il) to obtain an <identifier-list>,idl. Delete the first <identifier> in idl. If idl still contains any <identifier>s, attach idl to ref, and for each <identifier>,id, in idl, taken in order, perform Steps 2.1.2.1 through 2.1.2.3.

Step 2.1.2.1. Let dd be the <data-description> immediately contained in ref. It will have a <structure-data-description>,sdd. If dd has a <dimensioned-data-description>, let bpl be the <bound-pair-list> in dd; otherwise let bpl be a <bound-pair-list>. Let il be the <identifier-list> in sdd, and let mdl be the <member-description-list> in sdd. Let i be the integer such that the i'th <identifier> in il equals id, and then let mdd be the <data-description> immediately contained in the i'th <member-description> of mdl.

Step 2.1.2.2. If mdd has a <dimensioned-data-description>, append copies of the <bound-pair>s in mdd to bpl, and let tdd be the <item-data-description> or <structure-data-description> immediately contained in the <element-data-description> of mdd. Otherwise, let tdd be the <item-data-description> or <structure-data-description> immediately contained in mdd.

Step 2.1.2.3. If bpl contains <bound-pair>s, replace dd by a

```
<data-description>:
  <dimensioned-data-description>:
    <element-data-description>:
      tdd;
    bpl.
```

Otherwise replace dd by a <data-description>: tdd.

Step 2.1.3. If cr is a {reference}, then let br be the {basic-reference} immediately contained in cr, and perform collect-subscripts(br) to obtain a {subscript-commalist},sl. Otherwise let sl be a {subscript-commalist}.

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-subscripts(cr,ref,sl,al).

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-commalist}, and perform apply-subscripts(cr,ref,sl,al).

Case 2.3. dcl is a <declaration> that has <builtin>.

Step 2.3.1.

Case 2.3.1.1. al has an {arguments}.

In this case, al must have only one {arguments}. Let ar be the {arguments} in ca. Perform create-argument-list(ar) to obtain args. Delete the {arguments} from al.

Case 2.3.1.2. al does not have an {arguments}.

Let args be <absent>.

Step 2.3.2.

Case 2.3.2.1. targ is a <value-reference>.

Perform create-builtin-function-reference(dcl,args) to obtain a <builtin-function-reference>,ref.

Case 2.3.2.2. targ is a <target-reference>.

Perform create-pseudo-variable-reference(dcl,args) to obtain a <pseudo-variable-reference>,ref.

Case 2.3.2.3. targ 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. al 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. al does not have an {arguments}.

In this case targ 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},lq.

Let r be the {reference} immediately contained in lq. Perform create-reference(r,<value-reference>) to obtain a <value-reference>,vr. All the following conditions must hold:

(1) ref must be a <variable-reference>;

(2) the <declaration> designated by the <declaration-designator> immediately contained in ref must have <based>;

(3) the <data-description> immediately contained in vr must immediately contain an <item-data-description> whose <data-type> must have <locator>, and if the <data-type> has an <offset>,os, then os must have a <variable-reference> or a {reference-designator}.

Attach a <locator-qualifier>: vr; to ref.

Case 3.2. cr does not immediately contain a {locator-qualifier} and dcl is a <declaration> that has <variable>.

If dcl contains a <based>,b, then b must immediately contain a <value-reference> or a {reference-designator}.

Case 3.3. (Otherwise).

No action.

Step 4. If al contains an {arguments}, perform Steps 4.1 to 4.4.

Step 4.1. ref must not be a <subroutine-reference>. Perform create-value-reference(ref) to obtain evr. The <data-description> immediate component of evr must immediately contain an <item-data-description> whose <data-type> must have <entry>.

Step 4.2. Let ar be the first {arguments} in al. Perform create-argument-list(ar) to obtain args. Delete ar from al.

Step 4.3. Perform create-entry-reference(evr,args) to obtain ref.

Step 4.4. Go to Step 4.

Step 5. If bnpl in not «absent» and ref is a <variable-reference> then perform apply-by-name-parts(ref,bnpl) to obtain a <variable-reference>,ref. If bnpl is not «absent» and ref is not a <variable-reference>, then the <data-description> of ref must immediately contain <item-data-description>.

Step 6.

Case 6.1. targ is a <variable-reference>.

In this case ref must be a <variable-reference>. Return ref.

Case 6.2. targ is a <value-reference>.

ref must not be a <subroutine-reference>. Perform create-value-reference(ref) to obtain a <value-reference>,vr. Return vr.

Case 6.3. targ is a <subroutine-reference>.

If ref is a <subroutine-reference>, return ref. Otherwise perform create-value-reference(ref) to obtain a <value-reference>,vr whose immediately contained <data-description> must immediately contain an <item-data-description> whose <data-type> must have <entry>. Perform create-entry-reference(ref) to obtain sr, which must be a <subroutine-reference>. Return sr.

Case 6.4. targ is a <target-reference>.

In this case ref must be a <variable-reference> or <pseudo-variable-reference>. Let dd be a copy of the <data-description> immediately contained in ref. If ref is a <variable-reference>, perform trim-dd(dd). Return a <target-reference>: ref dd.

4.4.5.1 Collect-subscripts

Operation: collect-subscripts(br)

where br is a {basic-reference}.

result: a {subscript-commalist} which may have no components.

Case 1. br has a {structure-qualification},sq.

Let br2 be the {basic-reference} immediately contained in sq. Perform collect-subscripts(br2) to obtain a {subscript-commalist},sl. If sq has an {arguments},args, then args must have a {subscript-commalist},sl2, append the {subscript}s in sl2 to sl, appending {,} as required. Finally, return sl.

Case 2. br does not have a {structure-qualification}.

Return a {subscript-commalist}.

4.4.5.2 Apply-by-name-parts

Operation: apply-by-name-parts(vro,bnp)

where vro is a <variable-reference>,
bnp is a <by-name-parts-list>.

result: a <variable-reference>.

Case 1. The <data-description>,dd immediate component of vro has a <structure-data-description>.

Step 1.1. Let vr be a copy of vro. Let cdd be a

<data-description>:
<structure-data-description>:
<member-description-list>,mdla.

For each <identifier-list>,idl component of bnp, taken in order, perform Steps 1.1.1 and 1.1.2.

Step 1.1.1. For each <identifier>,id in idl, taken in order, perform Steps 1.1.1.1 through 1.1.1.3.

Step 1.1.1.1. dd will have a <structure-data-description>,sdd. If dd has a <dimensioned-data-description>, let bpl be the <bound-pair-list> in dd; otherwise let bpl be a <bound-pair-list> with no component. Let il be the <identifier-list> in sdd, and let mdl be the <member-description-list> in sdd. Let i be the integer such that the i'th <identifier> in il equals id, and let mdd be the <data-description> in the i'th <member-description> of mdl.

Step 1.1.1.2. If mdd has a <dimensioned-data-description>, append copies of the <bound-pair>s in mdd to bpl, and let tdd be the <item-data-description> or <structure-data-description> immediately contained in the <element-data-description> of mdd. Otherwise, let tdd be the <item-data-description> or <structure-data-description> immediately contained in mdd.

Step 1.1.1.3. If bpl contains <bound-pair>s then let dd be a

<data-description>:
<dimensioned-data-description>:
<element-data-description>:
tdd;
bpl.

Otherwise let dd be a <data-description>: tdd.

Step 1.1.2. Append dd to mdla.

Step 1.2. Replace dd by cdd and append bnp to vr.

Step 1.3. Return vr.

Case 2. (Otherwise).

Step 2.1. Return vro.

4.4.5.3 Apply-subscripts

Operation: apply-subscripts(cr,ref,sl,al)

where cr is a {reference},
ref is a <variable-reference> or a <named-constant-reference>,
sl is a {subscript-commalist},
al is an {arguments-list}.

Step 1. Let dd be the <data-description> immediately contained in ref. Let m be the number of <bound-pair>s in dd, and let n be the number of {subscript}s in sl (n may be 0). If n < m, and if al has a first immediate component, args, that has a {subscript-commalist}, then perform Step 1.1.

Step 1.1. Append the {subscript}s in args to sl, appending {,} as required. Delete args from al.

Step 2. Let n be the number of {subscript}s in sl which are not in any {expression} also in sl.

Case 2.1. n = 0.

Attach a <subscript-list> containing m occurrences of <asterisk> to ref.

Case 2.2. n ≠ 0.

Step 2.2.1. In this case n must equal m. Attach a <subscript-list>,sl2 to ref. For i=1,...,n, perform Step 2.2.1.1.

Step 2.2.1.1. Of those {subscript}s contained in cr which are not contained in any {expression} contained in cr, let s be the i'th one. If s immediately contains an {expression},e, perform create-expression(e) to obtain an <expression>,e2, and append a <subscript>: e2; to sl2. Otherwise append a <subscript>: <asterisk>; to sl2.

Step 2.2.2. For i=n,...,1, perform Step 2.2.2.1.

Step 2.2.2.1. If the i'th <subscript> in sl2 contains an <expression>, delete the i'th <bound-pair> in dd.

Step 2.2.3. If the <bound-pair-list> in dd now has no <bound-pair>s, let dd2 be the <item-data-description> or <structure-data-description> in dd, and then replace dd by a <data-description>: dd2.

4.4.5.4 Create-value-reference

Operation: create-value-reference(ref)

where ref is a <variable-reference>, <procedure-function-reference>, <builtin-function-reference>, or <named-constant-reference>.

result: a <value-reference>.

Step 1. Let dd be a copy of the <data-description> immediately contained in ref.

Step 2. If ref is a <variable-reference> perform trim-dd(dd). Return a <value-reference>: ref dd.

4.4.5.4.1 Trim-dd

Operation: trim-dd(dd)

where dd is a <data-description>.

Step 1. Delete from dd any occurrences of the following categories that are not contained in an <entry>:

<alignment>,
<initial>,
<identifier-list>,
<local>,
<options>,
<varying>,
<nonvarying>.

Replace by an <asterisk> the immediate component of each <maximum-length>, <area-size> or <bound-pair> that is not a component of an <entry>.

4.4.5.5 Create-named-constant-reference

Operation: create-named-constant-reference(dcl)

where dcl is a <declaration>.

result: a <named-constant-reference>.

Step 1. Let ct be a copy of the leftmost immediate component of the <named-constant> component of dcl. Let dt be an

<item-data-description>:
<data-type>:
<non-computational-type>:
ct.

Step 2.

Case 2.1. dcl contains a <bound-pair-list>,bp.

Let n be the number of <bound-pair>s in bp. Let bpl be a <bound-pair-list> containing n subnodes <bound-pair>: <asterisk>. Let dd be a

<dimensioned-data-description>:
<element-data-description>:
dt;
bpl.

Case 2.2. (Otherwise).

Let dd be dt.

Step 3. Let ddg be a <declaration-designator> designating dcl. Return a

<named-constant-reference>:
ddg
<data-description>:
dd.

4.4.5.6 Create-argument-list

Operation: create-argument-list(al)

where al is an {arguments}.

result: an <argument-list> or <absent>.

Step 1. If al does not contain a {subscript-commalist} then return <absent>; otherwise let scl be the {subscript-commalist} immediately contained in al.

Step 2. Let n be the number of {subscript} immediate components of scl. The {subscript} immediate components of scl must not immediately contain *.

Step 3. Let xal be an <argument-list> and let ec[i], i=1,...,n be the {expression} simple components of scl taken in left-to-right order.

Step 4. For each element, ec[i], i=1,...,n, perform Steps 4.1 and 4.2.

Step 4.1. Perform create-expression(ec[i]) to obtain an <expression>,eac[i].

Step 4.2. Let rdd be the <data-description> immediately contained in eac[i]. Append to xal an

```
<argument>:
     eac[i] rdd.
```

Step 5. Return xal.

4.4.5.7 Create-builtin-function-reference

A builtin-function-name is a sequence of uppercase letters and digits such that the corresponding sequence of lowercase letters and digits followed by "-bif" is the category-name of a subnode of <builtin-function>.

Operation: create-builtin-function-reference(ad,al)

where ad is a <declaration>,
al is an [<argument-list>].

result: a <builtin-function-reference>.

Step 1. Let id be the <identifier> contained in ad. There must be a <builtin-function>,bf, whose name or abbreviation (as listed in Section 2.7) corresponds to the concrete-representation of id. Let bfr be a <builtin-function-reference>: bf.

Step 2. The number of <argument>s in al must be as shown in the "Arguments" section of the description of bf (see Chapter 9).

Step 3.

Case 3.1. al is not <absent>.

Step 3.1.1. All <data-description>s immediately contained in the <argument>s in al must satisfy the constraints given in the "Constraints" section of the same builtin-function description.

Step 3.1.2. Append al to bfr.

Case 3.2. (Otherwise).

No action.

Step 4. Construct a <data-description>,rdd, as specified in the "Attributes" section of the same <builtin-function> description. Append rdd to bfr.

Step 5. Return bfr.

4.4.5.8 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>,pv whose name corresponds to the concrete-representation of id. Let pvr be a <pseudo-variable-reference>: pv.

Step 2. The number of <argument>s 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 <absent>.

Step 3.1.1. All <data-description>s immediately contained in the <argument>s in al must satisfy the constraints given in the "Constraints" section of the description of pv.

Step 3.1.2. Attach al to pvr.

Case 3.2. (Otherwise).

No action.

Step 4. Construct a <data-description>,rdd, as specified in the "Attributes" section of the same <pseudo-variable> description. Attach rdd to pvr.

Step 5. Return pvr.

4.4.5.9 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 <subroutine-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 <absent>. The number of elements in pdl must be equal to the number of elements in al.

Case 2.2. (Otherwise).

al must be <absent>. 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 pdd 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>,var. Perform test-matching(var,pdd) to obtain tv. If tv is <false>, then attach <dummy> to arg.

Case 3.2.2. (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 assignment to pdd (see Section 4.4.2.3).

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>: vr [al] rde.

Case 4.2. (Otherwise).

Return <subroutine-reference>: vr [al].

4.4.5.10 Test-matching

Operation: test-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> designated by the <declaration-designator> in var. If dcl contains a <defined> whose <base-item> contains an <isub>, return <false>.

Step 2. Let dd be a copy of the <data-description> immediately contained in var. Let pdd be a copy of pd.

Step 3. If any of the following subtrees exist as a component of dd or pdd 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 pdd and dd are not equal, disregarding comparison of the subnodes of any <maximum-length>, <bound-pair>, or <area-size>, then return <false>.

Step 5. For each <extent-expression>,e1, in pdd, 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 ex1 and ex2 be the <expression>s in e1 and e2, respectively. If ex1 or ex2 contains a <declaration-designator>, return <false>. Perform evaluate-restricted-expression(ex1) and evaluate-restricted-expression(ex2) to obtain v1 and v2. If v1 or v2 is not a <constant> having <computational-type>, return <false>. Otherwise, perform evaluate-expression-to-integer(ex1) and evaluate-expression-to-integer(ex2) to obtain <integer-value>s i1 and i2 respectively. If i1 and i2 are equal, return <true>; otherwise return <false>.

Step 6. Return <true>.

4.4.5.11 Select-generic-alternative

Operation: select-generic-alternative(ga,arg)

where ga is a {generic-attribute},
arg is an [<argument-list>].

result: a <value-reference>.

Step 1.

Case 1.1. arg is <absent>.

ga must have at least one {generic-element} that does not have a {generic-description-commalist}. Let rr be the {reference} immediately contained in the first such {generic-element}.

Case 1.2. arg is an <argument-list>.

Step 1.2.1. Let na be the number of <argument>s in arg. There must be at least one {generic-element} in ga. Let ge be the first such {generic-element}.

Step 1.2.2. Let ngd be the number of {generic-description}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,...,na, perform Steps 1.2.3.1 and 1.2.3.2.

Step 1.2.3.1. Let ai be the i'th <argument> in arg.

Let gd be the i'th {generic-description} in ge that does not declaration-contain MEMBER and let gdl be {generic-description-list}: gd.

Case 1.2.3.1.1. There is a {generic-description} in ge that follows gd and does not declaration-contain MEMBER.

Let gdf be the leftmost such {generic-description}. Append to gdl, in left-to-right order, copies of all {generic-description}s in ge between gd and gdf.

Case 1.2.3.1.2. (Otherwise).

Append to gdl in left-to-right order, copies of all {generic-description}s in ge following gd.

Step 1.2.3.2. Perform test-generic-matching(ai,gdl) to obtain tval. If tval is <false> go to Step 1.2.5.

Step 1.2.4. Let rr be the {reference} immediately contained in ge. Go to Step 2.

Step 1.2.5. Let ge be the next {generic-element} of ga following the current ge. There must be such a {generic-element}. Go to Step 1.2.2.

Step 2. Perform find-applicable-declaration(rr) to obtain a {declaration},dcl, which must not contain a {generic-attribute}.

Step 3. Perform create-reference(rr,<value-reference>) to obtain vr. Return vr.

4.4.5.12 Test-generic-matching

Operation: test-generic-matching(ai,gdl)

where ai is an <argument>,
gdl is a {generic-description-list}.

result: <true> or <false>.

Case 1. gdl contains only {generic-description}: *.

Return <true>.

Case 2. (Otherwise).

Step 2.1. If ai is of the form

<argument>:
<expression>:
<value-reference>:
<variable-reference>,vr;;;

then let dd be the <data-description> immediately contained in vr. Otherwise, let dd be the <data-description> immediately contained in ai.

Step 2.2. Perform test-generic-aggregation(gdl,dd) to obtain tval. Return tval.

4.4.5.13 Test-generic-aggregation

Operation: test-generic-aggregation(gdl,dd)

where gdl is a {generic-description-list},
dd is a <data-description>.

result: <true> or <false>.

Step 1. Let gd be the first {generic-description} component of gdl. If gd declaration-contains DIMENSION then perform Steps 1.1 through 1.3.

Step 1.1. If dd does not immediately contain a <dimensioned-data-description> then return <false>.

Step 1.2. If the number of * components of the {*-commalist} of gd is not equal to the number of <bound-pair> components of the <bound-pair-list> of dd, then return <false>.

Step 1.3. Delete the DIMENSION {asterisk-bounds} declaration-contained in gd.

Step 2. If dd immediately contains a <dimensioned-data-description> then let dd be a <data-description>: tdd; where tdd is the immediate component of the <element-data-description> of dd.

Step 3.

Case 3.1. gd declaration-contains STRUCTURE.

Step 3.1.1. Let lv be the value of the {level} of gd. Let nl be the number of {level} components of gdl with value equal to lv+1 which follow gd without any intervening component of gdl whose {level} has value less than or equal to lv.

Step 3.1.2. If dd does not immediately contain a <structure-data-description>,sdd then return <false>. Let ndd be the number of <data-description> simple components of sdd. If ndd does not equal nl then return <false>.

Step 3.1.3. For i=1,...,nl perform Steps 3.1.3.1 to 3.1.3.4.

Step 3.1.3.1. Let tgdl be a copy of gdl. Delete from tgdl the {generic-description} immediate components that precede the i'th {generic-description} whose {level} has value lv+1.

Step 3.1.3.2. Let tdd be the i'th <data-description> of dd.

Step 3.1.3.3. Perform test-generic-aggregation(tgdl,tdd) to obtain tval.

Step 3.1.3.4. If tval 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 tval.

Step 4. Return tval.

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 <nonvarying> then delete any VARYING or NONVARYING 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 absatr.

Step 4.2. If dd does not simply contain a node whose node type is the same as absatr 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 gdal contains a {generic-data-attribute},gda which immediately contains a {generic-precision},gprec then perform Steps 5.1 to 5.3.

Step 5.1. If dd does not have <precision> or if dd has <pictured>, then return <false>.

Step 5.2. Let prec be the <precision> of dd. Perform test-generic-precision(gprec,prec) to obtain tval.

Step 5.3. If tval is <false> then return <false>. Otherwise delete gda from gdal.

Step 6. If gdal contains a {generic-data-attribute},gda which immediately contains a {description-commalist}, 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 pdl be the <parameter-descriptor-list> simply contained in dd.

Step 6.3. gd must not contain any {identifier} components. Perform create-data-description(gd) to obtain a <data-description>,gdd. Perform replace-concrete-designators(gdd). Let gpdl be the <parameter-descriptor-list> in gdd. For each <parameter-descriptor>,gpd in gpdl, perform validate-descriptor(gpd).

Step 6.4. If gpdl is not equal to pdl then return <false>. Otherwise delete gda from gdal.

Step 7. If gdal 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 rd be the <returns-descriptor> simply contained in dd.

Step 7.3. gd must not contain any {identifier} components. Perform create-data-description(gd) to obtain a <data-description>,gdd. Perform replace-concrete-designators(gdd). Let grd be the <returns-descriptor> in gdd. Perform validate-descriptor(grd).

Step 7.4. If grd is not equal to rd then return <false>. Otherwise delete gda from gdal.

Step 8. If gdal 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 <pictured> then return <false>.

Step 8.2. Let pd be the <pictured> component of dd.

Step 8.3. Perform create-picture(p) to obtain a <pictured>,ap.

Step 8.4. If ap is not equal to pd then return <false>; otherwise delete gda from gdal.

Step 9. All nodes must have been deleted from gdal. Return <true>.

Table 4.1. Concrete Terminals of Significance to Test-generic-description.

ALIGNED	COMPLEX	FLOAT	POINTER
AREA	DECIMAL	FORMAT	PRECISION
BINARY	ENTRY	LABEL	REAL
BIT	FILE	NONVARYING	UNALIGNED
CHARACTER	FIXED	OFFSET	VARYING

4.4.5.15 Test-generic-precision

Operation: test-generic-precision(gprec,prec)

where gprec is a {generic-precision},
prec is a <precision>.

result: <true> or <false>.

Step 1.

Case 1.1. gprec contains {number-of-digits} {:} {number-of-digits}.

Let gplo be the value of the {integer} component of the first {number-of-digits} of gprec and let gphi be the value of the {integer} component of the second {number-of-digits} component of gprec.

Case 1.2. (Otherwise).

Let gplo and gphi both be the value of the {integer} component of the sole {number-of-digits} component of gprec.

Step 2. Let p be the value of the <integer> component of the <number-of-digits> of prec. If it is not the case that gplo ≤ p ≤ gphi then return <false>.

Step 3.

Case 3.1. gprec does not have a {scale-factor} and prec does not have a <scale-factor>.

Return <true>.

Case 3.2. gprec has a {scale-factor} and prec does not have a <scale-factor>.

Return <false>.

Case 3.3. gprec has {scale-factor} {:} {scale-factor}.

Let gslo be the value of the {integer} component of the first {scale-factor} of gprec and let gshi be the value of the {integer} component of the second {scale-factor} of gprec.

Case 3.4. (Otherwise).

Let gslo and gshi both be the value of the {integer} component of the sole {scale-factor} component of gprec.

Step 4. Let s be the value of the <signed-integer> component of the <scale-factor> of prec.

Step 5. If it is the case that gslo ≤ s ≤ gshi then return <true>; otherwise return <false>.

4.4.6 CREATE-PICTURE

A {picture} may occur as a component of a {declaration} or a {format-item}. In both cases it is translated to a <pictured>. 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 <pictured-character> or <pictured-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-element}::= A|X|9|C|D|I|R|S|T|V|Y|Z|$|+|-|*|E|K|B|/|.|,

{picture-scale-factor}::= F ({signed-integer})

Operation: create-picture(p)

where p is a {picture}.

result: a <pictured>.

Step 1. There must be a {string-or-picture-symbol-list},spsl in p. Let t be a {picture-content} whose concrete-representation is the same as that of spsl. 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 iv be the decimal value of the {integer} in rf. iv must not be equal to zero. Replace c by iv 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-contain REAL or COMPLEX. All terminal nodes of t must be A, X, or 9. Return <pictured>: <pictured-character>: <character-picture-element-list>,cpel;; where the concrete-representation of cpel is the same as that of t.

Case 3.2. (Otherwise).

Perform create-numeric-picture(t) to obtain a <pictured-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 <pictured>: pn.

4.4.6.1 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}]] |
             V {pic-digit-list} |
             {Z-list} [{scaled-digits-field}] |
             [{Z-list}] V {Z-list} |
             {*-list} [{scaled-digits-field}] |
             [{*-list}] V {*-list}

{scaled-digits-field}::= {pic-digit-list} [ V [{pic-digit-list}]] |
                         V [{pic-digit-list}]

{pic-digit}::= 9 | I | R | T | Y

{sign}::= S | + | -

{drifting-field}::= [ {drifting-sign-field} . [$] ] |
                    [ {drifting-dollar-field} . [{sign}] ] |
                      {drifting-dollar-field}  [{credit} | {debit}]

{credit}::= CR

{debit}::= DB

{drifting-sign-field}::= {signs} [{scaled-digits-field}] |
                         S [{S-list}] V {S-list} |
                         + [{plus-list}] V {plus-list} |
                         - [{minus-list}] V {minus-list}

{plus}::= +

{minus}::= -

{signs}::= S {S-list} | {plus} {plus-list} | {minus} {minus-list}

{drifting-dollar-field}::= $ {$-list} [{scaled-digits-field}] |
                           $ [{$-list}] V {$-list}

{floating-point-picture}::= {pic-mantissa} {pic-exponent}

{pic-mantissa}::= [{sign}] {digits} | {drifting-sign-field}

{pic-exponent}::= [E|X] [{sign}] [ {pic-digit-list} |
                  [[ {Z-list} | {*-list} ] [{pic-digit-list}] ]]
```

A digit-position is any occurrence of a {pic-digit} or Z or *, or any occurrence of S, +, -, or $ in an {S-list}, {plus-list}, {minus-list}, or {$-list}.

Operation: create-numeric-picture(p)

where p is of the form {picture-content}: {picture-item-list},pil [{picture-scale-factor},psf].

result: a <pictured-numeric>.

Step 1. Let pilw be a copy of pil. Delete from pilw any {picture-item} which contains a {picture-element} containing a

{.}, or
{,}, or
/, or
B if that {picture-item} does not immediately follow a {picture-item}: {picture-element}: D.

A terminal node so deleted must not have occurred immediately between a C and an R nor immediately between a D and a B.

Step 2. It must be possible to construct a {numeric-picture-specification},nps according to the picture-validation syntax (above), such that the concrete-representation of nps is the same as that of pilw.

If psf exists, then nps must contain a {fixed-point-picture}.

Step 3. Let pn be a partial tree

<pictured-numeric>:
<numeric-picture-specification>,anps
<arithmetic>:
<mode>:
<real>;
<base>:
<decimal>;
<scale>,s
<precision>,prec:
<number-of-digits>: n;;;;

where n is the number of digit-positions in {fixed-point-picture} or in {pic-mantissa}, in nps. n must not be greater than the maximum <number-of-digits> for <base>: <decimal>; and <scale> to be set below.

Note: The node <real> may be replaced by <complex> at a later stage.

Step 4. If nps contains a {floating-point-picture} then attach <float> to s; otherwise attach <fixed> to s.

Step 5.

Case 5.1. nps contains a {fixed-point-picture}.

Step 5.1.1. nps must not contain more than one {pic-digit} which has a T, I, or R. nps must not contain a {pic-digit} which has a T, I, or R if it also contains S, +, -, {credit}, or {debit}. Attach to anps a <fixed-point-picture> with the same concrete-representation as pil.

Step 5.1.2. If psf exists, then let v be the result of interpreting the {signed-integer},si, in psf as a decimal constant, and attach to anps a <picture-scale-factor> containing the <signed-integer> abstract-equivalent of si; otherwise, let v=0. Let v1=(n-v) where n is the number of digit-positions after the V in nps if V appears, and 0 otherwise. Attach a <scale-factor>: v1; to prec.

Case 5.2. nps contains a {floating-point-picture}.

Step 5.2.1. Let pm be a <numeric-picture-element-list> whose concrete-representation is the same as that of pil up to (but not including) the E or K. pm must not contain any of T, I, or R.

Step 5.2.2. Let pe be a <numeric-picture-element-list> whose concrete-representation is the same as that of pil beginning with the E or K. pe must not contain any of T, I, or R.

Step 5.2.3. Attach to anps a

```
<floating-point-picture>:
    <picture-mantissa>:
        pm;
    <picture-exponent>:
        pe.
```

Step 6. Return pn.

4.4.7 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 {symbol}s (in order) have the same concrete-representations as the {string-or-picture-symbol}s in c, except that each {string-or-picture-symbol}: {''}; in c becomes a {symbol}: {'}; in csv.

Step 1.1.2.2. If c contains a {replicated-string-constant},rsc, perform Step 1.1.2.2.1.

Step 1.1.2.2.1. Let j be the value obtained by interpreting the {integer} in rsc as a decimal constant; let i be the number of <character-value>s in csv.

If j=0, let v be a <null-character-string>; otherwise, let v be a <character-value-list> with i*j components, the (n*i+k)'th component equalling the k'th <character-value> of csv, for k=1,...,i, and n=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>, <nonvarying>, and <maximum-length>: <asterisk>.

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 s(i), i=1,...,k, be the {string-or-picture-symbol}s in c.

Each s(i) must have an entry in Table 4.2 which is valid for the value of m. Let bsv be a <bit-string-value> containing m*k <bit-value>s, such that <bit-value>s (i*m+1-m) through (i*m) are obtained from Table 4.2 as a function of m and s(i), 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 the {integer} in rsc as a decimal constant; let i be the number of <bit-value>s in bsv.

If j=0, let v be a <null-bit-string>; otherwise, let v be a <bit-value-list> with i*j components, the (n*i+k)'th component equalling the k'th <bit-value> of bsv, for k=1,...,i, and n=0,...,(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 <bit>, <nonvarying>, and <maximum-length>: <asterisk>.

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>: <real-value>,v <data-type>,t.

Step 3.2. Let ds be a partial {declaration} containing:

```
CONSTANT
if c contains F, then FIXED
if c contains E, then FLOAT
if c contains I, then COMPLEX
     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: {declare-statement},d; to the {procedure} or {begin-block} which block-contains c, attach a copy, cds, of ds to d, perform apply-defaults(cds), let ds be a copy of cds, 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-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 3.5.

Case 3.5.1. ac immediately contains a {real-constant}.

Perform convert(dt,t,v) to obtain a <real-value>,rv. Let bv be a <basic-value>: rv.

Case 3.5.2. ac contains an {imaginary-constant}.

Let rdt be a <data-type> which has <real> but is otherwise as dt. Perform convert(rdt,t,v) to obtain a <real-value>,rv. Let bv be a <basic-value>: <complex-value>; with real part: 0; and imaginary part: rv.

Step 3.6. Return a <constant>: bv dt.

Table 4.2. Table of <bit-value>s as a Function of {symbol}s and {radix-factor}s for Create-constant.

Contents of i'th {symbol} or {string-or-picture-symbol}	Contents of <bit-value>s (i*m+1-m) through (i*m)			
	m=1	m=2	m=3	m=4
0	0	00	000	0000
1	1	01	001	0001
2	-	10	010	0010
3	-	11	011	0011
4	-	-	100	0100
5	-	-	101	0101
6	-	-	110	0110
7	-	-	111	0111
8	-	-	-	1000
9	-	-	-	1001
A	-	-	-	1010
B	-	-	-	1011
C	-	-	-	1100
D	-	-	-	1101
E	-	-	-	1110
F	-	-	-	1111
Other	-	-	-	-

- indicates that the corresponding {symbol} or {string-or-picture-symbol} is invalid for this value of m
0 indicates <zero-bit>
1 indicates <one-bit>

4.5 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: validate-procedure(ap)

where ap is a <procedure>.

Step 1. Perform apply-constraints(ap).

Step 2. For each <declaration>,ad component of ap perform validate-declaration(ad).

Step 3. For each <parameter-descriptor>,d and for each <returns-descriptor>,d contained in ap perform 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 6.3.4.

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 such <declaration>s do represent realistic data entities.

Operation: validate-declaration(ad)

where ad is a <declaration>.

Step 1. If ad contains <automatic>, <based>, <controlled>, <defined>, <parameter>, or <static> then perform validate-automatic-declaration(ad), validate-based-declaration(ad), validate-controlled-declaration(ad), validate-defined-declaration(ad), validate-parameter-declaration(ad), or validate-static-declaration(ad), respectively.

Step 2. If ad contains <named-constant>, then perform validate-static-declaration(ad).

Step 3. If ad contains any <data-description>,dd which simply contains <initial>, then for each such dd perform Step 3.1.

Step 3.1. The <data-description> 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: validate-automatic-declaration(ad)

where ad is a <declaration>.

Step 1. ad must not contain an <area-size>: <asterisk>;, a <maximum-length>: <asterisk>;, or a <bound-pair>: <asterisk>;, except as subnodes of <entry>.

Step 2. Each <extent-expression> in ad must not contain a <refer-option>.

Step 3. If ad contains a <variable-reference>: <declaration-designator>,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>s 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>,ms.

Step 2.2. Let id[1],...,id[n] be the components of the <identifier-list> in ro. id[1] must be equal to the <identifier> immediately contained in bd; bd must simply contain a <structure-data-description>,sd[1]: idl[1] mdl[1]. For i=1,...,(n-1), idl[i] must have an <identifier>,idc[i] equal to id[i+1]; for i=1,...,(n-2), the <member-description> in mdl[i] corresponding to idc[i] in idl[i] must simply contain a <structure-data-description>,sd[i+1]: idl[i+1] mdl[i+1]; the <member-description>,mo in mdl[n-1] corresponding to idc[n-1] in idl[n-1] must immediately contain a <data-description>: <item-data-description>: <data-type>: <computational-type>;;; and mo must not be contained in a <dimensioned-data-description>.

Step 2.3. mo must occur to the left of ms in bd.

Step 2.4. For every <structure-data-description>,sdd other than sd[1] which contains both mo and ms perform Step 2.4.1.

Step 2.4.1. For every <item-data-description>,idd1 which is contained in sdd and is to the right of ms, there must exist at least one <item-data-description>,idd2 contained in sdd, which either is simply contained in ms or is to the left of ms, such that idd1 and idd2 match as defined in Step 2.4.1.1.

Step 2.4.1.1. Let idd1' and idd2' be copies of idd1 and idd2, modified as follows. Delete any occurrences of <initial>, <maximum-length>, <number-of-digits>, and <local>; delete any subnodes of <offset>, <entry>, and <area>; replace any occurrences of <pictured> by <string>: <string-type>: <character>; <nonvarying>.

For idd1 and idd2 to match, idd1' and idd2' must be equal, and, if idd1 and idd2 have <arithmetic>, the <number-of-digits> of idd1 must be less than or equal to the <number-of-digits> of idd2.

4.5.4 VALIDATE-CONTROLLED-DECLARATION

Operation: validate-controlled-declaration(cd)

where cd is a <declaration>.

Step 1. cd must not contain an <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>, e simple component of an <extent-expression> of cd, perform Step 3.1.

Step 3.1. Perform evaluate-restricted-expression(e) to obtain c. If c is a <constant> having <computational-type> then replace the first immediate component of e by c. (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: validate-defined-declaration(dd)

where dd is a <declaration>.

Step 1. Perform validate-automatic-declaration(dd).

Step 2. Let dd contain the form <defined>,def: <base-item>: <variable-reference>,vr. If vr contains an <isub> then def must not contain a <position> and vr must not contain an <asterisk>.

Step 3. Each <declaration-designator> in dd must not designate dd.

Step 4. The <data-description> immediately contained in vr and the <data-description> immediately contained in the <variable> in dd must not contain <varying>, other than as a subnode of <entry>.

4.5.6 VALIDATE-PARAMETER-DECLARATION

Operation: validate-parameter-declaration(pd)

where pd is a <declaration>.

Step 1. Each <extent-expression> in pd must not contain a <refer-option>.

Step 2. For each <extent-expression>: <expression>,e; in pd perform Steps 2.1.

Step 2.1. Perform evaluate-restricted-expression(e) to obtain c. c must be a <constant>. Replace the first immediate component of e by c.

4.5.7 VALIDATE-STATIC-DECLARATION

Operation: validate-static-declaration(sd)

where sd is a <declaration>.

Step 1. Perform validate-automatic-declaration(sd).

Step 2. For each <expression>,e simple component of sd which is not a component of an <offset> perform evaluate-restricted-expression(e) to obtain c, which must be a <constant> or a <value-reference>. If c is a <value-reference> then it must be a component of an <initial-element>. Replace the first immediate component of e by c.

4.5.8 VALIDATE-DESCRIPTOR

Operation: validate-descriptor(d)

where d is a <parameter-descriptor> or a <returns-descriptor>.

Step 1. For each <extent-expression>: <expression>,e; in d, perform Step 1.1.

Step 1.1. Perform evaluate-restricted-expression(e) to obtain c. c must be a <constant>. Replace the first immediate component of e by c.

Step 2. If d is a <parameter-descriptor>, any <entry> simply contained in d must not have any subnodes.

Step 3. d must not contain any <refer-option>s.

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-expression(ex) 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 <builtin-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 <cat>, then return <fail>.

Step 2.2.2. Perform evaluate-expression(e) to obtain an <aggregate-value>,av. av must immediately contain an <aggregate-type> which immediately contains <scalar>. Let v be the <basic-value> in av.

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 forms 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|
                         constraint-expression ⊥ multiple-constraint

multiple-constraint::=  constraint |
                        multiple-constraint & constraint

constraint::=     ¬ constraint | (area | 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,ce, 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 <target-reference>, 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>.

4.5.11 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,cce | multiple-constraint,mc.

Perform test-constraints(d,cce) 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 <entry>, return <true>; otherwise return <false>.

Case 7. (Otherwise).

Let cs be the immediate component of ce. Perform test-constraints(d,cs) 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 cdt be the <data-type> of c, and let cbv be the <basic-value> of c. Perform convert(tdt,cdt,cbv) to obtain a <basic-value>,bv. Replace c by <constant>: bv 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, and compare 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

5.0 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.

5.1 The Interpretation-state

```
M6.   <interpretation-state>::= <program-state> <allocated-storage> [<dataset-list>]

M7.   <program-state>::= <program-directory>
                         [<block-state-list>]
                         [<file-information-list>]
```

5.1.1 DIRECTORIES

```
M8.   <program-directory>::= <static-directory>
                             <controlled-directory>
                             <file-directory>

M9.   <static-directory>::= [<static-directory-entry-list>]

M10.  <static-directory-entry>::= ( <external> | <declaration-designator> )
                                  <identifier>  <generation>

M11.  <controlled-directory>::= [<controlled-directory-entry-list>]

M12.  <controlled-directory-entry>::= ( <external> | <declaration-designator> )
                                      <identifier> [<generation-list>]

M13.  <file-directory>::= [<file-directory-entry-list>]

M14.  <file-directory-entry>::= ( <external> | <declaration-designator> )
                                <identifier> [<subscript-value-list>]
                                <file-information-designator>

M15.  <subscript-value>::= <integer-value>
```

5.1.2 BLOCK STATE

```
M16.  <block-state>::= <block-directory>                 <block-control>
                       <linkage-part>                    [<block-environment>]
                       [<established-on-unit-list>]      [<condition-bif-value-list>]
                       [<copy-file>]

M17.  <block-directory>::= <automatic-directory>
                           <defined-directory>
                           [<parameter-directory>]

M18.  <automatic-directory>::= [<automatic-directory-entry-list>]

M19.  <automatic-directory-entry>::= <identifier> <generation>

M20.  <defined-directory>::= [<defined-directory-entry-list>]
```

```
M21.    <defined-directory-entry>::= <identifier> <evaluated-data-description>

M22.    <parameter-directory>::= [<parameter-directory-entry-list>]

M23.    <parameter-directory-entry>::= <identifier>
                                       {<undefined> | <established-argument>}

M24.    <established-argument>::= <generation> ( <dummy> | <not-dummy> )

M25.    <evaluated-entry-reference>::= <entry-value> [<established-argument-list>]

M26.    <block-control>::= <executable-unit-designator> <group-control>
                           <statement-control>          [<string-io-control>]
                           [<data-item-control-list>]   [<format-control-list>]
                           [<current-scalar-item-list>] [<remote-block-state>]
                           [<current-file-value>]

M27.    <remote-block-state>::= <block-state-designator>

M28.    <current-file-value>::= <file-value>

M29.    <group-control>::= [<controlled-group-state-list>]

M30.    <controlled-group-state>::= <spec-designator> <cv-target> <cv-type>
                                    [<by-value> <converted-by-type>]
                                    [<to-value> <converted-to-type>]

M31.    <cv-target>::= <evaluated-target>

M32.    <cv-type>::= <data-type>

M33.    <by-value>::= <real-value> | <complex-value>

M34.    <converted-by-type>::= <data-type>

M35.    <to-value>::= <real-value>

M36.    <converted-to-type>::= <data-type>

M37.    <statement-control>::= <operation-list>

M38.    <string-io-control>::= <character-string-value> [<string-limit>] [<first-comma>]

M39.    <string-limit>::= <integer-value>

M40.    <first-comma>::= <on> | <off>

M41.    <data-item-control>::= <data-list-indicator> <data-item-indicator>

M42.    <data-list-indicator>::= <designator>

M43.    <data-item-indicator>::= <designator> | <undefined>

M44.    <format-control>::= <format-specification-list-designator> <format-list-index>
                            [<format-iteration-value>]      [<format-iteration-index>]
                            [<format-statement-designator>] [<remote-block-state>]

M45.    <format-list-index>::= <integer-value>

M46.    <format-iteration-value>::= <integer-value>

M47.    <format-iteration-index>::= <integer-value>

M48.    <current-scalar-item>::= <basic-value> <data-type> [<data-name-field>] |
                                 <evaluated-target>

M49.    <data-name-field>::= {symbol-list}

M50.    <linkage-part>::= [<entry-point-designator>]
                          [<returned-value> | <returned-onsource-value>]
                          [<prologue-flag>]
```

```
M51.    <returned-value>::= <aggregate-value>

M52.    <returned-onsource-value>::= <character-string-value>

M53.    <block-environment>::= <block-state-designator>

M54.    <established-on-unit>::= <evaluated-condition> [<entry-value> | <system-action>]
                                 [<snap>]

M55.    <evaluated-condition>::= <computational-condition>     | <area-condition>   |
                                 <evaluated-io-condition>     | <error-condition>  |
                                 <programmer-named-condition> | <finish-condition> |
                                 <storage-condition>

M56.    <evaluated-io-condition>::= <io-condition> <file-value>

M57.    <condition-bif-value>::= <onchar-value>  | <oncode-value>  | <onfield-value> |
                                 <onfile-value>  | <onkey-value>   | <onloc-value>   |
                                 <onsource-value>

M58.    <onchar-value>::= <integer-value>

M59.    <oncode-value>::= <integer-value>

M60.    <onfield-value>::= <character-string-value>

M61.    <onfile-value>::= <character-string-value>

M62.    <onkey-value>::= <character-string-value>

M63.    <onloc-value>::= <character-string-value>

M64.    <onsource-value>::= <character-string-value>

M65.    <copy-file>::= <file-value>
```

5.1.3 FILE INFORMATION

```
M66.    <file-information>::= <open-state>       <filename>
                              <file-description> [<file-opening>]

M67.    <open-state>::= <open> | <closed>

M68.    <filename>::= <character-string-value>

M69.    <file-opening>::= <dataset-designator> <complete-file-description>
                          <current-position>   [<delete-flag>]
                          [<allocated-buffer>] [<page-number>] [<first-comma>]

M70.    <complete-file-description>::= <evaluated-file-description-list>

M71.    <evaluated-file-description>::= <stream> | <record>     | <input>  | <output> |
                                        <update> | <sequential> | <direct> | <print>  |
                                        <keyed>  | <environment>|
                                        <evaluated-tab-option>  | <evaluated-title>   |
                                        <evaluated-linesize>    | <evaluated-pagesize>

M72.    <evaluated-tab-option>::= <integer-value-list>

M73.    <evaluated-title>::= <character-string-value>

M74.    <evaluated-linesize>::= <integer-value>

M75.    <evaluated-pagesize>::= <integer-value>

M76.    <current-position>::= <designator> | <undefined>

M77.    <allocated-buffer>::= <generation> [<key>]

M78.    <page-number>::= <integer-value>
```

5.1.4 STORAGE AND VALUES

M79. <allocated-storage>::= [<allocation-unit-list>]

M80. <allocation-unit>::= <basic-value-list>

M81. <basic-value>::= <real-value> | <complex-value> |
<character-string-value> | <bit-string-value> |
<entry-value> | <label-value> |
<format-value> | <file-value> |
<pointer-value> | <offset-value> |
<area-value> | <undefined>

M82. <real-value>::= <real-number>

M83. <complex-value>::= <complex-number>

M84. <complex-number>::= [<real-number> | <undefined>] [<real-number> | <undefined>]

M85. <real-number>::=

The members of the set of real numbers are the alternative choices as immediate and terminal components of <real-number>.

M86. <integer-value>::=

The members of the set of integers are the alternative choices as immediate and terminal components of <integer-value>.

M87. <character-string-value>::= <character-value-list> | <null-character-string>

M88. <character-value>::= {symbol} | <undefined>

M89. <bit-string-value>::= <bit-value-list> | <null-bit-string>

M90. <bit-value>::= <zero-bit> | <one-bit> | <undefined>

M91. <entry-value>::= <entry-point-designator> [<block-state-designator>]

M92. <label-value>::= <executable-unit-designator> [<block-state-designator>]

M93. <format-value>::= <format-statement-designator> [<block-state-designator>]

M94. <file-value>::= <file-information-designator>

M95. <pointer-value>::= <generation> | <null>

M96. <offset-value>::= <evaluated-data-description>
<significant-allocation-list>
<storage-index-list> | <null>

M97. <area-value>::= <area-allocation-list> <significant-allocation-list> | <empty>

M98. <area-allocation>::= <significant-allocation-list> <allocation-unit>

M99. <significant-allocation>::= <evaluated-data-description> <occupancy>

M100. <occupancy>::= <allocated> | <freed>

M101. <aggregate-value>::= <aggregate-type> <basic-value-list>

M102. <aggregate-type>::= <dimensioned-aggregate-type> |
<structure-aggregate-type> | <scalar>

M103. <dimensioned-aggregate-type>::= <element-aggregate-type> <bound-pair-list>

M104. <element-aggregate-type>::= <structure-aggregate-type> | <scalar>

M105. <structure-aggregate-type>::= <member-aggregate-type-list>

M106. <member-aggregate-type>::= <aggregate-type>

5.1.5 GENERATIONS, EVALUATED DATA DESCRIPTIONS, AND EVALUATED TARGETS

M107. ⟨generation⟩::= ⟨evaluated-data-description⟩
⟨allocation-unit-designator⟩
⟨storage-index-list⟩

M108. ⟨evaluated-data-description⟩::= ⟨data-description⟩

Note: ⟨extent-expression⟩ components of ⟨evaluated-data-description⟩: ⟨data-description⟩ contain only ⟨integer-value⟩s. (See Section 7.1.)

M109. ⟨storage-index⟩::= ⟨basic-value-index⟩ [⟨position-index⟩]

M110. ⟨basic-value-index⟩::= ⟨integer-value⟩

M111. ⟨position-index⟩::= ⟨integer-value⟩

M112. ⟨evaluated-target⟩::= ⟨generation⟩ | ⟨evaluated-pseudo-variable-reference⟩

M113. ⟨evaluated-pseudo-variable-reference⟩::= ⟨pseudo-variable⟩
[⟨generation⟩ | ⟨aggregate-value⟩]
[⟨aggregate-value⟩]
[⟨aggregate-value⟩]

5.1.6 DATASET

M114. ⟨dataset⟩::= ⟨dataset-name⟩ [⟨record-dataset⟩ | ⟨stream-dataset⟩]

M115. ⟨dataset-name⟩::= ⟨character-string-value⟩

M116. ⟨record-dataset⟩::= ⟨sequential-dataset⟩ |
⟨keyed-dataset⟩ |
⟨keyed-sequential-dataset⟩

M117. ⟨sequential-dataset⟩::= ⟨alpha⟩ [⟨record-list⟩] ⟨omega⟩

M118. ⟨keyed-dataset⟩::= [⟨keyed-record-list⟩]

M119. ⟨keyed-sequential-dataset⟩::= ⟨alpha⟩ [⟨keyed-record-list⟩] ⟨omega⟩

M120. ⟨keyed-record⟩::= ⟨record⟩ ⟨key⟩

M121. ⟨record⟩::= ⟨evaluated-data-description⟩ ⟨basic-value-list⟩

M122. ⟨key⟩::= ⟨character-string-value⟩

M123. ⟨stream-dataset⟩::= ⟨alpha⟩ [⟨stream-item-list⟩] ⟨omega⟩

M124. ⟨stream-item⟩::= {symbol} | ⟨linemark⟩ | ⟨pagemark⟩ | ⟨carriage-return⟩

5.2 Terminology and Definitions

The following terms are employed at various places throughout the operations which comprise the interpretation phase.

5.2.1 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›.

5.2.2 BLOCK

The term block is used to refer to a ‹begin-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› designated by the ‹executable-unit-designator› simple component of the ‹block-state› in question.

5.3 The Interpret Operation and the Initialization of the Interpretation State

5.3.1 INTERPRET

First, the ‹interpretation-state› is initialized, and then the ‹program› is executed.

Operation: interpret(dl,ev)

where dl is a ‹dataset-list›,
ev is an ‹entry-value›.

Step 1. Perform initialize-interpretation-state(dl).

Step 2. Let eer be ‹evaluated-entry-reference›: ev. Perform activate-procedure(eer).

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(dl)

where dl 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›
    dl.
```

Step 2. Perform build-file-directory-and-informations.

Step 3. Perform build-controlled-directory.

Step 4. Perform allocate-static-storage-and-build-static-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-fdi(d).

Case 1.2.2. d does have a ‹bound-pair-list›,bpl.

Step 1.2.2.1. bpl must not contain ‹asterisk› 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›,elb[i] and perform evaluate-expression-to-integer(ub[i]) to obtain an ‹integer-value›,eub[i].

Step 1.2.2.4. For each distinct ‹subscript-value-list›,subl having n ‹subscript-value›s such that the i'th contained ‹integer-value› lies in the inclusive range defined by elb[i] and eub[i], selected in any order, perform build-fdi(d,subl).

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 {symbol}s 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: allocate-static-storage-and-build-static-directory

Step 1. For each <declaration>,d in the <program>, selected in any order:

Case 1.1. d has <static>.

Case 1.1.1. d has <external>, and there exists in the «static-directory» a «static directory-entry» with both <external> and an <identifier> that is equa to the <identifier> in d.

No action.

Case 1.1.2. (Otherwise).

Step 1.1.2.1. Let dd be the <data-description> in d. Perform evaluate-data description-for-allocation(dd) to obtain an «evaluated-data description»,edd.

Step 1.1.2.2. Perform allocate(edd) to obtain a «generation»,g.

Step 1.1.2.3. If d has <initial>, then perform initialize-generation(d,g).

Step 1.1.2.4. If d has <external>, then let sc be <external>; otherwise let s be a <declaration-designator> designating d. Let id be th <identifier> in d.

Step 1.1.2.5. Append, to the «static-directory-entry-list», a «static directory-entry»: sc id g.

Case 1.2. (Otherwise).

No action.

5.3.7 PROGRAM-EPILOGUE

Operation: program-epilogue

Step 1. For each «file-information»,fi containing «open» perform close(fv), where fv i a «file-value» designating fi.

Chapter 6: Flow of Control

6.0 Introduction

The definition of the control mechanism of the PL/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 5.3, 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 5.3.7).

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 6.3.

6.1.1.1 Execute-stop-statement

Operation: execute-stop-statement(ss)

where ss is a <stop-statement>.

Step 1. Perform raise-condition(<finish-condition>).

Step 2. Perform stop-program.

6.1.1.2 Stop-program

Operation: stop-program

Step 1. For each ⟨block-state⟩ contained in the ⟨program-state⟩ except the current ⟨block-state⟩, replace the ⟨statement-control⟩ by

⟨statement-control⟩:
⟨operation-list⟩:
⟨operation⟩ 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>s 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,cbifs)

where eer is an ⟨evaluated-entry-reference⟩,
cbifs is a [⟨condition-bif-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 <recursive>, unless p is the immediate component of an <on-unit>.

Step 3. Let eud 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 ble be ⟨block-environment⟩: bsd. Otherwise ble is ⟨absent⟩. If bsd 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⟩:
               eud;
          ⟨group-control⟩
          ⟨statement-control⟩:
               ⟨operation-list⟩:
                    ⟨operation⟩ for instal-arguments(eer);;;
     ⟨linkage-part⟩:
          epd.
```

If ble is a ⟨block-environment⟩ then attach ble to bs. If cbifs is a ⟨condition-bif-value-list⟩ then attach cbifs to bs.

Step 5. Append bs to the ⟨block-state-list⟩.

6.2.1.1 Instal-arguments

Operation: instal-arguments(eer)

where eer is an <evaluated-entry-reference>.

Step 1. If the <parameter-name-list>,pnl of the <entry-point> designated by the <entry-point-designator> of eer exists, then perform Step 1.1.

Step 1.1. Let eal be the <established-argument-list> of eer. Attach to the current <parameter-directory> a <parameter-directory-entry-list>,pdel, with the same number of immediate components as pnl and whose i'th immediate component is

```
<parameter-directory-entry>:
  <identifier> of the i'th immediate component of pnl
  <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 pnl, append to pdel a

```
<parameter-directory-entry>:
    id
    <undefined>.
```

Step 3. Perform prologue.

6.2.2 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 eud 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>: eud;
      <group-control>
      <statement-control>:
           <operation-list>:
                <operation> for prologue;;;
    <linkage-part>
    <block-environment>:
      bsd.
```

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>s specify initialization. Any <expression>s evaluated during the prologue, such as in <extent-expression>s or <expression>s 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(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.

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 edd.

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: epiloque

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 <onsource-value>, then attach its immediate subtree to the <returned-onsource-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 <executable-unit-designator> to designate the next <executable-unit>. This is normally the last action of each execution of an <executable-unit>. The special cases of execution of a <goto-statement> and <end-statement> may set the <executable-unit-designator> independently, while the <return-statement> and <stop-statement> have no further need for it. The <executable-unit-designator> is initialized in each activation of a <procedure> or <begin-block>, and may be reset on entering a <group> or <if-statement>.

6.3.1 NORMAL-SEQUENCE

Operation: normal-sequence

Step 1. Let eu be the <executable-unit> designated by the current <executable-unit-designator>. Let eu1 be the <executable-unit-list> or <entry-or-executable-unit-list> which contains eu, but does not contain any other <executable-unit-list> or <entry-or-executable-unit-list> which also contains eu.

Step 2. Let eu2 be that immediate component of eu1 which either contains eu or is exactly eu.

Case 2.1. eu1 is an <executable-unit-list>.

Let eu3 be that <executable-unit> which immediately follows eu2 as an immediate component of eu1.

Case 2.2. eu1 is an <entry-or-executable-unit-list>.

Step 2.2.1. Let eu4 be that <entry-or-executable-unit> which immediately follows eu2 as an immediate component of eu1. If eu4 immediately contains <entry-point>, then let eu2 be eu4 and go to Step 2.2.1.

Step 2.2.2. Let eu3 be the <executable-unit> immediate component of eu4.

Step 3. Set the current <executable-unit-designator> to designate eu3.

6.3.1.1 Advance-execution

This operation is the "driver" which initiates execution of each <executable-unit> as selected by the current <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 <executable-unit-designator> designates the <executable-unit> to be executed. Execution of an <executable-unit> consists of performing the appropriate "execute" operation. That operation normally terminates with the current <executable-unit-designator> designating some other <executable-unit> in the <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 <executable-unit> designated by the current <executable-unit-designator>.

Step 2. Perform execute-xxx(f), where "xxx" is replaced by the sequence of symbols forming the name of the type of f.

6.3.3 EXECUTE-BEGIN-BLOCK

Operation: execute-begin-block(b)

where b is a <begin-block>.

Step 1. Perform activate-begin-block.

Step 2. Perform normal-sequence.

6.3.4 EXECUTE-GROUP

Constraints: In a <do-spec>,dsp, let tr be the <target-reference> component. For each <spec> of dsp, let

e	be the <expression> immediately contained in the <spec>,
b	be the <expression> in the <by-option>,
te	be the <expression> in the <to-option>,
r	be the <expression> in the <repeat-option>,

if such options are present. The following constraints must hold for each <spec>:

(1) If te is present then tr and e must have <computational-type> and the derived modes of tr, e, b, and te, must all be <real>.

(2) If r is present then tr, e, and r must all have:

<computational-type>, or
<locator>, or
<non-computational-type>, with immediate subnodes of the <non-computational-type>s belonging to the same category other than <locator>.

(3) If tr has <pointer> and either or both of e and r has <offset>, then each such <offset> must contain a <variable-reference>. If tr has <offset> and either or both of e and r has <pointer>, then the <offset> in tr must contain a <variable-reference>.

Operation: execute-group(g)

where g is a <group>.

Step 1. Let feu be the first <executable-unit> simply contained in g.

Case 1.1. g has a <non-iterative-group>.

Set the current <executable-unit-designator> to designate feu.

Case 1.2. g has a <while-only-group>.

Let exp be the <expression> of the <while-option> of g. Perform establish-truth-value(exp) to obtain t. If t is <true>, then set the current <executable-unit-designator> to designate feu; otherwise, perform normal-sequence.

Case 1.3. g has a <controlled-group>.

Let dsp be the <do-spec> of g. Perform establish-controlled-group(dsp) to obtain t. If t is <true>, then set the current <executable-unit-designator> to designate feu; otherwise, perform normal-sequence.

6.3.4.1 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: establish-controlled-group(dsp)

where dsp is a <do-spec>.

result: <true> or <false>.

Step 1. Let tr be the <target-reference> of dsp, and dt be the <data-type> of tr. Perform evaluate-target-reference(tr) to obtain an <evaluated-target>,et.

Step 2. Let sp be the first <spec> of dsp. Append to the current <controlled-group-state-list>, the tree

```
<controlled-group-state>:
    <spec-designator>: a designator designating sp;
    <cv-target>: et;
    <cv-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>.

6.3.4.2 Initialize-spec-options

Operation: initialize-spec-options

Step 1. Let sp be the <spec> designated by the current <spec-designator>. Let e 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(e) 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 etdt be the <data-type> of et. Let dt be a

```
<data-type>:
  <computational-type>:
      <arithmetic>:
          <mode>:
              <real>;
          the derived common <base> of etdt and the current <cv-type>
          the derived common <scale> of etdt and the current <cv-type>
          the converted <precision> of etdt.
```

Step 2.2.2. Perform evaluate-expression(et) to obtain an ‹aggregate-value›,x. Let y be the ‹basic-value› in x. Perform convert(dt,etdt,y) to obtain a ‹basic-value›: ‹real-value›,z. 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> when the value of the ‹cv-target› is converted for comparison with the ‹to-value›.)

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.

Case 2.3.1.1. sp contains a <by-option>: <expression>,eb.

Perform evaluate-expression(eb) to obtain an ‹aggregate-value›,x. Let ebv be the ‹basic-value› in x. Let ebdt be the <data-type> of eb.

Case 2.3.1.2. sp contains a <to-option> but not a <by-option>.

Let ebv be a ‹basic-value›: ‹real-value›: 1. Let ebdt 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 <mode> of ebdt and current ‹cv-type›
               derived common <base> of ebdt and current ‹cv-type›
               derived common <scale> of ebdt and current ‹cv-type›
               converted <precision> of ebdt.
```

Perform convert(dt,ebdt,ebv) 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 m=min(N,max(p-q,r-s)+max(q,s)+1), where N is the maximum <number-of-digits> allowed for <fixed> with the <base> of dt. Let n=max(q,s). Change the value of the <number-of-digits> in dt to m, and change the value of the <scale-factor> in dt to n.

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-target›. Let dd be the <data-description> immediate component of e. Perform assign(cvt,av,dd).

6.3.4.3 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 «to-value»,y, perform Steps 1.1 through 1.3.

Step 1.1. Let cvt be the immediate component of the current «cv-target». Perform value-of-evaluated-target(cvt) to obtain an «aggregate-value» containing a «basic-value»,x. Let xt be the current «cv-type».

Step 1.2. Let dt be the current «converted-to-type». Perform convert(dt,xt,x) to obtain a «basic-value»,cx.

Step 1.3. Let bv be the current «by-value». If bv ≥ 0 and cx > y, or if bv < 0 and cx < y, return «false».

Step 2. If the <spec> designated by the current <spec-designator> contains a <while-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».

6.3.4.4 Establish-next-spec

This operation is used to advance through the list of <spec>s 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 «to-value» and a «converted-to-type» then delete them.

Step 4. Perform initialize-spec-options. Return «true».

6.3.4.5 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-item-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 evt be the ‹evaluated-target› of the current ‹cv-target›.

Case 1.1. sp contains a <repeat-option>: <expression>,re.

Let dd be the <data-description> immediate component of re. Perform evaluate-expression(re) to obtain an ‹aggregate-value›,av. Perform assign(evt,av,dd).

Case 1.2. sp contains a <by-option> or a <to-option>.

Step 1.2.1. Let bt be the <data-type> in the current ‹converted-by-type›. Let cvt be the <data-type> in the current ‹cv-type›. Let cvt2 be a <data-type> that is the same as bt except for its <precision>, which is the converted <precision> of cvt, with bt being the target <data-type> for determining the converted <precision>.

Step 1.2.2. Let et be the ‹evaluated-target› in the current ‹cv-target›. Perform value-of-evaluated-target(et) to obtain an ‹aggregate-value›,x1. Let x2 be the ‹basic-value› in x1. Perform convert(cvt2,cvt,x2) to obtain a ‹basic-value›,x3. Let x be the ‹real-value› or ‹complex-value› in x3.

Step 1.2.3. Let y be the ‹real-value› or ‹complex-value› in the current ‹by-value›. Perform arithmetic-result(x+y,bt) to obtain z, where z is a ‹real-value› or a ‹complex-value›.

Step 1.2.4. Let rtd be a

```
<data-description>:
    <item-data-description>:
        <data-type>:
            bt.
```

Let w be an

```
‹aggregate-value›:
    ‹aggregate-type›:
        ‹scalar›;
    ‹basic-value-list›:
        ‹basic-value›:
            z.
```

Perform assign(et,w,rtd).

Case 1.3. sp does not contain a <repeat-option>, a <to-option>, or a <by-option>.

Go to Step 3.

Step 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(ifs)

where ifs is an <if-statement>.

Step 1. Let e be the <expression> immediate component of the <test> of ifs. Perform establish-truth-value(e) 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 ifs.

Case 2.2. tv is <false> and ifs simply contains an <else-unit>,eu.

Replace the immediate component of the current <executable-unit-designator> by a designator of the <executable-unit> of eu.

Case 2.3. tv is <false> and ifs 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>,av: <basic-value-list>: <basic-value>,sv. Let sdt be the <data-type> of exp.

Step 2. Let tdt be a

```
<data-type>:
    <computational-type>:
      <string>:
           <string-type>:
                <bit>;
           <maximum-length>:
                <asterisk>.
```

Perform convert(tdt,sdt,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(cs)

where cs is a <call-statement>.

Step 1. Let sr be the <subroutine-reference> component of cs. Perform evaluate-entry-reference(sr) to obtain an <evaluated-entry-reference>,eer.

Step 2. Perform activate-procedure(eer).

Step 3. Perform normal-sequence.

6.3.6.1 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>.

6.3.6.1.1 Evaluate-entry-reference

Operation: evaluate-entry-reference(er)

where er is a <subroutine-reference> or a <procedure-function-reference>.

result: an <evaluated-entry-reference>.

Step 1. Let vr be the <value-reference> immediate component of er. Perform evaluate-value-reference(vr) to obtain an <aggregate-value>,ag. Let ev be the <entry-value> in ag.

Step 2. Let ep be the <entry-point> designated by the <entry-point-designator> in ev. Let id be the <identifier> in the <statement-name> in ep. Let p be the <procedure> that block-contains ep. Let b be the abstract-block that block-contains p. Let dl be the <declaration-list> immediately contained in b. Let d be the <declaration> in dl whose <identifier> equals id. Let de be a copy of the <entry> immediately contained in the <named-constant> of d, and let vre be a copy of the <entry> of the <data-description> of vr.

Delete from de and vre all <variable-reference>s which are immediate components of <offset>. In an implementation-defined fashion, compare corresponding <options> components of de and vre and, if either de or vre has <options> components and the other does not have corresponding <options> components, compare de and vre. Delete all <options> components of de and vre.

For each <extent-expression>,ee in de or vre containing a <constant>,c, let e be an <expression>: c; and perform evaluate-expression-to-integer(e) to obtain an <integer-value>,iv and replace ee by <extent-expression>: iv. de and vre 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. er does not contain an <argument-list>.

Return an <evaluated-entry-reference>: ev.

Case 3.2. er contains an <argument-list>,al.

Let n be the number of <argument>s in al. Let eal be an <established-argument-list> with n <established-argument> immediate components. For i=1,...,n, taken in any order, perform Step 3.2.1.

Step 3.2.1. Let arg be the i'th <argument> in al. Let dd be the <data-description> in the i'th <parameter-descriptor> in the <parameter-descriptor-list> in the <data-type> of vr. Perform establish-argument(arg,dd) to obtain an <established-argument>,x. Replace the i'th <established-argument> of eal by x.

Step 4. Return an <evaluated-entry-reference>: ev eal.

6.3.6.1.2 Establish-argument

Operation: establish-argument(arg,dd)

where arg is an <argument>,
dd is a <data-description>.

result: an <established-argument>.

Case 1. arg does not immediately contain <dummy>.

Let vr be the <variable-reference> simple component of arg. Perform evaluate-variable-reference(vr) to obtain a <generation>,g. Return an <established-argument>: g <not-dummy>.

Case 2. arg immediately contains <dummy>.

Step 2.1. Let e be the <expression> in arg. Perform evaluate-expression(e) to obtain an <aggregate-value>,av.

Step 2.2. Let avdd be the <data-description> immediate component of e. Perform promote-and-convert(dd,av,avdd) to obtain an <aggregate-value>,av2.

Step 2.3. Let cdd be a copy of dd. Replace each <bound-pair> in cdd that is not a component of <entry> by the corresponding <bound-pair> in av2. For each <area-size>,x, contained in cdd, let y be an <integer-value> determined by an implementation-defined algorithm, and replace x by an

```
<area-size>:
  <extent-expression>:
       y.
```

For each <data-type>,st simply contained in cdd that simply contains a tree of the form <string>: <maximum-length>: <asterisk>;; perform Step 2.3.1.

Step 2.3.1. Let m be the maximum of the lengths of all <character-string-value>s or <bit-string-value>s in av2 that correspond to st. Replace the <maximum-length> in st by a

```
<maximum-length>:
     <extent-expression>:
          <integer-value>:
               m.
```

Step 2.4. Perform evaluate-data-description-for-allocation(cdd) to obtain an <evaluated-data-description>,edd. Perform allocate(edd) to obtain a <generation>,g.

Step 2.5. Let bvl be the <basic-value-list> in av2. Perform set-storage(g,bvl).

Step 2.6. Return an <established-argument>: g <dummy>.

6.3.7 EXECUTE-GOTO-STATEMENT

Operation: execute-goto-statement(gs)

where gs is a <goto-statement>.

Step 1. Let vr be the <value-reference> of gs. Perform evaluate-value-reference(vr) to obtain an <aggregate-value>,ag. ag must contain the form

<label-value>,lv:
<executable-unit-designator>,tp
<block-state-designator>,bsn.

Step 2. The <block-state-list> must contain a <block-state>,bs, designated by bsn. (Its corresponding block contains the <executable-unit> designated by tp.)

Case 2.1. bs is the current <block-state>.

Perform local-goto(lv).

Case 2.2. bs is not the current <block-state>.

Step 2.2.1. vr must not immediately contain a <data-description> whose <data-type> has <local>.

Step 2.2.2. The <statement-control> component of bs must not contain an <operation> for execute-allocate-statement, execute-locate-statement, or prologue.

Step 2.2.3. Replace the <statement-control> component of bs by:

<statement-control>:
<operation-list>:
<operation> for advance-execution
<operation> for trim-io-control
<operation> for local-goto(lv).

Step 2.2.4. For each <block-state>,b that occurs after bs and before the current <block-state> in the <block-state-list> (of the <interpretation-state>) perform Steps 2.2.4.1 and 2.2.4.2.

Step 2.2.4.1. The <statement-control> component of b must not contain an <operation> for execute-allocate-statement, execute-locate-statement, or prologue.

Step 2.2.4.2. Replace the <statement-control> of b by <statement-control>: <operation-list>: <operation> for epilogue.

Step 2.2.5. Replace the current <statement-control> by <statement-control>: <operation-list>: <operation> for epilogue.

6.3.7.1 Local-goto

Operation: local-goto(lv)

where lv is a <label-value>.

Step 1. Let tp be the <executable-unit-designator> in lv. Let eu be the <executable-unit> designated by tp. If there is an <iterative-group>,g that contains eu but does not contain an <iterative-group> or <begin-block> that contains eu, then the current <executable-unit-designator> must designate an <executable-unit> that is contained in g.

Step 2. Perform trim-group-control(tp).

Step 3. Replace the current <executable-unit-designator> by tp.

6.3.7.2 Trim-group-control

Operation: trim-group-control(eud)

where eud is an <executable-unit-designator>.

Step 1. Let eu be the <executable-unit> designated by eud. Let b be the <begin-block> or <procedure> that block-contains eu.

Step 2.

Case 2.1. b contains a <controlled-group> that contains eu.

Let n be the number of <controlled-group>s that contain eu and are contained in b. If the current <controlled-group-state-list> contains more than n <controlled-group-state>s, delete those after the n'th <controlled-group-state>.

Case 2.2. (Otherwise).

If there is a current <controlled-group-state-list>, delete it.

6.3.8 EXECUTE-NULL-STATEMENT

Operation: execute-null-statement(ns)

where ns is a <null-statement>.

Step 1. Perform normal-sequence.

6.3.9 EXECUTE-RETURN-STATEMENT

Operation: execute-return-statement(rs)

where rs is a <return-statement>.

Step 1. Let p be the last <block-state> of the <block-state-list> whose corresponding block is a <procedure>.

Case 1.1. rs does not contain an <expression>.

There must not be a <returns-descriptor> in the <entry-point> designated by the <entry-point-designator> of p. If p is the first <block-state> in the <block-state-list> then perform raise-condition(<finish-condition>).

Case 1.2. rs contains an <expression>,e.

There must be a <returns-descriptor>,rd in the <entry-point> designated by the <entry-point-designator> of p. Let dd be the <data-description> immediate component of e. dd must be proper for assignment to rd (see Section 4.4.2.3).

Step 1.2.1. Perform evaluate-expression(e) to obtain ev.

Step 1.2.2. Perform promote-and-convert(rd,ev,dd) to obtain an <aggregate-value>,av. Attach <returned-value>: av; to the <linkage-part> of the <block-state> immediately preceding p in the <block-state-list>.

Step 2. For each <block-state> (if any) which is or which follows p in the <block-state-list> except the current <block-state>, replace its <statement-control> by a

```
<statement-control>:
    <operation-list>:
      <operation> for epilogue.
```

Step 3. Perform epilogue.

6.3.10 EXECUTE-END-STATEMENT

Operation: execute-end-statement(es)

where es is an <end-statement>.

Step 1. Let eul be that <executable-unit-list> or <entry-or-executable-unit-list> which contains es but does not contain any other <executable-unit-list> or <entry-or-executable-unit-list> which also contains es.

Let n be the node which immediately contains eul.

Case 1.1. n is a <procedure>.

Step 1.1.1. The <entry-point> designated by the <entry-point-designator> component of the current <linkage-part> must not contain a <returns-descriptor>.

Step 1.1.2. If the <block-state-list> contains only one <block-state> then perform raise-condition(<finish-condition>).

Step 1.1.3. Perform epilogue.

Case 1.2. n is a <begin-block>.

Perform epilogue.

Case 1.3. n is a <non-iterative-group>.

Step 1.3.1. Set the current <executable-unit-designator> to designate the <executable-unit> which simply contains n.

Step 1.3.2. Perform normal-sequence.

Case 1.4. n is a <while-only-group>.

Set the current <executable-unit-designator> to designate the <executable-unit> which simply contains n.

Case 1.5. n is a <controlled-group>.

Set the current <executable-unit-designator> to designate the <executable-unit> that simply contains n. Perform test-termination-of-controlled-group to obtain t.

Case 1.5.1. t is <true>.

Perform normal-sequence.

Case 1.5.2. t is <false>.

Set the current <executable-unit-designator> to designate the first <executable-unit> of eul.

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,cbifs)

where c is an <evaluated-io-condition>, a <programmer-named-condition>, or a terminal node of <condition-name> apart from the <named-io-condition>s or <programmer-named-condition>s,
cbifs is a [<condition-bif-value-list>].

Step 1. There must exist at least one <block-state>.

Step 2. If c is one of the terminal nodes of a <computational-condition>, then perform test-enablement(c) to obtain r, which must not be <disabled>.

Step 3. Let cc be a copy of cbifs. If c is a <conversion-condition> and the current <block-control> contains a <current-file-value>: <file-value>,fv;, and cc does not contain an <onfile-value>, then attach to cc an <onfile-value>: fn; where fn is the <character-string-value> in the <filename> in fi, where fi is the <file-information> designated by fv.

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 eu 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 eu.

Case 1.2. The current <linkage-part> does not contain a <prologue-flag> and the current <block-control> contains a <remote-block-state>,rbs.

Let fc be the last <format-control> of the current <format-control-list> that contains a <remote-block-state>. (This <remote-block-state> equals rbs.) Let tp be the <format-statement> designated by the <format-statement-designator> of fc.

Case 1.3. The current <linkage-part> contains a <proloque-flag>.

Let tp be the <procedure> or <begin-block> of which eu 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 cp.

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>.

6.4.1.3 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-io-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 test-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 cbifs be a

```
<condition-bif-value-list>:
     <condition-bif-value>:
          <onsource-value>:
               <character-string-value>:
                    <null-character-string>;;;
     <condition-bif-value>:
          <onchar-value>:
               <integer-value>: 0.
```

Case 1.2.2. c contains <name-condition>.

Let cbifs be a

```
<condition-bif-value-list>:
     <condition-bif-value>:
          <onfield-value>:
               <character-string-value>:
                    <null-character-string>.
```

Case 1.2.3. c contains <key-condition>.

Let cbifs be a

<condition-bif-value-list>:
<condition-bif-value>:
<onkey-value>:
<character-string-value>:
<null-character-string>.

Case 1.2.4. (Otherwise).

Let cbifs be <absent>.

Step 2. If c is a <named-io-condition> then perform evaluate-named-io-condition(c) to obtain ec; otherwise let ec be c.

Step 3. Perform interrupt(ec,cbifs).

Step 4. Perform normal-sequence.

6.4.1.4 Evaluate-named-io-condition

Operation: evaluate-named-io-condition(nioc)

where nioc is a <named-io-condition>.

result: an <evaluated-io-condition>.

Step 1. Let vr be the <value-reference> immediate component of nioc. Perform evaluate-file-option(vr) to obtain a <file-value>,f.

Step 2. Let ioc be the <io-condition> component of nioc. Return <evaluated-io-condition>: ioc f.

6.4.2 INTERRUPTS

The <on-statement> and <revert-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.

6.4.2.1 Execute-on-statement

Operation: execute-on-statement(os)

where os is an <on-statement>.

Step 1. For each <condition-name>,cn 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. cn has <named-io-condition>,nic.

Perform evaluate-named-io-condition(nic) to obtain an <evaluated-io-condition>,eic. Let ec be an <evaluated-condition>: eic.

Case 1.1.2. cn does not have <named-io-condition>.

Let cn1 be the immediate subtree of cn. Let ec be an <evaluated-condition>: cn1.

Step 1.2. If the current <established-on-unit-list> contains an <established-on-unit>,eou containing ec, then delete eou.

Step 1.3.

Case 1.3.1. os contains an <on-unit>,ou.

Let epd be an <entry-point-designator> designating the <entry-point> of ou. Let es be an

<entry-value>:
epd
<block-state-designator> designating the current <block-state>.

Case 1.3.2. (Otherwise).

Let es be <system-action>.

Step 1.4. Let neou be

<established-on-unit>:
ec
es.

Step 1.5. If os contains <snap>, attach <snap> to neou. Append neou 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 cnl be the <condition-name-list> immediate component of rs. For each <condition-name>,c in cnl 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>,nic.

Perform evaluate-named-io-condition(nic) to obtain an <evaluated-io-condition>,eioc. Let ec be an <evaluated-condition>: eioc.

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>, eou containing ec, then delete eou.

Step 2. Perform normal-sequence.

6.4.3 INTERRUPT

Operation: interrupt(c,cbifs)

where c is an <evaluated-io-condition>, a <programmer-named-condition>, or a terminal node of <condition-name> apart from the <named-io-condition>s or <programmer-named-condition>s,
cbifs is a [<condition-bif-value-list>].

Step 1. Let bs be the current <block-state>.

Step 2. Append to cbifs each of the following components whose immediate subnode it does not already possess (or let cbifs be a <condition-bif-value-list> with these components if cbifs is <absent>):

<condition-bif-value>:
<oncode-value>:
<integer-value>: an implementation-defined integer;;;

<condition-bif-value>:
<onloc-value>:
the <character-string-value> formed 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-io-condition> and cbifs does not contain an <onfile-value>:

<condition-bif-value>:
<onfile-value>:
the <character-string-value> in the <filename> of the <file-information> designated by the <file-value> of c.

Step 3. Let eoul be the <established-on-unit-list> of bs.

Case 3.1. eoul contains an <established-on-unit>,eou, 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>,ad containing <external>, and

(2) eoul contains an <established-on-unit>,eou which contains a <programmer-named-condition> designating a <declaration> equal to ad.

Step 3.1.1. If eou contains <snap>, 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. eou contains an <entry-value>,ev.

Let eer be an <evaluated-entry-reference>: ev. Perform activate-procedure(eer,cbifs).

Case 3.1.2.2. eou contains <system-action>.

Perform system-action(c,cbifs).

Case 3.2. (Otherwise).

Case 3.2.1. bs is not the first <block-state> of the <block-state-list>,bsl.

Let bs be the immediately preceding <block-state> of bsl and go to Step 3.

Case 3.2.2. (Otherwise).

Perform system-action(c,cbifs).

Step 4. c must not be <error-condition>, <fixedoverflow-condition>, <overflow-condition>, <size-condition>, <stringrange-condition>, <subscriptrange-condition>, or <zerodivide-condition>.

6.4.4 SYSTEM-ACTION

For every <evaluated-condition>, there is the possibility, as defined in Section 6.4.3, that an interrupt operation for it will lead to the operation system-action.

Operation: system-action(c,cbifs)

where c is an <evaluated-io-condition>, a <programmer-named-condition>, or a terminal node of a <condition-name> apart from the <named-io-condition>s or <programmer-named-condition>s,
cbifs is a <condition-bif-value-list>.

Case 1. c is <finish-condition> or <stringsize-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 8.7.2.12).

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>,cbifs).

6.4.4.1 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 suballocate 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>,g, have the immediate <data-description> component dd. Because dd is contained in an <evaluated-data-description>, each <bound-pair> of dd has two <extent-expression>s which contain only <integer-value>s; that is, no <bound-pair> contains an <asterisk> or <expression> with a <variable-reference>. The number of elements of the <storage-index-list> of g is the number of scalar-elements corresponding to dd. 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-description-list>,mdd.

For each tree of the form <member-description>: <data-description>,dd1[i]; in mdd, i=1,...,m, perform scalar-elements-of-data-description(dd1[i]) to obtain an integer, n[i]. Return the integer

$$\sum_{i=1}^{m} n[i].$$

Case 3. dd is of the form <data-description>:
<dimensioned-data-description>:
<element-data-description>,edd
<bound-pair-list>,bpl.

Let ic be the immediate component of edd. Let dd1 be a <data-description>: ic. Perform scalar-elements-of-data-description(dd1) to obtain the integer n. For each tree of the form <bound-pair>,bp[i] in bpl, i=1,...,m, let ub[i] and lb[i] be the <integer-value> components of the <upper-bound> and <lower-bound> respectively of bp[i]. Return the integer

$$n * \prod_{i=1}^{m} (ub[i]-lb[i]+1).$$

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 sdd be this <structure-data-description>. Let mdl be the <member-description-list> of sdd. Let n[i] be the number of scalar-elements corresponding to the i'th element of mdl, obtained by performing scalar-elements-of-data-description(dd[i]) where dd[i] is the <data-description> of the i'th element of mdl. Let sm[0] be 0. Let sm[i] be the sum of the n[j] for the first i elements of mdl. Let j be such that $sm[j-1] < ord \leq sm[j]$. Perform find-item-data-description(dd[j],k), where k = ord-sm[j-1], to obtain an <item-data-description>,idd. Return idd.

Case 3. The immediate component of dd is a <dimensioned-data-description>.

Let ddd be this <dimensioned-data-description>.

Case 3.1. The <element-data-description> of ddd has an <item-data-description> as the immediate component.

Return a copy of this <item-data-description>.

Case 3.2. (Otherwise).

Let dd1 be a <data-description> immediately containing a copy of the <structure-data-description> of ddd. Perform scalar-elements-of-data-description(dd1) to obtain an integer n. Perform find-item-data-description(dd1,k+1), where k is the value of the remainder obtained when ord-1 is divided by n, to obtain an <item-data-description>,idd. Return idd.

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-value>.

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 i be the <basic-value-index> of p.

Step 2. Let v[i] be the immediate component of the i'th <basic-value> of v.

Case 2.1. d does not have <string>: <nonvarying>; or <pictured>.

Let e be a copy of v[i].

Case 2.2. d has <string>: <nonvarying>; or <pictured>.

Let N be the <maximum-length> component of d (if d has <string>) or the associated character-string length of d (if d has <pictured>).

Case 2.2.1. N = 0.

Let e be the <null-character-string> (if d has <character> or <pictured>) 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) $i \le j \le k$ (i=j=k is possible),

(2) v[i], v[i+1],..., v[k] are either all <bit-string-value>s or all <character-string-value>s.

(3) $\sum_{s=i}^{j-1} L[s] < M \le \sum_{s=i}^{j} L[s]$, and

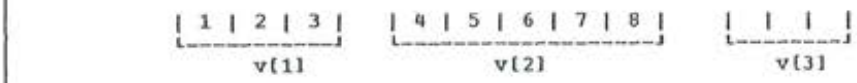
(4) $M+N-1 \le \sum_{s=i}^{k} L[s]$.

Here L[x] is the length of v[x]. Let e be a <character-string-value> (if v[i] is a <character-string-value>) or a <bit-string-value> (if v[i] is a <bit-string-value>). The length of e is N. The N components of e are the same N successive components of v[j], v[j+1], ..., v[k] beginning at component number (M-L[i]-L[i+1]-...-L[j-1]) in v[j].

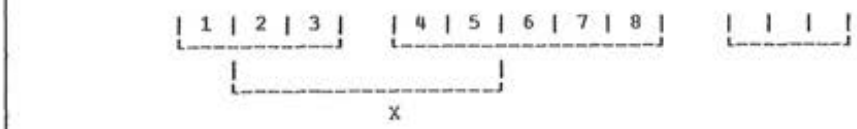
Step 3. Return a <basic-value> containing a copy of e.

```
DECLARE 1 A AUTOMATIC UNALIGNED,
          2 B CHARACTER(3),
          2 C CHARACTER(5),
          2 D CHARACTER(3),
        X CHARACTER(4) DEFINED A POSITION(2),
        Y CHARACTER(6) DEFINED A POSITION(6);

A.B = '123';
A.C = '45678';
```

After the assignments, the <generation> corresponding to A looks like



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.



Hence a <value-reference>,X yields '2345'. Evaluation of a <variable-reference> Y yields a <scalar> <generation> accessing the <generation> of A as follows.

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 <generation>s and <basic-value>s 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>,al, 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 al.

Step 1.2.

Case 1.2.1. The <storage-class> of d contains <controlled>.

Perform allocate-controlled-storage(al) to obtain the ‹generation›,g.

Case 1.2.2. The <storage-class> of d contains <based>.

Perform allocate-based-storage(al) 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(al)

where al is an <allocation>.

result: a ‹generation›.

Step 1. Let d be the <declaration> designated by the <declaration-designator>,dp, of al. Make a copy, dd, of the <data-description> of d.

Step 2. Perform evaluate-data-description-for-allocation(dd) to obtain the ‹evaluated-data-description›,edd.

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 <based> variable and the assignment of the resulting ‹generation› to a <locator> variable.

Operation: allocate-based-storage(al)

where al is an <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 <based> 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›,gs.

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 gi; otherwise the value ‹fail› will be obtained.

Step 5. Make a copy, dd, of the <data-description> immediately contained in the <variable> of d. Perform evaluate-data-description-for-allocation(dd) to obtain an ‹evaluated-data-description›,edd.

Step 6.

Case 6.1. A ‹generation›,gi, was obtained in Step 4.

Make an allocation for dd in the area ‹generation›,gi, by performing suballocate(edd,gi). The result obtained will be either the ‹generation›,g, or the value ‹fail›. In the latter case, perform raise-condition(<area-condition>); on normal return go to Step 4.

Case 6.2. ‹fail› was obtained in Step 4.

Make an allocation for dd in the ‹allocated-storage› by performing allocate(edd) 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›:
                    g.
```

Let ddp be of the form <data-description>: <item-data-description>: <pointer>. Let egs be ‹evaluated-target›: gs. 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.

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: evaluate-in-option(al,vr)

where al is an <allocation>,
vr is a <variable-reference>.

result: a <generation> or <fail>.

Step 1. Let ds be the <declaration> designated by the <declaration-designator> component of vr. ds is the <declaration> for the locator that will be used to identify the <allocation-unit> that will result from the <allocation>,al.

Step 2. If the component <locator> of ds has <offset> and this has the subnode <variable-reference>, then let this <variable-reference> be vro.

Step 3.

Case 3.1. al has an <in-option> component.

Let the <variable-reference> of the <in-option> be vri.

Case 3.2. al has no <in-option> component.

Case 3.2.1. ds has an <offset> component.

vro must have been created in Step 2. That is, the area base for the offset must have been specified. Let vri be the same as vro.

Case 3.2.2. ds has a <pointer> component.

Return the value <fail>, since the <allocation-unit> for al is to be constructed in the <allocated-storage>.

Step 4. Perform evaluate-variable-reference(vri) to obtain the <generation>,gia.

Step 5. If vro has been created in Step 2, then perform evaluate-variable-reference(vro) to obtain the <generation>,gib. gia must be the same as gib.

Step 6. Return 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: allocate(edd)

where edd is an <evaluated-data-description>.

result: a <generation>.

Step 1. Perform either Step 1.1 or Step 1.2.

Step 1.1. Perform raise-condition(<storage-condition>). Go to Step 1.

Step 1.2. Perform make-allocation-unit(edd) to obtain an <allocation-unit>,au.

Step 2. Append au to <allocated-storage>.

Step 3. Perform scalar-elements-of-data-description(dd), to obtain n, where dd is the <data-description> of edd. Let i be 1. Construct a <storage-index-list>,sil, by performing Steps 3.1 through 3.3 n times.

Step 3.1. Perform find-item-data-description(dd,i) to obtain an <item-data-description>,idd.

Step 3.2.

Case 3.2.1. idd contains either a <pictured> component or a <string> component that has <nonvarying>.

Construct a <storage-index>,si, that contains a <basic-value-index> that has an <integer-value> equal to i and also a <position-index> that has an <integer-value> equal to 1.

Case 3.2.2. (Otherwise).

Construct a <storage-index>,si, that contains a <basic-value-index> that has an <integer-value> equal to i.

Step 3.3. Append si to sil and set i to i+1.

Step 4. Return the <generation> constructed from an <allocation-unit-designator> designating au, edd, and sil.

7.2.6 SUBALLOCATE

This operation constructs an <area-allocation> and adds it to the <area-allocation-list> of an area <generation>. The result of this operation is the <generation> that identifies the new <area-allocation>.

Operation: suballocate(edd,g)

where edd is an <evaluated-data-description>,
g is a <generation>.

result: a <generation> or <fail>.

Step 1. The value of g must not be <undefined>.

Step 2. Perform either Step 2.1 or Step 2.2.

Step 2.1. Return the value <fail>.

Step 2.2. Perform make-allocation-unit(edd) to obtain an <allocation-unit>,au.

Step 3. Perform value-of-generation(g) to obtain an <aggregate-value>,agv whose <aggregate-type> must immediately contain <scalar> and whose <basic-value-list> contains an <area-value>,av. If av immediately contains <empty> then replace av by

```
<area-value>,av:
    <area-allocation-list>
    <significant-allocation-list>.
```

Append a

```
<significant-allocation>:
    edd
    <occupancy>:
        <allocated>;;
```

to the <significant-allocation-list>,sal immediate component of av. Append an <area-allocation>: sal au; to the <area-allocation-list> of av.

Step 4. Perform scalar-elements-of-data-description(dd), to obtain n, where dd is the <data-description> of edd. Let i be 1. Construct a <storage-index-list>,sil, by performing Steps 4.1 through 4.3 n times.

Step 4.1. Perform find-item-data-description(dd,i) to obtain an <item-data-description>,idd.

Step 4.2.

Case 4.2.1. idd contains either a <pictured> component or a <string> component that has <nonvarying>.

Construct a <storage-index>,si, that contains a <basic-value-index> that has an <integer-value> equal to i 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 i.

Step 4.3. Append si to sil and set i to i+1.

Step 5. Return the <generation> constructed from an <allocation-unit-designator> that designates au, edd, and sil.

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 cdd be a copy of the <data-description>,dd. For each <extent-expression> in cdd that immediately contains an <expression>, perform Step 1.1, with the <extent-expression>s chosen in any order.

Step 1.1. Let the chosen <extent-expression> be ee. Evaluate ee by performing evaluate-expression-to-integer(e) where e 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 cdd, 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 <maximum-length> component of cdd, the value of the <integer-value> must be greater than or equal to zero.

Step 4. In each <area-size> component of cdd, the value of the <integer-value> must be greater than or equal to zero.

Step 5. For each <refer-option>,ro, in cdd perform Step 5.1.

Step 5.1. Let mo be the <member-description> in cdd that ro references as in Step 2.2 of validate-based-declaration. Let n be an <integer-value> such that the result of find-item-data-description(cdd,n) is equal to the <item-data-description> in mo. Replace the <identifier-list> component of ro by n.

Step 6. Return the <evaluated-data-description>: cdd.

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 st be the ‹storage-type› of d.

Step 2.

Case 2.1. st contains ‹automatic›, ‹parameter›, or ‹defined›.

Perform find-block-state-of-declaration(dp) to obtain a ‹block-state›,bs. If st contains ‹automatic› or ‹defined› then bs must not contain a ‹prologue-flag› in its ‹linkage-part›. Find the directory entry, e, in the ‹automatic-directory›, ‹parameter-directory›, or ‹defined-directory› of bs, as appropriate, such that the ‹identifier› of e is equal to the ‹identifier› of d.

Case 2.2. st 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(edd)

where edd is an ‹evaluated-data-description›.

result: an ‹allocation-unit›.

Step 1. Let dd be the simply contained ‹data-description› component of edd. Perform scalar-elements-of-data-description(dd) to obtain n, the number of scalar-elements that correspond to edd. Construct a ‹basic-value-list›,svl, by performing Steps 1.1 through 1.2 for i=1,...,n.

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 edd.

Step 1.2.

Case 1.2.1. The ‹mode› of idd contains ‹complex› and idd does not contain ‹pictured›.

Append to svl a ‹complex-value› consisting of two ‹undefined› components.

Case 1.2.2. idd contains <pictured>.

Let m be the associated character-string length of the <pictured> component of idd. Append to svl a <character-value-list> consisting of m elements, each of which is <undefined>.

Case 1.2.3. idd contains <string> containing <nonvarying>.

Let m be the <integer-value> contained by the <maximum-length> component of <string>.

Case 1.2.3.1. The <string> of idd contains <character>.

If m is zero, append a <character-string-value> consisting of the <null-character-string> to svl; otherwise append a <character-string-value> consisting of a <character-value-list> with m elements containing <undefined>.

Case 1.2.3.2. The <string> of idd contains <bit>.

If m is zero, append a <bit-string-value> consisting of the <null-bit-string> to svl; otherwise append a <bit-string-value> consisting of a <bit-value-list> with m elements containing <undefined>.

Case 1.2.4. idd contains an <area> component.

Append an <area-value>: <empty>; to svl.

Case 1.2.5. (Otherwise).

Append an <undefined> element to svl.

Step 2. Return an <allocation-unit> containing svl.

7.2.10 INITIALIZE-REFER-OPTIONS

This operation initializes the object of each <refer-option> in a <generation>.

Operation: initialize-refer-options(g)

where g is a <generation> whose <aggregate-type> immediately contains <structure-aggregate-type>.

Step 1. Let edd be the <evaluated-data-description> of g and let dd be the <data-description> in edd. For each <refer-option>,ro, of edd, chosen in any order, perform Steps 1.1 and 1.2.

Step 1.1. The <refer-option>,ro, has an <integer-value> index, i, constructed by evaluate-data-description-for-allocation; this identifies the element of g that is the object of the <refer-option>,ro. Perform find-item-data-description(dd,i) to obtain an <item-data-description>,idd, of the element. Let eddr be an <evaluated-data-description> with idd as immediate component. Construct the <generation>,gr, from eddr, a copy of the <allocation-unit-designator> of g and the <storage-index-list> containing a single element, a copy of the i'th element of the <storage-index-list> of g.

Step 1.2. ro is a component of an <extent-expression> that has an <integer-value>,iv. Let ddi be a <data-description> with integer-type (see Section 9.1.2). Let av be of the form

<aggregate-value>:
<aggregate-type>:
<scalar>;
<basic-value-list>:
<basic-value>:
<integer-value>,iv.

Let et be <evaluated-target>: gr. Perform assign(et,av,ddi) to assign the value of the <extent-expression> to the object of the <refer-option>.

7.2.11 FIND-BLOCK-STATE-OF-DECLARATION

Operation: find-block-state-of-declaration(dp)

where dp is a <declaration-designator> designating the <declaration>,d.

result: a <block-state>.

Step 1. Let bs be the current <block-state>. If the <block-control> of bs contains the form <remote-block-state>,rbs, then let bs be the <block-state> designated by rbs.

Step 2. Let bb be the corresponding block of bs (see Section 5.2.2). If bb contains d, then return bs.

Step 3. bs must have a <block-environment>,bv. Let bs be the <block-state> designated by bv. 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: initialize-generation(g,d)

where g 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>,idd.

Perform initialize-scalar-element(g,idd).

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(g,d).

Case 3. The immediate component of the <data-description>,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 sdd be the simply contained <structure-data-description> in dd. For each <member-description> immediately contained in the <member-description-list> of sdd, 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-description>,md be the i'th immediate component of the <member-description-list>.

Step 3.2. Perform select-qualified-reference(g,il,d), where il is an <identifier-list> consisting of the single <identifier>,id, that is a copy of the i'th immediate component of the <identifier-list> of sdd, to obtain a modified <generation>,gl.

Step 3.3. Let dc be a copy of the <declaration>,d, and let sddc be the copy of sdd contained in dc.

Case 3.3.1. The immediate component of dd is <dimensioned-data-description> and the <data-description> of md has a <dimensioned-data-description>,ddd.

Append to the <bound-pair-list> of dc the immediate components of the <bound-pair-list> of ddd. Replace sddc by the immediate subtree of the <element-data-description> of ddd; this will always be a <structure-data-description> or an <item-data-description>.

Case 3.3.2. (Otherwise).

Replace sddc by the immediate subtree of the <data-description> of md.

Step 3.4. Perform initialize-generation(gl,dc).

7.3.2 INITIALIZE-SCALAR-ELEMENT

This operation initializes a generation consisting of a single element.

Operation: initialize-scalar-element(g,idd)

where g is a <generation>,
idd is an <item-data-description>.

Step 1. If idd does not contain an <initial-element> then terminate this operation.

Step 2. If the <initial-element> component of idd has an <asterisk> then terminate this operation.

Step 3. Perform evaluate-expression(e), where e is the <expression> simply contained in the <initial-element> of idd, to obtain an <aggregate-value>,v.

Step 4. Create an <evaluated-target>,et, and attach g to it.

Step 5. Perform assign(et,v,ddl), where ddl is the <data-description> immediately contained in e.

7.3.3 INITIALIZE-ARRAY

This operation initializes a <generation> that has an array of <basic-value>s.

<evaluated-initial-element>::= <asterisk> | <parenthesized-expression> (scalar) |
(<iteration-factor> | <evaluated-iteration-factor>)
<evaluated-initial-element-list> |
<evaluated-initial-item> |
<absent>

<evaluated-iteration-factor>::= <integer-value>

<evaluated-initial-item>::= <basic-value> <data-description>

Operation: initialize-array(g,d)

where g is a <generation>,
d is a <declaration>.

Step 1. Let edd be a copy of the <evaluated-data-description> of g in which the simply contained <dimensioned-data-description> has been replaced by the subtree of its <element-data-description>. If d does not have an <initial-element-list> then terminate this operation. Let iel be the <initial-element-list> of d.

Step 2. Let m and n be 1. Let mt be the number of elements in the <storage-index-list> of g. Let nt be the number of elements in the <initial-element-list>,iel.

Step 3. Construct an <evaluated-initial-element-list>,eiel, by making a copy of iel and replacing each <initial-element-list> node by an <evaluated-initial-element-list> and each <initial-element> node by an <evaluated-initial-element> in the copy.

Step 4. Perform Steps 4.1 through 4.7 while m ≤ mt and n ≤ nt.

Step 4.1. Let the n'th element of eiel be eie[n].

Step 4.2. If eie[n] immediately contains an <iteration-factor>,itf, then perform evaluate-expression-to-integer(e), where e is the <expression> of itf, to obtain the <integer-value>,v, and replace itf by an <evaluated-iteration-factor> containing v.

Step 4.3. If eie[n] immediately contains an <evaluated-initial-element-list>,eiel1, then for each <expression>,e, contained in eiel1, 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>,itf1.

Optionally perform evaluate-expression-to-integer(e) to obtain an <integer-value>,i1 and replace itf1 by an <evaluated-iteration-factor> containing i1.

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 eie[n] immediately contains an <evaluated-iteration-factor>,eif, then perform Steps 4.4.1 through 4.4.3.

Step 4.4.1. Let eiel2 be the <evaluated-initial-element-list> of eie[n]. Let i2 be the <integer-value> of eif.

Step 4.4.2.

Case 4.4.2.1. i2 ≤ 0.

Replace eie[n] by <absent>.

Case 4.4.2.2. (Otherwise).

Let k be the number of elements in eiel2. Replace eie[n] by the i2*k elements formed from i2 replications of the sequence of elements of eiel2. Let nt be nt+i2*k-1.

Step 4.4.3. Go to Step 4.1.

Step 4.5. If eie[n] is neither an <asterisk> nor an <absent> then perform Steps 4.5.1 and 4.5.2.

Step 4.5.1. If eie[n] is a <parenthesized-expression>,pe1, then perform evaluate-expression(e), where e is the <expression> of pe1, to obtain an <aggregate-value> having the <basic-value>,bv1, and replace pe1 by an <evaluated-initial-item> comprising of bv1 and a copy of the <data-description> immediate component of pe1.

Step 4.5.2. eie[n] is an <evaluated-initial-item>. Let v be of the form

```
<aggregate-value>:
     <aggregate-type>:
          <scalar>;
     <basic-value-list>:
          bv;;
```

where bv is the <basic-value> of eie[n]. Let dd be the <data-description> immediate component of eie[n]. Let si be a copy of the m'th element of the <storage-index-list> of g. Perform assign(<evaluated-target>:g1;,v,dd), where g1 is the <generation> comprising of edd constructed in Step 1, a copy of the <allocation-unit-designator> of g, and a <storage-index-list> consisting of the single element si.

Step 4.6. If eie[n] is not <absent> then let m be m+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: execute-free-statement(fs)

where fs is a <free-statement>.

Step 1. For each <freeing>,fr, in the <freeing-list> of fs, chosen in left-to-right order, perform Steps 1.1 and 1.2.

Step 1.1. Let d be the <declaration> designated by the <declaration-designator> component of fr.

Step 1.2.

Case 1.2.1. The <storage-class> of d contains <controlled>.

Perform free-controlled-storage(fr).

Case 1.2.2. The <storage-class> of d contains <based>.

Perform 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: free-controlled-storage(fr)

where fr is a <freeing>.

Step 1. Let d be the <declaration> designated by the <declaration-designator>,dp, of fr.

Step 2. Perform find-directory-entry(dp) to obtain a <controlled-directory-entry>,e, corresponding to d.

Step 3. If e contains a <generation-list>,gl, perform Steps 3.1 to 3.3.

Step 3.1. Let g be the last <generation> in gl.

Step 3.2. Perform free(g).

Step 3.3. Delete g from gl.

7.4.3 FREE-BASED-STORAGE

This operation frees a <based> 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-qualifier>

and an <in-option>.

Of these, only dp always exists.

Step 2. Let dd be the <data-description> of the <declaration> designated by dp. If fr contains a <locator-qualifier>,lq then let vr be <variable-reference>: lq dp dd. Otherwise let vr be <variable-reference>: dp dd. Let n be the number of <bound-pair>s in dd. 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>,gf to be freed. Let au be the <allocation-unit> designated by the <allocation-unit-designator> of gf.

Step 3.1. If the <data-type> components of dd 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 au must equal the corresponding <maximum-length> in the <evaluated-data-description> of gf.

Step 3.2. If there are n elements in the <basic-value-list> of au then there must be n elements in the <storage-index-list> of gf 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(fr). If an area containing gf can be inferred from fr, a <generation>,ga, will be obtained; otherwise <fail> will be obtained.

Step 5.

Case 5.1. ga exists.

Step 5.1.1. Let av be the <area-value> referred to by ga. The <area-allocation-list> of av must contain au. Let aa be the <area-allocation> containing au and let sal be the <significant-allocation-list> of av. Let n be the number of elements of the <significant-allocation-list> of aa. Replace the <allocated> component of the n'th element of sal by <freed>.

Step 5.1.2. If aa is the only <area-allocation> of av then replace av by <area-value>: <empty>. Otherwise, delete aa from the <area-allocation-list> of av and perform Step 5.1.2.1.

Step 5.1.2.1. If the last element, el, of sal contains <freed> then delete el and go to Step 5.1.2.1.

Case 5.2. deduce-in-option returned <fail> in Step 4.

The <allocation-unit> au must be an immediate component of the <allocation-unit-list> of <allocated-storage>. The <generation> gf 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

(2.1) the <automatic-directory>
(2.2) the <parameter-directory> as the component of an <established-argument> containing <dummy>

(3) for any <file-information>, be equal to a component of the <allocated-buffer>.

Perform free(gf).

7.4.4 DEDUCE-IN-OPTION

This operation infers, if possible, an area <generation> in which a freeing is to be applied.

Operation: deduce-in-option(fr) .

where fr is a <freeing>.

result: 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>,ga. Return ga.

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-qualifier>.

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 <based> 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>,ga. Return ga.

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 <pageno-pv>), or components in the ‹condition-bif-value›s of the current ‹block-state› (in the case of the <pseudo-variable>s <onchar-pv> and <onsource-pv>). The components to be changed are determined by evaluating a <target-reference> (Section 7.5.2), to obtain an ‹evaluated-target› (Section 7.5.2.1). The actual assignment is effected by the operation assign (Section 7.5.3), which, in general, will involve conversion of ‹basic-value›s and promotion of ‹aggregate-value›s. The full generality of assignment is available through the <assignment-statement> (Section 7.5.1), but other constructions cause invocation of the operation assign.

7.5.1 THE ASSIGNMENT STATEMENT

Operation: execute-assignment-statement(ast)

where ast 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 ast. Perform evaluate-target-reference(tr) to obtain the ‹evaluated-target›,et.

Step 1.2. Let e be the <expression> of ast. Perform evaluate-expression(e) to obtain an ‹aggregate-value›,v.

Step 2. Perform assign(et,v,d), where d is the <data-description> immediately contained in e.

Step 3.

Case 3.1. ast contains no unevaluated <target-reference>.

Perform normal-sequence.

Case 3.2. ast contains one or more unevaluated <target-reference>s.

Let tr be the leftmost unevaluated <target-reference>. Perform evaluate-target-reference(tr) to obtain the ‹evaluated-target›,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 same as the result <data-description> 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-target›.

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-target›: g.

Case 2. The immediate component of tr is a <pseudo-variable-reference>,pvr.

Perform evaluate-f(x[1],...,x[n]), where f is the immediate component of the <pseudo-variable> contained by pvr, and x[1],...,x[n] are the <argument>s, if any, in the <argument-list>. Let g be the result of evaluate-f(x[1],...,x[n]). (It will be either a <generation> or an <evaluated-pseudo-variable-reference>.) Return an <evaluated-target>: g.

7.5.2.1 Evaluated Targets

Operation: value-of-evaluated-target(t)

where t is an <evaluated-target>.

result: an <aggregate-value>.

Case 1. t immediately contains a <generation>,g.

Perform value-of-generation(g) to obtain an <aggregate-value>,av, which must not contain <undefined>. Return av.

Case 2. t contains <imag-pv> or <real-pv>.

Let g be the <generation> in t, and perform value-of-generation(g) to obtain an <aggregate-value>,x, which must not contain <undefined>. Let sr be the scalar-result of performing Steps 1 and 2 of imag-bif (see Section 9.4.4.40) or Steps 1 to 3 of real-bif (see Section 9.4.4.66), respectively, taking:

the scalar-value of x to be the <basic-value> in x;
the scalar-result-type to be the <data-type> in the <generation> of t, with <complex> replaced by <real>.

Return an <aggregate-value> containing sr.

Case 3. t contains <onchar-pv> or <onsource-pv>.

Perform onchar-bif (see Section 9.4.4.55) or onsource-bif (see Section 9.4.4.61), respectively, to obtain an <aggregate-value>,av. Return av.

Case 4. t contains <pageno-pv>.

The <file-value>,fv in t must obey the constraints of pageno-bif, Step 1 (see Section 9.4.4.62). Perform Steps 2 and 3 of pageno-bif.

Case 5. t contains <substr-pv>.

Perform value-of-generation(g) to obtain an <aggregate-value>,sa, where g is the <generation> in t. sa must not contain <undefined>. Let st be the first <aggregate-value> in t; let le be the second <aggregate-value> in t, if present.

Let sr be the scalar-result of performing Steps 1 to 4 of substr-bif (see Section 9.4.4.76), taking:

the scalar-value of sa (or of st or of le) to be the single <basic-value> in sa (or in st or in le);
the scalar-result-type to be the <data-type> in the <generation> in t.

Return an <aggregate-value> containing sr.

Case 6. t contains <unspec-pv>.

Let g be the <generation> in t. Perform Step 3 of unspec-bif (see Section 9.4.4.85).

7.5.3 THE ASSIGNMENT OPERATION

Operation: assign(et,sv,sd)

where et is an ‹evaluated-target›,
sv is an ‹aggregate-value›,
sd is a ‹data-description›.

Case 1. et immediately contains a ‹generation›,g.

Step 1.1. Let edd be the ‹evaluated-data-description› of g. Perform promote-and-convert(edd,sv,sd), to obtain an ‹aggregate-value›,av.

Step 1.2. Perform set-storage(g,svl), where svl is the ‹basic-value-list› of av.

Case 2. et immediately contains an ‹evaluated-pseudo-variable-reference›,epvr.

Let f be the component of the ‹pseudo-variable› in epvr. Thus f is the name of the ‹pseudo-variable›. Perform assign-f(epvr,sv,sd).

7.5.3.1 Promote-and-convert

Let x and y be ‹aggregate-type›s. Then x is promotable to y if x and y are compatible and y is the same as the common ‹aggregate-type› of x and y. However where one has a ‹bound-pair› which has ‹asterisk›, the other may have a ‹bound-pair› whose ‹expression› components contain ‹integer-value›s.

Operation: promote-and-convert(td,sv,sd)

where td is a ‹data-description› or ‹evaluated-data-description›,
sv is an ‹aggregate-value›,
sd is a ‹data-description›.

result: an ‹aggregate-value›.

Step 1. Let ats be the ‹aggregate-type› of sv.

Case 1.1. td is an ‹evaluated-data-description›.

Let tdd be td.

Case 1.2. td is not an ‹evaluated-data-description›.

Perform evaluate-data-description-for-allocation(td) to obtain an ‹evaluated-data-description›,tdd. For each tree of the form ‹bound-pair›,bp: ‹asterisk›; contained in tdd that is not a component of an ‹entry›, perform Step 1.2.1.

Step 1.2.1. If a ‹bound-pair›,cbp corresponding to bp exists in ats, then replace bp by cbp. Otherwise, replace bp by a ‹bound-pair› whose ‹lower-bound› and ‹upper-bound› each have an ‹integer-value› of 1.

Step 2. Each ‹bound-pair-list› in ats must equal the corresponding ‹bound-pair-list› in tdd.

Step 3. Perform scalar-elements-of-data-description(tdd) to obtain an integer, n. Let av be an ‹aggregate-value› whose ‹aggregate-type› equals the ‹aggregate-type› associated with the ‹data-description› of tdd, and whose ‹basic-value-list›,svl, contains n ‹basic-value› nodes, each of which has no 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 find-item-data-description(tdd,j) to obtain an ‹item-data-description› that describes the j'th scalar-element of av. Let ydt be the ‹data-type› of this ‹item-data-description›.

Step 4.2. Let x be the scalar-element of sv that corresponds to the j'th scalar-element of av. Let xdt be the <data-type> of the <item-data-description> in sd which corresponds to x in sv.

Step 4.3.

Case 4.3.1. Both xdt and ydt contain <computational-type>, or one contains <offset> and the other contains <pointer>.

Perform convert(ydt,xdt,x) to obtain y[j].

Case 4.3.2. Both xdt and ydt contain <area>.

Optionally perform raise-condition(<area-condition>). If there is a normal return from this operation let y[j] be a <basic-value> containing <undefined>. If raise-condition(<area-condition>) is not performed, then let y[j] be a copy of x.

Case 4.3.3. (Otherwise).

Let y[j] be a copy of x.

Step 5. Append the y[j] 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,ssvl)

where g is a <generation>,
ssvl is a <basic-value-list>.

Step 1. Let au be the <allocation-unit> designated by the <allocation-unit-designator> of g. au must be contained in <allocated-storage>. Let tsvl be the <basic-value-list> of au.

Step 2. Let dd be the <data-description> of the <evaluated-data-description> of g and let sil be the <storage-index-list> of g.

Step 3. For each of the elements of sil, taken in any order, perform Steps 3.1 through 3.3.

Step 3.1. Let si be the chosen element and let i be its ordinal in sil. Let j be a copy of the value of the <integer-value> of the <basic-value-index> of si.

Step 3.2. Let ssv be the i'th element of the <basic-value-list>,ssvl.

Step 3.3.

Case 3.3.1. si contains a <position-index>.

Let k be a copy of the value of the <integer-value> of the <position-index> of si. ssv will contain either a <character-string-value> or a <bit-string-value>. Perform find-item-data-description(dd,i) to obtain the <item-data-description>,idd, of the target scalar-element.

If idd contains <pictured> then let n be its associated string-length; otherwise let n be the value of the <maximum-length> in idd. Let m be 1. Perform Steps 3.3.1.1 through 3.3.1.4 n times.

Step 3.3.1.1. Let tl be the <character-value-list> or <bit-value-list> of the j'th element of tsvl.

Step 3.3.1.2. Replace the k'th element of tl by a copy of the m'th element of ssv.

Step 3.3.1.3. Let k be k+1. If k is now greater than the number of elements in tl, then let k be 1 and let j be j+1. Repeat Step 3.3.1.3.1 while the j'th element of tsvl contains <null-character-string> or <null-bit-string>.

Step 3.3.1.3.1. Set j to j+1.

Step 3.3.1.4. Let m be m+1.

Case 3.3.2. si does not contain a <position-index>.

Replace the j'th element of tsvl by a copy of ssv.

7.5.4 PSEUDO-VARIABLES

This Section presents the definitions of the <pseudo-variable>s in alphabetical order. For each <pseudo-variable>:

The Arguments Section indicates the number of arguments to the <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 <data-description> of the <pseudo-variable-reference> (and of the <target-reference> containing it) in terms of the <data-description>s of the arguments.

One or two operations are defined for each <pseudo-variable>. This first operation, evaluate-f, where f is the name of the <pseudo-variable>, is used by the operation evaluate-target-reference. This operation returns a <generation> or an <evaluated-pseudo-variable-reference>.

The second operation, assign-f, where f is the name of the <pseudo-variable>, is defined only for those <pseudo-variable>s whose evaluate-f operation yields an <evaluated-pseudo-variable-reference>.

7.5.4.1 Imag-pv

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> in <pictured-numeric>) 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-imag-pv(x)

Step 1. Let y be the <variable-reference> in x. Perform evaluate-variable-reference(y) to obtain a <generation>,g. Return an <evaluated-pseudo-variable-reference>: <pseudo-variable>: <imag-pv>; g.

Operation: assign-imag-pv(t,sv,sd)

where t is an <evaluated-pseudo-variable-reference>,
sv is an <aggregate-value>,
sd is a <data-description>.

Step 1. Let t have the form

<evaluated-pseudo-variable-reference>:
<generation>,g:
<evaluated-data-description>,dc
<allocation-unit-designator>,aud.

Let dr be the same as dc except that all <mode>s have <real>. Perform promote-and-convert(dr,sv,sd) to obtain vr.

Step 2. For each <storage-index>,p in g, perform Steps 2.1 and 2.2.

Step 2.1. Let sdc, svr and sdr be the scalar-elements of dc, vr and dr, respectively, corresponding to p.

Step 2.2.

Case 2.2.1. sdc has <arithmetic> but not <pictured-numeric>.

Let sg be a

<generation>:
<evaluated-data-description>:
sdc;
aud
<storage-index-list>:
p.

Let z1 be the value of sg (z1 contains a single <complex-value>). Let z2 be a <complex-value> with first component as z1 and second component equal to the component of svr. Perform set-storage(sg,bvl) where bvl is a <basic-value-list> containing z2.

Case 2.2.2. sdc has <pictured-numeric>

Let sg be a

<generation>:
<evaluated-data-description>:
sdr;
aud
<storage-index-list>:
p.

Let n be the associated character-string length of the <pictured-numeric> in sdr. If p contains a <position-index>, increment its value by n; otherwise, append to p a <position-index> of n+1. Perform set-storage(sg,bvl), where bvl is a <basic-value-list> containing svr.

7.5.4.2 Onchar-pv

Arguments: (none)

Attributes: The result has <aggregate-type>: <scalar>. The result <data-type> has <character>.

Operation: evaluate-onchar-pv

Step 1. Return an <evaluated-pseudo-variable-reference>: <pseudo-variable>: <onchar-pv>.

Operation: assign-onchar-pv(t,sv,sd)

where t is an <evaluated-pseudo-variable-reference>,
sv is an <aggregate-value>,
sd is a <data-description>.

Step 1. In either order, perform get-established-onvalue(<onchar-value>) to obtain i and perform get-established-onvalue(<onsource-value>) to obtain str. i and str must not be <fail>.

Step 2. Convert sv to <character> of length 1, using the <data-type> in sd as the source <data-type> for the conversion, to obtain c.

Step 3. If i>0, replace the i'th <character-value> in str by the <character-value> in c.

7.5.4.3 Onsource-pv

Arguments: (none)

Attributes: The result has <aggregate-type>: <scalar>. The result <data-type> has <character>.

Operation: evaluate-onsource-pv

Step 1. Return an <evaluated-pseudo-variable-reference>: <pseudo-variable>: <onsource-pv>.

Operation: assign-onsource-pv(t,sv,sd)

where t is an <evaluated-pseudo-variable-reference>,
sv is an <aggregate-value>,
sd is a <data-description>.

Step 1. Perform get-established-onvalue(<onsource-value>) to obtain str. str must not be <fail>. Let n be its length. Convert sv to <character> of length n, using the <data-type> in sd as the source <data-type> for this conversion, to obtain cv.

Step 2. Replace str by cv.

7.5.4.4 Pageno-pv

Arguments: fn

Constraints: fn must have <aggregate-type>: <scalar>. The <data-type> of fn must have <file>.

Attributes: The result has <aggregate-type>: <scalar>. The result <data-type> has integer-type.

Operation: evaluate-pageno-pv(fn)

Step 1. Perform evaluate-expression(fn) to obtain fv. Return an <evaluated-pseudo-variable-reference>: <pseudo-variable>: <pageno-pv>; fv.

Operation: assign-pageno-pv(t,sv,sd)

where t is an ‹evaluated-pseudo-variable-rereference›,
sv is an ‹aggregate-value›,
sd is a <data-description>.

Step 1. Let f be the ‹file-information› designated by the ‹file-value› in t. f must have ‹open›. The ‹complete-file-description› of f must have <print>.

Step 2. Let bv be the ‹basic-value› in sv. Let tdt be a <data-type> which is integer-type (Section 9.1.2). Perform convert(tdt,sd,bv) to obtain a ‹real-value›,rv. 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.

7.5.4.5 Real-pv

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> in <pictured-numeric>), 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-pv(x)

Step 1. Let y be the <variable-reference> in x.

Step 2. Perform evaluate-variable-reference(y) to obtain a ‹generation›,g. Return an ‹evaluated-pseudo-variable-reference›:<pseudo-variable>: <real-pv>; g.

Operation: assign-real-pv(t,sv,sd)

where t is an ‹evaluated-pseudo-variable-reference›,
sv is an ‹aggregate-value›,
sd is a <data-description>.

Step 1. Let t have the form

‹evaluated-pseudo-variable-reference›:
‹generation›,g:
‹evaluated-data-description›,dc
‹allocation-unit-designator›,aud.

Let dr be the same as dc except that all <mode>s have <real>. Perform promote-and-convert(dr,sv,sd) to obtain vr.

Step 2. For each ‹storage-index›,p in g, perform Steps 2.1 and 2.2.

Step 2.1. Let sdc, svr, and sdr be the scalar-elements of dc, vr, and dr, respectively, corresponding to p.

Step 2.2.

Case 2.2.1. sdc has <arithmetic>, but not <pictured-numeric>.

Let sg be a

```
<generation>:
   <evaluated-data-description>:
        sdc;
   aud
   <storage-index-list>:
        p.
```

Let z1 be the value of sg (z1 contains a single <complex-value>). Let z2 be a <complex-value> with first component equal to the component of svr and second component as z1. Perform set-storage(sg,bvl) where bvl is a <basic-value-list> containing z2.

Case 2.2.2. sdc has <pictured-numeric>.

Let sg be a

```
<generation>:
   <evaluated-data-description>:
        sdr;
   aud
   <storage-index-list>:
        p.
```

Perform set-storage(sg,bvl) where bvl is a <basic-value-list> containing svr.

7.5.4.6 String-pv

Arguments: x

Constraints: x must have the form <argument>:<expression>: <value-reference>: y;;; where y is a <variable-reference>. Let ad be the <declaration> designated by the <declaration-designator> in x. Each <item-data-description> in ad must have <unaligned>. Further, one of the following two conditions must hold:

(1) all <data-type>s in ad must have <character>: <nonvarying> or <pictured>, or

(2) all <data-type>s in ad must have <bit>: <nonvarying>.

Attributes: The result has <aggregate-type>: <scalar>. The result <data-type> has the derived common <string-type> of the <data-type>s of x, and <nonvarying>.

Operation: evaluate-string-pv(x)

Step 1. Let y be the <variable-reference> in x. Perform evaluate-variable-reference(y) to obtain a <generation>,gy, which must be connected. Let the <evaluated-data-description>, <allocation-unit-designator>, and <storage-index-list> components of gy be d, aud and s, respectively.

Step 2. If the <data-type> of the i'th scalar-element in the <generation> has <string>, let k[i] be the value of the corresponding <maximum-length> component in d; otherwise let k[i] be the associated character-string length of the <pictured> in d.

Step 3. Return a <generation>,g, with components as follows. The <evaluated-data-description> of g is described under Attributes, above, with <maximum-length> the sum of all the k[i]. The <allocation-unit-designator> of g is a copy of aud. The <storage-index-list> of g contains a copy of the first component of s.

7.5.4.7 Substr-pv

Arguments: t, st [,le]

Constraints: t must have the form <argument>: <expression>: <value-reference>: <variable-reference>.

All the <data-type>s of t must have <string>. The <aggregate-type>s of the <argument>s st and le must be promotable to the <aggregate-type> of t. All <data-type>s 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>,g, perform evaluate-expression(st) and evaluate-expression(le), if le occurs, to obtain <aggregate-value>s, x and y.

Step 2. Corresponding <bound-pair>s in the <aggregate-type>s of g, x, and y must be equal.

Step 3. Let x' and y' be <aggregate-value>s whose <aggregate-type>s 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 an <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>,g:
        <evaluated-data-description>,edd
        <allocation-unit-designator>,aud
        <storage-index-list>,sil;
    <aggregate-value>,x1
    [<aggregate-value>,y1];,
```

sv is an <aggregate-value>,
sd is a <data-description>.

Step 1. Corresponding <bound-pair>s in the <aggregate-type>s of sv and g must be equal. For each <storage-index>,p, in sil, taken in any order, perform Steps 1.1 through 1.3.

Step 1.1. Let pp be the ordinal of p within sil. Perform find-item-data-description(edd,pp) to obtain an <item-data-description>,idd. Let sg be a

```
<generation>:
    <evaluated-data-description>:
        <data-description>:
            idd;;
    aud
    <storage-index-list>:
        p.
```

Perform value-of-generation(sg) to obtain an <aggregate-value>,av. If sd has <varying> then av must not contain <undefined>. Let k be the length of the <character-string-value> or <bit-string-value> in av (cf. Section 9.1.3.4).

Step 1.2. Let i be the scalar-element in x1 corresponding to p in sil. If y1 exists, let j be the scalar-element in y1 corresponding to p in sil; otherwise let j=k-i+1.

Step 1.3.

Case 1.3.1. $0 \le i-1 \le j+i-1 \le k$.

Step 1.3.1.1. Let st be the <string-type> contained in sg. Let sgl be a

```
<generation>:
    <evaluated-data-description>:
       <data-description>:
         <item-data-description>:
           <data-type>:
             <computational-type>:
               <string>:
                   st
                   <maximum-length>:
                        <extent-expression>:
                             <integer-value>:
                                  j;;;
                   <nonvarying>;;;;;;
          aud
          <storage-index-list>:
                   p.
```

If p contains a <position-index>, increment its value by i-1; otherwise append to p a <position-index> of i.

Step 1.3.1.2. Perform convert(dtl,ssv,ssd) to obtain ssvl, where dtl is the <data-type> of sgl, ssv is the scalar-element in sv corresponding to p in sil, and ssd is the <data-type> in sd corresponding to p in sil.

Step 1.3.1.3. Perform set-storage(sgl,ssvl).

Case 1.3.2. (Otherwise).

Perform raise-condition(<stringrange-condition>).

7.5.4.8 Unspec-pv

Argument: x

Constraints: x must have the form <argument>: <expression>: <value-reference>: <variable-reference>.

x must have <aggregate-type>: <scalar>.

Attributes: The result has <aggregate-type>: <scalar>. The result <data-type> has <bit>.

Operation: evaluate-unspec-pv(x)

Step 1. Let y be the <variable-reference> in x. Perform evaluate-variable-reference(y) to obtain a <generation>,g. Return an <evaluated-pseudo-variable-reference>: <pseudo-variable>: <unspec-pv>; g.

Operation: assign-unspec-pv(t,sv,sd)

where t is an <evaluated-pseudo-variable-reference>,
sv is an <aggregate-value>,
sd is a <data-description>.

Step 1. sv must have <aggregate-type>: <scalar>. Let g be the <generation> in t. Convert sv to <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 sv a <basic-value>,v, depending on properties of g. This value may contain <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⟩.

result: a ⟨generation⟩.

Step 1. Let d be the ⟨declaration⟩ designated by the ⟨declaration-designator⟩,dp, of vr.

Step 2.

Case 2.1. The ⟨storage-type⟩ of d has ⟨based⟩.

Perform select-based-generation(vr) to obtain the ⟨generation⟩,g.

Case 2.2. The ⟨storage-type⟩ of d has ⟨defined⟩.

Perform evaluate-defined-reference(vr) to obtain the ⟨generation⟩,g. 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-directory⟩ 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-value⟩.

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⟩,g.

Step 4. If vr has a ⟨by-name-parts-list⟩ perform evaluate-by-name-parts-list(g,vr,d) to obtain a ⟨generation⟩,g.

Step 5. If vr has a ⟨subscript-list⟩, then perform select-subscripted-reference(g,sl), where sl is the ⟨subscript-list⟩, to obtain a ⟨generation⟩,g.

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 artifice to make the proper target ⟨offset⟩s available for conversion in the operation assign.

Step 7. Return g.

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>,edd:
      <data-description>,dd;
    <allocation-unit-designator>,aud
    <storage-index-list>,sil.
```

Let m be the number of elements in sil.

Step 2. Let bv[i] be the i'th element of the <basic-value-list> of the <allocation-unit> designated by aud.

Step 3. Let i be the <integer-value> contained in the <basic-value-index> of the first element of sil. Let j be the <integer-value> contained in the <position-index> of the first element of sil, if this <position-index> exists, and 1 otherwise. For k=1,...,m 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 <nonvarying>.

Step 3.2.1.1. Let the k'th element of sil 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 i, 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 ml be the <maximum-length> of idd. Let j be j+ml.

Step 3.2.1.5. Let nel[i] be the number of elements in the <character-value-list> or <bit-value-list> of bv[i]. Repeat Step 3.2.1.5.1 while j is greater than nel[i].

Step 3.2.1.5.1. Let j be j-nel[i]. Let i be i+1.

Case 3.2.2. (Otherwise).

Step 3.2.2.1. Let the k'th element of sil 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 i, 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 <based> 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 valr be the <value-reference> of lq.

Case 1.2. vr has no <locator-qualifier>.

The <based> component of the <declaration> designated by the <declaration-designator> of vr has a <value-reference>. Let this be valr.

Step 2. Perform evaluate-value-reference(valr) to obtain an ‹aggregate-value› with a ‹basic-value›,v.

Step 3. Let dt be the <data-type> of valr. 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›,aud, the ‹evaluated-data-description›,edd, and the ‹storage-index-list›,sil.

Step 3.

Case 3.1. The immediate component of dd is a <structure-data-description>,sdd.

Step 3.1.1. sdd contains an <identifier-list>,mil, and a <member-description-list>,mdl.

Step 3.1.2. If vr contains an <identifier-list>,il, 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.

Step 3.1.3. Let dd1 be

```
<data-description>:
    <structure-data-description>:
        <identifier-list>,il1
        <member-description-list>,mdl1;;
```

where il1 contains a copy of the first m elements of il, and mdl1 contains a copy of the first m elements of mdl.

Case 3.2. (Otherwise).

Let dd1 be a copy of dd.

Step 4. Perform evaluate-data-description-for-reference(dd1,g) to obtain an <evaluated-data-description>,edd1.

Step 5.

Case 5.1. edd1 has no <refer-option> node and both edd1 and edd are such that each <data-type> component contains <unaligned> and

either (1) each <data-type> component contains <nonvarying> and <bit>,

or (2) each <data-type> component contains either <pictured>, or <nonvarying> and <character>.

Perform overlay-strings(edd1,g,1) to obtain a <storage-index-list>,sil1.

Case 5.2. (Otherwise).

Let cedd be a copy of edd. Delete from cedd and edd2, a copy of edd1, all occurrences of <refer-option>, <initial>, <local>, and all <variable-reference> components of <offset>. Each subnode of edd2 must be equal to the corresponding subnode of cedd except that if edd2 is an

```
<evaluated-data-description>:
    <data-description>:
        <structure-data-description>:
            <member-description-list>,mdl2;;;
```

then the <member-description-list> in cedd corresponding to mdl2 may have more components than mdl2, and the excess components are ignored.

Let dd1e be the <data-descript.on> of edd1. Perform scalar-elements-of-data-description(dd1e) to obtain n, the number of scalar-elements corresponding to dd1e. Let sil1 be a <storage-index-list> containing a copy of the first n elements of sil.

Step 6. Return a <generation>: aud edd1 sil1.

7.6.4 OVERLAY-STRINGS

This operation constructs a <storage-index-list> that reflects the fact that in a string-overlay defined reference or a reference to a based variable, the resulting <generation> may describe strings that start inside a <character-string-value> or <bit-string-value>. The <position-index> is used to define the starting point of the strings.

Operation: overlay-strings(edd,g,indx)

where edd is an <evaluated-data-description>,
g is a <generation>,
indx is an integer.

result: a <storage-index-list>.

Step 1. Let dd be the <data-description> simply contained in edd. Perform scalar-elements-of-data-description(dd) to obtain the number of scalar-elements, n, described by edd. Perform Step 1.1 for i=1,...,n.

Step 1.1. Perform find-item-data-description(dd,i) to obtain the <item-data-description>,idd of the i'th scalar-element described by dd. If idd contains <string> then let m[i] be the value of the <maximum-length> of idd; otherwise, let m[i] be the associated character-string length of the <data-type> of idd.

Step 2. Let ddg be the <data-description> simply contained in g. Perform scalar-elements-of-data-description(ddg) to obtain the number of scalar-elements corresponding to g, ng. Perform Step 2.1 for i=1,...,ng.

Step 2.1. Perform find-item-data-description(ddg,i) to obtain the <item-data-description>,idd, of the i'th scalar element corresponding to g. If idd contains <string> then let mg[i] be the value of the <maximum-length> of idd; otherwise, let mg[i] be the associated character-string length of the <data-type> of idd.

Step 3. Let indxc be a copy of indx. Let ml be the value of the sum mg[1] +...+ mg[ng]. Let sil 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<ng and such that the value of the sum mg[1] +...+ mg[j] is less than indxc. Let the value of this sum be slg. Note that j and slg may be zero.

Step 3.2. Let j be j+1.

Step 3.3. Let bvi be the <basic-value-index> of the j'th element of the <storage-index-list> of g, and let p be the <position-index> of the same element.

Step 3.4. Let px be the <integer-value> containing the value indxc-slg-1+p.

Step 3.5. Append

```
<storage-index>:
  bvi
  <position-index>:
    px;;
```

to sil.

Step 3.6. Let indxc be indxc+m[i]. The value of indxc must not be greater than ml+1.

Step 3.7. Let i be i+1.

Step 4. Return sil.

7.6.5 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(dd,g)

where dd is a <data-description>,
g is a <generation>.

result: an <evaluated-data-description>.

Step 1. Let cdd be a copy of dd. For each <extent-expression>,ee, of cdd, taken in left-to-right order, perform Step 1.1.

Step 1.1.

Case 1.1.1. ee has a <refer-option>,ro.

Step 1.1.1.1. Let idl be the <identifier-list> of ro. Let n be the number of elements in idl. Let cdd1 be a copy of cdd and let ddp designate the <data-description>,cdd1. Let i = 1. Perform Steps 1.1.1.1.1 through 1.1.1.1.3 n times.

Step 1.1.1.1.1. Let id be the i'th element of idl. Let sdd be the <structure-data-description> that is the immediate component of the <data-description> designated by ddp. Let idl1 be the <identifier-list> of sdd. Let the m'th element of idl1 be identical with id.

Step 1.1.1.1.2. Let mdl1 be the <member-description-list> of sdd. Delete all elements following the m'th element in both idl1 and mdl1. Set ddp to designate the <data-description> immediately contained in the m'th element of mdl1.

Step 1.1.1.1.3. Set i to i+1.

Step 1.1.1.2. Perform scalar-elements-of-data-description(cdd1) to obtain j, the number of scalar elements corresponding to cdd1.

Step 1.1.1.3. Let bv be the j'th element of the <basic-value-list> designated by g. Replace the <expression> of ee 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 evaluate-expression-to-integer(e), where e is the <expression> of ee, to obtain the <integer-value>,j. Replace e by j.

Step 2. Return the <evaluated-data-description> containing cdd, as modified, as its immediate component.

7.6.6 SELECT-QUALIFIED-REFERENCE

This operation selects the part of a <generation> that corresponds to an <identifier-list>.

Operation: select-qualified-reference(g,idl,d)

where g 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 csil be a copy of the <storage-index-list> of g and cedd be a copy of the <evaluated-data-description> of g. Let redd and tedd both be the immediate component of the <data-description> of cedd.

Step 2. Let ecount be 1. Let m be the number of elements in idl. For j=1,...,m, perform Steps 2.1 through 2.9.

Step 2.1. If tdd is a <dimensioned-data-description> then perform Steps 2.1.1 and 2.1.2.

Step 2.1.1. Let bpl be the <bound-pair-list> of tedd. Let ub[i] and lb[i] be the <integer-value> components of the <upper-bound> and the <lower-bound> of the i'th <bound-pair> of bpl respectively. Let ecount be the value of the product

$$\prod_{i=1}^{k} (ub[i]-lb[i]+1)$$

where k is the number of elements of bpl.

Step 2.1.2. Let tdd and tedd be the immediate components of the <element-data-description>s of tdd and redd respectively.

Step 2.2. tdd and tedd will both be a <structure-data-description>. Let mdl and mil be, respectively, the <member-description-list> and the <identifier-list> of tdd. Let emdl be the <member-description-list> of tedd.

Step 2.3. Let the n'th element of mil be the one that is identical with the j'th element of idl.

Step 2.4. Let the number of elements of mdl be nm. Let dd[k] be the <data-description> of the k'th element of emdl. Perform scalar-elements-of-data-description(dd[k]) to obtain the number of scalar-elements nse[k] corresponding to dd[k] for 1≤k≤nm.

Step 2.5. Let i1 be the value of the sum nse[1] +...+ nse[n-1], i2 be nse[n], and i3 be the value of the sum nse[n+1]+...+nse[nm].

Step 2.6. Let indx be 1+(i1+i2+i3)*(ecount-1). Perform Steps 2.6.1 through 2.6.3 ecount times.

Step 2.6.1. Delete i3 successive elements of csil starting with the element whose index is indx+i1+i2.

Step 2.6.2. Delete i1 successive elements of csil starting with the element whose index is indx.

Step 2.6.3. Set indx to indx-(i1+i2+i3).

Step 2.7. Let tdd be the immediate component of the <data-description> of the n'th element of mdl and let tedd be the immediate component of the <data-description> of the n'th element of emdl.

Step 2.8.

Case 2.8.1. tdd is a <dimensioned-data-description> and redd is a <dimensioned-data-description>.

Append the elements of the <bound-pair-list> of tedd to the <bound-pair-list> of redd. Replace the <element-data-description> of redd by the <element-data-description> of tedd.

Case 2.8.2. tdd is a <dimensioned-data-description> and redd is a <structure-data-description>.

Replace the <element-data-description> of redd by tedd.

Case 2.8.3. (Otherwise).

Replace redd by tedd.

Step 2.9. Let tedd be redd.

Step 3. Return the <generation> comprising a copy of the <allocation-unit-designator> of g, csil, and cedd.

7.6.7 SELECT-SUBSCRIPTED-REFERENCE

This operation selects the part of a ‹generation› that corresponds to a given ‹subscript-list›.

Operation: select-subscripted-reference(g,sbl)

where g is a ‹generation›,
sbl is a ‹subscript-list›.

result: a ‹generation›.

Step 1. Let cg be a copy of g. cg will have the form

```
‹generation›:
     ‹evaluated-data-description›:
          ‹data-description›:
               ‹dimensioned-data-description›:
                    ‹element-data-description›,eldd
                    ‹bound-pair-list›,bpl;;;
     ‹allocation-unit-designator›
     ‹storage-index-list›,sil.
```

Step 2. Let cbpl be a copy of bpl.

Step 3. Let dd be a ‹data-description› that immediately contains a copy of the immediate subtree of eldd. Perform scalar-elements-of-data-description(dd) to obtain the integer n.

Step 4. Let m be the number of elements of sbl. Let sb[i] be the i'th element of sbl. Let cbp[i] be the i'th element of cbpl and let ub[i] and lb[i] be, respectively, the ‹integer-value› contained in the ‹upper-bound› and ‹lower-bound› of cbp[i]. For i=1,...,m, where the values are chosen in any order, perform Steps 4.1 and 4.2.

Step 4.1.

Case 4.1.1. sb[i] has ‹asterisk›.

Let ubp be a copy of cbp[i].

Case 4.1.2. (Otherwise).

Step 4.1.2.1. If sb[i] immediately contains an ‹expression›,e then perform evaluate-expression-to-integer(e) to obtain the ‹integer-value›,v. Otherwise let v be the ‹integer-value› contained in sb[i].

Step 4.1.2.2. If v is less than lb[i] or v is greater than ub[i], then perform raise-condition(‹subscript-range-condition›). Let ubp be

```
‹bound-pair›:
     ‹upper-bound›:
          ‹extent-expression›:
               v;;
     ‹lower-bound›:
          ‹extent-expression›:
               v.
```

Step 4.2. Replace cbp[i] by ubp.

Step 5. Perform extract-slice-of-array(bpl,cbpl,n,sil) to obtain a ‹storage-index-list›,nsil. Replace sil by nsil.

Step 6. Let nbpl be a ‹bound-pair-list› with no components. For i=1,...,m, if sb[i] has ‹asterisk› then append a copy of the i'th component of bpl to nbpl. If nbpl has no components then replace the ‹data-description› of cg by dd; otherwise replace bpl by nbpl.

Step 7. Return 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 g is a ⟨generation⟩,
vr is a <variable-reference>,
d is a <declaration>.

result:a ⟨generation⟩.

Step 1. Let aud be the ⟨allocation-unit-designator⟩ of g.
Let rg be

⟨generation⟩:
⟨evaluated-data-description⟩:
<data-description>:
<structure-data-description>:
<member-description-list>,mdl;;;
aud
⟨storage-index-list⟩,sil.

Step 2. Let m be the number of elements in the <by-name-parts-list>,bnpl, of vr. For i=1,...,m perform Steps 2.1 through 2.3.

Step 2.1. Let bnp be the i'th element of bnpl. Let idl be a copy of the <identifier-list> of vr, if one exists; otherwise let idl be an <identifier-list> with no elements. Append the elements of the <identifier-list> of bnp to idl.

Step 2.2. Perform select-qualified-reference(g,idl,d) to obtain a ⟨generation⟩,ng.

Step 2.3. Let dd be the <data-description> of the ⟨evaluated-data-description⟩ of ng. Let md be a <member-description>: dd. Append md to mdl. Append the elements of the ⟨storage-index-list⟩ of ng to sil.

Step 3. Return rg.

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-defined-reference(vr)

where vr is a <variable-reference>.

result: a ⟨generation⟩.

Step 1. Let dp be the <declaration-designator> of the <variable-reference>,vr, let d be the <declaration> designated by dp, and let dc be the <defined> component of d. Perform find-block-state-of-declaration(dp) to obtain the ⟨block-state⟩,bs.

Step 2. Let dde be the ⟨defined-directory-entry⟩ in bs whose <identifier> is equal to that of d. Let edd be the ⟨evaluated-data-description⟩ of dde.

Step 3. d has a <base-item> with a <variable-reference>,bvr1. Let bvr2 be a copy of bvr1. If bvr2 contains a <subscript-list> with <isub> components then delete the <subscript-list> in bvr2.

Step 4. Perform evaluate-variable-reference(bvr2) to obtain the ⟨generation⟩,g.

Step 5.

Case 5.1. dc has a <position> component.

Perform evaluate-string-overlay-defined-reference(vr,edd,g) to obtain the

<generation>,gd.

Case 5.2. dc has a <subscript-list> with an <isub> component.

Perform evaluate-isub-defined-reference(vr,edd,g) to obtain the <generation>,gd.

Case 5.3. (Otherwise).

Perform check-simply-defined-reference(vr). If the value obtained is <true>, then perform evaluate-simply-defined-reference(vr,edd,g) to obtain the <generation>,gd. Otherwise perform evaluate-string-overlay-defined-reference(vr,edd,g) to obtain the <generation>,gd.

Step 6. Return gd.

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>,
g is a <generation>.

result: a <generation>.

Step 1. Let eddg be the <evaluated-data-description> of g.

Step 2. For every <extent-expression>,e contained in edd 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 eddg.

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 eddg.

Case 2.3. (Otherwise).

v must be less than or equal to the corresponding <integer-value> of eddg.

Step 3. If edd contains any <bound-pair>, perform adjust-bound-pairs(g,edd) to yield a new <storage-index-list>,sil, corresponding to the <bound-pair>s of edd. Let g1 be the <generation> constructed from a copy of the <allocation-unit-designator> of g, edd, and sil.

Step 4. If vr has an <identifier-list>,idl, then perform select-qualified-reference(g1,idl,d), where d is the <declaration> designated by the <declaration-designator> of vr, to obtain a <generation>. Replace g1 with this <generation>.

Step 5. If vr has a <by-name-parts-list>, perform evaluate-by-name-parts-list(g,vr,d) to obtain a <generation>. Replace g1 with this <generation>.

Step 6. If vr has a <subscript-list>,sbl, perform select-subscripted-reference(g1,sbl) to obtain a <generation>. Replace g1 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: adjust-bound-pairs(g,edd)

where g is a ‹generation›,
edd is an ‹evaluated-data-description›.

result: a ‹storage-index-list›.

Step 1. Let eddg be the ‹evaluated-data-description› of g and let csil be a copy of the ‹storage-index-list› of g.

Step 2. If the <data-description> of edd immediately contains a <dimensioned-data-description>,ddd, then perform Steps 2.1 through 2.4.

Step 2.1. Let dddg be the simply contained <dimensioned-data-description> of eddg. Let bpl and bplg be, respectively, the <bound-pair-list>s of ddd and dddg.

Step 2.2. Let m be the number of elements in bpl. For i=1,...,m, the values being chosen in any order, perform Steps 2.2.1 through 2.2.5.

Step 2.2.1. Let bp[i] and bpg[i] be the i'th element of, respectively, bpl and bplg.

Step 2.2.2. Let ub[i] and ubg[i] be the ‹integer-value› contained in the <upper-bound> of, respectively, bp[i] and bpg[i].

Step 2.2.3. Let lb[i] and lbg[i] be the ‹integer-value› contained in the <lower-bound> of, respectively, bp[i] and bpg[i].

Step 2.2.4. If ub[i] is greater than ubg[i] then perform raise-condition(<subscriptrange-condition>).

Step 2.2.5. If lb[i] is less than lbg[i] then perform raise-condition(<subscriptrange-condition>).

Step 2.3. Let dd be the <data-description> of the <element-data-description> of dddg. Perform scalar-elements-of-data-description(dd) to obtain the integer n.

Step 2.4. Perform extract-slice-of-array(bpl,bplg,n,csil) to obtain a ‹storage-index-list›,nsil.

Step 3. Return nsil.

7.6.12 EVALUATE-ISUB-DEFINED-REFERENCE

This operation takes an isub-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-isub-defined-reference(vr,edd,g)

where vr is a <variable-reference>,
edd is an ‹evaluated-data-description›,
g is a ‹generation›.

result: a ‹generation›.

Step 1. Let ddd1 and bpl be, respectively, the immediately contained <dimensioned-data-description> and its <bound-pair-list> in the <data-description> of edd. Let n be the number of elements in bpl.

Step 2. Let dp and sbl be, respectively, the immediately contained <declaration-designator> and <subscript-list> of vr. Let sbla be a <subscript-list> containing a copy of the first n elements of sbl and let sblb be a <subscript-list> containing a copy of the remaining elements, if any, of sbl.

Step 3. Choose, in any order, each element of sbla that does not immediately contain <asterisk> and perform Steps 3.1 through 3.3.

Step 3.1. Let e be the chosen element of sbla and let i be its ordinal in sbla.

Step 3.2. Perform evaluate-expression(e) and convert the result obtained to integer-type of value v. Replace the <expression> component of the i'th element of sbla 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(<subscriptrange-condition>).

Step 4. Construct a <subscript-list-list>,sbll, whose single element is a copy of sbla. If sbla contains any <asterisk> component, then expand sbll by performing expand-list-of-subscript-lists(sbll,edd) to obtain a new <subscript-list-list>, and replace sbll with this.

Step 5. Let d be the <declaration> designated by dp. For each element, esbl, of sbll perform transform-subscript-list(esbl,d) to obtain a <subscript-list>, and replace esbl in sbll with this. This transforms each <subscript-list> of sbll 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>,nsil by performing Steps 6.1 and 6.2 for each element of sbll chosen in left-to-right order.

Step 6.1. Let sblc be the chosen element of sbll. Perform select-subscripted-reference(g,sblc) to obtain a <generation>,gl.

Step 6.2. Append the elements of the <storage-index-list> of gl to nsil.

Step 7. Let bplc be a copy of the <bound-pair-list>,bpl. Delete each element of sbla that is not <asterisk> and also delete the element of bplc with the same ordinal. Let vrc be a copy of vr. Replace the copy of sbl in vrc by a <subscript-list> constructed by appending the elements of a copy of sblb to a copy of sbla.

Step 8. Let eddgc be a copy of the <evaluated-data-description> of q. The <data-description> of eddgc has an immediate <dimensioned-data-description> component; let this be ddd2. Let eddc be a copy of edd and let its contained copy of ddd1 be ddd1c.

Case 8.1. As a result of Step 7, bplc no longer exists.

Step 8.1.1. Replace ddd1c in eddc by the subtree of its <element-data-description>.

Step 8.1.2. Replace ddd2 in eddgc by the subtree of its <element-data-description>.

Case 8.2. (Otherwise).

Replace the <bound-pair-list> of eddgc by bplc.

Step 9. Let g2 be a <generation> constructed from a copy of the <allocation-unit-designator> of g, eddgc, and nsil. Perform check-simply-defined-reference(vrc). The value obtained must be <true>. Perform evaluate-simply-defined-reference(vrc,eddc,g2) to obtain a <generation>,g3.

Step 10. Return g3.

7.6.13 EXPAND-LIST-OF-SUBSCRIPT-LISTS

This operation expands a <subscript-list-list> so that each element that is an <asterisk>, 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: expand-list-of-subscript-lists(sbll,edd)

where sbll is a <subscript-list-list>,
edd is an <evaluated-data-description>.

result: a <subscript-list-list>.

Step 1. Let csbll be a copy of sbll. Let bpl be the <bound-pair-list> of the <dimensioned-data-description> that is simply contained by edd. Perform Steps 1.1 through 1.3 until no <asterisk> remains in csbll.

Step 1.1. Choose an element, sbl, of csbll, such that one of the elements of sbl is <asterisk>. Let an <asterisk> be the i'th element of sbl.

Step 1.2. Let bp be the i'th element of bpl. bp has the two components <upper-bound>,ub, and <lower-bound>,lb.

Step 1.3. Replace sbl by (ub-lb+1) copies of sbl modified such that the j'th copy has its <asterisk> component in the i'th position replaced by an <expression> containing the <integer-value>, (lb-1+j).

Step 2. Return csbll.

7.6.14 TRANSFORM-SUBSCRIPT-LIST

This operation takes a <subscript-list> and performs the transformation specified by isub-defining to generate a new <subscript-list>.

Operation: transform-subscript-list(sbl,d)

where sbl is a <subscript-list>,
d is a <declaration>.

result: a <subscript-list>.

Step 1. In the <variable-reference> of the <defined> component of d there is a <subscript-list>. Let csbl be a copy of this <subscript-list>.

Step 2. For each <isub>,ic, of csbl perform Step 2.1.

Step 2.1. Let i be the value of the <integer> component of ic. i must be greater than zero and less than or equal to the number of elements in sbl. Replace ic in csbl by the <integer-value> of the i'th element of sbl.

Step 3. For each <expression>,e of csbl perform Step 3.1.

Step 3.1. Perform evaluate-expression-to-integer(e) to obtain an <integer-value>,iv, and replace e in csbl by iv.

Step 4. Return csbl.

7.6.15 EVALUATE-STRING-OVERLAY-DEFINED-REFERENCE

This operation takes a <variable-reference> for a string-overlay-defined <variable>, the <evaluated-data-description> 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,edd,g)

where vr is a <variable-reference>,
edd is an <evaluated-data-description>,
g is a <generation>.

result: a <generation>.

Step 1. g must be a connected <generation>. The scalar-elements of g and the scalar-elements described by edd must each contain <unaligned> and

either (1) each contain <nonvarying> and <bit>,

or (2) each contain either <pictured>, or <nonvarying> and <character>.

Step 2. Let d be the <declaration> designated by the <declaration-designator> of vr. If the <defined> component of d contains <position>, then perform evaluate-expression(e), where e is the <expression> of <position>, and convert the result to integer-type of value p. Otherwise, let p be 1. The value of p must be greater than or equal to 1.

Step 3. Perform overlay-strings(edd,g,p) to obtain a <storage-index-list>,nsil. Let gl be the <generation> comprising the <allocation-unit-designator> of g, a copy of edd, and nsil.

Step 4. If vr immediately contains an <identifier-list>,idl, then perform select-qualified-reference(gl,idl,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>,sbl, then perform select-subscripted-reference(g,sbl) to obtain a <generation>,gl.

Step 7. Return gl.

7.6.16 CHECK-SIMPLY-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-description> immediately contained in the <variable> of d. Replace each <bound-pair> in dd by <bound-pair>: <asterisk>;, each <maximum-length> by <maximum-length>: <asterisk>;, and each <area-size> by <area-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 test-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,nbpl,n,sil)

where obpl is a <bound-pair-list>,
nbpl is a <bound-pair-list>,
n is an integer,
sil is a <storage-index-list> or <basic-value-list>.

result: a <storage-index-list> or <basic-value-list>.

Step 1. For each <bound-pair>,obp[i], i=1,...,m, in obpl let oub[i] and olb[i] be the <integer-value> components of the <upper-bound> and <lower-bound>, respectively, of obp[i].

Step 2. For each <bound-pair>,nbp[i], i=1,...,m, in nbpl let nub[i] and nlb[i] be the <integer-value> components of the <upper-bound> and <lower-bound>, respectively, of nbp[i].

Step 3. Let n2 be

$$n * \prod_{i=1}^{m} (nub[i]-nlb[i]+1).$$

If sil is a <storage-index-list> then let nsil be a <storage-index-list> with n2 <storage-index> elements. Otherwise, let nsil be a <basic-value-list> with n2 elements of the form <basic-value>: <undefined>.

Step 4. For each set of integers s[i], i=1,...,m, such that nlb[i] ≤ s[i] ≤ nub[i], perform Steps 4.1 and 4.2.

Step 4.1. Let

$$k = 1+n*(s[m]-olb[m]+\sum_{i=1}^{m-1}((s[i]-olb[i])*\prod_{j=i+1}^{m}(oub[j]-olb[j]+1)))$$

and let

$$k2 = 1+n*(s[m]-nlb[m]+\sum_{i=1}^{m-1}((s[i]-nlb[i])*\prod_{j=i+1}^{m}(nub[j]-nlb[j]+1))).$$

Step 4.2. Replace nsil[k2+j] by sil[k+j] for j=0,...,n-1.

Step 5. Return nsil.

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(e) to obtain the <integer-value>,i and replace e by i.

Step 2.1.2. For each <bound-pair>,bp[i] in bpl, i=1,...,m, let ub[i] and lb[i] be the <integer-value> components of the <upper-bound> and <lower-bound>, respectively, of bp[i].

Step 2.1.3. Let n be the integer

$$\prod_{i=1}^{m} (ub[i]-lb[i]+1),$$

and let svl be a <basic-value-list> consisting of n <undefined> elements.

Case 2.2. (Otherwise).

Step 2.2.1. Let n 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 bsd be a <block-state-designator> designating bs. Let ev be an <entry-value>: epd bsd.

Step 3.2.2.2.

Case 3.2.2.2.1. ep has a <signed-integer-list>,sl.

Let s[i] be the i'th element of sl. Let sn be the integer

$$1+s[m]-lb[m]+\sum_{i=1}^{m-1}(s[i]-lb[i])*a[i]$$

where

$$a[i] = \prod_{j=i+1}^{m}(ub[j]-lb[j]+1).$$

Replace the <undefined> component of the sn'th element of svl by ev.

Case 3.2.2.2.2. (Otherwise).

Replace the <undefined> component of svl by ev.

Case 3.3. nc has a <file> component.

Perform search-file-directory(ncr,svl) to obtain a <basic-value-list>,bvl. Replace svl by bvl.

Case 3.4. nc has a <format> component.

Step 3.4.1. Perform find-block-state-of-declaration(d) to obtain the <block-state>,bs. Let fsl be the <format-statement-list> component of the <begin-block> or <procedure> that simply contains d.

Step 3.4.2. For each <format-statement>,fs, element of fsl 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 fsd be a <format-statement-designator> designating fs and let bsd be a <block-state-designator> designating bs. Let fv be a <format-value>: fsd bsd.

Step 3.4.2.2.

Case 3.4.2.2.1. The <statement-name> of fs has a <signed-integer-list>,sl.

Let s[i] be the i'th element of sl. Let sn be the integer

$$1+s[m]-lb[m]+\sum_{i=1}^{m-1}(s[i]-lb[i])*a[i]$$

where

$$a[i] = \prod_{j=i+1}^{m}(ub[j]-lb[j]+1).$$

Replace the <undefined> component of the sn'th element of svl by fv.

Case 3.4.2.2.2. (Otherwise).

Replace the <undefined> component of svl by fv.

Case 3.5. nc has a <label> component.

Step 3.5.1. Perform find-block-state-of-declaration(d) to obtain the <block-state>,bs. Let eul 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 eul 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 eud be an <executable-unit-designator> designating eu, and let bsd be a <block-state-designator> designating bs. Let lv be a <label-value>: eud bsd.

Step 3.5.2.2.

Case 3.5.2.2.1. The <statement-name> of eu has a <signed-integer-list>,sl.

Let s[i] be the i'th element of sl. Let sn be the integer

$$1+s[m]-lb[m]+\sum_{i=1}^{m-1}(s[i]-lb[i])*a[i]$$

where

$$a[i] = \prod_{j=i+1}^{m}(ub[j]-lb[j]+1).$$

Replace the <undefined> component of the sn'th element of svl by lv.

Case 3.5.2.2.2. (Otherwise).

Replace the <undefined> component of svl by lv.

Step 4.

Case 4.1. nc 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-value>: agt svl.

Step 6. If ncr has a <subscript-list> then perform Steps 6.1 through 6.4.

Step 6.1. Let m, lb[i] and ub[i], i=1,...,m, be as determined in Step 2. Let bpl2 be a <bound-pair-list> with m elements of the form

```
<bound-pair>:
    <upper-bound>,ub2[i]
    <lower-bound>,lb2[i];
```

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 i'th element of the <subscript-list> in ncr has an <asterisk>.

Let lb2[i] and ub2[i] be, respectively, lb[i] and ub[i].

Case 6.1.1.2. The i'th element of the <subscript-list> in ncr has an <expression>,exp.

Perform evaluate-expression-to-integer(exp) to obtain the <integer-value>,v and let lb2[i] and ub2[i] be the integer in v. If v is less than lb[i] or greater than ub[i] perform raise-condition(<subscriptrange-condition>).

Step 6.2. Perform extract-slice-of-array(bpl,bpl2,1,svl) to obtain a <basic-value-list>,bvl2.

Step 6.3. Replace the <basic-value-list> in av by a copy of bvl2.

Step 6.4.

Case 6.4.1. Each element of the <subscript-list> in ncr has <expression>.

Replace the <aggregate-type> of av by <aggregate-type>: <scalar>.

Case 6.4.2. (Otherwise).

Delete from the <bound-pair-list> of av the elements which correspond to elements of the <subscript-list> of ncr which have <expression>.

Step 7. The <aggregate-value> av must not contain <undefined>. Return av.

7.7.2 SEARCH-FILE-DIRECTORY

This operation searches the <file-directory> for the entry or entries that correspond to the <declaration> referred to in a <named-constant-reference>.

Operation: search-file-directory(ncr,svl)

where ncr is a <named-constant-reference>,
svl is a <basic-value-list>.

result: a <basic-value-list>.

Step 1. Let bvl be a copy of svl. Let d be the <declaration> designated by the <declaration-designator> of ncr. Perform Steps 1.1 through 1.2 until bvl contains no <undefined> elements.

Step 1.1. Search the <file-directory> for a <file-directory-entry>,e, whose <identifier> is identical with the <identifier> of d and which has the component <external> if d has the component <external>; otherwise it has a <declaration-designator> component that designates d.

Step 1.2. Let fid be the <file-information-designator> in e and let bv be a <basic-value>: <file-value>: fid. If e has a <subscript-value-list>, replace the element of bvl denoted by this <subscript-value-list> by bv. Otherwise, replace the single element of bvl by bv.

Step 2. Return bvl.

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 PL/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 PL/I program. The concrete-representation of a ‹dataset› is implementation-defined. ‹alpha› and ‹omega› are end-markers for ‹dataset›s that have a sequence.

8.1.1 RECORD DATASETS

A ‹record-dataset› may contain discrete ‹record›s; ‹record-dataset›s without any ‹record›s 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 ‹record› is transmitted, and under implementation-defined circumstances may cause the raising of the <record-condition>.

‹key›s are a means of identifying particular ‹record›s. Within a ‹dataset›, ‹key›s 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 {symbol}s are to be sent to the same line as the preceding ones. The ‹stream-item›s ‹pagemark› 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 PL/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›,dsd. dsd designates 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 ‹omega›. 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 ‹generation› 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 ‹omega›. 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 ‹input›.

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(cond,fv,str,int)

where cond is ‹endfile-condition›, ‹endpage-condition›, ‹key-condition›, ‹name-condition›, ‹record-condition›, ‹transmit-condition›, ‹undefinedfile-condition›, or ‹conversion-condition›,
fv is a [‹file-value›],
str is a [‹character-string-value›],
int is an [‹integer-value›].

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 eioc be an
<evaluated-io-condition>:
<io-condition>:
cond;
fv.

Case 2.2. cond is <conversion-condition>.

Let eioc be <conversion-condition>.

Step 3.

Case 3.1. cond is <name-condition>.

Let cbifs be a <condition-bif-value-list>: <condition-bif-value>: <onfield-value>: str.

Case 3.2. cond is <conversion-condition>.

Let cbifs be a <condition-bif-value-list> simply containing <onsource-value>: str; and <onchar-value>: int. If fv is a <file-value> then let fn be the <character-string-value> in the <filename> in fi and attach an <onfile-value>: fn; to cbifs.

Case 3.3. cond is <transmit-condition>, <record-condition> or <key-condition>, and str is present.

Let cbifs be a <condition-bif-value-list>: <condition-bif-value>: <onkey-value>: str.

Case 3.4. (Otherwise).

Let cbifs be <absent>.

Step 4. Perform raise-condition(eioc,cbifs).

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 <on-statement>s, <signal-statement>s, and <revert-statement>s containing <named-io-condition>s.

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: execute-open-statement(os)

where os is an <open-statement>.

Step 1. For each <single-opening>,sgo in os, in order, perform execute-single-opening(sgo).

Step 2. Perform normal-sequence.

8.5.1.2 Execute-single-opening

Operation: execute-single-opening(sgo)

where sgo is a <single-opening>.

Step 1. Let efdl 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 fo be the <value-reference> in the <file-option> in sgo. Perform evaluate-file-option(fo) to obtain a ⟨file-value⟩,fv.

Step 2.2. If sgo contains a <title-option>,tto, then perform evaluate-title-option(tto) to obtain an ⟨evaluated-title⟩,et, and append ⟨evaluated-file-description⟩: et; to efdl.

Step 2.3. If sgo contains a <tab-option>,tbo, then perform evaluate-tab-option(tbo) to obtain an ⟨evaluated-tab-option⟩,eto, and append ⟨evaluated-file-description⟩: eto; to efdl.

Step 2.4. If sgo contains a <linesize-option>, then let lzo be its <expression>, perform evaluate-expression-to-integer(lzo) to obtain an ⟨integer-value⟩,lz, which must be greater than zero, and append ⟨evaluated-file-description⟩: ⟨evaluated-linesize⟩: lz;; to efdl.

Step 2.5. If sgo contains a <pagesize-option>, then let pzo be its <expression>, perform evaluate-expression-to-integer(pzo) to obtain an ⟨integer-value⟩,pz, which must be greater than zero, and append ⟨evaluated-file-description⟩: ⟨evaluated-pagesize⟩: pz;; to efdl.

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, tn, of sgo which is a terminal node, append ⟨evaluated-file-description⟩: tn; to efdl.

Step 5. Perform open(fv,efdl) to obtain res.

Step 6. If res is ⟨fail⟩ then perform raise-io-condition(<undefinedfile-condition>,fv).

8.5.1.3 Open

Operation: open(fv,efdl)

where fv is a ‹file-value›,
efdl is an ‹evaluated-file-description-list›.

result: ‹succeed› or ‹fail›.

Step 1. Let fi be the ‹file-information› designated by fv. For each terminal node, tn, of the ‹file-description› of fi, append ‹evaluated-file-description›: tn; to efdl.

Step 2. Augment efdl with implied attributes as follows: for each terminal node in efdl which occurs under "Attribute" in the table below, append to efdl trees containing the corresponding "Implied Attributes" categories.

Attribute	Implied Attributes
<direct>	<record> <keyed>
<keyed>	<record>
<print>	<stream> <output>
<sequential>	<record>
<update>	<record>

Step 3. Augment efdl with default attributes as follows: when efdl does not contain any of the "Alternative Attributes" in a line of the table below, append to efdl a tree containing the corresponding "Default" category.

Alternative Attributes	Default
<stream> | <record>	<stream>
<input> | <output> | <update>	<input>

Step 4. If efdl contains <record> but does not contain either <sequential> or <direct>, append <sequential> to efdl.

Step 5. If the ‹filename› in fi contains in order precisely the {symbol}s of the word SYSPRINT, and the ‹file-directory-entry› whose ‹file-information-designator› designates fi has <external>, and efdl contains <stream> and <output>, then append ‹evaluated-file-description›: <print>; to efdl.

Step 6. If both <stream> and <record>, or any two of <input>, <output>, and <update>, or both <direct> and <sequential> are contained in efdl, then return ‹fail›.

Step 7. If efdl contains an ‹evaluated-linesize› then it must contain <stream> and <output>. If efdl contains <stream> and <output> but not an ‹evaluated-linesize› then append ‹evaluated-file-description›: ‹evaluated-linesize›: an implementation-defined ‹integer-value›;; to efdl.

If efdl contains an ‹evaluated-pagesize› then it must contain <print>. If efdl contains <print> but not an ‹evaluated-pagesize› then append ‹evaluated-file-description›: ‹evaluated-pagesize›: an implementation-defined ‹integer-value›;; to efdl.

If efdl contains an ‹evaluated-tab-option› then it must contain <print>. If efdl contains <print> but not an ‹evaluated-tab-option› then append ‹evaluated-file-description›: ‹evaluated-tab-option›: an implementation-defined ‹integer-value-list›;; to efdl.

If efdl does not contain an ‹evaluated-title› then let fn be the ‹filename› in fi and perform evaluate-filename(fn) to obtain an ‹evaluated-title›,t, and append ‹evaluated-file-description›: t; to efdl.

Step 8. Attempt to find, among the ‹dataset›s of the ‹machine-state› (if any), a ‹dataset›,ds, the ‹character-string-value› of the ‹dataset-name› of which matches the ‹character-string-value› of the ‹evaluated-title› in efdl in an implementation-defined manner. If the attempt fails, then return ‹fail›.

If efdl contains ‹record› then ds must contain a ‹record-dataset›.

If efdl contains ‹stream› then ds must contain a ‹stream-dataset›.

If efdl contains ‹sequential› and ‹keyed› then ds must contain a ‹keyed-sequential-dataset›.

If efdl contains ‹sequential› and ‹record› but not ‹keyed› then ds must contain a ‹sequential-dataset›.

If efdl contains ‹direct› then ds must contain a ‹keyed-dataset›.

If efdl contains ‹stream› and ‹input› then ds must not contain any ‹pagemark› or ‹carriage-return›.

If efdl contains ‹record› and ‹keyed› then ds must not contain two distinct ‹keyed-record›s whose ‹key›s are equal.

Step 9. Attach to fi a

```
‹file-opening›,fo:
    ‹dataset-designator›
    ‹complete-file-description›:
        efdl.
```

The ‹dataset-designator› designates ds, which has been associated with fv in Step 8. If efdl contains ‹record› then attach a ‹delete-flag› to fo. If efdl contains ‹print› then attach a ‹page-number› with 1 to fo. If efdl contains ‹stream› and ‹input› then attach ‹first-comma›: ‹on›; to fo.

Step 10.

Case 10.1. efdl contains ‹direct›.

Attach to fo a ‹current-position›: ‹undefined›.

Case 10.2. efdl contains ‹output› and not ‹direct›, and ds contains a ‹record-list›,rl, a ‹keyed-record-list›,rl, or a ‹stream-item-list›,rl.

Attach to fo a ‹current-position› designating the last immediate component of rl.

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 ‹succeed›.

8.5.1.4 Evaluate-tab-option

Operation: evaluate-tab-option(tbo)

where tbo is a ‹tab-option›.

result: an ‹evaluated-tab-option›.

Step 1. For each ‹expression›,e in tbo, in any order, perform evaluate-expression-to-integer(e) to obtain an ‹integer-value›. Let il be an ‹integer-value-list› containing these ‹integer-value›s in the same order as their original ‹expression›s in tbo.

Step 2. The <integer-value>s in il must each be greater than zero, and the list must be in ascending order.

Step 3. Return <evaluated-tab-option>: il.

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(e) to obtain an <aggregate-value>,av. Let dd be the <data-description> immediately contained in e.

Step 2. Let sdt be the <data-type> in dd and let tdt be a <data-type> containing <character>, <nonvarying> and an implementation-defined <maximum-length>. Let bv be the <basic-value> in av. Perform convert(tdt,sdt,bv) to obtain a <character-string-value>,csv.

Step 3. Return <evaluated-title>: 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-value>s in s. Let n be the implementation-defined maximum length of the <evaluated-title>.

Step 2.

Case 2.1. n = 0.

Let csv be <character-string-value>: <null-character-string>.

Case 2.2. m ≥ n > 0.

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, i=1,...,n.

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 1≤i≤n, and where the remaining <character-value>s have ⊠s.

Step 3. Return <evaluated-title>: csv.

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⟩ designated by the ⟨dataset-designator⟩ in fi.

Step 2. If fi contains <output> and an ⟨allocated-buffer⟩,abuf containing the ⟨generation⟩,g 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(g,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-io-condition(<key-condition>,fv,csv), where csv 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(g) 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:

abuf	for <allocated-buffer>
d	for <declaration>
ds	for <record-dataset>
edd	for <evaluated-data-description>
eio	for <evaluated-into-option>
eko	for <evaluated-keyto-option>
epso	for <evaluated-pointer-set-option>
efdl	for <evaluated-file-description-list>
fi	for <file-information>
fn	for <filename>
fv	for <file-value>
g	for <generation>
int	for <integer-value>
k	for <key>
kr	for <keyed-record> or <record>
pos	for <current-position>
r	for <record>

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-keyto-option>]

Operation: execute-read-statement(rs)

where rs is a <read-statement>.

Step 1. Let ers 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 ers.

Step 2.2. If rs contains an <into-option>,ito, perform evaluate-into-option(ito) to obtain an <evaluated-into-option>,eito and attach eito to ers.

Step 2.3. If rs contains a <pointer-set-option>,pso, perform evaluate-pointer-set-option(pso) to obtain an <evaluated-pointer-set-option>,epso and attach epso to ers.

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 ers.

Step 2.5. If rs contains a <key-option>,ko, perform evaluate-key-option(ko) to obtain a <key>,k and attach k to ers.

Step 2.6. If rs contains a <keyto-option>,kto, perform evaluate-keyto-option(kto) to obtain an <evaluated-keyto-option>,ekto and attach ekto to ers.

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⟩: ⟨input⟩; to efdl.

Step 4.3. Perform open(fv,efdl) to obtain rf. If rf is ⟨fail⟩ then perform raise-io-condition(⟨undefinedfile-condition⟩,fv). If, on normal return, fi contains ⟨closed⟩ then perform raise-condition(⟨error-condition⟩).

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 an ⟨evaluated-keyto-option⟩, then ⟨keyed⟩.

Step 6. Perform read(ers).

Step 7. Perform normal-sequence.

8.6.1.2 Read

Operation: read(ers)

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⟩. If ers contains a ⟨key⟩, then let k be this ⟨key⟩ and let k1 be the immediate component of k. Otherwise let k and k1 be ⟨absent⟩.

Step 2. If fi contains an ⟨allocated-buffer⟩,abuf, then perform free(g), where g is the ⟨generation⟩ in abuf, and delete abuf from fi.

Step 3. Perform position-file(ers).

Step 4. If ers does not contain an ⟨evaluated-keyto-option⟩,eko, then go to Step 5. Otherwise let ns be the number of {symbol} subnodes of the ⟨character-string-value⟩,csvk in the ⟨key⟩ in the ⟨keyed-record⟩ designated by the ⟨current-position⟩ in fi. Let dd be a ⟨data-description⟩ containing ⟨character⟩, ⟨maximum-length⟩ containing ns, and ⟨nonvarying⟩. Let avk be an ⟨aggregate-value⟩ containing csvk. Let ekoet be the ⟨evaluated-target⟩ in eko.

Step 5.

Case 5.1. ers contains an ⟨evaluated-into-option⟩,eio.

Let r be the ⟨record⟩ or ⟨keyed-record⟩ designated by the ⟨current-position⟩ in fi.

Step 5.1.1. Let edd1 be the ⟨evaluated-data-description⟩ in r and edd2 be the ⟨evaluated-data-description⟩ in eio. Perform evaluate-size(edd1) to obtain an ⟨integer-value⟩,int1, and perform evaluate-size(edd2) to obtain an ⟨integer-value⟩,int2.

Step 5.1.2. Let g be the ⟨generation⟩ in eio.

Case 5.1.2.1. int1 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. edd1 and edd2 must be equal. Perform set-storage(g,v), where v is the ⟨basic-value-list⟩ in r.

Step 5.1.2.1.2. If ers contains an ⟨evaluated-keyto-option⟩ then perform assign(ekoet,avk,dd).

Case 5.1.2.2. (Otherwise).

Step 5.1.2.2.1. Let nsi be the number of <storage-index>s in g. Let undef be a <basic-value-list> containing <basic-value>: <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 ers contains an <evaluated-keyto-option> then perform assign(ekoet,avk,dd).

Step 5.1.2.2.2. Perform raise-io-condition(<record-condition>,fv,k1).

Case 5.2. ers has an <evaluated-pointer-set-option>,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 fi. Let edd be the <evaluated-data-description> in r. Perform allocate(edd) to obtain a <generation>,g.

Step 5.2.2. Let abuf be an <allocated-buffer>: g. Attach abuf to the <file-opening> in fi.

Step 5.2.3. Let v be the <basic-value-list> in r. Perform set-storage(g,v).

Step 5.2.4. Let pdd be a <data-description> simply containing <pointer> and no other terminal subnodes. Let epsoet be an <evaluated-target> containing the <generation> contained in epso. Let agv be an <aggregate-value> containing the <pointer-value>: g. Perform assign(epsoet,agv,pdd).

Step 5.2.5. If ers contains an <evaluated-keyto-option> then perform assign(ekoet,avk,dd).

Case 5.3. (Otherwise).

If ers contains an <evaluated-keyto-option> then perform assign(ekoet,avk,dd).

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

<evaluated-write-statement>::= <file-value> <evaluated-from-option> [<evaluated-keyfrom-option>]

Operation: execute-write-statement(ws)

where ws is a <write-statement>.

Step 1. Let ews be an <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 ews.

Step 2.2. Let fr be the <from-option> in ws. Perform evaluate-from-option(fr) to obtain an <evaluated-from-option>,efo and attach efo to ews.

Step 2.3. If ws contains a <keyfrom-option>,ko, then perform evaluate-keyfrom-option(ko) to obtain an <evaluated-keyfrom-option>,ekfo and attach ekfo to ews.

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›: <output>; to efdl.

Step 4.3. Perform open(fv,efdl) to obtain sf. If sf is ‹fail› then perform raise-io-condition(<undefinedfile-condition>,fv). If on normal return fi contains ‹closed› then perform raise-condition(<error-condition>).

Step 5. fi must contain:

<record>;
<output>, or <update> and <direct>;
if and only if ews contains an ‹evaluated-keyfrom-option›, then <keyed>.

Step 6. Perform write(ews).

Step 7. Perform normal-sequence.

8.6.2.2 Write

Operation: write(ews)

where ews is an ‹evaluated-write-statement›.

Step 1. Let fi be the ‹file-information› designated by the ‹file-value›,fv, in ews. fi must contain ‹open›. If ews contains an ‹evaluated-keyfrom-option›: ‹character-string-value›,csv; then let k be a ‹key›: csv. 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 g be the ‹generation› immediately contained in abuf. If abuf contains a ‹key›, let kbu be this ‹key› and let csvb be the immediate component of kbu; otherwise let kbu and csvb be ‹absent›. Perform construct-record(g,kbu) to obtain kr.

Step 2.2.2. If fi contains <keyed> and if kbu is equal to any ‹key› in the ‹dataset›,ds, or if kbu is unacceptable to the implementation then perform raise-io-condition(<key-condition>,fv,csvb).

Step 2.2.3. Perform insert-record(kr,fv) 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›,g, then perform free(g) and delete abuf from fi.

Step 4. Let g be the ‹generation› in the ‹evaluated-from-option› in ews. Perform construct-record(g,k) 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-io-condition(<key-condition>,fv,csv).

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.

8.6.3 THE LOCATE STATEMENT

Purpose: The <locate-statement> causes allocation of the specified based variable in an <allocated-buffer>; it may also cause transmission of a based variable previously allocated in an <allocated-buffer>.

8.6.3.1 Execute-locate-statement

<evaluated-locate-statement>::= <declaration-designator> <file-value>
[<evaluated-pointer-set-option>]
[<evaluated-keyfrom-option>]

Operation: execute-locate-statement(ls)

where ls is a <locate-statement>.

Step 1. Let els be an <evaluated-locate-statement> without subnodes.

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 ls. Perform evaluate-file-option(f) to obtain a <file-value>,fv. Attach fv to els.

Step 2.2. If ls contains a <pointer-set-option>,pso, then perform evaluate-pointer-set-option(pso) to obtain an <evaluated-pointer-set-option>,epso and attach epso to els.

Step 2.3. If ls contains a <keyfrom-option>,kfo, then perform evaluate-keyfrom-option(kfo) to obtain an <evaluated-keyfrom-option>,ekfo and attach ekfo to els. If fi contains <keyed> then let chs be a copy of the immediate component of ekfo; otherwise let chs be <absent>. If fi contains <keyed> then let kk be <key>: chs.

Step 2.4. Let cdp be a copy of the <declaration-designator> immediately contained in ls. Attach cdp to els.

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> and <output>.

Step 4.2. Perform open(fv,efdl) to obtain sf. If sf is equal to <fail>, then perform raise-io-condition(<undefinedfile-condition>,fv). 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 <evaluated-keyfrom-option>, then <keyed>.

Step 6. If els does not contain an <evaluated-pointer-set-option> then the <declaration-designator> contained in els must designate a <declaration>,d, of the form

<based>:
<value-reference>:
<variable-reference>,v:
<declaration-designator>,p.

p must designate a <declaration> containing <pointer>. Perform evaluate-variable-reference(v) to obtain g. Let epso be an <evaluated-pointer-set-option>: g; and attach epso to els.

Step 7.

Step 7.1. If fi does not contain an <allocated-buffer>,abuf, then go to Step 8.

Step 7.2. Let g be the <generation> immediately contained in abuf. If abuf contains a <key>, let k be this <key> and let csvb be its immediate component; otherwise let k and csvb be <absent>. Perform construct-record(g,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-io-condition(<key-condition>,fv,csvb).

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(g) and delete abuf from fi.

Step 8. Let dd be the <data-description> immediately contained in the <variable> of the <declaration> designated by cdp. Perform evaluate-data-description-for-allocation(dd) to obtain edd.

Step 9. Perform evaluate-size(edd) to obtain an <integer-value>,int. If int is unacceptable to the implementation then perform raise-io-condition(<record-condition>,fv,chs) and optionally perform exit-from-io.

Step 10. Perform allocate(edd) to obtain g.

Step 11. Let desc be a <data-description> simply containing <pointer> without other terminal subnodes. Let epsog be an <evaluated-target> containing the <generation> in the <evaluated-pointer-set-option> in els. Let agv be an <aggregate-value> containing <pointer-value>: g. Perform assign(epsog,agv,desc).

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(g,d).

Step 13. Let abuf be an <allocated-buffer>: <generation>,g. If fi contains <keyed> then attach kk 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

<evaluated-rewrite-statement>::= <file-value>
[[<key>] <evaluated-from-option>]

Operation: execute-rewrite-statement(rws)

where rws is a <rewrite-statement>.

Step 1. Let erws be an <evaluated-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 erws.

Step 2.2. If rws contains a <from-option>,fr, then perform evaluate-from-option(fr)to obtain an <evaluated-from-option>,efo and attach efo to erws.

Step 2.3. If rws contains a <key-option>,ko, then perform evaluate-key-option(ko) to obtain a <key>,k and attach k to erws.

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> and <update>.

Step 4.2. Perform open(fv,efdl) to obtain sf. If sf is <fail> then perform raise-io-condition(<undefinedfile-condition>,fv). If on normal return fi contains <closed> then perform raise-condition(<error-condition>).

Step 5. fi must contain:

<record>;
<update>;
if erws does not contain a <key>, then <sequential>;
if erws contains a <key>, then <keyed>.

If fi contains <direct> then erws must contain a <key>. If fi contains <sequential> then erws may contain a <key> and an <evaluated-from-option>.

Step 6. Perform rewrite(erws).

Step 7. Perform normal-sequence.

8.6.4.2 Rewrite

Operation: rewrite(erws)

where erws is an <evaluated-rewrite-statement>.

Step 1. Let fi be the <file-information> designated by the <file-value>,fv in erws. fi must contain <open>. If erws contains a <key>, then let k be this <key>; otherwise let k be <absent>. If k is a <key> then let csv be its immediate component; otherwise let csv 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 csv be the immediate component of k.

Case 2.2. k is a <key>.

Perform position-file(erws).

Step 3.

Case 3.1. erws contains an <evaluated-from-option>,efo.

Let g be the <generation> in efo.

Case 3.2. (Otherwise).

fi must contain an <allocated-buffer>,abuf. Let g be the <generation> in abuf.

Step 4. Perform construct-record(g,k) to obtain r.

Step 5. Let edd1 be the <evaluated-data-description> in g and edd2 the <evaluated-data-description> in the <record> designated by the <current-position> in fi. Perform evaluate-size(edd1) to obtain an <integer-value>,int1 and evaluate-size(edd2) to obtain an <integer-value>,int2.

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-io-condition(<record-condition>,fv,csv), and optionally perform exit-from-io.

Step 7. If fi contains an <allocated-buffer>,abuf, then let g be the <generation> in abuf, perform free(g), and delete abuf 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(dls)

where dls is a <delete-statement>.

Step 1. Let eds be an <evaluated-delete-statement> without subnodes.

Step 2. Perform Steps 2.1 and 2.2 in either order.

Step 2.1. Let f be the immediate component of the <file-option> in dls. Perform evaluate-file-option(f) to obtain a <file-value>,fv. Attach fv to eds.

Step 2.2. If dls 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 4.

Step 4.1. Let efdl be an <evaluated-file-description-list> containing <update> and <record>.

Step 4.2. Perform open(fv,efdl) to obtain sf. If sf is <fail> then perform raise-io-condition(<undefinedfile-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 Delete

Operation: delete(eds)

where eds is an <evaluated-delete-statement>.

Step 1. Let fi be the <file-information> designated by the <file-value>,fv, in eds. fi must contain <open>. If eds contains a <key>, then let k be this <key> and let csv be the immediate component of k; otherwise let k and csv be <absent>.

Step 2.

Case 2.1. k is <absent>.

fi must not contain a <delete-flag>. The <current-position> in fi must not contain <undefined>.

Case 2.2. k is a <key>.

Perform position-file(eds).

Step 3. Let ds be the immediate component of the <record-dataset> designated by the <dataset-designator> in fi. Let kr be the node designated by the <current-position> in fi (kr is a <record> or a <keyed-record>). Delete kr from ds.

Step 4.

Case 4.1. fi contains <direct>.

Replace the immediate subnode of the <current-position> in fi by <undefined>.

Case 4.2. fi contains <sequential>.

Replace the immediate subnode of the <current-position> in fi by a <designator> designating the predecessor of kr in ds (this may be <alpha>, a <record> or a <keyed-record>).

Step 5. Attach a <delete-flag> to the <file-opening> in fi.

Step 6. If fi contains an <allocated-buffer>,abuf, then let g be the <generation> in abuf, perform free(g), and delete abuf from fi.

8.6.6 OPERATIONS APPLICABLE TO RECORD I/O

8.6.6.1 Evaluate-from-option

<evaluated-from-option>::= <generation>

Operation: evaluate-from-option(fr)

where fr is a <from-option>.

result: an <evaluated-from-option>.

Step 1. Let v be the <variable-reference> immediately contained in fr. Perform evaluate-variable-reference(v) to obtain a <generation>,g, which must be connected.

Step 2. Return an <evaluated-from-option>: g.

8.6.6.2 Evaluate-into-option

<evaluated-into-option>::= <generation>

Operation: evaluate-into-option(ito)

where ito is an <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>,g, which must be connected.

Step 2. Return an <evaluated-into-option>: g.

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>,g.

Step 2. Return an <evaluated-pointer-set-option>: g.

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>, <nonvarying>, and <maximum-length>: <asterisk>. Let rt be the <data-type> of e. Perform convert(tt,rt,bv) to obtain a <character-string-value>,chs.

Step 3. Return a <key>: chs.

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>, <nonvarying>, and <maximum-length>: <asterisk>. Let rt be the <data-type> of e. Perform convert(tt,rt,bv) to obtain a <character-string-value>,chs.

Step 3. Return an <evaluated-keyfrom-option>: chs.

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 e be the <expression> immediately contained in igo. Perform evaluate-expression-to-integer(e) 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 ktotr be the <target-reference> in kto. Perform evaluate-target-reference(ktotr) to obtain an <evaluated-target>,et.

Step 2. Return an <evaluated-keyto-option>: et.

8.6.6.8 Construct-record

Operation: construct-record(g,k)

where g is a connected <generation>,
k is a [<key>].

result: a <record> or a <keyed-record>.

Step 1. Let agv be an <aggregate-value> which is the value of g. (See Section 7.1.3.) Let edd be the <evaluated-data-description> in g. Let bvl be the <basic-value-list> in agv. Let r be a <record>: edd bvl.

Step 2.

Case 2.1. k is <absent>.

Return r.

Case 2.2. k is a <key>.

Return <keyed-record>: r k.

8.6.6.9 Insert-record

Operation: insert-record(kr,fv)

where kr is a <record> or a <keyed-record>,
fv is a <file-value>.

result: a <designator>.

Step 1. If kr is a <keyed-record> then let k be the <key> in kr and let csv be the immediate component of k; otherwise let k and csv be <absent>. Let fi be the <file-information> designated by fv. Let rl be the <record-list> or <keyed-record-list> in the <dataset> designated by the <dataset-designator> in fi. Let edd be the <evaluated-data-description> in kr. Perform evaluate-size(edd) to obtain an <integer-value>,int. If int is not acceptable to the implementation then perform Step 1.1.

Step 1.1. If kr is a <record>, let kr be a new implementation-defined <record>; otherwise let kr be an implementation-defined <keyed-record> with an equal <key>. Optionally perform Step 2. If this operation is preceded in its immediately containing <operation-list> by an <operation> for program-epilogue then perform some implementation-defined action; otherwise perform raise-io-condition(<record-condition>,fv,csv). Optionally perform exit-from-io.

Step 2.

Case 2.1. fi contains <direct>.

Attach kr to rl in a position chosen in an implementation-defined way.

Case 2.2. fi contains <keyed> and <sequential>.

Attach kr to rl in an implementation-defined position which may depend on the <key> in kr, on the <record> in kr and the <current-position> in fi.

Case 2.3. fi contains <sequential> and not <keyed>.

Append kr to rl in the position immediately following the immediate component of rl designated by the <current-position> in fi.

Step 3. Return a <designator> designating kr in rl.

8.6.6.10 Position-file

Operation: position-file(evst)

where evst is an <evaluated-read-statement>, an <evaluated-rewrite-statement> or an <evaluated-delete-statement>.

Step 1. If evst contains a <key> then let k be this <key> and let csv be its immediate component; otherwise let k and csv both be <absent>. Let fv be the <file-value> in evst, let fi be the <file-information> designated by fv, let ds be the <record-dataset> in the <dataset> designated by the <dataset-designator> in fi, and let pos be the <current-position> in fi.

Step 2.

Step 2.1. If k is a <key> and k is unacceptable to the implementation then perform raise-io-condition(<key-condition>,fv,csv).

Step 2.2.

Case 2.2.1. k is <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 fi does not contain a <delete-flag>, then attach a <delete-flag> to the <file-opening> in fi.

Step 2.2.2.2. Replace the immediate component of pos by <undefined>.

Step 2.2.2.3. Perform raise-io-condition(<key-condition>,fv,csv).

Case 2.2.3. k is equal to a <key> in a <keyed-record>,kr, in ds.

Step 2.2.3.1. If fi contains a <delete-flag>,dfl, then delete dfl from fi.

Step 2.2.3.2. Replace the immediate component of pos by a <designator> designating kr.

Step 3.

Case 3.1. evst is an <evaluated-read-statement> containing an <evaluated-ignore-option>,eigo.

Let int be the <integer-value> in eigo.

Case 3.2. evst 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(<endfile-condition>,fv)

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(<endfile-condition>,fv).

Step 4.4. If fi contains a <delete-flag> then delete it from fi.

Step 4.5. Decrement int by 1. Go to Step 4.

8.6.6.11 Evaluate-size

Operation: evaluate-size(edd)

where edd is an <evaluated-data-description>.

result: an <integer-value>.

Step 1. Return an implementation-defined <integer-value>, depending on edd.

8.6.6.12 Exit-from-io

Operation: exit-from-io(fv)

where fv is a <file-value>.

Step 1. Let fi be the <file-information> designated by fv. If fi contains an <allocated-buffer>,abuf, containing a <generation>,g, then perform free(g) and delete abuf from fi.

Step 2. Perform trim-io-control.

Step 3. Let eud be the current <executable-unit-designator>. Perform trim-group-control(eud).

Step 4. Replace the immediate component of the current <statement-control> by

```
<operation-list>:
  <operation> for advance-execution
  <operation> for normal-sequence.
```

8.6.6.13 Trim-io-control

Operation: trim-io-control

Step 1. Let bc be the current <block-control>.

Step 2. If bc contains a <data-item-control-list>,dicl, then delete dicl from bc.

Step 3. If bc contains a <current-scalar-item-list>,csil, then delete csil from bc.

Step 4. If bc contains a <string-io-control>,sioc, then delete sioc from bc.

Step 5. If bc contains a <format-control-list>,fcl, then delete fcl from bc.

Step 6. If bc contains a <remote-block-state>,rbs, then delete rbs from bc.

Step 7. If bc contains a <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: execute-get-statement(gs)

where gs is a <get-statement>.

Step 1.

Case 1.1. gs has a <get-file>,gf.

Perform execute-get-file(gf).

Case 1.2. gs has a <get-string>,gstr.

Perform execute-get-string(gstr).

Step 2. Perform trim-io-control.

Step 3. Perform normal-sequence.

8.7.1.2 Execute-get-file

Operation: execute-get-file(gf)

where gf is a <get-file>.

Step 1. Perform Steps 1.1, 1.2, and 1.3 in any order.

Step 1.1. Let fo be the <value-reference> in the <file-option> in gf. Perform evaluate-file-option(fo) to obtain a <file-value>,fv.

Step 1.2. If gf contains a <skip-option>,sko, then let e be the <expression> in sko, and perform evaluate-expression-to-integer(e) to obtain an <integer-value>,sk.

Step 1.3. If gf contains a <copy-option>,co, then let cv be the <value-reference> in co, and perform evaluate-file-option(cv) to obtain a <file-value>,cf.

Step 2. Let fi be the <file-information> designated by fv, and, if a <copy-option> is present in gf, let cfi be the <file-information> designated by cf. If fi contains <open> then go to Step 4.

Step 3. Let efdl be an <evaluated-file-description-list> containing <stream> and <input>. Perform open(fv,efdl) to obtain rf. If rf is <fail> then perform raise-io-condition(<undefinedfile-condition>,fv). If on normal return fi contains <closed> then perform raise-condition(<error-condition>).

Step 4. fi must contain <stream> and <input>.

Step 5. If gf does not have a <copy-option> then go to Step 9. If cfi contains <open> then go to Step 7.

Step 6. Let efdl1 be a <evaluated-file-description-list> containing <stream> and <output>. Perform open(cf,efdl1) to obtain rcf. If rcf is <fail> then perform raise-io-condition(<undefinedfile-condition>,cf). If on normal return fi contains <closed> then perform raise-condition(<error-condition>).

Step 7. cfi must contain <stream> and <output>.

Step 8. Attach a <copy-file>: cf; to the current <block-state>.

Step 9. If gf has a <skip-option> then perform skip(sk,fv).

Step 10. If gf has an <input-specification> then:

Case 10.1. gf has a <list-directed-input>,ldi.

Perform get-list(ldi,fv).

Case 10.2. gf has a <data-directed-input>,ddi.

Perform get-data(ddi,fv).

Case 10.3. gf has an <edit-directed-input>,edi.

Perform get-edit(edi,fv).

Step 11. If gf has a <copy-option> then delete the current <copy-file>.

Step 12. If gf has a <list-directed-input> then terminate this operation. Otherwise set the <first-comma> in fi to contain <on>.

8.7.1.3 Execute-get-string

Operation: execute-get-string(gstr)

where gstr 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 gstr. Perform evaluate-expression(e), to obtain an <aggregate-value>,av. Let sdt 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>, <nonvarying>, and <maximum-length>: <asterisk>. Perform convert(tdt,sdt,sv) to obtain a <character-string-value>,csv. Attach to the current <block-control> a <string-io-control>: csv <first-comma>: <on>.

Step 1.2. If gstr 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-value>,cf. Let cfi be the <file-information> designated by cf. If cfi contains <open> then go to Step 1.2.3.

Step 1.2.2. Let efdl be an <evaluated-file-description-list> containing <stream> and <output>. Perform open(cf,efdl) to obtain a result rcf. If rcf is <fail> then perform raise-io-condition(<undefinedfile-condition>,cf). If on normal return fi contains <closed> then perform raise-condition(<error-condition>).

Step 1.2.3. cfi must contain <stream> and <output>.

Step 1.2.4. Attach a <copy-file>: cf; to the current <block-state>.

Step 2.

Case 2.1. gstr has a <list-directed-input>,ldi.

Perform get-list(ldi).

Case 2.2. gstr has a <data-directed-input>,ddi.

Perform get-data(ddi).

Case 2.3. gstr has an <edit-directed-input>,edi.

Perform get-edit(edi).

Step 3. If gstr has a <copy-option> then delete the current <copy-file>.

8.7.1.4 Get-list

Operation: get-list(ldi,fv)

where ldi 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>:
<data-list-indicator> designating the <input-target-list> of ldi
<data-item-indicator>:
<undefined>.

Step 2. Perform establish-next-data-item to obtain a <current-scalar-item>,ndi or <none>,ndi. If ndi is <none> then terminate this operation.

Step 3. Perform parse-list-input(fv) to obtain a <character-string-value>,csv.

Step 4. If fv is a <file-value> then attach to the current <block-control> a <current-file-value>: 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-quote} [{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-value(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-value(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-value>s 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-io-condition(<conversion-condition>,fv,csv,intg). On normal return let csv be the immediate component of the current <returned-onsource-value>; csv must not contain only blanks; go to Step 4.

Step 5. Let et be the <evaluated-target> in ndi. Let agv be

```
                    <aggregate-value>:
                        <aggregate-type>:
                            <scalar>;
                        <basic-value-list>:
                            <basic-value>:
                                v.
```

Let dd be

```
                    <data-description>:
                        <item-data-description>:
                            <data-type>:
                                <non-computational-type>:
                                    <string>:
                                        <string-type>: st;
                                        <maximum-length>:
                                            <asterisk>;
                                        <nonvarying>.
```

Perform assign(et,agv,dd).

Step 6. If fv is a <file-value> then delete the current <current-file-value>. Go to Step 2.

8.7.1.4.1 Parse-list-input

Operation: parse-list-input(fv)

where fv is a [<file-value>].

result: a <character-string-value>.

Step 1.

Case 1.1. fv is a <file-value>.

Let fi be the <file-information> designated by fv, and let ds be the <dataset> designated by the <dataset-designator> in fi. Let cp be the <current-position> in fi.

Case 1.1.1. There is no <stream-item-list> in ds or cp designates the last <stream-item> in the <stream-item-list> in ds.

Perform raise-io-condition(<endfile-condition>,fv).

Case 1.1.2. (Otherwise).

Let sl be a <stream-item-list> containing the <stream-item>s in the <stream-item-list> in ds, following the <stream-item> designated by cp.

Case 1.2. fv is <absent>.

Let sl be the <character-string-value> in the current <string-io-control>. If sl contains no {symbol}s then perform raise-condition(<error-condition>).

Step 2.

Case 2.1. sl can be parsed as "[{leading-delimiter-list}] , [<stream-item-list>]".

Let ld be that sequence in sl which satisfies "[{leading-delimiter-list}] ," in this parse, and let its number of terminal nodes be ln. Perform input-stream-item(fv) ln times. If the current <first-comma> exists and contains <on> or the <first-comma> in the <file-information>,fi exists and contains <on> then return <character-string-value>: <character-value-list>: <character-value>: {symbol}: {,}. Otherwise set the current <first-comma> (respectively, the <first-comma> in fi) to contain <on>, perform parse-list-input(fv) to obtain csv, and return csv.

Case 2.2. sl can be parsed as "{leading-delimiter-list}".

Let ln be the number of terminal nodes in sl. Perform input-stream-item(fv) ln times. If the current <first-comma> exists and contains <on>, or the <first-comma> in fi exists and contains <on> then return <character-string-value>: <null-character-string>. Otherwise go to Step 1.

Case 2.3. sl can be parsed as "[{leading-delimiter-list}] ' [<string-symbol-or-linemark-list>]".

Let ln be the number of terminal nodes in sl. Perform input-stream-item(fv) ln times. Perform raise-condition(<error-condition>).

Case 2.4. sl can be parsed as "[{leading-delimiter-list}] {putative-list-constant} [[ß|,] [<stream-item-list>]]".

Let ln be the number of terminal nodes in sl preceding that part which satisfies "[<stream-item-list>]" in this parse. Perform input-stream-item(fv) ln times. Let csv be a <character-string-value> whose terminal nodes are those of the part of sl which satisfies "{putative-list-constant} [ß|,]" except those terminals that are <linemark>s. Let fc be the current <first-comma>, if that exists; otherwise let fc be the <first-comma> in fi. If the last {symbol} of csv is a {,} then set fc to contain <on>; otherwise set fc to contain <off>. Delete from csv the last {symbol}, if that {symbol} is a ß or a {,}. Return csv.

8.7.1.4.2 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:

{leading-delimiter}::= ß | <linemark>

{putative-list-constant}::= '[<string-symbol-or-linemark-list>]'
[{non-blank-comma-quote}
[{non-blank-comma-list}]) |
[{data-symbol} | {;} | = }
[{non-blank-comma-list}]

{data-symbol}::= {letter} | {digit} | _ | + | - | (|) | . | : | & | {|} | ¬ |
> | < | / | * | $ | % | {extralingual-character}

Note: The subnodes of {data-symbol} are those of {symbol} except for ß, {,}, {;}, {'} and {=}.

{non-blank-comma-quote}::= {data-symbol} | ; | = | <linemark>

{non-blank-comma}::= {non-blank-comma-quote} | '

<string-symbol-or-linemark>::= {string-or-picture-symbol} | <linemark>

8.7.1.5 Get-data

{data-basic-reference}::= [{data-structure-reference}]
[{%-list}] {identifier} [{%-list}] [{data-subscripts}]

{data-structure-reference}::= {data-basic-reference}.

{data-subscripts}::= ({data-subscript-commalist})[{%-list}]

{data-subscript}::= [{%-list}] [+|-] {integer} [{%-list}]

<scalar-facts>::= <identifier-list> <declaration-designator>
[<data-description>] <integer-value>

Operation: get-data(ddi,fv)

where ddi is a <data-directed-input>,
fv is a [<file-value>].

Step 1. Perform parse-data-input-name(fv) to obtain a <character-string-value>,nf.

Step 2.

Case 2.1. nf contains a <null-character-string> or has just one {symbol} which contains a {,}.

Go to Step 1.

Case 2.2 nf has just one {symbol} which contains a {;}.

Terminate this operation.

Case 2.3. The last {symbol} of nf contains an {=}.

Let nnf be a {symbol-list} containing, in order, all the {symbol}s of nf except its terminal {symbol}: {=}. If nnf is an empty list then let nnf be a <null-character-string>.

Case 2.4. (Otherwise).

Let vf be <character-string-value>: <null-character-string>. If the last {symbol} of nf contains a {;} then let li=1; otherwise let li=0. Go to Step 9.

Step 3. Perform parse-data-input-value(fv) to obtain a <character-string-value>,vf. If vf contains a {symbol} and its last {symbol} contains a {;} then let li=1; otherwise let li=0.

Step 4.

Step 4.1. If nnf conforms to the syntax for {data-basic-reference} then let dbr be a {data-basic-reference} whose terminal nodes are pairwise equal to the terminal nodes of nnf, both sets being taken in order. Otherwise go to Step 9.

Step 4.2. Let vr be a <variable-reference> without subnodes.

Step 4.2.1. Let idl1 be an {identifier-list} containing, in order, the {identifier}s in dbr. Let idl2 be an <identifier-list> without subnodes. For each {identifier},idc, in idl1 let ida be the corresponding <identifier> and append ida to idl2. If idl2 contains more than one element, then let idl3 be a copy of idl2, delete the first element of idl3, and attach idl3 to vr.

Step 4.2.2. If dbr does not contain any ⟨data-subscripts⟩s then let ssl be ⟨absent⟩ and go to Step 4.2.3.

Let dsl be a ⟨data-subscript-list⟩ containing all the ⟨data-subscript⟩s in all the ⟨data-subscripts⟩ in dbr, in order. Let ssl be a ⟨subscript-list⟩ without subnodes.

For each ⟨data-subscript⟩,dss, in dsl, in order, perform Step 4.2.2.1. Attach ssl to vr.

Step 4.2.2.1. Let csdss be the ⟨character-string-value⟩ corresponding to dss. Perform basic-numeric-value(csdss) to obtain a ⟨value-and-type⟩: ⟨real-value⟩,rv ⟨data-type⟩,dt. Let ss be

```
<subscript>:
    <expression>:
        <constant>:
            <basic-value>:
                rv;
            dt;
        <data-description>:
            <item-data-description>:
                dt.
```

Step 4.2.3. Let blo be the ⟨begin-block⟩ or ⟨procedure⟩ that simply contains the ⟨executable-unit⟩ designated by the current ⟨executable-unit-designator⟩.

Step 4.2.3.1. If blo contains a ⟨declaration-list⟩,dl, then go to Step 4.2.3.3.

Step 4.2.3.2. If blo is not contained in a ⟨procedure⟩ then go to Step 9. Let blo be the ⟨begin-block⟩ or ⟨procedure⟩ containing blo which contains no other ⟨begin-block⟩ or ⟨procedure⟩ containing blo. Go to Step 4.2.3.1.

Step 4.2.3.3. Let sfl be a ⟨scalar-facts-list⟩ containing all of the distinct ⟨scalar-facts⟩s which can be obtained by performing Steps 4.2.3.3.1 and 4.2.3.3.2.

Step 4.2.3.3.1. Let d be a ⟨declaration⟩ in dl. Let sfx be a ⟨scalar-facts⟩ containing a ⟨declaration-designator⟩ designating d, an ⟨identifier-list⟩,idl: id; where id is the ⟨identifier⟩ immediately contained in d, and an ⟨integer-value⟩: 0. If d contains ⟨variable⟩ which immediately contains a ⟨data-description⟩,dd, then perform Steps 4.2.3.3.1.1 and 4.2.3.3.1.2.

Step 4.2.3.3.1.1. If dd immediately contains a ⟨dimensioned-data-description⟩,ddd, then let nbp be the number of ⟨bound-pair⟩s in the ⟨bound-pair-list⟩ of ddd, add nbp to the value of the ⟨integer-value⟩ in sfx, replace that ⟨integer-value⟩ with the sum so obtained, and let dd be the ⟨element-data-description⟩ in ddd.

Step 4.2.3.3.1.2. If dd simply contains a ⟨structure-data-description⟩,sdd, which immediately contains an ⟨identifier-list⟩,idlx, then choose an ⟨identifier⟩,idx, from idlx, append idx to idl, let dd be the ⟨data-description⟩ in the ⟨member-description⟩ corresponding to idx in the ⟨member-description-list⟩ of sdd, and go to Step 4.2.3.3.1.1.

Step 4.2.3.3.2. If d contains a ⟨variable⟩ then let idd be the ⟨item-data-description⟩ simply contained in dd and attach a ⟨data-description⟩: idd; to sfx.

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 idl2 is not an ordered sublist of the <identifier-list> of any <scalar-facts> in sfl then go to Step 4.2.3.2. If idl2 is an ordered sublist of the <identifier-list>s of more than one <scalar-facts> in sfl but is not equal to one of them, then go to Step 9. If idl2 is equal to the <identifier-list> of a <scalar-facts>,sfx, then let idl4 be that <identifier-list>. If idl2 is an ordered sublist of the <identifier-list>,idl4, of exactly one <scalar-facts>,sfx in sfl, then idl2 must equal idl4.

If ssl is a <subscript-list> then let nssl be the number of <subscript>s in ssl. Otherwise let nssl be 0. If nssl is not equal to the value of the <integer-value> in sfx then go to Step 9.

Let d be the <declaration> designated by the <declaration-designator> in sfx. d must contain <declaration-type>: <variable>; and must not contain <based> without a subnode. The <data-description>,dd, in sfx must contain an <item-data-description> containing a <data-type>: <computational-type>.

Attach the <declaration-designator> in sfx to vr. Attach the <data-description> in sfx 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 ddi does not contain a <data-target-list> then go to Step 4.3. Let vrl be a <variable-reference-list> consisting of each <variable-reference> in a <data-target> in the <data-target-list> of ddi whose <declaration-designator> equals that of vr and whose <identifier-list>, if present, has fewer elements than idl4. If vrl is empty then go to Step 9. If any <variable-reference> in vrl is without an <identifier-list> then go to Step 4.3. Otherwise, go to Step 9 unless some <variable-reference> in vrl has an <identifier-list> which is an initial sublist of the <identifier-list> obtained by deleting the first <identifier> from idl4.

Step 4.3. Perform evaluate-variable-reference(vr) to obtain a <generation>,g.

Step 5. If fv is a <file-value> then attach <current-file-value>: fv; to the current <block-control>. If vf contains a <null-character-string> then go to Step 8. If vf contains the single {symbol}: {,}; then go to Step 8. If vf contains the single {symbol}: {;}; then go to Step 8. If vf terminates in the {symbol}: {;}; then remove this {symbol}.

Step 6.

Case 6.1. The terminal nodes of vf can be parsed as "{non-blank-comma-quote} [{non-blank-comma-list}]".

Let v be vf. Let st be <character>.

Case 6.2. The terminal nodes of vf can be parsed as "{simple-character-string-constant}".

Perform basic-character-value(vf) to obtain a <character-string-value>,v. Let st be <character>.

Case 6.3. The terminal nodes of vf can be parsed as "{simple-bit-string-constant}".

Perform basic-bit-value(vf) to obtain a <bit-string-value>,v. Let st be <bit>.

Case 6.4. (Otherwise).

Let intg be the smallest integer such that the ⟨character-string-value⟩ of length intg containing the first intg ⟨character-value⟩s 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-io-condition(⟨conversion-condition⟩,fv,vf,intg). On normal return let vf be the immediate component of the current ⟨returned-onsource-value⟩; vf must not contain only blanks; go to Step 6.

Step 7. Let etg be ⟨evaluated-target⟩: g. Let agv be

```
⟨aggregate-value⟩:
     ⟨aggregate-type⟩:
          ⟨scalar⟩;
     ⟨basic-value-list⟩:
          ⟨basic-value⟩:
               v.
```

Let dd be

```
⟨data-description⟩:
     ⟨item-data-description⟩:
          ⟨data-type⟩:
               ⟨non-computational-type⟩:
                    ⟨string⟩:
                         ⟨string-type⟩: st;
                         ⟨maximum-length⟩:
                              ⟨asterisk⟩;
                         ⟨nonvarying⟩.
```

Perform assign(etg,agv,dd).

Step 8. If fv is a ⟨file-value⟩ then remove the current ⟨current-file-value⟩. If li=1 then terminate this operation; otherwise go to Step 1.

Step 9. Perform raise-io-condition(⟨name-condition⟩,fv,str), where str is a ⟨character-string-value⟩ containing, in order, the ⟨symbol⟩s of nf and the ⟨symbol⟩s of vf (excepting a final ⟨symbol⟩: ⟨;⟩;, if any). On normal return if li=0 then go to Step 1; otherwise terminate this operation.

8.7.1.5.1 Parse-data-input-name

Operation: parse-data-input-name(fv)

where fv is a [⟨file-value⟩].

result: a ⟨character-string-value⟩.

Step 1.

Case 1.1. fv is a ⟨file-value⟩.

Let fi be the ⟨file-information⟩ designated by fv, and let ds be the ⟨dataset⟩ designated by the ⟨dataset-designator⟩ in fi. Let cp be the ⟨current-position⟩ in fi.

Case 1.1.1. There is no ⟨stream-item-list⟩ in ds, or cp designates the last ⟨stream-item⟩ in the ⟨stream-item-list⟩ in ds.

Perform raise-io-condition(⟨endfile-condition⟩,fv).

Case 1.1.2. (Otherwise).

Let sl be a ⟨stream-item-list⟩ containing, in the same order, ⟨stream-item⟩s equal to the ⟨stream-item⟩s in the ⟨stream-item-list⟩ in ds which follow the ⟨stream-item⟩ designated by cp.

Case 1.2. fv is <absent>.

Let sl be the <character-string-value> in the current <string-io-control>. If sl contains no {symbol}s then perform raise-condition(<error-condition>).

Step 2.

Case 2.1. sl can be parsed as "[{leading-delimiter-list}] (, | ; | =) [<stream-item-list>]".

Let ld be that sequence in sl which satisties "[{leading-delimiter-list}] (, | ; | =)" in this parse, and let the number of its terminal nodes be ln. Perform input-stream-item(fv) ln times. Let endld be the last terminal node in ld. Return <character-string-value>: <character-value-list>: <character-value>: {symbol}: endld.

Case 2.2. sl can be parsed as "{leading-delimiter-list}".

Let ln be the number of terminal symbols in sl. Perform input-stream-item(fv) ln times. Return <character-string-value>: <null-character-string>.

Case 2.3. sl can be parsed as "[{leading-delimiter-list}] {putative-name-field} [(; | =) [<stream-item-list>]]".

Let ln be the number of terminal nodes in sl before that part which satisfies "[<stream-item-list>]" in this parse. Perform input-stream-item(fv) ln times. Let csv be a <character-string-value> whose terminal nodes are those preceding the part of sl which satisfies "[<stream-item-list>]" in this parse, excluding the part of sl satisfying "[{leading-delimiter-list}]" and those terminals that are <linemark>s. Return csv.

8.7.1.5.2 Parse-data-input-value

Operation: parse-data-input-value(fv)

where fv is a [<file-value>].

result: a <character-string-value>.

Step 1.

Case 1.1. fv is a <file-value>.

Let fi be the <file-information> designated by fv, and let ds be the <dataset> designated by the <dataset-designator> in fi. Let cp be the <current-position> in fi.

Case 1.1.1. There is no <stream-item-list> in ds, or cp designates the last <stream-item> in the <stream-item-list> in ds.

Perform raise-io-condition(<endfile-condition>,fv).

Case 1.1.2. (Otherwise).

Let sl be a <stream-item-list> containing, in the same order, <stream-item>s equal to the <stream-item>s in the <stream-item-list> in ds which follow the <stream-item> designated by cp.

Case 1.2. fv is <absent>.

Let sl be the <character-string-value> in the current <string-io-control>. If sl contains no {symbol}s then perform raise-condition(<error-condition>).

Step 2.

Case 2.1. sl can be parsed as "[{leading-delimiter-list}] {,|;} [<stream-item-list>]".

Let ld be that sequence in sl which satisfies "[{leading-delimiter-list}] {,|;}" in this parse, and let the number of its terminal nodes be ln. Perform input-stream-item(fv) ln times. Return a <character-string-value> containing the last {symbol} in ld.

Case 2.2. sl can be parsed as "{leading-delimiter-list}".

Let ln be the number of terminal nodes in sl. Perform input-stream-item(fv) ln times. Return <character-string-value>: <null-character-string>.

Case 2.3. sl can be parsed as "[{leading-delimiter-list}] ' [{string-or-picture-symbol-list}]".

Let ln be the number of terminal nodes in sl. Perform input-stream-item(fv) ln times. Perform raise-condition(<error-condition>).

Case 2.4. sl can be parsed as "[{leading-delimiter-list}] {putative-data-constant} [[ß|,|;} [<stream-item-list>]]".

Let ln be the number of terminal nodes in sl preceding that part which satisfies "[<stream-item-list>]" in this parse. Perform input-stream-item(fv) ln times. Let csv be a <character-string-value> whose terminal nodes are those of the part of sl which satisfies "{putative-data-constant} [ß|,|;]" except those terminals that are <linemark>s. Delete from csv the last {symbol}, if that {symbol} is a ß or a {,}. Return csv.

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:

```
{putative-name-field}::= [{data-symbol} | ') [{field-element-1-list}]

{field-element-1}::= {data-symbol}|'|,|ß|<linemark>

{putative-data-constant}::= {simple-character-string-constant}
                            [({data-symbol} | <linemark> | =)
                            [{field-element-2-list}]]  |
                            [{data-symbol} | =)
                            [{field-element-2-list}]

{field-element-2}::= {data-symbol}|<linemark>|=|'
```

8.7.1.6 Get-edit

Operation: 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

```
<data-item-control-list>:
    <data-item-control>:
      <data-list-indicator> designating the first <input-target-list>
                of edi
      <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 edi
⟨format-list-index⟩:
⟨integer-value⟩:
0.

Step 3. Perform establish-next-data-item to obtain a ⟨current-scalar-item⟩,ndi: ⟨evaluated-target⟩,et; or ⟨none⟩,ndi. If ndi is ⟨none⟩ then go to Step 6.

Step 4. Perform establish-next-format-item to obtain a ⟨format-item⟩,efi.

Step 5.

Case 5.1. efi contains a ⟨control-format⟩,ecf.

Perform execute-input-control-format(ecf,fv). Go to Step 4.

Case 5.2. efi contains a ⟨data-format⟩,edf.

Perform execute-input-data-format(et,edf,fv). Go to Step 3.

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 ⟨input-target-list⟩ of edi.

Terminate this operation.

Case 6.2. (Otherwise).

Replace dpp 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⟩ of edi
⟨format-list-index⟩:
⟨integer-value⟩:
0.

Go to Step 3.

8.7.1.6.1 Execute-input-control-format

Operation: execute-input-control-format(ecf,fv)

where ecf is a ⟨control-format⟩,
fv is a [⟨file-value⟩].

Case 1. fv is a ⟨file-value⟩.

Step 1.1. Let fi be the ⟨file-information⟩ designated by fv. ecf must not have a ⟨page⟩, ⟨line-format⟩, or ⟨tab-format⟩. Let ds be the ⟨dataset⟩ 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,i), for i=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(fv) to obtain cc. If w = 0 then let w be 1. Let n be the number of {symbol}s in ds which follow the <stream-item> designated by the <current-position> in fi, up to the next <stream-item> which has a <linemark> or <omega>.

Step 1.2.3.2.

Case 1.2.3.2.1. cc < (w-1) and n ≥ (w-cc-1).

Perform input-stream-item(fv) (w-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-cc-1).

Perform skip(<integer-value>:1;,fv). Let n1 be the number of {symbol}s in ds which follow the <stream-item> designated by the <current-position> in fi, up to the next <stream-item> which has a <linemark> or <omega>. If n1 ≥ (w-1) > 0 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,i), for i=1,...,w.

8.7.1.6.2 Execute-input-data-format

Operation: execute-input-data-format(et,edf,fv)

where et is an <evaluated-target>,
edf is a <data-format>,
fv is a [<file-value>].

Step 1.

Case 1.1. edf immediately contains a <real-format> or a <string-format>.

Let w be the first <integer-value> in edf; w must be present.

Case 1.2. edf immediately contains a <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 <complex-format>,ecf.

Step 1.3.1. If the first immediate component of ecf is a <real-format> then let w1 be the first <integer-value> in ecf; otherwise let w1 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 ecf 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 ecf 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 v be a <basic-value> : <character-string-value> : <null-character-string>;; and go to Step 6. Otherwise perform input-stream-item-for-edit(fv,i), for i=1,...,w, to obtain w <stream-item>s, si[i] each containing a {symbol}. If fv is a <file-value> then attach to the current <block-control> a <current-file-value>: fv. Let csv be a <character-string-value> containing, in order, the w <character-value>s cv[i], i=1,...,w, where cv[i] contains the {symbol} si[i], for i=1,...,w.

Step 3. If edf has a <complex-format>,ecf, then perform Steps 3.1 through 3.7.

Step 3.1. If w1=0 then let csv1 be a <character-string-value>: <null-character-string>. Otherwise let csv1 be a <character-string-value> obtained by deleting the last w2 <character-value>s from a copy of csv. If w2=0 then let csv2 be a <character-string-value>: <null-character-string>. Otherwise let csv2 be a <character-string-value> obtained by deleting the first w1 <character-value>s from a copy of csv.

Step 3.2. Let c1 be the first component of ecf. If c1 is a <real-format> then let df1 be its immediate component; otherwise let df1 be c1.

Step 3.3. If ecf has no second component then let df2 be df1. Otherwise let c2 be the second component of ecf; if c2 is a <real-format> then let df2 be its immediate component; otherwise let df2 be c2.

Step 3.4. Perform validate-input-format(df1,csv1) to obtain a <value-and-type>,r1 or <invalid>,r1. If r1 is <invalid> then let intg be its immediate component and perform raise-io-condition(<conversion-condition>,fv,csv1,intg); on normal return let csv1 be the immediate component of the current <returned-onsource-value> and go to Step 3.4.

Step 3.5. Perform validate-input-format(df2,csv2) to obtain a <value-and-type>,r2 or <invalid>,r2. If r2 is <invalid> then let intg be its immediate component and perform raise-io-condition(<conversion-condition>,fv,csv2,intg); on normal return let csv2 be the immediate component of the current <returned-onsource-value> and go to Step 3.5.

Step 3.6. Let v1 and v2 be the first components of r1 and r2, respectively. Let dt1 and dt2 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. dt1 has <fixed>, and <precision>: r s; and dt2 has <fixed>, and <precision>: t u.

adt has <scale>: <fixed>; <precision>: p q; where:

p = min(N,max(r-s,t-u)+max(s,u))
q = max(s,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 dt1 and dt2, respectively. adt has <scale>: <float>; <number-of-digits>: min(N,max(r,t));, where N is the maximum <number-of-digits> for <float> and <decimal>.

Step 3.7. Perform convert(adt,dt1,v1) to obtain a <real-value> containing the <real-number>,cv1, and perform convert(adt,dt2,v2) to obtain a <real-value> containing the <real-number>,cv2. Let bv be a <basic-value> containing a <complex-value>: cv1 cv2. Let edd be an <evaluated-data-description> containing an <item-data-description> containing a <data-type> with <mode>: <complex>; but otherwise as adt. Go to Step 7.

Step 4. If edf 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 be its immediate component and perform raise-io-condition (<conversion-condition>,fv,csv,intg); on normal return let csv be the immediate component of the current <returned-onsource-value> and go to Step 4.2.

Step 4.3. Go to Step 6.

Step 5. If edf 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-io-condition (<conversion-condition>,fv,csv,intg); on normal return let csv be the immediate component fo the current <returned-onsource-value> and go to Step 5.1.

Step 6.

Case 6.1. v is a <character-string-value> or a <bit-string-value>.

Let edd be an <evaluated-data-description> containing an <item-data-description> of <character> or <bit> (respectively) and <maximum-length>: <asterisk>. Let bv be a <basic-value>: v.

Case 6.2. v is a <value-and-type>: val adt.

Let edd be an <evaluated-data-description> containing an <item-data-description> containing adt. Let bv be a <basic-value>: val.

Step 7. Let agv be an

```
<aggregate-value>:
  <aggregate-type>:
       <scalar>;
  <basic-value-list>:
       bv.
```

Let dd be the <data-description> immediate component of edd. Perform assign(et,agv,dd). If fv is a <file-value> then delete the current <current-file-value>.

8.7.1.6.2.1 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 <pictured-numeric>.

Perform validate-numeric-pictured-value(df,csv) to obtain a <picture-validity>,pv. If pv has <picture-invalid>: val; then return <invalid>: val. If pv has <picture-valid>: val; then return a <value-and-type>: val aadt; where aadt is the associated arithmetic data-type of df.

Case 1.2. df contains <pictured-character>.

Perform validate-character-pictured-value(df,csv) to obtain a <picture-validity>,pv. If pv has <picture-invalid>: val; then return <invalid>: val. Otherwise return csv.

Case 2. df is a <character-format>.

Return csv.

Case 3. df is a <bit-format>.

Case 3.1. csv conforms to the syntax "[{Ø-list}] [bs-list] [{Ø-list}]" where bs is one of the {symbol}s in the first column of Table 4.2 corresponding to the <radix-factor> in df.

Let bc be that part of csv which satisfies "[bs-list]". Let bcx be the <character-string-value> obtained by concatenating bc with a {'} on the left, and, on the right, with a 'B1, 'B2, 'B3 or 'B4, according to the <radix-factor> of df. Perform basic-bit-value(bcx) to obtain a <bit-string-value>,bsv. Return bsv.

Case 3.2. (Otherwise).

Let intg be the smallest value such that the first intg {symbol}s in csv 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-value>s 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. csv conforms to the syntax "{Ø-list}" or is a <character-string-value>: <null-character-string>.

Return <value-and-type>: <real-value>: 0; adt; where adt is a <data-type> containing <real>, <fixed>, <decimal>, <number-of-digits>: 1; and <scale-factor>: 0.

Case 4.2.2. csv conforms to the syntax "[{Ø-list}] [+|-] {decimal-number} [{Ø-list}]".

Perform basic-numeric-value(csv) to obtain a <value-and-type>: v dt. If csv contains a {.} then let s'=s; otherwise, let s'=s-d. Return a <value-and-type>: <real-value>: (v*10↑s'); adt; where adt 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 {symbol}s in csv 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-value>s 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. csv conforms to the syntax "{Ø-list}" or is a <character-string-value>: <null-character-string>.

Return <value-and-type>: <real-value>: 0; adt; where adt 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 "[{Ø-list}] [+|-] {decimal-number} [([E][+|-]|E) {integer}] [{Ø-list}]".

If csv contains a sign, but not an E, after a substring conforming to {decimal-number}, then replace that substring with the concatenation of that substring with E. Perform basic-numeric-value(csv) to obtain a <value-and-type>: v dt. If csv contains a {.} then let d'=0; otherwise let d'=d. Return a <value-and-type>: <real-value>: (v*10↑-d'); adt; where adt is a <data-type> equal to dt except that its <scale-factor> (if any) is incremented by d'.

Case 5.2.3. (Otherwise).

Let intg be the smallest value such that the first intg {symbol}s in csv do not have a valid continuation according to the syntax in Case 5.2.2. Return <invalid>: intg.

8.7.1.7 Input-stream-item

Operation: input-stream-item(fv)

where fv is a [<file-value>].

result: a <stream-item> or <omega>.

Step 1.

Case 1.1. fv is a <file-value>.

Step 1.1.1. Let fi be the <file-information> designated by fv; let cp be the <current-position> in fi, and let ds be the <dataset> designated by the <dataset-designator> in fi.

Step 1.1.2.

Case 1.1.2.1. cp designates <alpha>.

Replace the immediate component of cp by a <designator> to the first <stream-item> in ds, or to the <omega> in ds if there are no <stream-item>s in ds.

Case 1.1.2.2. cp designates a <stream-item>.

Replace the immediate component of cp by a <designator> to the next <stream-item> in ds, or to the <omega> in ds if there are no further <stream-item>s in ds.

Case 1.1.2.3. cp designates <omega>.

Go to Step 1.1.3.

Step 1.1.3. Let si be the node designated by cp.

Case 1.2. fv is <absent>.

Let sioc be the current <string-io-control>. Let csv be the <character-string-value> in sioc. If csv contains a <null-character-string> then perform raise-condition(<error-condition>). Otherwise let si be a <stream-item>: v;, 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 cf be its immediate component; otherwise go to Step 3. If si is not <omega> or <linemark> then perform output-string-item(sym,cf), where sym is a <stream-item> containing the {symbol} in si. If si is <linemark> then perform skip(<integer-value>:1;,cf).

Step 3. Return si.

8.7.1.8 Basic-character-value

Operation: basic-character-value(csvin)

where csvin is a ⟨character-string-value⟩.

result: a ⟨character-string-value⟩.

Step 1. The {symbol}s of csvin must conform to the syntax of {simple-character-string-constant}.

Step 2. Let csvo be a copy of csvin. Remove the first and last ⟨character-value⟩s of csvo. If no ⟨character-value⟩s remain then return ⟨character-string-value⟩: ⟨null-character-string⟩.

Replace all adjacent pairs of ⟨character-value⟩: {symbol}: {'}; originally in csvo with a single ⟨character-value⟩: {symbol}: {'}.

Step 3. Return csvo.

8.7.1.9 Basic-bit-value

Operation: basic-bit-value(csvin)

where csvin is a ⟨character-string-value⟩.

result: a ⟨bit-string-value⟩.

Step 1. csvin must conform to the syntax of {simple-bit-string-constant}.

Step 2. Let m be 1, 1, 2, 3, or 4 according as the {radix-factor} in csvin is B, B1, B2, B3, or B4. Remove the portion of csvin corresponding to the syntax for {radix-factor} and remove the first and last of the remaining ⟨character-value⟩s. Let n be the number of ⟨character-value⟩s remaining, if any. If n=0 then return ⟨bit-string-value⟩: ⟨null-bit-string⟩.

Step 3. Let sy(i), for i=1,...,n, be the {symbol}s, in order, of the remaining ⟨character-value⟩s in csvin. Each sy(i) must have an entry in Table 4.2 which is valid for the value of m. Let bsv be a ⟨bit-string-value⟩, containing m*n ⟨bit-value⟩s, such that ⟨bit-value⟩s (i*m+1-m) through (i*m) are obtained from Table 4.2 as a function of m and sy(i), for i=1,...,n.

Step 4. Return bsv.

8.7.1.10 Input-stream-item-for-edit

Operation: input-stream-item-for-edit(fv,i)

where fv is a [⟨file-value⟩],
i is an ⟨integer-value⟩.

result: a ⟨stream-item⟩.

Step 1. Perform input-stream-item(fv) to obtain a ⟨stream-item⟩,si or an ⟨omega⟩,si.

Step 2.

Case 2.1. si is ⟨omega⟩.

If i=1 then perform raise-io-condition(⟨endfile-condition⟩,fv); otherwise perform raise-condition(⟨error-condition⟩).

Case 2.2. (Otherwise).

If si contains ⟨linemark⟩ then go to Step 1. Otherwise return si.

8.7.2 THE PUT STATEMENT

8.7.2.1 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>,pstr.

Perform execute-put-string(pstr).

Step 2. Perform trim-io-control.

Step 3. Perform normal-sequence.

8.7.2.2 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>,sk.

Step 1.3. If pf has a <line-option>,lopt, then let el be the <expression> in lopt, and perform evaluate-expression-to-integer(el), to obtain an <integer-value>,l.

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 efdl be an <evaluated-file-description-list> containing <stream> and <output>. Perform open(fv,efdl) to obtain a result rf. If rf is <fail> then perform raise-io-condition(<undefinedfile-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(l,fv).

Step 7. If pf has a <skip-option> then perform skip(sk,fv).

Step 8. If pf has an <output-specification> then:

Case 8.1. pf has a <list-directed-output>,ldo.

Perform put-list(ldo,fv).

Case 8.2. pf has a <data-directed-output>,ddo.

Perform put-data(ddo,fv).

Case 8.3. pf has an <edit-directed-output>,edo.

Perform put-edit(edo,fv).

8.7.2.3 Execute-put-string

Operation: execute-put-string(pstr)

where pstr is a <put-string>.

Step 1. Let tr be the <target-reference> in pstr. Perform evaluate-target-reference(tr) to obtain an <evaluated-target>,et.

Case 1.1. et has a <generation>.

Let n be the <maximum-length> contained in et.

Case 1.2. et has an <evaluated-pseudo-variable-reference>.

Case 1.2.1. et contains <onchar-pv>.

There must be a current <onchar-value>. Let n=1.

Case 1.2.2. et contains <onsource-pv>.

There must be a current <onsource-value>. Let n be the length of the current <onsource-value>.

Case 1.2.3. et contains <substr-pv>.

If et contains four elements, let n be the fourth of its elements. Otherwise let n be lng-st+1, where lng is the length of the value of the <generation> contained in et and st is the third element of et.

Step 2. Attach to the current <block-control> a

```
<string-io-control>:
    <character-string-value>:
            <null-character-string>;
    <string-limit>:
            <integer-value>:
                n.
```

Step 3.

Case 3.1. pstr has a <list-directed-output>,ldo.

Perform put-list(ldo).

Case 3.2. pstr has a <data-directed-output>,ddo.

Perform put-data(ddo).

Case 3.3. pstr has an <edit-directed-output>,edo.

Perform put-edit(edo).

Step 4. Let os be the <character-string-value> of the current <string-io-control>. Let m be the number of {symbol}s in os. Perform assign(et,agv,dd) where agv is an <aggregate-value> containing os, and dd is a <data-description> containing <character>, <nonvarying>, and <maximum-length> with m.

Step 5. Delete the <string-io-control> from the current <block-control>.

8.7.2.4 Put-list

Operation: put-list(ldo,fv)

where ldo is a <list-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 <output-source-list> of ld
<data-item-indicator>:
<undefined>.

Step 2. If fv is a <file-value> then let fi be the <file-information> designated by fv

Step 3. Let t be an <integer-value>.

Case 3.1. fv is a <file-value> and fi contains <print>.

Step 3.1.1. Perform evaluate-current-column(fv) to obtain cc.

Step 3.1.2. If the <evaluated-tab-option> of fi contains an <integer-value>,iv, suc
that cc=iv-1, then let t=0; otherwise let t=1.

Case 3.2. (Otherwise).

Let t=0.

Step 4. Perform establish-next-data-item to obtain a <current-scalar-item>,ndi o
<none>,ndi. If ndi is <none> then perform output-string-item(<stream-
item>:{symbol}:B;;,fv) and terminate this operation.

Step 5. Let sdt be the <data-type> in ndi, and let tdt be a <data-type> simpl
containing <character>, <nonvarying>, and <maximum-length>: <asterisk>. Perfor
convert(tdt,sdt,bv), where bv is the <basic-value> in ndi, to obtain
<character-string-value>,csv.

Step 6.

Case 6.1. sdt contains <bit>.

Let cv be a <stream-item-list> containing, in order, the {symbol}s

{'}
the {symbol}s of csv, if any,
{'}
B

Case 6.2. sdt contains <character> or <pictured-character>, and either fv is <absent
or fv is a <file-value> and fi does not contain <print>.

Let cv be a <stream-item-list> containing, in order, the {symbol}s

{'}
the {symbol}s of csv, if any, but with each {symbol}: {'}; replaced
by two occurrences of {symbol}: {'};
{'}

Case 6.3. (Otherwise).

Let cv be a <stream-item-list> containing, in order, the {symbol}s of csv.

Step 7.

Case 7.1. fv is a <file-value> and fi contains <print>.

Perform tab(t,fv).

Case 7.2. (Otherwise).

If t=1 then perform output-string(<stream-item-list>: <stream-item>: {symbol}: %;;;,fv).

Step 8. Let t=1.

Step 9.

Case 9.1. fv is a <file-value>.

Perform evaluate-current-column(fv) to obtain ccol. Let lsz be the <evaluated-linesize> in fi. Let lcv be the number of {symbol}s in cv. If lcv > (lsz-ccol) and ccol≠0 then perform skip(<integer-value>:1;,fv). Perform output-string(cv,fv).

Case 9.2. fv is <absent>.

Perform output-string(cv,fv).

Step 10. Go to Step 4.

8.7.2.5 Put-data

Operation: put-data(ddo,fv)

where ddo is a <data-directed-output>,
fv is a [<file-value>].

Step 1.

Case 1.1. ddo contains a <data-source-list>,dsl.

Go to Step 2.

Case 1.2. ddo does not contain a <data-source-list>.

Let dsl be a <data-source-list> which contains an implementation-defined list of trees of the form

<data-source>,t:
<variable-reference>:
<declaration-designator>,dd
[<identifier-list>]
<data-description>,ddesc;;

where dd designates a <declaration> which contains <variable> but does not contain <based> without a subnode and ddesc does not contain a <non-computational-type>.

Step 2. Attach to the current <block-control> a

<data-item-control-list>:
<data-item-control>:
<data-list-indicator> designating dsl
<data-item-indicator>:
<undefined>.

Step 3. If fv is a <file-value> then let fi be the <file-information> designated by fv.

Step 4. Let t be an <integer-value>.

Case 4.1. fv is a <file-value> and fi contains <print>.

Step 4.1.1. Perform evaluate-current-column(fv) to obtain cc.

Step 4.1.2. If the <evaluated-tab-option> of fi contains an <integer-value>,iv, such that cc=iv-1 then let t=0; otherwise let t=1.

Case 4.2. (Otherwise).

Let t=0.

Step 5. Perform establish-next-data-item to obtain a <current-scalar-item>,ndi or <none>,ndi. If ndi is <none> then perform output-string-item(sc,fv) where sc is a <stream-item>: {symbol}: {;};; and terminate this operation. Otherwise ndi is a

<current-scalar-item>:
<basic-value>,bv
<data-type>,sdt
<data-name-field>:
{symbol-list},n.

sdt 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 ndi.

Step 7. Let tdt be a <data-type> simply containing <character>, <nonvarying>, and <maximum-length>: <asterisk>. Perform convert(tdt,sdt,bv) to obtain a <character-string-value>,csv.

Step 8.

Case 8.1. sdt contains <bit>.

Let cv be a <stream-item-list> containing, in order, the {symbol}s

{'}
the {symbol}s of csv, if any,
{'}
B

Case 8.2. sdt contains <character> or <pictured-character>.

Let cv be a <stream-item-list> containing, in order, the {symbol}s

{'}
the {symbol}s of csv, if any, but 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 csv.

Step 9.

Case 9.1. fv is a <file-value> and fi contains <print>.

Perform tab(t,fv).

Case 9.2. (Otherwise).

If t=1 then perform output-string(<stream-item-list>: <stream-item>: {symbol}: b;;;,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 ccol. Let lsz be the <evaluated-linesize> in fi. Let lcv be the number of {symbol}s in cv, and let ln be the number of {symbol}s in n.

Step 11.1.2. If (ln+1) > (lsz-ccol) and ccol ≠ 0 then perform skip(<integer-value>:1;,fv).

Step 11.1.3. Perform output-string(n,fv). Perform output-string-item(<stream-item>: {symbol}: {=};;,fv).

Step 11.1.4. Perform evaluate-current-column(fv) to obtain ccol. If lcv > (lsz-ccol) and ccol ≠ 0 then perform skip(<integer-value>:1;,fv).

Step 11.1.5. Perform output-string(cv,fv).

Case 11.2.fv is <absent>.

Perform output-string(n,fv); perform output-string-item(<stream-item>: {symbol}: {=};;,fv); perform output-string(cv,fv).

Step 12. Go to Step 5.

8.7.2.6 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>:
0.

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 6.

Step 4. Perform establish-next-format-item to obtain a <format-item>,efi.

Step 5.

Case 5.1. efi contains a <control-format>,ecf.

Perform execute-output-control-format(ecf,fv). Go to Step 4.

Case 5.2. efi contains a <data-format>,edf.

Perform execute-output-data-format(ndi,edf,fv). Go to Step 3.

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 <output-source-list> of
                        edo
  <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> of edo;
          <format-list-index>:
               <integer-value>:
                 0.
```

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 <absent> 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-string(str,fv) where str is a <stream-item-list> whose w terminal nodes are all Øs.

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(w,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(fv) 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-string(str,fv), where str is a <stream-item-list> containing (w-cc-1) terminal nodes, each of which is a Ø.

Case 2.6.2.2. cc = w-1.

Terminate this operation.

Case 2.6.2.3. cc > w-1.

Perform skip(<integer-value>:1;,fv). If w > 1 then perform output-string(str,fv), where str is a <stream-item-list> containing (w-1) terminal nodes, each of which is a 'b.

8.7.2.6.2 Execute-output-data-format

Operation: execute-output-data-format(csi,edf,fv)

where csi is a <current-scalar-item>,
edf is a <data-format>,
fv is a [<file-value>].

Case 1. edf contains a <bit-format>,ebf.

Let m be the value of the <radix-factor> in ebf. Perform convert(tdt,sdt,bv) to obtain a <bit-string-value>,bsv, where tdt is a <data-type> containing <bit>, <nonvarying>, and <maximum-length>: <asterisk>;, sdt is the <data-type> of csi, and bv is the <basic-value> immediate component of csi. Let n be the number of immediate components of bsv. Let nn=n. If n is not a multiple of m, then let nn be the next higher multiple of m and insert (nn-n) <bit-value>: <zero-bit>;s at the end of the <bit-value-list> of bsv. Let n=nn. Let k=n/m. If ebf contains an <integer-value>,w, and n > w*m then perform raise-condition(<stringsize-condition>) and let k=w. If ebf has no <integer-value> then let w be k. If w=0 then terminate this operation. Let csv be a <stream-item-list> containing w {symbol}s as follows:

The first k contain, in order, the {symbol}s in Table 4.2 corresponding to successive groups of m <bit-value>s in bsv.
The remaining w-k contain 'bs.

Perform output-string(csv,fv).

Case 2. edf contains a <character-format>,ecf.

Let tdt be a <data-type> containing <character> and <maximum-length>: the <integer-value> in ecf (or <asterisk> if there is no <integer-value> in ecf); let sdt be the <data-type> and v the <basic-value> of csi. Perform convert(tdt,sdt,bv) to obtain a <character-string-value>,csv. Let sil be a <stream-item-list> containing, in order, the {symbol}s of csv. Perform output-string(sil,fv).

Case 3. edf immediately contains a <picture-format> with a <pictured> node, pic.

Let sdt be the <data-type> and bv the <basic-value> of csi. Perform convert(pdt,sdt,bv), where pdt is a <data-type> containing pic, to obtain a <character-string-value>,csv. Let sil be a <stream-item-list> containing, in order, the {symbol}s of csv. Perform output-string(sil,fv).

Case 4. edf contains a <complex-format>,ecf.

Step 4.1. Let sdt be the <data-type> of csi. Let bv be the <basic-value> of csi.

Case 4.1.1. sdt contains <character>, <pictured-character>, or <bit>.

Perform convert-to-arithmetic(sdt,bv) to obtain a <value-and-type> either with components rp and rdt, or with immediate components rp, rdt, ip, and idt. If ip and idt do not exist, let ip be a <real-value>: 0; and let idt be rdt.

Case 4.1.2. sdt contains <real> (including <pictured-numeric>).

Let rp be bv. Let rdt and idt be sdt. 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. sdt contains <complex> (including <pictured-numeric>).

Let rdt and idt be <data-type>s containing <real> but otherwise as sdt. If sdt does not contain <pictured-numeric>, let rp and ip be <basic-value>s which respectively contain the two <real-number>s in csi. If sdt has <pictured-numeric>, let rp and ip be <character-string-value>s containing, respectively, the first n and last n <character-value>s in csi, where n is the associated character-string length of rdt and idt.

Step 4.2. Let c1 be the immediate component of the first immediate component of ecf. If ecf has a second immediate component, sc, then let c2 be the immediate component of sc; otherwise let c2 be c1.

Step 4.3. If c1 is <pictured>, then perform convert(p1,rdt,rp) to obtain a <character-string-value>,csv1, where p1 is a <data-type> containing c1. Otherwise perform edit-numeric-output(rp,rdt,c1) to obtain csv1.

Step 4.4. If c2 is <pictured>, then perform convert(p2,rdt,ip) to obtain a <character-string-value>,csv2, where p2 is a <data-type> containing c2. Otherwise perform edit-numeric-output(ip,idt,c2) to obtain csv2.

Step 4.5. Let sil be a <stream-item-list> containing, in order, the {symbol}s of csv1 and csv2. If sil contains one or more {symbol}s then perform output-string(sil,fv).

Case 5. (Otherwise).

Let fpf be the <fixed-point-format> or <floating-point-format> in edf. Perform edit-numeric-output(bv,dt,fpf) to obtain a <character-string-value>,csv, where bv is the <basic-value> in csi and dt is the <data-type> in csi. Let sil be a <stream-item-list> containing, in order, the {symbol}s of csv. If sil contains one or more {symbol}s then perform output-string(sil,fv).

8.7.2.6.3 Edit-numeric-output

Operation: edit-numeric-output(bv,sdt,fpf)

where bv is a <basic-value>,
sdt 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 cdt be sdt.

Case 1.2. bv has a <complex-value>.

Let cv be a <real-value> containing the real part of bv; let cdt be a <data-type> containing <real> but otherwise as sdt.

Case 1.3. bv has a <character-string-value> or a <bit-string-value>.

Perform convert-to-arithmetic(sdt,v) to obtain a <value-and-type> with first two components cv and dt, where v is the immediate component of bv. Let cdt be a <data-type> containing dt.

Step 2. Let w, d, and s be (respectively) the <integer-value>s 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 s=0. d must not be negative.

Step 3.1.2. Let tdt be a <data-type> containing <real>, <fixed>, <decimal>, <scale-factor>:d. N, used below, is implementation-defined.

Case 3.1.2.1. d = 0 and cv ≥ 0.

w must satisfy w>0. Let n=w, p=min(n,N), i=w-p, and j=p-1. Let pic be '(i)B(j)Z9'.

Case 3.1.2.2. d = 0 and cv < 0.

w must satisfy w>1. Let n=w-1, p=min(n,N), i=w-p-1, and j=p. Let pic be '(i)B(j)-9'.

Case 3.1.2.3. d > 0 and cv ≥ 0.

w must satisfy w>d+1. Let n=w-1, p=min(n,N), i=w-p-1, and j=p-d-1. Let pic be '(i)B(j)Z9V.(d)9'.

Case 3.1.2.4. d > 0 and cv < 0.

w must satisfy w>d+2. Let n=w-2, p=min(n,N), i=w-p-2, and j=p-d. Let pic be '(i)B(j)-9V.(d)9'.

Step 3.1.3. Attach <number-of-digits>: p; to tdt. Remove from pic any repetition factor equal to 0 and the character following it. Let pdt be the <data-type> corresponding to the {picture} with concrete-representation pic, defined above. (For a fuller treatment, see Section 4.4.6). Let x be the <real-value> calculated as x=cv*10↑s+0.5*sign(cv)*10↑-d. Perform convert(tdt,cdt,x) to obtain cv3 and then perform edit-numeric-picture(cv3,pdt) to obtain a <character-string-value>,csv. Return csv.

Case 3.2. fpf is a <floating-point-format>.

Step 3.2.1. If the second and third <integer-value>s are missing from fpf, let s be the converted <number-of-digits> of sdt for a target <data-type> of <real>, <float>, <decimal>, and let d=s-1. If only the third <integer-value> is missing, let s=d+1. d must not be less than 0 and s must not be less than d.

Step 3.2.2. Let tdt be a <data-type> which has <real>, <float>, <decimal>, <number-of-digits>: s. Perform convert-to-float-decimal(tdt,cdt,cv) to obtain a <real-value>,cv3.

Step 3.2.3. There exists a unique representation of cv3 in the form vm*10↑ve, where ve is an integer and either vm=ve=0 or 10↑(s-d-1)≤abs(vm)<10↑(s-d).

Step 3.2.4. If vm≥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>s. Let m=w-i-j-k-d-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; m is the excess of the field-size over the requirements of the specification for the particular value.) If m<0 or abs(ve)≥10↑n, perform raise-condition(<size-condition>).

Step 3.2.5. Let picx be the {picture} with concrete-representation '(m)B(i)-(j)9V(k).(d)9'. Remove from picx any repetition factor whose value is zero and the character following it. Let picm be the <data-type> corresponding to picx. Perform edit-numeric-picture(rvm,picm) to obtain a <character-string-value>,cvm, where rvm is a <real-value> containing vm. Let pice be the <data-type> corresponding to the {picture} with concrete-representation 'S(n)9'. Perform edit-numeric-picture(rve,pice) to obtain a <character-string-value>,cve, where rve is a <real-value> containing ve. Return a <character-string-value> containing the concatenation of cvm, the <character-value>: {symbol}: E;;, and cve.

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 lsz be the <evaluated-linesize> in fi, and psz be the <evaluated-pagesize> in fi, where they exist.

Step 2.2.

Case 2.2.1. fi contains <print>, cc = lsz and cl = psz.

Perform output-stream-item(<stream-item>:<linemark>;,fv). Perform raise-io-condition(<endpage-condition>,fv), and on normal return go to Step 2.1.

Case 2.2.2. cc = lsz, but Case 2.2.1 does not apply.

Perform output-stream-item(<stream-item>: <linemark>;,fv).

Case 2.2.3. (Otherwise).

Go to Step 3.

Step 3. Perform output-stream-item(si,fv).

8.7.2.9 Output-stream-item

Operation: output-stream-item(si,fv)

where si is a ‹stream-item›,
fv is a [‹file-value›].

Case 1. fv is a ‹file-value›.

Let fi be the ‹file-information› designated by fv. fi must contain <stream>. Let ds be the ‹dataset› designated by the ‹dataset-designator› in fi, and let cp be the ‹current-position› in fi.

If si has ‹pagemark› or ‹carriage-return› then fi must contain <print>. Append si to the ‹stream-item-list› in ds. Replace the immediate component of cp by a ‹designator› to the last ‹stream-item› in ds. If ti contains <print> and si is a ‹pagemark› then add 1 to the ‹integer-value› in the ‹page-number› in ti.

Case 2. fv is ‹absent›.

si must contain a {symbol}. Let csv be the ‹character-string-value› in the current ‹string-io-control› and let m be the ‹integer-value› of the ‹string-limit› of the current ‹string-io-control›. It csv contains m {symbol}s or it m=0 then perform raise-condition(<error-condition>).

Let s be the {symbol} in si. It csv contains ‹null-character-string› then replace the immediate component of csv with ‹character-value-list›: ‹character-value›: s. Otherwise, append ‹character-value›: s; to the ‹character-value-list› of csv.

8.7.2.10 Tab

Operation: tab(w,fv)

where w is an ‹integer-value›,
fv is a ‹file-value›.

Step 1. Let eto be the ‹evaluated-tab-option› in the ‹file-information›,fi designated by fv. w must be non-negative. If w=0 then terminate this operation.

Step 2. Perform evaluate-current-column(fv) to obtain an ‹integer-value›,cc.

Step 3.

Case 3.1. There are at least w ‹integer-value›s, t in eto satisfying cc < t-1 < lsz, where lsz is the ‹evaluated-linesize› in ti.

Perform output-tab(fv) w times.

Case 3.2. (Otherwise).

Step 3.2.1. Perform skip(‹integer-value›:1;,fv).

Step 3.2.2. If the first ‹integer-value› in eto is not 1 then perform output-tab(fv).

8.7.2.10.1 Output-tab

Operation: output-tab(fv)

where fv is a ‹file-value›.

Step 1. Perform evaluate-current-column(fv) to obtain an ‹integer-value›,cc.

Step 2. Let tv be the smallest ‹integer-value› in the ‹evaluated-tab-option› of the ‹file-information› designated by fv, which is greater than cc+1.

Step 3. Let bs be a ‹stream-item-list›, the (tv-cc-1) ‹stream-item›s of which all contain ⊠s. Perform output-string(bs,fv).

8.7.2.11 Put-line

Operation: put-line(w,fv)

where w is an ‹integer-value›,
fv is a ‹file-value›.

Step 1. Perform evaluate-current-line(fv) to obtain an ‹integer-value›,cl. Let ep be the ‹evaluated-pagesize› and let cp be the ‹current-position› in the ‹file-information› designated by fv.

Step 2.

Case 2.1. w = cl and the node designated by cp contains a ‹pagemark›, a ‹linemark› or a ‹carriage-return›.

Terminate this operation.

Case 2.2. cl < w ≤ ep.

Perform skip(‹integer-value›:(w-cl);,fv).

Case 2.3. ep ≥ cl ≥ w and Case 2.1 does not apply, or w > ep ≥ cl.

Perform output-stream-item(‹stream-item›:‹linemark›;,fv) (ep-cl+1) times.
Perform raise-io-condition(‹endpage-condition›,fv).

Case 2.4. (Otherwise).

Perform put-page(fv).

8.7.2.12 Put-page

Operation: put-page(fv)

where fv is a ‹file-value›.

Step 1. Perform output-stream-item(‹stream-item›: ‹pagemark›;,fv).

8.7.3 OPERATIONS APPLICABLE TO STREAM I/O

8.7.3.1 Skip

Operation: skip(w,fv)

where w is an ⟨integer-value⟩ which must not be negative,
fv is a ⟨file-value⟩.

Step 1. Let fi be the ⟨file-information⟩ designated by fv.

Step 2.

Case 2.1. The value of w is zero.

Perform output-stream-item(⟨stream-item⟩: ⟨carriage-return⟩;,fv).

Case 2.2. fi contains ⟨input⟩.

Perform repeatedly input-stream-item(fv) until either w ⟨linemark⟩s have been returned or an ⟨omega⟩ is returned. In the latter case perform raise-io-condition(⟨endfile-condition⟩,fv).

Case 2.3. fi contains ⟨output⟩ but not ⟨print⟩.

Perform output-stream-item(⟨stream-item⟩:⟨linemark⟩;,fv) w times.

Case 2.4. fi contains ⟨print⟩, and w ≠ 0.

Perform evaluate-current-line(fv) to obtain an ⟨integer-value⟩,cl. Let ep be the ⟨evaluated-pagesize⟩ in the ⟨file-information⟩ designated by fv. If cl > ep or if ep ≥ cl+w then perform output-stream-item(⟨stream-item⟩: ⟨linemark⟩;,fv) w times. Otherwise perform output-stream-item(⟨stream-item⟩:⟨linemark⟩;,fv) (ep-cl+1) times and then perform raise-io-condition(⟨endpage-condition⟩,fv).

8.7.3.2 Evaluate-current-column

Operation: evaluate-current-column(fv)

where fv is a ⟨file-value⟩.

result: an ⟨integer-value⟩.

Step 1. Let fi be the ⟨file-information⟩ designated by fv, and ds the ⟨dataset⟩ designated by the ⟨dataset-designator⟩ in fi. Let cp be the node in ds designated by the ⟨current-position⟩ in fi.

Step 2.

Case 2.1. cp contains an ⟨alpha⟩, a ⟨linemark⟩, a ⟨pagemark⟩, or a ⟨carriage-return⟩.

Return ⟨integer-value⟩: 0.

Case 2.2. cp contains a {symbol} or an ⟨omega⟩.

Let p be that ⟨alpha⟩, ⟨linemark⟩, ⟨pagemark⟩, or ⟨carriage-return⟩ in ds preceding cp and such that any ⟨stream-item⟩s in ds between p and cp contain {symbol}s. Let n be the number of {symbol}s between p and cp, inclusive. Return ⟨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 ds the ⟨dataset⟩ designated by the ⟨dataset-designator⟩ in fi. Let cp be the node in ds designated by the ⟨current-position⟩ in fi.

Step 2.

Case 2.1. cp contains an ⟨alpha⟩ or a ⟨pagemark⟩.

Return ⟨integer-value⟩: 1.

Case 2.2. (Otherwise).

Let p be that ⟨alpha⟩ or ⟨pagemark⟩ in ds preceding cp such that all ⟨stream-item⟩s in ds between p and cp contain {symbol}s, ⟨linemark⟩s, or ⟨carriage-return⟩s. Let n be the number of ⟨linemark⟩s 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 cdld be the last ⟨data-list-indicator⟩, and let cdid be the last ⟨data-item-indicator⟩.

Step 2.

Case 2.1. cdid is ⟨undefined⟩.

Set cdid to designate the first component of the list designated by cdld.

Case 2.2. cdid designates an immediate component of the list designated by cdld, but not the last component.

Set cdid to designate the next immediate component of the list designated by cdld.

Case 2.3. cdid designates the last immediate component of the list designated by cdld.

If cdid designates a ⟨current-scalar-item⟩ then delete the current ⟨current-scalar-item-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 ⟨true⟩ then go to Step 4. Otherwise, let cdid designate the first immediate component of the list designated by cdld.

Step 3.

Case 3.1. The node designated by cdid immediately contains an <expression>,e.

Perform evaluate-expression(e) to obtain an <aggregate-value>,av, having n <basic-value>s. Let csil be a <current-scalar-item-list> having n elements whose i'th element is a <current-scalar-item> containing the i'th <basic-value> in av, bv[i], and the <data-type> corresponding to bv[i] in the <data-description> immediately contained in e. Attach csil to the current <block-control>. Append to the current <block-control> a

<data-item-control>:
<data-list-indicator> designating csil
<data-item-indicator> designating the first element of csil.

Return the first element of csil.

Case 3.2. The node designated by cdid immediately contains a <target-reference>,tr.

Perform evaluate-target-reference(tr) to obtain an <evaluated-target>,et.

Step 3.2.1.

Case 3.2.1.1. et immediately contains a <generation>,g.

Perform expand-generation(g) to obtain a <generation-list>,ggl, having n elements. Let csil be a <current-scalar-item-list> having n elements whose i'th element is <current-scalar-item>: <evaluated-target>: gg[i];;, where gg[i] is the i'th element of ggl.

Case 3.2.1.2. et immediately contains an <evaluated-pseudo-variable-reference>, epvr.

Case 3.2.1.2.1. epvr contains an <onchar-pv>, <onsource-pv>, <pageno-pv>, <string-pv>, or <unspec-pv>.

Let csil be a <current-scalar-item-list>: <current-scalar-item>: et.

Case 3.2.1.2.2. epvr contains an <imag-pv> or <real-pv>.

Let g be the <generation> in epvr. Perform expand-generation(g) to obtain a <generation-list>,gg. Let n be the number of <generation>s in gg. Let csil be a <current-scalar-item-list> containing n copies of <current-scalar-item>: et. For i=1,...,n, replace the <generation> in the i'th <current-scalar-item> in csil by the i'th <generation> in gg.

Case 3.2.1.2.3. epvr contains a <substr-pv>.

Let g be the <generation> in epvr. Perform expand-generation(g) to obtain a <generation-list>,ggl. Let n be the number of elements in ggl and let gg[i] denote the i'th element in ggl. Let av1 be the first <aggregate-value> in epvr and let eav1[i] be the element of the <basic-value-list> in av1 which corresponds to gg[i]. If epvr has two <aggregate-value>s, then let av2 be the second and let eav2[i] be the element in its <basic-value-list> which corresponds to gg[i]. Let csil be a <current-scalar-item-list> containing n copies of <current-scalar-item>: et. For i=1,...,n, replace the subtrees of the i'th <evaluated-pseudo-variable-reference> in csil as follows:

replace the <generation> by gg[i],
replace the first <aggregate-value> by an <aggregate-value> containing eav1[i],
and, if a second <aggregate-value> is present, replace it by an <aggregate-value> containing eav2[i].

Step 3.2.2. Attach csil 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 csil
<data-item-indicator> designating the first element of csil.

Return the first element of csil.

Case 3.3. The node designated by cdid 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(vr,g) to obtain a <name-and-subscript-list>,nasl. Let edd be the <evaluated-data-description> in g. Perform expand-edd(edd) to obtain an <evaluated-data-description-list>,eddl having n elements. Let csil be a <current-scalar-item-list> having n elements whose i'th element contains:

the j'th <basic-value> in the <allocation-unit> designated by the <allocation-unit-designator> in g, where j is the i'th <storage-index> in g;

the <data-type> from the i'th <evaluated-data-description> of eddl;

a <data-name-field> containing a {symbol-list},dnfsl obtained from nas[i], the i'th <name-and-subscript> of nasl, by performing Step 3.3.1.1.

Step 3.3.1.1. Let dnfsl be the {symbol-list} obtained from the <dot-name-list> of nas[i] by concatenating the {symbol-list}s of the <dot-name>s in their natural order. Delete the final {symbol}: {.} from dnfsl.

If nas[i] has a <comma-subscript-list>,csl, then replace dnfsl by the {symbol-list} obtained by concatenating dnfsl, {symbol-list}: {symbol}: {(};;, and the {symbol-list}s in the <comma-subscript>s of csl in their natural order, and replacing the final {symbol}: {,} by {symbol}: {)}.

Step 3.3.2. Attach csil to the current <block-control>. Append to the current <data-item-control-list> a

<data-item-control>:
<data-list-indicator> designating csil
<data-item-indicator> designating the first element of csil.

Return the first element of csil.

Case 3.4. The node designated by cdid immediately contains a tree of the form <do-spec>,dsp.

Let dld1 be the first immediate component of the node designated by cdid. Let did1 be the first immediate component of dld1. Append to the current <data-item-control-list> a

<data-item-control>:
<data-list-indicator>:
<designator> designating dld1;
<data-item-indicator>:
<designator> designating did1.

Let cdid be the current <data-item-indicator>. Perform establish-controlled-group(dsp) to obtain t. If t is <true> then go to Step 3.

Case 3.5. The node designated by cdid is a <current-scalar-item>,csi.

Return csi.

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 <none>.

8.7.3.4.1 Expand-edd

Operation: expand-edd(edd)

where edd is an ⟨evaluated-data-description⟩.

result: an ⟨evaluated-data-description-list⟩.

Case 1. edd immediately contains a <data-description> immediately containing a <dimensioned-data-description>,ddd.

Let bpl be the <bound-pair-list> in ddd, let lb[i] and ub[i] be respectively the values of the i'th <lower-bound> and <upper-bound> in bpl and let ne be the number of elements in bpl. Let n be the integer

$$\prod_{i=1}^{ne} (ub[i]-lb[i]+1).$$

Let edd1 be an ⟨evaluated-data-description⟩: <data-description>: the immediate subtree of the ⟨element-data-description⟩ in ddd. Perform expand-edd(edd1) to obtain an ⟨evaluated-data-description-list⟩,edd1. Let edd1n be an ⟨evaluated-data-description-list⟩ with, in order, n replications of the subtrees of edd1. Return edd1n.

Case 2. edd immediately contains a <data-description> immediately containing a <structure-data-description>,sdd.

For each <data-description>,dd[i] in sdd perform expand-edd(⟨evaluated-data-description⟩: dd[i]) to obtain an ⟨evaluated-data-description-list⟩,edd1[i]. Let edd1n be an ⟨evaluated-data-description-list⟩ with, in order, the subtrees of these edd1[i]s. Return edd1n.

Case 3. edd immediately contains a <data-description> immediately containing an <item-data-description>.

Return ⟨evaluated-data-description-list⟩: edd.

8.7.3.4.2 Expand-generation

Operation: expand-generation(g)

where g is a ⟨generation⟩.

result: a ⟨generation-list⟩.

Step 1. Let aud be the ⟨allocation-unit-designator⟩ in g. Let edd be the ⟨evaluated-data-description⟩ in g. Let sil be the ⟨storage-index-list⟩ in g. Let n be the number of elements in sil. Perform expand-edd(edd) to obtain an ⟨evaluated-data-description-list⟩,edd1. Let gg be a ⟨generation-list⟩ without subnodes. For i=1,...,n, let si[i] be a copy of the i'th element of sil, let edd[i] be a copy of the i'th element of edd1, and attach to gg a ⟨generation⟩: edd[i] aud ⟨storage-index-list⟩: si[i].

8.7.3.4.3 Make-name-and-subscript-list

<name-and-subscript>::= [<dot-name-list>] [<comma-subscript-list>]

<dot-name>::= {symbol-list}

<comma-subscript>::= {symbol-list}

Operation: make-name-and-subscript-list(vr,g)

where vr is a <variable-reference>,
g is a <generation>.

result: a <name-and-subscript-list>.

Step 1. Let id be the <identifier> in the <declaration> designated by the <declaration-designator> of vr. Perform identifier-to-dotname(id) to obtain a <dot-name>,dn. Let dnl be a <dot-name-list>: dn. If vr has an <identifier-list>,idl, then for each <identifier>,xid in idl, in order, perform identifier-to-dotname(xid) to obtain a <dot-name>,xdn and append xdn to dnl. If vr has a <subscript-list>,ssl, then create a <comma-subscript-list>,icsl from ssl by performing Step 1.1.

Step 1.1. Let icsl be a <comma-subscript-list> without subnodes. Let nss be the number of immediate subtrees of ssl, and let ss[i], i=1,...,nss, be these subtrees. For i=1,...,nss, append to icsl a {symbol-list},sl obtained from ss[i] as follows: if ss[i] is a <subscript>: <asterisk>; then sl is {symbol-list}: {symbol}: {*}. Otherwise, let ex be the <expression> in ss[i], perform evaluate-expression-to-integer(ex) to obtain an <integer-value>,iv, and perform subscript-to-comma-subscript(iv) to obtain a {symbol-list},sl.

Step 2. g contains an <evaluated-data-description>: <data-description>,dd. Let xdd be the immediate subnode of dd. Perform expand-name-and-subscript(xdd) to obtain a <name-and-subscript-list>,nasl1.

Step 3. Perform Step 3.1 for each <name-and-subscript>,xnas, of nasl1.

Step 3.1. If xnas contains a <dot-name-list>,xdnl, then replace xdnl by the result of concatenating a copy of dnl with xdnl. Otherwise attach a copy of dnl to xnas.

Step 4. If vr has a <subscript-list>,ssl, then perform Step 4.1 for each <name-and-subscript>,xnas of nasl1.

Step 4.1. Let cicsl be a copy of icsl. Let nast be the number of trees {symbol-list}: {symbol}: {*};; in cicsl.

Case 4.1.1. nast = 0.

If xnas contains a <comma-subscript-list>,xcsl, then replace xcsl by the result of concatenating cicsl with xcsl; otherwise attach cicsl to xnas.

Case 4.1.2. (Otherwise).

For i=1,...,nast, replace the i'th <comma-subscript>: {symbol-list}: {*};; in cicsl by a copy of the i'th <comma-subscript> in xnas. Delete the first nast <comma-subscript>s from the <comma-subscript-list>,xcsl of xnas. If xcsl is now without subnodes then replace it with cicsl. Otherwise replace xcsl by the result of concatenating cicsl and xcsl.

Step 5. Return nasl1.

8.7.3.4.4 Expand-name-and-subscript

Operation: expand-name-and-subscript(dd)

where dd is an <item-data-description>, or a <dimensioned-data-description>, or a <structure-data-description>.

result: a <name-and-subscript-list>.

Case 1. dd is an <item-data-description>.

Let xnas be a <name-and-subscript> without subnodes. Return <name-and-subscript-list>: xnas.

Case 2. dd is a <dimensioned-data-description>.

Step 2.1. Let xdd be the <structure-data-description> or <item-data-description> in the <element-data-description> in dd. Perform expand-name-and-subscript(xdd) to obtain a <name-and-subscript-list>,nasl1.

Step 2.2. For each <bound-pair>,bp in the <bound-pair-list> of dd, 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-value>s equal to the <integer-value>s in the <lower-bound> and <upper-bound> of bp, respectively. Let n be ub-lb+1. Let nasl2(i), i=1,...,n, be n copies of nasl1. For each i, i=1,...,n, let nasl3 be nasl2(i), let iv be an <integer-value>: lb+i-1;, perform subscript-to-comma-subscript(iv) to obtain a <comma-subscript>,xcs, and perform Step 2.2.1.1.

Step 2.2.1.1. For each <name-and-subscript>,xnas in nasl3 perform Step 2.2.1.1.1.

Step 2.2.1.1.1. If xnas has a <comma-subscript-list>,xcsl, then insert xcs before the first component of xcsl. Otherwise attach to xnas a <comma-subscript-list>: xcs.

Step 2.2.2. Let nasl1 be the <name-and-subscript-list> obtained by concatenating the n nasl2(i) in the order i=1,...,n.

Step 2.3. Return nasl1.

Case 3. dd is a <structure-data-description>.

Step 3.1. Let n be the number of <identifier>s in the <identifier-list>,idl in dd. Let nasl1(i), i=1,...,n, be n <name-and-subscript-list>s without subnodes. For i=1,...,n, let nasl2 be nasl1(i), let id1 be the i'th <identifier> in idl, let dd1 be the <data-description> in the i'th <member-description> in dd, let xdd be the immediate subtree of dd1, and perform Step 3.1.1.

Step 3.1.1. Perform identifier-to-dotname(id1) to obtain a <dot-name>,dn. Perform expand-name-and-subscript(xdd) to obtain a <name-and-subscript-list>,nasl2. Perform Step 3.1.1.1 for each <name-and-subscript>,xnas in nasl2.

Step 3.1.1.1. If xnas has a <dot-name-list>,xdnl, then insert dn before the first subtree of dnl. Otherwise attach to xnas a <dot-name-list>: dn.

Step 3.2. Let nasl3 be the <name-and-subscript-list> obtained by concatenating the n nasl1(i), in the order i=1,...,n. Return nasl3.

8.7.3.4.5 Subscript-to-comma-subscript

Operation: subscript-to-comma-subscript(ss)

where ss is an <integer-value>.

result: a <comma-subscript>.

Step 1. If ss=0 then let n be 1 and let ssd[1] be the {symbol}: {0}. Otherwise let n be the number of digits required for the decimal representation of the absolute value of ss and let ssd[i], i=1,...,n, be the {symbol}s representing the digits of that representation.

Step 2. Let cssl be a {symbol-list} containing, in order, the ssd[i], i=1,...,n, followed by the {symbol}: {,}. If ss<0 then replace cssl by the result of concatenating a {symbol-list}: {symbol}: {-};; with cssl.

Step 3. Return <comma-subscript>: cssl.

8.7.3.4.6 Identifier-to-dotname

Operation: identifier-to-dotname(id)

where id is an <identifier>.

result: a <dot-name>.

Step 1. Let dnsl be a {symbol-list} obtained by concatenating the {symbol-list} of id with a {symbol-list}: {symbol}: {.}.

Step 2. Return <dot-name>: dnsl.

8.7.3.5 Establish-next-format-item

Operation: establish-next-format-item

result: a <format-item>.

Step 1. Let fc be the last <format-control> in the current <format-control-list>. In fc, let flp be the <format-specification-list-designator>, let fli be the immediate subnode of the <format-list-index>, and, where they exist, let fiv be the immediate subnode of the <format-iteration-value>, and let fii be the immediate subnode of the <format-iteration-index>. Let fsl be the <format-specification-list> designated by flp, and let n be the number of immediate components of fsl.

Step 2.

Case 2.1. fli is less than n.

Add 1 to fli, giving a value, m.

Case 2.1.1. The m'th element of fsl is a <format-item>,fi.

Let cfi be a copy of fi. Perform evaluate-format-item(cfi) to obtain a <format-item>,efi. Return efi.

Case 2.1.2. The m'th element of fsl is a <format-iteration>,fi.

Let e be the <expression> in the <format-iteration-factor> immediately contained in fi. 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 flp
    <format-list-index>:
        <integer-value>:
            0;;
    <format-iteration-value>:
        intg;
    <format-iteration-index>:
        <integer-value>:
            1.
```

Go to Step 1.

Case 2.2. fli equals n, and fc 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-control>s in the current <format-control-list>.

Case 2.3.1. fii is less than fiv.

Add 1 to fii, set fli to 0 and go to Step 1.

Case 2.3.2. fii equals fiv.

Delete fc, 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> (but with <integer-value>s 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(fi)

where fi is a <format-item>.

result: a <format-item>.

Case 1. fi contains a <control-format> or a <data-format>.

Let cfi be a copy of fi. 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. fi contains a <remote-format>,rf.

Step 2.1. If there is in the current <format-control-list> a <remote-block-state>, then let rbs be the last such <remote-block-state>. Attach a copy of rbs to the current <block-control>.

Case 2.1.1. rf contains a <variable-reference>,vr.

Perform evaluate-variable-reference(vr) to obtain a ‹generation› whose value (see Section 7.1.3) contains a single ‹format-value›,fv.

Case 2.1.2. rf 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. fv 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 rf immediately contains a <variable-reference> whose <data-type> has <local> then bsd must designate the current ‹block-state›.

Step 2.3. Let fsl designate the <format-specification-list> in the node designated by fsd. Append to the current ‹format-control-list› a

```
‹format-control›:
      <format-specification-list-designator›: fsl;
      ‹format-list-index›:
           ‹integer-value›:
                0;;
      fsd
      ‹remote-block-state›:
            bsd.
```

Step 2.4. Perform establish-next-format-item to obtain a <format-item>,efi. Return efi.

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›,intg. Delete the ‹remote-block-state› from the current ‹block-control›. Return intg.

Case 2. (Otherwise).

Perform evaluate-expression-to-integer(e) to obtain an ‹integer-value›,intg. Return intg.

Chapter 9: Expressions and Conversion

9.0 Introduction

This chapter defines operations that are involved in the evaluation of expressions. Included are definitions for builtin functions and the rules for data conversion. The main Sections are:

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>s and all components of <item-data-description>s, the shapes of the trees for the <data-description> and the «aggregate-type» are identical, and categories correspond as follows:

<data-description>	«aggregate-type»
<dimensioned-data-description>	«dimensioned-aggregate-type»
<element-data-description>	«element-aggregate-type»
<bound-pair-list> and all its components	<bound-pair-list> and equal components if «integer-value»s otherwise <asterisk> components
<structure-data-description>	«structure-aggregate-type»
<member-description>	«member-aggregate-type»
<item-data-description>	«scalar»

In many contexts, properties of <data-description>s are stated in terms of properties of their associated «aggregate-type»s 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».

9.1.1.3 Treatment of Scalars

To facilitate the systematic treatment of expression evaluation and assignment, operations such as evaluate-expression and evaluate-builtin-function-reference are defined so that they return or accept ⟨aggregate-value⟩s 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.

9.1.1.4 Compatibility

In general, the ⟨aggregate-value⟩s involved in operations, such as infix-add, are not required to have equal ⟨aggregate-type⟩s but only to have ⟨aggregate-type⟩s that are compatible. The result of such an operation has an ⟨aggregate-type⟩ that is the common ⟨aggregate-type⟩ of the operand ⟨aggregate-type⟩s. These terms are defined as follows:

Let t[1],...,t[n] be a set of ⟨aggregate-type⟩s. If n=1, the set is compatible, and its common ⟨aggregate-type⟩ is t[1]. If n>2, the set is compatible if and only if t[2],...,t[n] are compatible and t[1] is compatible with the common ⟨aggregate-type⟩,ct, of t[2],...,t[n]. In this case the common ⟨aggregate-type⟩ of t[1],...,t[n] equals the common ⟨aggregate-type⟩ of t[1] and ct. Finally, if n=2, compatibility and the common ⟨aggregate-type⟩ are determined as follows:

Case 1. At least one of t[1], t[2] immediately contains ⟨scalar⟩.

Assume t[1] immediately contains ⟨scalar⟩. Then the set t[1], t[2] is compatible, and its common ⟨aggregate-type⟩ is t[2].

Case 2. Both t[1] and t[2] immediately contain ⟨structure-aggregate-type⟩s.

The set t[1], t[2] is compatible if and only if

(1) t[1] and t[2] have the same number of components, m, in their ⟨member-aggregate-type-list⟩s;

(2) for 1≤i≤m, the i'th ⟨member-aggregate-type⟩ of t[1] is compatible with the i'th ⟨member-aggregate-type⟩ of t[2].

When t[1] and t[2] are compatible, their common ⟨aggregate-type⟩ immediately contains a ⟨structure-aggregate-type⟩ with m ⟨member-aggregate-type⟩s, the i'th ⟨member-aggregate-type⟩ having the common ⟨aggregate-type⟩ of the i'th ⟨member-aggregate-type⟩s of t[1] and t[2].

Case 3. One of t[1], t[2] immediately contains a ⟨dimensioned-aggregate-type⟩ and the other does not.

Assume t[1] immediately contains a ⟨dimensioned-aggregate-type⟩. The set t[1], t[2] is compatible if and only if the ⟨element-aggregate-type⟩ of t[1] is compatible with t[2].

In this case the common ⟨aggregate-type⟩ of t[1] and t[2] immediately contains a ⟨dimensioned-aggregate-type⟩ with a ⟨bound-pair-list⟩ equal to that of t[1] and ⟨element-aggregate-type⟩ containing the same immediate components as the common ⟨aggregate-type⟩ of t[2] and the ⟨element-aggregate-type⟩ of t[1].

Case 4. Both t[1] and t[2] have ⟨dimensioned-aggregate-type⟩s.

The set t[1] and t[2] is compatible if and only if

(1) the ⟨bound-pair-list⟩s of t[1] and t[2] are equal except that where a ⟨bound-pair⟩: ⟨asterisk⟩; occurs in one, the other may have a ⟨bound-pair⟩ containing a ⟨lower-bound⟩ and an ⟨upper-bound⟩;

(2) the ⟨element-aggregate-type⟩s of t[1] and t[2] are compatible.

When t[1] and t[2] are compatible, their common ⟨aggregate-type⟩,ct, immediately contains a ⟨dimensioned-aggregate-type⟩ whose ⟨element-aggregate-type⟩ contains

the same immediate components as the common <aggregate-type> of the <element-aggregate-type>s of t[1] and t[2]. The <bound-pair-list> of ct equals that of t[1] except that, when a <bound-pair> in t[1] has <asterisk>, the corresponding <bound-pair> of t[2] is used.

9.1.1.5 Correspondence

The following two sections extend the notion of node correspondence (see Section 1.3.1.1) to apply to scalar-elements of <aggregate-value>s and <generation>s whose (associated) <aggregate-type>s are compatible, and <data-type>s in <data-description>s whose associated <aggregate-type>s are compatible. The extended notion of correspondence is used extensively in defining aggregate operations and in defining the result <data-description>s of <expression>s.

9.1.1.5.1 Correspondence of Scalar Elements

Let atx and aty be compatible <aggregate-type>s, with number-of-scalar-elements nx and ny, respectively. Let x[1],...,x[nx] and y[1],...,y[ny] be sequences of scalar-elements. (For example, x[1],...,x[nx] might be the scalar-elements of an <aggregate-value> whose <aggregate-type> is atx.) The <aggregate-type>s atx and aty determine a correspondence between the x[1],...,x[nx] and the y[1],...,y[ny] as follows:

Case 1. At least one of atx and aty immediately contains <scalar>.

Assume atx immediately contains <scalar>. Then nx=1, and x[1] corresponds to all of the y[1],y[2],...,y[ny].

Case 2. Both atx and aty immediately contain <structure-aggregate-type>s.

Let p be the number of <member-aggregate-type>s in atx (also aty), and let mx[i] (respectively, my[i]), 1≤i≤p, be the number-of-scalar-elements of the i'th <member-aggregate-type> in atx (respectively, aty). Let sx[i] and sy[i] be 0, when i=1. Let sx[i] be the sum of mx[j], 1≤j<i, where 2≤i≤p, and let sy[i] be the sum of my[j], 1≤j<i, where 2≤i≤p. Then the correspondence between the x[1],..,x[nx] and the y[1],...,y[ny] is such that the x[sx[i]+j], 1≤j≤mx[i], where 1≤i≤p, correspond to the y[sy[i]+k], 1≤k≤my[i], where 1≤i≤p. (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-type>s of the i'th <member-aggregate-type>s of atx and aty.

Case 3. One of atx and aty immediately contains a <dimensioned-aggregate-type> and the other does not.

Assume atx has the <dimensioned-aggregate-type>. Let m be the number-of-scalar-elements of the <element-aggregate-type> of atx. Then the correspondence between the x[1],...,x[nx] and the y[1],...,y[ny] is such that the x[m*i+j], 1≤j≤m, i=0,1,..., correspond to y[1], y[2],...,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 atx and by aty.

Case 4. Both atx and aty have <dimensioned-aggregate-type>s.

Let mx (respectively, my) be the number-of-scalar-elements of atx (respectively, aty). The correspondence between the x[1],...,x[nx] and the y[1],...,y[ny] is such that the x[mx*i+j], 1≤j≤mx, correspond to the y[my*i+k], 1≤k≤my, where i=0,1,... . (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>s of atx and aty.

9.1.1.5.2 Correspondence of Data Types

Let x be an ⟨aggregate-value⟩ or a ⟨generation⟩, and let dd be a ⟨data-description⟩ whose associated ⟨aggregate-type⟩ equals the ⟨aggregate-type⟩ of x. Let x[1], x[2],... be the scalar-elements of x. Then the ⟨data-type⟩s in dd correspond to the x[1], x[2],... as follows:

Case 1. dd immediately contains an ⟨item-data-description⟩.

The single ⟨data-type⟩ simply contained in dd corresponds to the single scalar-element x[1].

Case 2. dd immediately contains a ⟨structure-data-description⟩.

Let the ⟨member-description⟩s simply contained in dd be md[1],...,md[p]. Let n[i], $1 \le i \le p$, be the number-of-scalar-elements of md[i]. Let s[i] be 0 when i=1. Let s[i] be the sum of n[j], $1 \le j < i$ for $2 \le i \le p$. The ⟨data-type⟩s in md[i] correspond to the scalar-elements x[s[i]+j], $1 \le j \le n[i]$, for $1 \le i \le p$, the correspondence being determined by the ⟨aggregate-type⟩ of md[i].

Case 3. dd immediately contains a ⟨dimensioned-data-description⟩.

Let ed be the ⟨element-data-description⟩ simply contained in dd, and let n be the number-of-scalar-elements of the ⟨aggregate-type⟩ of ed. Then for i=0,1,... the ⟨data-type⟩s in ed correspond to the scalar-elements x[n*i+j], $1 \le j \le n$, the correspondence being determined by the ⟨aggregate-type⟩ of ed.

Now the correspondence is easily extended to cover cases where the ⟨aggregate-type⟩s of dd and x are merely compatible. Suppose that all ⟨bound-pair⟩s in dd contain ⟨integer-value⟩s, and let y be any ⟨aggregate-value⟩ whose ⟨aggregate-type⟩ is the same as that of dd. Then a ⟨data-type⟩ in dd 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 dd have ⟨asterisk⟩s, consider instead a ⟨data-description⟩ that is equal to dd except for ⟨bound-pair⟩s and whose associated ⟨aggregate-type⟩ is the same as that of x.

Finally the correspondence can be established between ⟨data-type⟩s of two ⟨data-description⟩s, dd1 and dd2, as follows. Let x be an ⟨aggregate-value⟩ whose ⟨aggregate-type⟩ equals the common ⟨aggregate-type⟩ of dd1 and dd2 except that all ⟨bound-pair⟩s have ⟨lower-bound⟩ equal to 1 and ⟨upper-bound⟩ equal to 1. Then a ⟨data-type⟩ in dd1 corresponds to a ⟨data-type⟩ in dd2 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, bounds 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≥1. To perform f, substitute these Steps into the following macro body immediately after Step 3.2, renumber 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. Renumber 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, op[j+1], j=1,...,(number of operands)-1, taken in any order, perform evaluate-expression(op[j+1]) to obtain an <aggregate-value>,val[j].

Step 2. If two or more of the val[j] have <aggregate-type>s containing corresponding <bound-pair-list>s, then corresponding <bound-pair>s must be equal.

Step 3. Let atp be the common <aggregate-type> of the val[j]. Let nse be the number-of-scalar-elements of atp. For i=1,...,nse, taken in any order, perform Steps 3.1 and 3.2.

> Step 3.1. Let scalar-result-type be that <data-type> of rdd that corresponds to scalar-element, i, in an <aggregate-value> whose <aggregate-type> is atp. For each operand, op[j+1], let the scalar <data-type> of op[j+1] be the corresponding <data-type> of op[j+1], and let the scalar-value of op[j+1] be the corresponding scalar-element in val[j].
>
> Step 3.2. Perform Steps 3.2.1 through 3.2.nstep to obtain scalar-result. Let aval[i] 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 aval[1], aval[2],...,aval[nse].

Note: (1) An operation referencing generate-aggregate-result always has a first operand, rdd, which is a <data-description>. Its other operands will be <expression>s or <argument>s. To refer to the scalar-elements of the operands, the result, and their respective <data-type>s, 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-value, scalar-result, and scalar-result-type are local variables, together with nse, atp, aval[i], and val[j+1].

(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 rdd.

```
DECLARE 1 A(10),
          2 X FLOAT BINARY,
          2 Y CHAR(5),
        B(10) FIXED BINARY,
        C(9)  FLOAT DECIMAL,
        1 D, 2 X, 2 Y, 2 Z;
A = A(1)+D;  A = B+C;  A = B+C(1);
```

Consider the <infix-expression> "A(1)+D" 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 D also immediately contains a <structure-aggregate-type>, but its <member-aggregate-type-list> has three components. Thus the two <aggregate-type>s are not compatible (Section 9.1.1.4, Case 2). Because the two <aggregate-type>s are not compatible, "A(1)+D" does not satisfy the constraints for <infix-expression>s. No <data-description> is defined for "A(1)+D", 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>, t. This immediately contains a <dimensioned-aggregate-type> whose <bound-pair-list> contains a single <asterisk>, and with <element-aggregate-type>: <scalar>. Thus B and C have compatible <aggregate-type>s, and the <aggregate-type> of "B+C" is their common <aggregate-type>, i.e., t. Once the <aggregate-type> of "B+C" is known, determination of the result <data-description>, d, reduces to the determination of the <data-type> components of d (i.e., the result-type) from the corresponding components of the <data-description>s 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 <base> 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-type>s of B and C are checked for compatibility. Because of the general properties of expressions, the <aggregate-type>s, t(B) and t(C), of the values of B and C will be the same as the <aggregate-type>s of the expressions themselves, except that in t(B) and t(C) any <bound-pair> will have a pair of <integer-value>s as its components instead of an <asterisk>. Thus it is really only necessary to check that <bound-pair>s 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-type>s 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 i'th scalar-element of the result is generated by "addition" of b[i] and c. Here the precise result of the "addition" depends on the result-type and the <data-type>s of B and C as well as on the value b[i] and c.

Evaluation of the <variable-reference> "A" yields a <generation>, ga, whose <aggregate-type> immediately contains a <dimensioned-aggregate-type>, t(A), containing a single <bound-pair> with components 1 and 10. The <element-aggregate-type> of t(A) has a <structure-aggregate-type> with two <member-aggregate-type>s, both with <aggregate-type>: <scalar>. Hence ga has 20 scalar-elements, a[1],...,a[20]. The <aggregate-type> of the value of "B+C(1)" is t(B), which is promotable to the <aggregate-type>, t(A) (Section 7.5.3.1). 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 storage designated by a[1] and a[2]. The second element is assigned to a[3] and a[4], etc. The details of the assignment depend on the corresponding <data-type>s in the <data-description> of ga. Assignment of a[1], a[3],... is controlled by the <data-type> of A.X, which has <float> and <binary>; assignment to a[2], a[4],... is controlled by the <data-type> of A.Y, which has <character>.

Example 9.1. An Example of Scalar and Aggregate Types.

9.1.2 INTEGER TYPE

The term integer-type means a <data-type> which has <arithmetic> containing <real> <fixed> <binary> <number-of-digits>: n; and <scale-factor>: 0. n 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(e)

where e is an <expression>.

result: an <integer-value>.

Step 1. Perform evaluate-expression(e) to obtain an <aggregate-value>,av. Let bv be the <basic-value> in av. Let sdt be the <data-type> of e.

Step 2. Let tdt be a <data-type> which is integer-type (Section 9.1.2). Perform convert(tdt,sdt,bv) to obtain a <real-value>,rv.

Step 3. Return an <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 <pictured-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 <pictured-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.

9.1.3.2 Converted Precision

Given a source <data-type> which has <computational-type>, a target <base>, and a target <scale>, the source <data-type> has associated with it a converted <precision>, with a converted <number-of-digits> and a converted <scale-factor>. For a source <data-type> which has <arithmetic> (including <arithmetic> in <pictured-numeric>), the converted <precision>, converted <number-of-digits> and converted <scale-factor> are defined in Table 9.1.

For a source <data-type> which has <string> with <string-type>: <bit>; the converted <precision> is the same as for a source <data-type> which has <arithmetic> with <base>: <binary>;, <scale>: <fixed>;, <scale-factor>: 0; and <number-of-digits>: N;, where N is the maximum <number-of-digits> allowed for <binary> and <fixed>.

For a source <data-type> which has <string> with <string-type>: <character>;, or has <pictured-character>, the converted <precision> is the same as for a source <data-type> which has <arithmetic> with <base>: <decimal>;, <scale>: <fixed>;, <scale-factor>: 0; and <number-of-digits>: N;, where N is the maximum <number-of-digits> allowed for <decimal> and <fixed>.

Table 9.1. Table of Converted Precisions as a Function of Target and Source Attributes.

Target <base> and <scale>	Source <base> and <scale>			
	Binary Fixed	Decimal Fixed	Binary Float	Decimal Float
Binary Fixed	p'=p q'=q	p'=min(ceil (p*3.32) +1,N) q'=ceil(q*3.32)		
Decimal Fixed	p'=min(ceil (p/3.32) +1,N) q'=ceil(q/3.32)	p'=p q'=q		
Binary Float	p'=min(p,N)	p'=min(ceil (p*3.32) ,N)	p'=p	p'=min(ceil (p*3.32),N)
Decimal Float	p'=min(ceil (p/3.32) ,N)	p'=min(p,N)	p'=min(ceil (p/3.32),N)	p'=p

Each table entry shows the <number-of-digits>, p', of the converted <precision>, and the <scale-factor>, q', of the converted <precision> as functions of the <number-of-digits>, p, and <scale-factor>, q, of the source <precision>. Those table entries left blank are for cases which never arise in the language definition. N is the maximum <number-of-digits> allowed for the target <base> and <scale>.

In many cases, evaluation of an expression requires conversion of the values of its subexpressions to target <data-type>s which have <arithmetic> and whose <base>, <scale>, and <mode> are all the same, but whose <precision>s are determined individually using the rules for converted <precision>.

Consider the <infix-expression> "1.0E0 + '10'B". The rules for evaluating <infix-expression>s specify that, before the addition is performed, the values of the two expressions 1.0E0 and '10'B are to be converted to the derived common <base>, <scale>, and <mode> of the expressions. According to Section 9.1.3.1, this means conversion to a target <data-type> that has <binary>, <float>, and <real>. The <precision> of each target <data-type> is the converted <precision>.

For the <constant> "1.0E0", the source <base> has <decimal>, and the source <number-of-digits> is 2. Therefore, according to Table 9.1, the converted <number-of-digits> is 7. For the <bit> <constant> '10'B, the source <base> has <binary>, the source <scale> has <fixed>, and the value p in Table 9.1 is taken to be the maximum <number-of-digits>, N1, allowed for <fixed> and <binary>. Therefore the converted <number-of-digits> is min(N1,N2), where N2 is the maximum <number-of-digits> allowed for <float> and <binary>.

Example 9.2. An Example of Converted Precision.

9.1.3.3 Derived String Type

Let S be a set of one or more <data-type>s which have <computational-type>.

The derived <string-type> of S has <bit> if all the types in S have <string> with <string-type>: <bit>. Otherwise the derived <string-type> has <character>.

When the set S contains at least two <data-type>s, the terminology derived common <string-type> will generally be used.

9.1.3.4 Further Definitions for Character and Bit Strings

In the definitions of operations whose result <data-type> is either <character> or <bit>, the following definitions are used:

(1) The length of a ‹character-string-value› or of a ‹bit-string-value› is zero if it contains a ‹null-character-string› or a ‹null-bit-string›; otherwise it is equal to the number of components in its ‹character-value-list› or ‹bit-value-list›.

(2) A null-string is a ‹character-string-value› containing ‹null-character-string› when the result <string-type> has <character> or a ‹bit-string-value› containing a ‹null-bit-string› when the result <string-type> has <bit>.

(3) A string is a ‹character-string-value› when the result <string-type> has <character>, or a ‹bit-string-value› when the result <string-type> has <bit>.

(4) A substring, c, of a string, st, is a string of the same type as st. It either contains a null-string, or it contains a contiguous set of ‹character-value›s or ‹bit-value›s from st in the same order as in st.

9.1.4 ARITHMETIC RESULTS

In the definitions of the evaluation for many <infix-expression>s and <builtin-function-reference>s 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, m be the <number-of-digits> of rt, n be the <scale-factor> of rt (if rt has <fixed>), and N 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)*floor(v*b↑n), if v ≥ 0,
v' = (b↑-n)*ceil(v*b↑n), if v <0.

If abs(v') ≥ b↑(N-n), then optionally perform raise-condition(<fixedoverflow-condition>); otherwise, return v'.

Case 2. rt has <fixed> and <complex>.

Determine values x' and y' as follows:

x' = (b↑-n)*floor((b↑n)*x), if x ≥ 0,
x' = (b↑-n)*ceil((b↑n)*x), if x < 0,
y' = (b↑-n)*floor((b↑n)*y), if y ≥ 0,
y' = (b↑-n)*ceil((b↑n)*y), if y < 0.

If abs(x')≥b↑(N-n), or if abs(y')≥b↑(N-n), then optionally perform raise-condition(<fixedoverflow-condition>); otherwise, return a <complex-value>: x'+i*y'.

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>s 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↑m, 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'.

9.1.4.1 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(v)

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>.

Optionally perform raise-condition(<fixedoverflow-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 9.1.8 for the case where x is <isub>.)

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. vr immediately contains a <variable-reference>,x.

Perform evaluate-variable-reference(x) to obtain a <generation>,g. Perform value-of-generation(g) to obtain an <aggregate-value>,v. v must not contain <undefined>. Return v.

Case 4. vr 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 ISUBS

Note: There is no operation evaluate-isub. Before an <expression> containing an <isub> is evaluated, the <isub> is replaced by an <expression> without <isub>s. 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(e)

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-f(rdd,x) to obtain v, where f 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 <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-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.

9.2.2.2 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(rdd,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 <bit-string-value>: <null-bit-string>. 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>.

9.2.2.3 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(rdd,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/I. 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: evaluate-infix-expression(e)

where e is an <infix-expression>.

result: an <aggregate-value>.

Step 1. Let x and y be, in order, the <expression>s immediately contained in e. Let rdd be the <data-description> immediately contained in e. Perform infix-f(rdd,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 m, p, and r 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 N 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>.

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 <float>, then m = max(p,r). If the result <scale> has <fixed>, then

```
m = min(N,max(p-q,r-s)+max(q,s)+1)
n = max(q,s).
```

The result <aggregate-type> is the common <aggregate-type> of x and y.

Operation: infix-add(rdd,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>.

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(rdd,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-value>s of both x' and y' are <one-bit>; otherwise the i'th <bit-value> of the result is <zero-bit>.

9.3.2.3 Infix-cat

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 the derived common <string-type> of the corresponding scalar <data-type>s of x and y.

The result <aggregate-type> is the common <aggregate-type> of x and y.

Operation: infix-cat(rdd,x,y)

Perform generate-aggregate-result.

Step 1. Convert the scalar-values of x and y to the derived common <string-type> of the scalar <data-type>s of x and y. Let the converted values be x' and y'.

Step 2. Perform concatenate(x',y') to obtain the scalar-result.

9.3.2.3.1 Concatenation of String Values

Operation: concatenate(s1,s2)

where s1 and s2 have the same <string-type>.

result: a string of the same <string-type> 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 <character-value>s or <bit-value>s of s1, if any, in order, followed by the <character-value>s or <bit-value>s of s2, if any, in order.

9.3.2.4 Infix-divide

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> with the derived common <base>, <scale>, and <mode> of the corresponding scalar <data-type>s of x and y. If the result <scale> has <float>, then m = max(p,r). If the result <scale> has <fixed>, then:

m = N,
n = N-p+q-s.

The result <aggregate-type> is the common <aggregate-type> of x and y.

Operation: infix-divide(rdd,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. If y' = 0, perform raise-condition(<zerodivide-condition>); otherwise perform arithmetic-result(x'/y',scalar-result-type), to obtain the scalar-result.

9.3.2.5 Infix-eq

Constraints: Corresponding scalar <data-type>s 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>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 <bit>.

The result <aggregate-type> is the common <aggregate-type> of x and y.

Operation: infix-eq(rdd,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 <equal>, the scalar-result is a <bit-string-value> containing <one-bit>. Otherwise the scalar-result is a <bit-string-value> containing <zero-bit>.

9.3.2.5.1 Compare

<comparison-result>::= <equal> | <not-equal>

<not-equal>::= [<less-than> | <greater-than>]

Operation: 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-result>.

Case 1. At least one of t[1] and t[2] has <arithmetic> or <pictured-numeric>.

Convert v[1] and v[2] to target scalar <data-type>s which have <arithmetic> with the derived common <base>, <scale>, and <mode> of t[1] and t[2]. The <precision> of the target type for v[i] is the converted <precision> of t[i]. 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 <mode> is <complex>, and v[1]' ≠ v[2]'.

Return <not-equal> without any subnode.

Case 1.3. The derived common <mode> is <real>, and v[1]' < v[2]'.

Return <not-equal>: <less-than>.

Case 1.4. The derived common <mode> is <real>, and v[1]' > v[2]'.

Return <not-equal>: <greater-than>.

Case 2. Each of t[1] and t[2] has <string> or <pictured-character>.

Convert v[1] and v[2] to the derived common <string-type> of t[1] and t[2]. 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 <string-type> with specified length n. Let v[1]' and v[2]' be the final converted values.

Case 2.1. v[1]' = v[2]'.

Return <equal>.

Case 2.2. v[1]' and v[2]' are <bit-string-value>s, and i is the smallest integer such that the i'th <bit-value>s of v[1]' and v[2]' differ.

If the i'th <bit-value> of v[1]' is <zero-bit> (while the i'th <bit-value> of v[2]' is <one-bit>), return <not-equal>: <less-than>. Otherwise return <not-equal>: <greater-than>.

Case 2.3. v[1]' and v[2]' are <character-string-value>s, and i is the smallest integer such that the i'th <character-value> of v[1]' and v[2]' differ.

If the {symbol} in the i'th <character-value> of v[1]' precedes the {symbol} in the i'th <character-value> of v[2]' in the result of performing collate-bif, return <not-equal>: <less-than>. Otherwise return <not-equal>: <greater-than>.

Case 3. t[1] and t[2] both have <non-computational-type>.

If t[i] has <pointer> and t[j] has <offset>, perform convert(t[i],t[j],v[j]) to obtain x, and let v[j]' be a <basic-value>: x, and let v[i]' = v[i]. Otherwise let v[1]' = v[1], and v[2]' = v[2].

Case 3.1. v[1]' and v[2]' do not contain <pointer-value>s or <offset-value>s.

If v[1]' and v[2]' are identical, return <equal>; otherwise return <not-equal> with no subnode.

Case 3.2. v[1]' and v[2]' contain <pointer-value>s.

Step 3.2.1. Any <allocation-unit-designator> contained in v[1]' or v[2]' must designate an <allocation-unit> that exists.

Step 3.2.2. Let edd1 and edd2 be the <evaluated-data-description> of v[1]' and v[2]' respectively.

Step 3.2.3.

Case 3.2.3.1. v[1]' and v[2]' are equal.

Return <equal>.

Case 3.2.3.2. Either v[1]' or v[2]', but not both, has <null>.

Return <not-equal>.

Case 3.2.3.3. The <allocation-unit-designator>s of v[1]' and v[2]' are different and edd1 and edd2 both contain a <data-type> that has neither a <maximum-length> with 0 nor an <area-size> with 0.

Return <not-equal>.

Case 3.2.3.4. (Otherwise).

The <allocation-unit-designator>s of v[1]' and v[2]' must be equal. The first elements, si1[1] and si2[1], respectively, of the <storage-index-list>s of v[1]' and v[2]' must be different. Suppose si1[1] < si2[1]. Let k be such that si1[k] = si2[1], if such a value exists; otherwise let k be m+1, where m is the number of elements in si1. There must exist an n, 1≤n<k, such that the <item-data-description>,idd obtained by performing find-item-data-description(dd,n), where dd is the <data-description> of edd1, has a <data-type> that contains neither a <maximum-length> with 0 nor an <area-size> with 0.

Return <not-equal>.

Case 3.3. v[1]' and v[2]' contain <offset-value>s.

Case 3.3.1. v[1]' and v[2]' are equal, or differ only in their <occupancy> components.

Return <equal>.

Case 3.3.2. v[1]' or v[2]', but not both, has <null>.

Return <not-equal>.

Case 3.3.3. (Otherwise).

The <significant-allocation-list>s in v[1]' and v[2]' each have n1 and n2 components, respectively, and n1 must not equal n2 (say n1 < n2). Let edd1[i], i=1,...,n1, and edd2[i], i=1,...,n2, be respectively the <evaluated-data-description>s in the <significant-allocation-list>s of v[1]' and v[2]'. edd1[i] must equal edd2[i] for i=1,...,n1, and for some j, n1<j≤n2, edd2[j] must have a <data-type> which does not contain a <maximum-length> with 0 nor an <area-size> with 0.

Return <not-equal>.

9.3.2.6 Infix-ge

Constraints: All scalar <data-type>s of x and y must have <computational-type>. Further, each such <data-type> must not have <arithmetic> or <pictured-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: infix-ge(rdd,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>; or <equal>, the scalar-result is a <bit-string-value> containing <one-bit>; otherwise the scalar-result is a <bit-string-value> containing <zero-bit>.

9.3.2.7 Infix-gt

Constraints: All scalar <data-type>s of x and y must have <computational-type>. Further, each such <data-type> must not have <arithmetic> or <pictured-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: infix-gt(rdd,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>.

9.3.2.8 Infix-le

Constraints: All scalar <data-type>s of x and y must have <computational-type>. Further, each such <data-type> must not have <arithmetic> or <pictured-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: infix-le(rdd,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>.

9.3.2.9 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 <pictured-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: infix-lt(rdd,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>; then the scalar-result is a <bit-string-value> containing <one-bit>; otherwise the scalar-result is a <bit-string-value> containing <zero-bit>.

9.3.2.10 Infix-multiply

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> with the derived common <base>, <scale>, and <mode> of the corresponding scalar <data-type>s of x and y. If the result <scale> has <float>, then m = max(p,r). If the result <scale> has <fixed>, then

$$m = \min(N, p+r+1)$$
$$n = q+s.$$

The result <aggregate-type> is the common <aggregate-type> of x and y.

Operation: infix-multiply(rdd,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.

9.3.2.11 Infix-ne

Constraints: Corresponding scalar <data-type>s 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>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 <bit>.

The result <aggregate-type> is the common <aggregate-type> of x and y.

Operation: infix-ne(rdd,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>, 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>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 <bit>.

The result <aggregate-type> is the common <aggregate-type> of x and y.

Operation: infix-or(rdd,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 <zero-bit> if the i'th <bit-value>s 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>. The result <base>, <scale>, <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>: 0; 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

m =(p+1)*y'-1,
n = q*y'.

Case 2. The derived <scale> of y has <fixed>, the derived <mode> of y has <real>, s=0, but Case 1 does not hold.

The result <scale> has <float>, the result <base> and <mode> have the derived <base> and <mode> of x, and m=p.

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 m = max(p,r).

Operation: infix-power(rdd,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↑v; otherwise, perform raise-condition(<error-condition>).

Case 2.2. The result <mode> is <real>, and u = 0.

If v ≤ 0 then perform raise-condition(<error-condition>); otherwise let z = 0.

Case 2.3. The result <mode> is <real>, and u > 0.

Then:

$$z = 1/(u\uparrow -v), \text{ if } v < 0,$$
$$z = 1, \text{ if } v = 0,$$
$$z = u\uparrow v, \text{ if } v > 0.$$

Case 2.4. The result <mode> has <complex>.

Interpret both u and v as <complex-number>s.

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 raise-condition(<error-condition>).

Case 2.4.3. u ≠ 0.

$z = e \uparrow (v*\log(u))$, where log(u) is that value of the complex logarithm function whose imaginary part w is in the interval $-pi < w \leq pi$.

Step 3. Perform arithmetic-result(z,scalar-result-type) to obtain the scalar-result.

9.3.2.14 Infix-subtract

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> with the derived common <base>, <scale>, and <mode> of the corresponding scalar <data-type>s of x and y. If the result <scale> has <float>, then m = max(p,r). If the result <scale> has <fixed>, then

m = min(N,max(p-q,r-s)+max(q,s)+1)
n = max(q,s).

The result <aggregate-type> is the common <aggregate-type> of x and y.

Operation: infix-subtract(rdd,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.

9.4 Builtin-functions

This section describes the <builtin-function>s available in PL/I. Section 9.4.1 gives the general rules for evaluating a <builtin-function-reference>, and Section 9.4.4 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 <max-bif>, is indicated by ellipsis.

```
Arguments:    x,y[,p[,q]]

The letters x, y, p, and q will be used to refer to the <arguments>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 bif be the <builtin-function> immediately contained in bfr. Let x[1], x[2],...,x[n] be the <argument>s, if any, simply contained in bfr.

Step 2.

Case 2.1. bif is <collate-bif>, <date-bif>, <empty-bif>, <null-bif>, <onchar-bif>, <oncode-bif>, <onfield-bif>, <onfile-bif>, <onkey-bif>, <onloc-bif>, <onsource-bif>, or <time-bif>.

Perform f, where f is the operation whose name is the same as that of bif, to obtain an <aggregate-value>,v. Return v.

Case 2.2. (Otherwise).

Let rdd be the <data-description> immediately contained in bfr. Perform f(rdd,x[1],x[2],...,x[n]), where f is the operation whose name is the same as that of bif, 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>.

9.4.2.2 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>,dt;;;, 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 N, 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 <fixed>, 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 BUILTIN-FUNCTION DEFINITIONS

9.4.3.1 Get-established-onvalue

Operation: get-established-onvalue(tp)

where tp is one of «onchar-value», «oncode-value», «onfield-value», «onfile-value», «onkey-value», «onloc-value», or «onsource-value».

result: a «character-string-value», an «integer-value», or «fail».

Step 1. Let bs be the current «block-state».

Step 2.

Case 2.1. bs contains a «condition-bif-value»,cbv, with an immediate component of the type indicated by tp. Let r be the «integer-value» or «character-string-value» component of cbv.

Return r.

Case 2.2. (Otherwise).

Case 2.2.1. There is a «block-state» immediately preceding bs in the «block-state-list».

Let bs 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-function>s 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 <mode> of the corresponding <data-type> of x is <real>.

The <scale> and <precision> of the scalar-result-type are the derived <scale> and converted <precision> of the <data-type>.

Case 2. The derived <mode> 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(N,r+1), and the scalar-result-type <scale-factor> is s.

Operation: abs-bif(rdd,x)

Perform generate-aggregate-result.

Step 1.

Case 1.1. The derived <mode> of the <data-type> of x has <real>.

Convert the scalar-value of x to the scalar-result-type. Let y=|z| where z is the converted value.

Case 1.2. The derived <mode> 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 <mode> is <complex>. Perform conditions-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.

9.4.4.2 Acos-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 <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-bif(rdd,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(rt), where rt is the scalar-result-type. Let w be the arc cosine of y, in radians, such that

$$0 \leq w \leq pi.$$

Step 3. Perform arithmetic-result(w,rt), where rt is the scalar-result-type, to obtain the scalar-result.

9.4.4.3 Add-bif

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-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 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(rdd,x)

Step 1. Let dp be the <declaration-designator> immediately contained in y.

Step 2. If the <declaration>,d, 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>.

Return an <aggregate-value> containing <pointer-value>: <null>.

Case 2.2. (Otherwise).

Perform evaluate-variable-reference(y) to obtain a <generation>,g, which must be connected. Return an <aggregate-value> containing <pointer-value>: g.

9.4.4.5 After-bif

Arguments: sa,ca

Constraints: The <aggregate-type>s of sa and ca must be compatible. All <data-type>s of sa and ca must have <computational-type>.

Attributes: The result <aggregate-type> is the common <aggregate-type> of sa and ca. Each scalar-result-type has <string>. The <string-type> is the derived common <string-type> of corresponding <data-type>s of sa and ca.

Operation: after-bif(rdd,sa,ca)

Perform generate-aggregate-result.

Step 1. In either order, convert the scalar-value of sa to the scalar-result-type to obtain sb and convert the scalar-value of ca 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 cb is a null-string.

The scalar-result is the string sb.

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-value> of the leftmost occurrence of cb in sb. Let j denote the position of the last <bit-value> or <character-value> 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-value> (1≤k≤j-i) is the <bit-value> or <character-value> in position (i+k) in the string sb.

9.4.4.6 Allocation-bif

Arguments: x

Constraints: x must be of the form <argument>: <expression>: <value-reference>: <variable-reference>,y;;;, and y must not contain an <identifier-list> or <subscript-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(rdd,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>,ge.

Let n be the number of <generation>s immediately contained in ge.

Case 2.2. (Otherwise).

Let n be 0.

Step 3. Return an <aggregate-value> containing <real-value>: n.

9.4.4.7 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 <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: asin-bif(rdd,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(rt), where rt is the scalar-result-type. Let w be the arc sine of y, in radians, such that

$$-\pi/2 \leq w \leq \pi/2.$$

Step 3. Perform arithmetic-result(w,rt), where rt is the scalar-result-type, to obtain the scalar-result.

9.4.4.8 Atan-bif

Arguments: y[,x]

Constraints: The <aggregate-type>s 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>s 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 s and convert the scalar-value of x to the scalar-result-type to obtain r. The values of r and s must not both be 0.

Case 1.2. x is <absent>.

Convert the scalar-value of y to the scalar-result-type to obtain s. If the scalar-result-type has <complex>, s must not be ±i.

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(s/r) such that

if $s \geq 0$ then $0 \leq v \leq pi$, and
if $s < 0$ then $-pi < v < 0$.

Case 2.2. x is <absent> and rt has <real>.

Let v be the arc tangent of s, such that

$$-pi/2 < v < pi/2.$$

Case 2.3. rt has <complex>.

Let v be the arc tangent of s, where the real part of the result, w, satisfies

$$-pi < w \leq pi.$$

Step 3. Perform arithmetic-result(v,rt) to obtain the scalar-result.

9.4.4.9 Atand-bif

Arguments: y[,x]

Constraints: The <aggregate-type>s of y and x must be compatible. The derived <mode> of the <data-type>s of y and x, if specified, must have <real>.

Attributes: The result <aggregate-type> is the common <aggregate-type> of y and x. The <mode> of the scalar-result-type has <real>. The <scale> of the scalar-result-type has <float>. The <base> of the scalar-result-type is the derived common <base> of the <data-type>s of y and x. The <precision> of the scalar-result-type is the greater of the converted <precision>s of corresponding <data-type>s of y and x.

Operation: atand-bif(rdd,y[,x])

Perform generate-aggregate-result.

Step 1. Perform atan-bif(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.

9.4.4.10 Atanh-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 are the derived <base>, <mode>, and converted <precision> of the corresponding <data-type> of x.

Operation: atanh-bif(rdd,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 <real>, v must be less than 1 in absolute value. If the scalar-result-type has <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,rt), where rt is the scalar-result-type, to obtain the scalar-result.

9.4.4.11 Before-bif

Arguments: sa,ca

Constraints: The <aggregate-type>s of sa and ca must be compatible. All <data-type>s of sa and ca must have <computational-type>.

Attributes: The result <aggregate-type> is the common <aggregate-type> of sa and ca. The scalar-result-type has <string>, having the derived common <string-type> of the corresponding <data-type>s of sa and ca.

Operation: before-bif(rdd,sa,ca)

Perform generate-aggregate-result.

Step 1. In either order, convert the scalar-value of sa to the scalar-result-type to obtain sb and convert the scalar-value of ca 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 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 sb.

The scalar-result is the string sb.

Case 2.4. The string cb is a substring of sb.

Let i denote the position of the first <bit-value> or <character-value> of the leftmost occurrence of cb in sb. If i=1, then the scalar-result is a null-string. If i > 1, then the scalar-result is a string of length (i-1) whose j'th <bit-value> or <character-value> ($1 \leq j \leq i-1$) is the j'th <bit-value> or <character-value> in the string sb.

9.4.4.12 Binary-bif

Arguments: x[,p[,q]]

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(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.

9.4.4.13 Bit-bif

Arguments: x[,le]

Constraints: All <data-type>s of x and le must have <computational-type>. le 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(rdd,x[,le])

Perform generate-aggregate-result.

Step 1.

Case 1.1. le is <absent>.

Convert the scalar-value of x to <bit> to obtain the scalar-result.

Case 1.2. le is not <absent>.

Convert le 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,ca

Constraints: The <aggregate-type>s of x and y must be compatible. All <data-type>s of x, y, and ca must have <computational-type>. The <aggregate-type> of ca 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(rdd,x,y,ca)

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 le be the greater of the lengths of ra and rb. In either order, convert ra to <bit> of length le to obtain sa, and convert rb to <bit> of length le to obtain sb.

Step 2. Convert ca to <bit> of length 4 to obtain d. Let the <bit-value>s within d be named d[1], d[2], d[3], d[4], from left-to-right, respectively.

Step 3.

Case 3.1. le = 0.

The scalar-result is a null-string.

Case 3.2. le > 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-value>s of sa and sb as shown in Table 9.2.

Table 9.2. Table of Scalar-results as a Function of <bit-value>s for Bool-bif.

i'th <bit-value> of		
sa	sb	scalar-result
0	0	d[1]
0	1	d[2]
1	0	d[3]
1	1	d[4]

In this example BOOL is used to perform an exclusive-or operation on the first two arguments. The value '1001'B is the result of the evaluation of:

```
BOOL('1100'B,'0101'B,'0110'B)
```

Example 9.4. An Example of the BOOL bif.

9.4.4.15 Ceil-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>. 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 value of x. The <precision> of the scalar-result-type has:

<number-of-digits>: min(N,max(r-s+1,1));
<scale-factor>: 0.

Operation: ceil-bif(rdd,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(rt), where rt 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,rt), where rt is the scalar-result-type, to obtain the scalar-result.

9.4.4.16 Character-bif

Arguments: sa[,le]

Constraints: All <data-type>s of sa must have <computational-type>. le 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(rdd,sa[,le])

Perform generate-aggregate-result.

Case 1. le is <absent>.

Convert the scalar-value of sa to <character> to obtain the scalar-result.

Case 2. le is not <absent>.

Convert le 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.

9.4.4.17 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} just once in an implementation-defined order known as the collating sequence. (See Section 9.3.2.5.1 for use of this sequence in the comparison of <character-string-value>s.)

Step 2. Return an <aggregate-value> containing v.

9.4.4.18 Complex-bif

Arguments: x,y

Constraints: The <aggregate-type>s of x and y must be compatible. The derived common <mode> of the <data-type>s 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>s of x and y. Let r and s denote the converted <number-of-digits> and <scale-factor> of the <data-type> of x, let t and u 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 <scale> has <fixed>.

$$p = \min(N, \max(r-s, t-u) + \max(s,u))$$
$$q = \max(s,u).$$

Case 2. The scalar-result-type <scale> has <float>.

$$p = \max(r,t).$$

Operation: complex-bif(rdd,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 <mode>: <real>; 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 Conjg-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 <complex>. The scalar-result-type has the derived <base>, <scale>, and converted <precision> of the corresponding <data-type> of x.

Operation: conjg-bif(rdd,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.

9.4.4.20 Copy-bif

Arguments: sa,le

Constraints: All <data-type>s of sa and le must have <computational-type>. le must have <aggregate-type>: <scalar>.

Attributes: The result <aggregate-type> is the <aggregate-type> of sa. The scalar-result-type has the derived <string-type> of the corresponding <data-type> of sa.

Operation: copy-bif(rdd,sa,le)

Perform generate-aggregate-result.

Step 1. Perform Steps 1.1. and 1.2. in either order.

Step 1.1. Convert le to integer-type. The result, n, must not be negative.

Step 1.2. Convert the scalar-value of sa 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 concatenate(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-value> '121212'.

Example 9.5. An Example of the COPY bif.

9.4.4.21 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(rdd,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-in-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.

9.4.4.22 Cosd-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 <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(rdd,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(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.

9.4.4.23 Cosh-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 are the derived <base>, <mode>, and converted <precision> of the corresponding <data-type> of x.

Operation: cosh-bif(rdd,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(rt), 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.

9.4.4.24 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 6. It represents the date in the form "yymmdd" 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.

9.4.4.25 Decat-bif

Arguments: sa,ca,pa

Constraints: All <data-type>s 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 <bit> 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 cb.

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.

sb[1] and sb[2] are null-strings and sb[3] is the string sb.

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).

sb[1] is the string sb and sb[2] and sb[3] are null-strings.

Step 3. For i=1,2,3, let t[i] be s[i] if the i'th <bit-value> of pb has <one-bit>, otherwise let t[i] be a null-string. Perform concatenate(t[1],t[2]) to obtain v. Perform concatenate(v,t[3]) to obtain the scalar-result.

The value of DECAT('XYZ','Y','001'B) is the <character-string-value> 'Z'. Note that this result is the same as the value of AFTER('XYZ', 'Y'). The value of DECAT('XYZ','Y','011'B) is the <character-string-value> 'YZ'

Example 9.6. An Example of the DECAT bif.

9.4.4.26 Decimal-bif

Arguments: x[,p[,q]]

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 <decimal>. 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: decimal-bif(rdd,x[,p[,q]])

Perform generate-aggregate-result.

Step 1. Convert the 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.

9.4.4.27 Dimension-bif

Arguments: x,n

Constraints: The <data-type> of n must have <computational-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: dimension-bif(rdd,x,n)

Step 1. Let y be the <variable-reference> in x. Let dp be the <declaration-designator> immediately contained in y.

Step 2. Perform evaluate-expression-to-integer(n) 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 <generation>,g. Let k and i be the <lower-bound> and <upper-bound> respectively of the j'th <bound-pair> in the <bound-pair-list> of g's <evaluated-data-description>. Return an <aggregate-value> containing <real-value>: (i-k+1).

```
DECLARE A(10,2:6) CONTROLLED;
ALLOCATE A;
N = DIM(A,2);

The value of N is 5 after the execution of the statements shown.
```

Example 9.7. An Example of the DIM bif.

9.4.4.28 Divide-bif

Arguments: x,y,p[,q]

Constraints: The <aggregate-type>s of the <argument>s x and y must be compatible. All <data-type>s of x and y must have <computational-type>. Constraints for 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 in x and y. The <precision> of the scalar-result-type is determined as defined in Section 9.4.2.2.

Operation: divide-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 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. If the converted scalar-value of y is zero, perform raise-condition(<zerodivide-condition>).

Step 3.

Case 3.1. The derived <mode> of the <data-type>s of x and y has <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 <mode>s of the <data-type>s of x and y has <complex>.

Perform conditions-in-arithmetic-expression(rt), where rt is the scalar-result-type. Let v be the converted scalar-value of x divided by the converted scalar-value of y.

Step 4. Perform arithmetic-result(v,rt), where rt is the scalar-result-type, to obtain the scalar-result.

9.4.4.29 Dot-bif

Arguments: x,y[,p[,q]]

Constraints: x and y must both be of the form <argument>: <expression>: <data-description>: <dimensioned-data-description>: <element-data-description>: <item-data-description>: <bound-pair-list>,bpl;;;;;; 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 then the result-type has <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,y[,p[,q]])

Step 1. In either order, perform evaluate-expression(x) and evaluate-expression(y) to obtain <aggregate-value>s, 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 udt and vdt 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 udt and the scalar-elements of v to vdt to obtain ut[1],ut[2],...,ut[n] and vt[1],vt[2],...,vt[n]. Let

$$w = \sum_{i=1}^{n} ut[i] * vt[i].$$

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 z. Return an <aggregate-value> containing z.

9.4.4.30 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>.

9.4.4.31 Erf-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 <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: erf-bif(rdd,x)

Perform 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(rt), where rt is the scalar-result-type. Let w be

$$(2/\pi) * \int_0^y e^{-t^2} \, dt.$$

Step 3. Perform arithmetic-result(w,rt), where rt is the scalar-result-type, to obtain the scalar-result.

9.4.4.32 Erfc-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 <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: erfc-bif(rdd,x)

Perform 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(rt), where rt is the scalar-result-type. Let w be

$$1-(2/\pi) * \int_0^y e^{-t^2} \, dt.$$

Step 3. Perform arithmetic-result(w,rt), where rt is the scalar-result-type, to obtain the scalar-result.

9.4.4.33 Every-bif

Arguments: x

Constraints: All <data-type>s of x must have <computational-type>.

Attributes: The result <aggregate-type> immediately contains <scalar>. The result <data-type> has <bit>.

Operation: every-bif(rdd,x)

Step 1. Perform evaluate-expression(x) to obtain an <aggregate-value>,u.

Step 2. In any order, convert each scalar-element of u to <bit>, to obtain v.

Step 3.

Case 3.1. Every scalar-element of v that does not contain <null-bit-string> has a <bit-string-value> with every <bit-value> containing <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.

9.4.4.34 Exp-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 are the derived <base>, <mode>, and converted <precision> of the corresponding <data-type> of x.

Operation: exp-bif(rdd,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(rt), where rt is the scalar-result-type. Let v be e↑w, where e is the base of the natural logarithm system.

Step 3. Perform arithmetic-result(v,rt) to obtain the scalar-result.

9.4.4.35 Fixed-bif

Arguments: x,p[,q]

Constraints: All <data-type>s of x must have <computational-type>. Constraints on p and q are given 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 <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: 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.

9.4.4.36 Float-bif

Arguments: x,p

Constraints: All <data-type>s 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: 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.

9.4.4.37 Floor-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>. 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>: min(N,max(r-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 <mode>, <base>, <scale>, and converted <precision>.

Step 2. Perform conditions-in-arithmetic-expression(rt), where rt is the scalar-result-type. Let v be the greatest integer less than or equal to 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.

9.4.4.38 Hbound-bif

Arguments: x,n

Constraints: The <data-type> of n must have <computational-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: hbound-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(n) 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 <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.

9.4.4.39 High-bif

Arguments: le

Constraints: The <data-type> of le must have <computational-type>. le must have <aggregate-type>: <scalar>.

Attributes: The result <aggregate-type> immediately contains <scalar>. The result <data-type> has <character>.

Operation: high-bif(rdd,le)

Step 1. Perform evaluate-expression-to-integer(le) to obtain n. The value of n must not be negative.

Step 2.

Case 2.1. n ≠ 0.

Let v be a <character-string-value> of length n containing n occurrences of the last <character-value> in the result of performing collate-bif.

Case 2.2. n = 0.

Let v be a <character-string-value>: <null-character-string>.

Step 3. Return an <aggregate-value> containing v.

9.4.4.40 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(rdd,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(v,rt), where rt is the scalar-result-type, to obtain the scalar-result.

9.4.4.41 Index-bif

Arguments: sa,ca

Constraints: The <aggregate-type>s of sa and ca must be compatible. All <data-type>s of sa and ca must have <computational-type>.

Attributes: The result <aggregate-type> is the common <aggregate-type> of sa and ca. The scalar-result-type is integer-type.

Operation: index-bif(rdd,sa,ca)

Perform generate-aggregate-result.

Step 1. In either order, convert the scalar-values of sa and ca to their derived common <string-type>. Let sb be the converted scalar-value of sa, and cb be the converted scalar-value of ca.

Step 2.

Case 2.1. cb is not a substring of sb or the length of either sb or cb is 0.

The scalar-result is 0.

Case 2.2. cb is a substring of sb.

Let j denote the position of the first <bit-value> or <character-value> of the leftmost occurrence of cb in sb. The scalar-result is j.

```
The integer 3 is the result of evaluating:  INDEX('ABCDABCD','CD')
```

Example 9.8 An Example of the INDEX bif.

9.4.4.42 Lbound-bif

Arguments: x,n

Constraints: The <data-type> of n must have <computational-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: lbound-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(n) 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 <generation>,g. Let k be the <lower-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.

9.4.4.43 Length-bif

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-bif(rdd,sa)

Perform generate-aggregate-result.

Step 1. Convert the scalar-value of sa to the derived <string-type> of sa to obtain sb.

Step 2. The scalar-result is the length of sb.

9.4.4.44 Lineno-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: lineno-bif(rdd,fn)

Step 1. Perform evaluate-expression(fn) to obtain a «file-value»,fv. The «file-information»,fi, designated by fv must contain «open». The «evaluated-file-description» of fi must contain <print>.

Step 2. Perform evaluate-current-line(fv) to obtain an «integer-value»,iv. Return an «aggregate-value» containing iv.

9.4.4.45 Log-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 are the derived <base>, <mode>, and converted <precision> of the corresponding <data-type> of x.

Operation: log-bif(rdd,x)

Perform generate-aggregate-result.

Step 1. Convert the scalar-value of x to the scalar-result-type to obtain u. If the scalar-result-type has <mode>: <real>; then the value of u must be greater than 0. If the scalar-result-type has <mode>: <complex>; then the value of u must not equal 0.

Step 2. Perform conditions-in-arithmetic-expression(rt), where rt is the scalar-result-type. Let v be the natural logarithm of u, 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.

9.4.4.46 Log10-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: log10-bif(rdd,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(rt), where rt is the scalar-result-type. Let v be the common logarithm, i.e. base 10, of u.

Step 3. Perform arithmetic-result(v,rt), where rt is the scalar-result-type, to obtain the scalar-result.

9.4.4.47 Log2-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: log2-bif(rdd,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(rt), where rt is the scalar-result-type. Let v be the base 2 logarithm of u.

Step 3. Perform arithmetic-result(v,rt), where rt is the scalar-result-type, to obtain the scalar-result.

9.4.4.48 Low-bif

Arguments: le

Constraints: The <data-type> of le must have <computational-type>. le must have <aggregate-type>: <scalar>.

Attributes: The result <aggregate-type> immediately contains <scalar>. The result <data-type> has <character>.

Operation: low-bif(rdd,le)

Step 1. Perform evaluate-expression-to-integer(le) to obtain n. The value of n must not be negative.

Step 2.

Case 2.1. n ≠ 0.

Let v be a <character-string-value> of length n containing n occurrences of the first <character-value> in the result of performing collate-bif.

Case 2.2. n = 0.

Let v be a <character-string-value>: <null-character-string>.

Step 3. Return an <aggregate-value> containing v.

9.4.4.49 Max-bif

Arguments: x[1],x[2],...,x[n]

Constraints: n, the number of arguments, must be at least 1. The <aggregate-type>s of the <argument>s must be compatible. The derived common <mode> of the <data-type>s of the <argument>s must have <real>.

Attributes: The result <aggregate-type> is the common <aggregate-type> of the <argument>s. The <base> and <scale> of the scalar-result-type are the derived common <base> and <scale> of the corresponding <data-type>s of the <argument>s, and its <mode> is <real>.

Case 1. The derived common <scale> of the corresponding <data-type>s of the <argument>s has <float>.

The <precision> of the scalar-result-type is max(p[1],p[2],...,p[n]), where p[1],p[2],...,p[n] are the converted <number-of-digits> of the <data-type>s of x[1],x[2],...,x[n].

Case 2. The derived common <scale> of the corresponding <data-type>s of the <argument>s has <fixed>.

Let (p[1],q[1]), (p[2],q[2]),...,(p[n],q[n]) be the converted <number-of-digits> and <scale-factor> of the <data-type>s of x[1],x[2],...,x[n] respectively. Then the <precision> of the scalar-result-type is given by:

<number-of-digits>:min(N,max(p[1]-q[1],p[2]-q[2],...,p[n]-q[n])+max(q[1],q[2],...,q[n]));
<scale-factor>: max(q[1],q[2],...,q[n]).

Operation: max-bif(rdd,x[1],x[2],...,x[n])

Perform generate-aggregate-result.

Step 1. In any order, convert the scalar-values of x[1],...,x[n] to the scalar-result-type. Let the converted scalar-values be u[1],u[2],...,u[n].

Step 2. Let v be max(u[1],u[2],...,u[n]).

Step 3. Perform arithmetic-result(v,rt), where rt is the scalar-result-type, to obtain the scalar-result.

9.4.4.50 Min-bif

Arguments: x[1],x[2],...,x[n]

Constraints: n, the number of arguments, must be at least 1. The <aggregate-type>s of the <argument>s must be compatible. The derived common <mode> of the <data-type>s of the <argument> must have <real>.

Attributes: The result <aggregate-type> is the common <aggregate-type> of the <argument>s. The <base> and <scale> of the scalar-result-type are the derived common <base> and <scale> of the corresponding <data-type>s of the <argument>s, and its <mode> is <real>.

Case 1. The derived common <scale> of corresponding <data-type>s of the <argument>s has <float>.

The <precision> of the scalar-result-type is max(p[1],p[2],...,p[n]), where p[1],p[2],...,p[n] are the converted <number-of-digits> of the <data-type>s of x[1],x[2],...,x[n].

Case 2. The derived common <scale> of corresponding <data-type>s of the <argument>s has <fixed>.

Let (p[1],q[1]),(p[2],q[2]),...,(p[n],q[n]) be the converted <number-of-digits> and <scale-factor> of the <data-type>s of x[1],x[2],...,x[n] respectively. Then the <precision> of the scalar-result-type is given by:

<number-of-digits>:min(N,max(p[1]-q[1],p[2]-q[2],...,p[n]-q[n])+max(q[1],q[2],...,q[n]));
<scale-factor>: max(q[1],q[2],...,q[n]).

Operation: min-bif(rdd,x[1],x[2],...,x[n])

Perform generate-aggregate-result.

Step 1. In any order, convert the scalar-values of x[1],...,x[n] to the scalar-result-type. Let the converted scalar-values be u[1],u[2],...,u[n].

Step 2. Let v be min(u[1],u[2],...,u[n]).

Step 3. Perform arithmetic-result(v,rt), where rt is the scalar-result-type, to obtain the scalar-result.

9.4.4.51 Mod-bif

Arguments: x,y

Constraints: The <aggregate-type>s of x and y must be compatible. All derived <data-type>s of x and y must have <real>.

Attributes: The result <aggregate-type> is the common <aggregate-type> of x and y. The <base> and <scale> of the scalar-result-type are the derived common <base> and <scale> of the corresponding <data-type>s of x and y, and its <mode> is <real>.

Case 1. The <scale> of the scalar-result-type has <float>.

The <precision> of the scalar-result-type is the greater of the converted <precision>s of the corresponding <data-type>s of x and y.

Case 2. The <scale> of the scalar-result-type has <fixed>.

Let (p[1],q[1]) and (p[2],q[2]) be the respective converted <number-of-digits> and <scale-factor> of the corresponding <data-type>s of x and y. Then the <precision> of the scalar-result-type is given by:

<number-of-digits>: min(N,p[2]-q[2]+max(q[1],q[2]));
<scale-factor>: max(q[1],q[2]).

Operation: mod-bif(rdd,x,y)

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 the converted scalar-value of x be u and the converted scalar-value of y be w.

Case 2.1. w ≠ 0.

Perform conditions-in-arithmetic-expression(rt), where rt is the scalar-result-type. Let v be given by:

v=u-w*floor(u/w).

Case 2.2. w = 0.

Let v=u.

Step 3. Perform arithmetic-result(v,rt), where rt is the scalar-result-type, to obtain the scalar-result.

9.4.4.52 Multiply-bif

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 given 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: multiply-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 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.

Case 2.1. The scalar-result-type has <real>.

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 <complex>.

Perform conditions-in-arithmetic-expression(rt), where rt 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 arithmetic-result(v,rt), where rt 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 resul <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>. ar must have <aggregate-type> <scalar>. The <data-type> of ar must have <area>. ar must have the for <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 subnode.

Operation: offset-bif(rdd,pt,ar)

Perform generate-aggregate-result.

Step 1. Let odt 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>,odt to obtain the scalar-result.

9.4.4.55 Onchar-bif

Arguments: (none)

Attributes: The result <aggregate-type> immediately contains <scalar>. The resul <data-type> has <character>.

Operation: onchar-bif

Step 1. Perform get-established-onvalue(<onchar-value>) to obtain j.

Case 1.1. j is <fail> or j=0.

Let v be a <character-value> containing ⊮.

Case 1.2. j > 0.

Perform get-established-onvalue(<onsource-value>) to obtain a <character string-value>,sa. Let v be the j'th <character-value> in sa.

Step 2. Return an <aggregate-value> containing <character-string-value>: <character value-list>: v.

9.4.4.56 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-onvalue(<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.

9.4.4.57 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-onvalue(<onfield-value>) to obtain sa.

Case 1.1. sa is <fail>.

Let v be a <character-string-value>: <null-character-string>.

Case 1.2. (Otherwise).

Let v be sa.

Step 2. Return an <aggregate-value> containing v.

9.4.4.58 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-onvalue(<onfile-value>) to obtain sa.

Case 1.1. sa is <fail>.

Let v be a <character-string-value>: <null-character-string>.

Case 1.2. (Otherwise).

Let v be sa.

Step 2. Return an <aggregate-value> containing v.

9.4.4.59 Onkey-bif

Arguments: (none)

Attributes: The result <aggregate-type> immediately contains <scalar>. The result <data-type> has <character>.

Operation: onkey-bif

Step 1. Perform get-established-onvalue(<onkey-value>) to obtain sa.

Case 1.1. sa is <fail>.

Let v be a <character-string-value>: <null-character-string>.

Case 1.2. (Otherwise).

Let v be sa.

Step 2. Return an <aggregate-value> containing v.

9.4.4.60 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 sa.

Case 1.1. sa is <fail>.

Let v be a <character-string-value>: <null-character-string>.

Case 1.2. (Otherwise).

Let v be sa.

Step 2. Return an <aggregate-value> containing v.

9.4.4.61 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 sa.

Case 1.1. sa is <fail>.

Let v be a <character-string-value>: <null-character-string>.

Case 1.2. (Otherwise).

Let v be sa.

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(fn) to obtain a <file-value>,fv. The <file-information>,fi, designated by fv must contain <open>. The <evaluated-file-description> of fi must contain <print>.

Step 2. Let v be the <integer-value> in <page-number> in fi.

Step 3. Return an <aggregate-value> containing <real-value>: v.

9.4.4.63 Pointer-bif

Arguments: ofe,ar

Constraints: All <data-type>s of ofe 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 ofe. The scalar-result-type has <pointer>.

Operation: pointer-bif(rdd,ofe,ar)

Perform generate-aggregate-result.

Step 1. Perform Steps 1.1 and 1.2 in either order.

Step 1.1. Perform evaluate-expression(ofe) to obtain ofs.

Step 1.2. Let odt be a <data-type> containing <offset>: vr; where vr is the <variable-reference> in ar.

Step 2. Perform convert(pdt,odt,ofs) to obtain the scalar-result, where pdt is a <data-type> containing <pointer>.

9.4.4.64 Precision-bif

Arguments: x,p[,q]

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> 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. The <precision> of the scalar-result-type is determined as defined in Section 9.4.2.2.

Operation: precision-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.

9.4.4.65 Prod-bif

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 <aggregate-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.

Operation: prod-bif(rdd,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 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> of rdd, to obtain z. Return an <aggregate-value> containing z.

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(rdd,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: sa

Constraints: Each <data-type> of sa must have <computational-type>.

Attributes: The result <aggregate-type> is the <aggregate-type> of sa. The scalar-result-type has the derived <string-type> of the corresponding <data-type> of sa.

Operation: reverse-bif(rdd,sa)

Perform generate-aggregate-result.

Step 1. Convert the scalar-value of sa 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+1)'th <bit-value> or <character-value> in sb.

9.4.4.68 Round-bif

Arguments: x,n

Constraints: All <data-type>s of x must have <computational-type>. n must have <constant> 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 r and s be the converted <number-of-digits> and <scale-factor> of the corresponding <data-type> of x.

The <precision> of the scalar-result-type is given by:

<number-of-digits>: max(1,min(r-s+1+n,N));
<scale-factor>: n.

Case 2. The derived <scale> and <mode> of the <data-type> of x are <real> and <float>.

The <number-of-digits> in the scalar result-type is given by min(n,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: round-bif(rdd,x,n)

Perform generate-aggregate-result.

Step 1. Convert the scalar-value of x to its derived <mode>, <base>, <scale>, and converted <precision>. Let v 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 v have <real> and <fixed>.

Perform conditions-in-arithmetic-expression(rt), where rt is the scalar-result-type. Let w be given by:

$$w=sign(v)*(b\uparrow-n)*floor(abs(v)*(b\uparrow n)+1/2).$$

Case 2.2. The <scale> and <mode> of the <data-type> of v have <real> and <float>.

Let c be the unique integer such that

$$b\uparrow(c-1) \le abs(v) < b\uparrow c.$$

Perform conditions-in-arithmetic-expression(rt), where rt is the scalar-result-type. Let w be an implementation-defined value that satisfies the inequality:

$$abs(w-v) \le (b\uparrow(c-n))/2.$$

Case 2.3. The <mode> of the <data-type> of v 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 value 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.

9.4.4.69 Sign-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 scalar-result-type is integer-type.

Operation: sign-bif(rdd,x)

Perform generate-aggregate-result.

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.

9.4.4.70 Sin-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 are the derived <base>, <mode>, and converted <precision> of the corresponding <data-type> of x.

Operation: sin-bif(rdd,x)

Perform generate-aggregate-result.

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 conditions-in-arithmetic-expression(rt), where rt is the scalar-result-type. Let v be the sine 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.

9.4.4.71 Sind-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: sind-bif(rdd,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(rt), where rt is the scalar-result-type. Let v be the sine 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.

9.4.4.72 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 <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: sinh-bif(rdd,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(rt), where rt is the scalar-result-type. Let v be the hyperbolic sine of w.

Step 3. Perform arithmetic-result(v,rt) to obtain the scalar-result.

9.4.4.73 Some-bif

Arguments: x

Constraints: All <data-type>s of x must have <computational-type>.

Attributes: The result <aggregate-type> immediately contains <scalar>. The result <data-type> has <bit>.

Operation: some-bif(rdd,x)

Step 1. Perform evaluate-expression(x) to obtain an <aggregate-value>,u.

Step 2. In any order, convert each scalar-element of u 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.

9.4.4.74 Sqrt-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 are the derived <base>, <mode>, and converted <precision> of the corresponding <data-type> of x.

Operation: sqrt-bif(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(rt), where rt 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(rt), where rt 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 u 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.

9.4.4.75 String-bif

Arguments: sa

Constraints: All <data-type>s of sa 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 sa.

Operation: string-bif(rdd,sa)

Step 1. Perform evaluate-expression(sa) to obtain an <aggregate-value>,v.

Step 2. In any order, convert all the scalar-elements in v to the derived common <string-type> of all the <data-type>s of sa. Let the converted values be cv[i] 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 concatenate(r,cv[i]) to obtain r.

Step 4. Return an <aggregate-value> containing r.

9.4.4.76 Substr-bif

Arguments: sa,st[,le]

Constraints: All <data-type>s of sa, st and le must have <computational-type>. The <aggregate-type>s of sa, st and le must be compatible.

Attributes: The result <aggregate-type> is the common <aggregate-type> of sa, st and le. Each scalar-result-type has <string> with the derived <string-type> of the corresponding <data-type> of sa.

Operation: substr-bif(rdd,sa,st[,le])

Perform generate-aggregate-result.

Step 1. Perform Steps 1.1 to 1.3 in any order.

Step 1.1. Convert the scalar-value of sa 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 i.

Step 1.3. If le is not <absent>, convert its scalar-value to integer-type to obtain j.

Step 2. If le is <absent>, let j=k-i+1.

Step 3. Test the inequalities:

$$1 \le i \le k + 1$$
$$0 \le j \le k - i + 1.$$

If these inequalities are not satisfied, perform raise-condition(<stringrange-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> (1≤n≤j) is the (i+n-1)'th <bit-value> or <character-value> in the string sb, n=1,...,j.

9.4.4.77 Subtract-bif

Arguments: x,y,p[,q]

Constraints: The <aggregate-type>s of x and y must be compatible. Each <data-type> 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: 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 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 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.

9.4.4.78 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 <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>: N; 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(x) to obtain an <aggregate-value>,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 z. Return an <aggregate-value> containing z.

9.4.4.79 Tan-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 are the derived <base>, <mode>, and converted <precision> of the corresponding <data-type> of x.

Operation: tan-bif(rdd,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(rt), where rt 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.

9.4.4.80 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(rdd,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(rt), where rt 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.

9.4.4.81 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 <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: tanh-bif(rdd,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(rt), where rt is the scalar-result-type. Let v be the hyperbolic tangent of w.

Step 3. Perform arithmetic-result(v,rt) to obtain the scalar-result.

9.4.4.82 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+t, where t is an implementation-defined non-negative integer. v represents the time-of-day on a 24-hour scale, in the form hhmmss[d...], where each letter represents one <character-value> and each contained {symbol} is a {digit}. hh,mm,ss are respectively in the ranges 0C: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.

9.4.4.83 Translate-bif

Arguments: sa,ra[,pa]

Constraints: The <aggregate-type>s 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: translate-bif(rdd,sa,ra[,pa])

Perform 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 m 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 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≤i≤m, let s[i] be the i'th <character-value> in sb and let t[i] be the i'th <character-value> in the scalar-result. The string pb is searched for the leftmost occurrence of s[i]. If s[i] is not found in pb, then let t[i] be s[i]. If s[i] occurs in pb, let j be the ordinal of the occurrence in pb and let t[i] be the j'th <character-value> in the string rb. The scalar-result is t.

```
The result of evaluating:

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(N,max(r-s+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(rt), where rt 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,rt), where rt is the scalar-result-type, to obtain the scalar-result.

9.4.4.85 Unspec-bif

Arguments: x

Constraints: x must be of the form <argument>: <expression>: <value-reference>: <variable-reference>: <data-description>: <item-data-description>.

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>,g.

Step 3. Return an <aggregate-value> containing a <bit-string-value>,v, which depends on the properties of g in an implementation-defined manner. This value may be <undefined>.

9.4.4.86 Valid-bif

Arguments: sa

Constraints: All <data-type>s of sa must have <pictured>.

Attributes: The result <aggregate-type> is the <aggregate-type> of sa. The scalar-result-type has <bit>.

Operation: valid-bif(rdd,sa)

Perform generate-aggregate-result.

Step 1. Let sv be the scalar-value of sa; let dt be the scalar <data-type> corresponding to sa. Perform validate-numeric-pictured-value(dt,sv) or validate-character-pictured-value(dt,sv) according as dt has <pictured-numeric> or <pictured-character>, 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>: <one-bit>.

Case 2.2. pv has <picture-invalid>.

The scalar-result is a <bit-string-value>: <bit-value-list>: <bit-value>: <zero-bit>.

9.4.4.87 Verify-bif

Arguments: sa,ca

Constraints: The <aggregate-type>s of sa and ca must be compatible. All <data-type>s of sa and ca must have <computational-type>.

Attributes: The result <aggregate-type> is the common <aggregate-type> of sa and ca. The scalar-result-type is integer-type.

Operation: verify-bif(rdd,sa,ca)

Perform generate-aggregate-result.

Step 1. In either order, convert the scalar-value of sa to <character> to obtain sb and convert the scalar-value of ca to <character> to obtain cb. Let m be the length of sb.

Step 2. Each <character-value> of sb, sb[i], (1 ≤ i ≤ m), is compared in turn with the <character-value>s 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, st 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>s, 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 9.5.1.1.).

Operation: convert(tt,st,sval)

where tt is a <data-type>,
st is a <data-type>,
sval 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 an <offset-value>,sv,
or a <basic-value> containing an immediate component, sv,
of one of the above categories,
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 <fixed> but not <pictured-numeric>, and sv is a <real-value>.

Perform convert-to-fixed(tt,st,sv) to obtain cv. Return cv.

Case 2. 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 3. tt has <complex> but not <pictured-numeric>, and sv is a <real-value>.

Let rtt be a <data-type> with <mode>: <real>; but which is otherwise the same as tt. Perform convert(rtt,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 rtt and rst be <data-type>s 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,rst,x) to obtain x' and convert(rtt,rst,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 rst be a <data-type> with <mode>: <real>; but otherwise the same as st. Perform convert(tt,rst,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-arithmetic(st,sv) to obtain a <value-and-type>: rv rdt; or a <value-and-type>: rv rdt iv idt.

Step 6.2.

Case 6.2.1. tt has <real>.

Perform convert(tt,rdt,rv) to obtain v. Return v.

Case 6.2.2. tt has <complex>.

If iv and idt do not exist, let iv be a <real-value>: 0; and idt 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,idt,iv) 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 9.5.2) 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-arithmetic(st,sv) to obtain a <value-and-type> whose first two components are cv and ct.

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,ct,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-arithmetic(st,sv) to obtain a <value-and-type>: rv rdt; or a <value-and-type>: rv rdt iv idt. If iv and idt do not exist, let iv be a <real-value>: 0; and let idt be rdt.

Case 7.3.3.1.2. sv is a <complex-value>.

Let rv and iv be the real and imaginary parts of sv; let rdt and idt be <data-type>s which have <real> but are otherwise as st.

Case 7.3.3.1.3. sv is a <real-value>.

Let rv be sv; let iv be a <real-value>: 0; let rdt and idt be st.

Step 7.3.3.2. Let ratt be a <data-type> which has <real> but is otherwise as att. Perform convert-to-float-decimal(ratt,rdt,rv) to obtain rv1; perform convert-to-float-decimal(ratt,idt,iv) to obtain iv1. Let v be a <complex-value> with real part: rv1; and imaginary part: iv1.

Step 7.4. Perform edit-numeric-picture(v,tt) to obtain a <character-string-value>, csv.

Return csv.

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-bit(st,sv) to obtain s; otherwise, perform convert-to-character(st,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 m be the length of s. If m>n then perform raise-condition(<stringsize-condition>).

Step 8.4.

Case 8.4.1. n = 0.

Return a null-string.

Case 8.4.2. m ≥ 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. m < n.

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 1≤i≤m, and whose remaining <bit-value>s or <character-value>s have, respectively, <zero-bit>s or Øs.

Case 9. tt has <pictured-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 <nonvarying> and whose <maximum-length> contains the associated character-string length of tt. Perform convert(ct,st,sv) to obtain s.

Step 9.2. Perform validate-character-pictured-value(tt,s) to obtain a <picture-validity>,pv. If pv has <picture-valid>, return s.

Step 9.3. Let cbifs be a <condition-bif-value-list> containing <onsource-value>: s; and <onchar-value>: n;, where n is the <integer-value> in pv. Perform raise-condition(<conversion-condition>,cbifs). Let s be the immediate component of the current <returned-onsource-value>, and go to Step 9.2.

Case 10. tt has <pointer>, st has <offset>: <variable-reference>,vr; 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(vr) to obtain a <generation>,g.

Step 10.2. Let edd, sal, and sil be, respectively, the ‹evaluated-data-description›, ‹significant-allocation-list›, and ‹storage-index-list› components of sv. The value of the ‹generation›,g (see Section 7.1.3), has an ‹area-value›,av. av must not immediately contain ‹empty› and there must be an immediate component, aa, of the ‹area-allocation-list› of av such that the ‹significant-allocation-list› of aa is identical with sal. Let aud be an ‹allocation-unit-designator› that designates aa's ‹allocation-unit›. Return a ‹pointer-value›: ‹generation›: edd aud sil.

Case 11. tt has ‹offset›: ‹variable-reference›,vr;, st has ‹pointer› and sv is a ‹pointer-value›.

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-value›: ‹null›.

Step 11.1.2. Perform evaluate-variable-reference(vr) to obtain a ‹generation›,g.

Step 11.2. Let edd, aud, and sil be, respectively, the ‹evaluated-data-description›, ‹allocation-unit-designator›, and ‹storage-index-list› components of sv. The value of the ‹generation›,g (see Section 7.1.3) has an ‹area-value›,av. av must not immediately contain ‹empty› and aud must designate an ‹allocation-unit› that is an immediate component of one of the elements, aa, of the ‹area-allocation-list› of av. Let sal be the ‹significant-allocation-list› of aa. Return an ‹offset-value›: edd sal sil.

9.5.1.1 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(x), the source-type is by implication the scalar <data-type> of x that corresponds to the ‹basic-value›,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 <nonvarying> is explicitly specified.

(2) When <base>, <scale>, and <mode> components are specified, then tt has <arithmetic> and not <pictured-numeric> unless <pictured-numeric> is explicitly specified.

9.5.1.2 Convert-to-fixed

Operation: convert-to-fixed(tt,st,sv)

where tt is a <data-type> that has <real> and <fixed>,
st is 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↑-q)*sign(sv)*floor((b↑q)*abs(sv)).

Step 3.

Case 3.1. st has <fixed> and <scale-factor>: 0;, or st has <fixed> and the same <base> as tt.

Let v = cv.

Case 3.2. (Otherwise).

Let v be an implementation-dependent approximation to cv such that v=w*b↑-q for some integer w.

Step 4.

Case 4.1. abs(v) ≥ b↑(p-q).

Perform raise-condition(<size-condition>).

Case 4.2. (Otherwise).

Return a <real-value>: v.

9.5.1.3 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 b=2 or b=10, 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-1)≤abs(sv)<b↑e. Let v be a value such that abs(v-sv)≤b↑(e-p). 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 <pictured-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 v=abs(x), where x is the real part of sv. Let ct be a <data-type> with <mode>: <real>; but otherwise as st.

Case 1.1.3. st has <pictured-numeric>, and sv is a <character-string-value>.

Perform validate-numeric-pictured-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 ct. According to the <base> and <scale> of ct, determine p as follows:

```
<binary><fixed>   : p = min(N,max(r-s,0))
<decimal><fixed>  : p = min(N,max(ceil(3.32*(r-s)),0))
<binary><float>   : p = min(N,r)
<decimal><float>  : p = min(N,ceil(3.32*r)).
```

N 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>: 0. Perform convert(tt,ct,v) to obtain a non-negative integer, n.

Step 1.4. If n≥2↑p, then perform raise-condition(<size-condition>); otherwise n can be exactly represented as the sum of c[i]*2↑(p-i) for i=1 to p where each c[i] 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 c[i] is 0 or 1.

Case 2. st has <bit>, and sv is a <bit-string-value>.

Return sv.

Case 3. st has <character> or <pictured-character>, and sv is a <character-string-value>.

Step 3.1. If sv contains a <character-value> that has neither {0} nor {1}, 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 {0} or {1}.

Step 3.3. Let cbifs be a <condition-bif-value-list> containing <onsource-value>: sv; and <onchar-value>: n; where n is the position of the first <character-value> in sv that has neither {0} nor {1}. Perform raise-condition(<conversion-condition>,cbifs). Let sv be the immediate component of the current <returned-onsource-value>, and go to Step 3.1.

9.5.1.5 Convert-to-character

In this operation the notation '(n){picture-element}' (or some other lowercase letter in place of n) is used to indicate n (n≥0) consecutive occurrences of that {picture-element}.

Operation: convert-to-character(st,sv)

where st is a <data-type>,
sv is a <real-value>, <complex-value>, <bit-string-value>, or <character-string-value>.

result: a <character-string-value>.

Case 1. st has <real> and <fixed> but not <pictured-numeric>.

Step 1.1. If st has <binary>, let dst be a <data-type> which has <real> <fixed> <decimal> with the converted <precision> of st for that target; otherwise, let dst be st. Perform convert(dst,st,sv) to obtain cv.

Let p and q be the <number-of-digits> and <scale-factor> of dst.

Step 1.2.

Case 1.2.1. p ≥ q ≥ 0.

Let pic be the <data-type> corresponding to the {picture} with concrete-representation as follows:

```
if q=0:   '(2)B(p)-9'
if p=q>0: '-9V.(q)9'
if p>q>0: 'B(n)-9V.(q)9' where n=p-q.
```

Perform edit-numeric-picture(cv,pic) to obtain a <character-string-value>,v.

Return v.

Case 1.2.2. q > p or q < 0.

Let pic1 be the <data-type> corresponding to the {picture} with concrete-representation '(p)-9'.

Perform edit-numeric-picture(x,pic1) to obtain a <character-string-value>,v1, where x is a <real-value>: cv*10↑q.

Let k be the unique integer such that 10↑(k-1)≤abs(q)<10↑k. Let pic2 be the <data-type> corresponding to the {picture} with concrete-representation 'S(k)9'. Perform edit-numeric-picture(-q,pic2) to obtain a <character-string-value>,v2.

Let f be a <character-string-value> containing the single {symbol}: {F}. Perform concatenate(v1,f) to obtain s1; perform concatenate(s1,v2) to obtain s2.

Return s2.

Case 2. st has <real> and <float> but not <pictured-numeric>.

Step 2.1. If st has <binary>, let dst be a <data-type> which has <real> <float> <decimal> with the converted <precision> of st for that target; otherwise, let dst be st. Perform convert-to-float-decimal(dst,st,sv) to obtain cv.

Let p be the <number-of-digits> of dst.

Step 2.2. Let k be the implementation-defined size of the exponent field for <floating-point-format>s. Let pic be the <data-type> corresponding to the {picture} with concrete-representation '-9V.(n)9ES(k)9', where n=p-1.

Perform edit-numeric-picture(cv,pic) to obtain a <character-string-value>, v.

Return v.

Case 3. st has <complex> but not <pictured-numeric>, and sv is a <complex-value>.

Step 3.1. Let rst 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(rst,x) to obtain x' and convert-to-character(rst,y) to obtain y'.

Step 3.2. Let n be the length of the <character-string-value>,x' (and necessarily, also of y'). Let i be the number of leading <character-value>s in y' which have 1s. Let y'[j], j=1,...,n, be the <character-value>s in y'.

Step 3.3.

Case 3.3.1 y'[i+1] has {-}.

Let yi be a <character-string-value> of length n+1 containing, in order, y'[i+1] through y'[n], {symbol}: {I}; and y'[1] through y'[i].

Case 3.3.2. (Otherwise).

Let yi be a <character-string-value> of length n+1 containing, in order, {symbol}: {+};, y'[i+1] through y'[n], {symbol}: {I}; and y'[1] through y'[i-1].

Step 3.4. Perform concatenate(x',yi) to obtain a <character-string-value>,v, of length (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>, <pictured-character>, or <pictured-numeric>, and sv is a <character-string-value>.

Return sv.

9.5.1.6 Conversion to Float Decimal

This operation returns a <real-value> which is exactly representable in floating notation in a given number (p) of significant digits. It is used in the conversion of numeric values to character representations.

Operation: convert-to-float-decimal(tt,st,sv)

where tt is a <data-type> which has <real>, <float>, and <decimal>,
st is a <data-type> which has <real>,
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 tt. Let e be the unique integer such that 10↑(e-1)≤abs(sv)<10↑e.

Let cv = 10↑(e-p)*sign(sv)*floor(10↑(p-e)*abs(sv)+0.5).

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*10↑n, where m and n are integers and abs(m)<10↑p.

9.5.1.7 Conversion from String or Picture to Arithmetic

Operation: convert-to-arithmetic(st,sv)

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 <pictured-numeric>.

Step 1.1. Perform validate-numeric-pictured-value(st,sv) to obtain a <picture-validity>,pv, which must have <picture-valid>.

Step 1.2.

Case 1.2.1. st has <real>.

Let rv be the <real-value> in pv. Let ast be the associated arithmetic data-type of st. Return a <value-and-type>: rv ast.

Case 1.2.2. st has <complex>.

Let rv and iv be the real and imaginary parts of the <complex-value> in pv. Let rast be a <data-type> which has <real> but is otherwise as the associated arithmetic data-type of st. Return a <value-and-type>: rv rast iv rast.

Case 2. st has <bit>.

Step 2.1.

Case 2.1.1. sv has <null-bit-string>.

Let cv=0 and n=1.

Case 2.1.2. (Otherwise).

Let n be the length of sv. Let x[i] 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 x[i]*2↑(n-i), for 1≤i≤n.

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 <pictured-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 {P}.

Perform basic-numeric-value(sv) to obtain a <value-and-type>,vt. Return vt.

Case 3.1.2. (Otherwise).

Let cbifs be a <condition-bif-value-list> containing <onsource-value>: sv; and <onchar-value>: n;, where n is the smallest integer such that the <character-string-value> of length n containing the first n <character-value>s of sv does not have a continuation conforming to the syntax of <numeric-string> without {P}. (If the whole of sv has such a continuation, let n be the length of sv.)

Perform raise-condition(<conversion-condition>,cbifs). Let sv be the immediate component of the current <returned-onsource-value>, and go to Step 3.1.

9.5.1.8 Basic Numeric Value of a String

<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>.

<numeric-string>::= <blanks> |
<blanks> [[+|-] {arithmetic-constant} |
<complex-expression>) <blanks>

<blanks>::= [{B-list}]

<complex-expression>::= <sign-r> {real-constant}
<sign-i> {imaginary-constant}

<sign-r>::= [+|-]

<sign-i>::= +|-

Operation: basic-numeric-value(str)

where str is a <character-string-value> whose terminal components are permitted terminal components of <numeric-string>.

result: a <value-and-type>.

Step 1. Let ns be the <numeric-string> whose components are the elements of str, taken in order.

Step 2.

Case 2.1. ns contains only <blanks>.

Return 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. ns immediately contains {arithmetic-constant}: {real-constant},rc.

Perform evaluate-real-constant(rc) to obtain a <value-and-type>,vt. If ns immediately contains {-}, return a <value-and-type> equal to vt except that the sign of the <real-value> is negative; otherwise, return vt.

Case 2.3. ns immediately contains {arithmetic-constant}: {imaginary-constant}: {real-constant},rc {I}.

Perform evaluate-real-constant(rc) to obtain a <value-and-type>: rv rdt. If ns immediately contains {-}, let v=-rv; otherwise, let v=rv. Return a <value-and-type>: <real-value>: 0; rdt <real-value>: v; rdt.

Case 2.4. ns contains <complex-expression>,cex.

Step 2.4.1. Perform evaluate-real-constant(rrc), where rrc is the {real-constant} immediately contained in cex, to obtain a <value-and-type>: rv rdt. Perform evaluate-real-constant(irc), where irc is the {real-constant} in the {imaginary-constant} in cex, to obtain a <value-and-type>: iv idt.

Step 2.4.2. If <sign-r> has {-}, let rv1=-rv; otherwise, let rv1=rv. If <sign-i> has {-}, let iv1=-iv; otherwise let iv1=iv.

Step 2.4.3. Return a <value-and-type>: rv1 rdt iv1 idt.

9.5.1.9 Evaluate-real-constant

Operation: evaluate-real-constant(rc)

where rc is a {real-constant}.

result: a <value-and-type> (see Section 9.5.1.8).

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 components of {exponent} as a decimal constant; otherwise, let x=0.

Step 3. Let ar be a <data-type> containing <arithmetic> containing <mode>: <real>; and <scale>, <base>, and <precision> as defined below:

Step 3.1. <base> has <decimal> or <binary> as b=10 or b=2.

Step 3.2. <number-of-digits> is the total number of {digit}s or {binary-digit}s in {decimal-number} or {binary-number}.

Step 3.3.

Case 3.3.1. rc contains {scale-type}: {E}.

<scale> has <float>.

Case 3.3.2. (Otherwise).

<scale> has <fixed>. <scale-factor> is (q-x), where q is the number of {digit}s or {binary-digit}s following {.} (if any) in {decimal-number} or {binary-number}.

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>: v*b↑x; ar.

9.5.2 NUMERIC PICTURES

Informally, <pictured-numeric> is a way of holding a numeric value in a ⟨character-string-value⟩. For a <pictured-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 <pictured-numeric> in a <format-item> the value will be a string of characters transmitted to or from a ⟨stream-dataset⟩, an <expression> in a <get-string>, or a <target-reference> in a <put-string>, and represents a real numeric value.

The <numeric-picture-specification> in <pictured-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 <mode>, which is declared or defaulted (or always <real> for a <format-item>), these make up the <arithmetic> subnode of <pictured-numeric> established in Section 4.4.6.1. The <data-type> which contains <arithmetic> with the same subnodes is known as the associated arithmetic data-type of the <pictured-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 {K} and of {V}.

    The associated character-string length of a <data-type> (or <format-item>) containing <pictured-numeric> and <real> is equal to that of its <numeric-picture-specification>.

    The associated character-string length of a <data-type> containing <pictured-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 <pictured-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 <pictured-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-mantissa>
<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 ‹pictured-numeric› with associated arithmetic data-type, adt.

result: 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 d[j] :

$$\text{sgn} * \sum_{j=1}^{p} (d[j] * 10\uparrow(p-q-j))$$

where $0 \le d[j] \le 9$, j=1 to p
sgn = +1 if vc ≥ 0
= -1 if vc < 0.

Perform edit-numeric-picture-field(spec,d,sgn) to obtain a ‹character-string-value›,s, 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[j].

Step 1.2. Return s.

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 4.4.6.1) following the {V} ‹numeric-picture-element› in ‹picture-mantissa› of pic (q = 0 if there is no {V}); let px be the number of digit-positions in ‹picture-exponent› of pic.

Then there is a unique representation of vc as vm * 10 ↑ vx, where vx is a signed integer such that:

$10\uparrow(p-q-1) \le \text{abs}(vm) < 10\uparrow(p-q)$ if vc ≠ 0

or vm = vx = 0 if vc = 0.

Step 2.2. There is a unique representation of vm in terms of integers d[j]:

$$\text{sgn} * \sum_{j=1}^{p} (d[j] * 10\uparrow(p-q-j))$$

where $0 \le d[j] \le 9$, j=1 to p
sgn = +1 if vm ≥0
= -1 if vm <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[j].

Step 2.3. If abs(vx) ≥ 10 ↑ px, perform raise-condition(<size-condition>); otherwise, there is a unique representation of vx in terms of integers d[j] :

$$\text{sgn} * \sum_{j=1}^{px} (d[j] * 10 \uparrow (px-j))$$

where $0 \le d[j] \le 9$, j=1 to px
sgn = +1 if vx ≥0
= -1 if vx <0.

Perform edit-numeric-picture-field(spec,d,sgn) to obtain a <character-string-value>,cx, 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 d[j].

Step 2.4. Perform concatenate(cm,cx) to obtain c. Return c.

Case 3. adt has <mode>: <complex>.

Step 3.1. Let vcr, vci be the real and imaginary parts of vc; let picr be the <data-type> which is the same as pic except that it has <mode>: <real>.

Perform edit-numeric-picture(vcr,picr) to obtain cr; perform edit-numeric-picture(vci,picr) to obtain ci.

Step 3.2. Perform concatenate(cr,ci) to obtain c. Return c.

9.5.2.2 Editing a Numeric Picture Field

<pic-status>::= <suppression> [<suppression-type>]

<suppression>::= <on> | <off>

<suppression-type>::= [{$} | {S} | {+} | {-} | {Z} | {*}]

Operation: edit-numeric-picture-field(pic,d,sgn)

where pic is a <numeric-picture-element-list> containing ns <numeric-picture-element>s,
d is a <character-string-value> containing p {symbol}s, where p is the number of digit-positions in pic and each {symbol} corresponds to a digit,
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 sgn = -1, perform raise-condition(<size-condition>) unless pic contains at least one <numeric-picture-element> which immediately contains {S}, {+}, {-}, {T}, {I}, {R}, <credit>, or <debit>.

Step 2. If pic contains no <numeric-picture-element> which immediately contains {9}, {T}, {I}, {R}, or {Y}, and if all {symbol}s in d have {0}, then return a <character-string-value> of length nc all of whose <character-value>s contain {*} or Ø according to whether pic contains an {*} or not.

Step 3. Let s[i], i=1,...,ns, be the <numeric-picture-element>s in pic.

Let d[j], j=1,...,p, be the <character-value>s in d.

Let c[k], k=1,...,nc, be <character-value>s; the remainder of this operation completes the trees c[k] as a function of the s[i] and d[j].

Let pstat be a <pic-status>:
<suppression>,sup:
<off>.

Let each of i, j, k be initially 1.

Step 4. Select the appropriate Case depending on s[i].

Case 4.1. s[i] immediately contains {9}, {Y}, {T}, {I}, or {R}.

Step 4.1.1.

Case 4.1.1.1. sup is <suppression>: <on>.

Replace sup by <suppression>: <off>. If <suppression-type> has {$}, {S}, {+}, or {-}, replace the {symbol} contained in c[k-1] by the {symbol} obtained from Table 9.3 as a function of <suppression-type> and (possibly) sgn.

Case 4.1.1.2. sup is <suppression>: <off>.

No action.

Step 4.1.2. Attach to c[k] the {symbol} obtained from Table 9.4 as a function of s[i], d[j], and (possibly) sgn.

Step 4.1.3. i=i+1; j=j+1; k=k+1.

Case 4.2. s[i] has {Z} or {*}.

Step 4.2.1. If pstat does not have <suppression-type>, replace sup by <suppression>: <on>; and append <suppression-type>: pc; to pstat, where pc is the component of s[i].

Step 4.2.2.

Case 4.2.2.1. sup is <suppression>: <on>; and d[j] has {0}.

If s[i] has {Z}, attach Ø to c[k];
if s[i] has {*}, attach {*} to c[k].

Case 4.2.2.2. sup is <suppression>: <off>; or d[j] does not have {0}.

Attach the immediate component of d[j] to c[k]. Replace sup by <suppression>: <off>.

Step 4.2.3. i=i+1; j=j+1; k=k+1.

Case 4.3. s[i] has {$}, {S}, {+}, or {-}, and there is no other n such that s[n] = s[i].

Step 4.3.1. Attach to c[k] the {symbol} obtained from Table 9.3 as a function of s[i] and (possibly) sgn.

Step 4.3.2. i=i+1; k=k+1.

Case 4.4. s[i] has {$}, {S}, {+}, or {-}, and s[n] = s[i] for some n greater than i, but for no n less than i.

Step 4.4.1. Replace sup by <suppression>: <on>; append <suppression-type>: pc; to pstat, where pc is the component of s[i].

Step 4.4.2. Attach Ø to c[k].

Step 4.4.3. i=i+1; k=k+1.

Case 4.5. s[i] has {$}, {S}, {+}, or {-}, and s[n] = s[i] for some n less than i.

Step 4.5.1.

Case 4.5.1.1. sup is <suppression>: <on>; and d[j] has {0}.

Attach Ø to c[k].

Case 4.5.1.2. sup is <suppression>: <on>; and d[j] does not have {0}.

Replace the immediate component of c[k-1] by the {symbol} obtained from Table 9.3, as a function of s[i] and (possibly) sgn. Attach the immediate component of d[j] to c[k]. Replace sup by <suppression>: <off>.

Case 4.5.1.3. sup is <suppression>: <off>.

Attach the immediate component of d[j] to c[k].

Step 4.5.2. i=i+1; j=j+1; k=k+1.

Case 4.6. s[i] has {V}.

Step 4.6.1. Perform Step 4.1.1.

Step 4.6.2. If pstat does not have <suppression-type>, append <suppression-type> (with no subnode) to pstat.

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. s[i] has <insertion-character>.

Step 4.7.1.

Case 4.7.1.1. sup is <suppression>: <off>.

If s[i] has {B}, attach Ø to c[k]; otherwise attach to c[k] the terminal component of s[i].

Case 4.7.1.2. sup is <suppression>: <on>.

If <suppression-type> has {*}, attach {*} to c[k], otherwise, attach Ø to c[k].

Step 4.7.2. i=i+1; k=k+1.

Case 4.8. s[i] has <credit> or <debit>.

Step 4.8.1.

Case 4.8.1.1. sgn=+1.

Attach Ø to each of c[k] and c[k+1].

Case 4.8.1.2. sgn=-1.

Attach to c[k] and c[k+1] {C} and {R} (respectively) if s[i] has <credit>, or {D} and {B} (respectively) if s[i] has <debit>.

Step 4.8.2. i=i+1; k=k+2.

Case 4.9. s[i] has {E}.

Attach {E} to c[k].

i=i+1; k=k+1.

Case 4.10.s[i] has {K}.

i=i+1.

Step 5. If i≤ns then go to Step 4.

Step 6. Return a <character-string-value> containing c[k], k=1,...,nc, in order.

Table 9.3. Table of {symbol}s as a Function of <suppression-type> for Edit-numeric-picture-field.

sgn	s[i] or <suppression-type>			
	{S}	{+}	{-}	{$}
+1	{+}	{+}	Ø	*
-1	{-}	Ø	{-}	*

The positions indicated by * represent a {symbol} which is implementation-defined. This {symbol} represents a currency symbol.

Table 9.4. Table of {symbol}s as a Function of <numeric-picture-element>s and <character-value>s for Edit-numeric-picture-field.

s[i]	sgn (if applicable)	d[j]									
		{0}	{1}	{2}	{3}	{4}	{5}	{6}	{7}	{8}	{9}
{Y}		Ø	{1}	{2}	{3}	{4}	{5}	{6}	{7}	{8}	{9}
{9} {R} {I}	+1 -1	{0}	{1}	{2}	{3}	{4}	{5}	{6}	{7}	{8}	{9}
{T} {I}	+1 +1	*	*	*	*	*	*	*	*	*	*
{T} {R}	-1 -1	*	*	*	*	*	*	*	*	*	*

The 20 positions indicated by * represent 20 {symbol}s which are implementation-defined. These {symbol}s represent a digit and a sign in one {symbol}.

9.5.2.3 Validity of a Numeric Pictured Value

<picture-validity>::= <picture-valid> | <picture-invalid>

<picture-valid>::= [<real-value> | <complex-value>]

<picture-invalid>::= <integer-value>

The subnode of <picture-valid> is the associated numeric value of v with respect to pic. The subnode of <picture-invalid> is the ordinal of the first <character-value> in v which is invalid with respect to pic.

This operation is invoked by the operation valid-bif, by the operation convert for <pictured-numeric> source-type, and by the operation validate-input-format for a <picture-format>.

Operation: validate-numeric-pictured-value(pic,v)

where pic is a <data-type> or <picture-format> containing <pictured-numeric>,
v is a <character-string-value> of length equal to the associated character-string length of pic.

result: a <picture-validity>.

Case 1. pic has associated arithmetic data-type with <mode>: <real>; and <scale>: <fixed>.

Step 1.1. Perform validate-field-of-pictured-value(plst,v) to obtain a <picture-validity>,pv, where plst is the <numeric-picture-element-list> in pic.

Step 1.2.

Case 1.2.1. pv has <picture-valid>.

Let rv be the <real-value> in pv. Return a <picture-validity>: <picture-valid>: <real-value>: rv*10↑(-q);;;, where q is the <scale-factor> in the associated arithmetic data-type of pic.

Case 1.2.2. pv has <picture-invalid>.

Return pv.

Case 2. pic has associated arithmetic data-type with <mode>: <real>; and <scale>: <float>.

Step 2.1. Let pm, px be the <numeric-picture-element-list>s of <picture-mantissa> and <picture-exponent> in pic. Let im, ix be the associated character-string lengths of pm and px. Let vm, vx be <character-string-value>s containing the first im and last ix <character-value>s of v, respectively.

Step 2.2. Perform validate-field-of-pictured-value(pm,vm) to obtain pvm; perform validate-field-of-pictured-value(px,vx) to obtain pvx.

Step 2.3.

Case 2.3.1. pvm has <picture-invalid>.

Return pvm.

Case 2.3.2. pvm has <picture-valid>, and pvx has <picture-invalid>.

Return a <picture-validity>: <picture-invalid>: <integer-value>: (im+n);;;, where n is the value in pvx.

Case 2.3.3. (Otherwise).

Return a <picture-validity>: <picture-valid>: <real-value>: (m*10↑(x-q));;;, where m is the value in pvm, x is the value in pvx, and q is the number of digit-positions following {V} in pm (q=0 if there is no {V} in pm).

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 picr; that of pic is therefore 2*ir. Let vr, vi be <character-string-value>s containing the first ir and last ir <character-value>s 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 pvi.

Step 3.3.

Case 3.3.1. pvr has <picture-invalid>.

Return pvr.

Case 3.3.2. pvr has <picture-valid>, and pvi has <picture-invalid>.

Return a <picture-validity>: <picture-invalid>: <integer-value>: n;;;, where n is the sum of ir and the value in pvi.

Case 3.3.3. pvr and pvi both have <picture-valid>.

Return a <picture-validity>: <picture-valid>: cv;;, where cv is a <complex-value> with real and imaginary parts equal to the values in pvr and pvi, respectively.

9.5.2.4 Validity of a Field of a Numeric Pictured Value

Operation: validate-field-of-pictured-value(plst,v)

where plst is a <numeric-picture-element-list> of a <fixed-point-picture>, <picture-mantissa>, or <picture-exponent>,
v is a <character-string-value> of length equal to the associated character-string length of plst.

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-picture-field(plst,d,sgn) where d is the <character-string-value> containing as many digits as there are digit-positions in plst, each digit independently takes each value 0 through 9 in turn, and sgn takes values +1 and -1 in turn.

Let vd and vsign be the unique values of d and sgn which edited to v. Let vx be the <real-value> which is the integer containing the same digits as vd.

Return a <picture-validity>: <picture-valid>: <real-value>: vx*vsign.

Case 2. (Otherwise).

Let n be the lowest integer such that the first n <character-value>s of v are different from the first n <character-value>s of all the values obtainable from the editing operations in the predicate of Case 1.

Return a <picture-validity>: <picture-invalid>: <integer-value>: n.

9.5.3 CHARACTER PICTURES

The operation validate-character-pictured-value defines the checking of a <character-string-value> for validity with respect to a <data-type> (or <format-item>) which contains <pictured-character>. This operation is invoked by operation valid-bif, by operation convert for a <pictured-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 <pictured-character>, or of a <character-picture-element-list> is the number of <character-picture-element>s 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 <pictured-character>,
v is a <character-string-value> whose length, n, is the associated character-string length of pic.

result: a <picture-validity> (Section 9.5.2.3).

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 {9}.

If c has a digit or a Ø, return <true>; otherwise, return <false>.

Case 2. pc has {A}.

If c has a letter or a Ø, then return <true>; otherwise, return <false>.

Case 3. pc has {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 first number references the page of definition. For other specially defined terms, the numbers indicate where a definition may be found. Category-names ending in '-list', '-commalist', and '-designator' are indexed with the unsuffixed category.

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Z