TECHNICAL REPORT
(Version 1)

TR 23.071
30 DECEMBER 1966

FORMAL DEFINITION OF PL/1
(Universal Language Document No. 3)

by the

PL/1 - Definition Group of
the Vienna Laboratory
THIS DOCUMENT IS BASED ON THE PL/I LANGUAGE SPECIFICATIONS (FORM NO. C28-6571-3) BUT DIFFERS IN DETAILS AND EXTENT OF DESCRIPTION FROM THESE SPECIFICATIONS.
ABSTRACT:
This document provides a formal definition of PL/I based on the PL/I Language Specifications (Form No. C28-6571-3) but deviates in some details. The semantics of PL/I is given by an abstract machine called the PL/I-machine which interprets PL/I programs. The structure of PL/I programs is specified in an abstract form without referring to the concrete representation.

TR 25.071
30 December 1966
(Abstract and Preface revised 27 Jan 1967)
During 1967, a revised version will be issued which will supersede the present one.
PREFACE

This report is the result of a common effort in the Vienna Laboratory based on all previous interest and engagement in programming languages and, in particular, in PL/I since its first documents.

Method

The method of definition and the basic notions of the presented form of PL/I-definition have essentially been developed by P. LUCAS and K. WALK.

The authors and their contributions

In the following, the authors are listed according to their main contributions by chapters.

Method and Notation

K. WALK: 1, 2.8, 3.2
P. LUCAS: 1, 2.1 to 2.7, 3.1 (except 3.1.35 and 3.1.7)

Expression Evaluation, Assignment, Storage Allocation

K. WALK: 4 (except 4.3, 4.8, 4.10, and 4.11), 5.3, 5.7, 5.8
G. CHROUST: 4.3
H. BEKIC: 4.8, 5.11

Flow of Control Statements, Prepass

P. LUCAS: 4.10, 4.11, 5.1, 5.2, 5.4, 5.5, 5.9, 5.10

Conditions, Tasks

K. BANDAT: 3.1.35, 5.6, 5.12, 7

Input, Output

P. OLIVA: 6.1 to 6.3, 6.5, 3.1.7
V. KUDIELKA: 6.4

Abstract Syntax

K. ALBER: Appendix I
Organization

The editing work, a particular problem with the present text, has been done by F. SCHWARZENBERGER and G. CHROUST. The printout of the index of function and instruction names (Appendix II) has been prepared by K.F. KOCH. The insertion of special graphics was done by H. HOJA and G. LEHMAYER. Our secretaries W. SCHATZL and H. DEIM did a very good job typing the document. Besides the authors, K. ALBER and H. BEKIC have proofread most of the text.

History

This Technical Report, called "Formal Definition of PL/I" (Universal Language Document No. 3: 'ULD 3'), describes PL/I as gathered from the SRL - Manual Form C 28-6571-3. It formally describes the programming language, but not a specific implementation in a specific environment.

The project was initiated by a formal proposal made by members of the Hursley Laboratory, of the Poughkeepsie Laboratory, and of the Vienna Laboratory in October 1965. This proposal was accepted by the PL/I manager and the Vienna Laboratory was committed subsequently to prepare the document by end of 1966 in the form of a technical report.

The early ideas of the Vienna group on the method are documented in the "Tentative Steps" /1/ and have been discussed with responsible and interested IBM advanced programming specialists on many occasions. There was a frequent exchange of working documents and there were regular working meetings between the Hursley language definition group, the Poughkeepsie language evaluation group, and the Vienna language definition group. K. BANDAT represented the Vienna Laboratory in the Language Resolution Board and served as liaison to the other IBM PL/I groups.
The method of definition and its application to PL/I have been presented at various occasions internally in IBM to groups involved in PL/I and at the IBM Programming Symposium in Skytop in November 1965 \(^2\), to SHARE representatives in Vienna in October 1966, and to language committees outside IBM \(^3\).

Remark

This first version of the document contains essentially the formal description as such, with a minimum of reading aids. It certainly is not a teaching device or a kind of text-book. But some additional work will yield a considerable improvement in this direction.

We believe that formal definition is a necessary step in the development of the art of programming. This document on PL/I not only removes the ambiguities that no informal description can avoid. It also establishes the way of discussing and asking questions in precise terms. It constitutes a ground for the future development of PL/I.

We hope that this document and its coming improved versions will be a good service for IBM and IBM's customers.

Vienna, 30 December 1966.
REFERENCES


## CONTENTS

### PREFACE

### CONTENTS

1. INTRODUCTION

1.1 Aims and Scope of the Document
1.2 Outline of the Definition Method
1.3 Special Comments to the Present Version
1.4 Acknowledgements

2. DEFINITION TOOLS AND NOTATIONAL CONVENTIONS

2.1 Conditional Expressions
2.2 Equality
2.3 Truth Values, Logical Operators and Quantifiers
2.4 Arithmetic Operators and Relations
2.5 Rules of Precedence
2.6 Set Operators, Relations and Definition of Sets
2.7 Functional Composition
2.8 Objects and Operations on Objects
   2.8.1 Objects
      2.8.11 Intuitive Treatment
      2.8.12 Formal Treatment
   2.8.2 Select-Functions and Elementary Objects Occurring in this Document
   2.8.3 Notational Conventions for the $\mu$-functions and $\delta$-functions
   2.8.4 Definition of Classes of Objects

Page

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. INTRODUCTION</td>
<td>1-1</td>
</tr>
<tr>
<td>1.1 Aims and Scope of the Document</td>
<td>1-1</td>
</tr>
<tr>
<td>1.2 Outline of the Definition Method</td>
<td>1-3</td>
</tr>
<tr>
<td>1.3 Special Comments to the Present Version</td>
<td>1-6</td>
</tr>
<tr>
<td>1.4 Acknowledgements</td>
<td>1-7</td>
</tr>
<tr>
<td>2. DEFINITION TOOLS AND NOTATIONAL CONVENTIONS</td>
<td>2-1</td>
</tr>
<tr>
<td>2.1 Conditional Expressions</td>
<td>2-1</td>
</tr>
<tr>
<td>2.2 Equality</td>
<td>2-2</td>
</tr>
<tr>
<td>2.3 Truth Values, Logical Operators and Quantifiers</td>
<td>2-2</td>
</tr>
<tr>
<td>2.4 Arithmetic Operators and Relations</td>
<td>2-5</td>
</tr>
<tr>
<td>2.5 Rules of Precedence</td>
<td>2-6</td>
</tr>
<tr>
<td>2.6 Set Operators, Relations and Definition of Sets</td>
<td>2-6</td>
</tr>
<tr>
<td>2.7 Functional Composition</td>
<td>2-7</td>
</tr>
<tr>
<td>2.8 Objects and Operations on Objects</td>
<td>2-8</td>
</tr>
<tr>
<td>2.8.1 Objects</td>
<td>2-8</td>
</tr>
<tr>
<td>2.8.11 Intuitive Treatment</td>
<td>2-8</td>
</tr>
<tr>
<td>2.8.12 Formal Treatment</td>
<td>2-9</td>
</tr>
<tr>
<td>2.8.2 Select-Functions and Elementary Objects Occurring in this Document</td>
<td>2-15</td>
</tr>
<tr>
<td>2.8.3 Notational Conventions for the $\mu$-functions and $\delta$-functions</td>
<td>2-15</td>
</tr>
<tr>
<td>2.8.4 Definition of Classes of Objects</td>
<td>2-17</td>
</tr>
</tbody>
</table>
3. THE PL/I MACHINE

3.1 The Constituents of the State of the PL/I Machine
   3.1.1 Survey of the Constituents of the State
   3.1.2 The Name Lists A, B, E, N, F
   3.1.3 Block Activation and Scope of Names (D, E, ID, RI, CS, C)
   3.1.4 Attribute Directory AT and Denotation Directory DN
   3.1.5 Storage S
   3.1.6 The Aggregate Directory AG
   3.1.7 Parts of the State for Input and Output (E, FD, FU, ES)

3.2 State Transitions and Control of the PL/I Machine
   3.2.1 The Computation of the PL/I Machine
   3.2.2 The Control-Part of the PL/I Machine
   3.2.3 The Definition of Instructions

4. BASIC INTERPRETATION FUNCTIONS AND INSTRUCTIONS

4.1 Predicates and Functions Defined over Attributes
4.2 Reference to Variables
   4.2.1 Interpretation of Subscripts and Qualifications
   4.2.2 Data Attributes and Reference-Lists
   4.2.3 Sub-Generations of Generations
   4.2.4 Preliminaries on the Expansion of Aggregate Expressions
   4.2.5 Interpretation of the Reference to Variables
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2.6 Interpretation of isub-Defined Variables</td>
<td>4-28</td>
</tr>
<tr>
<td>4.2.7 The Instructions <code>expand-gen</code> and <code>expand-value</code></td>
<td>4-32</td>
</tr>
<tr>
<td>4.3 Operations on Data and Data Conversions</td>
<td>4-34</td>
</tr>
<tr>
<td>4.3.1 Objects Occurring in Expressions</td>
<td>4-34</td>
</tr>
<tr>
<td>4.3.2 Abbreviations and Conventions</td>
<td>4-36</td>
</tr>
<tr>
<td>4.3.3 Internal Representation</td>
<td>4-38</td>
</tr>
<tr>
<td>4.3.4 Interpretation of Infix Operations</td>
<td>4-41</td>
</tr>
<tr>
<td>4.3.5 The Target Attributes of an Operand</td>
<td>4-45</td>
</tr>
<tr>
<td>4.3.6 <code>infix-op(op_1,op_2,opor)</code></td>
<td>4-49</td>
</tr>
<tr>
<td>4.3.7 Prefix Operations</td>
<td>4-55</td>
</tr>
<tr>
<td>4.3.8 The Convert Function</td>
<td>4-56</td>
</tr>
<tr>
<td>4.4 Interpretation of References and Expressions</td>
<td>4-65</td>
</tr>
<tr>
<td>4.5 Interpretation of Arguments</td>
<td>4-68</td>
</tr>
<tr>
<td>4.6 Interpretation of Integer and Scalar Expressions</td>
<td>4-79</td>
</tr>
<tr>
<td>4.7 Scalar Assignment</td>
<td>4-81</td>
</tr>
<tr>
<td>4.8 Interpretation of Pseudo-Variables</td>
<td>4-83</td>
</tr>
<tr>
<td>4.9 Basic Allocation and Initialization</td>
<td>4-88</td>
</tr>
<tr>
<td>4.10 Basic Freeing of Storage</td>
<td>4-93</td>
</tr>
<tr>
<td>4.11 Some Frequently Used Simple Instructions</td>
<td>4-94</td>
</tr>
<tr>
<td>5. CENTRAL PROCESSING STATEMENTS</td>
<td>5-1</td>
</tr>
<tr>
<td>5.1 Initial State and Program Initiation</td>
<td>5-1</td>
</tr>
<tr>
<td>5.2 The Prepass</td>
<td>5-3</td>
</tr>
<tr>
<td>5.3 Interpretation of Statement Lists</td>
<td>5-11</td>
</tr>
<tr>
<td>5.3.1 Case Distinctions for Statements</td>
<td>5-11</td>
</tr>
<tr>
<td>5.3.2 Statement Lists</td>
<td>5-13</td>
</tr>
<tr>
<td>5.3.3 Iterated Statement Lists</td>
<td>5-14</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>5.4</td>
<td>Block Activation and Exit</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Basic Functions and Instructions</td>
</tr>
<tr>
<td>5.4.2</td>
<td>The Procedure Call</td>
</tr>
<tr>
<td>5.4.3</td>
<td>Interpretation of Blocks</td>
</tr>
<tr>
<td>5.4.4</td>
<td>Return Statement</td>
</tr>
<tr>
<td>5.5</td>
<td>The Goto-Statement</td>
</tr>
<tr>
<td>5.6</td>
<td>The Task Call</td>
</tr>
<tr>
<td>5.7</td>
<td>Assignment-Statement</td>
</tr>
<tr>
<td>5.8</td>
<td>Allocate-Statement</td>
</tr>
<tr>
<td>5.8.1</td>
<td>Initialization</td>
</tr>
<tr>
<td>5.8.2</td>
<td>Controlled Allocate</td>
</tr>
<tr>
<td>5.8.3</td>
<td>Based Allocate</td>
</tr>
<tr>
<td>5.8.4</td>
<td>Area Allocate</td>
</tr>
<tr>
<td>5.9</td>
<td>Free-Statement</td>
</tr>
<tr>
<td>5.10</td>
<td>Conditional Statement</td>
</tr>
<tr>
<td>5.11</td>
<td>Built-in Functions</td>
</tr>
<tr>
<td>5.12</td>
<td>Conditions</td>
</tr>
<tr>
<td>5.12.1</td>
<td>Condition Prefixes</td>
</tr>
<tr>
<td>5.12.2</td>
<td>Condition Denotation</td>
</tr>
<tr>
<td>5.12.3</td>
<td>Initial State $CS_0$</td>
</tr>
<tr>
<td>5.12.4</td>
<td>On-Statement</td>
</tr>
<tr>
<td>5.12.5</td>
<td>Revert-Statement</td>
</tr>
<tr>
<td>5.12.6</td>
<td>Signal-Statement</td>
</tr>
<tr>
<td>5.12.7</td>
<td>Interpretation of Condition Denotation</td>
</tr>
<tr>
<td>5.12.8</td>
<td>Raising Check Condition and I/O Conditions</td>
</tr>
<tr>
<td>5.12.9</td>
<td>Undefined Raising of Conditions</td>
</tr>
</tbody>
</table>
6. INPUT AND OUTPUT STATEMENTS

6.1 Abbreviations

6.2 Opening of a File
   6.2.1 File Description Attributes
   6.2.2 Interpretation of the Open-Statement

6.3 Stream Transmission
   6.3.1 The Get-Statement
      6.3.11 Source Items and Elementary Input Instructions
      6.3.12 The Specification Information
      6.3.13 Notes to the Interpretation
      6.3.14 Interpretation of the Get-Statement
   6.3.2 The Put-Statement
      6.3.21 Elementary Output Functions and Instructions
      6.3.22 Notes to the Interpretation
      6.3.23 Interpretation of the Put-Statement

6.4 Record Transmission
   6.4.1 Sequential and Direct Access to Record Data Sets
   6.4.2 Unbuffered Record Transmission
   6.4.3 Buffered Record Transmission
   6.4.4 Locking and Unlocking of Records
   6.4.5 Interpretation of the Expressions in Key-, Keyfrom-, Ignore-Options
   6.4.6 The Read-Statement
   6.4.7 The Rewrite-Statement
   6.4.8 The Delete-Statement
6.4.9 The Unlock-Statement 6-58
6.4.10 The Locate-Statement 6-59
6.4.11 The Write-Statement 6-60
6.5 Closing of a File 6-63
6.5.1 Interpretation of the Close-Statement 6-63

7. PARALLEL ACTIONS 7-1

APPENDIX I. ABSTRACT SYNTAX OF TEXT AI-1
1. Notations AI-1
2. List of Non-Elementary Predicate Definitions AI-2
3. List of Elementary Predicates AI-14
4. Additional Conditions AI-17
5. Comments AI-23

APPENDIX II. INDEX AII-1
1. INTRODUCTION

1.1 Aims and Scope of the Document

The aim of this document is, first, to present a method for the formal definition of programming languages and, second, to give a formal definition of PL/I.

Only a small number of methods for the definition of programming languages have been demonstrated as yet and none of them has been applied to a language of the size of PL/I. It was mandatory, therefore, that new definitional concepts were introduced and economic notational conventions were developed. It has been attempted, however, to remain within the well-established notions of logic and, especially, of automata theory and also to fit the notation as close as possible to the conventional notation in these fields. An essential property of the method is the easiness of manipulation of abstract objects of a rather general class.

The method adopted is based on the definition of an abstract machine which is characterized by the set of its states and its state transition function. A PL/I-program defines an initial state of the machine and the subsequent behaviour of the machine is said to define the interpretation of the PL/I-program.

The definition method, evidently, is applicable to a class of languages where PL/I is only a member. This brings about that the definition is transparent with respect to the various decisions that have been made in the course of the development of PL/I and with respect to the consequences of these decisions. Although the present document is not in itself an evaluation of PL/I, it would seem beneficial to refer to the document for an evaluation.
It should be clear that the definition does not solve the problem of compiler-design for PL/I, specifically it does not reflect any considerations of efficient implementation. On the other hand, the definition may be considered to state precisely the problem to the compiler designer. The definition also specifies for him those areas of the language where he is free to choose any interpretation out of a given class.

The application of the definition method resulted in a completely formal definition of PL/I. The mnemonic names frequently used for the sake of the readability of the formulas should not induce the idea that they convey any meaning inherent in the names.

It should be possible and it is suggested to use the formal definition for cleaning up the PL/I terminology as used in prose descriptions, especially for tutorial purposes. It is considered essential, particularly for discussions on PL/I, to have a vocabulary of terms with a precise and agreed-upon meaning, allowing for economic communication among PL/I-users. It will require additional effort to clean up the present document to achieve this aim.

In a certain sense the document is self-contained, but it will be advantageous for the reader to have some familiarity with PL/I as such and also some skill in reading logical formulas and algorithms. It may be expected that especially a programmer will not find it difficult to penetrate the formal building.

The document is organized as follows. Definition tools and notational conventions are described in chapter 2, especially the definition of the class of objects considered and of the operations for the manipulation of objects is given in 2.8. The set of states of the PL/I-machine and its state transition function with respect to a set of instructions is described in chapter 3. This chapter also introduces the notation used for the definition of instructions. Chapter 4
gives the basic building blocks for the definition of PL/I-statements, which are defined in chapters 5 and 6. The PL/I-machine is not capable of reflecting the parallel execution of actions and thus is not adequate to model PL/I-tasking. This problem is dealt with separately in chapter 7.

It is suggested to the reader to study chapters 2, 3, and 4 in their sequential order. This will give him the knowledge necessary to use chapters 5 and 6 as source of reference for the interpretation of individual PL/I-statements.

1.2 Outline of the Definition Method

The basis for the development of the present method are the publications of J. McCARTHY, C.C. ELGOT and P. LANDIN. Especially the notions of instruction and computation are similar to those given by ELGOT. The notion of abstract syntax is due to McCARTHY.

---


The following is a summary of the salient properties of the PL/I-machine.

The program interpreted by the PL/I-machine is an abstract object that exhibits the essential structure of a PL/I-program, without reflecting the representation as a string of characters. The definition of the class of objects that are considered syntactically correct programs is called abstract syntax (given in Appendix I). It is assumed that a translator exists that maps character strings representing PL/I-programs into their associated abstract object (this translator is given in a separate document). The abstract object can be considered as the normal form of the program.

It is an essential property that the behaviour of the PL/I-machine is not fully specified:

a) there are functions used in the definition whose specification is left to the particular implementations (implementation-defined functions).

b) there are certain actions whose order of execution is undefined. This feature is modeled by the property of the state transition function, which defines a set of successor states for a given state.

c) instructions of the machine in general are defined by partial functions (mapping states into states), i.e. they are not defined for all possible states of the machine. Invalid programs may lead to instructions that are not defined for the current state. The meaning of a program is completely undefined in such a case, i.e. there is not necessarily a correlation between this current state and the result of the execution of the corresponding program in an implementation. No explicit error messages are given for this reason in the definition.
Some of the definitions given in the document are implicit definitions, i.e. the form of the definition does not immediately specify an algorithm, although it is guaranteed that an algorithm always exists. Implicit definitions have been used in cases where special algorithms would obscure the effect of the definition.

A number of features of PL/I have been eliminated from the core of the definition and are dealt with separately. There are four cases to be distinguished:

a) there are a number of features in PL/I that are considered as notational conventions and dealt with in the translator, such as factoring of attributes, default rules etc.

b) there are a few features which cannot be described in the framework of the present definition, such as the secondary attribute, the delay statement etc.

c) there are a number of features that give the programmer the possibility to make assertions about his program, such as the attributes abnormal, reducible, recursive etc. The definition should give a precise meaning to these properties which can be asserted, on the basis of the present model (these definitions will be given in a later version).

d) the problem of tasking is the definition of the interrelation of actions executed in parallel. Since only a few interrelations are defined in PL/I it seems desirable to describe these interrelations separately from the execution of a simple task. (Note that parallel execution has to be distinguished from execution in unspecified order.)
1.3 Special Comments to the Present Version

Principally, the present version is based on the status of PL/I in May 1966. Practically the PL/I Language Specification Form C28-6571-3 has been used so that all features described in this version are in accordance with this publication. A minor part of the language points issued since May 1966 have been incorporated. A good deal of additional information used stems from working documents for ULD2 as produced by the Hursley Language Definition Group and from discussions with members of this group. Further additional information was gained from discussions with the PL/I implementation groups and almost all parties involved in PL/I definition.

Remark on the omissions:

There are some features of PL/I which can be defined within the present definitional frame-work but are not contained in this version, such as e.g. the generic attribute. There are other cases that require a different definitional frame-work, e.g., the secondary attribute requires the concept of storage levels, delay requires the concept of real time.

Remark on the deviations:

In some cases the available information left open a number of interpretations of a PL/I-feature. In each case a decision has been made for the present definition. A few deviations are intentional and may be considered as proposals for a redefinition.

A list of omissions and deviations will be given in a separate memo.
1.4 Acknowledgements

The authors gratefully acknowledge the pleasant and beneficial cooperation with the Hursley Language Definition Group, the helpful discussions with Dr. E.F. Codd and the members of the Language Resolution Board and the implementation groups. For encouraging discussions and advise the authors are indebted to J.L. Cox and Prof. H. Zemanek.
2. DEFINITION TOOLS AND NOTATIONAL CONVENTIONS

This chapter describes in general the notational conventions and definition methods adopted for the definition of the PL/I machine. Further conventions and abbreviations will be introduced in those chapters where they are needed for specific parts of the definition. Especially the conventions for the definition of instructions are introduced in chapter 3.2.

A major portion of the notation is adopted from LISP (conditional expressions), predicate-calculus, arithmetic expressions and relations and set-theory with the conventional meaning. Sometimes the definition of meaning will be given below either to repeat the conventional meaning or because of slight deviations from it.

However, for the purposes of the present document it has proved to be appropriate to introduce definitional methods and operations for a very general class of objects. These are described in chapter 2.3.

2.1 Conditional Expressions

**Form:**

\[ \left( p_1 \rightarrow e_1 , p_2 \rightarrow e_2 , \ldots , p_n \rightarrow e_n \right) \]

- \( p_i \) expression denoting a truth-value (proposition)
- \( e_i \) expression denoting some object (the value of \( e_i \))

**Meaning:**

A conditional expression denotes the value of \( e_i \) where \( i \) is the smallest integer, \( 1 \leq i \leq n \), for which \( p_i \) is true and \( p_j \),
1 ≤ j < i is false. If there is no integer i such that \( p_i \) is true then the expression has no denotation.

Comment:

It is important to note that the left to right order in a conditional expression is relevant. It is an important consequence of the definition that the values of \( p_j, i < j ≤ n \), are irrelevant and may even be undefined if \( p_i \) is true.

2.2 Equality

\[ \begin{align*}
&= \quad \text{equal} \\
\neq \quad &\text{not equal} \\
\text{EQU } &x_i = x_j \text{ for all } i, j, 1 ≤ i ≤ n, 1 ≤ j ≤ n
\end{align*} \]

The equality and not equality relations are used with no specific restriction to the range of arguments.

2.3 Truth Values, Logical Operators and Quantifiers

2.3.1 Truth Values

\[ \begin{align*}
&T \quad \text{true} \\
&F \quad \text{false}
\end{align*} \]

2.3.2 Logical Operators

The meaning of the logical infix operators deviates from the conventional meaning. The meaning given is best described using conditional expressions.
\[ \neg \text{ not} \]
\[ \& \text{ logical and} \]
\[ (p_1 \& p_2) = (\neg p_1 \rightarrow F, T \rightarrow p_2) \]
\[ \lor \text{ logical or (vel)} \]
\[ (p_1 \lor p_2) = (p_1 \rightarrow T, T \rightarrow p_2) \]
\[ \equiv \text{ equivalence} \]
\[ (p_1 \equiv p_2) = (p_1 \& p_2) \lor (\neg p_1 \& \neg p_2) \]
\[ \neq \text{ non equivalence} \]
\[ (p_1 \neq p_2) = \neg (p_1 \equiv p_2) \]
\[ \Rightarrow \text{ implication} \]
\[ (p_1 \Rightarrow p_2) = (\neg p_1 \rightarrow T, T \rightarrow p_2) \]

The operators \&, \lor and \Rightarrow are right associative, i.e.:

\[ p_1 \& p_2 \& \ldots \& p_{n-1} \& p_n \]
\[ p_1 \lor p_2 \lor \ldots \lor p_{n-1} \lor p_n \]
\[ p_1 \Rightarrow p_2 \Rightarrow \ldots \Rightarrow p_{n-1} \Rightarrow p_n \]

is equivalent to
\[ (p_1 \& (p_2 \& \ldots \ldots (p_{n-1} \& p_n) \ldots ))) \]
\[ (p_1 \lor (p_2 \lor \ldots \ldots (p_{n-1} \lor p_n) \ldots )) \]
\[ (p_1 \Rightarrow (p_2 \Rightarrow \ldots \ldots (p_{n-1} \Rightarrow p_n) \ldots )) \]

\[ \bigwedge_{i=1}^{n} p_i \]
\[ \bigvee_{i=1}^{n} p_i \]

is equivalent to
\[ p_1 \& p_2 \& \ldots \& p_n \]
\[ p_1 \lor p_2 \lor \ldots \lor p_n \]
2.3.3 Quantifiers

\((\exists x_1, x_2 \ldots \ldots x_n)(p(x_1,x_2 \ldots \ldots x_n))\) existential quantifier

The variables \(x_1, x_2 \ldots \ldots x_n\) are called the bound variables of the expression.

The expression is true iff there exists at least one \(n\)-tuple \(x_1, x_2 \ldots \ldots x_n\) such that \(p(x_1,x_2 \ldots \ldots x_n)\) is true, otherwise the expression is false.

\((\forall x_1, x_2 \ldots \ldots x_n)(p(x_1,x_2 \ldots \ldots x_n))\) universal quantifier

The variables \(x_1, x_2 \ldots \ldots x_n\) are called the bound variables of the expression.

The expression is true iff for all possible \(n\)-tuples \(x_1, x_2 \ldots \ldots x_n\) (in the range of the variables) \(p(x_1,x_2 \ldots \ldots x_n)\) is true, otherwise the expression is false.

It is important that the range of the bound variables in expressions of the above form always be defined. This will either be done explicitly by the expression or implicitly by using a convention that associates a range with a specific class of variable names.

In some instances for convenience composite objects (see chapter 2.8) containing bound variables as subparts will be written in place of the bound variable part of the expression,

\[ \text{e.g.} \quad (\exists \langle x,y \rangle)(p(x,y,\langle x,y \rangle)) \]
2.3.4 Description

\((\iota x)(p(x))\)  

The \(\iota\) operator  

\(x\) is called the bound variable of the expression. The expression denotes the object (out of the range of \(x\)) for which \(p(x)\) is true. The expression has no denotation (is undefined) if no or more than one object in the range of \(x\) has the property \(p\).

The \(\iota\) operator is sometimes used to give an implicit definition of a function.

2.4 Arithmetic Operators and Relations

2.4.1 Operators

+ \hspace{1cm} \text{prefix plus, infix plus}
- \hspace{1cm} \text{prefix minus, infix plus}
. \hspace{1cm} \text{multiplication}
/ \hspace{1cm} \text{integer division, yields the next higher integer to the quotient}

\[ \sum_{i=1}^{n} a_i \] \hspace{1cm} \text{is equivalent to} \hspace{1cm} a_1 + a_2 + \ldots + a_n

\[ \prod_{i=1}^{n} a_i \] \hspace{1cm} \text{is equivalent to} \hspace{1cm} a_1 \cdot a_2 \cdot \ldots \cdot a_n

Operations and relations are used for integers only.
2.4.2 Relations

- <   less
- ≤   less or equal
- =   equal
- ≠   not equal
- ≥   greater or equal
- >   greater

2.5 Rules of Precedence

The following rules of precedence on logical operators, arithmetic operators and relations have been adopted.

\[
\begin{array}{cccc}
+ , - & \text{prefix} & \text{highest precedence} & \text{(binds strongest)} \\
\times , / & \text{infix} & \text{lowest precedence} & \text{(binds weakest)} \\
< , ≤ , = , ≠ , ≥ & & & \\
\neg & & & \\
\land & & & \\
\lor & & & \\
\equiv , \neq & & & \\
\supset & & & \\
\end{array}
\]

2.6 Set Operators, Relations and Definition of Sets

2.6.4 Set Operators

- \text{U}   union
- \text{\cap}   intersection
- \text{-}   minus

the expression \( A - B \) is only defined if \( A \supset B \)
2.6.2 Relations

$\in$ is element of
$\notin$ is not element of
$\subset$ is proper subset of
$\subseteq$ is subset or equal
$\supset$ is superset or equal
$\supseteq$ is proper superset

2.6.3 Definition of Sets

\{a, b, c \ldots\} The elements a, b, c \ldots are the
the elements of the set.
\{\} the empty set
\{x \mid p(x)\} implicit definition of a set

The x is called the bound variable of the expression. The expression defines the set of all elements such that $p(x)$ is true.

In some instances for convenience composite objects (see chapter 2.5) containing bound variables as subparts will be written in place of the bound variable part of the expression e.g.

\{ <x,y> \mid p(x,y, <x,y>) \}

$\text{card}(S)$ Gives the number of elements of the set $S$

2.7 Functional Composition

functional composition
If $g$, $h$ are both functions of one argument and the range of values of $h$ is the range of arguments of $g$ then
\[ f = g \circ h \] is equivalent to $f(x) = g(h(x))$
2.8 Objects and Operations on Objects

2.8.1 Objects

The entities manipulated in the PL/I-Machine are objects and sets of objects. In fact, the state of the PL/I-Machine is itself an object. In the following, the notion of object is explained and the various functions for the composition and decomposition of objects are introduced.

2.8.11 Intuitive treatment

An object is either an elementary object, or a composite object. If an object is composite, there exists a certain set of select-functions that may be applied to the object and the application of a certain select-function gives one of the objects of which the composite object is composed. This object again may be elementary or composite. If an object is elementary, there is no select-function (except the identity function) which is applicable to it.

Example: Let the object x be composed of the objects a, b, and c. There are three select-functions, say select-1, select-2, select-3 such that a = select-1(x), b = select-2(x), c = select-3(x). Let a be composed of the objects a₁ and a₂ so that there are two select-functions, say select-first and select-second: a₁ = select-first(a) and a₂ = select-second(a). Then obviously is a₂ = select-second (select-1(x)), or a₂ = select-second·select-1(x).

The dot · is used for functional composition of select-functions. For a certain object there can be only a certain set of composite select-functions that are applicable to the object. A subset of this set is the set of composite select-functions that give the set of elementary objects of which the object is composed.
Example: Let $a_1, a_2, b, c$ in the above example be elementary objects. The set of select-functions that are applicable to $x$ is then

\[
\{ \text{select-1, select-2, select-3, select-first\cdot select-1, select-second\cdot select-1} \}
\]

and the set of select-functions that give elementary objects is

\[
\{ \text{select-first\cdot select-1, select-second\cdot select-1, select-2, select-3} \}.
\]

The basic philosophy is that an object is characterized by the totality of its parts together with the totality of select-functions that give access to these parts. It may be seen that the specification of the set of select-functions that give elementary objects, together with the results of their application, is a sufficient characterization of an object.

There is one general make-function (the $\mu$-function) for the composition of new objects from other objects.

Example: The above $x$ may be represented with the help of the $\mu$-function as follows:

\[
x = \mu_o(<\text{select-1}:a>,<\text{select-2}:b>,<\text{select-3}:c>)
\]

(It is assumed that the meaning of this notation is obvious, for an exact treatment see the later chapter.)

2.8.12 Formal treatment

2.8.12 Given an enumerable set $F$ of symbols

\[
F = \{ \omega_1, \ldots \} ,
\]

we define $F^*$ as the free semigroup with identity with basis $F$ and composition $\cdot$, i.e.:
a) \( F^* = \{ I \} \cup \{ \omega_1 \cdot \omega_2 \cdot \ldots \cdot \omega_n \mid \omega_1, \omega_2, \ldots, \omega_n \in F; n \neq 1 \text{ finite} \} \)

(especially the elements with \( n = 1 \) are the elements of \( F \), so \( F \subseteq F^* \), and \( I \) is considered to be the unique element with \( n = 0 \)).

b) \( (\kappa = \omega_1 \cdot \ldots \cdot \omega_n \in F^* \land \kappa' = \omega_1' \cdot \ldots \cdot \omega_m' \in F^*) \Rightarrow \kappa \cdot \kappa' = \omega_1 \cdot \ldots \cdot \omega_n \cdot \omega_1' \cdot \ldots \cdot \omega_m' \in F^* \)

(from this follows especially the associative rule:
\( \kappa' \cdot \kappa'' = (\kappa' \cdot \kappa) \cdot \kappa'' \))

c) \( I \cdot \kappa = \kappa \cdot I = \kappa \) for any \( \kappa \in F^* \)

We call the elements of \( F^* \) select-functions.

2.8.122 The following two functions are defined:

a) \"\( \kappa_1 \) is main-part of \( \kappa_2 \)" : \( \text{is-mp}(\kappa_1, \kappa_2) \)

\( \text{is-mp}(\kappa_1, \kappa_2) \equiv (\exists \kappa)(\kappa_2 = \kappa \cdot \kappa_1) \)

b) \"the rest-part of \( \kappa_2 \) with respect to \( \kappa_1 \)" : \( \text{rp}(\kappa_1, \kappa_2) \)

\( \kappa = \text{rp}(\kappa_1, \kappa_2) \equiv (\kappa_2 = \kappa \cdot \kappa_1) \)

(if \( \neg \text{is-mp}(\kappa_1, \kappa_2) \), then \( \text{rp}(\kappa_1, \kappa_2) \) is undefined).

2.8.123 Given an enumerable set \( EO \) of symbols, called "elementary objects" :

\[ EO = \{ e_1, \ldots \} \]

we consider the set \( O \) of pairs \( \langle S, G \rangle \), where

a) \( S \subseteq F^* \) is a finite, non-empty set, satisfying the condition:

\( (\kappa_1, \kappa_2 \in S \land \kappa_1 \neq \kappa_2 \Rightarrow \neg \text{is-mp}(\kappa_1, \kappa_2) \)

(i.e., \( S \) is a finite, non-empty subset of \( F^* \) satisfying the condition that no element of \( S \) is a main-part of another element of \( S \); especially, if \( I \in S \), then \( S = \{ I \} \))
b) \( G : S \rightarrow \text{EO} \)

(i.e. \( G \) is a mapping of \( S \) into \( \text{EO} \)).

We call the elements of \( O \) objects and identify the special objects of the form \(<\{I\}, I \rightarrow e>\) with the corresponding elementary objects \( e \) in \( \text{EO}\):

\[ e = <\{I\}, I \rightarrow e> . \]

i.e. \( \text{EO} \subset O \).

We denote the pair \(<\{\}, \text{nf}\>\), where \( \{\} \) is the empty set and \( \text{nf} \) the null-function (the function whose range of arguments is empty), by \( \Omega \). \( \Omega \) is not element of \( O \) and is called the non-existing object.

2.8.124 We now define the application of a select-function

\[ \kappa_1 \in F^* \text{ to an existing or non-existing object } b = <S,G> \text{ by} \]

\[ \kappa_1 (<S,G>) = (\exists \kappa) (\kappa \cdot \kappa_1 \in S) \rightarrow <S_1,G_1> \]

\[ T \rightarrow \Omega \]

where \( S_1 = \{ \kappa \mid \kappa \cdot \kappa_1 \in S \} \), and \( G_1 \) is defined by

\[ G_1(\kappa) = \kappa \in S_1 \rightarrow G(\kappa \cdot \kappa_1) \]

It is easily seen that:

a) \( (b = <S,G>) \supset (\forall \kappa) (\kappa \in S = \kappa(b) \in \text{EO}) \)

(i.e. if \( b = <S,G> \), then \( S \) is the set of those and only those functions which are applicable to \( b \) and yield as results elementary objects).
b) \[ \chi_1 \cdot \chi_2 (b) = \chi_1 (\chi_2 (b)) \]
   (i.e. the composition \( \cdot \) is the functional composition).

c) \[ I (b) = b \]
   (i.e. \( I \) is the identity function, which is applicable to any object).

d) \[ \{ b = \langle S, G \rangle : \chi \in S \supset (\chi (b) = G (\chi)) \} \]
   (i.e. the mapping \( G \) defines the results of the applications of the elements of \( S \) on \( b \)).

e) \[ \chi (\varnothing) = \varnothing \]
   (the application of any select-function to \( \varnothing \) gives \( \varnothing \)).

An object may be represented by a tree whose branches are labeled with elements of \( F \) and whose terminal nodes are labeled with elements of \( EO \). The members of \( S \) correspond exactly to the paths from the root to the terminal nodes, and \( G \) is represented by the mapping of these paths to their terminal nodes.

Example: The object
\[ \langle \{ f_1, f_5 \circ f_3 \circ f_2, f_3 \circ f_3 \circ f_2, f_1 \circ f_2 \} \rangle, \]
\[ (f_1 \rightarrow e_1, f_5 \circ f_3 \circ f_2 \rightarrow e_2, f_3 \circ f_3 \circ f_2 \rightarrow e_1, \]
\[ f_1 \circ f_2 \rightarrow e_2) \]
may be represented by the tree:
2.8.126 We now define the basic $\mu$-function. The function $\mu(b, \langle \kappa_1 : b_1 \rangle)$, where $b, b_1 \in O$ and $\kappa_1 \in F$, has as its denotation an object.

Let $b = \langle S, G \rangle$, $b_1 = \langle S_1, G_1 \rangle$, and

$$b_0 = \langle S_0, G_0 \rangle = \mu(b, \langle \kappa_1 : b_1 \rangle)$$

then

$$S_0 = S - \{ \kappa \mid \kappa \in S \cap (\text{is-mp}(\kappa_1, \kappa) \vee \text{is-mp}(\kappa, \kappa_1)) \} \cup \{ \kappa \mid \text{rp}(\kappa_1, \kappa) \in S_1 \cap \text{is-mp}(\kappa_1, \kappa) \}$$

$$G_0(\kappa) = \kappa \in S_0 \quad \text{(is-mp}(\kappa_1, \kappa) \rightarrow G_1(\text{rp}(\kappa_1, \kappa)), T \rightarrow G(\kappa))$$

It may be seen that this definition corresponds to the insertion of the tree representing $b_1$ in the tree representing $b$ at the node defined by $\kappa_1$, including both cases that this node is or is not present in the tree representing $b$.

2.8.127 The basic delete-function $\delta(b, \kappa_1)$ is defined as follows:

Let $b = \langle S, G \rangle$, $b_0 = \langle S_0, G_0 \rangle = \delta(b, \kappa_1)$,

then

$$S_0 = S - \{ \kappa \mid \kappa \in S \cap \text{is-mp}(\kappa_1, \kappa) \}$$

$$G_0(\kappa) = \kappa \in S_0 \rightarrow G(\kappa)$$

$\mu(b, \langle \kappa_1 : G \rangle)$ is considered to be equivalent to $\delta(b, \kappa_1)$.

2.8.128 Equality of objects:

$$(\langle S_1, G_1 \rangle = \langle S_2, G_2 \rangle) \equiv (S_1 = S_2) \& (G_1 = G_2)$$

For $\mathcal{Q}$ the convention holds that:

$$(\mathcal{Q} = \mathcal{Q}) \equiv T$$
This convention e.g. allows the proposition:

\[(b_1 = b_2) \equiv (\forall \kappa \subseteq \kappa (b_1) = \kappa (b_2))\]

2.8.129 The predicate exists is defined as

\[\text{exists}(x) = x \in O.\]

Especially is

\[\text{exists}(\emptyset) = F.\]
2.8.2 Select-functions and elementary objects occurring in this document

1. The set $F$ consists of
   
   1.1 the set of names of the form $s\text{-name}$, where 'name' is a string of small letters, hyphens and digits;
   1.2 the set of PL/I-identifiers (or the result of the mapping of PL/I identifiers into an abstract form);
   1.3 the set of elements of the lists $\mathcal{A}$, $\mathcal{B}$, $\mathcal{N}$, ... which are part of the PL/I-machine (described in chapter 3).

2. The terms $\text{elem}(i)$, $i = 1, 2, 3, ...$ are used as shorthand notation for $s\text{-first}$, $s\text{-second}$, $s\text{-third}$, ... The terms $\text{succ}(i)$, $i = 1, 2, 3, ...$ are used as shorthand notation for $s\text{-first-succ}$, $s\text{-second-succ}$, $s\text{-third-succ}$, ... The terms $\text{loc}(a, i)$, $i = 1, 2, 3, ...$, $a \in \mathcal{A}$, are used as shorthand notation for $s\text{-first-loc-a}$, $s\text{-second-loc-a}$, ... In some cases terms of the form $\text{sel}_1(\text{sel}_2)$, when $\text{sel}_1$ and $\text{sel}_2$ are members of the set $F$, are used as shorthand notation for $\text{sel}_1 - \text{sel}_2$ (another member of $F$).

3. The elements of the set $EO$ are the truth-values $T$ and $F$, special objects represented by strings of capital letters like BIN, DEC, CTR, AUTO, ..., the objects EMPTY and $<$ (the empty list), all members of the set $F^*$, and sets of objects.

2.8.3 Notational conventions for the $\mu$-functions and $\delta$-functions

The $\mu$-function is used in three forms:

1. the basic form of the $\mu$-function defined in chapter 2.8.126

2. the form $\mu(b, <x_1:b_1>, <x_2:b_2>, ..., <x_n:b_n>)$
   (containing a list of pairs $<x_i:b_i>$), which is reducible iteratively to the basic form according to the definition:
\[ \mu(b, <\kappa_1:b_1>, <\kappa_2:b_2>, \ldots <\kappa_n:b_n>) = \]
\[ \mu(\mu(b, <\kappa_1:b_1>), <\kappa_2:b_2>, \ldots <\kappa_n:b_n>) \]

Note that in general the order of the pairs \(<\kappa_1:b_1>\) is significant.

3. the form \(\mu(b, \{ <\kappa:x> \mid p(\kappa,x) \})\) (containing a set of pairs \(<\kappa_i:x_i>\)). This form is reducible to the above form in the following way: if the elements of the set of pairs are written in any linear order and used as argument in the above form 2), then the result is the result of this form provided that the ordering of pairs is not significant. If the order is significant, the function is undefined. The function is defined for the empty set of pairs: \(\mu(b,\{\}) = b\).

There are special conventions for the cases that the object in the first argument place of the \(\mu\)-function is EMPTY, or the empty list \(<>\):

- the function \(\mu(\text{EMPTY}, \ldots )\) may be written as \(\mu_0(\ldots )\),
- the function \(\mu(<> , \ldots )\) may be written as \(\mu_1(\ldots )\).

The \(\delta\)-function is used in three forms:

1. the basic form described in 2.8.127
2. the form \(\delta(b, \kappa_1, \kappa_2, \ldots \kappa_n)\)
3. the form \(\delta(b, \{\kappa \mid p(\kappa)\})\)

The reduction to the basic form is analogous to the reduction of the forms 2) and 3) of the \(\mu\)-function to its basic form.
2.8.4 Definition of classes of objects

This chapter is concerned with the definition of classes of objects, i.e. with the definition of predicates that are true for elements of certain sets of objects and only for these. Notational conventions are introduced to facilitate the definition of predicates. These conventions become important primarily for the definition of the abstract syntax of PL/I-programs and of the state of the PL/I-machine. A certain category of the abstract syntax is a certain class of objects.

The definition tools introduced so far allow the definition of predicates as shown in the following example:

\[ is\text{-}op(z) = (\exists x, y) (z = \mu_o(<s\text{-}da:x>, <s\text{-}vr:y>) \& is\text{-}da(x) \& is\text{-}vr(y)) \]

The predicate is-op applied to an object \( z \) is true, iff there exist objects \( x \) and \( y \), for which the predicates is-da and is-vr are true, respectively, such that if the selector-function \( s\text{-}da \) is applied to \( z \) the result is \( x \), and if the selector-function \( s\text{-}vr \) is applied to \( z \) the result is \( y \). In other words, the predicate is-op defines a class of objects which are composed of two parts to be selected by the selector-functions \( s\text{-}da \) and \( s\text{-}vr \), respectively, where the parts in turn belong to classes of objects defined by the predicates is-da and is-vr, respectively.

2.8.41 The form

\[ is\text{-}pred = (<\%_1; is\text{-}pred_1>, \ldots <\%_n; is\text{-}pred_n>) \]

where \( \%_1, \ldots \%_n \) are select-functions, \( pred_1, \ldots pred_n \) are predicates, has the meaning that the predicate is-pred is defined as follows:
is-pred(z) = \exists(x_1, \ldots, x_n)(\text{is-pred}_1(x_1) \& \ldots \& \text{is-pred}_n(x_n) \&
\neg z = \mu_0(<x_1:x_1>, \ldots, <x_n:x_n>))

2.8.42 The predicate is-pred-list, where is-pred is any predicate, is defined as follows: the elements of the associated class of objects are lists whose elements belong to the class of objects associated with the predicate is-pred:

is-pred-list(x) = (\exists k, x_1, \ldots, x_k)(\text{is-integer}(k) \& k \geq 0 \&
\forall i \leq k \text{ is-pred}(x_i) \&
x = \mu_0(\{<\text{elem}(i):x_i> | 1 \leq i \leq k\})).

2.8.43 The predicate is-pred-tree, where is-pred is any predicate, is defined as follows: the elements of the associated class of objects are trees whose successors belong to the class of objects associated with the predicate is-pred:

is-pred-tree(x) = (\exists k, x_1, \ldots, x_k)(\text{is-integer}(k) \& k \geq 0 \&
\forall i \leq k \text{ is-pred}(x_i) \&
x = \mu_0(\{<\text{succ}(i):x_i> | 1 \leq i \leq k\})

2.8.44 The predicate is-pred-set, where is-pred is any predicate, is defined as follows: the elements of the associated class of objects are finite sets (possibly empty) whose elements belong to the class of objects associated with the predicate is-pred.
2.8.45 The form

\[ is-pred = \{<\kappa: is-pred_1 | is-pred_2(\kappa)>\} \]

has the meaning that the predicate is-pred is defined as follows:

\[ is-pred(z) = (\exists K)(\forall \kappa)((\exists \kappa \in K)(\exists \kappa_1(z))) \]

\[ z = \mu_0(\{<\kappa_1: \kappa \in K>\}) \]

i.e., the elements of the class of objects associated with is-pred are composed of a finite number of objects belonging to the class associated with is-pred_1, where these parts are selected by select-functions belonging to the class associated with is-pred_2.

2.8.46 Notation for predicates and variables

Predicates are written in the form is-name, where 'name' is any string of small letters, digits and hyphens. An exception is the predicate exists.

Given a predicate is-name, then 'name' is normally used as a variable name for the associated class of objects. (So, for instance gen is used as the variable name for the class of generations, which is defined by the predicate is-gen.)

Special variables are:

- $\xi$ for the state of the PL/I-machine
- $\%$ for select-functions
- $i,j,k$ for integers
- $r,x,y,z$ for objects of any class
- $t$ for PL/I-text.
- $\eta$ for lists of integers
2.8.47 Notation and operations for special classes of objects

2.8.471 Lists

a) If \( l \) is a list, then the length of \( l \) is defined as

\[
\text{length}(l) = (l = <>). O \\
T \rightarrow (\exists i) (\exists \text{elem}(i, l) \& \neg \exists \text{elem}(i+1, l))
\]

b) The functions head and tail are defined for lists as:

\[
\text{head}(l) = \text{elem}(1, l) \\
\text{tail}(l) = (l \neq <>). \mu_1(\{\text{elem}(i) : \text{elem}(i, l) \} \cup \{\text{elem}(i) : \text{elem}(i+1, l) \})
\]

c) The operation concatenation is defined for lists as:

\[
l_1 \cdot l_2 = \mu_1(\{\text{elem}(i) : \text{elem}(i, l_1) \} \cup \{\text{elem}(i) : \text{elem}(i, l_2) \} \mid 1 \leq i \leq \text{length}(l_1) \} \cup \{\text{elem}(i) : \text{elem}(i, l_2) \} \mid 1 \leq i \leq \text{length}(l_2) \})
\]

d) A special notation is defined for lists:

\(<> \) is called the empty list, and

\(<b_1, \ldots, b_n> = \mu_1(\{\text{elem}(i) : b_1 \}, \ldots, \{\text{elem}(n) : b_n \})
2.8.472 Pairs and Triples

A pair is a list with two elements:

$$\text{is-pair}(x) = \text{is-list}(x) \land \text{length}(x) = 2$$

A triple is a list with three elements:

$$\text{is-triple}(x) = \text{is-list}(x) \land \text{length}(x) = 3$$

2.8.473 Integers

An integer is an operand of the class (see chapter on PL/I-operations):

$$\text{is-integer} = (<\text{da}: (<\text{base:is-dec}, <\text{scale:is-fix}>,
    <\text{mode:is-real}, <\text{prec:is-def-prec}>),
    <\text{vr:is-vr}>$$

i.e. a PL/I real, decimal, fixed operand with default precision.

Decimal digits are used as shorthand notation for integers. The operations +, -, *, / are defined for these objects with conventional meaning.
3. THE PL/I MACHINE

The PL/I machine is specified by the description of the states that the machine can assume and a state transition function, the language function $\Lambda$. PL/I requires that the order of execution of certain actions be undefined. This situation is modeled by the specific property of the transition function, namely being a mapping from states into sets of states.

The programs to be executed by the machine as well as the states are considered to be abstract objects. The description of these objects is called the abstract syntax of text and of states respectively. By abstract syntax is meant the description of classes of objects and relation between these classes.

The abstract syntax of state together with comments to the particular components of states are given in chapter 3.1.

Chapter 3.2 defines the notion of computation for the machine and introduces the notational conventions adopted for the definition of the state transitions.

The PL/I machine and its computation as specified in the central part of the present document defines the execution of one single task. This is to mean that the definition is valid only under the assumption that the task being considered is not influenced by the execution of parallel tasks. However, the attempt is made in chapter 7 to extend the considerations to the notion of parallel execution of tasks. Specifically the interaction of parallel tasks as far as defined by PL/I is considered in that chapter.
3.1 The Constituents of the State of the PL/I Machine

3.1.1 Survey of the constituents of the state

This chapter describes the overall structure of the state of the PL/I machine. A comment to each particular component of the state will be given in the subsequent chapters.

Abstract Syntax

\[
\text{is-state} = \\
\quad (<s-a : is-a-list>, 1) \\
\quad (<s-b : is-b-list>, 1) \\
\quad (<s-ab : is-ab-list>, 1) \\
\quad (<s-n : is-n-list>, 1) \\
\quad (<s-f : is-f-list>, 1) \\
\quad (<\text{s-s} : (\{ <\text{loc}(a,i) : (<\text{n-value: is-va}) | <\text{n-contents: is-gen-list}) | \\
\quad \quad (\text{is-ab(a) vis-a(a)) & is-integer(i) } \}) >, \\
\quad (<\text{s-ag} : (\{ <\text{b: is-gen-list} | \text{is-b(b) } \}) >, \\
\quad (<\text{s-at} : (\{ <\text{n} : (<\text{s-attr: is-attr}, <\text{e: is-e}) \ | \text{is-n(n) } \}) >, \\
\quad (<\text{s-dn} : (\{ <\text{n} : (<\text{s-den: is-den}) \ | \text{is-n(n) } \}) >, \\
\quad (<\text{s-es} : is-es), \\
\quad (<\text{s-fd} : is-fd), \\
\quad (<\text{s-fu} : is-fu), \\
\quad (<\text{s-e} : is-e), \\
\quad (<\text{s-d} : is-d), \\
\quad (<\text{s-cs} : is-cs), \\
\quad (<\text{s-ri} : is-ri), \\
\quad (<\text{s-id} : is-name-set), \\
\quad (<\text{s-c} : is-c>)
\]

1) The suffix "list" in the above cases means an enumerably infinite list while in all other places of the document the suffix "list" means a finite list.
For convenience the following abbreviations and terms for the components of a given state have been adopted:

<table>
<thead>
<tr>
<th>component</th>
<th>abbreviation</th>
<th>term</th>
</tr>
</thead>
<tbody>
<tr>
<td>s-a (s)</td>
<td>A</td>
<td>list of free addresses</td>
</tr>
<tr>
<td>s-b (s)</td>
<td>B</td>
<td>list of free aggregate names</td>
</tr>
<tr>
<td>s-ab(s)</td>
<td>Ab</td>
<td>list of free based addresses</td>
</tr>
<tr>
<td>s-n (s)</td>
<td>N</td>
<td>list of free unique names</td>
</tr>
<tr>
<td>s-f (s)</td>
<td>F</td>
<td>list of free file names</td>
</tr>
<tr>
<td>s-s (s)</td>
<td>S</td>
<td>storage</td>
</tr>
<tr>
<td>s-ag(s)</td>
<td>AG</td>
<td>aggregate directory</td>
</tr>
<tr>
<td>s-at(s)</td>
<td>AT</td>
<td>attribute directory</td>
</tr>
<tr>
<td>s-dn(s)</td>
<td>DN</td>
<td>denotation directory</td>
</tr>
<tr>
<td>s-es(s)</td>
<td>ES</td>
<td>external storage</td>
</tr>
<tr>
<td>s-fd(s)</td>
<td>FD</td>
<td>file directory</td>
</tr>
<tr>
<td>s-fu(s)</td>
<td>FU</td>
<td>file-union-directory</td>
</tr>
<tr>
<td>s-e (s)</td>
<td>E</td>
<td>environment</td>
</tr>
<tr>
<td>s-d (s)</td>
<td>D</td>
<td>dump</td>
</tr>
<tr>
<td>s-cs(s)</td>
<td>CS</td>
<td>condition status</td>
</tr>
<tr>
<td>s-ri(s)</td>
<td>RI</td>
<td>return information</td>
</tr>
<tr>
<td>s-id(s)</td>
<td>ID</td>
<td>set of declared identifiers</td>
</tr>
<tr>
<td>s-c (s)</td>
<td>C</td>
<td>control</td>
</tr>
</tbody>
</table>
3.1.2 The name lists $A, A_b, B, N, F$

It is frequently the case that certain objects (representing a piece of information) are to be available in two or more parts of the state. If no updating of the information can occur during the interpretation it is sufficient to copy the object representing the information into those parts of the state. If, however, it is possible that the information might be updated during the interpretation, then the object representing the information is named uniquely and made available in some part of the state under this name. Instead of having a copy of the object it is referred to via this name.

There are five enumerable infinite lists of names $A, A_b, B, N,$ and $F,$ whose elements are used to name objects in the course of interpretation. The elements of a list are mutually different. Furthermore, no element of $A$ is also an element of $A_b$, i.e. one can always decide whether a name is element of $A$ or of $A_b$. The five lists correspond to five classes of objects that have to be named during interpretation.

The lists $A, A_b, B, N$ and $F$ of a certain state of a computation contain those names of the corresponding lists $A, A_b, ..., F$ that have not been used in preceding states in the computation.

3.1.3 Block activation and scope of names

There are six constituents of the state that are associated with specific block activations, namely $E, D, CS, RI, ID$ and $C$.

The following sections describe in detail the significance of each of the components of the state associated with a specific block activation.
3.1.31 The dump $D$

Abstract Syntax

$$is-d = (\langle s-e:is-e\rangle, \langle s-cs:is-cs\rangle, \langle s-ri:is-ri\rangle, \langle s-id:is-name-set\rangle, \langle s-c:is-c\rangle, \langle s-d:is-d \mid is-empty\rangle)$$

**Note:** The above abstract syntax describes the structure of the dump. The dump may be considered to be a stack where the components selected by $s-e$, $s-cs$, $s-ri$, $s-id$ and $s-c$ form the top element and the component selected by $s-d$ is again a stack (or empty).

The dump keeps track of the different levels of block activations.

Block activations are nested in the sense that if a block activation is established in the course of interpretation it is terminated only after all other block activations that were established later are terminated.

To establish a new block activation means to put the above mentioned six components of the current state on top of a stack which is called the dump $D$. The other five components $E, CS, RI, ID, C$ are redefined according to the block being activated. The stack $D$ corresponds
to the nested structure of the block activations. If a block activation is to be terminated it must be the current block activation according to the nested structure of block activations. To terminate a block activation means to redefine $E, CS, RI, ID, D$ and possibly $C$ according to the top element of the stack $D$ of the current state. The exceptions for $C$ are the termination of a block activation because of a goto statement in which case the control is redefined according to the denotation of the label and the termination of a block activation because of a return statement with expression in which case the control (in the top of the stack) must be updated with the value returned.

3.1.32 The environment $E$

**Abstract Syntax**

$$is-e = \{ <id:is-n> \mid is-name(id) \}$$

The environment $E$ of the current state contains all identifiers known to the current block activation. Every time an identifier is supplied with a new meaning it is associated with a name unique to the task, and the meaning is kept under the unique name in the state of the PL/I machine. More specific, the unique name is in general associated with a so-called denotation in the denotation directory $DN$ and with attributes in the attribute directory $AT$.

Any time a new block activation is established the identifiers declared in the corresponding block are associated with a unique name and via this unique name with their meaning. The meaning attached to the unique name remains constant throughout a task. A specific
environment is then to be updated with the newly declared identifiers and associated unique names. To update an environment with an identifier and the associated unique name means to create a new component of the environment if the identifier is not yet present and to replace the respective component if the identifier is present. It is important to note that all identifiers declared in a block are associated with newly created unique names at block entry.

This specific and updated environment remains unchanged throughout the block activation and is deleted upon termination of the associated block activation.

The specific environment that is updated is in case of begin blocks that of the dynamically preceding block activation, in the case of a block activation caused by a procedure call the environment which is part of the current denotation of the procedure identifier.

The latter environment is a copy of the environment that was current at the time the denotation was established.

This step of indirection that leads from a given identifier through the unique name associated in the environment to the meaning of the identifier is indeed necessary, since the denotation of a procedure name contains an environment which again contains the procedure identifier and the associated unique name. The process of replacing the unique names by their associated denotations and attributes in the environment would therefore in general fail to terminate.

Due to the association of unique names it is unnecessary to represent the nesting of block activations in the directories.
3.1.33 The set of declared identifiers ID

Abstract Syntax

\[ \text{is-name-set}(ID) \]

The set ID associated with a block activation is the set of identifiers declared in the corresponding block. It is created at the time the block is activated and deleted upon termination. The set is used for the termination of the block activation.

3.1.34 The return information RI

Abstract Syntax

\[ \text{is-ri} = (\langle \text{is-ret-point} : \text{is-ret-point} \rangle, \]
\[ \langle \text{is-ret-type} : \text{is-ret-type} \rangle) \]

\text{is-ret-point} \ldots \text{pointer to a place in the control}
\text{is-ret-type} \ldots \text{as defined in the abstract syntax App I}

The return information is only significant in block activations that are caused by function calls. The first component of the return information specifies where the result of the function is to be placed, the second component specifies the return type to which the result is to be converted. The return information is established when the block is activated.
3.1.35 The condition status $CS$

Abstract Syntax:

$$is-cs = (<s-bps:is-cond-part>,\n<s-sps:is-cond-part>,\n<s-cds:({ <s(cond-sel):(<s-snap:is-opt>,\n<s-den:is-den>) \mid \n\text{is-cond-sel}(\text{cond-sel}) }>)$$

$$\text{is-cond-sel} = (\text{is-name} \mid \text{is-simple-check-ref} \mid \text{is-io-sel} \mid \text{is-n-io-cond})$$

$$\text{is-simple-check-ref} = (<s-id:is-name>,<s-qual-list:is-name-list>)$$

$$\text{is-n-io-cond} = (\text{is-conv} \mid \text{is-fofl} \mid \text{is-ofl} \mid \text{is-size} \mid \text{is-subrg} \mid \text{is-ufl} \mid \text{is-zdiv} \mid \text{is-area-size} \mid \text{is-error} \mid \text{is-finish})$$

$$\text{is-io-sel} = (<s-first:is-name>,\n<s-second:is-io-cond>)$$

$CS$ keeps the current information on condition prefixes and condition denotations. Condition actions are controlled by condition prefixes established for blocks and executable statements and by condition denotations established either as system action or by the execution of ON and REVERT statements.

Condition prefixes exist only for part of the conditions. There exists an initial set of prefixes as part of $CS_0$, which is modified throughout a program by block and statement prefixes. The validity of a prefix follows normal scope rules. A prefix controlled condition is raised only if the corresponding situation occurs and the condition is in scope of an enabling prefix. In a block the block prefixes control the condition enabling situation for the block activation, i.e., for the expression evaluation before the interpretation of the body. At block entry the complete set of prefixes of the block be-
comes the value of the block prefix part of the current CS. On interpretation of executable statements the respective statement prefixes become the value of the statement prefix part of the current CS.

The condition denotations make up the third part of CS. The denotations are inherited from the dynamically preceding block and are modified by the execution of ON and REVERT statements.

The various situations leading to the raising of a condition can only partly be defined, since for computational conditions the condition situation can not be expressed in PL/I.

If a condition is raised, the subsequent action is determined by the condition denotation. Again, an initial denotation for conditions is defined in CS. The interpretation of the action specified for a condition is performed similar to a procedure call.

3.4.36 The control C

In any state the control contains the instruction to be executed. Each block activation has its own level of control that is established when a block is activated and deleted upon termination of a block activation. Actually the last instruction contained in the control of a block activation is the instruction that performs the termination of the block activation. A detailed description of the structure and function of the control is given in chapter 3.2.
3.1.4 Attribute directory AT and denotation directory DN

Abstract Syntax

The structure of the attribute directory is:

\[
\{ \langle n : (s-attr:is-attr), (s-e:is-e) \mid is-n(n) \rangle \}\)

The structure of the denotation directory is:

\[
\{ \langle n : (s-den:is-den) \rangle \}
\]

is-attr are either attributes is-attr as defined in the abstract syntax (see App.I) or evaluated attributes i.e. attributes where expressions describing upper bounds, lower bounds and length are replaced by their respective values.

is-den are the possible denotations that can be associated with unique names. A detailed case distinction is given later in this section.

Every unique name newly created for an identifier at block entry is associated in AT with the attributes or evaluated attributes of the respective declaration of the identifier and the environment of the respective block activation (see abstract syntax App.I under is-decl).

The associated environment is necessary in those cases where the attributes have to be evaluated possibly in another block activation. In this case the environment associated with the attributes is used for their evaluation.

The denotation directory associates unique names with denotations. The structure of the denotation depends on the category of the name.
In the following the necessary case distinctions are made.

3.1.41 Procedures

The structure of a procedure denotation is:

\[(\text{<s-body:is-body>},\text{<s-e:is-e>})\]

where

- is-body as defined in the abstract syntax App.I
- is-e an environment

The environment is the environment of the activation of the block where the procedure has been declared. It is this environment under which the procedure is activated. For internal procedures the body is that found under s-den in the declaration of the procedure. For external procedures the prepass has entered the body of the external procedure in DN under a specific unique name. This name is found under s-den in any of the declarations of the external procedure, i.e. the body in the procedure denotation is a copy of the body found in DN under the unique name.

The exception for external procedures is necessary because the process of substituting the body of an external procedure into all the declarations of the external identifier is in general not terminating.

3.1.42 Label-constants

The structure of a label denotation is:

\[(\text{<s-st-list:is-st-list>},\text{<s-evs:is-evs>})\]
In the abstract syntax of program it is assumed that each label constant that occurs internal to a block has got a declaration in that block. It is further assumed that this declaration contains under s-den the statement list associated with the label (i.e., the list of statements to be executed beginning with the labeled statement up to the end of the block to which the label is internal).

The first component of a label denotation is this statement list. The second component is an identification of the activation of the block in which the label is declared. The second component is used in the interpretation of a goto statement to find the block activation to which the goto has to return.

3.1.43 Proper variables

The denotation of a proper variable is an aggregate-name i.e. a name out of the list \( B \). For static and controlled variables (whether internal or external) this aggregate name has been created by the pre-pass before the proper execution of the program. It is found under s-den in the corresponding declarations.

For automatic variables the aggregate name is newly created at entry into the block where the variable is declared.

3.1.44 Based and defined variables

The denotation of based and defined variables is empty.

3.1.45 Files

The denotation of a file is an aggregate name. For external files this aggregate name is created by the prepass and found in the declaration under s-den. For internal files this aggregate name is created at entry into the block where the file is declared. Further details on files are found in chapter 3.17.
3.1.46 Parameters

The denotation of a parameter depends on the kind of argument associated and the kind of attributes of the parameter.

The evaluated arguments passed at a procedure or function call are objects composed of two parts namely attributes and argument denotations. The attributes of the argument are matched against the attributes of the parameter and the denotation of the parameter in general becomes the denotation of the argument.

3.1.47 Built-in functions

Built-in functions do not have denotations. It is assumed, and for some of the built-in functions actually so defined, that the definition of the instruction that interprets built-in functions is self-contained and does not refer to the state to get the meaning of the built-in function. It is therefore sufficient that a unique name corresponding to a built-in function has the attribute BUILT IN in AT.

3.1.5 Storage $S$

The structure of storage is:

$$\{ <\text{loc}(a,i): ((<\text{s-value}\text{-is-vr}>) | <\text{s-contents}\text{-is-ge-list}>) | (\text{is-ab}(a) \lor \text{is-a}(a)) \& \text{is-integer}(i) ) \}$$

- is-vr: admissible elementary PL/I value representation
- is-gen-list: list of generations (see 3.16)
- is-a: the elements of $A$
- is-ab: the elements of $A_b$
A pair <a,i> where a is an element of the list \( A \) or \( A_b \) and i is an integer may be considered as an elementary address of the storage of the PL/I machine.

The elementary addresses of the storage of the PL/I machine are considered to be partially ordered. While in the definition no essential use is made of the order of elements in \( A \) and \( A_b \) the natural ordering of indices is significant. The ordering of these indices is used to model the concept of contiguity of storage in PL/I.

The structure given to the storage of the PL/I machine has two levels as shown in the abstract syntax above, i.e. the value that may be associated with an elementary address <a,i> is selected by the composition of two select functions

\[
\text{loc}(a, i) \cdot \text{s-value} \quad \text{or} \quad \text{loc}(a, i) \cdot \text{s-contents}.
\]

This peculiar structure was designed to make it possible to distinguish between non-allocated storage and allocated storage that has not yet been initialized.

The distinction is made by:

- non-allocated <a,i> : \( \neg \exists \text{loc}(a, i)_S \)
- allocated, but not initialized:
  - \( \exists \text{loc}(a, i)_S \) &
  - \( \neg \exists (s\text{-value}\cdot \text{loc}(a, i)_S) \) &
  - \( \neg \exists (s\text{-contents}\cdot \text{loc}(a, i)_S) \)

The assignment of an elementary value to an elementary address is so designed that it is undefined if the associated address has not been allocated.
The selector s-contents is exclusively used to select the contents of an area (which is a list of generations). Thus it is possible to distinguish between addresses associated with areas and other addresses by:

area: \( \neg \exists \text{s-value}\cdot \text{loc}(a,i) \)  
not area: \( \neg \exists \text{s-contents}\cdot \text{loc}(a,i) \)

This distinction is important for the freeing of storage.

The elements of the list \( A_b \) are exclusively used for based storage. Thus it is possible to distinguish between based storage and non-based storage by inspecting the address, i.e.:

based storage: \( \text{is-ab}(a) \)  
non based storage: \( \text{is-a}(a) \)

Note, that it was assumed that no element of \( A \) is also element of \( A_b \) (chapter 3.1.2)

The basic philosophy for the values is as follows. What is contained in storage is the representation of a value. It is not possible to interpret the representation of a value as such but only with the help of an appropriate set of attributes. There exists a function that given a representation of a value and an appropriate set of attributes yields an object which is the interpretation of the representation (see 4.3.3).

The value of an elementary storage location may only be the representation of elements of bit and character strings and, with this exception, scalar values of PL/I. The selector s-contents yields, if anything at all, a list of generations (see 3.1.6).
For convenience in the definition for certain types of values it has been assumed, that the selector s-value applied to a location loc(a,i)(S) yields directly an object. These types are the following:

1. labels: (s-st-list:is-st-list), (s-evs:is-evs)
2. pointers: (s-da:is-da),
   <s-addr:is-a v is-ab>,
   <s-dens:is-dens>,
   <s-index:is-integer> | is-null
3. complex numbers: there are two selectors applicable, s-first and s-second which select the representation of the real part and of the imaginary part respectively.

3.1.6 The aggregate directory AG

Abstract Syntax

The structure of the aggregate directory is:

{ 
  { <b:is-gen-list> | is-b(b) } 
}

is-gen = (s-da:is-da),
   <s-dens:is-dens>,
   <s-addr:is-a v is-ab>,
   <s-il:is-integer-list>)

As described in chapter 3.1.4 the denotation of a proper variable is an aggregate name, i.e. an element of list B. The aggregate directory associates these aggregate names with lists of generations, called the generations of the proper variables. A generation is created any time an allocation is performed and put on top of the generation list associated with the appropriate aggregate name, i.e. the head of the
generation list is always the most recent generation. For controlled variables the head of the generation list is deleted upon freeing of the controlled variable.

The notion of a generation is a central one for the entire document. A generation according to the abstract syntax given above has four components: an evaluated data-attribute(s-da), a density (s-dens), an address(s-addr) and an index list(s-il) which is a list of integers.

The address together with any one of the integers in the index list forms elementary addresses, the elementary addresses of the piece of storage corresponding to the given generation.

The evaluated data attributes associated with a generation serve two purposes. Firstly, the data attributes give the necessary information that makes it possible to interpret the values or contents of the elementary addresses. Secondly, while the index list provides a linear ordering of the elementary parts, the data attributes supply a hierarchical order of elements.

In fact there are three criteria relevant for the ordering of elements in a generation. Firstly, the linear ordering of elementary parts to which the index list corresponds, secondly the linear ordering of scalar parts where bit and character strings are taken as single elements, and the hierarchical ordering that corresponds to the data-attributes of a given generation. The term reference-list refers always to the latter principle of orderings.

The density (either packed or aligned) is in fact a necessary input to the storage mapping function of an implementation (which is not describable in terms of this document). The density has therefore no significance in this document except in some cases for the matching of attributes.
There are two important properties of a generation with respect to the associated storage namely connected and level-one.

A generation is called connected if the elements of the associated index list are serial. This property models the notion of contiguity of storage.

Let "a" be the address associated with a generation then the generation is called level-one if there exists the location (a,i) in storage only if i is an element of the index list of the generation.

A pointer, whose structure has been given in the previous chapter identifies a connected generation.
3.1.7 Parts of the State for Input and Output

There are four constituents of the state that are associated with input and output:

- the list of free file unions \( F \),
- the file directory \( FD \),
- the file union directory \( FU \), and
- the external storage \( ES \).

For completeness it should be noted that get- and put-statements with a string-option do not refer to the above constituents of the state.

3.1.71 The list of free file unions \( F \)

Elements of \( F \) are used as a link between corresponding entries in \( FD \) and \( FU \).

The organization of \( F \) is described in 3.1.2.

3.1.72 The file directory \( FD \)

Abstract Syntax:

\[
\begin{align*}
\text{is-fd} & = (\langle \text{s-nfa:is-fa}\rangle,\langle \text{s-ofa:is-fa}\rangle) \\
\text{is-fa} & = (\{\text{b:is-f} \mid \text{is-b(b)}\})
\end{align*}
\]

Entries in the file directory \( FD \) are made whenever a filename is associated with a data set by opening the filename. In this case a file association \( \text{is-fa} \) is appended to \( \text{s-nfa} \). This file association stays valid as long as no closing is executed. Closing causes the dissociation of the filename from the data set and the deletion of the file association.
Note: Opening and closing is achieved by the execution of the instructions \texttt{int-open-st()} and \texttt{int-close-st()}. However, the instructions do not in every case result in opening or closing.

Attaching of a task causes the creation of a new file directory \( \text{FD}' \) which is returned by the instruction \texttt{int-copy-fd(s-nfa(FD), s-ofa(FD))}.

Definition:

\[
\text{int-copy-fd}(\text{new,old}) = \\
\text{RETURNS: } \mu_{\aleph_0} \{ \langle \kappa \cdot s \cdot \text{ofa}: x \rangle \mid \text{exists} \cdot \kappa(\text{old}) \land x = \kappa(\text{old}) \lor \text{exists} \cdot \kappa(\text{new}) \land x = \kappa(\text{new}) \}
\]

\[3.1.73\] The file union directory \( \text{FU} \)

Abstract Syntax:

\[
is-fu = \{ \langle f : is-fu-elem > | is-f(f) \}
\]
\[
is-fu-elem = \langle s \cdot \text{ddname}: is-ddname>, \langle s \cdot \text{csa}: is-complete-file-attr>, \langle s \cdot \text{count}: (is-integer | is-\Omega)>, \langle s \cdot \text{locate-exec}: (is-keyed | is-unkeyed | is-\Omega)>, \langle s \cdot \text{bu}: (is-gen | is-\Omega)>, \langle s \cdot \text{kbu}: (is-gen | is-\Omega)>, \langle s \cdot \text{locking}: is-locking \rangle
\]
\[
is-locking = \{ \langle \text{un-task-ident}: is-key-set | un-task-ident \in \text{TA} \}
\]
The file union directory FU contains all entries needed for a filename association which are under complete control of the interpreter.

The different parts are used as follows:

- **s-ddname**: provides a link to the external storage ES;
- **s-csa**: gives the complete set of file attributes originating from opening;
- **s-count**: is used for data-directed get- and put-statements only and provides the current number of data items;
- **s-locate-exec**: is used for signaling to a WRITE or LOCATE statement that the record in the buffer has to be written into the output data set; the same is used for closing.
- **s-bu**: contains the generation of the buffer;
- **s-kbu**: contains the generation of the key-part of the buffer;
- **s-locking**: contains for each task under the unique task identification all keys of those records which are locked by that task.

The predicate is-ddname is implementation-defined.
3.1.74 The external storage ES

Abstract Syntax:

\[
\text{is-es} = \{ \langle \text{ddname:is-ext-stor} \rangle \ \text{is-ddname}(\text{ddname}) \}
\]

\[
\text{is-ext-stor} = (\text{is-dataset} \ \text{is-undf-dataset})
\]

\[
\text{is-dataset} = (\langle \text{s-data:is-data-elem-list} \rangle,
\langle \text{s-ext-contr:is-ext-contr} \rangle)
\]

\[
\text{is-data-elem} = (\langle \text{s-key:(is-key} \ \text{is-Ω}) \rangle,
\langle \text{s-rec-length:(is-integer} \ \text{is-Ω}) \rangle,
\langle \text{s-datum:is-datum} \rangle)
\]

\[
\text{is-datum} = (\text{is-char-datum} \ \text{is-record-datum} \ \text{is-printable-datum})
\]

\[
\text{is-printable-datum} = (\langle \text{s-page:is-integer} \rangle,
\langle \text{s-line:is-integer} \rangle,
\langle \text{s-char:is-integer} \rangle,
\langle \text{s-character:is-char-datum} \rangle)
\]

\[
\text{is-ext-contr} = (\langle \text{s-header:(is-user-header-label} \ \text{is-Ω}) \rangle,
\langle \text{s-trailer:(is-user-trailer-label} \ \text{is-Ω}) \rangle,
\langle \text{s-next:(is-integer} \ \text{is-Ω}) \rangle,
\langle \text{s-end-file:(is-empty} \ \text{is-Ω}) \rangle,
\langle \text{s-curr-page:(is-integer} \ \text{is-Ω}) \rangle,
\langle \text{s-curr-line:(is-integer} \ \text{is-Ω}) \rangle,
\langle \text{s-curr-char:(is-integer} \ \text{is-Ω}) \rangle,
\langle \text{s-pagesize:(is-integer} \ \text{is-Ω}) \rangle,
\langle \text{s-linesize:(is-integer} \ \text{is-Ω}) \rangle,
\langle \text{s-end-page:is-empty} \ \text{is-Ω}) \rangle)
\]

In contrast to the other constituents of the state the external storage ES is not under complete control of the interpreter. However, changes of the external storage coming from elsewhere may cause not intended actions of the interpreter in the moment the interpreter references the changed parts.

In the following the use of the different parts of the external storage ES will be described; also it will be tried to give a motivation of the structure specified above.
The external storage ES is the last step in a chain of parts of the state which is inspected whenever an associated and united filename accesses the data set. Beginning with the filename specified in a file-option, the chain starts with the environment E giving the unique filename, continues with the denotation directory DN, the file directory FD, the file union directory FU, and ends with the external storage (is-ext-stor) selected by ddname. For every ddname does an entry exist in the external storage ES but it can be an entry for an undefined data set (is-undf-dataset) for which opening cannot be performed. The predicate is-undf-dataset is implementation-defined. If the predicate is-dataset applies the selectors s-data and s-ext-contr provide the data part of the data set and additional control information, respectively.

The data part (is-data-elem-list) is organized sequentially, the elements of the sequence having an optional key, an optional record length, and data (is-datum). To the data either apply the implementation-defined predicates is-char-datum (used in stream, non print transmission), or is-record-datum (used in record transmission), or is-printable-datum (used in stream, print transmission). In the latter case data have associated all information needed for exact positioning.

The additional control information selected by s-ext-contr is optional. The selectors s-header and s-trailer yield the user header and user trailer label (the predicates are implementation-defined).

The current position in the data set, i.e. the current data-elem in the data-elem-list is indicated by the selector s-next.

Whether a data set is in the end of file status or not is specified by the selector s-end-file. The data set gets in the end of file status if the integer selected by s-next is outside of the length of the data-elem-list.
The remaining selectors are used in stream, print transmission only. The selectors s-pagesize and s-linesize yield the maximum pagesize and linesize, respectively (entries originating from opening); the selectors s-curr-page, s-curr-line, and s-curr-char yield the exact position of the next datum. The selector s-end-page makes sure that the endpage condition is raised only once per page.
3.2 State Transitions and Control of the PL/I-Machine

3.2.1 The computation of the PL/I-machine

3.2.11 The state $\xi$ of the PL/I-machine is an object as described in the previous chapter 3.1. The control part $C = s-c(\xi)$ of the state is of special significance for the definition of the changes of the state of the machine. It is described in detail in 3.2.2. For the definition of the computation of the machine it suffices to assume that:

a) there is a function $T$ which, if applied to $C$ gives the set of exactly those selector-functions (called instruction locations) which, if applied to $C$ give an instruction which is a candidate for execution.

b) there is a function $Q$ which, if applied to a member $\tau$ of $T(C)$ gives the selector-function $\nu$ which denotes the location in $C$ in which the value defined by the instruction $\tau(C)$ is to be inserted.

c) $C$ may be $\varnothing$.

3.2.12 The application of an instruction $\text{instr}$ to a state $\xi$ and an instruction location $\tau$ gives a pair of objects $<\xi',\nu>$, whose first part $\xi'$ is a state and whose second part $\nu$ is a value:

$$\text{instr}(\xi,\tau) = <\xi',\nu>$$

3.2.13 The function $\Phi(\xi,\text{instr},\tau)$ is defined as:

$$\Phi(\xi,\text{instr},\tau) = \mu(\text{first}(\text{instr}(\xi,\tau)), <Q(\tau)\circ s-c: \text{second}(\text{instr}(\xi,\tau))>),$$

i.e. given a state, an instruction and an instruction location the function defines a new state, which is obtained from the state (defined by application of instr to $\xi$), by inserting the value (defined by application of instr to $\xi$) at the value-return-point $\nu = Q(\tau)$. 
3.2.14 The function $\Psi(\xi, \tau)$ is defined as:

$$\Psi(\xi, \tau) = \Phi(\delta(\xi, \tau), \tau \circ s \circ (\xi), \tau)$$

i.e. it defines the arguments of the function $\Phi$ to be the state $\xi$ with the instruction deleted at the location $\tau$, the instruction at the location $\tau$, and $\tau$.

3.2.15 The language-function $\Lambda(\xi_1)$ is defined as:

$$\Lambda(\xi_1) = \{\xi \mid \xi = \Psi(\xi, \tau) \& \tau \in t(s \circ (\xi))\}$$

3.2.16 A computation of the machine is a finite sequence of states

$$\xi_0, \xi_1, \ldots \xi_n$$

where $\xi_0$ is the "initial state", $\xi_n$ is the "end state", and where

$$\xi_i \in \Lambda(\xi_{i-1}), i = 1, \ldots, n.$$ 

The computation is "successfully terminated" if $s \circ (\xi_n)$.

3.2.17 The functions $\Phi$, $\Psi$ and the instructions are partial functions, i.e. they are not defined for all possible states $\xi$. We consider the set $\text{COMP}(\xi_0)$ of all possible successful computations with initial state $\xi_0$ and call a computation $\text{comp}'$ an initial part of a computation $\text{comp}$ if

$$\text{comp}' = \xi_0, \xi_1, \ldots \xi_m$$

$$\text{comp}'' = \xi_1, \xi_2, \ldots \xi_m, \xi_{m+1}, \ldots \xi_n$$

and $n \geq m$. 
A program is called valid if it defines an initial state $s_0$ such that any possible computation with initial state $s_0$ is initial part of a computation in the set $\text{COMP}(s_0)$.

Note: There are two kinds of undefinedness in the present document. First, there are functions used in the definitions of instructions that are not defined, or only partially defined. These functions are called "implementation-defined functions". Second, the definition gives, for a valid program, in general not a single end state, but a set of end states (provided that all implementation-defined functions are specified).

An implementation is considered as correct with respect to a certain valid program, if the execution of the program leads to a state which maps into one of the members of the set of possible end-states (provided by the definition if the implementation defined functions are specified). An implementation is correct if it is correct with respect to any valid program.

It is not guaranteed that a relation can be established between the execution of an invalid program in an implementation and a computation associated with this program. Specifically, there is not necessarily a correlation between the end-state of an unsuccessfully terminated computation and the result of the execution of the corresponding program in an implementation.

### 3.2.2 The control-part of the PL/I-machine

An instruction is an object specified by an instruction name and a number of objects (the "arguments" of the instruction). The control part $C$ of the PL/I-machine may informally be described as a tree whose nodes are instructions.
For the definition of the abstract syntax of the control we define a special set $\Sigma^*$ of selector-functions. Let $\Sigma$ be the set of names

$$\Sigma = \{\text{succ}(x) \mid \text{is-name}(x)\}$$

then $\Sigma^*$ is the free semigroup with identity with base $\Sigma$ and composition "*" (see 2.8.12).

Abstract syntax of the control:

$$\text{is-c} = (\langle \text{instr-name:is-name},\rangle,$$
$$\{\langle f:\text{is-object} \rangle \mid \text{is-name}(f) \& f \notin \Sigma^*\},$$
$$\{\langle \text{succ}(g):\text{is-c} \rangle \mid \text{is-name}(g)\} \mid \text{is-Q}$$

i.e. a control specifies an instruction name, a set of objects (the arguments of this instruction) and a set of "successor-controls" (possibly the empty set).

Those successor controls whose set of successor controls is empty specify exactly one instruction. It is the set of instructions which are at these locations which are candidates for execution (i.e. the instructions at the terminal nodes of the control tree).

The set $T(\Sigma)$ of those selector-functions which, if applied to $\Sigma$, give an instruction at a terminal node is defined as :

$$T(\Sigma) = \{ \tau \mid \tau \in \Sigma^* \& \exists (\tau'(\Sigma)) \& \forall f (\neg \exists \text{succ}(f)(\tau'(\Sigma))) \}$$

The function $Q(\tau), \tau \in T$, is defined as follows :
if $\tau = \text{succ}(f) \cdot \tau'$, then the value-return-point $\nu$ for the instruction $\tau(\Sigma)$ is $\nu = f \cdot \tau'$. 
This means that for an instruction which is the succ(f)-part of another instruction the return-point for the value defined by the application of the instruction to $\xi$, is the f-part of this other instruction.

It is this mechanism which allows the "computing of the argument of an instruction by another instruction".

3.2.3 The definition of instructions

There are special notational conventions for the definition of instructions, i.e. for the definition of the pair $<\xi', v>$ resulting from the application of an instruction to a state $\xi$ and an instruction location $\tau$.

Conditional expressions are used for the definition of instructions; the following format is permitted:

```
instr-name(x_1, ..., x_n) =

P_1 \rightarrow \{ group1

P_2 \rightarrow \{ group2

... ...

P_m \rightarrow \{ group_m
```

A group may be one of the following:

a) a control representation
b) a basic instruction definition
The names $x_1, \ldots x_n$ may occur in the predicates $p_1, \ldots p_m$ and in the groups $\text{group}_1, \ldots \text{group}_m$.

Let "instr" be the instruction to be executed and $\tau$ the instruction location. A definition of the above form defines the result of the application of instr if instr-name = s-instr-name(instr).

Under the assumption that a certain group is selected by the predicates $p_i$ (which is explained later), the meaning of the definition is described separately for the two types of groups in the following.

3.2.31 A control-representation is the following:

$$c\text{-rep} = \text{instr-name}(f_1, \ldots f_n);$$

$$\{ \kappa_1 : c\text{-rep}_1, \ldots \kappa_m : c\text{-rep}_m \},$$

where instr-name is the name of an instruction, $f_1, \ldots f_n$ are names, $\kappa_1 \ldots \kappa_m$ are selector-functions and $c\text{-rep}_1, \ldots c\text{-rep}_m$ are control-representations, $n$ and $m$ may be zero. The set

$$\{ \kappa_1 : c\text{-rep}_1, \ldots \kappa_m : c\text{-rep}_m \}$$

is called the set of successors. A control-representation $c\text{-rep}_i$ is called "value-returning" if $\kappa_i$ has the special form $\kappa_i = x_k$ where $f_k$ is one of the names $f_1 \ldots f_n$.

A control-representation of the above form represents the following object $c$ (which is of type control):

$$c = \mu_o ( <\text{s-instr-name:instr-name}>,$$

$$\{ <s(1):f_1>, \ldots <s(n):f_n> \},$$

$$\{ <\text{succ}(\kappa'\cdot s(i)):c_k> \mid \kappa'\cdot f_i = \kappa_k \},$$

$$\{ <\text{succ}(\kappa_i):c_1> \mid c\text{-rep}_i \text{ is not value-returning} \} )$$

where $c_1$ is the object represented by $c\text{-rep}_i$. 
Note: The argument of an instruction corresponding to the $i^{th}$ place of the argument-list in the above representation is selected by the selector-function $s(i)$. A successor-instruction that returns its value to the $\chi'$-part of the $i^{th}$ argument is selected by the selector-function $\text{succ}(\chi' \cdot s(i))$.

Note that the form of the function names $s(i)$ is a rather special one, which is used only for the present purpose of the explication of the notation.

Now we define the application of an instruction $\text{instr}$ to a state $\xi$ and an instruction-location $\tau$, for the case that the relevant definition reads as follows:

$$\text{instr-name-1}(x_1, \ldots, x_k) = \text{c-rep}$$

where $\text{c-rep}$ is a control-representation of the above form and

$$\text{instr-name-1} = \text{s-instr-name(instr)} :$$

$$\text{instr}(\xi, \tau) = \langle \mu(\xi, \langle \tau \cdot s-c:c \rangle, \Omega) \rangle$$

where $c$ is the object obtained by replacing all occurrences of $x_i$ in the object represented by $\text{c-rep}$, by $s(i)(\text{instr})$, for all $i = 1, \ldots, k$.

This means that the instruction $\text{instr}$ is replaced in the control-part of $\xi$ by the object $c$, and that the application of $\text{instr}$ to $\xi$ does not define a value.
3.2.32 A basic instruction definition has the format

\[
\text{RETURNS} : e_0 \\
\text{s-p}_1 : e_1 \\
\vdots \\
\text{s-p}_n : e_n
\]

where the terms $e_i$, $0 \leq i \leq n$, denote objects and $s-p_i$ are selector-functions, which give one of the major parts (as listed in 3.1.1) if applied to $\xi$.

The application of an instruction instr to a state $\xi$ and an instruction-location $\tau$, for the case that the definition reads:

\[
\text{instr-name-1}(x_1, \ldots x_k) = \text{RETURNS} : e_0 \\
\text{s-p}_1 : e_1 \\
\vdots \\
\text{s-p}_n : e_n
\]

is defined as

\[
\text{instr}(\xi, \tau) = \langle \mu(\xi, <s-p_1:e_1'>, \ldots <s-p_n:e_n'>), e_0 \rangle
\]

where $e_i'$, $i = 1, \ldots n$ is obtained by replacing all occurrences of $x_j$ in $e_i$ by $s(j)($instr$)$, for $j = 1, \ldots k$. 
3.2.33 The truth values for the selection of a certain group in a conditional expression defining an instruction instr

\[ \text{instr-name}(x_1, \ldots, x_k) = \]

\[ p_i \rightarrow \{ \text{group}_i \} \]

\[ \vdots \]

\[ p_n \rightarrow \{ \text{group}_n \} \]

are obtained from the predicates \( p_i, i = 1, \ldots, n \) by replacing all occurrences of \( x_j \) by \( s(j)(\text{instr}) \), for all \( j = 1, \ldots, k \).

3.2.34 Further notational conveniences for control-representations are the following:

a) if the set of successors is the empty set, it may be omitted. The semicolon preceding the set may also be omitted.

b) if \( \nu_i : \text{c-rep}_i \) is an element of the successor-set and \( \text{c-rep}_i \) is not value-returning, then \( \nu_i \) and the succeeding colon may be omitted.

c) the set-parentheses may be omitted in the representation of the set of successors, if either there is only one element in the set, or if proper indentation is used:
\text{instr-name}(x_1, \ldots, x_k); \{ \kappa_1 : c\text{-rep}_1, \kappa_2 : c\text{-rep}_2, \ldots \kappa_n : c\text{-rep}_n \}

is equivalent to

\text{instr-name}(x_1, \ldots, x_k);
\kappa_1 : c\text{-rep}_1
\kappa_2 : c\text{-rep}_2
\ldots
\kappa_n : c\text{-rep}_n

d) all instruction-names used in this document are underlined.
e) \nu in the following definition will always denote the value return-point for the instruction being executed:

\nu = \varphi(\tau)
4. BASIC INTERPRETATION FUNCTIONS AND INSTRUCTIONS

This chapter contains the definitions of those functions and instructions which are common building blocks for the definition of the PL/I statements in chapter 5 and 6. These instructions correspond to elementary constituents of the processes that we described by PL/I programs such as the assignment of an elementary value to a storage location, reference to variables, allocation of storage etc.
4.1 Predicates and Functions defined over Attributes

This chapter gives a collection of predicates and functions over attributes that are used more frequently at several places in the following chapters.

4.1.1 Auxiliary data-attributes

Definition:

\[
\begin{align*}
\text{is-array} &= \text{is-named-array} \mid \text{is-array-descr} \\
\text{is-struct} &= \text{is-named-struct} \mid \text{is-struct-descr} \\
\text{is-scalar} &= \text{is-named-scalar} \mid \text{is-scalar-descr}
\end{align*}
\]

4.1.2 \text{equal}(\text{da}_1, \text{da}_2)

The predicate defines the equality of data-attributes \(\text{da}_1, \text{da}_2\), when the names are ignored.

Definition:

\[
\begin{align*}
\text{equal}(\text{da}_1, \text{da}_2) &= \\
&= \text{is-array}(\text{da}_1) \& \text{is-array}(\text{da}_2) \iff \\
&\quad \text{s-lbd}(\text{da}_1) = \text{s-lbd}(\text{da}_2) \& \text{s-ubd}(\text{da}_1) = \text{s-ubd}(\text{da}_2) \& \\
&\quad \text{equal}(\text{s-elem}(\text{da}_1), \text{s-elem}(\text{da}_2)) \\
&\quad \text{is-struct}(\text{da}_1) \& \text{is-struct}(\text{da}_2) \& \text{order}(\text{da}_1) = \text{order}(\text{da}_2) \iff \\
&\quad \text{order}(\text{da}_1) \\
&\quad \text{Et} \quad \text{equal}(\text{s-da-succ}(i, \text{da}_1), \text{s-da-succ}(i, \text{da}_2)) \\
&\quad \text{T} \iff \text{da}_1 = \text{da}_2
\end{align*}
\]
4.1.3 is-isub-def(attr)

The function is true if attr is isub-defined.

Definition:

\[
\text{is-isub-def(attr)} = \text{is-defined(attr)} \land \\
(\exists \chi)(\text{is-isub}(\chi \cdot s\text{-base}(\text{attr})))
\]

4.1.4 is-bit-class(da), is-char-class(da)

The functions is-scalar-part(da,η) and new-da(da,η) are defined in 4.2.2 and 4.2.3.

Definition:

\[
\text{is-bit-class(da)} = \\
(\forall \eta)(\text{is-scalar-part}(\text{da},\eta) \supset \text{is-bit}\cdot s\cdot \text{base}\cdot \text{new-da}(\text{da},\eta))
\]

Definition:

\[
\text{is-char-class(da)} = \\
(\forall \eta)(\text{is-scalar-part}(\text{da},\eta) \supset \text{is-char}\cdot s\cdot \text{base}\cdot \text{new-da}(\text{da},\eta))
\]

4.1.5 order(da)

The function gives the number of immediately contained sub-structures in a structure.

Definition:

\[
\text{order(da)} = \\
\text{is-struct(da)} \rightarrow (\exists i)(\text{exists(succ}(i,\text{da})) \land \\
\neg \text{exists(succ}(i+1,\text{da}))
\]
4.1.6 elem-extent(da)

The function gives the number of elementary parts in da.

**Definition:**

\[
\text{elem-extent}(da) = \\
\begin{cases}
\text{is-array}(da) & \rightarrow (s-\text{ubd}(da) - s-\text{lbd}(da) + 1) \cdot \text{elem-extent}(s-\text{elem}(da)) \\
\text{is-struct}(da) & \\
\sum_{i=1}^{\text{order}(da)} \text{elem-extent}(s-\text{da} \cdot \text{succ}(i, da)) \\
\text{is-string}(da) & \rightarrow s-\text{length}(da) \\
T & \rightarrow 1
\end{cases}
\]

4.1.7 scalar-extent(da)

The function gives the number of scalar parts in da.

**Definition:**

\[
\text{scalar-extent}(da) = \\
\begin{cases}
\text{is-array}(da) & \rightarrow (s-\text{ubd}(da) - s-\text{lbd}(da) + 1) \cdot \text{scalar-extent}(s-\text{elem}(da)) \\
\text{is-struct}(da) & \\
\sum_{i=1}^{\text{order}(da)} \text{scalar-extent}(s-\text{da} \cdot \text{succ}(i, da)) \\
T & \rightarrow 1
\end{cases}
\]
4.1.8 $\text{dim}(da)$

The function gives the number of dimensions of the array $da$.

**Definition:**

$$\text{dim}(da) = \begin{cases} 
\text{is-array}(da) & \rightarrow 1 + \text{dim}(s\text{-elem}(da)) \\
T & \rightarrow 0 
\end{cases}$$

4.1.9 $\text{m-si}(da)$

The function gives the structure information associated with $da$.

**Definition:**

$$\text{m-si}(da) = \begin{cases} 
\text{is-array}(da) & \rightarrow \\
\mu_x (\langle s\text{-lbd} : s\text{-lbd}(da) \rangle, \langle s\text{-ubd} : s\text{-ubd}(da) \rangle, \\
\langle s\text{-elem} : \text{m-si}(s\text{-elem}(da)) \rangle) & \\
\text{is-struct}(da) & \rightarrow \\
\mu_x (\{ \langle s\text{-da.succ}(i) : \text{m-si}(s\text{-da.succ}(i,da)) \rangle \mid \text{is-sorder}(da) \}) & \\
\text{is-scalar}(da) & \rightarrow \text{EMPTY} 
\end{cases}$$
4.1.10 \texttt{eval-da(da,env)}

The instruction returns the data attribute \( da \) with expressions specifying bounds and lengths being evaluated in the environment \( env \).

\textbf{Definition}:

\begin{align*}
\text{eval-da(da,env)} & = \\
& \text{pass(da)} ; \\
& \{ x \cdot da : \text{-integer-expr}(x(da), env) \mid \\
& (\exists x_1)(x = s-\text{lb} \cdot x_1 \lor x = s-\text{ub} \cdot x_1 \lor \\
& \quad x = s-\text{length} \cdot x_1) \}
\end{align*}

4.1.11 \texttt{bds-set(da)}

The function gives the set of those selectors that give bounds if applied to \( da \).

\textbf{Definition}:

\begin{align*}
\text{bds-set(da)} & = \{ x \mid (\exists x_1)(x = s-\text{lb} \cdot x_1 \lor x = s-\text{ub} \cdot x_1) \land \exists \text{ exists. } x(da) \}\n\end{align*}

4.1.12 \texttt{is-variable(attr)}

\textbf{Definition}:

is-variable = is-prop-variable \lor is-data-param \lor is-defined \lor is-based
4.2 Reference to Variables

This chapter defines the reference to proper, based and defined variables and data parameters. There are three major steps:

1) evaluation of the reference to the variable in the text (i.e. the interpretation of subscripts and qualification),
2) the evaluation of the generation associated with the variable at the point of reference,
3) reference to this generation, i.e. passing of a sub-generation, with the help of the reference information obtained in step 1.

Step 1 is the same for all kinds of variables. Step 2 is the same for proper variables and data parameters, but different for based variables and again different for defined variables. Step 3 is accomplished by one common instruction. Except in the case of isub-defined variables, the reference first is made without referring to a possible expansion due to the rules for expansion of aggregate expressions, i.e. that generation is the result of the reference that is defined by the reference information obtained from the text. The three steps are followed by a further step which makes the selection either of a scalar generation or of a scalar value, depending on the context of the reference. Only in the case of isub-defined variables both expansion and reference have to be made in one step. In order to make this treatment possible, preliminary information about the expansion of aggregate expressions is given in 4.2.4.

4.2.1 Interpretation of subscripts and qualifications

A subscripted, qualified name is interpreted by the instruction int-name (t,env), where t is the text and env is the environment in which the subscripts are to be evaluated. The result is a list
(the reference-list) whose elements are either integers or empty. The integers are the result of the evaluation of subscript expressions in the case of reference to an array or the index of substructures in the case of reference to a structure; the empty positions derive from empty subscripts.

Abstract syntax of a subscripted, qualified name (simple reference):

\[
is\text{-simple-ref} = (\langle s\text{-id:is-name} \rangle,
\langle s\text{-qual-list:is-name-list} \rangle
\langle s\text{-arg-list:is-arg-list} \rangle)
\]

\[
is\text{-arg} = (is\text{-expr} \mid is\text{-empty})
\]

Abbreviations: \[da_t = s\text{-da}\ast s\text{-attr\{s\text{-id}(t)(env)\}}(AT)\]

Definition:

\[
int\text{-name}(t, env) =
make-rl(da_t, list, s\text{-qual-list}(t));
\{elem(i)\cdot list :
\begin{align*}
& int\text{-integer}\text{-expr}(elem(i, s\text{-arg-list}(t)), env) \\
& 1 \leq i \leq length(s\text{-arg-list}(t))
\end{align*}
\]

Note: It is assumed that the text contains fully qualified names and that the identifier s\text{-id}(t) is sufficient to identify the variable in the environment.

4.2.11 \textit{make-rl}(da, subs\text{-list, qual\text{-list})}

The instruction returns the reference list given the data-attributes da, the list of evaluated subscripts subs\text{-list and the qualification list qual\text{-list}}.
**Definition:**

\[
\text{make-rl}(\text{da}, \text{subs-list}, \text{qual-list}) = \\
\text{RETURNS: ref-list}(\text{da}, \text{subs-list}, \text{qual-list})
\]

4.2.12 ref-list(\text{da}, \text{subs-list}, \text{qual-list})

The function yields as value the reference-list given the data-attributes \text{da}, the list of evaluated subscripts \text{subs-list} and the qualification list \text{qual-list}.

**Definition:**

\[
\text{ref-list}(\text{da}, \text{subs-list}, \text{qual-list}) = \\
\text{subs-list} = < > \rightarrow \text{ref-list-1}(\text{da}, \text{qual-list}) \\
T \qquad \rightarrow \text{ref-list-2}(\text{da}, \text{subs-list}, \text{qual-list})
\]

**Definition:**

\[
\text{ref-list-1}(\text{da}, \text{qual-list}) = \\
is-array(\text{da}) \rightarrow \text{<EMPTY>ref-list-1}(\text{s-elem(\text{da})}, \text{qual-list}) \\
\text{qual-list} = < > \rightarrow < > \\
is-struct(\text{da}) \rightarrow \langle \text{nu(\text{da}, head(\text{qual-list}))} \rangle \text{ref-list-1}(\text{s-da\cdot succ(nu(\text{da}, head(\text{qual-list})), \text{da})}, \text{tail(\text{qual-list})})
\]
Definition:

\[
\text{ref-list-2}(\text{da,subs-list,qual-list}) = \\
\quad \begin{cases}
\text{is-array}(\text{da}) & \rightarrow \langle \text{head}(\text{subs-list}) \rangle
\end{cases}
\]

\[
\text{ref-list-2}(\text{s-elem}(\text{da}),\text{tail}(\text{subs-list}),\text{qual-list})
\]

\[
\text{subs-list} = \text{qual-list} = \langle \rangle \quad \longrightarrow \quad \langle \rangle
\]

\[
\text{is-struct}(\text{da}) \quad \rightarrow \quad \langle \text{nu}(\text{da,head}(\text{qual-list})) \rangle
\]

\[
\text{ref-list-2}(\text{s-da}*\text{succ}(\text{nu}(\text{da,head}(\text{qual-list})),\text{da}),
\quad \text{subs-list, tail}(\text{qual-list}))
\]

where \(\text{nu}(\text{da},\text{id})\) computes the number of that sub-structure of \(\text{da}\) whose identifier is equal to \(\text{id}\):

\[
\text{nu}(\text{da},\text{id}) = (\text{li})\quad (\text{s-id}*\text{succ}(\text{i,da}) = \text{id})
\]

Note: The function \(\text{ref-list-1}\) computes the reference-list in the case that no subscript is specified (i.e. that the variable is not an array and does not contain arrays, or that all subscripts are not specified). The function \(\text{ref-list-2}\) computes the reference-list in the case that all subscripts are specified (possibly by EMPTY). No other case is allowed.

4.2.2 Data attributes and reference-lists

A reference-list resulting from the interpretation of a subscripted, qualified name defines a certain sub-part of the data-attribute of the data-aggregate to which the reference is made. The elementary sub-parts of data-attribute are considered to be arranged also in a linear order, so that a specific elementary or scalar sub-part of a data-attribute may be specified either

1. by the reference-list defining it, or
2. by its element-index, or its scalar-index, respectively.
There are in fact two indices defined: the element-index of an elementary part of a data-attribute, which is the position in the linear sequence in which the elementary parts are defined to be arranged; the scalar-index of a scalar part which is the position in the linear sequence in which the scalar parts are defined to be arranged. Elementary parts are scalar parts except strings, and elements of strings (bits and characters).

In the following a set of functions is defined which establish the relation between the reference-list and the element-index and scalar-index with respect to a data-attribute.

4.2.21 The function is-elem-part(da, rl) is true if the reference-list rl defines an elementary part of the data-attribute da, false otherwise.

**Definition:**

\[
is\text{-elem-part}(da, rl) = \begin{cases} 
\text{is-array}(da) \land \text{length}(rl) \geq 1 \quad & \text{(s-lbd}(da) \leq \text{head}(rl) \leq \text{s-ubd}(da)) \land \\
\text{is-elem-part}(\text{s-elem}(da), \text{tail}(rl)) \quad & \text{is-struct}(da) \land \text{length}(rl) \geq 1 \quad & \text{(1} \leq \text{head}(rl) \leq \text{order}(da)) \land \\
\text{is-elem-part}(\text{s-da}\cdot\text{succ}(\text{head}(rl), da), \text{tail}(rl)) \quad & \text{is-string}(da) \land \text{length}(rl) = 1 \Rightarrow \text{1} \leq \text{head}(rl) \leq \text{s-length}(da) \land \\
\text{is-scalar}(da) \land \text{length}(rl) = 0 \Rightarrow T \quad & T \Rightarrow F
\end{cases}
\]
4.2.22 is-scalar-part(da, rl)

The function is true if the reference-list rl defines a scalar part of the data-attribute da, false otherwise.

Definition:

\[
is-scalar-part(da, rl) =
\begin{align*}
is-array(da) & \land length(rl) \geq 1 \\
(s-lbd(da) \leq head(rl) \leq s-ubd(da)) & \land \\
is-scalar-part(s-elem(da), tail(rl)) \\
is-struct(da) & \land length(rl) \geq 1 \\
(1 \leq head(rl) \leq order(da)) & \land \\
is-scalar-part(s-da\cdot succ(head(rl), da), tail(rl))
\end{align*}
\]

\[is-empty(da) \lor is-scalar(da) \land length(rl) = 0 \rightarrow T\]

\[T \rightarrow F\]

4.2.23 elem-index(da, rl)

The function gives the position-number of the elementary part of the data-attribute da which is defined by the reference-list rl.

Definition:

\[
\begin{align*}
elem-index(da, rl) = 
\begin{cases} 
is-elem-part(da, rl) & \\
(is-array(da) & \\
(head(rl) - s-lbd(da)) \cdot elem-extent(s-elem(da)) + \\
 elem-index(s-elem(da), tail(rl)) & \\
is-struct(da) & \\
\sum_{i=1}^{head(rl)-1} elem-extent(s-da\cdot succ(i, da)) + \\
 elem-index(s-da\cdot succ(head(rl), da), tail(rl)) & \\
is-string(da) & \\
head(rl) & \\
T & 1
\end{cases}
\end{align*}
\]
4.2.24 scalar-index(da,rl)

The function gives the position-number of the elementary part of the data-attribute da which is defined by the reference-list rl.

Definition:

\[
\text{scalar-index}(da, rl) = \\
\quad \text{is-scalar-part}(da, rl) \\
\quad \quad (\text{is-array}(da) \\
\quad \quad \quad (\text{head}(rl) - \text{s-lbd}(da)) \cdot \text{scalar-extent}(\text{s-elem}(da)) + \\
\quad \quad \quad \text{scalar-index}(\text{s-elem}(da), \text{tail}(rl)) \\
\quad \quad \text{is-struct}(da) \rightarrow \sum_{i=1}^{h} \text{scalar-extent}(\text{s-da-\text{succ}(i, da)}) + \\
\quad \quad \text{scalar-index}(\text{s-da-\text{succ}(\text{head}(rl), da}), \\
\quad \quad \quad \text{tail}(rl)) \\
\quad \quad T \rightarrow 1 )
\]

4.2.3 Sub-generations of generations

The function sub-gen(gen,rl) gives the subgeneration of the generation gen that is defined by the reference-list rl.

Abstract syntax of a generation:

\[
is-gen = ( <s-da:is-da>, <s-dens:is-dens>, \\
\quad \quad <s-addr:is-a>, <s-il:is-integer-list> )
\]
Definition:

\[ \text{sub-gen}(\text{gen}, \text{rl}) = \]
\[ \mu_\circ ( \langle \text{s-da: new-da}(\text{s-da}(\text{gen}), \text{rl}) \rangle, \]
\[ \langle \text{s-dens: s-dens}(\text{gen}) \rangle, \langle \text{s-addr: s-addr}(\text{gen}) \rangle, \]
\[ \langle \text{s-il: new-il}(\text{s-da}(\text{gen}), \text{s-il}(\text{gen}), \text{rl}) \rangle ) \]

Note: Address and density remain the same for the new generation. The data-attributes and the index-list are changed.

4.2.31 new-da(\text{da}, \text{rl})

The function gives that part of the data-attribute \text{da} that is defined by the reference-list \text{rl}.

Definition:

\[ \text{new-da}(\text{da}, \text{rl}) = \]
\[ \text{rl} = < > \rightarrow \text{da} \]
\[ \text{is-array}(\text{da}) \land \text{is-empty}(\text{head}(\text{rl})) \rightarrow \]
\[ \mu_\circ ( \langle \text{s-lbd: s-lbd}(\text{da}) \rangle, \langle \text{s-ubd: s-ubd}(\text{da}) \rangle, \]
\[ \langle \text{s-elem: new-da}(\text{s-elem}(\text{da}), \text{tail}(\text{rl})) \rangle ) \]
\[ \text{is-array}(\text{da}) \rightarrow \text{new-da}(\text{s-elem}(\text{da}), \text{tail}(\text{rl})) \]
\[ \text{is-struct}(\text{da}) \rightarrow \text{new-da}(\text{s-da: succ}(\text{head}(\text{rl}), \text{da}), \text{tail}(\text{rl})) \]

Note: The function is not defined if \text{da} is scalar and \text{rl} is not the empty-list, and if \text{da} is a structure and the head of \text{rl} is empty.
4.2.32 new-il(da,il,rl)

The function gives the new index-list which is composed of that parts of the index-list il that correspond to the indices of those scalar parts of the data-attribute da that are defined by the reference-list rl.

Definition:

\[ \text{new-il}(\text{da}, \text{il}, \text{rl}) = \mu_1( \{ <\text{elem}(i) : \text{elem}(k, \text{il})> | k \in \text{index-set}(\text{da}, \text{rl}) & i = \text{card}(\{j | j \in \text{index-set}(\text{da}, \text{rl}) & j \leq k\}) \} ) \]

where the function "card" applied to a set yields the number of elements of the set.

4.2.321 index-set(da,rl)

The function gives the set of indices of that elementary parts of da that are defined by rl. (See chapter 4.1 for the definition of elem-index(da,rl).) It is this set of indices that is arranged by the function new-il to a new index-list.

Definition:

\[ \text{index-set}(\text{da}, \text{rl}) = \{ i | (\exists \eta)( \text{is-elem-part}(\text{da}, \eta) & (i = \text{elem-index}(\text{da}, \eta)) & \text{length}(\text{rl}) \leq \sum_{i=1}^{\text{length}(\text{rl})} \text{is-empty(\text{elem}(i, \text{rl}))} ) \} \]

Note: If rl is the empty-list then the result is the set of indices of all elementary parts of da.
4.2.33 is-connected(gen)

The predicate is true if the generation gen is a connected generation, false otherwise.

Definition:

\[
is\text{-}connected(gen) =
(\forall i)((is\text{-}is\text{length}\cdot s\text{-}il(gen) - 1) \lor (elem(i+1, s\text{-}il(gen)) =
\quad elem(i, s\text{-}il(gen)+1)))
\]
4.2.4 Preliminaries on the expansion of aggregate expressions

Some knowledge on the mechanism of the expansion of the interpretation of aggregate expressions into a sequence of interpretation passes is necessary in order to define the reference to variables. This chapter gives the necessary information and presents the definition of the functions for the treatment of the expansion information. The interpretation of expressions is described fully in chapter 4.3

The text of an aggregate expression is interpreted as many times as there are scalar parts of the operands in the expression. The interpretation is done depending on additional information, the expansion information. The expansion information specifies the data structure that is common to all operands in the expression (with the exception of scalar operands) and a reference-list (called the expansion-list) which defines that scalar part in the operands that has to be selected at the current interpretation.

The result of the interpretation of one scalar part of an expression is an object:

\[(s-ei:is-ei),(s-op:is-op)\]

where s-ei selects the expansion information of this interpretation, and s-op selects the resulting operand.

The abstract syntax of an expansion information is

\[is-ei = (s-el:is-integer-list),(s-si:is-si)\]

where s-el selects the expansion-list and s-si selects the structure information.
The structure information gives the structuring defined by a data-attribute, i.e. bounds of arrays and the structuring of structures. The structure information associated with a data-attribute da is obtained by the function m-si(da), defined in chapter 4.1.

4.2.41 is-exhaust(ei)

The predicate is true if the expansion-list s-el(ei) does not define a scalar part in the structure information s-si(ei), false otherwise.

Definition :

\[
\text{is-exhaust(ei)} = \\
\text{is-empty(ei)} \quad \text{F} \\
\text{T} \quad \text{is-scalar-part(s-si(ei),s-el(ei))}
\]

Note : The predicate is false if ei is empty (ei may not be known at the first interpretation of an expression).

4.2.42 update(ei)

The function gives an expansion information such that its structure information is equal to s-si(ei) and its expansion-list is such that the next scalar part in the structure is defined.
Definition:

update(ei) =
scalar-index(s-si(ei),s-el(ei))<
   \langle scalar-extent(s-si(ei)) \rightarrow \\
   \mu_0(\langle s-si:s-si(ei)\rangle, \\
   \langle s-el: (\eta)(is-scalar-part(s-si(ei),\eta) & \\
   scalar-index(s-si(ei),\eta) = \\
   scalar-index(s-si(ei),s-el(el)+1) \rangle \\
   T \rightarrow \mu_0(\langle s-si:s-si(ei)\rangle, \\
   \langle s-el: \langle head(s-el(ei)) + 1 \rangle \rangle ^{tail(s-el(si))} \rangle)

4.2.43 init-ei(da)

The function gives an expansion information such that the first scalar part in the data-attribute da is defined.

Definition:

init-ei(da) =
\mu_0(\langle s-si:m-si(da)\rangle, \\
\langle s-el:(\eta)(scalar-index(da,\eta) = 1)\rangle)

The constant scalar expansion information is defined as:

Definition:

SC-EI = \mu_0(\langle s-el: > \rangle,\langle s-si:EMPTY\rangle)
4.2.44 compare-ei(ei₁, ei₂)

The function is defined if ei₁ and ei₂ are equal or if one of them is scalar. The function is used for the comparison of the expansion informations of the two operands of an infix-operation.

Definition:

\[
\text{compare-ei}(ei₁, ei₂) =
\begin{align*}
\text{is-empty}(s-si(ei₁)) & \quad \rightarrow \quad ei₂ \\
\text{is-empty}(s-si(ei₂)) & \quad \rightarrow \quad ei₁ \\
ei₁ = ei₂ & \quad \rightarrow \quad ei₁
\end{align*}
\]

4.2.45 check(da, ei)

The function gives the initial structure information of the data-attribute da if ei is empty, otherwise ei if it is compatible with da. The function is used to check the current expansion information against the data-attributes of a variable referenced to at the interpretation of an expression.

Definition:

\[
\text{check}(da, ei) =
\begin{align*}
\text{is-empty}(ei) & \quad \rightarrow \quad \text{init-ei}(da) \\
(s-si(ei) = m-si(da)) & \quad \lor \quad \text{is-scalar}(da) \quad \rightarrow \quad ei
\end{align*}
\]
### 4.2.5 Interpretation of the reference to variables

The reference to variables is accomplished by the instruction `int-variable(t, env)`, where `t` is the text of the reference and `env` is the environment in which `t` is to be interpreted. The environment is an explicit argument of the instruction so that it may be used for the interpretation of references to variables in an environment which is not the current one (e.g., for the interpretation of the base of a defined variable). The instruction returns a generation.

The abstract syntax of reference is:

\[
\begin{align*}
\text{is-ref} & = (\text{is-ptr-qual-ref} \mid \text{is-simple-ref}) \\
\text{is-ptr-qual-ref} & = (\langle \text{s-ptr-qual:is-ref} \rangle, \\
& \langle \text{s-based-ref:is-simple-ref} \rangle)
\end{align*}
\]

**Abbreviations:**

\[
\begin{align*}
\mathbf{n}_t & = \text{is-simple-ref}(t) \rightarrow \text{s-id}(t)(env) \\
\text{is-ptr-qual-ref}(t) & \rightarrow (\text{s-id}\cdot\text{s-based-ref}(t))(env) \\
\text{attr}_t & = \text{s-attr} \cdot \mathbf{n}_t(\text{AT}) \\
\text{da}_t & = \text{s-da} \cdot \text{attr}_t \\
\text{env}_t & = \text{s-env} \cdot \mathbf{n}_t(\text{AT}) \\
\text{dens}_t & = \text{s-dens} \cdot \text{attr}_t
\end{align*}
\]

The instruction `expand-value (<gen, ei>)` used in the following definition is defined separately in chapter 4.2.7. It gives the value of the sub-generation defined by the expansion-information `ei`. 
Definition:

\[
\text{int-variable}(t, \text{env}) = \\
is-ptr-qual-ref(t) \rightarrow \\
pass-sub-gen(gen, rl); \\
gen : \text{based-action}(ptr, da, dens_t) \\
ptr : \text{expand-value}(<ptr-gen, SC-EI>); \\
ptr-gen : \text{int-variable}(s-ptr-qual(t), env_t) \\
da : \text{eval-da}(da_t, env_t) \\
rl : \text{int-name}(s-based-ref(t), env_t) \\
is-based(attr_t) \rightarrow \\
pass-sub-gen(gen, rl); \\
gen : \text{based-action}(ptr, da, dens_t) \\
ptr : \text{expand-value}(<ptr-gen, SC-EI>) \\
ptr-gen : \text{int-variable}(s-ptr(attr_t), env_t) \\
da : \text{eval-da}(da_t, env_t) \\
rl : \text{int-name}(s-based-ref(t), env_t) \\
is-defined(attr_t) \& \neg is-isub-def(attr_t) \rightarrow \\
pass-sub-gen(gen, rl); \\
gen : \text{defined-action}(base-gen, attr_t) \\
base-gen : \text{int-variable}(s-base(attr_t), env_t) \\
rl : \text{int-name}(t, env) \\
is-prop-variable(attr_t) \rightarrow \\
pass-sub-gen(gen, rl); \\
gen : g(n_t) \\
rl : \text{int-name}(t, env)
4.2.51 based-action(ptr, da, dens)

The instruction returns the generation defined by the pointer ptr and the evaluated data-attributes da of the based variable. It checks the matching of data-attributes and density of the generation (referred to by the pointer) and of the based variable. In the case of structures the matching is required only up to a certain substructure.

Abbreviations:

\[ \begin{align*}
\text{da}_p &= s\text{-da} \cdot s\text{-vr} \cdot s\text{-op}(ptr) \\
\text{dens}_p &= s\text{-dens} \cdot s\text{-vr} \cdot s\text{-op}(ptr) \\
a &= s\text{-addr} \cdot s\text{-vr} \cdot s\text{-op}(ptr) \\
\text{index} &= s\text{-index} \cdot s\text{-vr} \cdot s\text{-op}(ptr)
\end{align*} \]

Definition:

\[ \text{based-action}(ptr, da, dens) = \]
\[ (\text{equal}(\text{da}_p, da) \lor \text{equ-order}(\text{da}_p, da) \geq 1) \land (\text{dens}_p = \text{dens}) \]

\[ \text{RETURNS: } \mu_o(\langle s\text{-da:}\text{nda}(\text{da}_p, da)\rangle, \langle s\text{-dens:}\text{dens}_p\rangle, \langle s\text{-addr:a}\rangle, \langle s\text{-il:mil}(\text{index,mda}(\text{da}_p, da))\rangle) \]

4.2.511 equ-order(da_1, da_2)

The function gives the number of the substructure up to which the two structure attributes da_1 and da_2 are equal.
**Definition:**

\[
e\text{equ-order}(\text{da}_1, \text{da}_2) =
\begin{align*}
is\text{-struct}(\text{da}_1) & \land is\text{-struct}(\text{da}_2) \quad \mapsto \\
& (\neg \text{equal} (s\cdot\text{da}\cdot\text{succ}(i, \text{da}_1), s\cdot\text{da}\cdot\text{succ}(i, \text{da}_2)) \quad \mapsto 0
\end{align*}
\]

\[T \mapsto (n\mid (i\geq 1) \quad \& \quad \exists_{t\geq 1} (\text{equal} (s\cdot\text{da}\cdot\text{succ}(k, \text{da}_1),
\begin{align*}
s\cdot\text{da}\cdot\text{succ}(k, \text{da}_2)) \quad & \land \\
\neg\text{equal} (s\cdot\text{da}\cdot\text{succ}(k+1, \text{da}_1),
\begin{align*}
s\cdot\text{da}\cdot\text{succ}(k+1, \text{da}_2))
\end{align*}
\end{align*}
\]

4.2.512 \( m\text{da}(\text{da}_1, \text{da}_2) \)

The function gives the data-attribute that is a common initial part to the data-attributes \( \text{da}_1 \) and \( \text{da}_2 \).

**Definition:**

\[
m\text{da}(\text{da}_1, \text{da}_2) =
\begin{align*}
is\text{-struct}(\text{da}_1) & \land is\text{-struct}(\text{da}_2) \land \text{equ-order}(\text{da}_1, \text{da}_2) \geq 0 \quad \mapsto \\
\mu_0 ( \{ \langle \text{succ}(i):\text{succ}(i, \text{da}_1) \rangle : i \leq \text{equ-order}(\text{da}_1, \text{da}_2) \} ) \\
\text{equal}(\text{da}_1, \text{da}_2) \quad \mapsto \quad \text{da}_1
\end{align*}
\]

4.2.513 \( m\text{il}(\text{index}, \text{da}) \)

The function gives the list of indices of the elementary parts of the data-attribute \( \text{da} \), starting with the integer index.

**Definition:**

\[
m\text{il}(\text{index}, \text{da}) = \mu_0 ( \{ \langle \text{elem}(i):\text{index}+i-1 \rangle : i \leq \text{elem}\cdot\text{extent}(\text{da}) \} )
\]
4.2.52 **defined-action**(gen,attr)

The instruction selects that part of the generation gen (the
generation of the base of the defined variable) that is defined by
the attributes attr of the defined variable. There are two cases:
overlay-defining (for packed string-class-aggregates) and
correspondence-defining (for arrays of equal number of dimensions
and equal data-attributes of their elements, and for structures and
scalars with equal data-attributes).

**Abbreviations:**

\[
\begin{align*}
da_b &= s-da(gen) \\
dens_b &= s-dens(gen) \\
a &= s-addr(gen) \\
il &= s-il(gen)
\end{align*}
\]

**Definition:**

\[
\text{defined-action}(\text{gen},\text{attr}) = \\
is-overlay(\text{gen},\text{attr}) \& is-packed(dens_b) \& is-packed(s-dens(\text{attr})) \& (\exists(\text{s-pos(\text{attr})} \vee is-corresp(da_b, s-da(\text{attr})))) \\
\text{RETURNS: } \mu_0(\langle s-da:s-da(\text{attr})\rangle, \langle s-dens:\text{PACKED} \rangle, \langle s-addr:a \rangle, \langle s-il:corr-il(da_b, s-da(\text{attr}), il) \rangle)
\]

\[
\text{is-corresp}(da_b, s-da(\text{attr})) \\
\text{RETURNS: } \mu_0(\langle s-da:s-da(\text{attr})\rangle, \langle s-dens:dens_b \rangle, \langle s-addr:a \rangle, \langle s-il:corr-il(da_b, s-da(\text{attr}), il) \rangle)
\]
Note: In the case that \( da_b \) and \( s-da\text{ (attr)} \) are both packed arrays of strings with equal string-attributes, and with equal number of dimensions, correspondence-defining is assumed.

4.2.521 \( \text{is-overlay}(\text{gen}, \text{attr}) \)

The function is true, if the generation \( \text{gen} \) and the attribute \( \text{attr} \) are both string-class, if the generation is connected, and if the extent of the generation overlaps the extent of \( \text{attr} \); the function is false otherwise.

Definition:

\[
\text{is overlay}(\text{gen}, \text{attr}) = \\
(is\text{-bit-class}(s-da(\text{gen})) \& is\text{-bit-class}(s-da(\text{attr})) \mid \\
is\text{-char-class}(s-da(\text{gen})) \& is\text{-char-class}(s-da(\text{attr}))) \& \\
is\text{-connected}(\text{gen}) \& \\
(elem\text{-extent}(s-da(\text{gen}))-\text{pos}(\text{attr}) \leq elem\text{-extent}(s-da(\text{attr}))-1)
\]

4.2.522 \( \text{is-corresp}(da_1, da_2) \)

The function is true if \( da_1 \) and \( da_2 \) are equal, or if they are arrays of equal number of dimensions and equal description of their elements, false otherwise.

Definition:

\[
\text{is-corresp}(da_1, da_2) = \\
is\text{-array}(da_1) \& is\text{-array}(da_2) \\
is\text{-corresp}(s\text{-elem}(da_1), s\text{-elem}(da_2)) \\
equal(da_1, da_2) \\
T \rightarrow T \\
F \rightarrow F
\]
4.2.523 pos(attr)

If the position attribute is present in the attribute attr, the function gives the specified integer, otherwise it gives 1.

Definition:

pos(attr) = exists(s-pos(attr)) → s-pos(attr)

4.2.524 corr-il(dab, dad, il)

The function makes the index list which in that positions that are defined for both dad and dab contains the corresponding element of il, and in positions that are defined only for dad contains EMPTY.

Definition:

corr-il(dab, dad, il) =

\[ \mu \{ \langle \text{elem}(\text{elem-index}(\text{dad}, \gamma)), k \rangle \mid \text{is-elem-part}(\text{dad}, \gamma) \land (\text{is-elem-part}(\text{dab}, \gamma) \land k = \text{elem}(\text{elem-index}(\text{dab}, \gamma), il) \lor \neg \text{is-elem-part}(\text{dab}, \gamma) \land k = \text{EMPTY}) \} \]
4.2.53 \texttt{pass-sub-gen}(gen,rl)

The instruction returns the subgeneration of the generation \texttt{gen} defined by the reference-list \texttt{rl}.

\textbf{Definition}:

\texttt{pass-sub-gen}(gen,rl) =

\hspace{1cm} \text{RETURNS} : \texttt{sub-gen}(gen,rl)

4.2.54 \texttt{g}(n)

The instruction returns the most recent generation of the variable whose unique name is \texttt{n}.

\textbf{Definition}:

\texttt{g}(n) = \text{RETURNS} : \texttt{head}^*(n(\texttt{DN})\texttt{AG})

4.2.6 Interpretation of isub-defined variables

References to isub-defined variables are interpreted by the instruction \texttt{isub-action}(n,rl,ei), where \texttt{n} is the unique name of the isub-defined variable, \texttt{ei} is the current expansion information (possibly empty) and \texttt{rl} is the reference-list computed from subscripts and qualifications to the variable. The reference-list \texttt{rl} and the expansion-list \texttt{s-el}(ei) have to be merged into a new reference-list (this is done by the function \texttt{merge}). If \texttt{ei} is empty, the initial expansion information has to be established (by the function \texttt{init-isub-ei}) and this expansion information is used to perform the merge with the reference-list. As many initial elements of the new reference-list are taken as there are dimensions of the isub-defined array-variable. These elements become the values of the isubs. The isubs are entered into the environment in which the defined variable
had been declared and associated directly with their integer values. This new environment is used to compute the generation of the base of the defined variable. The rest-list of the new reference-list (that part that has not been used to give values to the isubs) is then used to select a scalar sub-generation of the base-generation.

Abbreviations:

\[
\begin{align*}
\text{da} & = \text{s-da\cdot s-attr\cdot n(AT)} \\
\text{tb} & = \text{s-base\cdot s-attr\cdot n(AT)} \\
\text{env}_b & = \text{s-env\cdot n(AT)}
\end{align*}
\]

Definition:

\[
\text{isub-action}(n, rl, ei) = \\
\quad \text{pass-isub}(\text{gen}, \text{isub-check}(\text{da}, rl, ei));
\]
\[
\text{gen} : \text{pass-sub-gen}(\text{base-gen}, \text{rest-list}(\text{da}, rl, ei));
\]
\[
\text{base-gen} : \text{int-variable}(tb, \text{isub-env}(\text{env}_b, \text{isub-list}(\text{da}, rl, ei)))
\]

4.2.61 isub-list(da, rl, ei)

Gives the sublist of the result of the merging of the reference-list and the expansion-list, whose length is equal to the number of dimensions of the array da.

Definition:

\[
\text{isub-list}(\text{da}, rl, ei) = \\
\lambda_i \{ \text{elem}(i): \text{elem}(i, \text{merge}(rl, s-el \text{ isub-check}(\text{da}, rl, ei))) \mid 1 \leq i \leq \text{dim}(da) \}
\]
4.2.62 isub-check(da,rl,ei)

The function checks the matching of the expansion information ei with that sub-part of da that is defined by the reference-list rl; if ei is empty, the initial expansion-information is established corresponding to this sub-part.

Definition:

\[
isub-check(da,rl,ei) =
\begin{align*}
is-empty(ei) & \rightarrow \text{init-ei(new-da(da,rl))} \\
is-scalar(new-da(da,rl)) & \rightarrow \text{SC-EI} \\
(s-si(ei) = m-si(new-da(da,rl))) & \rightarrow ei
\end{align*}
\]

Note: If new-da(da,rl) is scalar and s-si(ei) is not scalar (which is a permissible case), the result is the scalar expansion information, since this is used in merging the reference-list and the expansion-list (see chapter 4.2.61).

4.2.63 merge (rl,el)

The function gives a list which is the result of merging the lists rl and el (reference-list and expansion-list).

Definition:

\[
\text{merge}(rl,el) =
\begin{align*}
\text{rl} = < > & \rightarrow el \\
\text{el} = < > & \rightarrow rl \\
is-empty*head(rl) & \rightarrow <\text{head}(el)>^*\text{merge(tail}(rl), \text{tail}(el)) \\
T & \rightarrow <\text{head}(rl)>^*\text{merge(tail}(rl),el)
\end{align*}
\]
4.2.64 rest-list(da,rl,ei)

The function gives that part of the merged lists that is not used to give values to the isubs.

Definition :

\[
\text{rest-list}(da,rl,ei) = \\
(\forall x) \left( \text{is-integer-list}(x) \land \\
\text{isub-list}(da,rl,ei) \land \\
x = \\
\text{merge}(rl,s-el\text{-isub-check}(da,rl,ei)) \right)
\]

4.2.65 isub-env(env,isub-list)

The function enters the isubs into the environment env together with their values taken from the isub-list.

Definition :

\[
\text{isub-env}(env,isub-list) = \\
\mu( \text{env}, \{ \langle \text{sub}(i)\text{-elem}(i,isub-list) \rangle \mid \\
1 \leq i \leq \text{length}(isub-list) \})
\]

4.2.66 paps-isub(gen,ei)

The instruction returns the pair \langle gen,ei \rangle.

Definition :

\[
\text{paps-isub}(gen,ei) = \\
\text{RETURNS} : \langle gen,ei \rangle
\]
4.2.7 The instructions \texttt{expand-gen} and \texttt{expand-value}

The instructions have a pair \(<\text{gen},\text{ei}>\) as argument. They return the subgeneration, or the value of the sub-generation, respectively, of the generation \text{gen} that is defined by the expansion information \text{ei}, together with the expansion information, that is the result of the function \text{check}(s\text{-da}(\text{gen}),\text{ei}).

4.2.71 \texttt{expand-gen}(\langle\text{gen},\text{ei}\rangle)

Definition:

\[
\text{expand-gen}(\langle\text{gen},\text{ei}\rangle) = \\
is\text{-scalar}(s\text{-da}(\text{gen})) \\
\downarrow \quad \downarrow \\
\text{RETURNS} : \mu_o(<\text{s-gen:gen}>,<\text{s-ei:check}(s\text{-da}(\text{gen}),\text{ei})>) \\
\exists\text{check}(s\text{-da}(\text{gen}),\text{ei}) \\
\downarrow \quad \downarrow \\
\text{RETURNS} : \mu_o(<\text{s-gen:sub-gen}(\text{gen}, \\
\quad \text{s-ei:check}(s\text{-da}(\text{gen}),\text{ei})), \\
\quad <\text{s-ei:check}(s\text{-da}(\text{gen}),\text{ei})>) \\
\]

Note: The instruction is defined only if the data-attributes of the generation match the expansion information.

4.2.72 \texttt{expand-value}(\langle\text{gen},\text{ei}\rangle)

Definition:

\[
\text{expand-value}(\langle\text{gen},\text{ei}\rangle) = \\
is\text{-scalar}(s\text{-da}(\text{gen})) \\
\downarrow \quad \downarrow \\
\text{RETURNS} : \mu_o(<\text{s-op:value}(\text{gen},\tilde{\xi}>, \\
\quad <\text{s-ei:check}(s\text{-da}(\text{gen}),\text{ei})>) \\
\exists\text{check}(s\text{-da}(\text{gen}),\text{ei}) \\
\downarrow \quad \downarrow \\
\text{RETURNS} : \mu_o(<\text{s-op:value}(\text{sub-gen}(\text{gen}, \\
\quad \text{s-ei:check}(s\text{-da}(\text{gen}),\text{ei}))), \\
\quad <\text{s-ei:check}(s\text{-da}(\text{gen}),\text{ei})>) \\
\]
Note: The instruction is defined only if the data attributes of the
generation match the expansion information.

4.2.73 value(gen,S)

The function makes an operand, given a scalar generation and the
storage-part S.

Definition:

value(gen,S) =

\[
\text{is-string}(s-da(gen)) \& \exists \text{s-var}\cdot s-da(gen) \rightarrow \\
\mu_0( \langle s-da: s-da(gen) \rangle, \\
\langle s-vr: \mu_1( \{ \langle \text{elem}(i): \text{s-value}\cdot \text{loc}(s-addr(gen),k)(S) \rangle \mid \\
k=\text{elem}(i,s-il(gen)) \& \exists \text{s-value}\cdot \text{loc}(s-addr(gen),k)(S) \} \rangle \rangle )
\]

\[
is-string(s-da(gen)) \rightarrow \\
\mu_0( \langle s-da: s-da(gen) \rangle, \\
\langle s-vr: \mu_1( \{ \langle \text{elem}(i): \text{s-value}\cdot \text{loc}(s-addr(gen),k)(S) \rangle \mid \\
k=\text{elem}(i,s-il(gen)) \& i \leq \text{length}(s-il(gen)) \} \rangle )
\]

\[
is-scalar(s-da(gen)) \rightarrow \\
\mu_0( \langle s-da: s-da(gen) \rangle, \\
\langle s-vr: \text{s-value}\cdot \text{loc}(s-addr(gen),\text{head}(s-il(gen)))(S) \rangle )
\]

Note: Elements of strings are stored in separate places in S.
The values of these places are collected to form a list, which
is the value-representation of the resulting operand. In the
case of varying-length strings currently not used storage
places have no value.
4.3 Operations on Data and Data Conversions

Operations are evaluated by the two instructions \texttt{int-infix-op} and \texttt{int-prefix-op}, both of which are described in this section.

In 4.3.1 the types of arguments are discussed, 4.3.2 gives abbreviations and conventions valid throughout all of chapter 4.3. Section 4.3.3 describes how the representation of values is modelled, 4.3.4 through 4.3.6 discuss the interpretation of infix operations. In 4.3.7 prefix operations are reduced to infix operations, and 4.3.8 describes the general convert function.

4.3.1 Objects occurring in expressions

In this section the different objects which will occur in the consequent sections, will be discussed.

4.3.1.1 Operands

A PL/I operand is an object such that \texttt{is-operand = (<s-da:is-da>, <s-vr:is-vr>)}, where da is a scalar data-attribute (see abstract syntax below). The value representation vr of the operand is the actual object stored in \texttt{S} (see 4.3.31 for further discussion).

Abstract Syntax:

\begin{align*}
\text{is-named-scalar} &= (\text{is-arithm} \mid \text{is-string} \mid \text{is-pict} \mid \text{is-label} \mid \text{is-task} \mid \text{is-event} \mid \text{is-ptr} \mid \text{is-named-cell} \mid \text{is-area}) \\
\text{is-arithm} &= (<s\text{-mode}:(\text{is-real} \mid \text{is-compl}>) , \<s\text{-base}:(\text{is-bin} \mid \text{is-dec})>, \<s\text{-scale}:(\text{is-fix} \mid \text{is-flt})>, \<s\text{-prec}:(\text{is-integer} \mid \text{is-integer-pair})>)
\end{align*}
is-string = (is-base:(is-bit | is-char),
    is-length:(is-expr | is-empty),
    is-var:is-opt)

is-opt = (is-true | is-\emptyset)

is-ptr = no detailed syntax is needed for the purpose of this chapter

From the data types listed above only operands of the following types: is-arith, is-string, is-picture and is-ptr are admissible. In this document numeric fields are not dealt with.

Names for some of the objects, which are denoted by above operands will be needed. In order to distinguish them from their representation the following symbols are used:

- \texttt{BO} \quad \text{the bit '0'}
- \texttt{BI} \quad \text{the bit '1'}
- \texttt{0} \quad \text{the number '0'}
- \texttt{1} \quad \text{the number '1'}
- \texttt{CHO} \quad \text{the character '0'}
- \texttt{CHI} \quad \text{the character '1'}
- \texttt{CHBL} \quad \text{the blank character}

\subsection*{4.3.12 Operators}

The following operators may occur in an expression:

Abstract Syntax:

\[
\text{is-infix-operator} = (\text{is-cat} | \text{is-or} | \text{is-and} | \text{is-lt} | \text{is-le} |
\text{is-eq} | \text{is-ge} | \text{is-gt} | \text{is-ne} | \text{is-add} |
\text{is-subtr} | \text{is-mult} | \text{is-div} | \text{is-power})
\]

\[
\text{is-prefix-operator} = (\text{is-minus} | \text{is-not})
\]
For reasons of convenience the following additional predicates are defined:

\[
\begin{align*}
\text{is-comp} &= \text{is-lt} \mid \text{is-le} \mid \text{is-eq} \mid \text{is-ge} \mid \text{is-gt} \mid \text{is-ne} \\
\text{is-bitop} &= \text{is-and} \mid \text{is-or} \mid \text{is-not} \\
\text{is-arithop} &= \text{is-add} \mid \text{is-subtr} \mid \text{is-mult} \mid \text{is-div} \mid \text{is-power} \mid \text{is-minus}
\end{align*}
\]

Above classes of operators will be called comparison, bit- and arithmetic operators, respectively.

4.3.13 Results of expressions

The value of an expression is (see also 4.2.4):

\[
(<\text{ei:is-ei}, <\text{op:is-operand}>).
\]

\text{ei is the expansion information (see 4.2.4) and op the operand part as described in 4.3.11.}

4.3.2 Abbreviations and conventions

For reasons of readability and convenience, a number of abbreviations will be introduced. These abbreviations are valid throughout chapter 4.3; additional abbreviations will be introduced where they are needed.
The following conventions are adopted for the range of variables:

name:                          range:

op, op_1, op_2                  operands
opor                          operators
x_1, x_2                      single characters or bits
n_1, n_2                      real or complex numbers
r_1, r_2                      expressions
da_t                          data attributes ("target-da")
v                            value (a character, a bit or a number)

The following abbreviations will be used:

abbreviation:                  condition:
da = s-da(op)                   is-operand(op)
da_1 = s-da(op_1)              is-operand(op_1)
da_2 = s-da(op_2)              is-operand(op_2)
p_1 = s-first*s-prec(da_1)     is-fix*s-scale(da_1)
s_1 = s-second*s-prec(da_2)    is-fix*s-scale(da_2)
p_2 = s-first*s-prec(da_2)     is-fix*s-scale(da_2)
s_2 = s-second*s-prec(da_2)    is-fix*s-scale(da_2)
N = the length of the largest number in the implementation
S = implementation defined precision of binary integers

INTG = μ_0 (<s-base:DEC>,
          <s-mode:REAL>,
          <s-scale:FIX>,
          <s-prec:DEF-PREC>)
**DEF-PREC** = default precision of a real fixed-point decimal integer. The scale-factor is 0 (i.e. 
DEF-PREC = \(<p,0>\).

**LOGICAL** = \(\mu^o(\langle s\text{-base:BIT},\langle s\text{-length:1}>\rangle)\)

**CHAR** = \(\mu^o(\langle s\text{-base:CHAR},\langle s\text{-length:1}>\rangle)\)

**BINTEGER** = \(\mu^o(\langle s\text{-base:BIN},\langle s\text{-mode:REAL},\langle s\text{-scale:FIX},\langle s\text{-prec:<S,0>>}\rangle\rangle\rangle\)

INTG is the data-attribute of an integer, LOGICAL that of a single bit, CHAR of a single character and BINTEGER of a binary integer.

### 4.3.3 Internal representation

Certain concepts exist, in which the implementation has an important influence; these concepts can be divided into two groups:

1) assumptions concerning the value representation

2) functions handling value representations.

#### 4.3.3.1 Assumptions dealing with the value representation

These assumptions are valid throughout the document. No assumptions are made about the representation of arithmetic data.

In the case of bit or character strings it is assumed that \(vr\) is a list, each of its elements containing exactly one charac-
ter or binary digit, respectively.

The respective representations of the numbers '0' and '1', the binary digits '0' and '1' and the characters '0' and '1' need not be the same. It is even admissible that the same value has more than one representation. As far as character strings are concerned, the restriction must be imposed, that the collating number (see 4.3.324) of such differing representations is the same.

4.3.32 Functions dependent on the representation

Since an operand contains a vr-part which represents a value, it is necessary to have certain functions which allow the transition between vr and the value represented by vr. These functions cannot be specified in PL/I, but some of their properties will be informally discussed.

4.3.321 represent(da,v)

The result of this function is a value representation (vr) of the value v for an operand with the data-attributes da. Only the following cases are admitted:

a) da is arithmetic and v is a real or complex number
b) da is of type bit string and v is a single bit
c) da is of type character string and v is a single character.

4.3.322 num-val(op)

The result of this function is the numeric value of the operand op, op has to be arithmetic. The result may be real or complex.
4.3.323  string-val(op)

The argument must be a string of length 1 or one element of a list of a value representation of a bit or character string. The result of this function is the bit or character represented by op.

4.3.324  collat(v)

This function yields the collating number of the argument v, where v is a single bit or character. It is assumed that this function yields a different real number (the collating number) for every character in the character set. Ordering the characters according to their collating sequence is assumed to be equivalent to ordering their collating numbers in ascending order.

For practical reasons it might be more useful to define a different function collat(vr) = collat(represent(da,vr)), whose argument is the representation of a character or bit in storage.

4.3.325  Relations between representation functions

The following identities must hold between the functions defined in 4.3.21 through 4.3.23:

\[
\begin{align*}
v &= \alpha\text{-val}(\mu_\circ(<s\text{-}da:da>, \langle s\text{-}vr:\text{represent}(da,v)\rangle)) \\
op &= \mu_\circ(<s\text{-}da:da>, \langle s\text{-}vr:\text{represent}(da, \alpha\text{-val}(op))\rangle)
\end{align*}
\]

where \(\alpha\text{-val}\) is num-val or string-val and op is arithmetic or string type of length 1, respectively.
4.3.4 Interpretation of infix operations

Infix operations are interpreted by the instruction \texttt{int-infix-op} \((r_1, r_2, \text{opor})\) where \(r_1\) and \(r_2\) are values of expressions (see 4.3.13) and \text{opor} is an operator (see 4.3.11).

The instruction \texttt{int-infix-op} will also be used for prefix operations. A prefix operation will be treated like an infix operation with two identical operands.

Definition:

\[
\text{int-infix-op}(r_1, r_2, \text{opor}) =
\]

\[
\text{RETURNS :}
\]

\[
<\text{compare-ei (s-ei(r_1), s-ei(r_2))},
\]

\[
\text{infix-op(final-op_1, final-op_2, \text{opor})}>
\]

where

\[
\text{final-op}_1 = \text{convert(target(da}_1, da_2, \text{fo}_1(\text{opor})),}
\]

\[
\text{s-op(r}_1)\]

\[
\text{final-op}_2 = \text{convert(target(da}_1, da_2, \text{fo}_2(\text{opor})),}
\]

\[
\text{s-op(r}_2)\]

and

\[
\text{da}_1 = \text{s-da}\ast \text{s-op(r}_1)\]

\[
\text{da}_2 = \text{s-da}\ast \text{s-op(r}_2)\]

\[
\text{fo}_1(\text{opor}) = \text{is-c-power(opor)} \quad \rightarrow \quad \text{CB-POWER}
\]

\[
\text{is-power(opor)} \quad \rightarrow \quad \text{B-POWER}
\]

\[
\text{T} \quad \rightarrow \quad \text{opor}
\]

\[
\text{fo}_2(\text{opor}) = (\text{is-c-power v is-power})(\text{opor}) \quad \rightarrow \quad \text{E-POWER}
\]

\[
\text{T} \quad \rightarrow \quad \text{opor}
\]

PL/I allows for a great variety of different types of operands and operators which may be combined in an expression. In the case that their types do not "match" (in the sense of chapter 4.3.5 there are two possibilities: either the combination is permissible, then conversion is performed on the operands or the combination is
illegal, then the result is undefined. An example for the first case is concatenation of two numbers, for the second case, concatenation of two pointers.

The interpretation of an infix operation can readily be split into three separate steps:

1) Calculation of the "proper" data attributes (in the sense of 4.3.5) for each operand. This is accomplished by the function target(da_1,da_2,opor).

In the course of the evaluation of the target attributes, exponentiation must be dealt with in a very special manner. For this reason three different types of exponentiation operators are used in the function "target" (see 4.3.5). All other operators are left unchanged by the functions fo_1,fo_2.

2) Conversion of the operands to their respective "target" data-attributes:

This is accomplished by the function convert(da_c,op), for details see 4.3.3. The function yields an operand which has the data-attributes da_c and whose value is the value of the operand op converted according to the rules of the SRL. The data-attributes da_c may be slightly more general than those described in the abstract syntax: In addition to the "regular" characteristics of da (see abstract syntax), the characteristic "EMPTY" may also appear, with the convention that no conversion is done for this specific characteristic. This particular characteristic of the resulting operand will be derived from other characteristics. da_c = EMPTY specifies that it is not necessary for any conversion to take place.

3) Execution of the infix operation:

The execution is accomplished by the function infix-op(op_1,op_2,opor). The arguments are made up of two operands and an operator. With the exception of three cases all the data-attributes of the two operands have to be the same.
These three exceptions are:
   a) If the operator is arithmetic, precisions of the operands may differ.
   b) If the operation is exponentiation, scale and mode, in addition to a may differ in several special cases (see 4.3.41).
   c) If the operator is concatenation, the length of the two strings may differ.

The function infix-op returns an operand which has the characteristics and value representation belonging to the result of the operation's execution (see 4.3.6).

### 4.3.41 Exceptions for exponentiation

This section applies solely to interpretation of exponentiation. Considering the data-attributes only, one finds that the results of an infix operation are independent of the order of the two operands, with the exception of exponentiation and, where applicable, resulting precisions.

As far as the case of exponentiation is concerned, a great difference exists between the treatment of the first and second operand. The target scale and target mode may be different for both operands. It was found necessary to either introduce two different target functions exclusively for the sake of exponentiation or to signal to the target function and its auxiliary function, as to whether a first or a second operand should be treated. The second approach was found to be much more convenient.

For the scope of the function target (see 4.3.5) the operators "B-POWER" ("base") and "E-POWER" ("exponent") are used together with the first and second operand respectively, instead of working with the operator "POWER". Instead of the operator "C-POWER" the operators "CB-POWER" and "E-POWER" are used. The "prefer-functions", which select
the appropriate data-attribute if the corresponding characteristics differ, have for mode and scale the operator as a third argument. This enables them to distinguish between a first and a second operand.

For the sake of perspicuity the rules for the scale targets have been collected and slightly reworded below:

Scale: If the scales of the two operands differ and the operation is not exponentiation, the fixed-point operand will be converted to floating-point. In the case of exponentiation, the first operand will be floated if the second operand is not a real, fixed-point, unsigned integer constant with precision \((p,0)\). The second operand is floated if it is not fixed-point with precision \((p,0)\). In addition the first operand is floated if \((p_1+1)\cdot \text{num-val}(\text{op}_2)-1 \geq \mathbb{N} \) or \(\text{num-val}(\text{op}_2)<0\) (for "\(p_1\)" see 4.3.2, for "\(\text{num-val}\)" 4.3.322). However, the last condition is different, since it takes into account not only the data-attributes, but also the value of the second operand.

Furthermore a missing specification for a result attribute had to be added. "If the first operand is floating-point and the second is fixed-point and the operation is exponentiation, then the base, mode, scale and precision of the result are the same base, mode, scale and precision, which would result from multiplication of the first operand with itself."
4.3.5 The target attributes of an operand

Infix operations are only performed with operands of essentially identical data-attributes. In addition the type of the operand has to match the type of the operator. Therefore the first step in the execution of an infix operation is the computation of the target attributes, i.e. the attributes to which an operand has to be converted. The function target \((da_1, da_2, opor)\) accomplishes this for each operand. Except for a special case of exponentiation (see 4.3.61) the target evaluation does not require a consideration of the values of the operands.

Definition:

\[
\text{target} (da_1, da_2, opor) = \\
\quad \text{is-arithop}(opor) \rightarrow \\
\quad \mu_0 (<s\text{-base:pref-a}_b(da_1, da_2)>, \\
\quad \quad <s\text{-mode:pref-a}_m(da_1, da_2, opor)>, \\
\quad \quad <s\text{-scale:pref-a}_s(da_1, da_2, opor)>, \\
\quad \quad <s\text{-prec:EMPTY}>)
\]

\[
\text{is-cat}(opor) \rightarrow \\
\quad \mu_0 (<s\text{-base:pref-s}_b(da_1, da_2)>, \\
\quad \quad <s\text{-length:EMPTY}>)
\]

\[
\text{is-comp}(opor) \& \text{is-arithm}(da_1) \& \text{is-arithm}(da_2) \rightarrow \\
\quad \mu_0 (<s\text{-base:EMPTY}>, \\
\quad \quad <s\text{-mode:EMPTY}>, \\
\quad \quad <s\text{-scale:EMPTY}>, \\
\quad \quad <s\text{-prec:EMPTY}>)
\]
is-comp(opor) & is-string(da₁) & is-string(da₂) →
μ₀(⟨s-basc:pref-sₜ(da₁,da₂)⟩,
    ⟨s-length:pref-s₁(da₁,da₂)⟩)

is-comp(opor) & is-ptr(da₁) & is-ptr(da₂) → EMPTY

is-bitop(opor) →
μ₀(⟨s-base:is-bit⟩,
    ⟨s-length:pref-sₜ(da₁,da₂)⟩)

The function target selects the type of the target attributes, while auxiliary "prefer" functions (see 4.3.51 and 4.3.52) handle the adaption of the individual characteristics (scale, length, etc.) within a type. For some prefer-functions the type of the operator is relevant (see 4.3.41). When converting to arithmetic, the precision is irrelevant and will be supplied by the convert function. For concatenation the length does not have to be changed. Comparison of arithmetic operands does not need any further conversions.

The only difficulty in the definition of "target" was introduced by the operator power (see 4.3.512, 4.3.513 and 4.3.41 for further comment).

4.3.51 Arithmetic prefer-functions

These functions yield as a result the "dominant" characteristics for one pair of data-attributes of the two operands.

4.3.511 pref-aₜ(da₁,da₂)

If the base of one of the operands is binary, then the target is binary, decimal otherwise. Bit strings are treated as binary, character strings as decimal, according to the rules of conversion.
Definition:

\[
\text{pref-a}_{b}(da_1, da_2) = \\
\text{IF } \text{is-bin } \lor \text{is-bit} \lor \text{s-base}(da_1) \lor \\
\text{is-bin } \lor \text{is-bit} \lor \text{s-base}(da_2) \rightarrow \text{BIN} \\
\text{T} \rightarrow \text{DEC}
\]

4.3.512 \text{pref-a}_m(da_1, da_2, opor)

For evaluating the target mode, an exception exists for the second operand of exponentiation. If the second operand is fixed-scale with zero scale factor, then its mode is not converted. The operator \text{E-POWER} signals to the function, informing it that the second operand is dealt with.

Definition:

\[
\text{pref-a}_m(da_1, da_2, opor) = \\
\text{is-e-power(opor) } \land \text{is-fix\_s-scale}(da_2) \land (s_2=0) \rightarrow \text{s-mode}(da_2) \\
\text{is-compl\_s-mode}(da_1) \lor \text{is-compl\_s-mode}(da_2) \rightarrow \text{COMPL} \\
\text{T} \rightarrow \text{REAL}
\]

4.3.513 \text{pref-a}_s(da_1, da_2, opor)

Except for the case of exponentiation, the target is complex if at least one of the operands is complex. In the case of exponentiation, the different operators \text{B-POWER}, \text{CD-POWER} and \text{E-POWER} allow the distinction between first and second operand. The rules for the target scale are described in 4.3.41. The last condition in 4.3.41, which involves the numeric value of the second operand, cannot be checked at this point, since only the data-attributes are available. The last condition has therefore be delegated to the function \text{infix-op}(see 4.3.6). A further complication is introduced by non-arithmetic operands: these are treated as fixed-point integers.
**Definition:**

\[
\text{pref-a}_5(da_1, da_2, opor) = \\
\begin{align*}
&\text{is-e-power}(opor) \& \text{is-fix-s-scale}(da_2) \& (s_2 = 0) \quad \rightarrow \quad \text{FIX} \\
&\text{is-e-power}(opor) \& \neg \text{is-arithm}(da_2) \quad \rightarrow \quad \text{FIX} \\
&\text{is-e-power}(opor) \quad \quad \rightarrow \quad \text{FLT} \\
&\text{is-b-power}(opor) \quad \quad \rightarrow \quad \text{FLT} \\
&\text{is-cb-power}(opor) \& \neg (\text{is-fix-s-scale}(da_2) \& (s_2 = 0) \& \\
&\quad \quad \quad \quad \text{is-real-s-mode}(da_2)) \quad \rightarrow \quad \text{FLT} \\
&\text{is-cb-power}(opor) \quad \quad \rightarrow \quad \text{s-scale}(da_2) \\
&\text{is-flt-s-scale}(da_1) \lor \text{is-flt-s-scale}(da_2) \quad \rightarrow \quad \text{FLT} \\
&\quad \quad \quad \quad \quad \quad \quad \rightarrow \quad \text{FIX}
\end{align*}
\]

4.3.52 String prefer-functions

The result of these functions is the "dominant" string attribute.

4.3.521 \text{prefer-s}_b(da_1, da_2)

This function selects the base of a string.

**Definition:**

\[
\text{prefer-s}_b(da_1, da_2) = \\
\begin{align*}
&\text{is-bit-s-base}(da_1) \& \text{is-bit-s-base}(da_2) \quad \rightarrow \quad \text{BIT} \\
&\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \rightarrow \quad \text{CHAR}
\end{align*}
\]

4.3.522 \text{prefer-s}_l(da_1, da_2)

This function yields the maximum length of two strings. The function is not defined for conversions from arithmetic to character string, since this was not needed for the function "target". The function \text{eval-bit-1} (defined in 4.3.34) gives the length of a bit string, which results from a conversion of arithmetic data to bit strings.
Definition:

\[
\text{prefer-s}_1 = \\
\text{is-string}(da_1) \& \text{is-string}(da_2) \rightarrow \\
\max(s\text{-length}(da_1), s\text{-length}(da_2)) \\
\text{is-arithm}(da_1) \rightarrow \\
\text{prefer-s}_1(\mu_o(s\text{-base:BIT}, \\
<\text{s-length:eval-bit-l}(da_1)>), da_2) \\
\text{is-arithm}(da_2) \rightarrow \\
\text{prefer-s}_1(da_1, \mu_o(s\text{-base:BIT}, \\
<\text{s-length:eval-bit-l}(da_2)>))
\]

4.3.6 \text{infix-on}(op_1, op_2, opor)

This function takes two operands with "matching" attributes (i.e. which had been converted to their appropriate target attributes), computes the attributes of the result and computes the value of the result. The value of the result of arithmetic operations is not defined in the SRL. Therefore the function exec-numeric, which depends upon the implementation was used in this document to model this situation. The rather complicated evaluation of the result precision of arithmetic data is left to the function res-ar-da(da_1, da_2, opor).

An exceptional case with exponentiation might arise. The last condition for the target scale of an operand depends on the numeric value of the second operand (see 4.3.41). This case has to be checked by this function, too. If the condition is fulfilled the first exponent has to be floated.

A further assumption had to be made with respect to comparisons. It was decided that the comparison of two bit strings should result in a bit string of the same length. The function exec-st(op_1, op_2, opor) accomplishes both comparisons of strings and bit-operations on strings. A comparison of two empty strings yields an empty string again.
Furthermore, the result of a comparison of bits is not specified by the SRL. It was assumed that the bits would also be compared according to their collating sequence.

**Definition:**

\[
\text{infix-op}(op_1, op_2, opor) = \\
\text{is-cat}(opor) \rightarrow \\
\mu_0(<s-da: \mu_0(<s-base:s-base(da_1)>), \\
<s-length:s-length(da_1) + (s-length(da_2).FILL hate>)>, \\
<s-vr:s-vr(op_1) \& s-vr(op_2)>) \\
\text{is-comp}(opor) \& \text{is-string}(da_1) \rightarrow \\
\text{exec-st}(op_1, op_2, opor) \\
\text{is-comp}(opor) \rightarrow \\
\mu_0(<s-da:LOGICAL>, \\
<s-vr:represent(LOGICAL, exec-s-comp(op_1, op_2, opor))>> \\
\text{is-bitop}(opor) \rightarrow \text{exec-st}(op_1, op_2, opor) \\
\text{is-c-power} \& \text{is-fix:s-scale}(da_1) \& (\text{num-val}(op_2)<0) \& \\
((p_1+1).\text{num-val}(op_2)-1 \times N) \rightarrow \\
\text{infix-op}(\text{convert}(<s-base:EMPTY>), \\
<s-scale:FLT>, \\
<s-mode:EMPTY>, \\
<s-prec:EMPTY>), \\
op_1), \\
op_2, opor) \\
\text{is-arithop}(opor) \rightarrow \\
\mu_0(<s-da:res-ar-da(da_1, da_2, opor)>, \\
<s-vr:represent(res-ar-da(da_1, da_2, opor), \\
\text{exec-numeric}(op_1, op_2, opor)>) \\
\]

The function "exec-numeric" is discussed in 4.3.623, the function "represent" in 4.3.321.
**Note:** When considering the function `infix-op` one has to bear in mind, that this function will also be used to interpret prefix operations (see 4.3.7).

4.3.61 \( \text{exec-st}(o_1, o_2, o_{por}) \)

This is a rather universally used function which takes two strings of equal length and performs the operation designated by `opor` on each pair of elements (bits or characters) of the operands. The operation may be a comparison or a bit-string-infix-operation. Additionally, the function is applicable to negation. In this case, the second operand has to be equal to the first, but will not be used.

If the input are two strings of zero length, the result is in all cases the empty string again.

**Definition:**

\[
\text{exec-st}(o_1, o_2, o_{por}) = \\
\text{s-length}(d_{a_1}) = 0 \\ \\
\mu_o(<s-da: \mu_o(<s-base:BIT>,<s-length:0>), \\
<s-vr: < >>) \\
T \\ \\
\mu_o(<s-da: \mu_o(<s-base:BIT>, \\
<s-length:s-length(d_{a_1})> \\
<s-vr: \mu_o(<\text{elem}(i):\text{exec-i-st}(i,o_1,o_2,o_{por}) \\
| 1\leq i\leq \text{s-length}(d_{a_1})}>)
\]

4.3.641 \( \text{exec-i-st}(i,o_1,o_2,o_{por}) \)

This function executes the operation for the i-th positions in the strings.
Definition:

exec-i-st(i, op₁, op₂, opor) =
  represent(LOGICAL,
    exec-single(string-val(elem(i, s-vr(op₁))),
                string-val(elem(i, s-vr(op₂))),
                opor))

4.3.612  exec-single(x₁, x₂, opor)

This function performs the operation for one single position. The arguments are two elements of a bit or character string and an operator. The result is always a single bit.

Definition:

exec-single(x₁, x₂, opor) =
  is-and(opor) & is-bi(x₁) & is-bi(x₂)  →  BI
  is-or(opor) & (is-bi(x₁) v is-bi(x₂))  →  BI
  is-not(opor) & is-bo(x₁)               →  BI
  is-comp(opor)  →  exec-n-comp(collat(x₁), collat(x₂), opor)
  T                →  BO

4.3.62  exec-a-comp(op₁, op₂, opor)

This function executes the comparison of arithmetic data and pointers. Its result is a single bit.

Definition:

exec-a-comp(op₁, op₂, opor) =
  is-ptr(da₁) →  exec-ptr-comp(op₁, op₂, opor)
  is-real-s-mode(da₁) v
  is-compl-s-mode(da₁) & (is-eq v is-ne)(opor)  →
    exec-n-comp(num-val(op₁), num-val(op₂), opor)
4.3.621 exec-ptr-comp(op₁, op₂, opor)

The comparison of pointers is not fully specified. The function is left undefined for the cases, where pointers are not identical.

Definition:

\[
\text{exec-ptr-comp}(\text{op₁, op₂, opor}) =
\begin{align*}
\text{is-eq(opor)} & \land (\text{op₁} = \text{op₂}) \quad \rightarrow \text{BI} \\
\text{is-ne(opor)} & \land (\text{op₁} = \text{op₂}) \quad \rightarrow \text{BO}
\end{align*}
\]

4.3.622 exec-n-comp(n₁, n₂, opor)

This function compares the values of two real or two complex numbers.

Definition:

\[
\text{exec-n-comp}(n₁, n₂, opor) =
\begin{align*}
\text{is-eq(opor)} & \land (n₁ = n₂) \quad \rightarrow \text{BI} \\
\text{is-ne(opor)} & \land (n₁ \neq n₂) \quad \rightarrow \text{BI} \\
(\text{is-gt} \lor \text{is-ge} \lor \text{is-nl})(\text{opor}) & \land (n₁ > n₂) \quad \rightarrow \text{BI} \\
(\text{is-lt} \lor \text{is-le} \lor \text{is-ng})(\text{opor}) & \land (n₁ < n₂) \quad \rightarrow \text{BI}
\end{align*}
\]

4.3.623 exec-numeric(op₁, op₂, opor)

In the case that the operation is arithmetic, only the data-attributes of the operation are defined in the SRL, while the values are left undefined. To cope with this fact the function exec-numeric was introduced into the function infix-op; its definition is left to the implementation.
The result of this function is supposed to be a real or complex number, which is the value of the execution of the arithmetic operation with respect to a particular implementation.

1.3.63 Data-attributes of an arithmetic result

The data attributes of the result of an arithmetic operation are those of the first operand except for the precision. The rules for evaluation of the precision are rather complicated and the evaluation is delegated to the function res-ar-prec. Its result is either an integer or an integer pair.

The function below will also be used by the prefix-operation "minus". In this case the second operand will be identical to the first one.

Definition:

\[ \text{res-ar-da}(\text{da}_1, \text{da}_2, \text{opor}) = \mu(\text{da}_1, <\text{s-prec}; \text{res-ar-prec}(\text{da}_1, \text{da}_2, \text{opor})>) \]

Definition:

\[ \text{res-ar-prec}(\text{da}_1, \text{da}_2, \text{opor}) = \]

\[ \begin{align*}
&\text{is-flt\cdot s-scale}(\text{da}_1) \ &\& \ &\text{is-flt\cdot s-scale}(\text{da}_2) \quad \rightarrow \\
&\quad \max(\text{s-prec}(\text{da}_1), \text{s-prec}(\text{da}_2)) \\
&\text{is-flt\cdot s-scale}(\text{da}_1) \quad \rightarrow \quad \text{s-prec}(\text{da}_1) \\
&\text{(is-add v is-subtr)}(\text{opor}) \quad \rightarrow \\
&\hspace{1cm} <\min(N, \max(p_1-s_1, p_2-s_2) + \max(s_1, s_2) + 1), \\
&\hspace{1cm} \max(s_1, s_2)> \\
&\text{is-mult}(\text{opor}) \quad \rightarrow \\
&\hspace{1cm} <\min(N, p_1+p_2+1), s_1+s_2> \\
&\text{is-div}(\text{opor}) \quad \rightarrow \quad <N, N-p_1+s_1-s_2>
\end{align*} \]
According to the abstract syntax, there are two prefix operations: minus and negation. Both operations can be treated very advantageously by considering them as infix operations with two identical operands. The interpretation of a prefix operation reduces to a rewriting rule, which allows to treat them as infix operations. In the interpretation of infix operations some minor adjustments for prefix operations had to be made.

Definition:

\[
\text{int-prefix-op}(r_1, \text{opor}) = \\
(is\text{-minus} \lor is\text{-neg})(\text{opor}) \rightarrow \\
\text{int-infix-op}(r_1, r_1, \text{opor})
\]
4.3.8 The convert function

The convert function converts operands to different data-attributes. It was desirable to use this function for all conversions. Therefore it has to allow conversions for some or for all of the data-attributes of the operand. Characteristics to which no conversion should take place, are signalled by EMPTY (see 4.3.82).

4.3.81 Auxiliary definitions

4.3.811 Auxiliary attributes

\[
\text{is-ps-arithm} = (\langle \text{s-base} : (\text{is-bin} \mid \text{is-dec} \mid \text{is-empty}) \rangle, \\
\langle \text{s-scale} : (\text{is-fix} \mid \text{is-flt} \mid \text{is-empty}) \rangle, \\
\langle \text{s-mode} : (\text{is-real} \mid \text{is-compl} \mid \text{is-empty}) \rangle, \\
\langle \text{s-prec} : (\text{is-integer} \mid \text{is-integer-pair} \mid \text{is-empty}) \rangle
\]

\[
\text{is-ps-string} = (\langle \text{s-base} : (\text{is-bit} \mid \text{is-char} \mid \text{is-empty}) \rangle, \\
\langle \text{s-length} : (\text{is-expr} \mid \text{is-empty}) \rangle)
\]

4.3.812 Mathematical functions

\[
\text{ceil}(n) = \text{ceiling of the number } n \\
\text{abs}(n) = \text{absolute value of } n \\
\text{sign}(n) = \begin{cases} +1 & \text{if } n > 0 \\ -1 & \text{if } n < 0 \end{cases}
\]

4.3.82 convert(da_t, op)

This function does the actual conversion of an operand op to the target attributes da_t. Those elements of the target which are empty, are ignored in the course of conversion and no conversion is performed on them. The definitions are straightforward descriptions of the specification in the SRL.
4.3.83 String-to-string conversion

**Definition**:  

\[
\text{convert}(d_{at}, op) = \\
\text{is-string}(da) \& \text{is-ps-string}(d_{at}) \rightarrow \text{str-str-convert}(d_{at}, op) \\
\text{is-arithm}(da) \& \text{is-ps-string}(d_{at}) \rightarrow \text{ar-str-convert}(d_{at}, op) \\
\text{is-string}(da) \& \text{is-ps-arithm}(d_{at}) \rightarrow \text{str-ar-convert}(d_{at}, op) \\
\text{is-arithm}(da) \& \text{is-ps-arithm}(d_{at}) \rightarrow \text{ar-ar-convert}(d_{at}, op)
\]
is-char-s-base(da) & is-bit-s-base(da) →
((s-length(da) = 0) → < >
T → str-str-convert(da,
μ₀(<s-da:μ₀(<s-base:BIT>,
<s-length:s-length(da)>)
<s-vr:μ₀({<elem(i):
single-c-b-conv(elem(i,s-vr(op)))> | 1≤i≤s-length(da)}>)})

is-bit-s-base(da) & is-char-s-base(da) →
(s-length(da) = 0) → < >
T → str-str-convert(da,μ₀(<s-da:μ₀(<s-base:CHAR>,
<s-length:s-length(da)>)
<s-vr:μ₀({<elem(i):
single-b-c-conv(elem(i,s-vr(op)))> | 1≤i≤s-length(da)}>)})

4.3.831 extend(op,n₁,x₁)

The arguments are an operand of string type, an integer and a bit or character.

The function extends or shortens the string, represented by op to n₁. If extension takes place, the value is extended by x₁'s.

Definition:

extend(op,n₁,x₁) =
μ₀(<s-da:μ₀(<s-base:s-base(da)>,
<s-length:n₁>),
<s-vr:{<elem(i) = x₁ | 1≤i≤n₁> &
((1≤i≤min(n₁,s-length(da))) →
(x₂ = elem(i,s-vr(op))),
((min(n₁, s-length(da)) <i≤n₁) →
(x₂ = x₁))}))}→

μ₀(<s-base:s-base(da)>,
<s-length:n₁>),
<s-vr:{<elem(i) = x₁ | 1≤i≤n₁> &
((1≤i≤min(n₁,s-length(da))) →
(x₂ = elem(i,s-vr(op))),
((min(n₁, s-length(da)) <i≤n₁) →
(x₂ = x₁))}))}→
4.3.832 Conversions of single bits and characters

Definition:

\[ \text{single-c-b-conv}(\text{vr}) = \begin{cases} \text{is-cho}(\text{s-vr}(\text{op})) & \rightarrow \text{BO} \\ \text{is-chi}(\text{s-vr}(\text{op})) & \rightarrow \text{BI} \end{cases} \]

Definition:

\[ \text{single-b-c-conv}(\text{vr}) = \begin{cases} \text{is-bo}(\text{s-vr}(\text{op})) & \rightarrow \text{CHO} \\ \text{is-bi}(\text{s-vr}(\text{op})) & \rightarrow \text{CHI} \end{cases} \]

Note: The predicates \text{is-cho}, \text{is-chi}, \text{is-bo}, and \text{is-bi} have as argument a value representation. Nevertheless, the functions specified in this section return bits or characters.

4.3.84 Arithmetic-to-string conversion

Abbreviations:

\[ p_{ab} = \text{eval-bit-l}(\text{da}) \]
\[ e_{ab}(i, \text{op}) = \text{single-a-b-conv}(i, \text{num-val}( \text{convert}(\mu_0(\text{s-da}; \mu_0(\text{s-base:BIN}, \text{s-mode:REAL}, \text{s-scale:FIX}, \text{s-spec: p_{ab}, 0})), \text{op}))) \]
\[ p = \text{s-first\cdot s-prec}(\text{da}) \]
\[ s = \text{s-second\cdot s-prec}(\text{da}) \]
**Definition:**

\[
eval-bit-l(da) =
\begin{align*}
&\text{is-fix\*s-scale}(da) & \& \text{is-bin\*s-base}(da) \rightarrow \\
&\min(N, \max(p - s, 0)) \\
&\text{is-fix\*s-scale}(da) & \& \text{is-dec\*s-base}(da) \rightarrow \\
&\min(N, \max(\text{CEIL}((p - s).3,32),0)) \\
&\text{is-flt\*s-scale}(da) & \& \text{is-bin\*s-base}(da) \rightarrow \\
&\min(N, \text{s-prec}(da)) \\
&\text{is-flt\*s-scale}(da) & \& \text{is-dec\*s-base}(da) \rightarrow \\
&\min(N, \text{CEIL}(\text{s-prec}(da).3,32))
\end{align*}
\]

**Definition:**

\[
ar-str-convert(da_t',op) =
\begin{align*}
&\text{is-bit\*s-base}(da_t) \rightarrow \\
&\mu_o(<s-da:\mu_o(<s-base:BIT> \\
&<s-length:p_{ab}>)) \\
&s-vr: \{\text{elem}(i):\text{el}_{ab}(i,op) \mid 1 \& s-p_{ab}\}) \\
&\text{is-char\*s-base}(da_t) \rightarrow \text{not defined in the present document}
\end{align*}
\]

4.3.841 Evaluation of single bits

**Definition:**

\[
single-a-b-conv(i,n_1) =
\begin{align*}
&\mu_o(<s-da:\mu_o(<s-base:BIT>, \\
&<s-length:1>)) \\
&s-vr: (2,\text{CEIL}(n_1:2^i) - n_1)
\end{align*}
\]
4.3.85 String-to-arithmetic conversions

Definition:

\[
\text{str-ar-convert}(\text{da}_t, \text{op}) = \\
\text{is-char-s-base}(\text{da}_t) \rightarrow \\
\text{convert}(\text{da}_t, \text{op}) = \\
\langle \text{da}: \text{INTG} \rangle, \\
\langle \text{vr}: \text{represent}(\text{INTG}, \\
\text{single-c-a-conv}(\text{op})) \rangle)
\]

\[
\text{is-bit-s-base}(\text{da}_t) \rightarrow \\
\text{convert}(\text{da}_t, \text{op}) = \\
\langle \text{da}: \text{BINTEGER} \rangle, \\
\langle \text{vr}: \text{represent}(\text{BINTEGER}, \\
\text{single-b-a-conv}(\text{vr}(\text{op}))) \rangle)
\]

4.3.851 Auxiliary string to arithmetic conversion functions

Definition:

\[
\text{single-c-a-conv}(\text{op}) = \\
\text{s-length}(\text{da}) = 0 \rightarrow 0 \\
\text{T} \rightarrow \text{the value is left undefined}
\]

\[
\text{single-b-a-conv}(\text{vr}) = \\
\text{vr} = < > \rightarrow 0 \\
\text{T} \rightarrow (\text{is-bi}(\text{head}(\text{vr}(\text{op}))) \\
\text{1+single-b-a-conv}(\text{tail}(\text{vr}))) \\
\text{T} \rightarrow \text{single-b-a-conv}(\text{tail}(\text{vr})))
\]

Note: Both functions return a numeric value.
4.3.86 Arithmetic conversions

Definition:

\[ \text{ar-ar-convert}(\text{da}_t, \text{op}) = \]
\[ \text{s-mode}(\text{da}_t) = \text{s-mode}(\text{da}) \land \text{is-empty}\cdot\text{s-prec}(\text{da}_t) \]  
\[ \text{ar-b-sc-conv}(\text{da}_t, \text{op}) \]
\[ \text{is-real}\cdot\text{s-mode}(\text{da}_t) \land \text{is-compl}\cdot\text{s-mode}(\text{da}) \land \]
\[ \text{is-empty}\cdot\text{s-prec}(\text{da}_t) \]  
\[ \text{ar-b-sc-conv}(\text{da}_t, \mu_o (\text{da}_t, \mu_o (\text{s-mode:REAL}, \text{s-scale:scale}(\text{da}), \text{s-base:base}(\text{da}), \text{s-prec:prec}(\text{da})), \text{s-vr:vr}(\text{op}))))) \]
\[ \text{is-compl}\cdot\text{s-mode}(\text{da}_t) \land \text{is-real}\cdot\text{s-mode}(\text{da}) \]  
\[ \text{ar-b-sc-conv}(\text{da}_t, \mu_o (\text{s-mode:COMPL}, \text{s-scale:scale}(\text{da}), \text{s-base:base}(\text{da}), \text{s-prec:prec}(\text{da})), \text{s-vr:vr}(\text{op}, 0))) \]

Note: The conversions of scale and base are left implementation-defined, if a target precision is given.

4.3.86.1 Base and scale conversions

Definition:

\[ \text{ar-b-sc-conv}(\text{da}_t, \text{op}_1) = \]
\[ \mu_o (\text{da}_t, \mu_o (\text{s-prec:prec}(\text{da}_t, \text{op}_1), \text{s-vr:vr}'(\text{op}_1))) \]

Note: The value of the converted operand (s-vr'(op_1)) is left implementation-defined.
4.3.862 Precision of the converted operand

Definition:

\[ ar\text{-}b\text{-}sc\text{-}p(da_t, op_1) = \]
\[ \text{is-fix}s\text{-}scale(da_1) \& \text{is-fix}s\text{-}scale(da_t) \longrightarrow \]
\[ ar\text{-}b\text{-}sc\text{-}1(da_t, op_1) \]
\[ \text{is-fix}s\text{-}scale(da_1) \& \text{is-flt}s\text{-}scale(da_t) \longrightarrow \]
\[ ar\text{-}b\text{-}sc\text{-}2(da_t, op_1, p_1) \]
\[ \text{is-flt}s\text{-}scale(da_1) \& \text{is-flt}s\text{-}scale(da_t) \longrightarrow \]
\[ ar\text{-}b\text{-}sc\text{-}2(da_t, op_1, s\text{-}prec(da_1)) \]

4.3.863 Precision for fixed-point target

Definition:

\[ ar\text{-}b\text{-}sc\text{-}1(da_t, op_1) = \]
\[ \text{is-fix}s\text{-}scale(da) \& (s\text{-}base(da) = s\text{-}base(da_t)) \longrightarrow <p_1, s_1> \]
\[ \text{is-fix}s\text{-}scale(da) \& \text{is-dec}s\text{-}base(da) \longrightarrow \]
\[ <\min(\text{ceil}(p_1, 3, 32), N), \]
\[ \text{ceil}(\text{abs}(s_1), 3, 32).\text{sign}(s_1) > \]
\[ \text{is-fix}s\text{-}scale(da) \& \text{is-bin}s\text{-}base(da) \longrightarrow \]
\[ <\text{ceil}(p_1, 3, 32), \]
\[ \text{ceil}(\text{abs}(s_1), 3, 32).\text{sign}(s_1) > \]

4.3.864 Precision for floating-point target

Definition:

\[ ar\text{-}b\text{-}sc\text{-}2(da_t, op_1, v) = (s\text{-}base(da) = s\text{-}base(da_t)) \longrightarrow v \]
\[ \text{is-dec}s\text{-}base(da) \longrightarrow \min(\text{ceil}(p, 3, 32), N) \]
\[ \text{is-bin}s\text{-}base(da) \longrightarrow \text{ceil}(p, 3, 32) \]
4.3.87 truth(op)

Definition:

\[
\text{truth}(op) = (\exists i) (\text{is-bi}(\text{elem}(i, s-vr(\text{convert}(\mu_\circ (<s-base:BIT>, <s-length:EMPTY>), op)))))
\]
4.4 Interpretation of References and Expressions

This chapter defines the case distinctions for the interpretation of an aggregate expression together with an expansion information. The interpreting instruction is \( \text{int-expr}(t,ei) \), where \( t \) is the text of the expansion and \( ei \) the expansion information. The iteration of this interpretation (for all scalar parts of the operands in the expression) is described with the assignment statement (chapter 5.7) and the interpretation of arguments to functions and procedures (chapter 4.5).

The case distinctions made in the definition of the instruction \( \text{int-expr}(t,ei) \) are distinctions according to the syntactic properties of \( t \). The distinctions made in the definition of \( \text{int-ref}(t,ei) \) are distinctions according to the attributes of the object referenced to in \( t \). Both instructions return an object with the predicate

\[
<s-op:is-op>, <s-ei:is-ei>.
\]

The expansion information is described in chapter 4.2.4, operands and operations are described in 4.3, the instruction \( \text{int-name} \) in 4.2.1, the instruction \( \text{int-variable} \) in 4.2.5, the instruction \( \text{expand-value} \) in 4.2.7, the instruction \( \text{int-call} \) in 5.4.2.

4.4.1 \( \text{int-expr}(t,ei) \)

**Definition:**

\[
\text{int-expr}(t,ei) = \\
\text{is-infix-expr}(t) \& \text{is-power}(s-operator(t)) \& \text{is-constant}(s-op-2(t)) \rightarrow \\
\text{int-infix-expr}(r_1,r_2,C-POWER) ; \\
r_1 : \text{int-expr}(s-op-1(t),ei) , \\
r_2 : \text{int-expr}(s-op-2(t),ei)
\]
is-infix-expr(t)  →
  int-infix-expr(r₁, r₂, s-operator(t))
  r₁ : int-infix-expr(s-operator(s-op-1(t),ei))
  r₂ : int-infix-expr(s-operator(s-op-2(t),ei))

is-prefix-expr(t)  →
  int-prefix-expr(r, s-operator(t));
  r : int-prefix-expr(s-operator(s-op(t),ei))

is-ref(t)  →  int-ref(t,ei)

is-constant(t)  →  RETURNS : μ่อย<s-op(t),<s-ei:ei>)

is-isub(t)  →  RETURNS : μ่อย<s-op:sub(s-i(t))(E),<s-ei:ei>)

Note: The first alternative in the definition is necessary because of the special rules for the operation exponentiation with the second operand being a constant.

4.4.2 int-ref(t,ei)

Abbreviations:

attrₜ = is-simple-ref(t)  →
  s-attr(s-id(t)(E))(AT)

Definition:

int-ref(t,ei) =
  is-ptr-qual-ref(t) v
  is-variable(attrₜ) & ~is-isub-def(attrₜ)  →
    expand-value(<gen,ei>);
    gen : int-variable(t,E)
is-sub-def(\texttt{attr}_t) \rightarrow
\texttt{expand-value}(r);

\quad r : \texttt{isub-action}(s-id(t)(E), rl, ei);

\quad rl : \texttt{int-name}(t, E)

is-built-in(\texttt{attr}_t) \rightarrow
\texttt{int-built-in}(t, ei)

is-generic(\texttt{attr}_t) \rightarrow
\texttt{int-generic}(t, ei)

is-entry(\texttt{attr}_t) & \texttt{is-empty}(s-param-list(\texttt{attr}_t)) \rightarrow
\texttt{int-call}(\texttt{attr}_t, (s-id(t)(E))(DN), arg-list);

\quad \{\text{elem}(i) \cdot \text{arg-list} :

\quad \quad \texttt{int-arg}(\text{elem}(i, \text{s-arg-list}(t)),

\quad \quad \quad \text{elem}(i, \text{s-param-list}(\texttt{attr}_t))) \mid

\quad \quad \quad \quad 1 \leq i \leq \text{length}(\text{s-arg-list}(t))\}

is-entry(\texttt{attr}_t) \rightarrow
\texttt{int-call}(\texttt{attr}_t, (s-id(t)(E))(DN), arg-list);

\quad \{\text{elem}(i) \cdot \text{arg-list} :

\quad \quad \texttt{int-arg}(\text{elem}(i, \text{s-arg-list}(t)), \text{EMPTY}) \mid

\quad \quad \quad 1 \leq i \leq \text{length}(\text{s-arg-list}(t))\}

is-label-const(\texttt{attr}_t) \rightarrow
\text{RETURNS} : \mu_0(<s-op\mu_0(<s-da:attr>_t>,

\quad <s-vr: (s-id(t)(E))(DN)>,

\quad <s-ei: ei> )}
4.5 Interpretation of Arguments

Arguments to functions and procedures are interpreted by the instruction \texttt{int-arg(t,pa)}, where \( t \) is the text of the argument and \( pa \) are the attributes (possibly empty) declared for the corresponding parameter in the ENTRY-declaration of the called procedure. The instruction returns a value with the predicate

\[
<s\text{-}attr:is\text{-}attr>, <s\text{-}den:is\text{-}den>
\]

i.e., an object specifying attributes and denotation of the evaluated argument.

There are three major cases in passing an argument:

1) the passing of the whole denotation of the name of the argument. This case comprises the passing of entries, files and controlled variables;
2) the passing of part of the denotation of an aggregate. This is the case for references to variables, when the data-attributes of the referenced part match the data-attributes specified for the parameter;
3) the passing of a dummy variable. The argument expression is evaluated and assigned to a newly created dummy variable.

The rest of this section gives the definition of the instruction \texttt{int-arg} which makes the distinction between the three cases defined above. In 4.5.1 those instructions are defined which are used for the different cases of parameter interpretation. In 4.5.2 the various predicates and functions are defined that are used for comparing and combining argument and parameter attributes.

The predicate \texttt{paren(t)} is true, if \( t \) is a parenthesized reference, false otherwise. The predicate is not effectively defined in the present document.
4.5.1 The instruction \texttt{int-arg}

**Abbreviations:**

$$
n_t = \text{is-simple-ref}(t) \rightarrow s\text{-id}(t)(E)$$

$$
d_t = n_t(DN)$$

$$
\text{attr}_t = s\text{-attr}\cdot n_t(\text{AT})$$

$$
da_t = s\text{-da}(\text{attr}\cdot t)$$

$$
da_p = s\text{-da}(p)$$

$$
rl_t = \text{is-simple-ref}(t) \rightarrow$$

$$
\text{ref-list}(da_t, s\text{-arg-list}(t), s\text{-qual-list}(t))$$

$$
dar_t = \text{new-da}(da_t, rl_t)$$

**Definition:**

\[
\text{int-arg}(t, pa) = \begin{cases} 
\text{is-simple-name}(t) & \text{&} \neg \text{is-paren}(t) \text{&} \text{direct-match}(\text{attr}_t, pa) \rightarrow \\
\text{RETURNS} : f'\left(\begin{array}{l}
\langle \text{s-attr:comb-dir-attr}(\text{attr}_t, pa) >, \\
\langle \text{s-den:dt} >
\end{array}\right) \\
\text{is-ref}(t) & \text{&} \neg \text{is-paren}(t) \text{&} \neg \text{is-label-const}(\text{attr}_t) \text{&} \\
\neg \text{is-entry}(\text{attr}_t) & \text{&} (\text{is-data-param-descr}(pa) \text{&} \\
\neg \text{is-ctf\cdot s-stg-cl}(pa) \text{&} \text{match-1}(\text{dar}_t, da_p) \text{&} \text{is-empty}(pa) \rightarrow \\
\text{pass-gen} \left(\begin{array}{l}
\text{gen}, da_p \\
\text{gen} : \text{int-variable}(t, E)
\end{array}\right) \\
\text{is-expr}(t) & \text{&} \\
\left(\text{is-data-param-descr}(pa) \text{&} \neg \text{is-ctf\cdot s-stg-cl}(pa) \text{&} \text{is-empty}(pa) \right) \rightarrow \\
\text{dummy-assign}(r); \\
\text{x:allocate-dummy}(a, b, pa, opl); \\
a : \text{un-a} \\
b : \text{un-b} \\
opl : \text{collect-aggr}(t, \langle >)
\end{cases}
\]
4.5.11 pass-gen\((gen, da_p)\)

The instruction checks the matching of the data-attributes of the generation to be passed and the data-attributes of the parameter; it returns the attributes, and the aggregate-name which denotes the aggregate formed from the generation. A corresponding entry is made in AG.

Definition:

\[
\text{pass-gen}(gen, da_p) = \\
\text{match-2}(s-da(gen), da_p) \\
\text{RETURNS : } \mu_0(\langle s-attr \rangle \mu_0(\langle s-da : s-da(gen) \rangle, \\
\langle s-dens : s-dens(gen) \rangle, \\
\langle s-stg-class : \text{EMPTY} \rangle), \\
\langle s-den : \text{head}(B) \rangle) \\
\text{s-ag : } \mu(AG, \langle \text{head}(B) : gen \rangle) \\
\text{s-b : } \text{tail}(B)
\]

4.5.12 collect-aggr\((t, opl)\)

The instruction evaluates the expressions in sequence for each of the scalar parts of its operands, and collects the results in a list. After completion of the evaluation it returns the list. See 4.4 for evaluation of expressions.

Abbreviation:

\[
\text{last}(opl) = (\text{length}(opl) \geq 1) \rightarrow \text{elem}(\text{length}(opl), opl)
\]
Definition:

\[
\text{collect-aggr}(t, opl) = \\
opl = < > \quad \rightarrow \\
\text{collect-aggr}(t, opl); \\
\text{elem}(1) \cdot opl : \text{int-expr}(t, \text{EMPTY}) \\
\text{is-exhaust}(\text{update}(s-ei\cdot\text{last}(opl))) \rightarrow \text{RETURNS} : opl \\
\]

\[
T \rightarrow \text{collect-aggr}(t, opl); \\
\text{elem}(\text{length}(opl) + 1) \cdot opl : \\
\text{int-expr} \ (t, \text{update}(s-ei\cdot\text{last}(opl)))
\]

Note: The result of an expression evaluation contains the expansion-information; which may be selected by s-ei. The expansion-information for a new interpretation is therefore obtained by updating the expression-information \(s-ei\cdot\text{last}(opl)\) of the previous one.

4.5.13 allocate-dummy(a,b,pa,opl)

The instruction allocates a dummy variable, with a,b, and data attributes combined from pa and the attributes collected from opl (pa may be empty, missing expressions in pa are replaced by corresponding values taken from the collected attributes). The instruction returns a pair containing the denotation of the dummy and the list of operands contained in opl (which is obtained by \text{collect-aggregate}). See 4.5.29 for collecting of the data-attributes from the list opl by the function collect-attr, and 4.8 for allocation of a variable.

Abbreviation:

\[
dummy-da = \text{comb-da}(\text{collect-attr}(s-si\cdot s-ei\cdot\text{head}(opl), opl), pa)
\]

Note: dummy-da are the data-attributes of the dummy to be allocated.
**Definition:**

\[
\text{allocate-dummy}(a, b, pa, opl) =
\]

\[
\text{RETURNS } : \langle b, opl \rangle
\]

\[
\begin{align*}
\text{s-s} & : \mu (S, \{ <\text{loc}(a, i): \text{EMPTY}> \mid 1 \leq i \leq \text{elem-extent}(\text{dummy-da}) \}) \\
\text{s-ag} & : \mu (AG, <b: <\text{m-gen}(\text{dummy-da}, \\
\text{comb-dens}(pa, \text{dummy-da}), a)>)\rangle
\end{align*}
\]

4.5.14 **dummy-assign**\((<b, opl>)\)

The instruction starts the assignment of the operands in the list \(opl\) to the dummy-aggregate \(b\). It returns attributes and denotation of the dummy-aggregate.

**Abbreviation:**

\[g_b = \text{head}\cdot b(AG)\]

**Definition:**

\[
\text{dummy-assign}(<b, opl>) =
\]

\[
\text{pass} \left( \mu_o (\langle \text{s-attr}: \mu_o (\langle \text{s-da}: \text{s-da}(g_b), \langle \text{s-dens}: \text{s-dens}(g_b) \rangle \\
\text{s-stg-class: DUMMY>)\rangle, <\text{s-den}: b > ) \rangle \right) ;
\]

\[
\{ \text{assign} (\text{sub-gen}(g_b, \text{s-el} \cdot \text{si} \cdot \text{elem}(i, opl))), \\
\text{s-op} \cdot \text{elem}(i, opl)) \mid 1 \leq i \leq \text{length}(opl) \} \]
4.5.2 Matching and combining functions for attributes of arguments and parameters

4.5.21 direct-match(attr, pa)

The function is true if attr and pa match, where attr and pa are entry or file attributes, or attributes of a controlled variable.

Definition:

\[
direct-match(attr, pa) = \\
(is-entry(attr) \& (is-empty(pa) \lor (is-entry(pa) \& entry-match(attr, pa)))) \lor \\
(is-file(attr) \& (is-empty(pa) \lor is-file(pa))) \lor \\
(is-ctr(s-stg-cl(attr)) \& \\
(is-empty(pa) \lor (is-ctr(s-stg-cl(pa)) \& ctr-match(attr, pa))))
\]

Note: If the parameter description pa is empty, matching is assumed.

4.5.22 entry-match(attr, pa)

The function is true if entry attribute attr of argument matches entry-description of parameter.
Definition:

\[
\text{entry-match}(\text{attr}, \text{pa}) = \begin{align*}
\text{is-entry}(\text{attr}) & \land \text{is-entry}(\text{pa}) \\
\text{entry-match}(\text{s-ret-type}(\text{attr}), \text{s-ret-type}(\text{pa})) & \land \\
\lnot \text{is-empty}(\text{s-param-list}(\text{attr})) & \land \\
\lnot \text{is-empty}(\text{s-param-list}(\text{pa})) & \land \\
(\text{length}(\text{s-param-list}(\text{attr})) = & \text{length}(\text{s-param-list}(\text{pa}))) & \land \\
\bigwedge_{i=1}^{\text{l}_1} \text{entry-match}(\text{elem}(i, \text{s-param-list}(\text{attr})), & \text{elem}(i, \text{s-param-list}(\text{pa}))) \\
\text{is-empty}(\text{attr}) \lor & \text{is-empty}(\text{pa}) \lor \\
\text{is-file}(\text{attr}) & \land \text{is-file}(\text{pa}) \rightarrow T & \rightarrow \text{attr} = \text{pa}
\end{align*}
\]

where \(l_1 = \text{length}(\text{s-param-list}(\text{attr}))\)

Note: Matching is assumed for the return type if it is not specified in attr or in pa; for the parameter-description list if it is not specified in attr or in pa, or if it is specified in both and corresponding positions are either equal, or one of them is empty.

4.5.23 \(\text{ctr-match}(\text{attr}, \text{pa})\)

The function is true if the attributes attr and pa of a controlled argument and a controlled parameter match. Complete matching is required except for expressions for bounds and lengths.

Definitions:

\[
\text{ctr-match}(\text{attr}, \text{pa}) = \begin{align*}
\text{s-dens}(\text{attr}) & = \text{s-dens}(\text{pa}) \land \text{da-ctr-match}(\text{s-da}(\text{attr}), \text{s-da}(\text{pa}))
\end{align*}
\]
match-1(dar, dap)

The function is true if the data-attributes dar of the referred argument match the data-attributes of the parameter. Complete matching is required with the exception of bounds, and the exception of lengths if unspecified in dap.

Definition:

match-1(dar, dap) =

is-array(dar) & is-array(dap) →

match-1(s-elem(dar), s-elem(dap))

is-struct(dar) & is-struct(dap) & (order(dar) = order(dap)) →

\[
\begin{align*}
\text{order}(dar) \\
\text{is-string}(dar) & \text{ & is-string}(dap) \\
\end{align*}
\]

\[
\begin{align*}
\delta (dar, s\text{-length}) &= \delta (dap, s\text{-length}) \\
T & \quad \text{dar} = \text{dap}
\end{align*}
\]
4.5.25  match-2(da,dap)

The function compares attributes of data-argument and parameter after the call. Complete matching is required except for bounds and lengths, if they are unspecified in dap.

**Definition:**

match-2(da,dap) =

is-array(da) & is-array(dap) →

match-2(s-elem(da), s-elem(dap)) &
(is-empty(s-lbd(dap)) & is-empty(s-ubd(dap)) v
(s-lbd(da) = s-lbd(dap) & is-ubd(da) = s-ubd(dap)))

is-struct(da) & is-struct(dap) & (order(da) = order(dap)) →

order(da)

\[ E_t \]

match-2(s-da\_suc(i,da), s-da\_suc(i,dap))

is-string(da) & is-string(dap) & is-empty(s-length(dap)) →

\[ \delta \text{(da, s-length)} = \delta \text{(dap, s-length)} \]

T → da = dap

4.5.26  comb-da(da,pa)

Gives the data-attributes of pa (except when pa is empty); if all bounds in pa are unspecified, these are inserted from the corresponding places in da.

**Definition:**

comb-da(da,pa) =

is-empty(pa) → da

si-match(m-si(da), m-si(s-da(pa))) &

(\( \forall \kappa \) (\( \kappa \in \text{bds-set}(s-da(pa)) \Rightarrow \text{is-empty}(\kappa\cdot s-da(pa)) \) →

\( \mu (s-da(pa), \{ \langle \kappa: \kappa(da) \rangle | \kappa \in \text{bds-set}(s-da(pa)) \}) \)

m-si(s-da(pa)) = m-si(da) → s-da(pa)

**Note:** It is assumed that all string-lengths are specified in pa.
4.5.261 \( \text{si-match}(s_{i_1}, s_{i_2}) \)

The function compares the structure-informations \( s_{i_1} \) and \( s_{i_2} \). Matching is assumed for bounds if they are equal, or unspecified in \( s_{i_2} \).

**Definition:**

\[
\text{si-match}(s_{i_1}, s_{i_2}) = \\
is-array(s_{i_1}) \land is-array(s_{i_2}) \\
\quad si-match(s_{\text{elem}(s_{i_1})}, s_{\text{elem}(s_{i_2})}) \land \\
\quad ((s_{\text{lbd}}(s_{i_1}) = s_{\text{lbd}}(s_{i_2}) \land s_{\text{ubd}}(s_{i_1}) = s_{\text{ubd}}(s_{i_2})) \lor \\
\quad (is-empty(s_{\text{lbd}}(s_{i_2})) \land is-empty(s_{\text{ubd}}(s_{i_2})))) \\
is-\text{struct}(s_{i_1}) \land is-\text{struct}(s_{i_2}) \land (order(s_{i_1}) = order(s_{i_2})) \\
\exists i \quad si-match(succ(i, s_{i_1}), succ(i, s_{i_2})) \\
\land is-empty(s_{i_1}) \land is-empty(s_{i_2})
\]

4.5.27 \( \text{comb-dir-attr}(\text{attr}, \text{pa}) \)

Gives the merged attributes \( \text{attr} \) and \( \text{pa} \), if they are entry, file or attributes of a controlled variable.

**Definition:**

\[
\text{comb-dir-attr}(\text{attr}, \text{pa}) = \\
(is\text{-entry}(\text{attr}) \land is\text{-entry}(\text{pa})) \lor is\text{-empty}(\text{pa}) \lor \\
(is\text{-ctr}(s\text{-stg-cl}(\text{attr})) \land is\text{-ctr}(s\text{-stg-cl}(\text{pa}))) \rightarrow \text{attr} \\
is\text{-file}(\text{attr}) \land is\text{-file}(\text{pa}) \rightarrow \text{combine-file-attr-1}(\text{attr}, \text{pa})
\]

**Note:** \( \text{combine-file-attr-1} \) is not yet defined in the present document.
4.5.28  comb-dens(da,pa)

Definition:

\[ \text{comb-dens}(da,pa) = \text{is-empty}(pa) \rightarrow \text{def-dens}(da) \rightarrow \text{s-dens}(pa) \]

where

\[ \text{def-dens}(da) = \text{is-array}(da) \rightarrow \text{ALIGNED} \rightarrow \text{PACKED} \]

4.5.29  collect-attr(si,opl)

The function gives the data-attributes with structure-information si and those scalar attributes that are taken from the corresponding element of opl.

Definition:

\[
\text{collect-attr}(si,opl) = \\
(lx)(\text{is-da}(x) \& (m-si(x) = si) \& \\
((\forall \eta)(\text{is-scalar-part}(x,\eta) \rightarrow \\
((\text{new-da}(x,\eta) = s-attr\cdot s-op\cdot elem \\
\text{(scalar-index}(x,\eta), opl)) \lor \\
\text{is-label}(\text{new-da}(x,\eta)) \& \\
\text{is-label-const}(s-attr\cdot s-op\cdot elem(\text{scalar-index}(x,\eta), opl))))))
\]
4.6 Interpretation of Integer and Scalar Expressions

4.6.1 Integer-expressions

For convenience there is the instruction \texttt{int-integer-expr(t,env)} which evaluates the scalar expression \(t\) in the environment \(\text{env}\) and returns the result converted to integer. The instruction is used for the evaluation of bounds, lengths, subscripts etc., which in many cases has to be done in an environment that is not the current one.

It is necessary to establish a new active block for the evaluation. For a complete description of general block-entry and exit see 5.4. \texttt{INTEGER} is the abbreviation for the data attribute

\[
\text{INTEGER} = \nu_0\left(\langle s\text{-mode:REAL}\rangle,\langle s\text{-base:DEC}\rangle, \\
\langle s\text{-scale:FIX}\rangle,\langle s\text{-prec:DEF-PREC}\rangle\right)
\]

where DEF-PREC is the default-precision for integers.

Definition:

\[
\begin{align*}
\text{int-integer-expr}(t,\text{env}) &= \\
\text{s-e} & : \text{env} \\
\text{s-d} & : \text{stack}(\xi) \\
\text{s-cs} & : \text{CS} \\
\text{s-id} & : \{\} \\
\text{s-si} & : \nu_0\left(\langle s\text{-ret-type:INTEGER}\rangle,\langle s\text{-ret-point:V}\rangle\right) \\
\text{s-c} & : \text{integer-return}(r); \\
r & : \text{int-sc-expr}(t)
\end{align*}
\]
4.6.11 integer-return(r)

Definition:

\[
\begin{align*}
\text{s-e} & : s-e(D) \\
\text{s-d} & : s-d(D) \\
\text{s-cs} & : s-cs(D) \\
\text{s-id} & : s-id(D) \\
\text{s-ri} & : s-ri(D) \\
\text{s-c} & : \mu_0(s-c(D),<s-ret-point(RI):convert(INTEGER,r)>)
\end{align*}
\]

4.6.2 Scalar expressions

The instruction \text{int-sc-expr(t)} is applicable to scalar expressions and returns scalar operands.

Definition:

\[
\begin{align*}
\text{int-sc-expr(t)} &= \\
& \text{pass-op(r);} \\
& \quad r: \text{int-expr(t,SC-EI)}
\end{align*}
\]

Definition:

\[
\begin{align*}
\text{pass-op(r)} &= \\
& \text{RETURNS : s-op(r)}
\end{align*}
\]
4.7 Scalar Assignment

Assignment of scalar operands to scalar generations is performed by the instruction `assign(gen, op)`. If gen is a proper generation, then the operand op is converted to fit the data-attributes of gen (if possible), and the value-representation of the result is entered into the storage $S$. If gen is a pseudo-generation, i.e. the result of the interpretation of a pseudo-variable, the assignment is performed by the instruction `pseudo-assign(gen, op)` (see §4.8).

The value representation of string operands is a list of representations (of bits or characters). The elements of this list are assigned to separate storage locations in $S$. In the case of assignment to variable-length string generations, unused storage locations get non-existent values. In the case of label-assignment, a check is made whether the label to be assigned is within the range of the label-variable.

The function `convert(da, op)` is explained in chapter §4.3.

Abbreviations:

\[
d_{a} = s-da(gen) \\
a = s-addr(gen) \\
il = s-il(gen) \\
op_{C} = convert(da, op) \\
o_{0} = head(il)
\]
**Definition:**

\[
\text{assign}(gen, op) = \begin{cases} 
\text{pseudo-assign}(gen, op) & \text{is-ps-gen}(gen) \\
\text{is-string}(da) & \end{cases}
\]

\[
\text{s-s:}\mu(\text{loc}(a,i_0+i-1),\text{loc}(a,i_0+i-1))\text{,(s,settings-value:x)} | \\
(1 \leq \text{length}\cdot s\text{-vr}(op_c) & \text{x} = \text{elem}(i,s\text{-vr}(op_c)) v \\
\text{(length}\cdot s\text{-vr}(op_c)+1 \leq s\text{-length}(da)) & \text{x} = \Omega \} 
\]

\[
\text{is-arith}(da) \quad \rightarrow \\
\text{s-s:}\mu(\text{loc}(a,i_0),\text{loc}(a,i_0))\text{,(s,(s-value:s\text{-vr}(op_c))}) 
\]

\[
\text{is-pointer}(da) & \text{is-pointer}\cdot s\cdot da(op) v \\
\text{is-rest-label}(da) & \text{is-label-const}\cdot s\cdot da(op) & \\
(\exists id)(id \epsilon s\cdot label\cdot const\cdot set(da) & s\cdot vr(op) = (id(E))(DN))v \\
\text{is-unrest-label}(da) & \text{is-label-const}\cdot s\cdot da(op) \quad \rightarrow \\
\text{s-s:}\mu(\text{loc}(a,i_0),\text{loc}(a,i_0))\text{,(s,(s-value:s\text{-vr}(op_c))}) 
\]

**Note:** The instruction is undefined if gen had not been allocated.
4.8 Interpretation of Pseudo-Variables

Pseudo-variables may occur as the left-side of the assignment-statements and as control variable in iterated statement-lists. There are two steps of their interpretation:
1. the establishment of the pseudo-generation,
2. the assignment of a value to the pseudo-generation.
The two steps are performed by the instruction int-ps-leftpart(t,ei), where t is the reference to the pseudo-variable and ei is the expansion information, and pseudo-assign(ps-gen,op), where ps-gen is the pseudo-generation and op is the operand to be assigned.

A pseudo-generation is an object

\[
is-ps-gen = (<s-ps-var-name:is-name>, <s-arg-list:is-object-list>)\]

Names of pseudo-variables are: REAL, IMAG, COMPLEX, SUBSTR, PRIORITY, EVENT, ONCHAR, ONSOURCE, UNSPEC.

4.8.1 int-ps-leftpart(t,ei)

The instruction interprets the reference "t" to a pseudo-variable and returns an object \(<s-gen:is-ps-gen>, <s-ei:is-ei>\)

Abbreviations:

\[
id_t = s-id(t)\\
lt = length\cdot s-arg-list(t)\\
t_i = elem(i, s-arg-list(t))
\]

The instruction int-prop-leftpart is defined in 5.7.4.
Definition:

\[
\text{int-ps-leftpart}(t, e_i) = \\
\text{mk-ps-gen}(id_t, ol, \langle 1 \rangle); \\
\text{is-complex}(id_t) & l_t = 2 \rightarrow \\
\{ \text{mk-ps-gen}(id, ol, \langle 1, 2 \rangle) \\
\text{elem}(i, ol) : \text{int-prop-leftpart}(t_i, e_i) \} \\
\text{is-priority} \lor \text{is-onchar} \lor \text{is-onsource}(id_t) & l_t = 0 \rightarrow \\
\text{RETURNS:} \\
\mu_0(\langle s-gen: \mu_0(\langle s-ps-var-name:id_t>, \langle s-arg-list:<> >) >, \\
< s-ei:SC-EI >) \\
\]

4.8.11 \text{mk-ps-gen}(id, ol, il)

The instruction returns a pseudo-generation. id is a pseudo-variable name, ol an object-list (the list of evaluated arguments to the pseudo-variable), il is an index-list, designating those objects in ol that are expanded generations.

Abbreviations:

\[
l = \text{length}(il) \\
k_i = \text{elem}(i, il) \\
\]
Definition:

\[
\text{mk-ps-gen}(id, ol, il) = \text{RETURNS:} \\
\mu_o(\langle s-gen: \\
\mu_o(\langle s-ps-var-name:id>, \\
\langle s-arg-list: \mu(ol, \{ \langle \text{elem}(k_i): s-gen\cdot\text{elem}(k_i, ol) \rangle \mid 1 \leq i \leq l \} ) \rangle >, \\
\langle s-ei: (\text{Equ} s-ei\cdot\text{elem}(k_i, ol) \rightarrow s-ei\cdot\text{elem}(1, ol)) \rangle >)
\]

4.8.2 \text{pseudo-assign}(ps-gen, op)

The instruction performs assignment of the operand op to the pseudo-generation ps-gen. Since the rules for assignment for the various types of pseudo-variables are rather heterogeneous, the instruction is defined separately for the different types.

4.8.21 \text{s-ps-var-name}(ps-gen) = \text{REAL}

Abbreviations:

\[
\begin{align*}
\text{v} & = \text{elem}(1, s-\text{arg-list}(ps-gen)) \\
\text{da} & = s-\text{da}(v) \\
\text{a} & = s-\text{addr}(v) \\
\text{il} & = s-\text{il}(v) \\
\text{r-da} & = \mu(\text{da}, <s-\text{mode}, \text{REAL}>)) \\
\text{i}_0 & = \text{head}(\text{il})
\end{align*}
\]

Definition:

\[
\text{pseudo-assign}(ps-gen, op) = \\
\text{is-compl}\cdot s-\text{mode}(\text{da}) \rightarrow \\
\text{s-s}: \mu(S, <\text{loc}(a, i_0): \mu(\text{loc}(a, i_0)(S), \\
\quad <s-\text{first}\cdot s-\text{value}: s-\text{vr-convert}(r-da, op)>))
\]
4.8.22 \( s\)-ps-var-name(ps-gen) = IMAG

**Abbreviations:** as under 4.8.21

**Definition:**

\[
\text{pseudo-assign}(ps-gen, op) = \\
\text{assign}(\text{subst-gen}(ps-gen), op)
\]
4.8.24  \textit{s-ps-var-name(ps-gen)} = \textit{EVENT}

Abbreviations:

\begin{align*}
\text{v} &= \text{elem}(1, \text{s-arg-list}(\text{ps-gen})) \\
\text{i}_0 &= \text{head} \cdot \text{s-il}(\text{v}) \\
\text{a} &= \text{s-addr}(\text{v})
\end{align*}

Definition:

\begin{align*}
\text{pseudo-assign}(\text{ps-gen}, \text{op}) = \\
&\text{is-event} \cdot \text{s-da}(\text{v}) & \neg \exists \text{assoc-status} \cdot \text{s-value} \cdot \text{loc}(a, i_0)(S) \rightarrow \\
S = s &\text{ s-addr}(S), \langle \text{loc}(a, i_0): \mu(\text{loc}(a, i_0)(S), <\text{s-compl-status} \cdot \text{s-value}: \\
&\text{s-vr-convert}(\mu(\langle \text{s-base}: \text{BIT}, \\
&<\text{s-length}: 1 >), \text{op} >) >)
\end{align*}

4.8.25  The assignment to the pseudo-variables \textit{PRIORITY}, \textit{ONCHAR}, \textit{ONSOURCE}, \textit{UNSPEC} is undefined in the present document.
4.9 Basic Allocation and Initialization

4.9.1 The basic allocation instruction is allocate(a, b, da, dens), where a is an address-name (from the list A), b is an aggregate-name (from the list B), and da are the data-attributes of the aggregate to be allocated, and dens is the density. The instruction enters a location, i.e. an object that may be selected by a select-function loc(a, i), into the storage S for each elementary part of da and enters the newly formed generation into the aggregate-directory AG with the aggregate-name b.

Definition:

\[
\text{allocate}(a, b, da, dens) =
\]

\[
s-s : \mu(S, \{\text{loc}(a, i) : \text{EMPTY} \mid \text{isIselem-extent}(da)\})
\]

\[
s-ag : \mu(AG, <b : \text{m-gen}(da, dens, a) \cup b(AG)>)
\]

Note: 1) The result of an allocation is always a connected generation.
2) The generation is put on top of the list of generations contained in AG under the name b. It is assumed that such a list exists already at the point of allocation. If b does not yet have an entry in AG, there must be the instruction null-allocate (b) first.

4.9.11 null-allocate(b)

The instruction enters the null-generation NULL in AG.

Definition:

\[
\text{null-allocate}(b) =
\]

\[
s-ag : \mu(AG, <b : \text{NULL}>)
\]
4.9.12 \texttt{m-gen(da,dens,a)}

The function makes a generation from the data-attributes \texttt{da}, the density \texttt{dens} and the address \texttt{a}.

\textbf{Definition:}

\begin{equation}
\texttt{m-gen(da,dens,a)} = \\
\mu_o(\langle s\text{-}\texttt{da}:da, s\text{-}\texttt{dens}:dens, s\text{-}\texttt{addr}:a, \\
\langle s\text{-}\texttt{il}: \mu_o(\{ \langle \text{elem}(i):i \rangle \mid 1 \leq i \leq \text{elem-extent}(da) \}) >\rangle)
\end{equation}

4.9.2 Initialization of a variable is performed immediately after its allocation. The information necessary for initialization is taken from the initial-attribute. The abstract syntax of the initial-attribute is:

\begin{itemize}
  \item \texttt{is-init} = (\langle \texttt{s-first:is-simple-ref}, \\
  \langle \texttt{s-second:(is-init-item-list | is-call-st)} \rangle \rangle)
  \item \texttt{is-init-item} = (\texttt{is-init-const} | \texttt{is-empty} | \texttt{is-init-iter})
  \item \texttt{is-init-iter} = (\langle \texttt{s-rep-factor:is-expr}, \\
  \langle \texttt{s-item-list:is-init-item-list} \rangle \rangle)
\end{itemize}

An initial(\texttt{init}) is element of the initial-set declared for a variable. The action necessary for one initial is the assignment of the elements of the initial-item-list (after interpretation of the replication factors) to the generation defined by the simple reference; or the execution of the call-statement specified.

4.9.21 \texttt{make-vl(inl,env)}

The instruction evaluates the replication factors contained in the initial-item-list \texttt{inl} in the environment \texttt{env} and starts the expansion of \texttt{inl}.
**Definition:**

\[
\text{make-vl}(\text{inl}, \text{env}) = \\
\quad \text{eval-initial}(\text{inl}); \\
\quad \{ \text{s-rep-factor} \inl \cdot \text{inl} : \\
\quad \quad \text{int-integer-expr}(\text{s-rep-factor} \inl \cdot \text{inl}, \text{env}) | \\
\quad \quad \exists \text{s-rep-factor} \inl \cdot \text{inl} \}\]

**Note:** The instruction has the environment env as explicit argument since it is used for the evaluation of the initial-attribute of a controlled variable (the evaluation has to be performed in the environment in which the controlled variable has been declared, which may not be the current environment).

### 4.9.22 eval-initial(inl)

The instruction returns the list of initial items obtained from the interpretation of the replication factors in the initial-item-list inl.

**Definition:**

\[
\text{eval-initial}(\text{inl}) = \\
\quad \text{RETURNS : vl}(\text{inl})
\]

### 4.9.23 vl(inl)

The function makes the list of initial items obtained from the interpretation of the replication factors in inl.
Definition:

\[ vl(inl) = \]
\[ is-init-iter(inl) \rightarrow \text{concat}(s-rep-factor(inl), vl(s-item-list(inl))) \]
\[ is-init-item-list(inl) \rightarrow vl(head(inl)) \land vl(tail(inl)) \]
\[ is-init-item(inl) \rightarrow <inl> \]

where: \( \text{concat}(i, \text{list}) = (i=1) \rightarrow \text{list} \)
\[ (i>1) \rightarrow \text{list} \land \text{concat}(i-1, \text{list}) \]

4.9.24 \textbf{initial-assign}(gen, opl)

The instruction starts iteration of the assignment of the elements of opl to the scalar parts of the generation gen.

Definition:

\[ \text{initial-assign}(gen, opl) = \]
\[ \text{iterate-initial-assign}(gen, opl, \text{init-ei}(s-da(gen))) \]

4.9.24 \textbf{iterate-initial-assign}(gen, opl, ei)

The instruction iterates the assignment of the elements of opl to the scalar parts of gen. For empty elements of opl no assignment is made.
Definitions:

iterate-initial-assign(gen,opl,ei) =
  is-exhaust(ei) v (opl=<>) → null
  is-empty(head(opl)) → iterate-initial-assign(gen,tail(opl),
                           update(ei))
  T → iterate-initial-assign(gen,tail(opl),
                           update(ei));
  assign(sub-gen(gen,s-el(ei)), v);
  v:int-sc-expr(head(opl))

Note: The definition does not reflect the restriction of the elements of opl to the class given in the abstract syntax.

4.9.3 pointer(gen)

The function pointer(gen) gives the pointer operand which identifies the connected generation gen.

Definition:

pointer(gen) = is-connected(gen) →
               μ₀(<s-da:PTR>,<s-vr:μ₀(<s-da:s-da(gen)>,<s-dens:s-dens(gen)>,
                <s-addr:s-addr(gen)>,<s-il:head(s-il(gen))>))
4.10 Basic Freeing of Storage

4.10.1 free-1(gen)

This instruction frees the storage associated with the generation "gen" of a variable. This task is reduced to freeing one single address (a, 1) in storage.

Definition:

\[
\text{free-1(gen)} = \begin{cases} \text{null}; \\ \{ \text{free-2(a, 1)} \mid (a = s-addr(gen)) \land (\exists j)(\text{elem}(j, s-il(gen)) = 1) \} \end{cases}
\]

4.10.2 free-2(a, i)

The instruction frees storage given a location "a" and an index "i". The instruction takes care of areas also.

Abstract Syntax: The abstract syntax for storage reads as follows:

\[
\text{is-s} = \{ \langle \text{loc}(a, i) : (\langle \text{s-value:is-vr} \rangle \mid \langle \text{s-contents:is-gen-list}\rangle) \rangle : \text{is-a}(a) \lor \text{is-ab}(a) \land \text{is-integer}(i) \}
\]

Definition:

\[
\begin{align*}
\text{free-2}(a, i) & = \neg \text{is-area-loc}(a, i, s) \rightarrow s-s : \delta(s, \text{loc}(a, i)) \\
\text{is-area-loc}(a, i, s) & \rightarrow \\
\text{free-area}(a, i); \\
\{ \text{free-1}(\text{elem}(j, \text{s-contents:loc}(a, i)(s))) \mid 1 \leq j \leq \text{length}(\text{s-contents:loc}(a, i)(s)) \}
\end{align*}
\]
Definition:

free-area(a,i) =

\[ s \rightarrow s : \delta(g, \text{loc}(a,i)) \]

Definition:

is-area-loc(a,i,s) = \text{exists } s-\text{contents } \text{loc}(a,i)(s)

Note: Positions of areas are distinguished from other positions by the selector that selects the value. This selector is "s-contents" in the case of area and "s-value" otherwise.

4.11 Some Frequently Used Simple Instructions

4.11.1 Null Instruction

Definition:

null = I : \xi

Note: I is the identity function, i.e. the state is replaced by itself.

4.11.2 Pass Instruction

This instruction is used to pass a value unchanged to the next level in the control tree.

Definition:

pass(x) =

\text{RETURNS } x
4.11.3 Creation of names

Frequently it is the case that names have to be created that are unique. The constituents of the states $A$, $A_b$, $B$, $N$, $F$ are infinite lists of names so that there is always an infinite supply available. The method that guarantees that a name once taken will never be used again is the following: Each time a unique name is needed, the head of one of the lists is taken as the name and deleted from the list.

**Definition:**

\[
\begin{align*}
un-a &= \text{RETURNS} : \text{head}(A) \\
\hspace{0.5cm} s-a &= \text{tail}(A) \\
un-ab &= \text{RETURNS} : \text{head}(A_b) \\
\hspace{0.5cm} s-ab &= \text{tail}(A_b) \\
un-b &= \text{RETURNS} : \text{head}(B) \\
\hspace{0.5cm} s-b &= \text{tail}(B) \\
un-n &= \text{RETURNS} : \text{head}(N) \\
\hspace{0.5cm} s-n &= \text{tail}(N) \\
un-f &= \text{RETURNS} : \text{head}(F) \\
\hspace{0.5cm} s-f &= \text{tail}(F)
\end{align*}
\]
5. CENTRAL PROCESSING STATEMENTS

This chapter contains the definition of those PL/I statements which refer to a single task except input and output statements. Comparing the PL/I-machine to a real computer one can think of these statements as those statements that refer to the central processing unit.

5.1 Initial State and Program Initiation

In the following the initial state of the PL/I machine is described.

The lists of free addresses, aggregate names, unique names and file names are the untouched lists $A, A_b, \mathcal{B}, N$ and $\mathcal{F}$.

The condition status is the initial condition status and the external storage contains the initial data-sets. Any other constituent is empty except the control. The control contains the instruction:

$$\text{initial-instruction}(t_0, t_c)$$

where $\text{is-program}(t_0)$

and $t_c$ is of unspecified structure.

The second argument $t_c$ contains in some form the specification of a call-statement that is executed after the prepass (see next chapter) and initiates the proper execution of the program.

The first argument $t_0$ is the program whose abstract syntax may be specified as follows.
Abstract Syntax

is-program = ( { <id : is-ext-proc> \mid is-name(id) } )
is-ext-proc = (<s-id : is-name>,
          <s-body : is-body>)

Definition of initial state:

\[
\begin{align*}
    A &= A \\
    A_D &= A_D \\
    B &= B \\
    N &= N \\
    F &= F \\
    AT &= \text{EMPTY} \\
    S &= \text{EMPTY} \\
    AG &= \text{EMPTY} \\
    DN &= \text{EMPTY} \\
    ES &= ES_0 \\
    FU &= \text{EMPTY} \\
    FD &= \text{EMPTY} \\
    E &= \text{EMPTY} \\
    D &= \text{EMPTY} \\
    CS &= CS_0 \\
    ID &= \text{EMPTY} \\
    RI &= \text{EMPTY} \\
    C &= \text{initial-instruction}(t_0, t_c) \\
\end{align*}
\]

- list of free addresses
- list of free based addresses
- list of free aggregate names
- list of free unique names
- list of free file names
- attribute-directory
- storage
- aggregate-directory
- denotation-directory
- external storage
- file union directory
- file directory
- environment
- dump
- condition-status
- set of declared identifiers
- return information
- control
5.2 The Prepass

The prepass performs all actions to be done "before execution" and modifies the program.

The actions can roughly be described as follows:

a) All static and controlled variable declarations are associated with aggregate names. These aggregate names are mutually different except for external declarations which declare identical identifiers.

b) All external file identifiers are associated with a unique aggregate-name.

c) All external procedures are associates with an unique name.

d) The aggregate names of the controlled variables of the program are entered into the aggregate-directory together with the one-list of the null-generation.

e) The aggregate names of static variables are entered into the aggregate directory, allocated and initialized.

f) The external procedures are entered into the denotation-directory under their associated unique name.

g) All aggregate names and unique names created as described above are inserted in the program into the corresponding declarations under "s-den".

The prepass starts with the initial state of the PL/I machine and thus with the execution of the initial instruction initial-instruction(t0,tc), where t0 is the program (collection of external procedures) and tc is a piece of information that allows to
construct the initial procedure call whose execution starts the proper execution of the program. Eventually the prepass leaves the control with this initial procedure call.

The instruction initial-instruction expands into a control tree

\[ \text{prepass-execute}(t_0, t_c, iv, evf, ep); \ldots \text{etc.,} \]

where iv, evf, ep are auxiliary objects of the following nature:

- **iv** associates each selector pointing to an occurrence of a declaration of an internal, static or an internal, controlled proper variable with a unique aggregate name.

- **evf** associates each external identifier declared static or controlled or file with a unique aggregate name.

- **ep** associates each external procedure identifier with a unique name \( n \).

Definition:

\[
\text{initial-instruction}(t_0, t_c) = \text{prepass-execute}(t_0, t_c, iv, evf, ep);
\]

\[
(\{ \text{select}(\kappa) \cdot iv : \text{un-b} | \kappa \in K_o(t_0) \} \cup \{ \text{select}(id) \cdot evf : \text{un-b} | id \in ID_o(t_0) \} \cup \{ \text{select}(id) \cdot ep : \text{un-n} | id \in EP(t_0) \})
\]

5.2.1 \( K_o(t_0) \)

\( K_o \) is the set of selectors with respect to \( t_0 \) which point to declarations of internal proper variables of static or controlled storage-class.

Definition

\[
K_o(t_0) = \{ \kappa | \text{is-decl}(\kappa(t_0)) \land \text{is-int-scope}\cdot \kappa(t_0) \land
(is-static \lor \text{is-contr})(s-stg-cl\cdot s-attr\cdot \kappa(t_0)) \}
\]
5.2.2 $\text{ID}_0(t_0)$

$\text{ID}_0$ is the set of all identifiers declared external static or external controlled or external file in $t_0$.

**Definition**

$$\text{ID}_0(t_0) = \{ id \mid (\exists \ k) (\text{is-decl} \ k(t_0) \land (id(k(t_0)))=id \land \text{is-ext-scope} \ k(t_0) \land ((\text{is-static} \lor \text{is-contr}) (\text{s-stg-cl-s-attr} \ k(t_0) \lor \text{is-file-s-attr} \ k(t_0))) \}$$

5.2.3 $\text{EP}(t_0)$

$\text{EP}$ is the set of all external procedure identifiers with respect to the program $t_0$.

**Definition**

$$\text{EP}(t_0) = \{ id \mid \text{is-name}(id) \land \text{exists}(id(t_0)) \}$$
5.2.4 Prepass-execute \((t_o, t_c, iv, evf, ep)\)

The instruction expands into a series of instructions which allocate and initialize static variables, enter controlled variables together with the null generation into the aggregate directory and establish the denotation of external procedures. The body of the external procedures is modified by the function subst-den which enters the appropriate denotation for static, controlled variables and external files and external procedures into the text. The auxiliary objects iv, evf and ep are described in the comment of section 5.2. Having completed the above actions the control is left with the instruction \texttt{initial-call} \((ep, t_c)\) which then initiates the proper execution of the program.

\textbf{Definition:}

\[
\text{prepass-execute} (t_o, t_c, iv, evf, ep) =
\text{initial-call} (ep, t_c);
\]

\[
\begin{align*}
& \{ \text{int-static} (s\text{-attr} \cdot \chi (t_o), \text{select}(\chi, iv)) \mid \\
& \quad \text{exists} \cdot \text{select}(\chi, iv) \& \text{is-static} \cdot s\cdot \text{-stg-cl} \cdot s\cdot \text{-attr} \cdot \chi (t_o) \} \cup \\
& \{ \text{int-static} (\text{ex-attr}(id, t_o), \text{select}(id, evf)) \mid \\
& \quad \text{exists} \cdot \text{select}(id, evf) \& \text{is-static} \cdot s\cdot \text{-stg-cl} \cdot \text{ext-attr}(id, t_o) \} \cup \\
& \{ \text{int-contr} (\text{select}(\chi, iv)) \mid \\
& \quad \text{exists} \cdot \text{select}(\chi, iv) \& \text{is-contr} \cdot s\cdot \text{-stg-cl} \cdot s\cdot \text{-attr} \cdot \chi (t_o) \} \cup \\
& \{ \text{int-contr} (\text{select}(id, evf)) \mid \\
& \quad \text{exists} \cdot \text{select}(id, evf) \& \text{is-contr} \cdot s\cdot \text{-stg-cl} \cdot \text{ext-attr}(id, t_o) \} \cup \\
& \{ \text{int-ep} (id(ep), s\cdot \text{body}\cdot \text{id}(\text{subst-den}(t_o, iv, evf, ep)) \mid \\
& \quad \text{exists} \cdot \text{select}(id, ep) \} \}
\]
5.2.41 int-static(attr,b), pre-initial(attr,b)

Given the aggregate-name b and the attributes attr the instruction int-static allocates and initializes static variables.

Definition:

\[\text{int-static}(\text{attr}, b) = \text{pre-initial}(\text{attr}, b); \]
\[\text{allocate}(a, b, s-da(\text{attr}), s-dens(\text{attr})); \]
\[\{ a : \text{un-a}, \text{null-allocate}(b) \}\]

\[\text{pre-initial}(\text{attr}, b) = \]
\[\text{null}; \]
\[\{ \text{initial-assign} (\text{sub-gen}(\text{head} \cdot b(\text{AG})), \]
\[\text{ref-list-1}(s-da(\text{attr}), \]
\[s-qual-list \cdot s-first(x)), \]
\[v-list(s-second(x)) \mid x \in s-init(\text{attr}) \}\]

Note: since the instruction allocate expects evaluated data-attributes as its third argument, only constants are permitted in the data attribute of the static variables; the machine runs into an undefined state otherwise. A similar statement holds for initial-assign.

5.2.42 int-contr(b)

Definition:

\[\text{int-contr}(b) = \text{null-allocate}(b)\]
5.2.43 int-ep(n, body)

Definition:

\[
\text{int-ep}(n, \text{body}) = \mu(DN, n : \mu(<s-body; \text{body}>), <s-e; \text{EMPTY}>, <s-ce; \text{EMPTY}>) >)
\]

Note: a) For the denotation of a procedure the body is associated with the empty environment, i.e. all identifiers used in an external procedure must be declared in that external procedure.

b) Whenever a block is entered where an external procedure is declared the identifier is associated with a new unique name. The attribute of the new name is essentially the entry attribute of the identifier in the current block, whereas the denotation is the one established by \text{int-ep} "before execution".

5.2.44 initial-call(ep, tc)

This instruction initiates the proper execution of the program. It is assumed that the initial call statement can be constructed from \text{ep} and \text{tc} (programmer specified piece of information, e.g. control card). The attributes of the initial procedure are constructed by \varphi_1(t_c), the denotation of the procedure is constructed by \varphi_1'(t_c, ep) and the argument list of the initial call is evaluated by the instruction \text{instr-}\varphi_2(t_c).

The functions \varphi_1(t_c), \varphi_1'(t_c, ep) and the instruction \text{instr-}\varphi_2(t_c) are unspecified in PL/I (implementation defined).
**Definition:**

\[
\text{initial-call}(\text{ep}, t_c) = \\
\text{int-call}(\Phi_1(t_c), \Phi_1(t_c), \text{arg-list}); \\
\text{arg-list} : \text{instr-}\Phi_2(t_c)
\]

Note: The instruction \text{int-call} is specified in chapter 5.4.21.

**5.2.45 subst-den(t_o, iv, evf, ep)**

Given a program \(t_o\) and the auxiliary objects \(iv, evf\) and \(ep\) the function yields a modified program where the appropriate denotations for static and controlled variables and for external files and external entry declarations have been substituted.

**Definition:**

\[
\text{subst-den}(t_o, iv, evf, ep) = \\
\mu(t_o, \{ <s\text{-den}\kappa : \text{select}(\kappa, iv)> | \exists \text{select}(\kappa, iv) \} \cup \\
\{ <s\text{-den}\kappa : \text{select}(id, evf)> | \exists \text{select}(id, evf) \land \kappa \in \text{ex-k-set}(id, t_o) \} \cup \\
\{ <s\text{-den}\kappa : \text{select}(id, ep)> | \exists \text{select}(id, ep) \land \kappa \in \text{ex-k-set}(id, t_o) \} \})
\]

**5.2.451 ex-k-set(id, t_o)**

The function yields the set of all selectors which point to a declaration of a given external identifier \(id\) in a program \(t_o\).

**Definition:**

\[
\text{ex-k-set}(id, t_o) = \{ \kappa | \text{is-decl}\kappa(t_o) \land \text{is-ext-scope}\kappa(t_o) \land (id = s\text{-id}\kappa(t_o)) \}
\]
5.2.46 ext-attr(id, t₀)

It is assumed that "id" is an external and static or controlled identifier of t₀.

The function collects all declarations for that identifier and compares the attributes. If these attributes match (according to the rules of the SRL) the function yields an attribute, otherwise the function is undefined.

Definition: not yet specified.
5.3 Interpretation of statement-lists

5.3.1 Case distinctions for statements

Abstract Syntax:

\[
\begin{align*}
is-st = ( & \text{is-block} \mid \text{is-group} \mid \text{is-if-st} \mid \text{is-on-st} \mid \\
& \text{is-simple-st}) \\
is-block = (<s-decl-part:is-decl-part>, \\
& <s-cond-part:is-cond-part>, \\
& <s-label-list:is-name-list>, \\
& <s-st-list:(is-st-list|is-st-list-while | \\
& \text{is-st-list-contr}>) \\
is-st-list-while = (<s-while-expr:is-expr>, \\
& <s-st-list:is-st-list>) \\
is-st-list-contr = (<s-contr-var:is-ref>, \\
& <s-spec-list:is-do-spec-list>, \\
& <s-st-list:is-st-list>)
\end{align*}
\]

One element of a statement-list is interpreted by the instruction \text{int-labeled-st}(t). The instruction updates the statement-prefix part of \text{CS}, raises the check condition and initializes the interpretation of the proper statement.

Definition :

\[
\begin{align*}
\text{int-labeled-st}(t) = \\
is-simple-st(t) & \rightarrow \text{int-st}(s-p-st(t)); \\
& \text{call-check-cond}(\text{check-list}(s-label-list(t), \\
& s-check \cdot s-bps(CS))); \\
& \text{sps-update}(s-cond-part(t)) \\
T & \rightarrow \text{int-st}(t); \\
& \text{call-check-cond}(\text{check-list}(s-label-list(t), \\
& s-check \cdot s-bps(CS))); \\
& \text{sps-update}(s-cond-part(t))
\end{align*}
\]
5.3.11 int-st(t)

The instruction makes the case distinction for the various statements.

Definition:

\[
\begin{align*}
\text{is-assign-st(t)} & \rightarrow \text{int-assign-st(t)} \\
\text{is-allocate-st(t)} & \rightarrow \text{int-allocate-st(t)} \\
\text{is-block(t)} & \rightarrow \text{int-block(t)} \\
\text{is-call-st(t)} & \rightarrow \text{int-task-call(t)} \\
\text{is-call-st-(t)} & \rightarrow \text{int-call-st(t)} \\
\text{is-close-st(t)} & \rightarrow \text{int-close-st(t)} \\
\text{is-delete-st(t)} & \rightarrow \text{int-delete-st(t)} \\
\text{is-exit-st(t)} & \rightarrow \text{int-exit-st(t)} \\
\text{is-free-st(t)} & \rightarrow \text{int-free-st(t)} \\
\text{is-get-st(t)} & \rightarrow \text{int-get-st(t)} \\
\text{is-goto-st(t)} & \rightarrow \text{int-goto-st(t)} \\
\text{is-group(t)} & \rightarrow \text{int-st-list(s-st-list(t))} \\
\text{is-if-st(t)} & \rightarrow \text{int-if-st(t)} \\
\text{is-locate-st(t)} & \rightarrow \text{int-locate-st(t)} \\
\text{is-on-st(t)} & \rightarrow \text{int-on-st(t)} \\
\text{is-open-st(t)} & \rightarrow \text{int-open-st(t)} \\
\text{is-put-st(t)} & \rightarrow \text{int-put-st(t)} \\
\text{is-read-st(t)} & \rightarrow \text{int-read-st(t)} \\
\text{is-return-st(t)} & \rightarrow \text{int-return-st(t)} \\
\text{is-revert-st(t)} & \rightarrow \text{int-revert-st(t)} \\
\text{is-rewrite-st(t)} & \rightarrow \text{int-rewrite-st(t)} \\
\text{is-signal-st(t)} & \rightarrow \text{int-signal-st(t)} \\
\text{is-stop-st(t)} & \rightarrow \text{int-stop-st(t)} \\
\text{is-unlock-st(t)} & \rightarrow \text{int-unlock-st(t)} \\
\text{is-wait-st(t)} & \rightarrow \text{int-wait-st(t)} \\
\text{is-write-st(t)} & \rightarrow \text{int-write-st(t)} \\
\text{is-null-st(t)} & \rightarrow \text{null}
\end{align*}
\]
5.3.2 Statement-lists

A general statement-list may be a statement-list, a while-statement-list or a controlled statement-list. It is interpreted by the instruction \texttt{int-gen-st-list(t)}.

Definition:

\[
\text{int-gen-st-list}(t) = \\
\begin{align*}
\text{is-st-list}(t) & \rightarrow \text{int-st-list}(t) \\
\text{is-st-list-while}(t) & \rightarrow \text{int-while-st-list}(t) \\
\text{is-st-list-contr}(t) & \rightarrow \text{int-ctr-st-list}(t)
\end{align*}
\]

5.3.21 \textit{int-st-list}(t)

Definition:

\[
\text{int-st-list}(t) = \begin{cases} 
\text{null} & \text{if } t = \langle \rangle \\
\text{int-st-list}(\text{tail}(t)); & \text{else if } T \\
\text{int-labeled-st}(\text{head}(t)) & \end{cases}
\]
5.3.3 Iterated statement lists

Statement lists contained in a block may be given an iteration specification. The two cases to be distinguished are the while-statement-lists and the controlled-statement-lists.

5.3.31 Statement lists with a while specification are interpreted by the instruction int-while-st-list(t).

The function truth(op) converts a scalar-operand to a truth value, see 4.3

**Definition:**

\[
\text{int-while-st-list}(t) = \text{iterate-while}(s-st-list(t), v, s-while-expr(t)); \\
v : \text{int-sc-expr}(s-while-expr(t))
\]

**Definition:**

\[
\text{iterate-while}(t, v, t_w) = \text{truth}(v) \rightarrow \text{iterate-while}(t, v, t_w); \\
v : \text{int-sc-expr}(t_w); \\
\text{int-st-list}(t)
\]

5.3.32 Controlled-statement-lists are interpreted by the instruction int-ctr-st-list(t). The text of the control-variable is retained as argument for the call of the check condition (see 5.3.322).
Definition:

\[ \text{int-ctr-st-list}(t) = \]
\[ \text{iterate-spec-list}(s-st-list(t), s-spec-list(t), \]
\[ \text{gen}, s-contr-var(t)); \]
\[ \text{gen}; \text{pass-gen-part}(r); \]
\[ r: \text{int-leftpart}(s-contr-var(t), \text{SC-EI}); \]

Definition:

\[ \text{pass-gen-part}(r) = \]
\[ \text{RETURNS} : s-gen(r) \]

5.3.321 \[ \text{iterate-spec-list}(t, \text{spec-list}, \text{gen}, t_c) \]

Abbreviation:

\[ \text{spec} = \text{head}(\text{spec-list}) \]

Definition:

\[ \text{iterate-spec-list}(t, \text{spec-list}, \text{gen}, t_c) = \]
\[ \text{spec-list} = \langle \rangle \rightarrow \text{null} \]
\[ T \rightarrow \text{iterate-spec-list}(t, \text{tail}(\text{spec-list}), \text{gen}, t_c); \]
\[ \text{iterate-do}(t, \text{gen}, v_2, v_3, v_4, \text{s-while-expr}(\text{spec}), t_c); \]
\[ v_2 : \text{int-sc-expr}(\text{s-by-expr}(\text{spec})) \]
\[ v_3 : \text{int-spec-expr}(\text{s-to-expr}(\text{spec})) \]
\[ v_4 : \text{int-sc-expr}(\text{s-while-expr}(\text{spec})) \]
\[ \text{assign}(\text{gen}, v_1); \]
\[ v_1 : \text{int-sc-expr}(\text{s-init-expr}(\text{spec})) \]

Definition:

\[ \text{int-spec-expr}(\text{expr}) = \]
\[ \text{is-empty}(\text{expr}) \rightarrow \text{RETURNS} : \text{EMPTY} \]
\[ T \rightarrow \text{int-sc-expr}(\text{expr}) \]
5.3.322  \texttt{iterate-do}(t,\textit{gen},v_2,v_3,v_4,\textit{expr},t_c)

**Definition:**

\[
\texttt{iterate-do}(t,\textit{gen},v_2,v_3,v_4,\textit{expr},t_c) = \\
\neg \text{truth}(v_4) \rightarrow \text{null} \\
\text{is-empty}(v_3) \lor \\
\neg((\text{truth}\circ \text{infix-op}(v_2,o,\text{GE}) \land \\
\text{truth}\circ \text{infix-op}(\text{value}(\textit{gen},S),v_3,\text{GT})) \lor \\
(\text{truth}\circ \text{infix-op}(v_2,0,\text{LT}) \land \\
\text{truth}\circ \text{infix-op}(\text{value}(\textit{gen},S),v_3,\text{LT}))) ightarrow \\
\texttt{iterate-do}(t,\textit{gen},v_2,v_3,v_4,\textit{expr},t_c) \\
\{ v_4 : \texttt{int-sc-expr}(\textit{expr}) ; \}
\texttt{assign}(\textit{gen},\text{infix-op}(v_2,\text{value}(\textit{gen},S),\text{ADD})) ; \\
\texttt{int-st-list}(t) ; \\
\texttt{call-check-cond}(\texttt{check-list}(\texttt{do-check-ref}(t_c) , \\
\texttt{s-bps}(CS)))
\]

\[
T \rightarrow \text{null}
\]

**Note:** The translation from concrete to abstract text is such that the while-expression and the by-expression always exist.
5.4 Block Activation and Exit

The following chapters describe the call of procedures and the activation of blocks as required by function references, subroutine calls and begin blocks. Normal exits from a block, i.e. except goto statements leading out of the blocks, are also described. The definitions include the prologue activity and epilogue activity.

Chapter 5.4.1 is a summary of functions and instructions common to the procedure call, the interpretation of blocks and the return statement.

5.4.1 Basic Functions and Instructions

5.4.11 Interpretation of declarations

It is assumed that each identifier (for variables level-one identifiers only) declared in a certain block has got its own single declaration structured according to the definition of "is-decl" in the abstract syntax, and modified as described in the pre-pass (chapter 5.2).

The interpretation of declarations except parameter declarations is performed by the instruction int-decl (id, den, attr, scope) where "id" is the identifier part, "den" is the denotation part, "attr" is the attribute part, and "scope" is the scope part of a given declaration. The instruction updates the attribute directory with the attributes declared plus the current environment (the environment is necessary in those cases where the attributes must be evaluated at a point where the current environment is no longer valid). The instruction also updates the denotation directory according to the type of the declaration.
The denotation according to the type is:

for label constants the associated statement list and the current environment stack (The environment stack identifies uniquely the current block activation; see also chapter 5.5),

for automatic variables the newly created aggregate-name which at the same time is allocated and initialized,

for static and controlled variables the aggregate-name created for them at the time of the prepass,

for defined and based variables empty,

for external procedures the body plus the current environment where the body is found in the denotation directory under the unique name found in the declaration under "s-den",

for internal procedures the body plus the current environment,

for files if external the aggregate name found in the declaration, if internal a newly created aggregate name.

**Abstract Syntax**

$$
is\text{-}decl = (<s\text{-}id:\text{is-name}>,
<s\text{-}scope:\text{(is-int | is-ext | is-param)}>,
<s\text{-}attr:\text{is-attr}>,
<s\text{-}den: ...>)$$
The possible denotation parts of the declarations (except parameters) to be considered are shown in the following table, where scope is the scope part, attr is the attribute part and den is the denotation part of a given declaration.

<table>
<thead>
<tr>
<th>scope</th>
<th>attr</th>
<th>den</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>is-label-const</td>
<td>is-st-list</td>
</tr>
<tr>
<td>is-ext</td>
<td>is-static.s-stg-cl</td>
<td>is-b</td>
</tr>
<tr>
<td>is-int</td>
<td>is-ctr.s-stg-cl</td>
<td>is-b</td>
</tr>
<tr>
<td>is-ext</td>
<td>is-auto.s-stg-cl</td>
<td>is-empty</td>
</tr>
<tr>
<td></td>
<td>is-defined</td>
<td>is-empty</td>
</tr>
<tr>
<td></td>
<td>is-based</td>
<td>is-empty</td>
</tr>
<tr>
<td></td>
<td>is-ext</td>
<td>is-n</td>
</tr>
<tr>
<td></td>
<td>is-int</td>
<td>is-body</td>
</tr>
<tr>
<td></td>
<td>is-ext</td>
<td>is-b</td>
</tr>
<tr>
<td></td>
<td>is-int</td>
<td>is-empty</td>
</tr>
</tbody>
</table>

Definition:

\[
\text{int-decl}(id, \text{den}, \text{attr}, \text{scope}) = \\
\text{is-label-const}(\text{attr}) \quad \rightarrow \\
\text{update-dn-at}(id, \mu_1(\langle \text{s-st-list}; \text{den} \rangle, \langle \text{s-evs}; \text{m-evs}(E,D) \rangle, \text{attr}), \\
\text{is-static}(\text{s-stg-cl}(\text{attr}) \lor \text{is-ctr}.s-stg-cl(\text{attr}) \quad \rightarrow \\
\text{update-da-at}(id, \text{den}, \text{attr}) \\
\]

\[
\text{is-auto}(\text{s-stg-cl}(\text{attr})) \quad \rightarrow \\
\text{initialize}(\text{s-init}(\text{attr})); \\
\text{allocate}(a, (id(E))(DN), \text{eda}, \text{s-dens}(\text{attr})); \\
\text{eda : eval-da (s-da(\text{attr}),E)}, \\
\text{null-allocate}(id(E)(DN)); \\
\text{update-dn-at}(id,b,\text{attr}); \\
\text{b : un-b}
\]

Continued
is-defined(attr) →
update-dn-at(id, EMPTY, at);
at : eval-da(attr, E),

is-based(attr) →
update-dn-at(id, EMPTY, attr),

is-entry(attr) & is-ext(scope) →
update-dn-at(id, den(DN), at);
at : eval-da(attr, E)

is-entry(attr) & is-int(scope) →
update-dn-at(id, μ_o(<s-body:den>, <s-e:E>), at);
at : eval-da(attr, E)

is-file(attr) & is-ext(scope) →
update-dn-at(id, den, attr)

is-file(attr) & is-int(scope) →
update-dn-at(id, b, attr);
b : un-b

Note: The instructions allocate (a, b, eda, dens) and null-allocate(b) are defined in chapter 4.9.

5.4.111 update-dn-at(id, den, attr)

Definition:

update-dn-at(id, den, attr) =

s-dn : μ(DN, <id(E) : den>)
s-at : μ(AT, <id(E) : μ_o(<s-attr:attr>, <s-e:E>))
Note: All attributes entered into the attribute directory are associated with the environment current at the time of the interpretation of the corresponding declaration, though the environment is only needed in some cases.

5.4.112 initialize(is)

Definition:

initialize(is) =
  null;
  ( { initial-assign(gen, vl);
    { gen : int-variable(first(x), E),
      vl : make-vl(second(x), E) } |
      x € is & ¬ is-call-st(second(x)) } u
    { int-call-st(second(x)) | x € is &
      is-call-st(second(x)) })

Note: This initial instruction is a variant of the similar instruction in chapter 4.9. All constituents of this definition are defined there.

5.4.12 stack(ξ)

The function results in a new dump where the constituents of the state which are local to the current block activation including the dump are "on top".

Definition:

stack(ξ) = μ₀ ( <s-e:s-e(ξ)>, <s-c:s-c(ξ)>,
             <s-cs:s-cs(ξ)>, <s-ri:s-ri(ξ)>,
             <s-id:s-id(ξ)>, <s-d:s-d(ξ)>)
5.4.13 ID(decl-part)

The function yields the set of all identifiers declared in the declaration part.

Definition:

\[
\text{ID(decl-part)} = \{ \text{id} \mid \text{is-name}(\text{id}) \land \text{exists}(\text{id}(\text{decl-part})) \}
\]

5.4.14 exit

The instruction performs a "normal" block exit, i.e. it frees all automatic and dummy variables and retrieves the dynamically encompassing block level.

Definition:

\[
\text{exit} = \begin{align*}
\text{pop-up; } \\
\text{free-auto}
\end{align*}
\]

5.4.141 free-auto

The instruction frees all automatic and dummy variables of the current block activation by inspecting the attributes of all declared identifiers of the block activation ID.

Definition:

\[
\text{free-auto} = \null; \{ \text{free-1}\,(\text{genId}) \mid \text{id} \in \text{ID} \land (\text{is-auto}(\text{stg-clId}) \lor \text{is-dummy}(\text{stg-clId})) \}
\]
The instruction retrieves the components of the state current to the dynamically encompassing block activation and makes them the current ones, i.e. "control is returned" to the dynamically encompassing block activation.

Definition:

\[ \text{pop-up} = \]
\[ s-e : s-e \ (D) \]
\[ s-c : s-c \ (D) \]
\[ s-d : s-d \ (D) \]
\[ s-cs : s-cs \ (D) \]
\[ s-id : s-id \ (D) \]
\[ s-ri : s-ri \ (D) \]

The instruction takes the set of all identifiers declared in the block activation, associates each of them with a unique name and updates the environment accordingly.
Definition:

\[
\text{update-env} = \\
\text{null;} \{ \text{update-id}(\text{id}, n); n : \text{un-n} | \text{id} \in \text{ID} \}
\]

where

\[
\text{update-id}(\text{id}, n) = \\
\text{se} : \mu (E, \langle \text{id} : n \rangle)
\]
5.4.2 The procedure call

The instruction int-call-st(t) initiates the procedure call given a call statement 't' without the task option. Call statements that attach a new task are treated separately in chapter 5.6. The arguments are evaluated first by using the instruction int-arg(arg, attr) where arg is an argument given by the call statement and attr are the attributes given by the entry attribute of the called procedure (see chapter 4.5). The evaluation of arguments produces a list called "arg-list" of pairs whose first component is the evaluated argument and whose second component are the associated attributes. After the evaluation of the arguments control is left with the instruction int-call(...) which establishes the new block activation and initiates the execution of the procedure.

Abstract syntax:

is-call-st = (<s-st:is-call>,
              <s-id:is-name>,
              <s-arg-list:is-expr-list>,
              <s-opt:(is-pa-opt | is-Ω)>)

The entry attribute is of the following structure:

is-entry = (<s-param-list>: (is-param-descr-list | is-empty)>,
            <s-ret-type: ...>)

Abbreviations:

attr_t = s-attr((s-id(t)E)AT)

dn_t = ((s-id(t)E)DN)
**Definition:**

\[
\text{int-call-st}(t) = \\
\neg \text{is-empty}(s\text{-param-list}(\text{attr}_t)) \\
\text{int-call}(\text{attr}_t, \text{dn}_t, \text{arg-list}) \\
\{ \text{elem}(i)\cdot \text{arg-list} : \text{int-arg}(\text{elem}(i, s\text{-arg-list}(t)), \text{elem}(i, s\text{-param-list}(\text{attr}_t))) \mid 1 \leq i \leq \text{length}(s\text{-arg-list}(t)) \}
\]

\[
\text{T} = \\
\text{int-call}(\text{attr}_t, \text{dn}_t, \text{arg-list}) \\
\{ \text{elem}(i)\cdot \text{arg-list} : \text{int-arg}(\text{elem}(i, s\text{-arg-list}(t)), \text{EMPTY}) \mid 1 \leq i \leq \text{length}(s\text{-arg-list}(t)) \}
\]

**5.4.21 Block activation**

The instruction \text{int-call}(\text{attr}, \text{den}, \text{arg-list}) establishes a new block activation. The arguments are: \text{attr} is the evaluated entry attribute of the called procedure; \text{den} is the denotation of the called procedure and \text{arg-list} is the list of evaluated arguments of the corresponding call statement. The instruction is used for the interpretation of the call statement as well as for the evaluation of functions. To establish a new block activation means to redefine the dump by putting the environment, the set of declared identifiers, the return information, the control status and the control of the current state on top and to redefine all of the previously mentioned components. The PL/I machine then continues to interpret the newly established control, i.e. to perform the following actions in the given order.

a) the environment is updated (see chapter 5.4.1)

b) a preliminary passing of arguments takes place (by \text{pre-int-params}). This action makes it possible to use arguments which in the following interpretation of declarations do not require further evaluation.
c) All declarations are interpreted and the passing of arguments is completed.

d) Check on equality of the return type of the corresponding entry attribute and of the return type associated with the procedure body is accomplished.

e) Eventually the statement list of the body of the procedure is executed.

f) "Control is returned" to the previous block activation.

Abstract Syntax

The denotation (second argument) has the following structure:

\[
\text{is-den} = (\langle \text{s-e:is-env} \rangle, \langle \text{s-body:is-body} \rangle)
\]

\[
\text{is-body} = (\langle \text{s-param-list:is-name-list} \rangle,
\langle \text{s-decl-part:is-decl-part} \rangle,
\langle \text{s-cond-part:is-cond-part} \rangle,
\langle \text{s-ret-type:...} \rangle,
\langle \text{s-st-list:is-st-list} \rangle)
\]

Abbreviation:

\[\text{den}_b = \text{s-body}(\text{den})\]
Definition:

\[
\text{int-call(attr,den,arg-list)} =
\]
\[
\begin{align*}
\text{s-e} &: s-e(den) \\
\text{s-d} &: \text{stack}(\xi) \\
\text{s-id} &: \text{ID}(s-\text{decl-part}(den_b)) \\
\text{s-ri} &: \mu_o(<s-\text{ret-type}:s-\text{ret-type}(attr)>,<s-\text{ret-point}:\forall>) \\
\text{s-cs} &: \mu_o(<s-\text{cds}:s-\text{cds}(CS)>, \\
& \hspace{1cm}<s-\text{bps}:s-\text{cond-part}(den_b)>, \\
& \hspace{1cm}<s-\text{sps}:\Omega>) \\
\text{s-c} &: \text{exit} \\
\text{int-st-list}(s-st-list(den_b)); \\
\text{check-ri}(rt); \\
rt &: \text{eval-da}(s-\text{ret-type}(den_b)); \\
\text{int-\text{proc-decl-part}}(s-\text{decl-part}(den_b), \\
& s-\text{param-list}(den_b), \\
& \text{arg-list}); \\
\text{pre-\text{int-params}}(s-\text{param-list}(den_b), \\
& \text{arg-list}, \\
& s-\text{decl-part}(den_b)); \\
\text{update-env}
\end{align*}
\]

Note: stack, ID, update-env, exit see chapter 5.4.1.
Since \text{eval-da} has been defined implicitly in an appropriate way it can also be applied to evaluate the expressions in return types.

5.4.211 \text{pre-\text{int-params}}(param-list, arg-list, decl-part)

The arguments of the instruction are the parameter list, the list of evaluated arguments and the declaration part of the body. The instruction tests whether parameter list and argument list are of equal length. For each parameter the denotation directory is updated with the denotation of the argument (selector function s-den applied to the
argument) and the attribute directory is updated with the unevaluated attributes of the parameter. Therefore any succeeding interpretation can refer successfully to a parameter if the associated attributes do not need to be evaluated.

Definition:

\[
\text{pre-int-params}(\text{param-list}, \text{arg-list}, \text{decl-part}) = \\
\text{length(}\text{param-list}\text{)} = \text{length(}\text{arg-list}\text{)} \rightarrow \text{null;}
\]

\[
\{ \text{update-dn-at(} \text{elem(}i\text{,} \text{param-list)}\text{),} \\
\text{s-de}n\text{e}m(e\text{lem(}i\text{,} \text{arg-list)}\text{),} \\
\text{s-attr(} \text{elem(}i\text{,} \text{param-list)}\text{,} \\
\text{(decl-part))} \mid \\
\text{is\_is\_length(} \text{param-list}\text{)} \} \]

5.4.212 int-proc-decl-part(decl-part, param-list, arg-list)

The arguments of the instruction are the declaration part of a called procedure, the parameter list of the called procedure and the list of evaluated arguments passed. The instruction interprets all declarations according to their type as described in chapter 5.4.11 and completes the passing of arguments.

Abstract Syntax

\[
is\text{-decl-part} = (\{ \langle id:\text{is-decl} \rangle \mid \text{is\_name(id)} \} )
\]

Abbreviation:

\[
decl\_id = \text{id(decl-part)}
\]
Definition:

\[
\text{int-proc-decl-part}(\text{decl-part}, \text{param-list}, \text{arg-list}) = \begin{cases} \text{null} \\
\{ \text{int-decl}(\text{id}, \text{s-den(decl_id)}, \text{s-attr(decl_id)}, \\
\text{s-scope(decl_id)}) | \\
\exists \text{id(decl-part)} \& \\
-\text{is-param}\cdot\text{s-scope(decl_id)} \} \cup \\
\{ \text{int-param}(\text{elem(i, param-list)}, \text{s-attr\cdot elem(i, arg-list)}) | \\
i \leq \text{is\cdot length(param-list)} \} \}
\end{cases}
\]

Note: The order in which declarations and parameters are interpreted is completely open. This definition therefore leaves more cases open than does the PL/I-SRL.

5.4.2121 \text{int-param}(\text{id}, \text{attr})

The instruction completes the parameter passing. The arguments of the instruction are the identifier of the parameter and the attributes of the associated argument. At the point where this instruction is interpreted the denotation directory contains already the denotation of the argument and the attribute directory contains the unevaluated attributes of the parameter.

Abbreviation:

\[
\begin{align*}
\text{n}_{id} &= \text{id(E)} \\
\text{p}_{id} &= \text{s-attr\cdot n}_{id}(\text{AT})
\end{align*}
\]
Definition:

\[
\text{int-param}(\text{id, attr}) = \\
\text{is-file}(\text{pa}_{\text{id}}) \& \text{is-file}(\text{attr}) \\
\text{s-at : } \mu(\text{AT}, <\text{s-at} \cdot \text{n}_{\text{id}} : \text{combine-file-attr-2}(\text{attr}, \text{pa}_{\text{id}})>) \\
\text{is-entry}(\text{pa}_{\text{id}}) \& \text{is-entry}(\text{attr}) \\
\text{check-entry}(\text{n}_{\text{id}}, \text{attr}, \text{entry}); \\
\text{entry : eval-da}(\text{pa}_{\text{id}}, E) \\
\text{is-ctr}(\text{s-stg-cl}(\text{pa}_{\text{id}})) \& \text{is-ctr}(\text{s-stg-cl}(\text{attr})) \& \\
\text{ctr-match}(\text{attr}, \text{pa}_{\text{id}}) \\
\text{null} \\
\text{is-empty}(\text{s-stg-cl}(\text{pa}_{\text{id}})) \& \\
(\text{is-empty} \vee \text{is-dummy})(\text{s-st-cl}(\text{attr})) \\
\text{check-da}(\text{n}_{\text{id}}, \text{attr}, \text{dap}); \\
\text{dap : eval-da}(\text{s-da}(\text{pa}_{\text{id}}), E)
\]

Note: The definition of the function combine-file-attr-2 is not specified in the present document, for \text{ctr-match} see chapter 4.5.

5.4.21211 \text{check-entry}(n, \text{attr}, \text{pa})

The arguments are: \text{n} is the unique name of a parameter, \text{attr} are the attributes of the associated argument and \text{pa} are the attributes of the parameter.

The instruction checks the matching of entry attributes and inserts attributes into \text{AT}.

Definition:

\[
\text{check-entry}(n, \text{attr}, \text{pa}) = \\
\text{entry-match}(\text{attr}, \text{pa}) \\
\text{s-at : } \mu(\text{AT}, <\text{s-at} \cdot \text{n} : \text{pa}>)
\]
Note: for entry-match see chapter 4.5.

5.4.21212  check-da(n,attr,dap)

The instruction checks the matching of data-attributes and inserts attributes into AT.

Definition:

\[
\text{check-da}(n,\text{attr},\text{dap}) = \text{match-2}(s-\text{da}(\text{attr}),\text{dap})
\]

\[\text{AT} : \mu(\text{AT}) , \langle s-\text{attr} : n \mu_0 (\langle s-\text{stg-cl} : s-\text{stg-cl}(\text{attr}) \rangle, \langle s-\text{dens} : s-\text{dens} \cdot s-\text{attr} \cdot n(\text{AT}) \rangle, \langle s-\text{da} : \text{dap} \rangle) \rangle\]

Note: The density is taken over from the parameter, but does not have any significance. For match-2(da,dap) see chapter 4.5.

5.4.213  check-ri(rt)

The argument of the instruction is a return type, namely the return type of the entry attribute of the called procedure. The instruction tests on the equality of the argument with the evaluated return type of the procedure.

Definition:

\[
\text{check-ri}(rt) = (rt = s-\text{ret-type}(\text{RT})) \rightarrow \text{null}
\]
5.4.3 Interpretation of Blocks

The interpretation of a block is performed by the instruction `int-block(t)` where t is a block. The instruction is similar to `int-call` except for

a) there are no parameters to be dealt with,

b) the statement-list of a block may be iterated,

c) there is no value returned, therefore there is no return information and no check on return types,

d) the environment to be updated is that of the dynamically encompassing block-activation.

Abstract Syntax:

\[
\text{is-block} = (\langle s\text{-decl-part} : is\text{-decl-part}\rangle, \\
\quad \langle s\text{-cond-part} : is\text{-cond-part}\rangle, \\
\quad \langle s\text{-st-list} : (is\text{-st-list-while} | \\
\quad \quad \text{is-st-list-contr} | \text{is-st-list}) \rangle \\
\quad \langle s\text{-label-list} : is\text{-name-list} \rangle)
\]

Definition:

\[
\text{int-block(t)} = \\
\text{s-d : stack( )} \\
\text{s-id : ID(s-decl-part(t))} \\
\text{s-ri : EMPTY} \\
\text{s/cs : } \mu_{\alpha} (\langle s\text{-cds}; s\text{-cds(CS)} \rangle, \\
\quad \langle s\text{-bps}; s\text{-cond-part(t)} \rangle, \\
\quad \langle s\text{-sps}; \emptyset \rangle) \\
\text{s-c : exit;} \\
\quad \text{int-gen-st-list(s-st-list(t));} \\
\quad \text{int-block-decl-part(s-decl-part(t));} \\
\quad \text{update-env}
\]
Note: DO-groups with iteration are treated as if they were begin blocks with an iterated statement-list instead of a simple one.

This leads to the following consequences:

(1) Identifiers (especially labels) that are declared within a DO-group (with iteration) are local to the DO-group and therefore unknown outside the DO-group.

(2) The cases where goto-statements can lead into a DO-group are the same as for begin-blocks. Especially goto-statements leading into a DO-group are only defined if the DO-group is active (i.e. has been initialized).

(3) Identifiers that are known outside the DO-group can be redeclared inside the DO-group. (This is a deviation from the definition in the SRL.)

5.4.31 \text{int-block-decl-part}(\text{decl-part})

Definition:

\begin{verbatim}
\text{int-block-decl-part}(\text{decl-part}) =
null;
\{ \text{int-decl}(s-id(\text{decl}), s-den(\text{decl}), s-attr(\text{decl}), s-scope(\text{decl})) | 
(\exists id)(\text{is-name}(id) \& \text{exists}(id(\text{decl-part})) \& 
(\text{decl}=id(\text{decl-part}))) \}
\end{verbatim}

Note: The order of interpretation of declarations is completely unspecified. This definition leaves therefore more cases open than the PL/I-SRL does.
5.4.4 Return Statement

The return statement is interpreted by the instruction \texttt{int-ret-st(t)} where \(t\) is a return statement.

The first option where the return statement has no associated expression is interpreted just by a normal exit instruction. For the second option the expression must be evaluated, the value converted to the return type that has been kept in \texttt{RI} and passed to the return point, also kept in \texttt{RI}.

Abstract syntax:

\[
is\text{-return-st} = (\langle s\text{-st:is-return} \rangle, \langle s\text{-expr:(is-expr | is-?)} \rangle)
\]

Definition:

\[
\text{int-ret-st}(t) = \begin{cases} 
\text{exists}(s\text{-expr}(t)) & \rightarrow \text{exit} \\
T & \rightarrow \text{exit;} \\
\text{return}(r); & \\
r : \text{int-sc-expr}(s\text{-expr}(t)) & 
\end{cases}
\]

5.4.41 \texttt{return(r)}

This instruction converts the value to be returned to the return type and inserts the converted value into the place denoted by the return point (kept in \texttt{RI}) of the control of the dynamically encompassing activation (i.e. the control found on top of the dump).
Abbreviation:

\[ r_c = \mu \left( \langle \text{s-op:convert} \langle \text{s-ret-type} \langle \text{RI} \rangle, r \rangle, \langle \text{s-ei:SC-EI} \rangle \right) \]

Definition:

\[
\text{return}(r) = \mu \left( D, \langle \text{s-ret-point} \langle \text{RI} \rangle \rangle \langle \text{s-c:r} \rangle \right)
\]
The Goto-Statement

The interpretation of a goto-statement is performed by the instruction \texttt{int-goto-st(t)} where \( t \) is the text of the goto-statement. The goto-statement refers to its designation either by a reference or by a label constant. The first step in the interpretation of a goto-statement is to determine the label-denotation which is either the value of the reference or the denotation of the label-constant. The rest of the interpretation of a goto-statement is done by the instruction \texttt{goto-1(ld)} where \( ld \) is the label-denotation. This instruction first finds the correct block-activation (i.e. the block-activation where the label denotation originates) and then establishes the appropriate control. The instruction is undefined if the block-activation to which the label points is no longer existent.

Abstract Syntax:

\[
\text{is-goto-statement} = (\langle \text{st}: \text{is-goto},
\langle \text{lab-ref}: \text{is-ref} \rangle \rangle)
\]

Definition:

\[
\text{int-goto-st}(t) = \text{is-simple-name}(s \rightarrow \text{lab-ref}(t)) \\
\text{is-label-const(attr}_{t}) \rightarrow \text{goto-1}(dn_{t}),
\]

\[
T \rightarrow \text{goto-1}(ld);
\]

\[
ld : \text{expand-value} \; \text{gen, SC-EI};
\]

\[
\text{gen} : \text{int-variable}(s \rightarrow \text{lab-ref}(t), E)
\]

where:

\[
\text{attr}_{t} = s \rightarrow \text{attr}((s \rightarrow \text{id}(s \rightarrow \text{lab-ref}(t))E) \rightarrow \text{AT})
\]

\[
dn_{t} = ((s \rightarrow \text{id}(s \rightarrow \text{lab-ref}(t))E) \rightarrow \text{DN})
\]
5.5.1 goto-1(ld)

The argument of the instruction is a label-denotation. The task performed by this instruction is to return to the block-activation to which ld points and to close all activations that have been activated since. To close a block-activation means essentially to free all automatic and dummy variables and to pop up the dump. This is done by the instructions free-auto and block-exit-1 respectively. The machine then goes on to interpret the statement-list which is part of the label-denotation.

The environment stack (evs) which is part of the label-denotation identifies the block-activation to which return has to be made. The environment stack associated with a state of the PL/I-machine (by the function m-evs(E,D)) remains constant while an activation is active and current. It uniquely identifies an activation if at least one identifier has been declared in the corresponding block. All block-activations that can be pointed to by a label-denotation have at least one declaration in the corresponding block (either a label variable or a label). Therefore, if the block-activation to which the ld points is still active, the instruction finds the correct block-activation by comparing the environment stack of the label-denotation with the state of the PL/I machine as it pops up the dump. If the activation is not active any more, the result of the instruction is undefined.

Abstract Syntax:

\[
\text{is-label-denotation} = (<s-st-list : is-st-list>, <s-evs : is-evs>)
\]

\[
is-evs = (<s-e : is-e>, <s-evs : (is-evs | is-empty) >)
\]
Definition:

\[ \text{goto-1}(ld) = \]
\[ (m\text{-evs}\,(E,D) = s\text{-evs}(ld)) \rightarrow s\text{-c} : \text{int-st-list}(s\text{-st-list}(ld)) \]
\[ T \rightarrow \text{block-exit-1}(ld) : \text{free-auto} \]

range of arguments: is-label-denotation(ld)

where:

\[ m\text{-evs-1}(d) = ((d=\text{EMPTY}) \rightarrow \text{EMPTY}, \]
\[ T \rightarrow \lambda_0(<s\text{-e}s\text{-e}(d)>,<s\text{-evs}:m\text{-evs-1}\cdot s\text{-d}(d)>) \]

range of arguments: is-d(d)

\[ m\text{-evs}(e,d) = \lambda_0(<s\text{-e}:e>,<s\text{-evs}:m\text{-evs-1}(d)>) \]

range of arguments: is-e(e), is-d(d)

Note: The statement-list which is part of the label-denotation is the list of statements which starts with the statement with which the label was originally associated and ends with the end of the block to which the label is internal. (see App.I,5.15)

5.5.11 block-exit-1(ld)

This instruction pops up the dump.

Definition:

\[ \text{block-exit-1}(ld) = \]
\[ s\text{-e} : s\text{-e}(D) \]
\[ s\text{-d} : s\text{-d}(D) \]
\[ s\text{-ri} : s\text{-ri}(D) \]
\[ s\text{-id} : s\text{-id}(D) \]
\[ s\text{-cs} : s\text{-cs}(D) \]
\[ s\text{-c} : \text{goto-1}(ld) \]
5.6 The Task Call

The generation of a new task is performed by the instruction
\[ \text{int-task-call}(t) \]
where the argument \( t \) syntactically is a call-statement with a task option. This instruction establishes a new state \( \xi' \), which has the same abstract syntax as \( \xi \). The interrelations of two parallel tasks represented by \( \xi \) and \( \xi' \) remain undefined by the interpretation of one state \( \xi \), to some extent they are undefined in the language. If enough is known about the effects of parallel execution of tasks, a task supercontrol can be designed which defines the effects of asynchronous interrupts and sharing of storage and files (see 7).

Abbreviations:

\[
\begin{align*}
\text{opt}_t &= \text{s-opt}(t) \\
\text{attr}_t &= \text{s-attr}((\text{s-id}(t)\in\text{AT})) \\
\text{dn}_t &= ((\text{s-id}(t)\in\text{AT})) \\
\text{den}_t &= \text{s-body}([\text{den}])
\end{align*}
\]

Definition:

\[
\text{int-task-call}(t) = \\
\text{is-empty}(\text{s-param-list}([\text{attr}_t])) \\
\text{int-pa-call}([\text{attr}_t, \text{dn}_t, \text{arg-list}, \text{opt}_t]); \\
\{\begin{align*}
\text{elem}(i) \cdot \text{arg-list}: \\
\text{int-arg}(\text{elem}(i, \text{s-arg-list}(t)), \\
\text{elem}(i, \text{s-param-list}([\text{attr}_t]))) & \mid \\
\text{is Length}(\text{s-arg-list}(t))
\end{align*}
\}
\]

T \rightarrow \text{int-pa-call}([\text{attr}_t, \text{dn}_t, \text{arg-list}, \text{opt}_t]); \\
\{\begin{align*}
\text{elem}(i) \cdot \text{arg-list}: & \text{int-arg}(\text{elem}(i, \text{s-arg-list}(t)), \\
\text{EMPTY}) & \mid \\
\text{is Length}(\text{s-arg-list}(t))
\end{align*}
\}
The instruction \texttt{int-na-call(attr,den,arg-list,opt)} makes the necessary assignments to the associated task and event variables and through \texttt{int-ev-opt} and \texttt{ev-c-assign} checks whether the association of the event-variable to a task is legal.

The function \texttt{mk-\$'} defines a new state \$', copying those parts of \$ which can be used by the new task and establishing the other parts of the state with the proper contents. In the new control the instruction \texttt{int-call(attr,den,arg-list)} is to be interpreted as shown in 5.4.2.

The effects of the task supercontrol and the information which has to be passed to this control is not given in this chapter. In the single state interpretation the instruction \texttt{mk-na(argument)} results in the \texttt{null} instruction.

\textbf{Definition:}

\begin{align*}
\texttt{int-na-call(attr,den,arg-list,opt)} &= \\
&= \texttt{mk-na} (\texttt{mk-\$'} (\$ , attr, den, arg-list, opt));
\end{align*}

\begin{align*}
\texttt{int-ev-opt(opt,ev-gen)}, \\
\texttt{ev-gen} :&= \texttt{ev-ta-gen} (\texttt{s-event-var(opt)} ) \\
\texttt{int-ta-opt(opt,ta-gen)} \\
\texttt{ta-gen} &= \texttt{ev-ta-gen} (\texttt{s-task-var(opt)} )
\end{align*}

\textbf{Definition:}

\begin{align*}
\texttt{ev-ta-gen(t)} &= \\
\texttt{is-ref(t)} &\rightarrow \texttt{int-variable(t,\$)} \\
\texttt{T} &\rightarrow \texttt{RETURNS : EMPTY}
\end{align*}
Definition:

\[ \text{int-ev-opt}(\text{opt}, \text{ev-g}) = \]
\[ \neg \text{is-empty}(\text{ev-g}) \rightarrow \text{ev-c-assign}((\text{loc}(s-\text{addr}(\text{ev-g})), \text{head}\cdot s-il(\text{ev-g}))) \]
\[ T \rightarrow \text{null} \]

Definition:

\[ \text{int-ta-opt}(\text{opt}, \text{ta-g}) = \]
\[ \neg \text{is-empty}(\text{ta-g}) \rightarrow \text{ta-c-assign}(\text{opt}, (\text{loc}(s-\text{addr}(\text{ta-g})), \text{head}\cdot s-il(\text{ta-g}))) \]
\[ T \rightarrow \text{null} \]

Definition:

\[ \text{mk-na}(\text{argument}) = \]
\[ \text{null} \]

Definition:

\[ \text{ev-c-assign}(\text{loc}(a,i_o)) = \]
\[ s-s:(\neg \text{exists}(s-\text{assoc-status}\cdot s-value: \text{loc}(a,i_o)(S)) \rightarrow \]
\[ \mu(S, \langle loc(s,i_o): \mu(\text{loc}(a,i_o)(S), \]
\[ <s-compl-status\cdot s-value:\text{represent}(\text{LOGICAL,DI}>, \]
\[ <s-normal-status\cdot s-value:\text{represent}(\text{LOGICAL,DI}>, \]
\[ <s-assoc-status\cdot s-value: \text{ASSOCIATED}>) \}) \]

Definition:

\[ \text{ta-c-assign}(\text{opt}, \text{loc}(a,i_o)) = \]
\[ \text{priority-assign}(s-\text{priority}(\text{opt}), \text{loc}(a,i_o)) \]
Definition:

\[ \text{priority-assign} (\text{expr}, \text{loc}(a, i_o)) = \text{undefined} \]

Definition:

\[ \text{mk-} \xi' (\xi, \text{attr}, \text{den}, \text{arg-list}) = \]
\[ \mu_0 (\langle \text{s-a:} \xi, \text{s-b:} \text{B}, \text{s-ab:} \text{A}_\text{B}, \text{s-n:} \text{N}, \text{s-f:} \text{I}, \text{s-s:} \text{S}, \text{s-ag:} \text{\mu}_0 (\{ \langle \text{b: head-b(NG)} \mid \text{exists(b(NG))} \} ) \rangle, \text{s-at:} \text{AT}, \text{s-dn:} \text{DN}, \text{s-es:} \text{ES}, \text{s-fd:} \text{\mu}_0 (\{ \langle \text{x: s-ofa:x} \mid \right. \]
\[ \left. \text{(exists(\text{\kappa s-ofa(FD)}) \& x = \kappa s-ofa(FD)) \mid (exists(\text{\kappa s-nfa(FD)}) \& x = \kappa s-nfa(FD))} \}\rangle, \text{s-fu:} \text{FU}, \text{s-e:} \text{EMPTY}, \text{s-d:} \text{EMPTY}, \text{s-cs:} \text{CS}, \text{s-ri:} \text{EMPTY}, \text{s-id:} \text{EMPTY}, \text{s-c:} \text{\text{int-call}(\text{attr, den, arg-list})} \rangle) \]
5.7 Assignment-Statement

The assignment-statement (without by-name-option) is interpreted by the instruction \texttt{int-ass-st(t)}. The abstract syntax of the assignment-statement is

\[ \text{is-assign-st} = (\text{s-st:is-assign}, \text{s-lp:is-ref-list}, \text{s-rp:is-expr}) \]

Evaluation of left-part and right-part and scalar assignment are made for each scalar part of the operands. The scalar assignment is defined in chapter 4.7, the interpretation of pseudo-variables in the left-part and the assignment to pseudo-variables is defined in chapter 4.8. The expansion information used in interpretation of aggregate expressions is described in chapter 4.2.4, the interpretation of expressions in chapter 4.4.

After execution of the assignment the instruction \texttt{call-check-cond} performs the condition calls for all checked identifiers in the left-part.

\textbf{Definition:}

\[ \text{int-ass-st}(t) = \text{call-check-cond}(\text{check-list(assign-check-list(s-lp(t),E,AT), s-lps(CS))}, \text{iterate-ass-st}(t, \text{EMPTY})) \]
5.7.1 \textit{iterate-ass-st}(t,ei)

The instruction iterates the interpretation of the assignment statement \(t\).

\textbf{Definition}:

\[
\text{iterate-ass-st}(t,ei) =
\begin{cases}
\text{is-exhaust}(ei) & \rightarrow \text{null} \\
T & \rightarrow \text{iterate-ass-st}(t,x);
\end{cases}
\]

\[
x : \text{int-sc-ass-st}(vl,s-rp(t));
\begin{cases}
\text{elem}(i) &: vl \\
\text{int-leftpart}(\text{elem}(i,s-lp(t)),ei) & | 1 \leq \text{length}(s-lp(t))
\end{cases}
\]

5.7.2 \textit{int-sc-ass-st}(vl,expr)

The instruction evaluates the expression \(expr\), makes the assignment of the result to the list of generations in \(vl\) and returns the updated expansion information (to the instruction \textit{iterate-ass-st}).

\textbf{Definition}:

\[
\text{int-sc-ass-st}(vl,expr) =
\begin{cases}
\sum_{i=1}^{\text{length}(vl)} s-ei \circ \text{elem}(i,vl) & \rightarrow \\
\text{pass}(\text{update}(s-ei\circ\text{head}(vl))); \\
\text{assign-list}(vl,r); \\
\text{assign-list}(vl,r);
\end{cases}
\]

\[
r : \text{int-expr}(expr,s-ei\circ\text{head}(vl))
\]
5.7.3 \texttt{assign-list(vl,r)}

The instruction makes the assignment of the expression-result \( r \) to all generations in the list \( vl \).

\textbf{Definition}:

\[
\begin{align*}
\text{assign-list}(vl,r) &= \text{null;} \\
&\{ \text{assign}(\text{s-gen*elem}(i,vl),\text{s-op}(r)) \mid 1 \leq i \leq \text{length}(vl) \}
\end{align*}
\]

5.7.4 \texttt{int-leftpart(t,ei)}

The instruction returns an expanded generation \( (<\text{s-gen}:\text{is-gen}>,<\text{s-ci},\text{is-ei}>) \) from the interpretation of the reference \( t \) with the expansion information \( ei \). It makes the distinction between proper-leftparts (i.e. references to variables) and pseudo-leftparts (i.e. references to pseudo-variables). The reference to variables is defined in chapter 4.2.5 and chapter 4.2.6, the reference to pseudo-variables in 4.8. The instruction \texttt{expand-gen} is defined in 4.2.7.

\textbf{Abbreviation}:

\[
\begin{align*}
\text{n}_t &= \text{is-ptr-qual-ref}(t) \quad \rightarrow \quad (s-id\cdot s-based-ref(t))(\mathcal{E}) \\
\text{is-simple-ref}(t) &= \quad \rightarrow \quad (s-id(t))(\mathcal{E}) \\
\text{attr}_t &= s-attr\cdot n_t(\mathcal{AT})
\end{align*}
\]
**Definition:**

\[
\text{int-leftpart}(t, ei) = \\
\begin{align*}
\text{is-built-in}(\text{attr}_t) & \rightarrow \text{int-ps-leftpart}(t, ei) \\
\text{is-variable}(\text{attr}_t) & \rightarrow \text{int-prop-leftpart}(t, ei)
\end{align*}
\]

**Definition:**

\[
\text{int-prop-leftpart}(t, ei) = \\
\begin{align*}
\text{is-isub-def}(\text{attr}_t) & \rightarrow \text{expand-gen}(r) \\
& \quad r : \text{isub-action}(r_t, rl, ei) \\
& \quad rl : \text{int-name}(t, E) \\
\text{is-variable}(\text{attr}_t) & \rightarrow \text{expand-gen}(\langle\text{gen}, ei\rangle) \\
& \quad \text{gen} : \text{int-variable}(t, E)
\end{align*}
\]
5.8 Allocate Statement

The allocate statement specifies a list of allocations, where each allocation is a controlled-, based-, or area-allocation. The allocate statement is interpreted by the instruction `int-allocate(t)`.

The abstract syntax of the allocate statement is:

\[
\text{is-allocate-st} = \text{is-allocate-list}
\]

\[
\text{is-allocate} = (\text{is-contr-allocate} \mid \text{is-based-allocate} \mid \text{is-area-allocate})
\]

\[
\text{is-contr-allocate} = (\langle \text{s-st:is-alloc}, \text{s-id:is-name}, \text{s-al:is-al}, \text{s-is:is-init-set} \rangle)
\]

\[
\text{is-al} = (\text{is-struct-al} \mid \text{is-array-al} \mid \text{is-string-al} \mid \text{is-empty})
\]

\[
\text{is-struct-al} = \text{is-al-tree}
\]

\[
\text{is-array-al} = (\langle \text{s-lbd:(is-expr} \mid \text{is-empty}), \text{s-ubd:(is-expr} \mid \text{is-empty}), \text{s-elem:is-al} \rangle)
\]

\[
\text{is-string-al} = (\langle \text{s-base:(is-bit} \mid \text{is-char}), \text{s-length:(is-expr} \mid \text{is-empty}) \rangle)
\]

\[
\text{is-based-allocate} = (\langle \text{s-st:is-alloc}, \text{s-id:is-name}, \text{s-set-variable:is-ref} \rangle)
\]

\[
\text{is-area-allocate} = (\langle \text{s-st:is-alloc}, \text{s-id:is-name}, \text{s-set-variable:is-ref}, \text{s-area:is-ref} \rangle)
\]

**Abbreviations:**

\[
\begin{align*}
  t_1 & = \text{head}(t) \\
  n_1 & = (\text{s-id}(t_1))(E) \\
  \text{attr}_1 & = s\text{-attr}n_1(\text{AT}) \\
  g_1 & = \text{head}(n_1(\text{BN}))(\text{AG}) \\
  \text{env}_1 & = s\text{-e}n_1(\text{AT})
\end{align*}
\]
The basic instructions for allocation and initialization are defined in 4.9.

**Definition:**

\[
\text{int-allocate}(t) = \begin{cases} 
\text{null} & \text{if } t = < > \\
\text{int-allocate}(\text{tail}(t)) & \text{otherwise}
\end{cases}
\]

\[
\text{is-contr-allocate}(t_1) \rightarrow 
\text{int-allocate}(\text{tail}(t)); \\
\text{contr-initialize}(s\text{-is}(t_1), s\text{-init}(\text{attr}_1), \text{env}_1); \\
\text{allocate}(a, n_1(\text{DN}), da, s\text{-dens}(\text{attr}_1)); \\
a : \text{un-a} \\
da : \text{eval-da}(da_1, env_1) \\
da_1 : \text{make-all-attr}(al, s\text{-da}(\text{attr}_1), g_1) \\
al : \text{eval-da}(s\text{-al}(t_1), E)
\]

\[
is\text{-based-allocate}(t_1) \rightarrow 
is\text{-allocate}(\text{tail}(t)); 
\text{based-allocate}(a, b, ptr\text{-gen}, da, s\text{-dens}(\text{attr}_1)) 
\]

\[
a : \text{un-ab} \\
b : \text{un-b} \\
ptr\text{-gen} : \text{int}\text{-variable}(s\text{-set}\text{-variable}(t_1), E) \\
da : \text{eval-da}(s\text{-da}(\text{attr}_1), env_1)
\]

\[
is\text{-area-allocate}(t_1) \rightarrow 
is\text{-allocate}(\text{tail}(t)); 
\text{area-allocate}(a, b, ptr\text{-gen}, ar\text{-gen}, da, s\text{-dens}(\text{attr}_1)) 
\]

\[
a : \text{un-a} \\
b : \text{un-b} \\
ptr\text{-gen} : \text{int}\text{-variable}(s\text{-set}\text{-variable}(t_1), E) \\
ar\text{-gen} : \text{int}\text{-variable}(s\text{-area}(t_1), E) \\
da : \text{eval-da}(s\text{-da}(\text{attr}_1), env_1)
\]
5.8.1 Initialization

Initialization of a controlled variable is performed by the instruction \texttt{contr-initialize} \( (\text{ins}_t, \text{ins}_d, \text{env}) \), where \( \text{ins}_t \) is the set of initials specified in the allocate statement, \( \text{ins}_d \) is the set of initials specified in the attributes of the variable to be initialized, and \( \text{env} \) is the environment in which the variable has been declared.

The two initial sets are merged, where the elements of \( \text{ins}_t \) have priority over the elements of \( \text{ins}_d \). Replication factors have to be evaluated in \( E \) for elements of \( \text{ins}_t \), and in \( \text{env} \) for elements of \( \text{ins}_d \).

Definition:

\[
\text{contr-initialize}(\text{ins}_t, \text{ins}_d, \text{env}) = \\
\text{null;}
\{ \text{initial-assign} (\text{gen}, v_l); \{ \text{gen} : \text{int-variable}(s\text{-first}(x), E), \} \\
\quad \text{v} \quad \text{vl} : \text{make-}\text{vl}(s\text{-second}(x), y) \} \}
\]
5.8.2 Controlled allocate

The attributes used for the allocation of the controlled variable are merged from those specified in the allocate statement, those given in the declaration of the variable and those taken from the most recent generation of the variable. The merging is performed by the instruction `make-all-attr(al,da,gen).

**Definition:**

\[
\text{make-all-attr}(al, da, gen) = \text{RETURNS : fal}(al, da, s-da(gen))
\]

**Definition:**

\[
fal(al, da, dag) =
\]

\[
is-empty(al) \rightarrow da
\]

\[
is-array-al(al) \& is-array(da) \& is-empty(s-lbd(al)) \& is-empty(s-ubd(al)) \rightarrow
\]

\[
\mu_0(<s-lbd:s-lbd(dag)>, <s-ubd:s-ubd(dag)>, <s-elem:fal(s-elem(al), s-elem(da), s-elem(dag))>)
\]

\[
is-array-al(al) \& is-array(da) \rightarrow
\]

\[
\mu_0(<s-lbd:s-lbd(al)>, <s-ubd:s-ubd(al)>, <s-elem:fal(s-elem(al), s-elem(da), s-elem(dag))>)
\]

\[
is-array(da) \rightarrow
\]

\[
\mu_0(<s-lbd:s-lbd(da)>, <s-ubd:s-ubd(da)>, <s-elem:fal(al, s-elem(da), s-elem(dag))>)
\]

\[
is-struct-al(al) \& is-struct(da) \& order(al) = order(da) \rightarrow
\]

\[
\mu_0(\{ <s-da+succ(i):fal(succ(i,al), s-da+succ(i,da), s-da+succ(i,dag))> | 1 \leq i \leq order(da) \})
\]

\[
is-string(al) \& is-string(da) \& is-empty(s-length(al)) \rightarrow dag
\]

\[
is-string(al) \& is-string(da) \rightarrow al
\]
5.8.3 Based-allocate

The allocation of based variables is performed by the instruction $\text{based-allocate}(a, b, \text{ptr-gen}, da, dens)$. The address-names used for allocation of based variables are taken from the list $\Sigma_b$, in order to distinguish the resulting storage-locations from those not resulting from a based-allocate. The distinction is needed in free-statements using based-variables: the pointer-qualified reference written in a free-statement must not refer to storage locations not allocated by a based-allocate.

**Definition:**

$\text{based-allocate}(a, b, \text{ptr-gen}, da, dens) =$

```
null;
{ allocate(a, b, da, dens); null-allocate(b),
  assign(ptr-gen, pointer(m-gen(da, dens, a))) }
```

**Note:** The entry of the new generation in $AG$ is only for the purpose of freeing storage that has been allocated in one task, at the exit from this task.

5.8.4 Area-allocation

At the allocation of a variable in an area, the new generation is put on top of the list in $s$-contents-loc$(a, i)(S)$, where loc$(a, i)$ selects location of the area in $S$. This list represents the allocation status of the area. The allocation status together with the attributes of the area are sufficient information for the area-condition (the raising of the condition, however, is not defined in the present document).
Definition:

\[\text{area-allocate}(a,b,\text{ptr-gen},\text{area},\text{da},\text{dens}) = \]
\[\text{null; } \{ \text{allocate}(a,b,\text{da},\text{dens}); \text{null-allocate}(b), \]
\[\text{assign}(\text{ptr-gen},\text{pointer}(\text{m-gen}(\text{da},\text{dens},a))), \]
\[\text{in-area}(\text{area},\text{m-gen}(\text{da},\text{dens},a)) \}\]

Abbreviations:

\[a = \text{s-addr(area)}\]
\[i_0 = \text{head-s-il(area)}\]

Definition:

\[\text{in-area}(\text{area},\text{gen}) = \]
\[\text{s-s : } \mu(\ S, \ <\text{loc}(a,i_0):\mu(\ \text{loc}(a,i_0)(S), \]
\[<\text{s-contents}::\text{gen}^*\text{s-contents}^*\text{loc}(a,i_0)(S)>)>\]

Note: Freeing of an area means freeing of all generations it contains; see the free-statement in 5.9.
5.0 Free Statement

The instruction \texttt{int-free-st(t)} where \( t \) is a free statement performs the interpretation of the free statement. The actual freeing of storage for each of the items of the free list is done by the instruction \texttt{free} which distinguishes between two cases. The two case distinctions in the definition of the instruction correspond to the two options of the free statement (see SRL). The first case deals with freeing of generations of controlled variables. The second case deals exclusively with based storage.

Abstract Syntax:

\[
\text{is-free-st} = \langle \text{s-st:is-free}, \\
\quad \langle \text{s-free-list:is-level-one-ref-list} \rangle \rangle
\]

\[
\text{is-level-one-ref} = (\text{is-simple-name} | \\
\quad \text{is-ptr-qual-level-one-ref})
\]

\[
\text{is-ptr-qual-level-one-ref} = \\
\quad (\langle \text{s-ptr-qual:is-ref} \\
\quad \langle \text{s-based-ref:is-simple-name} \rangle \rangle)
\]

\[
\text{is-simple-name} = (\langle \text{s-id:is-name}, \\
\quad \langle \text{s-arg-list:is-emptylist}, \\
\quad \langle \text{s-qual-list:is-emptylist} \rangle \rangle)
\]

Definition:

\[
\text{int-free-st}(t) = \\
\quad \text{null; \\
\quad \{ \text{free(elem(i,s-free-list(t)))} \} \\
\quad \text{isislength(s-free-list(t))} \}
\]
5.9.1 \texttt{free(t)}

The argument is an item of the free list of a free statement.

\textbf{Abbreviations:}

\begin{align*}
\text{attr}_t &= \text{s-attr}((\text{s-id}(t)E)AT) \\
\text{bt} &= ((\text{s-id}(t)E)DN) \\
\text{gen}_t &= \text{head}(\text{bt}(AG)) \\
\text{addr} &= \text{s-addr}(\text{gen}) \\
\text{il} &= \text{s-il}(\text{gen})
\end{align*}

\textbf{Definition:}

\begin{align*}
\text{free}(t) &= \\
&= \text{is-simple-name}(t) \land \text{is-ctor-s-stg-cl}(\text{attr}) \land \\
&\quad \lnot (\text{tail}(\text{bt}(AG)) = <>) \rightarrow \\
&\quad \text{del-gen}(\text{bt}); \\
&\quad \text{free-1}(\text{gen}_t) \\
&\quad \text{is-ptr-qual-level-one-ref}(t) \rightarrow \\
&\quad \text{free-1}(\text{gen}-2); \\
&\quad \text{gen}-2 : \text{check-based-level-one}(\text{gen}-1); \\
&\quad \text{gen}-1 : \text{int-variable}(t,E)
\end{align*}

\textbf{Definition:}

\begin{align*}
\text{del-gen}(b) &= \\
&= \text{s-ag} : \mu (AG, <b: \text{tail}(b(AG))>)
\end{align*}

\begin{align*}
\text{check-based-level-one}(\text{gen}) &= \\
&= \text{is-ab}(\text{addr}_{\text{gen}}) \land \\
&\quad ((\text{elem}(1,\text{il}_{\text{gen}}) = 1) \land \\
&\quad \lnot \exists \text{lo}(\text{addr}_{\text{gen}},\text{length}(\text{il}_{\text{gen}}) + 1)) \rightarrow \\
&\quad \text{RETURNS} : \text{gen}
\end{align*}
Note: In the first case of the `free` instruction the action is only defined if more than one generation is in the aggregate. This corresponds to the rule of the SRL that it is an error to free a variable that has not been allocated. For an attached task (see chapter 5.6) this guarantees also that the task does not free storage allocated by the mother task.

In the second case a check is made to guarantee that the storage freed is indeed based and that only level-one aggregates are freed.

For `free-1(gen)` see chapter 4.10.
5.10 Conditional Statement

The conditional statement is interpreted by the instruction \( \text{int-if-st}(t) \) where \( t \) is a conditional statement.

Abstract syntax:

\[
\text{is-if-st} = (\text{s-cond-part:is-cond-part}, \text{s-decision:is-expr}, \text{s-then-st:is-st}, \text{s-else-st:is-st})
\]

Definition:

\[
\text{int-if-st}(t) = \\
\text{int-if}(v, \text{s-then-st}(t), \text{s-else-st}(t)), \\
v : \text{int-sc-expr}(\text{s-decision}(t))
\]

where:

\[
\text{int-if}(v, s_1, s_2) = \\
\text{truth}(v) \rightarrow \text{int-labeled-st}(s_1) \\
\neg \text{truth}(v) \rightarrow \text{int-labeled-st}(s_2)
\]

Note: The translation from concrete to abstract reduces the two options permitted for a conditional statement to one by inserting a null statement.
5.11 Built-in functions

5.11.1 Denotation of a built-in function identifier

An identifier id satisfying

\[ \text{is-built-in}\_s\_attr(id)(AT) \]

is a built-in function identifier. It has a denotation which is not found via \( \text{DIJ} \), but given by

\[ \text{built-in-fct-den}(id). \]

The function \( \text{built-in-fct-den} \) is defined by a table in section 5.11.3.

The denotation of a built-in function identifier is a built-in function denotation. A built-in function denotation is an object:

\[
\text{is-built-in-fct-den} = \text{is-cond-built-in-fct-den-list} \\
\text{is-cond-built-in-fct-den} = (\text{s-al-cond:is-al-cond}, \\
\quad \text{s-simple-built-in-fct-den:is-simple-built-in-fct-den}) \\
is-al-cond = \ldots
\]

Note: The class of objects satisfying \( \text{is-al-cond} \) is left unspecified. These objects are primitive; they are functions from argument lists to truth values and are used by the instruction \( \text{int-built-in}(t,ei) \) (see section 5.11.2) to select a particular \( \text{simple-built-in-fct-den} \), depending on \( \text{s-arg-list}(t) \).

\[
\text{is-simple-built-in-fct-den} = \\
(\text{s-arg-int-list:is-arg-int-list}, \\
\text{s-res-attr-fct:is-res-attr-fct}, \\
\text{s-res-fct:is-res-fct}) \\
is-res-attr-fct = \ldots \\
is-res-fct = \ldots
\]
Note: These two predicates are left unspecified. The objects satisfying them are primitive, and are functions. A result-attribute-function computes the attributes of the result of the built-in-function reference from the attr-const-list (i.e. the attributes and decimal integer constant arguments) of the evaluated and converted arg-list of the reference. A result-function computes the result of the reference from the evaluated and converted arg-list (see section 5.11.2).

\[
is\text{-arg-int} = (<s\text{-repl}:(is-o-ai \mid is-r-ai \mid is-Q)>),
<s\text{-expd}:(is-e-ai \mid is-Q)>,
<s\text{-intp}:(is-v-ai \mid is-g-ai \mid is-s-ai \mid is-f-ai \mid
is-a-ai \mid is-c-ai)>,
<s\text{-conv-arg-fct}:is\text{-conv-arg-fct}>)
\]

Note: The primitive objects defined by is-o-ai, is-r-ai, ... are interpreted as follows (see section 5.11.2):

- **is-o-ai**: argument optional
- **is-r-ai**: argument zero or more times
- **is-e-ai**: argument inherits expansion
- **is-v-ai**: argument is interpreted as an expression yielding a scalar value
- **is-g-ai**: argument is interpreted as generation
- **is-s-ai**: argument is interpreted as controlled stack
- **is-f-ai**: argument is a file name
- **is-a-ai**: argument is interpreted as an aggregate
- **is-c-ai**: argument is a possibly signed decimal integer constant
is-conv-arg-fct = ...

Note: This predicate is left unspecified. The class of objects satisfying it are primitive, and are functions. A convert-argument-function operates on the attr-const-list of the evaluated arg-list of the built-in function reference. The value produced by the function is again a function which is then used to convert an evaluated argument of the reference (see section 5.11.2).

5.11.2 The instruction int-built-in(t,ei)

This instruction has the following arguments:

t ... built-in-function reference,

ei ... expansion-information (see section 4.2.4)

\[
\text{int-built-in}(t,ei) = \\
\text{int-built-in-1}(adl,rf,vl,ei); \\
\{ \text{elem}(i) \circ vl: \text{int-arg-of-built-in-fct}(\text{elem}(i,adl), \\
\text{elem}(i,al),ei) \mid \\
1 \leq \text{islength}(al) \}
\]

where

\[
al = \text{s-arg-list}(t) \\
cdl = \text{built-in-fct-den} \circ \text{s-id}(t) \\
den = \text{s-simple-built-in-fct-den} \circ \text{elem}(p,cdl) \\
p = (\forall k)((s-al-cond \circ \text{elem}(k,cdl))(al) \& \\
(\forall i)(1 \leq i \leq \text{islength}(al)) \\
ail = \text{s-arg-int-list}(den) \\
rf = \text{s-res-fct}(den) \\
adl = \text{mk-adl}(ail,al)
\]
mk-adl(ail, al) =
(al = < > & (ail = < > v
(is-o-ail v is-r-ail) * s-repl-head(ail)) ——> < >,
is-o-ail * s-repl-head(ail) ——>
mk-adl(δ(ail, s-repl.elem(1)), al),
is-r-ail * s-repl-head(ail) ——>
(δ(head(ail), s-repl) * mk-adl(tail(ail), tail(ail)),
T ——> <head(ail) * mk-adl(tail(ail), tail(ail)))

Note: The number p is the index of the first component of cdl, whose condition, when applied to al, is satisfied. The arg-description-list adl results from the arg-interpretation-list ail by adding or deleting components according to replicators and al, and by deleting the replicators.

int-arg-of-built-in-fct (ad, t, ci) =
(is-c-ail * s-expd & is-v-ail * s-intp)(ad) ——> int-expr(t, ci)
(is-c-ail * s-expd & is-g-ail * s-intp)(ad) ——> int-prop-leftpart(t, ci)
is-v-ail * s-intp(ad) ——> int-sc-expr(t)
is-g-ail * s-intp(ad) ——> int-variable(t, Β)
(is-s-ail * s-intp(ad) & is-ctf * s-stg-cl(attrt)) v
(is-f-ail * s-intp(ad) & is-file(attrt)) ——>
RETURNS: μ_0 (<s-attr: attrt>, <s-den: don_t>)
is-a-ail * s-intp(ad) ——> int-aggr(t)
is-c-ail * s-intp(ad) & p-s-dec-int-const(t) ——> int-sc-expr(t)

where
attr_t = s-attr(s-id(t)(E)(AT), don_t = s-id(t)(E)(DM)
p-s-dec-int-const(t) = ...
(t is a possibly signed decimal integer constant)
int-aggr(t) = extract-attr(opl);
opl : collect-aggr(t, < >)
extract-attr(opl) =

RETURNS: μ₀(<s-da:collect-attr(s-si·s-ei·head(opl),opl)>,
           <s-opl:δ(opl, {s-ei·elem(i) | 1≤i≤length(opl)})>)

For the definition of int-cxnr see section 4.4.1, of int-prop-leftpart 5.7.4, of int-sc-cxnr 4.6.2, of int-variable 4.2.5, of collect-attr 4.5.12, of collect-attr 4.5.29.

Note: The instruction int-arg-of-built-in-fct(ad,t,ei) interprets an argument t according to the interpretation-part and expand-part of the corresponding argument-description. The result has a structure which depends on ad.

int-built-in-1(adl,rf,vl,ei) =

RETURNS: μ₀(<s-op:rf(μ₁(<elem(i):caf₁(acl)(opl) >
                   | 1≤i≤length(opl)>),
           <s-ei:(eil = < > —> ei,
                   T —> comp-ei-1(eil)>))

where

eil = μ₁(<elem(i):s-ei·elem(i,vl) > | exists(s-ei·elem(i,vl)))
opl = δ(vl, { s-ei·elem(i) | exists(s-ei·elem(i,vl)) })
acr = attr-const-list(adl,opl)
caf₁ = s-conv-arg-fct(adl₁)
adl₁ = elem(i,adl)
opl₁ = elem(i,opl)

attr-const-list(adl,opl)
attr-const-list(adl,opl) =
μ₁(<elem(i):attr-const(adl₁,opl₁) > | 1≤i≤length(opl)>)
attr-const(ad,op) = (is-c-ai·s-intp(ad) —> op,
                     (is-s-ai v is-f-ai)·s-intp(ad) —> s-attr(op),
                     T —> s-da(op))

caf₁ = s-conv-arg-fct(adl₁)
The caf_i(acl)(opl_i) yield converted operands. The conversion operation applied to opl_i is caf_i(acl), i.e. it depends on acl. To the list of converted operands, the result-function rf is applied. The ei-components are compared and deleted from vl.

5.11.3 Table of built-in-function denotations

Notation:

The following table gives for each built-in function identifier its denotation built-in-fct-den(id). This denotation is a list of conditional denotations (see section 4.11.1). For each element cd the components cond = s-al-cond(cd), ail = s-arg-int-list(den), and raf = s-res-attr-fct(den) are given (where den = s-simple-built-in-fct-den(cd)); the component rf = s-res-fct(den) is not specified.

If cd_i has more than one element, the elements are written in successive lines. The elements ai of ail are separated by commas, list brackets are omitted. (Hence, an empty ail is represented by a blank.) For each ai, the component caf = s-conv-arg-fct(ai) is given, possibly preceded by "*" and one or more capitals. These letters specify the occupants of the positions s-repl, s-expd, s-intp. For example, E specifies is-e-ai-s-expd(ai), etc. (see definition of is-arg-int, section 5.11.1.) If for a position no letter is specified, Q is implied, except in the case of s-intp, where V is implied.
Those of the components which are functions are represented by giving their values for a variable argument: cond by cond(al), caf by caf(acl) (which is again a function), raf by raf(acl). If cond(al) is unspecified, T is implied; if caf(acl) is unspecified, the identity function I is implied.

**Objects and functions left unspecified:**

The following objects and functions used in the table are left unspecified for the moment:

a) ar, real, compl, ..., string, ... :
   These are objects that result from scalar da's by making certain components EMPTY.

b) hbs( ), hbs( ), ... :
   These are functions on lists of (incomplete) da's yielding the highest characteristics of the elements.

c) c( ), cl( ) :
   These are functions converting da's, or list of da's, to given da's.

d) conv( ), conv-array( ), ..., filt( ), filt-lin-array( ), ... :
   These are conversion functions, converting operands to given attributes, and filtering functions, checking operands against given attributes.

e) pr-abs( ), pr-med( ), ..., l-bit( ), ..., a-round( ) ... :
   These are functions computing certain precisions, lengths, or composed attributes from given static-parts or static-part-lists.

f) sc-pt( ), sc-pt-l( ) :
   These are functions selecting the scalar part from an array-attribute, or from a list of array-attributes.
**Abbreviations:**

\[ l = \text{length}(al), \quad acl_i = \text{elem}(i, acl) \]

The table:

<table>
<thead>
<tr>
<th>id</th>
<th>cond</th>
<th>ail</th>
<th>raf</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS</td>
<td></td>
<td>( E^* \text{conv}(ar) )</td>
<td>( \mu(acl_1, &lt;s\text{-mode:REAL}&gt;, \text{pr-abs}(acl_1)) )</td>
</tr>
<tr>
<td>SIGN</td>
<td></td>
<td>( E^* \text{filt}{\text{real}} \circ (\text{conv}(ar)) )</td>
<td>INT</td>
</tr>
<tr>
<td>MOD</td>
<td></td>
<td>( E^* \text{filt}{\text{real}} \circ )</td>
<td>( \mu(acl_1, &lt;s\text{-prec:pr-mod}(acl)&gt;) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{conv}(\text{hbs_cl}(ar, acl)) ), ( E^* \text{filt}{\text{real}} \circ )</td>
<td>( \text{conv}(\text{hbs_cl}(ar, acl)) )</td>
</tr>
<tr>
<td>COMPL</td>
<td></td>
<td>( E^* \text{conv}(\text{real}), E^* \text{conv}(\text{real}) )</td>
<td>( \mu(\text{hbsmp}(acl), &lt;s\text{-mode:COMPL}&gt;) )</td>
</tr>
<tr>
<td>REAL, IMAG</td>
<td></td>
<td>( E^* \text{conv}(\text{compl}) )</td>
<td>( \mu(acl_1, &lt;s\text{-mode:REAL}&gt;) )</td>
</tr>
<tr>
<td>CONJS</td>
<td></td>
<td>( E^* \text{conv}(\text{compl}) )</td>
<td>acl_1</td>
</tr>
<tr>
<td>FLOOR, CEIL, TRUNC</td>
<td></td>
<td>( E^* \text{filt}{\text{real}} \circ (\text{conv}(ar)) )</td>
<td>( \mu(acl_1, &lt;s\text{-prec:pr-trunc}(acl_1)&gt;) )</td>
</tr>
<tr>
<td>FIXED</td>
<td></td>
<td>( E^* \text{conv}(ar), \text{OC, OC} )</td>
<td>( \mu(acl_1, &lt;s\text{-scale:FIX}, &lt;s\text{-prec:pr-fixed}(acl)&gt;) )</td>
</tr>
<tr>
<td>FLOAT</td>
<td></td>
<td>( E^* \text{conv}(ar), \text{OC} )</td>
<td>( \mu(acl_1, &lt;s\text{-scale:FLT}, &lt;s\text{-prec:pr-float}(acl)&gt;) )</td>
</tr>
<tr>
<td>id</td>
<td>cond</td>
<td>ail</td>
<td>raf</td>
</tr>
<tr>
<td>-------</td>
<td>---------------</td>
<td>------------------------------------------</td>
<td>------------------------------------------</td>
</tr>
<tr>
<td>BINARY</td>
<td>E*conv(ar), OC, OC</td>
<td>(\mu(acl_1, \langle s\text{-base:BIN}, \langle s\text{-prec:} \text{pr-bin}(acl) \rangle \rangle))</td>
<td></td>
</tr>
<tr>
<td>DECIMAL</td>
<td>E*conv(ar), OC, OC</td>
<td>(\mu(acl_1, \langle s\text{-base:DEC}, \langle s\text{-prec:} \text{pr-bin}(acl) \rangle \rangle))</td>
<td></td>
</tr>
<tr>
<td>PRDC</td>
<td>E*conv(ar), C, OC</td>
<td>(\mu(acl_1, \langle s\text{-prec:} \text{pr-prec-1}(acl) \rangle))</td>
<td></td>
</tr>
<tr>
<td>ADD,</td>
<td>E<em>conv(ar), E</em>conv(ar), C, OC</td>
<td>(\mu(hbsm(acl_1, acl_2), \langle s\text{-prec:} \text{pr-prec-2}(acl) \rangle))</td>
<td></td>
</tr>
<tr>
<td>MULTIPLY,</td>
<td></td>
<td>hbsmp(acl)</td>
<td></td>
</tr>
<tr>
<td>DIVIDE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAX,</td>
<td>E<em>filt({real})</em>(conv(ar)), E<em>filt({real})</em>(conv(ar)), RE<em>filt({real})</em>(conv(ar))</td>
<td>hbsmp(acl)</td>
<td></td>
</tr>
<tr>
<td>MIN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOG2,</td>
<td>E<em>filt({real})</em>(conv(flt))</td>
<td>acl_1</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ln</td>
<td>E*conv(flt)</td>
<td>acl_1</td>
<td></td>
</tr>
<tr>
<td>ATAN</td>
<td>E*conv(flt)</td>
<td>acl_1</td>
<td></td>
</tr>
<tr>
<td>1=1</td>
<td>E<em>filt({real})</em> (conv(hbsocl(ar,acl)))</td>
<td>(\mu(acl_1, \langle s\text{-prec:} \text{pr-atan}(acl) \rangle))</td>
<td></td>
</tr>
<tr>
<td>1=2</td>
<td>E<em>filt({real})</em> (conv(hbsocl(ar,acl)))</td>
<td>(\mu(acl_1, \langle s\text{-prec:} \text{pr-atan}(acl) \rangle))</td>
<td></td>
</tr>
<tr>
<td>ATAND</td>
<td>E<em>filt({real})</em>(conv(flt))</td>
<td>acl_1</td>
<td></td>
</tr>
<tr>
<td>1=1</td>
<td>E<em>filt({real})</em> (conv(hbsocl(ar,acl)))</td>
<td>(\mu(acl_1, \langle s\text{-prec:} \text{pr-atan}(acl) \rangle))</td>
<td></td>
</tr>
<tr>
<td>1=2</td>
<td>E<em>filt({real})</em> (conv(hbsocl(ar,acl)))</td>
<td>(\mu(acl_1, \langle s\text{-prec:} \text{pr-atan}(acl) \rangle))</td>
<td></td>
</tr>
<tr>
<td>id</td>
<td>cond</td>
<td>ail</td>
<td>raf</td>
</tr>
<tr>
<td>------</td>
<td>-----------------------</td>
<td>------------------------------------------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>BIT</td>
<td>E*conv(string), OC</td>
<td></td>
<td>$\mu(acl_1, &lt;s\text{-base:BIT}&gt;)$</td>
</tr>
<tr>
<td>CHAR</td>
<td>E*conv(string), OC</td>
<td></td>
<td>$\mu(acl_1, &lt;s\text{-base:CHAR}&gt;)$</td>
</tr>
<tr>
<td>SUBSTR</td>
<td>E<em>conv(string), conv(INT), O</em>conv(INT)</td>
<td></td>
<td>$\mu(acl_1, &lt;s\text{-var:TRUE}&gt;)$</td>
</tr>
<tr>
<td>INDEX</td>
<td>E<em>conv(hbocl(string,acl)), E</em>conv(hbocl(string,acl))</td>
<td></td>
<td>INT</td>
</tr>
<tr>
<td>LENGTH</td>
<td>E*conv(string)</td>
<td></td>
<td>INT</td>
</tr>
<tr>
<td>HIGH</td>
<td>C</td>
<td></td>
<td>$\mu_0 (&lt;s\text{-base:CHAR}&gt;)$</td>
</tr>
<tr>
<td>LOW</td>
<td></td>
<td></td>
<td>$\mu_0 (&lt;s\text{-length:acl_1}&gt;)$</td>
</tr>
<tr>
<td>REPEAT</td>
<td>E*conv(string), C</td>
<td></td>
<td>$\mu(acl_1, &lt;s\text{-length:1-repeat(acl) &gt;})$</td>
</tr>
<tr>
<td>UNSPEC</td>
<td>E*filt({ar,string,ptr})</td>
<td></td>
<td>$\mu_0 (&lt;s\text{-base:BIT},&lt;s\text{-var:v-unspec(acl_1)&gt;}, &lt;s\text{-length:1-unspec(acl_1) &gt;})$</td>
</tr>
<tr>
<td>BOOL</td>
<td>E<em>conv(bit), E</em>conv(bit), E*conv($\mu_0 (&lt;s\text{-base:bit}, &lt;s\text{-length:4}&gt;)$)</td>
<td></td>
<td>$h_{bvl}(acl_1,acl_2)$</td>
</tr>
<tr>
<td>ROUND</td>
<td>E*filt({ar,string}), C</td>
<td></td>
<td>$a\text{-round}(acl)$</td>
</tr>
<tr>
<td>EVENT</td>
<td>E*filt({event})</td>
<td></td>
<td>$\mu_0 (&lt;s\text{-base:BIT},&lt;s\text{-length:1}&gt;)$</td>
</tr>
<tr>
<td>id</td>
<td>cond</td>
<td>ail</td>
<td>raf</td>
</tr>
<tr>
<td>------</td>
<td>----------</td>
<td>--------------------------</td>
<td>------------------------------------------</td>
</tr>
<tr>
<td>PRIORITY</td>
<td>E*filt{task}</td>
<td>?</td>
<td>µ(sc→pt(acl_1),<a href="">s-prec:?</a>)</td>
</tr>
<tr>
<td>PROD</td>
<td>A*conv-array(flt)•filt-array</td>
<td>sc-pt(acl_1)</td>
<td>hbsmp sc-pt-l(acl)</td>
</tr>
<tr>
<td>ALL, ANY</td>
<td>A*conv-array(bit)•filt-array</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LBOUND, HBOUND, DIM</td>
<td>?, conv(INT)</td>
<td>INT</td>
<td></td>
</tr>
<tr>
<td>ONFILE, DATA-FIELD</td>
<td></td>
<td>µo (<a href="">s-base:CHAR</a>, <a href="">s-var:TRUE</a>, <a href="">s-length:1-onfile</a>)</td>
<td></td>
</tr>
<tr>
<td>ONCHAR</td>
<td></td>
<td>µo (<a href="">s-base:CHAR</a>, <a href="">s-length:1</a>)</td>
<td></td>
</tr>
<tr>
<td>OPCODE</td>
<td></td>
<td>INT</td>
<td></td>
</tr>
<tr>
<td>ADDR</td>
<td>G</td>
<td>PTR</td>
<td></td>
</tr>
<tr>
<td>NULL</td>
<td></td>
<td>PTR</td>
<td></td>
</tr>
<tr>
<td>DATE</td>
<td></td>
<td>µo (<a href="">s-base:CHAR</a>, <a href="">s-length:6</a>)</td>
<td></td>
</tr>
<tr>
<td>TIME</td>
<td></td>
<td>µo (<a href="">s-base:CHAR</a>, <a href="">s-length:9</a>)</td>
<td></td>
</tr>
<tr>
<td>id</td>
<td>cond</td>
<td>ail</td>
<td>raf</td>
</tr>
<tr>
<td>----------</td>
<td>-------</td>
<td>------------------------------------------</td>
<td>------------------------------------------</td>
</tr>
<tr>
<td>ALLOCATION</td>
<td>S</td>
<td></td>
<td>$\mu_0(\langle s\text{-base}!:!\text{BIT}\rangle, \langle s\text{-length}!:!1\rangle)$</td>
</tr>
<tr>
<td>LINENO, COUNT</td>
<td>F</td>
<td></td>
<td>INT</td>
</tr>
<tr>
<td>STRING</td>
<td>G*filt-packed-structure</td>
<td></td>
<td>a-string(acl$_1$)</td>
</tr>
</tbody>
</table>
5.12 Conditions

Under the name "conditions" various situations occurring in the interpretation of a program are collected. Computational conditions occur in expression evaluation. As the arithmetic for expression evaluation is partly not defined by the language but rather is implementation defined, the occurrence of condition situations depends on the specific implementation. The area condition, too, is not defined in the language since the occurrence of the condition situation depends on the data representation in a specific implementation. The check condition allows to check for various references to names. The I/O conditions describe defined I/O error situations.

For all conditions the action to be performed when the condition is raised can either be specified in a program by condition prefixes and the execution of ON and REVERT statements, or are system defined.

The individual parts of CS contain the informations about the current prefix and condition denotation status.

5.2.1 Condition prefixes

Condition prefixes determine whether the occurrence of a condition situation leads to the raising of a condition.

The condition parts selected by s-ond-part of procedures, blocks and single executable statements contain the complete collection of condition prefixes, which in case of procedure and block are valid during expression evaluation at activation time, for executable statements during interpretation of the statement, unless a new statement is entered during interpretation of an if-statement.

The condition parts have been established by the pretranslator by the merging of all prefixes from the static environment. In the case of a
check-condition for a structure name in the concrete text, the condition part contains separate check-conditions for all substructures.

In the state of the PL/I machine condition prefixes for a block are kept in CS during block interpretation and are accessible by \texttt{s-bps(CS)}. The statement prefixes are kept during interpretation of a statement and are accessible by \texttt{s-sps(CS)}.

Definition:

\[
\text{sps-update}(t) = \\
s-CS : \mu(CS, <s-sps:t>)
\]

5.12.2 Condition denotation

For each condition the language specifies the action to be performed if the condition is raised. This action is either system defined or specified by the execution of \texttt{ON} and \texttt{REVERT} statements. The current condition denotations are kept in CS and are accessible by \texttt{s-cds(CS)}. The part accessible by \texttt{s-dens\{cond-sel\}.s-cds(CS)} contains a proper denotation of the on-unit of the establishing \texttt{ON} statement. The on-unit is called like a normal procedure when the condition is raised. The condition denotations are inherited dynamically, i.e. whenever a procedure is called, the condition denotation part is copied from the condition denotation of the calling procedure or block.
5.12.3 Initial state $CS_0$

In establishing $S$ for a program the existence of an initial condition state $CS_0$ is assumed. $CS_0$ contains the prefixes which are system enabled and the denotations which exist before execution of any ON-statement in a program. The occurring system-action remains undefined.

$$CS_0 = \mu_0 (<s-bps: \mu_0 (<s-conv: \text{TRUE}, <s-fofl: \text{TRUE},>
<s-ofl: \text{TRUE}, <s-ufl: \text{TRUE},>
<s-zdiv: \text{TRUE}, <s-check: \{ \} >),
<s-st-list \cdot s-body \cdot s-den \cdot s(x) \cdot s-cds:
\text{SYSTEM-ACTION} > | \n\forall x \in \{ \text{CONV, FOFL, OFL, UFL, ZDIV,}
\text{SUBRG, AREA-SIZE, FINISH, ERROR} \} )$$

5.12.4 ON-statement

The interpretation of an ON-statement changes the denotation of the condition referenced in the statement. For a check-condition the denotation for a list of check references can be changed. The simple reference set, like in a check condition prefix, contains all substructures of a structure name originally appearing in the identifier list of the concrete ON-statement. If SNAP is specified, the information is kept but the SNAP action is not defined.
\[
\text{int-on-st}(t) = \\
\quad \text{is-prog-named-cond}(s\text{-on-cond}(t)) \rightarrow \\
\quad \text{on-update}(s\text{-cond-name}\cdot s\text{-on-cond}(t), t) \\
\text{is-named-io-cond}(s\text{-on-cond}(t)) \rightarrow \\
\quad \text{on-update}(\langle s\text{-file}\cdot s\text{-on-cond}(t), s\text{-cond}\cdot s\text{-on-cond}(t) \rangle, t) \\
\text{is-check}(s\text{-on-cond}(t)) \rightarrow \\
\quad \text{null}; \{ \text{on-update}(\delta(x, s\text{-arg-list}), t) \mid x \in s\text{-check}\cdot s\text{-on-cond}(t) \} \\
\text{T} \rightarrow \\
\quad \text{on-update}(s\text{-on-cond}(t), t) \\
\text{on-update}(x, t) = \\
\quad s\text{-cs} : \mu(CS, \langle s(x)\cdot s\text{-cds}\cdot s\text{-on-den}(t, E) \rangle) \\
\text{on-den}(t, \text{env}) = \\
\quad \mu_0(\langle s\text{-snap}\cdot s\text{-snap}(t) \rangle, \\
\quad \langle s\text{-den} : \mu_0(\langle s-e, \text{env} \rangle, \\
\quad \quad \langle s\text{-body} : \mu_0(\langle s\text{-param-list} : \langle \rangle, \\
\quad \quad \quad \langle s\text{-cond-part} : \text{EMPTY} \rangle, \\
\quad \quad \quad \langle s\text{-ret-type} : \text{EMPTY} \rangle, \\
\quad \quad \quad \langle s\text{-label-list} : \langle \rangle, \\
\quad \quad \quad \langle s\text{-st-list} : s\text{-on-unit}(t) \rangle \rangle \rangle) \rangle)
\]

5.12.5 REVERT statement

The interpretation of a REVERT statement nullifies the effect of a preceding ON statement interpreted in the same block for the same condition. The effect is described by copying the respective condition denotation from CS of the dynamically preceding block to the condition denotation part of the current CS.
\[ \text{int-revert-st}(t) = \]
\[ \quad \text{is-prog-named-\text{cond}}\left(s-\text{on-\text{cond}}(t)\right) \rightarrow \]
\[ \quad \text{on-revert} \left(s-\text{cond-name} * s-\text{on-\text{cond}}(t), t\right) \]
\[ \quad \text{is-named-io-\text{cond}}\left(s-\text{on-\text{cond}}(t)\right) \rightarrow \]
\[ \quad \text{on-revert} \left(<s-\text{file} * s-\text{on-\text{cond}}(t), s-cond * s-\text{on-\text{cond}}(t)>, t\right) \]
\[ \quad \text{is-check}\left(s-\text{on-\text{cond}}(t)\right) \rightarrow \]
\[ \quad \text{null; \{ on-revert} \left(\delta(x, s-\text{arg-list}), t\right) \mid x \in s-\text{check} * s-\text{on-\text{cond}}(t) \} \]
\[ \rightarrow \quad \text{on-revert} \left(s-\text{on-\text{cond}}(t), t\right) \]
\[ \text{on-revert}(x, t) = \]
\[ \quad s-cs : \mu(\text{CS}, <s(x) * s-\text{c-ds}: s(x) * s-\text{c-ds} * s-cs(D)>) \]

5.12.6 SIGNAL statement

The SIGNAL statement raises the condition specified in the statement.

\[ \text{int-signal-st}(t) = \]
\[ \quad \text{is-prog-named-\text{cond}}\left(s-\text{on-\text{cond}}(t)\right) \rightarrow \]
\[ \quad \text{c-call} \left(s-\text{cond-name} * s-\text{on-\text{cond}}(t)\right) \]
\[ \quad \text{is-named-io-\text{cond}}\left(s-\text{on-\text{cond}}(t)\right) \rightarrow \]
\[ \quad \text{c-call} \left(<s-\text{file} * s-\text{on-\text{cond}}(t), s-cond * s-\text{on-\text{cond}}(t)>, \right) \]
\[ \quad \text{is-check}\left(s-\text{on-\text{cond}}(t)\right) \rightarrow \]
\[ \quad \text{null; \{ c-call} \left(\delta(x, s-\text{arg-list}) \mid x \in s-\text{check} * s-\text{on-\text{cond}}(t) \} \}
\[ \rightarrow \quad \text{c-call} \left(s-\text{on-\text{cond}}(t)\right) \]
5.12.7 Interpretation of condition denotation

A condition call establishes the snap action if specified in the condition denotation and establishes a standard call int-call or the condition system action.

\[
c\text{-call}(t) = \begin{cases} 
\text{int-snap}(t), \\
\text{int-c}(t) 
\end{cases}
\]

\[
\text{int-snap}(t) = \begin{cases} 
\text{exists}(s\text{-snap}s(t)\ast s\text{-cds}(CS)) \rightarrow \text{snap-action}(t) \\
T \rightarrow \text{null}
\end{cases}
\]

\[
\text{int-c}(t) = \begin{cases} 
\text{is-system}(s\text{-st-list} \ast s\text{-body} \ast s\text{-den} \ast s(t) \ast s\text{-cds}(CS)) \rightarrow \text{c-system-action}(t) \\
T \rightarrow \text{int-call}(\text{attr}, \text{den}, \text{arg-list}); \\
\text{attr} : \text{pass}(\mu (\langle s\text{-param-list}:<>\rangle, \\
\langle s\text{-ret-type}:\text{EMPTY}\rangle)) \\
\text{d} : \text{pass}(s\text{-den} \ast s(t) \ast s\text{-cds}(CS)) \\
\text{arg-list} : \text{pass}(<>)
\end{cases}
\]

\[
\text{snap-action}(t) = \text{undefined} \\
\text{c-system-action}(t) = \text{undefined}
\]

5.12.8 Raising check condition and I/O conditions

In some cases check conditions are raised for a list of check references, because the checks for a statement are collected and raised before or after interpretation of the statement.
\[
\text{call-check-cond}(t) = \\
\quad \begin{align*}
\text{is-emptylist}(t) & \rightarrow \text{null} \\
T & \rightarrow \text{call-check-cond}(\text{tail}(t)); \\
& \quad \text{c-call}(\text{head}(t))
\end{align*}
\]

For collecting the checked references from a list of references the function \text{check-list} is used. The list of references has to contain expanded structure references, i.e. only references to substructures which do not contain substructures.

\[
\text{check-list}(l\text{-list}, p\text{-set}) = \\
\quad \begin{align*}
\text{is-emptylist}(l\text{-list}) & \rightarrow < > \\
(\exists x)((x \in p\text{-set}) & \quad (x = \delta(\text{head}(l\text{-list}), s\text{-subscr-list})) & \rightarrow \\
\quad (\delta(\text{head}(l\text{-list}), s\text{-subscr-list}))) & \rightarrow \\
\text{check-list}(\text{tail}(l\text{-list}), p\text{-set}) \\
T & \rightarrow \text{check-list}(\text{tail}(l\text{-list}), p\text{-set})
\end{align*}
\]

The function \text{assign-check-list} generates a list of references from the left-part of an assignment statement by expanding the references to pseudo variables and structures.

\textbf{Definition:}

\[
\text{assign-check-list}(t, \text{env}, \text{at}) = \\
\quad \begin{align*}
\neg \text{is-emptylist}(t) & \rightarrow \text{assign-check}(\text{head}(t), \text{env}, \text{at}) \\
\text{assign-check-list}(\text{tail}(t), \text{env}, \text{at}) \\
T & \rightarrow < >
\end{align*}
\]

The function \text{assign-check} generates from a reference in the left-part the list of references obtainable from structures and complex-pseudo-variables, and a single reference for a scalar.
Abbreviation:

\[ \text{attr}_t = \text{s-attr}(\text{s-id}(t)(\text{env}))(\text{at}) \]

Definition:

\[
\text{assign-check}(t, \text{env}, \text{at}) = \\
\begin{array}{l}
\text{is-variable}(% \text{attr}_t) \rightarrow \text{check-exp}(t, \text{env}, \text{at}) \\
\text{is-built-in}(\text{attr}_t) \rightarrow \\
\quad \text{is-compl}(\text{s-id}(t) \rightarrow \text{check-exp}(\text{elem}(1) \cdot \text{s-arg-list}(t), \\
\quad \text{env}, \text{at})) \\
\quad \text{check-exp}(\text{elem}(2) \cdot \text{s-arg-list}(t), \\
\quad \text{env}, \text{at}) \\
\end{array}
\]

\[ T \rightarrow \text{check-exp}(\text{elem}(1) \cdot \text{s-arg-list}(t), \text{env}, \text{at}) \]

The function check-exp expands a reference \( t \) to a list of references.

Definition:

\[
\text{check-exp}(t, \text{env}, \text{at}) = \\
\begin{array}{l}
\mu_o(\{<\text{elem}(i):x> \\
\quad | (1\leq i \leq \text{scalar-extent}(\text{new-da}(\text{s-da} \cdot \text{attr}_t, \text{ref-list}(
\text{s-da} \cdot \text{attr}_t, \text{s-arg-list}(t), \\
\text{s-qual-list}(t)))))) \land \\
\quad (x = \mu_o(<\text{id}:\text{id}(t)>, \\
\quad <\text{s-arg-list}:< >>, \\
\quad <\text{s-qual-list}:% \text{s-qual-list}(t)>> \\
\quad \text{h}(% \text{new-da}(\text{s-da} \cdot \text{attr}_t, \\
\quad \text{ref-list}(\text{s-da} \cdot \text{attr}_t, \text{s-arg-list}(t), \\
\quad \text{s-qual-list}(t)))) )))\} \\
\end{array}
\]

The auxiliary function \( h(da) \) generates for a structure the list of identifier lists which make up complete qualification lists.
\[ h(da) = \mu_o\left( \left\{ \langle \text{elem}(i) : f(da, rl) \rangle \mid (1 \leq \text{is-scalar-extent}(da)) \& \ (rl=(i \& (\text{scalar-index}(da, rl)=i))) \right\} \right) \]

\[ f(da, rl) = \text{is-scalar}(g(da)) \quad \text{< >} \]
\[ T \quad k(\text{succ}(\text{head}(rl)))(g(da)), \text{tail}(rl) \]

\[ k(da, rl) = \text{is-scalar}(da) \quad \text{< >} \]
\[ T \quad <s-id(da)> \cap k(\text{succ}(\text{head}(rl))(da), \text{tail}(rl)) \]

\[ \gamma(da) = \text{is-array}(da) \quad \rightarrow \quad g(\text{s-elem}(da)) \]
\[ \text{is-struct}(da) \quad \mu_o(\left\{ \langle \text{succ}(i) : g(\text{succ}(i, da)) \rangle \mid 1 \leq \text{is-order}(da) \right\} ) \]
\[ \text{is-scalar}(da) \quad \rightarrow \quad da \]

For I/O conditions the following cases are possible:

\[
\begin{align*}
\text{raise-endfile-cond}(t) &= \text{c-call}(\langle s\text{-file-opt}(t), \text{endfile} \rangle) \\
\text{raise-name-cond}(t) &= \text{c-call}(\langle s\text{-file-opt}(t), \text{data-name} \rangle) \\
\text{raise-endpage-cond}(t) &= \text{c-call}(\langle s\text{-file-opt}(t), \text{endpage} \rangle) \\
\text{raise-transmit-cond}(t) &= \text{c-call}(\langle s\text{-file-opt}(t), \text{transmit} \rangle) \\
\text{raise-undf-cond}(t) &= \text{c-call}(\langle s\text{-file-opt}(t), \text{undf} \rangle) \\
\text{raise-key-cond}(t) &= \text{c-call}(\langle s\text{-file-opt}(t), \text{key} \rangle) \\
\text{raise-record-cond}(t) &= \text{c-call}(\langle s\text{-file-opt}(t), \text{record-size} \rangle)
\end{align*}
\]
5.42.9 Undefined raising of conditions

For several conditions the situation which leads to the raising of these conditions are not defined by the language but rather is implementation defined. Nevertheless, these conditions have a proper denotation and prefix state and it is defined what action is to be performed if a condition is raised.

The AREA condition could only be defined, if the detailed storage mapping were defined in the language.

Computational conditions can be raised at points during expression evaluation which are at a level of detail beyond the definition of the language. These interrupts are similar to asynchronous interrupts in a parallel task situation. They can only be completely defined if a supercontrol is introduced which is described in terms of uninterruptable elementary operations. Establishing these elementary operations would result in serious restrictions for implementations since all implementations would have to guarantee the same uninterruptable elementary operations.
6. INPUT AND OUTPUT STATEMENTS

6.1 Abbreviations

The following is a summary of all abbreviations which apply throughout chapter 6 without being referenced at the place where the abbreviations are used.

The abbreviations mainly concern the parts of the state for input and output introduced in 3.1.7.

Assuming is-name(fid),

\[ n(fid) = \text{fid} \]
\[ \text{den}(fid) = \text{n}(fid) \]
\[ \text{attr}(fid) = \text{s-attr}\cdot(n(fid)) \]
\[ \text{new-fa}(fid) = \text{den}(fid)\cdot\text{s-nfa}(FD) \]
\[ \text{old-fa}(fid) = \text{den}(fid)\cdot\text{s-ofa}(FD) \]
\[ \text{fa}(fid) = (\exists \text{new-fa}(fid) \implies \text{new-fa}(fid)) \]
\[ \text{exists}\left(\text{old-fa}(fid)\right) \implies \text{old-fa}(fid) \]
\[ \text{fu-elem}(fid) = \text{fa}(fid) \]
\[ \text{ddname}(fid) = \text{s-ddname}\cdot\text{fu-elem}(fid) = \text{ddname-sel}(fid) \]
\[ \text{csa}(fid) = \text{s-csa}\cdot\text{fu-elem}(fid) = \text{csa-sel}(fid) \]
\[ \text{count}(fid) = \text{s-count}\cdot\text{fu-elem}(fid) = \text{count-sel}(fid) \]
\[ \text{locate-exec}(fid) = \text{s-locate-exec}\cdot\text{fu-elem}(fid) = \text{locate-exec-sel}(fid) \]
\[ \text{bu}(fid) = \text{s-bu}\cdot\text{fu-elem}(fid) = \text{bu-sel}(fid) \]
\[ \text{kbu}(fid) = \text{s-kbu}\cdot\text{fu-elem}(fid) = \text{kbu-sel}(fid) \]
\[ \text{locking}(fid) = \text{s-locking}\cdot\text{fu-elem}(fid) = \text{locking-sel}(fid) \]
ext-stor(fid) = ddname(fid) (ES)
data-elem(fid,i) = elem(i,s-data(ext-stor(fid))) =
data-elem-sel(fid,i) (ES)
key(fid,i) = s-key(data-elem(fid,i)) =
key-sel(fid,i) (ES)
rec-length(fid,i) = s-rec-length(data-elem(fid,i)) =
rec-length-sel(fid,i) (ES)
datum(fid,i) = s-datum(data-elem(fid,i)) =
datum-sel(fid,i) (ES)
page(fid,i) = s-page(datum(fid,i)) = page-sel(fid,i) (ES)
line(fid,i) = s-line(datum(fid,i)) = line-sel(fid,i) (ES)
char(fid,i) = s-char(datum(fid,i)) = char-sel(fid,i) (ES)
character(fid,i) = s-character(datum(fid,i)) =
character-sel(fid,i) (ES)
ext-contr(fid) = s-ext-contr(ext-stor(fid))
header(fid) = s-header(ext-contr(fid)) =
header-sel(fid) (ES)
trailer(fid) = s-trailer(ext-contr(fid)) =
trailer-sel(fid) (ES)
next(fid) = s-next(ext-contr(fid)) = next-sel(fid) (ES)
end-file(fid) = s-end-file(ext-contr(fid)) =
end-file-sel(fid) (ES)
curr-page(fid) = s-curr-page(ext-contr(fid)) =
curr-page-sel(fid) (ES)
curr-line(fid) = s-curr-line(ext-contr(fid)) =
curr-line-sel(fid) (ES)
curr-char(fid) = s-curr-char(ext-contr(fid)) =
curr-char-sel(fid) (ES)
pagesize(fid) = s-pagesize(ext-contr(fid)) =
pagesize-sel(fid) (ES)
linesize(fid) = s-linesize(ext-contr(fid)) =
linesize-sel(fid) (ES)
end-page(fid) = s-end-page(ext-contr(fid)) =
end-page-sel(fid) (ES)
6.2 Opening of a File

Opening comprises all actions necessary for proper execution of subsequent I/O-statements. These actions mainly consist in checking entries which are already in the file directory FD, the file union directory FU, and the external storage ES, or in making new entries in the above constituents of the state. The difference between explicit and implicit opening is preserved.

6.2.1 File description attributes

Abstract Syntax:

\[
\text{is-file} = (\text{s-file-attr:is-file-attr-set},),
\text{s-io-env:is-io-opt-set},
\text{s-title:is-const})
\]

\[
\text{is-file-attr} = (\text{is-stream} | \text{is-record} | \text{is-input} | \text{is-output} |
\text{is-update} | \text{is-sequential} | \text{is-direct} |
\text{is-buffered} | \text{is-unbuffered} | \text{is-print} |
\text{is-backwards} | \text{is-exclusive} | \text{is-keyed})
\]

The selector s-io-env yields the evaluated options of the environment attribute. They are not used in the interpretation of statements. The selector s-title gives a value of type character string which is used again to give the name of a data set to be associated at opening if no title-option exists in the open-statement. This is necessary because of the rules for passing arguments to filename parameters.

The remaining thirteen file attributes to which is-file-attr-set applies are the attributes which are placed in the file union directory FU during opening, and which are accessible through s-csa.

The subject of the following is the definition of the function make-csa(open-attrset,attrset) used in 6.2.2 to yield the so-called
complete set of file attributes from the file attributes specified at opening and the attributes in the attribute directory AT.

The following identities apply to the above thirteen file attributes:

\[
\text{is-stream(set)} \equiv \text{STREAM} \in \text{set} \\
\text{is-keyed(set)} \equiv \text{KEYED} \in \text{set}
\]

Definition:

\[
\text{complete-file-attrset-list} = \\
\langle \{\text{STREAM, INPUT}\} , \\
\{\text{STREAM, OUTPUT}\} , \\
\{\text{STREAM, OUTPUT, PRINT}\} , \\
\{\text{RECORD, INPUT, SEQUENTIAL, BUFFERED}\} , \\
\{\text{RECORD, INPUT, SEQUENTIAL, BUFFERED, BACKWARDS}\} , \\
\{\text{RECORD, INPUT, SEQUENTIAL, BUFFERED, KEYED}\} , \\
\{\text{RECORD, INPUT, SEQUENTIAL, BUFFERED, KEYED, BACKWARDS}\} , \\
\{\text{RECORD, OUTPUT, SEQUENTIAL, BUFFERED}\} , \\
\{\text{RECORD, OUTPUT, SEQUENTIAL, BUFFERED, KEYED}\} , \\
\{\text{RECORD, UPDATE, SEQUENTIAL, BUFFERED}\} , \\
\{\text{RECORD, UPDATE, SEQUENTIAL, BUFFERED, KEYED}\} , \\
\{\text{RECORD, INPUT, SEQUENTIAL, UNBUFFERED}\} , \\
\{\text{RECORD, INPUT, SEQUENTIAL, UNBUFFERED, BACKWARDS}\} , \\
\{\text{RECORD, INPUT, SEQUENTIAL, UNBUFFERED, KEYED}\} , \\
\{\text{RECORD, INPUT, SEQUENTIAL, UNBUFFERED, KEYED, BACKWARDS}\} , \\
\{\text{RECORD, OUTPUT, SEQUENTIAL, UNBUFFERED}\} , \\
\{\text{RECORD, OUTPUT, SEQUENTIAL, UNBUFFERED, KEYED}\} , \\
\{\text{RECORD, UPDATE, SEQUENTIAL, UNBUFFERED}\} , \\
\{\text{RECORD, UPDATE, SEQUENTIAL, UNBUFFERED, KEYED}\} , \\
\{\text{RECORD, INPUT, DIRECT, KEYED}\} , \\
\{\text{RECORD, OUTPUT, DIRECT, KEYED}\} , \\
\{\text{RECORD, UPDATE, DIRECT, KEYED}\} , \\
\{\text{RECORD, UPDATE, DIRECT, KEYED, EXCLUSIVE}\} \rangle
\]
is-incompl-file-attrset(set, i) =
    set ∈ elem(i, complete-file-attrset-list)

assoc-index(set) =
    (∃ i)(is-incompl-file-attrset(set, i))
    (li)(∀ k)(is-incompl-file-attrset(set, i) &
        ¬is-incompl-file-attrset(set, k) &
        k < i)
    T  →  EMPTY

assoc-complete-file-attrset(set) =
    is-empty(assoc-index(set))  →  EMPTY
    T  →  elem(assoc-index(set), complete-file-attrset-list)

make-csa(open-attrset, attrset) =
    assoc-complete-file-attrset(open-attrset ⨆ attrset)
6.2.2 Interpretation of the open-statement

Abstract Syntax:

\[
\text{is-open-st} = \text{is-open-list} \\
\text{is-open} = (\langle s-st:is-op \rangle, \\
\langle s-file-opt:is-name \rangle, \\
\langle s-ident-opt: (is-expr | is-\emptyset) \rangle, \\
\langle s-title-opt: (is-expr | is-\emptyset) \rangle, \\
\langle s-pagesize-opt: (is-expr | is-\emptyset) \rangle, \\
\langle s-linesize-opt: (is-expr | is-\emptyset) \rangle, \\
\langle s-open-attr:is-file-attr-set \rangle)
\]

Function Definition:

\[
\text{select-alternat}(\text{impl},\text{expl}) = \\
\begin{array}{ll}
\text{is-empty}(\text{expl}) & \rightarrow \text{expl} \\
\text{T} & \rightarrow \text{impl}
\end{array}
\]

\[
\text{select-value}(\text{opt},\text{ident}) = \\
\begin{array}{ll}
\text{is-simple-ref}(\text{opt}) & \rightarrow \text{value}(\text{ident},S) \\
\text{T} & \rightarrow \text{ident}
\end{array}
\]

Implementation-defined functions:

\[
\text{translate-ddname}(\text{op}) \\
\text{translates the operand op to a ddname.}
\]

\[
\text{translate-char}(\text{user-label}) \\
\text{translates an user-label to a character-string operand}
\]

\[
\text{translate-user-label}(\text{op}) \\
\text{translates op which is a character string operand to an user-label}
\]
Abbreviation:

\[ \text{fid} = s\text{-file-opt}(t) \]

Definition:

\[
\text{int-open-st}(t) = \text{int-open-undf}(t, vlist); \\
\{ \text{elem}(i) \cdot vlist: \text{int-open}(\text{elem}(i, t), i) \\
\mid 1 \leq \text{islength}(t) \}
\]

\[
\text{int-open-undf}(t, \text{ind-list}) = \\
\text{is-emptylist}(\text{ind-list}) \rightarrow \text{null} \\
\text{is-empty}(\text{head}(\text{ind-list})) \rightarrow \\
\text{int-open-undf}(t, \text{tail}(\text{ind-list})) \\
T \rightarrow \text{int-open-undf}(t, \text{tail}(\text{ind-list})); \\
\text{raise-undf-cond}(\text{elem}(\text{head}(\text{ind-list}), t))
\]

\[
\text{int-open}(t, \text{ind}) = \\
\text{associate}(t, \text{ind}, \text{page}, \text{line}, \text{ident}, \text{make-csa}(s\text{-open-attr}(t), \text{attr}(\text{fid}))), \text{ddname}; \\
\text{page} : \text{int-opt-expr}(\text{INTG}, s\text{-pagesize-opt}(t)) \\
\text{line} : \text{int-opt-expr}(\text{INTG}, s\text{-linesize-opt}(t)) \\
\text{ident} : \text{int-ident-opt}(s\text{-ident-opt}(t)) \\
\text{open-title} : \text{int-opt-expr}(\mu_0(<s\text{-type}: \text{CHAR}>, <s\text{-length}: \text{EMPTY}>) , s\text{-title-opt}(t))
\]

where \[ \text{ddname} = \text{translate-ddname}(\text{select-alternat}(s\text{-title}(\text{attr}(\text{fid})), \text{open-title})) \]

\[
\text{int-opt-expr}(\text{spec}, \text{expr}) = \\
\neg \text{exists}(\text{expr}) \rightarrow \text{RETURNS} : \text{EMPTY} \\
T \rightarrow \text{int-expr-conv}(\text{spec}, s\text{-op}(v)); \\
v : \text{int-expr}(\text{expr}, \text{SC-EI})
\]
\[\text{int-expr-conv}(\text{spec,op}) = \text{RETURNS} : \text{convert}(\text{spec,op})\]

\[\text{int-ident-opt}(\text{idopt}) =\]
\[\begin{cases} \text{is-simple-ref}(\text{idopt}) & \rightarrow \text{int-variable}(\text{idopt,E}) \\ T & \rightarrow \text{int-opt-expr}(\mu_0(\langle s\text{-type:\text{CHAR}},<s\text{-length:\text{EMPTY}}\rangle), \text{idopt}) \end{cases}\]

\[\text{associate}(\text{te}, \text{ind}, \text{page}, \text{line}, \text{ident}, \text{new-csa}, \text{ddname}) =\]
\[\begin{cases} \text{is-empty}(\text{new-csa}) \lor \\ \text{is-undf-dataset}(\text{ddname}(\text{DS})) \lor \\ \neg\text{is-output}(\text{new-csa}) \land \text{exists}(\text{s-ident-opt}(\text{te})) \lor \\ \text{is-simple-ref}(\text{s-ident-opt}(\text{te})) & \rightarrow \text{RETURNS} : \text{ind} \\ \text{exists}(\text{fa}(\text{fid})) \land (\text{new-csa} \supset \text{csa}(\text{fid})) \\ \land (\text{new-csa} \subset \text{csa}(\text{fid})) & \rightarrow \text{RETURNS} : \text{EMPTY} \\ T & \rightarrow \text{update-states}(\text{f,te,page,line,ident,new-csa,ddname}); \text{f : un-f} \end{cases}\]

\[\text{update-states}(\text{f,te,page,line,ident,csa,ddname}) =\]
\[\begin{cases} \text{null}; \\ \{ \text{int-open-label}(\text{te,ident}), \\ \text{update-es-part}(\text{end-file-sel}(\text{fid}), \mathcal{Q}) \} ; \\ \text{int-print}(\text{te,select-alternat}(\text{INTPS,page}), \\ \text{select-alternat}(\text{INTLS,page})); \\ \{ \text{update-fu-open}(\text{f,csa,ddname}), \\ \text{update-fd-open}(\text{f,den(fid)}) \} \end{cases}\]

\[\text{Note:} \quad \text{INTPS and INTLS are implementation-defined integers.}\]

\[\text{update-fd-open}(\text{den,f}) =\]
\[\text{FD} : \mu(\text{PD}, \langle \text{den\text{-s-nfa:f}} \rangle)\]

\[\text{update-fu-open}(\text{f,csa,ddname}) =\]
\[\text{FU} : \mu(\text{FU}, \langle \text{f}\mu_0(\{ \langle \text{s-ddname:ddname},<s\text{-csa:csa} \rangle \} ) \rangle)\]
\textbf{int-print}(te,p,l) = \\
is-print(csa(fid)) \rightarrow \\
\quad \text{null; } \\
\quad \text{update-es-part(pagesize-sel(fid),p), } \\
\quad \text{update-es-part(linesize-sel(fid),l), } \\
\quad \text{update-es-part(curr-page-sel(fid),INTCP), } \\
\quad \text{update-es-part(curr-line-sel(fid),INTCL), } \\
\quad \text{update-es-part(curr-char-sel(fid),INTCC), } \\
\quad \text{update-es-part(end-page-sel(fid),\varnothing)} \\
T \rightarrow \text{null}

\textbf{Note:} \quad \text{INTCP, INTCL and INTCC are implementation-defined integers.}

\textbf{int-open-label}(te,ident) = \\
is-empty(ident) \rightarrow \text{null} \\
\neg \text{is-output(csa(fid)) } \& \\
\text{is-backwards(csa(fid)) } \rightarrow \\
\quad \text{assign(ident,translate-char(trailer(fid)))} \\
\neg \text{is-output(csa-(fid)) } \& \\
\neg \text{is-backwards(csa(fid)) } \rightarrow \\
\quad \text{assign(ident,translate-char(header(fid)))} \\
T. \rightarrow \text{update-es-part(header-sel(fid),header-label)}

\textbf{where} \quad \text{header-label = } \\
\quad \text{translate-user-label(select-value(s-ident-opt(te),ident))}
6.3 Stream Transmission

6.3.1 The Get-Statement

6.3.11 Source items and elementary input instructions

Source items are the elementary data coming from a string or the stream.

Abstract syntax of source items:

is-data-string-item = is-data-string-ass | is-string-end
is-data-string-ass = (<s-leftpart:is-concr-name | is-inv-name>,
                    <s-rightpart:is-concr-data | is-inv-data>,
                    <s-rest-string:is-pair>)

is-data-stream-item = is-data-stream-ass | is-group-end | is-end-file
is-data-stream-ass = (<s-leftpart:is-concr-name | is-inv-name>,
                     <s-rightpart:is-concr-data | is-inv-data>,
                     <s-concr-rep:is-concr-rep>)

is-group-end = (<s-concr-rep:is-concr-rep>)

is-concr-rep = is-data-elem-list

Note: is-data-elem-list is defined with ES.

is-list-string-item = is-list-string | is-string-end
is-list-string = (<s-data:is-concr-data |
                  is-inv-data |
                  is-concr-comma>,
                  <s-rest-string:is-pair>)

is-list-stream-item = is-list-stream | is-end-file
is-list-stream = (<s-data:is-concr-data |
                  is-inv-data |
                  is-concr-comma>,
                  <s-concr-rep:is-concr-rep>)}
is-edit-string-item = is-edit-string | is-string-end
is-edit-string = (<s-data:is-concr-data | is-inv-data>,
                 <s-rest-string:is-pair>)

is-edit-stream-item = is-edit-stream | is-end-file
is-edit-stream = (<s-data:is-concr-data | is-inv-data>,
              <s-concr-rep:is-concr-rep>)

The concrete syntax of source items, and the translation from
the concrete representation to the abstract representation is not
dealt with in the sequel. Whenever the source is a string, source items
come from the storage S; when the source is the stream, source items
come from the external storage ES.

Elementary input instructions are instructions which return ob­
jects corresponding to the abstract syntax of source items.

Let the names "string", "fid", and "eval-format" represent a
scalar part of type character-string, a file identifier, and an evalu­
ated format item, respectively. The following elementary input in­
structions apply if input is from a string:

\texttt{get-data-string} \texttt{(string)} used for data-directed input returns a
source item which conforms to \texttt{is-data-string-item};

\texttt{get-string} \texttt{(string,eval-format)} is used for list-directed input
if eval-format is undefined (source item returned conforms to
\texttt{is-list-string-item}) or for edit-directed input otherwise (source
item returned conforms to \texttt{is-edit-string-item}).

The state is not changed.

If input is from the stream

\texttt{act-data-stream} \texttt{(fid)} is executed similar to
\texttt{get-data-string} \texttt{()} but the predicate is to be replaced by
\texttt{is-data-stream-item},
\texttt{get-stream} (fid, eval-format) is executed similar to \texttt{get-string} () but the predicates have to be replaced by \texttt{is-list-stream-item} and \texttt{is-edit-stream-item}, respectively.

The state can be changed when the \textsc{transmit}-condition is raised at some unspecified point.

The elementary input instructions are not defined formally at the present.

Whenever the names \texttt{inp-inf}, \texttt{string-inf}, \texttt{stream-inf}, \texttt{ret-string-inf}, \texttt{ret-stream-inf} are used by the interpreter, they are names of source items.

Let \texttt{inp-inf} be the source item returned by an elementary input instruction and \texttt{concr-rep} be \texttt{s-concr-rep (inp-inf)}. In the case of input from a string, \texttt{concr-rep} is undefined and so is \texttt{length (concr-rep)}. In the case of input from the stream, \texttt{concr-rep} is always defined and is an exact copy of the concrete representation of the source item. In the case of edit-directed input, \texttt{length (concr-rep)} is determined by \texttt{eval-format}, and \texttt{s-data (inp-inf)} is the result of the application of \texttt{eval-format} to \texttt{concr-rep}.

6.3.12 The Specification-Information

Abstract Syntax:

\[
is\texttt{-spec-inf} = (<\texttt{s-string:is-pair} \mid \texttt{is-∅}) <\texttt{s-fl:is-format-elem-list} \mid \texttt{is-∅} >, <\texttt{s-first:is-empty} \mid \texttt{is-∅} >, <\texttt{s-comma:is-empty} \mid \texttt{is-∅} >, <\texttt{s-end:is-empty} \mid \texttt{is-∅} >)
\]
Comments:

The specification information comprises all the information needed for proper instruction execution. It behaves much like an object initiated by the instruction \texttt{int-get-st} which is currently updated dependent on the source items etc., and with the property that after every updating only the latest version of the object is available during the subsequent interpretation. In order not to introduce a new concept in the control the specification-information is treated like any other argument.

All names ending on \texttt{spec-inf} represent specification-information conforming to \texttt{is-spec-inf}.

The different selectors yield (in the same sequence as in the abstract syntax) the current string (in the case of input from a string), the current format-elem-list (in the case of edit-directed input only), the information whether the next source item is the very first or not (in the case of data- and list-directed input), the information whether the last source item corresponded to \texttt{is-concr-comma} (in the case of list-directed input only), and the information whether end of the stream or end of the string have occurred or not.

6.3.13 Notes to the interpretation

Conditions other than I/O conditions have not been considered. The functions \texttt{ge,gt,le,lt,eq} are equivalent with the function \texttt{infix-op} (with appropriate arguments) used in 5.3.3. The sequence of the instructions \texttt{get-data-stream} or \texttt{get-stream} and \texttt{update-stream} is not realistic. However, it simplifies interpretation and is not in conflict with the language.
6.3.14 Interpretation of the Get-Statement

Abstract Syntax

\[
\text{is-get-st} = (\text{is-stream-get} | \text{is-string-get})
\]

\[
\text{is-stream-get} = (\langle \text{s-st} : \text{is-get} \rangle, \\
\langle \text{s-file-opt}:\text{is-name} \rangle, \\
\langle \text{s-spec}:\text{is-get-spec} \rangle, \\
\langle \text{s-copy-opt}:\text{is-opt} \rangle, \\
\langle \text{s-mark-opt}:\text{is-opt} \rangle)
\]

\[
\text{is-string-get} = (\langle \text{s-st}:\text{is-get} \rangle, \\
\langle \text{s-string-opt}:\text{is-ref} \rangle, \\
\langle \text{s-spec}:\text{is-get-spec} \rangle)
\]

\[
\text{is-get-spec} = ((\langle \text{s-type}:\text{is-data-dir} \rangle, \\
\langle \text{s-ref-set}:\text{is-ref-set} \rangle) | \\
(\langle \text{s-type}:\text{is-edit-dir} \rangle, \\
\langle \text{s-data-list}:\text{is-ref-elem-list} \rangle, \\
\langle \text{s-format-list}:\text{is-format-elem-list} \rangle) | \\
(\langle \text{s-type}:\text{is-iList-dir} \rangle, \\
\langle \text{s-data-list}:\text{is-ref-elem-list} \rangle))
\]

\[
\text{is-ref-elem} = (\text{is-ref} | \text{is-ref-contr})
\]

\[
\text{is-ref-contr} = (\langle \text{s-contr-var}:\text{is-simple-ref} \rangle, \\
\langle \text{s-spec-list}:\text{is-do-spec-list} \rangle, \\
\langle \text{s-data-list}:\text{is-ref-elem-list} \rangle)
\]

\[
\text{is-do-spec} = (\langle \text{s-init-expr}:\text{is-expr} \rangle, \\
\langle \text{s-by-expr}: (\text{is-expr} | \text{is-Q}) \rangle \\
\langle \text{s-to-expr}: (\text{is-expr} | \text{is-Q}) \rangle \\
\langle \text{s-while-expr}:\text{is-expr} \rangle)
\]

1) SYSIN inserted by default by prepass
2) proposed in TNL to SRL-3 to be released
3) is-expr(s-to-exr) $\supset$ is-expr(s-by-exr)
4) T inserted by default by prepass
is-format-elem = is-format-item |
   (\<s-rep-factor:is-expr>,
   \<s-elem:is-format-item |
      is-format-elem-list>)

is-format-item = is-data-format | is-control-format | 2)
   is-remote-format

is-data-format = is-real-format | is-compl-format |
   is-string-format | is-pict-format

is-real-format = (\<s-format-type:is-fix | is-flt>,
   \<s-w:is-expr>,
   \<s-d:is-expr | is-تقدم>,
   \<s-p:is-expr | is-تقدم>,
   \<s-s:is-expr | is-تقدم>
   is-pict-format)

is-compl-format = (\<s-format-type:is-compl>,
   \<s-real-part:is-real-format |
   \<s-imag-part:is-real-format |
   is-pict-format>)

is-string-format = (\<s-format-type:(is-bit | is-char)>,
   \<s-w:is-expr>)

is-pict-format = (\<s-format-type:is-picture>,
   \<s-pict-spec:is-const>)

is-control-format = (\<s-format-type:is-control-format-type>,
   \<s-w:(is-expr | is-تقدم)>)

is-control-format-type = (is-space | is-page | is-skip |
   is-line | is-column)

is-remote-format = (\<s-format-type:is-remote>,
   \<s-label-ref:is-ref>)

1) is-const ⊆ is-expr
2) abstract syntax of format-item applies to the non-evaluated and evaluated case
3) is-get-st(t) ⊆
   \neg (\exists \alpha)(is-control-format(\alpha \cdot t) &
   \neg is-space(s-format-type \alpha \cdot t))
Definition:

\[
\text{int-get-st}(t) = \text{is-stream-get}(t) \quad \rightarrow \quad \text{int-stream-get}(t); \\
\quad \text{int-stream-open}(t)
\]

\[
\text{is-string-get}(t) \rightarrow 
\]

\[
\text{int-end-string}(\text{ret-spec-inf}); \\
\quad \text{ret-spec-inf} : \text{int-data-spec}(t, \text{spec-inf}); \\
\quad \text{gen: int-variable}(\text{s-string-opt}(t), \text{spec-inf})
\]

where \( \text{spec-inf} = \mu_\circ \{ \langle \text{s-string:gen}, \text{init-ei(s-da(gen))} \rangle, \langle \text{s-first:EMPTY} \rangle \} \)

\[
\text{int-stream-open}(t) = \text{exists}(\text{s-copy-opt}(t)) \rightarrow \text{int-open-st}(\text{tinp} \circ \text{tout}) \\
\quad \rightarrow \text{int-open-st}(\text{tinp})
\]

where \( \text{tinp} = \mu_\circ (\langle \text{s-file-opt:s-file-opt}(t) >, \langle \text{s-open-att: \{ \text{STREAM,INPUT} \} \rangle \rangle) \)

\( \text{tout} = \mu_\circ (\langle \text{s-file-opt:SYSPRINT}, \langle \text{s-open-attr: \{ \text{STREAM,OUTPUT} \} \rangle \rangle) \)

\[
\text{int-stream-get}(t) = \text{exists}(\text{end-file}(\text{s-file-opt}(t))) \rightarrow \text{raise-endfile-cond}(t) \\
\quad \rightarrow \text{int-end-file}(t, \text{ret-spec-inf}); \\
\quad \text{ret-spec-inf} : \\
\quad \text{int-data-spec}(t, \mu_\circ (\langle \text{s-first:EMPTY} \rangle))
\]

\[
\text{int-end-file}(t, \text{spec-inf}) = \quad \rightarrow \text{raise-endfile-cond}(t) \\
\quad \text{exists}(\text{s-end}(\text{spec-inf})) \rightarrow \text{raise-endfile-cond}(t) \\
\quad \text{exists}(\text{s-mark-opt}(t)) \rightarrow \quad 1) \\
\quad \text{update-es-part}(\text{next-sel}(\text{s-file-opt}(t)), \text{INTNEXT}) \\
\quad \rightarrow \text{null}
\]

1) implementation defined integer. The mark-option is proposed in a TNL to SRL-3.
update-es-part(selector, object) =
    ES : μ(ES, <selector:object>)

int-end-string(spec-inf) =
    exists(s-end(spec-inf)) → raise-error-cond
    T → null

Let

\[
\text{type} = \text{s-type}\cdot\text{s-spec}(t) \\
\text{data-list} = \text{s-data-list}\cdot\text{s-spec}(t) \\
\text{format-list} = \text{s-format-list}\cdot\text{s-spec}(t)
\]

int-data-spec(t,spec-inf) =
    is-data-dir(type) → int-data-dir(t,spec-inf)
    T → int-data-list(t,data-list,edit-spec-inf)

where:

\[
\text{edit-spec-inf} = \mu(\text{spec-inf}, <s-fl:format-list>)
\]

int-data-list(t, data-list, spec-inf) =
    exists(s-end(spec-inf)) ∨ (data-list = < >) →
    \text{RETURNS}: spec-inf
    T → int-data-list(t,tail(data-list),ret-spec-inf);
    ret-spec-inf :
        int-data-list-elem(t,head(data-list), spec-inf)

int-data-list-elem(t,elem,spec-inf) =
    is-expr(elem) → int-elem(t,elem,spec-inf)
    T → interpreter-spec-list(t,s-data-list(elem),
                                s-spec-list(elem),gen,spec-inf);
    gen : int-variable(s-contr-var(elem),E)

1) may cause raise-transmit-cond(t) to be executed.
Let \( \text{spec} = \text{head} (\text{spec-list}) \)

\[
\text{interpr-spec-list}(t, \text{data-list}, \text{spec-list}, \text{gen}, \text{spec-inf}) = \\
\quad \exists (s\text{-end}(\text{spec-inf})) \vee (\text{spec-list} = < >) \quad \rightarrow \\
\qquad \text{RETURNS : spec-inf} \\
T \rightarrow \text{interpr-spec-list}(t, \text{data-list}, \text{tail}(\text{spec-list}), \text{gen}, \text{ret-spec-inf});
\]

\[
\text{ret-spec-inf : interpr-do}(t, \text{data-list}, \text{gen}, s\text{-op}(v_2), s\text{-op}(v_3), s\text{-op}(v_4), \\
\quad s\text{-while-expr}(\text{spec}), \text{spec-inf});
\]

\[
v_2 : \text{int-spec-part}(s\text{-by-expr}(\text{spec})), \\
v_3 : \text{int-spec-part}(s\text{-to-expr}(\text{spec})), \\
v_4 : \text{int-expr}(s\text{-while-expr}(\text{spec}), \text{SC-EI}), \\
\text{assign}(\text{gen}, s\text{-op}(v_1)); \\
v_1 : \text{int-expr}(s\text{-init-expr}(\text{spec}), \text{SC-EI})
\]

\[
\text{int-spec-part}(\text{expr}) = \\
\quad \neg \exists (\text{expr}) \quad \rightarrow \text{RETURNS : EMPTY} \\
T \rightarrow \text{int-expr}(\text{expr}, \text{SC-EI})
\]

\[
\text{interpr-do}(t, \text{data-list}, \text{gen}, v_2, v_3, v_4, \text{while-expr}, \text{spec-inf}) = \\
\quad \exists (s\text{-end}(\text{spec-inf})) \vee \neg \text{truth}(v_4) \quad \rightarrow \\
\qquad \text{RETURNS : spec-inf} \\
\neg \exists (v_2) \& \neg \exists (v_3) \quad \rightarrow \\
\quad \text{int-data-list}(t, \text{data-list}, \text{spec-inf}) \\
\neg \exists (v_3) \quad \rightarrow \text{execute-iterate}(t, \text{data-list}, \text{gen}, v_2, v_3, v_4, \text{while-expr}, \text{spec-inf}) \\
\neg (\text{truth}(\text{ge}(v_3, 0)) \& \text{truth}(\text{gt}(\text{value}(\text{gen}, S), v_2)) \vee \\
\text{truth}(\text{lt}(v_3, 0)) \& \text{truth}(\text{lt}(\text{value}(\text{gen}, S), v_2))) \quad \rightarrow \\
\quad \text{execute-iterate}(t, \text{data-list}, \text{gen}, v_2, v_3, v_4, \text{while-expr}, \text{spec-inf}) \\
T \rightarrow \text{RETURNS : spec-inf}
execute-iterate(t, data-list, gen, v₂, v₃, v₄ while-expr, spec-inf) = 
   iterate-contr-var(t, data-list, gen, v₂, v₃, v₄ while-expr, 
                  ret-spec-inf);
   ret-spec-inf : int-data-list(t, data-list, spec-inf)

iterate-contr-var(t, data-list, gen, v₂, v₃, v₄ while-expr, spec-inf) = 
   internr-do(t, data-list, gen, v₂, v₃ s-op(v), while-expr, spec-inf);
   v : int-expr(while-expr, SC-EL),
   assign(gen, s-op(expr));
   expr : int-infix-op(v₂, value(gen, S), ADD)

int-elm(t, elem, spec-inf) = 
   is-get-st(t) 
   int-list-edit-gen(t, gen, init-ei(s-da(gen)), spec-inf);
   gen : int-variable(elem, E)
   is-put-st(t) 
   int-put-elm(t, elem, spec-inf)

int-data-dir(t, spec-inf) = 
   is-string-get(t) 
   int-data-item(t, ret-string-inf, spec-inf);
   ret-string-inf : get-data-string(s-string(spec-inf))
   is-stream-get(t) 
   int-data-stream-item(t, ret-stream-inf, spec-inf);
   ret-stream-inf: get-data-stream(s-file-opt(t))

int-data-stream-item(t, stream-inf, spec-inf) = 
   null;
   int-data-item(t, stream-inf, spec-inf);
   update-stream(t, stream-inf, spec-inf)
Let \( \text{length} = \text{length}(\text{s-concr-rep}(\text{stream-inf})) \)

\[
\text{fid} = \text{s-file-opt}(t) \\
\text{type} = \text{s-type-s-spec}(t) \\
\text{update-stream}(t, \text{stream-inf}, \text{spec-inf}) = \\
\text{is-end-file}(\text{stream-inf}) \rightarrow \text{update-es-part}(\text{end-file-sel}(\text{fid}), \text{EMPTY}) \\
\text{le}({\text{length}}, 0) \rightarrow \text{null} \\
\text{exists}(\text{s-first}(\text{spec-inf})) \& \text{is-data-dir}(\text{type}) \rightarrow \\
\text{int-copy}(t, \text{stream-inf}); \\
\text{int-next}(\text{fid}, \text{length}), \\
\text{int-count}(\text{fid}, 0) \\
\text{is-data-dir}(\text{type}) \rightarrow \text{int-copy}(t, \text{stream-inf}); \\
\text{int-next}(\text{fid}, \text{length}), \\
\text{int-count}(\text{fid}, \text{count}(\text{fid})) \\
T \rightarrow \text{int-copy}(t, \text{stream-inf}); \\
\text{int-next}(\text{fid}, \text{length})
\]

\[
\text{int-next}(\text{fid}, \text{length}) = \\
\text{update-es-part}(\text{next-sel}(\text{fid}), \text{val-next}); \\
\text{val-next} : \text{int-infix-op}(\text{next}(\text{fid}), \text{length}, \text{ADD})
\]

\[
\text{int-count}(\text{fid}, \text{val}) = \\
\text{update-fu-part}(\text{count-sel}(\text{fid}), \text{val-count}); \\
\text{val-count} : \text{int-infix-op}(\text{val}, 1, \text{ADD})
\]

\[
\text{update-fu-part}(\text{selector}, \text{object}) = \\
\text{FU} : \mu(\text{FU}, <\text{selector}: \text{object}>)
\]

\[
\text{int-copy}(t, \text{stream-inf}) = \\
\text{exists}(\text{s-copy-opt}(t)) \rightarrow \\
\text{copy}(\text{s-concr-rep}(\text{stream-inf})) \\
T \rightarrow \text{null}
\]
Let name = s-leftpart(inp-inf)
data = s-rightpart(inp-inf)

\[ \text{int-data-item}(t, \text{inp-inf}, \text{spec-inf}) = \]
\[ \text{is-string-end}(\text{inp-inf}) \lor \text{is-end-file}(\text{inp-inf}) \]
\[ \text{RETURNS : } \mu(\text{spec-inf}, <\text{s-end}: \text{EMPTY}>) \]
\[ \text{is-group-end}(\text{inp-inf}) \]
\[ \text{RETURNS : } \text{spec-inf} \]
\[ \text{is-inv-name}(\text{name}) \lor \text{is-inv-data}(\text{data}) \]
\[ \text{int-data-dir}(t, \text{ret-spec-inf}); \]
\[ \text{raise-error-cond} \]
\[ T \]
\[ \text{int-data-dir}(t, \text{ret-spec-inf}); \]
\[ \text{int-data-ass}(t, \text{abstr-name}, \text{abstr-data}); \]
\[ \text{abstr-name: } \text{translate-name}(\text{name}), \]
\[ \text{abstr-data: } \text{translate-data}(\text{data}) \]

\[ \text{where } \text{ret-spec-inf} = \mu(\text{spec-inf}, <\text{s-string}: \text{s-rest-string}(\text{inp-inf})>, <\text{s-first}: \emptyset>) \]

Let ref-set = s-ref-set \cdot s-spec(t)

\[ \text{int-data-ass}(t, \text{name}, \text{data}) = \]
\[ \neg \text{is-scalar-ref}(\text{name}) \]
\[ \text{int-error}(t) \]
\[ \text{subpart}(\text{name}, \text{ref-set}) \]
\[ \text{assign}(\text{gen}, s-op(\text{val})); \]
\[ \text{gen: } \text{int-variable}(\text{name}, E), \]
\[ \text{val: } \text{int-expr}(\text{data}, SC-EI) \]
\[ T \]
\[ \text{int-error}(t) \]

1) \text{translate-name}(\text{concr-name}) \text{ translates a concrete reference to an abstract reference. If the abstract reference is invalid, the result is } \emptyset .

2) \text{translate-data}(\text{concr-data}) \text{ translates a concrete constant or a complex expression to the abstract object.}
where \( \text{is-scalar-ref}(t) = \text{is-scalar}(\text{new-da}(\text{da},rl)) \)

and where \( rl = \text{ref-list}(\text{da},\text{subs-list},\text{qual-list}) \)
\[
\begin{align*}
\text{da} &= \text{s-da}\cdot\text{s-attr}((\text{id}(t)(E))(AT)) \\
\text{subs-list} &= \text{s-arg-list}(t) \\
\text{qual-list} &= \text{s-qual-list}(t)
\end{align*}
\]

Note: \( \text{subpart}((\text{single-ref},\text{ref-set}) \) is true if \( \text{ref-set} \) is the null set or if \( \text{single-ref} \) is a part contained in \( \text{ref-set} \), otherwise false.

\[
\begin{align*}
\text{int-error}(t) &= \quad \text{is-string-get}(t) \quad \rightarrow \quad \text{raise-error-cond} \\
\text{int-error}(t) &= \quad \text{is-stream-get}(t) \quad \rightarrow \quad \text{raise-name-cond}(t)
\end{align*}
\]

Let \( \text{hfe} = \text{head}(\text{s-fl}(\text{spec-inf})) \)
\( \text{tfe} = \text{tail}(\text{s-fl}(\text{spec-inf})) \)

\[
\begin{align*}
\text{int-list-edit-gen}(t,\text{gen},\text{ei},\text{spec-inf}) &= \quad \text{is-exhaust}(\text{ei}) \quad \rightarrow \quad \text{RETURNS} : \text{spec-inf} \\
\text{int-list-dir}(\text{s-type}\cdot\text{s-spec}(t)) &= \quad \text{int-list-edit-gen}(t,\text{gen},\text{ei},\text{spec-inf}) \\
\text{s-fl}(\text{spec-inf}) &= \langle \rangle \quad \rightarrow \quad \\
\text{int-list-edit-gen}(t,\text{gen},\text{ei},\text{format-spec-inf}) &= \quad \text{is-format-item}(\text{hfe}) \quad \rightarrow \quad \\
\text{int-list-edit-gen}(t,\text{gen},\text{ei}, \mu(\text{spec-inf}, \langle\text{s-fl:ret-fl}\rangle)) &= \quad \text{ret-fl} : \text{int-format-head}(\text{convert}(\text{INT},\text{s-op}(\text{rep})), \text{s-element}(\text{hfe},\text{tfe})); \\
\text{rep:int-expr} &= \text{s-rep-factor}(\text{hfe},\text{SC-EI}) \quad \rightarrow \quad \\
\text{is-remote-format}(\text{hfe}) &= \quad \text{int-list-edit-gen}(t,\text{gen},\text{ei}, \mu(\text{spec-inf}, \langle\text{s-fl:s-vr:s-op(val) tfe}\rangle)) \quad \rightarrow \quad \\
\text{val} : \text{int-ref}(\text{s-label-ref}(\text{hfe})) &= \quad \rightarrow \quad 
\end{align*}
\]
(int-list-edit-gen(), continued)

\[
T \mapsto \text{int-elem-inp}(t, \text{gen}, ei, \mu(\text{spec-inf},
\text{head}\cdot s-fl:val-hfi)));
\]

\(
\{\kappa\cdot \text{val-hfi} : \text{int-expr-op}(\kappa\cdot hfe) \mid
\text{is-expr}(\kappa\cdot hfe)\}
\)

where

\[
\text{format-spec-inf} = \mu(\text{spec-inf}, <s-fl : s-format-list\cdot s-spec(t)>)
\]

\[
\text{int-format-head}(\text{rep}, \text{elem}, \text{tail}) =
\]

\[
\text{le}(\text{rep}, 0) \mapsto \text{RETURNS} : \text{tail}
\]

\[
T \mapsto \text{RETURNS} : (\text{Cat elem}) \cup \text{tail}
\]

\[
\text{int-expr-op}(\text{expr}) =
\]

\[
\text{return-op}(\text{val});
\]

\[
\text{val} : \text{int-expr}(\text{expr}, SC-EI)
\]

\[
\text{return-op}(\text{val}) = \text{RETURNS} : \text{s-op}(\text{val})
\]

Let \( hfi = \text{head}(s-fl(\text{spec-inf})) \)

\[
\text{int-elem-inp}(t, \text{gen}, ei, \text{spec-inf}) =
\]

\[
\text{is-string-get}(t) \mapsto \text{int-item}(t, \text{ret-string-inf}, \text{gen}, ei,
\text{spec-inf});
\]

\[
\text{ret-string-inf} : \text{get-string}(s-string(\text{spec-inf}), hfi)
\]

\[
\text{is-stream-get}(t) \mapsto
\]

\[
\text{int-stream-item}(t, \text{ret-stream-inf}, \text{gen}, ei, \text{spec-inf});
\]

\[
\text{ret-stream-inf} : \text{get-stream}(s-file-opt(t), hfi)
\]

\[
\text{int-stream-item}(t, \text{stream-inf}, \text{gen}, ei, \text{spec-inf}) =
\]

\[
\text{null};
\]

\[
\text{int-item}(t, \text{stream-inf}, \text{gen}, ei, \text{spec-inf}),
\text{update-stream}(t, \text{stream-inf}, \text{spec-inf})
\]
Let
\[ tfl = \text{tail}(s-fl(spec-inf)) \]
\[ hfi = \text{head}(s-fl(spec-inf)) \]
\[ \text{length} = \text{length}(\text{s-concr-rep}(\text{inp-inf})) \]
\[ \text{data} = \text{s-data}(\text{inp-inf}) \]

\[ \text{int-item}(t,\text{inp-inf},\text{gen},\text{ei},\text{spec-inf}) = \]
\[ \text{is-string-end}(\text{inp-inf}) \lor \text{is-end-file}(\text{inp-inf}) \rightarrow \]
\[ \text{RETURNS} : \mu(\text{spec-inf}, <s-end:EMPTY>) \]

\[ \text{is-inv-data}(\text{data}) \rightarrow \]
\[ \text{int-list-edit-gen}(t,\text{gen},\text{update}(\text{ei}),\text{ret-spec-inf}); \]
\[ \text{raise-error-condition} \]
\[ \text{is-space}(s\text{-format-type}(hfi)) \rightarrow \]
\[ \text{int-list-edit-gen}(t,\text{gen},\text{ei},\text{tail-spec-inf}) \]
\[ \text{lt}(\text{length},0) \rightarrow \text{int-list-edit-gen}(t,\text{gen},\text{update}(\text{ei}),\text{tail-spec-inf}) \]
\[ \text{eg}(\text{length},0) \land \text{is-string-format}(hfi) \rightarrow \]
\[ \text{int-list-edit-gen}(t,\text{gen},\text{update}(\text{ei}),\text{ret-spec-inf}); \]
\[ \text{assign}(\text{sub-gen}(\text{gen},s\_el(\text{ei})), \text{NULLSTRING}) \]
\[ \text{is-concr-data}(\text{data}) \rightarrow \]
\[ \text{int-list-edit-gen}(t,\text{gen},\text{update}(\text{ei}),\text{ret-spec-inf}); \]
\[ \text{assign}(\text{sub-gen}(\text{gen},s\_el(\text{ei})), s\_op(\text{val})); \]
\[ \text{val} : \text{int-exp}(\text{abstr-data},\text{SC-EI}); \]
\[ \text{abstr-data} : \text{translate-data}(\text{data}) \]
\[ \text{is-concr-comma}(\text{data}) \land (\exists(s\_first(\text{spec-inf})) \lor \exists(s\_comma(\text{spec-inf}))) \rightarrow \]
\[ \text{int-list-edit-gen}(t,\text{gen},\text{update}(\text{ei}),\text{ret-spec-inf}) \]
\[ \text{is-concr-comma}(\text{data}) \rightarrow \]
\[ \text{int-list-edit-gen}(t,\text{gen},\text{ei},\text{comma-spec-inf}) \]
where

\[\text{tail-spec-inf} = \mu(\text{spec-inf}, \{<\text{s-string: }\text{s-rest-string}(\text{inp-inf})>,
\text{s-fl: tfl}\})\]

\[\text{ret-spec-inf} = \delta(\delta(\text{tail-spec-inf}, \text{s-first}), \text{s-conr-comma})\]

\[\text{comma-spec-inf} = \mu(\text{spec-inf}, \{<\text{s-string: }
\text{s-rest-string}(\text{inp-inf})>,
\text{s-first: }\emptyset>,
\text{s-concr-comma: }\text{EMPTY}\})\]
6.3.2 The put-statement

6.3.21 Elementary output functions and instructions

The definition of source items is the basis for the definition of elementary input instructions in 6.3.11. In the case of output there is no problem of recognizing parts of data in the stream or in a string. As a consequence the structuring of these parts (which could be called target items by analogy) is of minor interest.

The elementary output function make-stream-rep(elem-inf,type,format,print) has as its value a list whose elements conform to the predicate is-char-datum. The list is the result of the transformation of a scalar part of data, i.e. a scalar part of an expression or of a variable (identified by the argument elem-inf). The result depends on the type of the put-statement (data-, list-, or edit-directed type, identified by the second argument), and in the case of edit-directed type on the associated evaluated format-item (third argument). The fourth argument "print" is true or false whether the result is for print or nonprint output. If the scalar part is a variable and the type is not edit-directed, the result may also depend on the attributes of the scalar part (so-called numeric field data).

Whether output is to the stream or to a string does not affect the result. However, for a string nonprint output applies only.

The elementary output functions make-string-end-rep(type) and make-stream-end-rep(type) have values of the same type as the above elementary output function but the list is not the result of a transformation of a scalar part of data.

Whenever output is to a string, the elementary output instruction put-string(pair,rep) transmits rep which is make-stream-rep(elem-inf,type,format,F) or make-string-end-rep(type) to the string which is identified by the first argument pair contain-
ing the generation and expansion information of the string. The in-
struction returns the new expansion information and generation of the
string valid after placing the target item in the string.

The elementary output functions and the instruction put-string( )
are not yet defined formally.

Whenever output is to the stream, the instruction
put-stream(t,spec-inf,rep) transmits rep which is
make-stream-rep(elem-inf,type,format,print) or
make-stream-end-rep(type) to the stream; spec-inf is the specification-
information described in 6.3.12 (except that s-comma is not used), and
t conforms to the predicate is-stream-put.

6.3.22 Notes to the interpretation

Target items are identical with the values of the elementary out-
put functions for nonprint output only; for print output target items
are expanded by the instruction expand-rep( ). Changes in the language
transmitted by a TNL to SRL-3 would require that the elementary out-
put functions be instructions.

The instruction int-data-list( ) defined in 6.3.14 is used in the
interpretation of the put-statement.

Some other comments which apply to the interpretation of the put-
statement too are contained in 6.3.13.
6.3.23 Interpretation of the put-statement

Abstract Syntax:

\[
\text{is-put-st} = (\text{is-stream-put} \mid \text{is-string-put})
\]

\[
\text{is-stream-put} = (\langle \text{s-st:is-put} \rangle, \langle \text{s-file-opt:is-name} \rangle, \langle \text{s-spec:is-put-spec} \rangle, \langle \text{s-page-opt:is-empty} \mid \text{is-Ω} \rangle, \langle \text{s-line-opt:is-expr} \mid \text{is-Ω} \rangle, \langle \text{s-skip-opt:is-expr} \mid \text{is-Ω} \rangle)
\]

\[
\text{is-string-put} = (\langle \text{s-st:is-put} \rangle, \langle \text{s-string-opt:is-ref} \rangle, \langle \text{s-spec:is-put-spec} \rangle)
\]

\[
\text{is-put-spec} = (\langle \text{s-type:is-data-dir} \rangle, \langle \text{s-data-list:is-ref-elem-list} \rangle \mid (\langle \text{s-type:is-edit-dir} \rangle, \langle \text{s-data-list:is-expr-elem-list} \rangle, \langle \text{s-data-list:is-expr-elem-list} \rangle, \langle \text{s-data-list:is-expr-elem-list} \rangle) \mid (\langle \text{s-type:is-list-dir} \rangle, \langle \text{s-data-list:is-expr-elem-list} \rangle))
\]

\[
\text{is-expr-elem} = (\text{is-expr} \mid \text{is-expr-contr})
\]

\[
\text{is-expr-contr} = (\langle \text{s-contr-var:is-ref} \rangle, \langle \text{s-spec-list:is-do-spec-list} \rangle, \langle \text{s-data-list:is-expr-elem-list} \rangle)
\]

1) SYSPRINT inserted by default by pretranslator

2) \(\text{is-stream-put(t)} \& \text{exists}(\text{s-skip-opt(t)}) \Rightarrow \neg \text{exists}(\text{s-page-opt(t)}) \& \neg \text{exists}(\text{s-line-opt(t)})\)

3) if the concrete skip-option implies the constant "1", the constant is inserted by the pretranslator
Definition:

\begin{align*}
\text{int-put-st}(t) &= \text{is-stream-put}(t) \rightarrow \\
&\quad \text{int-end-stream}(t); \\
&\quad \text{int-put-data-spec}(t, \text{stream-spec-inf}); \\
&\quad \text{int-print-options}(t, T); \\
&\quad \text{int-open-st}(\text{tout})
\end{align*}

\begin{align*}
\text{is-string-put}(t) &= \rightarrow \\
&\quad \text{int-end-string}(t, \text{ret-spec-inf}); \\
&\quad \text{ret-spec-inf} : \text{int-put-data-spec}(t, \text{string-spec-inf}); \\
&\quad \text{gen} : \text{int-variable}(\text{s-string-opt}(t), \emptyset)
\end{align*}

where

\begin{align*}
\text{tout} &= \mu_0(\langle \text{s-file-opt:s-file-opt}(t)\rangle, \\
&\quad \langle \text{s-open-attr:}\{\text{STREAM,OUTPUT}\}\rangle) \\
\text{stream-spec-inf} &= \mu_0(\langle \text{s-first:EMPTY}\rangle, \\
&\quad \langle \text{s-fl:s-format-list.s-spec}(t)\rangle) \\
\text{string-spec-inf} &= \mu(\text{stream-spec-inf}, \\
&\quad \langle \text{s-string:<gen,int-ei(s-da(gen))>}\rangle)
\end{align*}

Let

\begin{align*}
\text{page} &= \text{s-page-opt}(t) \\
\text{line} &= \text{s-line-opt}(t) \\
\text{skip} &= \text{s-skip-opt}(t)
\end{align*}
\[\text{int-print-options}(t, \text{first}) =\]

\[\text{is-expr}(\text{skip}) \quad \longrightarrow \quad \text{int-print}(t, \text{SKIP}, \text{convert}(\text{INTEGER}, \text{op})); \]
\[\quad \text{op} : \text{int-expr-op}(\text{skip})\]

\[\text{first} \&\text{ is-empty}(\text{page}) \quad \longrightarrow \quad \text{null};\]
\[\quad \text{int-print-options}(t, F), \]
\[\quad \text{int-print}(t, \text{PAGE}, \text{page})\]

\[\text{is-expr}(\text{line}) \quad \longrightarrow \quad \text{int-print}(t, \text{LINE}, \text{convert}(\text{INTEGER}, \text{op})); \]
\[\quad \text{op} : \text{int-expr-op}(\text{line})\]

\[T \quad \longrightarrow \quad \text{null}\]

\[\text{int-print}(t, \text{idem}, \text{op}) = \text{int-printing-format}(t, \text{idem}, \text{op})\]

\[\text{int-end-stream}(t) =\]
\[\quad \text{put-stream}(t, \text{EMPTY}, \text{make-stream-end-rep}(\text{s-type}\cdot\text{s-spec}(t)))\]

\[\text{int-end-string}(t, \text{spec-inf}) =\]
\[\quad \text{exists}(\text{s-end}(\text{spec-inf})) \quad \longrightarrow \quad \text{raise-error-cond}\]
\[\quad T \quad \longrightarrow \quad \text{put-string}(\text{s-string}(\text{spec-inf}),\]
\[\quad \quad \text{make-string-end-rep}(\text{s-type}\cdot\text{s-spec}(t)))\]

\[\text{int-put-data-spec}(t, \text{spec-inf}) =\]
\[\quad \text{int-data-list}(t, \text{s-data-list}\cdot\text{s-spec}(t), \text{spec-inf})\]

\[\text{int-put-elem}(t, \text{elem}, \text{spec-inf}) =\]
\[\quad \text{is-ref}(\text{elem}) \quad \longrightarrow \quad \text{int-data-list-edit}(t, \text{ref-inf}, \text{spec-inf});\]
\[\quad \text{gen} : \text{int-variable}(\text{elem}, E)\]
\[\quad T \quad \longrightarrow \quad \text{int-data-list-edit}(t, \text{expr-inf}, \text{spec-inf});\]
\[\quad \quad \text{aggr-list} : \text{collect-aggr}(\text{elem}, E)\]
where
\[\quad \text{ref-inf} = \mu_0(\langle \text{s-elem}:\text{elem}\rangle, \langle \text{s-gen}:\text{gen}\rangle, \]
\[\quad \langle \text{s-ei:}\text{init-ei}(\text{s-da}(\text{gen}))\rangle)\]
\[\quad \text{expr-inf} = \mu_0(\langle \text{s-elem}:\text{elem}\rangle, \langle \text{s-ag:aggr-list}\rangle)\]
Let $hfe = \text{head}(s-fl(spec-inf))$
$\text{tfe} = \text{tail}(s-fl(spec-inf))$
$\text{type} = s\text{-type}\ast s\text{-spec}(t)$

\begin{align*}
\text{int-data-list-edit}(t, elem-inf, spec-inf) &= \\
\text{is-ref}(s-elem(elem-inf)) &\text{& is-exhaust}(\text{s-ei}(elem-inf)) \rightarrow \\
&\text{RETURNS : spec-inf} \\
\sim\text{is-ref}(s-elem(elem-inf)) &\text{& is-emptylist}(s-ag(elem-inf)) \rightarrow \\
&\text{RETURNS : spec-inf} \\
\text{is-string-put}(t) &\text{& is-exhaust}(\text{elem}(2, s-string(spec-inf))) \rightarrow \\
&\text{RETURNS : end-spec-inf} \\
\sim\text{is-edit-dir}(\text{type}) \\
\text{int-out}(t, elem-inf, spec-inf, type, EMPTY) \\
\text{is-emptylist}(s-fl(spec-inf)) \\
\text{int-data-list-edit}(t, elem-inf, format-spec-inf) \\
\sim\text{is-format-item}(hfe) \\
\text{int-data-list-edit}(t, elem-inf, \\
\quad\mu(\text{spec-inf, } \langle s-fl: \text{ret-fl} \rangle)); \\
\text{ret-fl : int-format-head(\text{convert}(\text{INTEGER}, \\
\quad\text{s-op(rep), s-element(hfe), \text{tfe}); \\
\text{rep : int-expr(s-rep-factor(hfe),} \\
\quad\text{SC-EI})} \\
\text{is-remote-format}(hfe) \\
\text{int-data-list-edit}(t, elem-inf, \\
\quad\mu(\text{spec-inf, } \langle s-fl: s-\text{vr*op(val)*tfe} \rangle)); \\
\text{val : int-ref(s-label-ref(hfe))} \\
T \rightarrow \text{int-format}(t, elem-inf, spec-inf, val-hfi); \\
\{ \text{\%val-hfi : int-expr-op(\%hfe)} \mid \\
\text{is-expr(\%hfe)} \}\}
\end{align*}

where $\text{end-spec-inf} = \mu(\text{spec-inf, <s-end:EMPTY>})$
$\text{format-spec-inf} = \mu($ spec-inf,
\quad\langle s-fl:s\text{-format-list}\ast s\text{-spec}(t) \rangle)$
\[ \text{int-format}(t, \text{elem-inf}, \text{spec-inf}, \text{hfi}) = \]
\[ \text{is-space}(s\text{-format-type}(\text{hfi})) \land \text{gt}(s-w(\text{hfi}), 0) \rightarrow \]
\[ \text{int-space-format}(t, \text{elem-inf}, \text{format-tail-spec-inf}, \]
\[ \text{make-blanks} \left( \text{convert}(\text{INTEGER}, s-w(\text{hfi})) \right) \]
\[ \text{is-space}(s\text{-format-type}(\text{hfi})) \rightarrow \]
\[ \text{int-data-list-edit}(t, \text{elem-inf}, \text{format-tail-spec-inf}) \]
\[ \text{is-control-format}(\text{hfi}) \rightarrow \]
\[ \text{int-data-list-edit}(t, \text{elem-inf}, \text{format-tail-spec-inf}); \]
\[ \text{int-printing-format}(t, s\text{-format-type}(\text{hfi}), \]
\[ \text{convert}(\text{INTEGER}, s-w(\text{hfi})) \]
\[ \text{is-data-format}(\text{hfi}) \rightarrow \]
\[ \text{int-out}(t, \text{elem-inf}, \text{format-tail-spec-inf}, \]
\[ \text{s-type}\cdot s\text{-spec}(t), \text{hfi} \]

where \[
\text{format-tail-spec-inf} = \mu(\text{spec-inf}, \langle s\text{-fl:tail}(s\text{-fl(\text{spec-inf}))} \rangle) \]

and \[
\text{make-blanks}(1) = \mu_0(\{ \langle \text{elem}(i): \text{BLANK} \rangle \mid 1 \leq i \leq 1 \}) \]

where \[
\text{is-char-datum}(\text{BLANK}) \]

Let \[
\text{fid} = s\text{-file-opt}(t) \]

\[ \text{int-out}(t, \text{elem-inf}, \text{spec-inf}, \text{type}, \text{format}) = \]
\[ \text{is-string-put}(t) \rightarrow \]
\[ \text{int-string-put}(t, \text{elem-inf}, \text{spec-inf}, \text{pair}); \]
\[ \text{pair : put-string}(s\text{-string}(\text{spec-inf}), \]
\[ \text{make-stream-rep}(\text{elem-inf}, \text{type}, \text{format}, F); \]

\[ T \rightarrow \text{int-stream-put}(t, \text{elem-inf}, \text{spec-inf}); \]
\[ \text{put-stream}(t, \text{spec-inf}, \]
\[ \text{make-stream-rep}(\text{elem-inf}, \text{type}, \text{format}, \]
\[ \text{is-print}(\text{csa}(\text{fid}))) \]
\[ \text{int-string-put}(t, \text{elem-inf}, \text{spec-inf}, \text{pair}) = \]
\[
is-\text{ref}(s-\text{elem}(\text{elem-inf})) \rightarrow \]
\[
\text{int-data-list-edit}(t, \text{ref-elem}, \text{string}) \]
\[
T \rightarrow \text{int-data-list-edit}(t, \text{expr-elem}, \text{string})
\]

where
\[
\text{ref-elem} = \mu(\text{elem-inf}, \langle s-\text{ei}:\text{update}(s-\text{ei}(\text{elem-inf})) \rangle)
\]
\[
\text{string} = \mu(\text{spec-inf}, \langle s-\text{string}:\text{pair}, \langle s-\text{first}:\emptyset \rangle \rangle)
\]
\[
\text{expr-elem} = \mu(\text{elem-inf}, \langle s-\text{ag}:\text{tail}(s-\text{ag}(\text{elem-inf})) \rangle)
\]

\[ \text{int-stream-put}(t, \text{elem-inf}, \text{spec-inf}) = \]
\[
is-\text{ref}(s-\text{elem}(\text{elem-inf})) \rightarrow \]
\[
\text{int-data-list-edit}(t, \text{ref-elem}, \text{stream}) \]
\[
T \rightarrow \text{int-data-list-edit}(t, \text{expr-elem}, \text{stream})
\]

where
\[
\text{stream} = \delta(\text{spec-inf}, s-\text{first})
\]
\[
\text{ref-elem} \} \text{ see above}
\]
\[
\text{expr-elem} \}
\]

\[ \text{int-space-format}(t, \text{elem-inf}, \text{spec-inf}, \text{blanks}) = \]
\[
is-\text{string-put}(t) \rightarrow \]
\[
\text{int-data-list-edit}(t, \text{elem-inf}, \text{string});
\]
\[
\text{pair} : \text{put-string}(s-\text{string}(\text{spec-inf}), \text{blanks})
\]
\[
T \rightarrow \text{int-data-list-edit}(t, \text{elem-inf}, \text{spec-inf});
\]
\[
\text{put-stream}(t, \text{spec-inf}, \text{blanks})
\]

where
\[
\text{string} = \mu(\text{spec-inf}, \langle s-\text{string}:\text{pair} \rangle)
\]
Let \( \text{fid} = s\text{-file-opt}(t) \)

\[
\text{put-stream}(t, \text{spec-inf}, \text{rep}) = \begin{cases} 
\text{null} ; & \\
\{ \text{update-count}(t, \text{spec-inf}), \\
\text{update-es-part}((\text{next-sel})\text{(fid)}, \text{op}); \\
\text{op} : \text{int-infix-op}((\text{next})(\text{fid}), \text{length}(\text{rep}), \text{ADD}), \\
\text{update-data-elem-list}(t, \text{exp-rep}); \\
\text{exp-rep} : \text{expand-rep}(t, \text{rep}) \} 
\end{cases}
\]

Let \( \text{fid} = s\text{-file-opt}(t) \)

\[
\text{type} = s\text{-type-s-spec}(t) \]

\[
\text{expand-rep}(t, \text{rep}) = \begin{cases} 
\text{is-print}((\text{csa}(\text{fid}))) & \text{RETURNS : rep} \\
\text{is-edit-dir}(\text{type}) & \text{expand-direct}(t, \text{rep}, < >) \\
\text{T} & \text{int-next-tab}(t, \text{exp-rep}, \text{TABLIST}); \\
\text{exp-rep} & \text{expand-direct}(t, \text{rep}, < >); \\
\text{int-init-tab}(t, \text{op}_1, \text{op}_2); \\
\{ \text{op}_1 : \text{int-infix-op}((\text{curr-char})(\text{fid}), \text{length}(\text{rep}), \text{ADD}), \\
\text{op}_2 : \text{int-infix-op}((\text{pagesize})(\text{fid}), 1, \text{ADD}) \} 
\end{cases}
\]

Let \( \text{fid} = s\text{-file-opt}(t) \)

\[
\text{update-data-elem-list}(t, \text{rep}) = \begin{cases} 
\text{ES} = \mu(\text{ES}, \{ <\text{data-elem-sel}(\text{fid}, i): \text{elem}(i, \text{rep})> \mid 1 \leq i \leq \text{length}(\text{rep}) \}) 
\end{cases}
\]

1) \text{TABLIST} is a list of implementation-defined tabulators proposed in a TNL to SRL-3.

2) may cause \text{raise-transmit-cond}(t) to be executed
Let $\text{fid} = \text{s-file-opt}(t)$
\[\text{type} = \text{s-type} \cdot \text{s-spec}(t)\]

\[\text{update-count}(t, \text{spec-inf}) =\]
\[\exists (\text{s-first}(\text{spec-inf}))\land \text{is-data-dir}(\text{type}) \rightarrow \text{int-count}(\text{fid}, 0)\]
\[\text{is-data-dir}(\text{type}) \rightarrow \text{int-count}(\text{fid}, \text{count}(\text{fid}))\]
\[T \rightarrow \text{null}\]

Let $\text{fid} = \text{s-file-opt}(t)$

\[\text{expand-direct}(t, \text{rep}, \text{list}) =\]
\[\text{is-emptylist}(\text{rep}) \rightarrow \text{RETURNS} : \text{list}\]
\[\text{le}(\text{curr-char}(\text{fid}), \text{linesize}(\text{fid})) \rightarrow \]
\[\text{expand-direct}(t, \text{tail}(\text{rep}), \text{list} \cdot \text{elem});\]
\[\text{update-es-part}(\text{curr-char-sel}(\text{fid}), \text{op});\]
\[\text{op} : \text{int-infix-op}(\text{curr-char}(\text{fid}), 1, \text{ADD})\]
\[T \rightarrow \text{expand-direct}(t, \text{rep}, \text{list});\]
\[\text{int-printing-format}(t, \text{SKIP}, 1)\]

where $\text{elem} = \mu_0(<\text{s-page}: \text{curr-page}(\text{fid})>,$
\[<\text{s-line}: \text{curr-line}(\text{fid})>,\]
\[<\text{s-char}: \text{curr-char}(\text{fid})>,\]
\[<\text{s-character}: \text{head}(\text{rep})>)\]

\[\text{int-init-tab}(t, \text{op}_1, \text{op}_2) =\]
\[\text{le}(\text{op}_1, \text{op}_2) \rightarrow \text{null}\]
\[T \rightarrow \text{int-printing-format}(t, \text{SKIP}, 1)\]
\[
\text{int-next-tab}(t, \text{exp-rep}, \text{tablist}) = \\
\text{is-emptylist}(\text{tablist}) \lor \neg \text{exists}(\text{tablist}) \quad \text{RETURNS : exp-rep} \\
\text{(is-emptylist}(\text{tail}(\text{tablist})) \land \\
\text{gt}(\text{curr-char}(\text{fid}), \text{head}(\text{tablist}))) \lor \\
\text{gt}(\text{head}(\text{tablist}), \text{linesize}(\text{fid})) \quad \text{int-next-tab}(t, \text{exp-rep}, \text{TABLIST}); \\
\text{int-printing-format}(t, \text{SKIP}, 1); \quad \\
\text{int-printing-format}(t, \text{COLUMN}, 1) \\
\text{le}(\text{curr-char}(\text{fid}), \text{head}(\text{tablist})) \quad \text{ret-exp-rep}(\text{exp-rep}); \\
\text{update-es-part}(\text{curr-char-sel}(\text{fid}), \text{head}(\text{tablist})) \\
T \quad \text{int-next-tab}(t, \text{exp-rep}, \text{tail}(\text{tablist}))
\]

where \( \text{fid} = \text{s-file-opt}(t) \)

\[
\text{ret-exp-rep}(\text{exp-rep}) = \text{RETURNS : exp-rep}
\]

\[
\text{copy}(\text{rep}) = \text{put-stream}(t, \text{EMPTY}, \text{rep})
\]

where \( t = \mu_0(\langle \text{s-st:PUT}, \text{s-file-opt:SYSPRINT}, \langle \text{s-spec:mu_0(\langle \text{s-type:EDIT} >) \rangle} \rangle)
\)

where \( \text{is-put}(\text{PUT}) \)

and \( \text{is-edit-dir}(\text{EDIT}) \) are true.
Let \( \text{fid} = s\text{-file-opt}(t) \)

\[
\text{int-printing-format}(t, \text{ident}, \text{op}) = \\
\text{is-print}(\text{csa}(\text{fid})) \rightarrow \text{raise-error-cond} \\
\text{is-page}(\text{ident}) \rightarrow \\
\text{null}; \\
\{ \text{update-es-part}(\text{curr-page-sel}(\text{fid}), \text{op}_1); \\
\text{op}_1 : \text{int-infix-op}(\text{curr-page}(\text{fid}), 1, \text{ADD}), \\
\text{update-es-part}(\text{curr-line-sel}(\text{fid}), 1), \\
\text{update-es-part}(\text{end-page-sel}(\text{fid}), \Omega) \} \\
\text{is-line}(\text{ident}) \& \text{le}(\text{op}, 0) \rightarrow \text{int-printing-format}(t, \text{ident}, 1) \\
\text{is-line}(\text{ident}) \& (\text{le}(\text{op}, \text{curr-line}(\text{fid})) \lor \\
\text{gt}(\text{op}, \text{pagesize}(\text{fid}))) \rightarrow \text{int-endpage}(t) \\
\text{is-line}(\text{ident}) \rightarrow \\
\text{update-es-part}(\text{curr-line-sel}(\text{fid}), \text{op}) \\
\text{is-skip}(\text{ident}) \& \text{le}(\text{op}, 0). \rightarrow \\
\text{update-es-part}(\text{curr-char-sel}(\text{fid}), 1) \\
\text{is-skip}(\text{ident}) \rightarrow \text{int-skip-format}(t, \text{op}, \text{op}_1); \\
\text{op}_1 : \text{int-infix-op}(\text{curr-line}(\text{fid}), \text{op}, \text{ADD}) \\
\text{is-column}(\text{ident}) \& (\text{le}(\text{op}, 0) \lor \text{gt}(\text{op}, \text{linesize}(\text{fid}))) \rightarrow \\
\text{int-printing-format}(t, \text{ident}, 1) \\
\text{is-column}(\text{ident}) \& \text{le}(\text{op}, \text{curr-char}(\text{fid})). \rightarrow \\
\text{null}; \\
\{ \text{update-es-part}(\text{curr-line-sel}(\text{fid}), \text{op}_1); \\
\text{op}_1 : \text{int-infix-op}(\text{curr-line}(\text{fid}), 1, \text{ADD}), \\
\text{update-es-part}(\text{curr-char-sel}(\text{fid}), \text{op}) \} \\
\text{is-column}(\text{ident}) \rightarrow \\
\text{update-es-part}(\text{curr-char-sel}(\text{fid}), \text{op})
Let \( \text{fid} = \text{s-file-opt}(t) \)

\[
\text{int-skip-format}(t, \text{op}, \text{new-line}) =
\begin{align*}
\text{gt}(\text{new-line}, \text{pagesize}(\text{fid})) & \quad \rightarrow \quad \text{int-endpage}(t) \\
\text{T} & \quad \rightarrow \quad \text{update-es-part}(\text{curr-line-sel}(\text{fid}), \text{new-line})
\end{align*}
\]

Let \( \text{fid} = \text{s-file-opt}(t) \)

\[
\text{int-endpage}(t) =
\begin{align*}
\text{exists}(\text{end-page}(\text{fid})) & \quad \rightarrow \quad \text{int-printing-format}(t, \text{PAGE}, \text{EMPTY}) \\
\text{T} & \quad \rightarrow \quad \text{raise-endpage-cond}(t); \\
& \quad \begin{cases} 
\text{update-es-part}(\text{curr-line-sel}(\text{fid}), \text{op}); \\
\quad \text{op} : \text{int-infix-op}(\text{pagesize}(\text{fid}), 1, \text{ADD}), \\
\text{update-es-part}(\text{end-page-sel}(\text{fid}), \text{EMPTY})
\end{cases}
\end{align*}
\]
6.4 Record Transmission

In this chapter no interpretation is given of the EVENT-option in the READ, WRITE, REWRITE and DELETE statement. Also the correct action of the KEY condition, i.e. skipping the current statement, is not shown.

In addition to the abbreviations in chap. 6.1 the following is a convention for this chapter:

\[ \text{fid} = \text{s-file-opt(t)} \]
6.4.1 Sequential and direct access to record data sets

Direct as well as sequential data sets are represented in ES as lists. This may not show explicitly the real situation for direct data sets, however, sequential keyed data sets have two distinct possibilities of access and the sequential access must be adjustable by the direct access. Since one structure of the data set only can be shown, a sequential one in the form of a list has been chosen. "next(fid)" contains the current index to the list.

**Definition:**

\[
\text{sequential-access}(t, \text{key}) = \text{pass}(\text{next}(\text{fid}));
\]

\[
\text{check-multiple-key}(t, \text{key}) =
\]

\[
\text{is-keyed}(\text{csa}(\text{fid})) & (\exists m)(\text{key}(\text{fid}, m) = \text{key}) \rightarrow
\]

\[
\text{raise-key-cond}(t)
\]

\[
T \rightarrow \text{null}
\]

The direct access is defined by comparison of the key and referencing the index.

**Definition:**

\[
\text{keyed-access}(t, \text{key}) = \text{pass}((tm)(\text{key}(\text{fid}, m) = \text{key}));
\]

\[
\text{check-no-key}(t, \text{key}) =
\]

\[
\neg((\exists m)(\text{key}(\text{fid}, m) = \text{key})) \rightarrow
\]

\[
\text{raise-key-cond}(t)
\]

\[
T \rightarrow \text{null}
\]

Note that consequently DELETE statements cause undefined elements within the list representing the data set.
After a sequential access to a data set the index must be updated by ±1 with respect to the BACKWARDS option. The same update instruction is also used to interpret the IGNORE option.

**Definition:**

\[
\text{update-next}(t, \text{key}, \text{ign}) = \]
\[
is\text{-backwards}(\text{csa}(\text{fid})) \rightarrow \text{next-backwards}(t, m, \text{ign});
\]
\[
T \rightarrow \text{next}(t, m, \text{ign});
\]
\[
m : \text{sequential-access}(t, \text{key})
\]

\[
\text{next-backwards}(t, m, \text{ign}) =
\]
\[
\text{seq} : \mu(\text{ES}, \langle \text{next-sel}(\text{fid}) : m - \text{ign} \rangle)
\]
\[
\text{next}(t, m, \text{ign}) =
\]
\[
\text{seq} : \mu(\text{ES}, \langle \text{next-sel}(\text{fid}) : m + \text{ign} \rangle)
\]

For sequential keyed data sets which are accessed directly the updating is done as follows:

**Definition:**

\[
\text{new-next}(t, \text{key}) = \]
\[
is\text{-backwards}(\text{csa}(\text{fid})) \rightarrow \text{next-backwards}(t, m, \text{ign});
\]
\[
T \rightarrow \text{next}(t, m, \text{ign});
\]
\[
m : \text{keyed-access}(t, \text{key})
\]

Note, that there is no initialisation of the index in \text{next}(\text{fid}) defined.
5.4.2 Unbuffered record transmission

The kind of data to be transmitted as records is restricted:

\[
\text{is-record-ref(ref)} = \\
\text{is-simple-name(ref)} \& \\
\neg \exists \forall \exists \text{s-var \& s-attr((ref\_E)\_AT)} \& \\
\text{is-proper-var\_s\_attr((ref\_E)\_AT)}
\]

Two basic functions are introduced for record transmission:

\[
S = \text{record-assign(S,gen,rec)}
\]

"record-assign" yields the storage with a record rec assigned to a generation gen.

\[
\text{record} = \text{from(S,gen)}
\]

"from" yields a record from the generation gen in the storage S.

The following condition is true:

\[
S = \text{record-assign(S,gen,from(S,gen))}
\]

With these basic functions instructions can be defined:

**Definition:**

\[
\text{record-assign(gen,rec)} =
\]

\[
\text{s-s : record-assign(S,gen,rec)}
\]

\[
\text{from(gen)} =
\]

\[
\text{RETURNS : from(S,gen)}
\]

Similarly the instruction \text{k-from(gen)} should return a key, i.e. a character string, from the generation.
record-assign causes the execution of raise-record-cond(t) if the
size of the record is not equal to the size of variable. Note, that
in the case of inequality of sizes the operation is not defined (loss
of excess data, no transmission, no alteration).

With the two basic instructions and the instructions store and
store-k which take a record or a key, respectively, to a data set,
six further instructions for record transmission to and from sequential
and direct data sets are defined.

Definition:

\[
\text{store}(t, m, \text{rec}) = \\
\quad \text{s-cs : } \mu(\text{ES}, \text{datum-sel}(f(m), \text{rec})) \\
\text{store-k}(t, m, \text{key}) = \\
\quad \text{s-cs : } \mu(\text{ES}, \text{key-sel}(f(m), \text{key})) \\
\text{from-next}(t, \text{key}) = \text{store}(t, m, \text{rec}); \\
\quad \{ m : \text{sequential-access}(t, \text{key}), \\
\quad \text{rec : from}(\text{gen}); \\
\quad \text{gen : int-variable}(s-\text{from-opt}(t), E) \} \\
\text{from-direct}(t, \text{key}) = \text{store}(t, m, \text{rec}); \\
\quad \{ m : \text{keyed-access}(t, \text{key}), \\
\quad \text{rec : from}(\text{gen}); \\
\quad \text{gen : int-variable}(s-\text{from-opt}(t), E) \} \\
\text{keyfrom-next}(t, \text{keyfrom}) = \text{store-k}(t, m, \text{keyfrom}); \\
\quad \{ m : \text{sequential-access}(t, \text{EMPTY}) \}
\]

1) May cause the execution of raise-transmit-cond(t).
Definition:

\[
\begin{align*}
\text{next-into}(t) &= \text{record-assign}(\text{gen}, \text{rec}) ; \\
&\{ \text{gen} : \text{int-variable}(s-\text{into-opt}(s-\text{destination}(t)), \text{E}), \\
&\text{rec} : \text{pass}(\text{datum}(\text{fid}, m)); \\
&m : \text{sequential-access}(t, \text{EMPTY}) \} \\
\text{dir-into}(t, \text{key}) &= \text{record-assign}(\text{gen}, \text{rec}) ; \\
&\{ \text{gen} : \text{int-variable}(s-\text{into-opt}(s-\text{destination}(t)), \text{E}), \\
&\text{rec} : \text{pass}(\text{datum}(\text{fid}, m)); \\
&m : \text{keyed-access}(t, \text{key}) \} \\
\text{next-keyto}(t) &= \text{assign}(\text{gen}, \text{key}) ; \\
&\{ \text{gen} : \text{int-variable}(s-\text{keyto-opt}(s-\text{key-keyto}(t)), \text{E}), \\
&\text{key} : \text{pass}(\text{key}(\text{fid}, m)); \\
&m : \text{sequential-access}(t, \text{EMPTY}) \} \\
\end{align*}
\]

1) May cause execution of \text{raise-record-cond}(t) or \text{raise-endfile-cond}(t) .
6.4.3 Buffered record transmission

For buffered record transmission the buffer is allocated anew with each READ, WRITE and LOCATE statement and freed at the end of a WRITE statement (also after the locate-execution) and before the READ statement. Since for allocation the data attributes have to be known, it is assumed that these can be gained for input from the record in the data set, while for output "da" is unambiguous. The generation of the record-buffer is stored in "bu(fid)" and that of the key-part in "kbu(fid)". No check is made if the data attributes of the record and the variable to which the record is assigned, are equal or do match.

The following definitions are arranged according to the statements where the instructions are used and also with respect to the cross-references (instructions involved are either defined before or follow immediately).

For WRITE:

\[
\text{from-bu}(t) = \\
\text{bu-assign}(t, \text{op}) ; \\
\text{bu-assign}(t, \text{op}) = \\
\text{assign}(\text{bu}(\text{fid}), \text{op}) ; \\
\text{m-bu}(t, \text{op})
\]

\[
\text{m-bu}(t, \text{op}) = \\
\text{p-bu}(a, b, s-\text{da}(\text{op}), s-\text{dens}(s-\text{da}(\text{op})), t) ; \\
a: \text{un-a} \\
b: \text{un-b}
\]
\[ p\text{-}bu(a,b,da,dens,t) = \]
\[
\text{null};
\]
\[
\text{allocate}(a,b,da,dens),
\]
\[
\text{null-allocate}(b),
\]
\[
m\text{-}bu\text{-}gen(da,dens,a,t)
\]
\[
m\text{-}bu\text{-}gen(da,dens,a,t) =
\]
\[
s\text{-}fu : \mu(FU, \langle bu\text{-}sel(fid) : m\text{-}gen(da,dens,a) \rangle)
\]
\[
\text{keyfrom}\text{-}kbu(t,\text{keyfrom}) =
\]
\[
\text{assign}(kbu(fid),\text{keyfrom});
\]
\[
m\text{-}kbu(t,\text{keyfrom})
\]
\[
m\text{-}kbu(t,\text{keyfrom}) =
\]
\[
p\text{-}kbu(a,b,\mu_0(\langle s\text{-}type:CHAR, s\text{-}length:length(s\text{-}da(keyfrom)) \rangle),
\]
\[
\text{PACKED},t);
\]
\[
a : \text{un}\text{-}a
\]
\[
b : \text{un}\text{-}b
\]
\[
p\text{-}kbu(a,b,da,dens,t) =
\]
\[
\text{null};
\]
\[
\text{allocate}(a,b,da,dens),
\]
\[
\text{null-allocate}(b),
\]
\[
m\text{-}kbu\text{-}gen(da,dens,a,t)
\]
\[
m\text{-}kbu\text{-}gen(da,dens,a,t) =
\]
\[
s\text{-}fu : \mu(FU, \langle kbu\text{-}sel(fid) : m\text{-}gen(da,dens,a) \rangle)
\]
\[
bu\text{-}next\text{-}f(t) =
\]
\[
\text{free\text{-}1}(bu(fid));
\]
\[
bu\text{-}next(t)
\]
\[
bu\text{-}next(t) =
\]
\[
\text{store}(t,m,rec);
\]
\[
m : \text{sequential-access}(t,EMPTY),
\]
\[
rec : \text{from}(bu(fid))
\]

Note: \( bu\text{-}next \) is also used in \text{REWRITE}. 
\[ \text{kbu\text{-}next-f}(t,\text{key}) = \]
\[ \text{null}; \]
\[ \text{free-1}(\text{bu}(\text{fid})); \]
\[ \text{store}(t,m,\text{rec}); \]
\[ m : \text{sequential\text{-}access}(t,\text{key}), \]
\[ \text{rec} : \text{from}(\text{bu}(\text{fid})) \]
\[ \text{free-1}(\text{kbu}(\text{fid})); \]
\[ \text{store-k}(t,m,\text{key}); \]
\[ m : \text{sequential\text{-}access}(t,\text{key}), \]
\[ \text{key} : \text{k\text{-}from}(\text{kbu}(\text{fid})) \]

For LOCATE:

\[ \text{int\text{-}set\text{-}locate}(t) = \]
\[ \text{m\text{-}based\text{-}allocate}(a,b,\text{ptr\text{-}gen},\text{da},\text{dens},t); \]
\[ a : \text{un\text{-}ab} \]
\[ b : \text{un\text{-}b} \]
\[ \text{ptr\text{-}gen} : \text{int\text{-}variable}(\text{s\text{-}set\text{-}opt}(t),E) \]
\[ \text{da} : \text{s\text{-}da\text{-}s\text{-}attr}((\text{s\text{-}variable}(t)\cdot E)(\text{AT}) \]
\[ \text{dens} : \text{s\text{-}dens\text{-}s\text{-}attr}((\text{s\text{-}variable}(t)\cdot E)(\text{AT}) \]
\[ \text{m\text{-}based\text{-}allocate}(a,b,\text{ptr\text{-}gen},\text{da},\text{dens},t) = \]
\[ \text{null}; \]
\[ \text{based\text{-}allocate}(a,b,\text{ptr\text{-}gen},\text{da},\text{dens}), \]
\[ \text{m\text{-}bu\text{-}gen}(\text{da},\text{dens},a,t) \]

For READ with INTO-option:

\[ \text{next-bu}(t) = \]
\[ \text{ds\text{-}bu\text{-}assign}(t,\text{op}); \]
\[ \text{op} : \text{pass}(\text{datum}(\text{fid},m)); \]
\[ m : \text{sequential\text{-}access}(t,\text{EMPTY}); \]
\[ \text{free-b}(t) \]
\[\begin{align*}
\text{ds-bu-assign}(t, \text{op}) &= \\
&= \text{record-assign}(\text{bu}(\text{fid}), \text{op}); \\
&= \text{m-bu}(t, \text{op})
\end{align*}\]

\[\begin{align*}
\text{free-b}(t) &= \\
&= \text{null}; \\
&= \text{free-bu}(t), \\
&= \text{free-kbu}(t)
\end{align*}\]

\[\begin{align*}
\text{free-bu}(t) &= \\
&= \text{exists}(\text{bu}(\text{fid})) \rightarrow \text{free-1}(\text{bu}(\text{fid})) \\
&= \text{T} \rightarrow \text{null}
\end{align*}\]

\[\begin{align*}
\text{free-kbu}(t) &= \\
&= \text{exists}(\text{kbu}(\text{fid})) \rightarrow \text{free-1}(\text{kbu}(\text{fid})) \\
&= \text{T} \rightarrow \text{null}
\end{align*}\]

\[\begin{align*}
\text{next-kbu}(t) &= \\
&= \text{kevfrom-kbu}(t, \text{op}); \\
&= \text{op} : \text{pass}(\text{key}(\text{fid}, \text{m})); \\
&= \text{m} : \text{sequential-access}(t, \text{EMPTY})
\end{align*}\]

\[\begin{align*}
\text{dir-bu}(t, \text{key}) &= \\
&= \text{ds-bu-assign}(t, \text{op}); \\
&= \text{op} : \text{pass}(\text{datum}(\text{fid}, \text{m})); \\
&= \text{m} : \text{keved-access}(t, \text{key}); \\
&= \text{free-b}(t)
\end{align*}\]

\[\begin{align*}
\text{bu-into}(t) &= \\
&= \text{assign}(\text{into}, \text{bu}); \\
&= \text{into} : \text{int-variable}(\text{s-into-opt}(t), \text{E}) \\
&= \text{bu} : \text{from}(\text{bu}(\text{fid}))
\end{align*}\]
\begin{verbatim}

kbu-keyto(t) =
    assign(keyto, kbu);
    keyto : int-variable(s-keyto-opt(s-key-keyto(t)), E)
    kbu : k-from(kbu(fid))

For READ with SET option:

next-cbu(t) =
    ds-cbu-assign(t, op);
    op : pass(datum(fid, m));
    m : sequential-access(t, EMPTY); free-b(t)

ds-cbu-assign(t, op) =
    record-assgin(bu(fid), op);
    m-cbu(t, op)

m-cbu(t, op) =
    m-based-allocate(a, b, ptr-gen, da, dens, t);
    a : un-ab
    b : un-b
    ptr-gen : int-variable(s-set-opt(s-destination(t)), E)
    da : pass(s-da(op))
    dens : pass(s-dens(op))

dir-cbu(t, key) =
    ds-cbu-assign(t, op);
    op : pass(datum(fid, m));
    m : keyed-access(t, key);
    free-b(t)
\end{verbatim}
6.4.4 Locking and unlocking of records

In every file union directory element there exists the locking information. Under each unique task identifier the set of the keys of all records locked by that task is to be found. The predicate, that a record is locked is defined as follows:

\[
\text{is-locked}(fid, key) = \\
(\exists \text{ta-id})(\text{ta-id} \in \text{TA}) \& (\text{c-ta-id} \neq \text{ta-id}) \& \\
((\text{ta-id} \cdot \text{locking}(fid)) \ni \text{key})
\]

where TA is the set of all unique task identifiers and c-ta-id is the current task identifier; i.e. a record is locked, if under any task identifier, not equal to the current task identifier, the key is found.

The locking information is added and deleted from the file union directory elements by the following two instructions:

**Definition:**

\[
\text{int-lock}(t, key) = \\
\neg((\text{c-ta-id} \cdot \text{locking}(fid)) \ni \text{key}) \quad \rightarrow \quad \text{lock}(t, key) \\
\quad \quad \quad \text{T} \quad \rightarrow \quad \text{null} \\
\text{int-unlock}(t, key) = \\
((\text{c-ta-id} \cdot \text{locking}(fid)) \ni \text{key}) \quad \rightarrow \quad \text{unlock}(t, key) \\
\quad \quad \quad \text{T} \quad \rightarrow \quad \text{null}
\]

\[
\text{lock}(t, key) = \\
\text{s-fu} : \mu( \text{FU}, \langle \text{key} \cdot \text{c-ta-id} \cdot \text{locking-sel}(fid) : \text{key} \rangle ) \\
\text{unlock}(t, key) = \\
\text{s-fu} : \delta( \text{FU}, \text{key} \cdot \text{c-ta-id} \cdot \text{locking-sel}(fid))
\]
6.4.5 Interpretation of the expressions in key-, keyfrom-, ignore-options

**Definition:**

\[
\text{int-key}(k-opt) = \begin{array}{l}
\text{exists}(k-opt) \quad \rightarrow \quad \text{con-key}(r); \\
\quad \text{R} \quad \rightarrow \quad \text{Returns} : \text{EMPTY}
\end{array}
\]

\[
\text{CON-key}(r) = \begin{array}{l}
\text{Returns} : \text{convert}(\mu_0(<\text{base:CHAR},<\text{length:EMPTY}>,s-op(r)))
\end{array}
\]

\[
\text{int-ign(ign-opt)} = \begin{array}{l}
\text{exists(ign-opt)} \quad \rightarrow \quad \text{con-ign}(r); \\
\quad \text{R} \quad \rightarrow \quad \text{Returns} : 1
\end{array}
\]

\[
\text{CON-ign}(r) = \begin{array}{l}
\text{convert} \left( \text{INTG}, s-op(r) > 0 \rightarrow \right) \\
\quad \text{Returns} : \text{convert} \left( \text{INTG}, s-op(r) \right) + 1
\end{array}
\]
6.4.6 The Read-statement

Abstract Syntax:

\[
\text{is-read-st = <s-st :is-read>,}
\]
\[
<\text{s-file-opt:is-name>,}
\]
\[
<\text{s-destination: (is-into-opt | is-set-opt | is-∅)>,}
\]
\[
<\text{s-key-keyto: (is-key-opt | is-keyto-opt | is-∅)>,}
\]
\[
<\text{s-ignore-opt: (is-expr | is-∅)>,}
\]
\[
<\text{s-nolock-opt:is-opt>,}
\]
\[
<\text{s-event-opt: (is-ref | is-∅)>}
\]

\[
is-into-opt = (<\text{s-into-opt:is-level-one-ref}>)
\]
\[
is-set-opt = (<\text{s-set-opt:is-ref}>)
\]
\[
is-key-opt = (<\text{s-key-opt:is-expr}>)
\]
\[
is-keyto-opt = (<\text{s-keyto-opt:is-ref}>)
\]

Definition:

\[
\text{int-read-st(t) =}
\]
\[
\text{exists (end-filc(fid)) null}
\]
\[
\text{T --- int-read(t,key,ign);}
\]
\[
\text{- int-open-st(open);}
\]
\[
\text{open: m-open-read(t)}
\]
\[
\text{key : int-key(s-key-opt(s-key-keyto(t)))}
\]
\[
\text{ign : int-ign(s-ignore-opt(t))}
\]
\[
\begin{align*}
m\text{-open-read}(t) &= \mu_0 (<s\text{-file-opt}:s\text{-file-opt}(t)>, \\
&\quad <s\text{-open-attr}:\text{read-attr}_t>) \\
\text{where } \text{read-attr}_t &= \text{is-set-opt}(s\text{-destination}(t)) \rightarrow \\
&\quad \text{is-keyto-opt}(s\text{-key-keyto}(t)) \rightarrow \\
&\quad \{ \text{FILE, RECORD, BUFFERED, SEQUENTIAL, KEYED} \} \\
&\quad \text{is-key-opt}(s\text{-key-keyto}(t)) \rightarrow \\
&\quad \text{T} \rightarrow \{ \text{FILE, RECORD, BUFFERED} \} \\
T \rightarrow \\
&\quad \text{is-keyto-opt}(s\text{-key-keyto}(t)) \rightarrow \\
&\quad \{ \text{FILE, RECORD, SEQUENTIAL, KEYED} \} \\
&\quad \text{is-key-opt}(s\text{-key-keyto}(t)) \rightarrow \\
&\quad \{ \text{FILE, RECORD, KEYED} \} \\
&\quad \text{T} \rightarrow \{ \text{FILE, RECORD} \} \\
\end{align*}
\]

\[
\begin{align*}
\text{int-read}(t, \text{key}, \text{ign}) &= \\
&\quad \text{(is-input } \vee \text{ is-update)}(\text{csa}(\text{fid})) \land \\
&\quad (\neg \text{is-keyto-opt}(s\text{-key-keyto}(t)) \land \\
&\quad \text{is-proper-va}\text{r}.s-\text{attr}((s\text{-keyto-opt} \\
&\quad \text{(s\text{-key-keyto}(t)\.E)\.AT } \land \\
&\quad (\neg \text{is-into-opt}(s\text{-destination}(t)) \land \\
&\quad \text{is-proper-va}\text{r}.s-\text{attr}((s\text{-into-opt} \\
&\quad \text{(s\text{-destination}(t)\.E)\.AT }) \land \\
&\quad (\neg \text{is-set-opt}(s\text{-destination}(t)) \land \\
&\quad \text{is-proper-va}\text{r}.s-\text{attr}((s\text{-set-opt} \\
&\quad \text{(s\text{-destination}(t)\.E)\.AT }) \\
&\quad \text{int-proper-read}(t, \text{key}, \text{ign})
\end{align*}
\]
\[
\text{int-proper-read}(t, \text{key}, \text{ign}) = \\
\text{is-exclusive}(\text{csa}(\text{fid})) \rightarrow \text{exclusive-read}(t, \text{key}); \\
\text{int-wait}(\text{is-locked}(\text{fid}, \text{key})) \\
\text{is-direct}(\text{csa}(\text{fid})) \rightarrow \text{dir-into}(t, \text{key}) \\
\text{is-buffered}(\text{csa}(\text{fid})) \rightarrow \text{buffered-read}(t, \text{key}, \text{ign}) \\
T \rightarrow \text{unbuffered-read}(t, \text{key}, \text{ign})
\]

\[
\text{exclusive-read}(t, \text{key}) = \\
\text{exists}(\text{s-nolock-opt}(t)) \rightarrow \text{null;} \\
\text{dir-into}(t, \text{key}), \text{int-unlock}(t, \text{key}) \\
T \rightarrow \text{null;} \\
\text{dir-into}(t, \text{key}), \text{int-lock}(t, \text{key})
\]

\[
\text{buffered-read}(t, \text{key}, \text{ign}) = \\
\text{is-into-opt}(\text{s-destination}(t)) \rightarrow \text{read-buffered-into}(t, \text{key}) \\
\text{is-set-opt}(\text{s-destination}(t)) \rightarrow \text{read-buffered-set}(t, \text{key}) \\
T \rightarrow \text{update-next}(t, \text{key}, \text{ign})
\]

\[
\text{unbuffered-read}(t, \text{key}, \text{ign}) = \\
\text{is-key-opt}(\text{s-key-keyto}(t)) \rightarrow \text{new-next}(t, \text{key}); \\
\text{dir-into}(t, \text{key}) \\
\text{is-keyto-opt}(\text{s-key-keyto}(t)) \rightarrow \text{update-next}(t, \text{key}, 1); \\
\text{next-into}(t), \text{next-keyto}(t) \\
\text{is-into-opt}(\text{s-destination}(t)) \rightarrow \text{update-next}(t, \text{key}, 1); \\
\text{next-into}(t) \\
T \rightarrow \text{update-next}(t, \text{key}, \text{ign})
\]
\[
\text{read-buffered-into}(t, \text{key}) = \begin{cases} 
\text{is-key-opt}(s \text{-key-keyto}(t)) \rightarrow \text{new-next}(t, \text{key}); \\
\text{bu-into}(t); \\
\text{dir-bu}(t, \text{key}) \\
\text{is-keyto-opt}(s \text{-key-keyto}(t)) \rightarrow \text{update-next}(t, \text{key}, 1); \\
\text{kbu-keyto}(t); \\
\text{bu-into}(t); \\
\text{next-kbu}(t); \\
\text{next-bu}(t) \\
\end{cases}
\]

\[
T \rightarrow \text{update-next}(t, 1); \\
\text{bu-into}(t); \\
\text{next-bu}(t)
\]

\[
\text{read-buffered-set}(t, \text{key}) = \begin{cases} 
\text{is-key-opt}(s \text{-key-keyto}(t)) \rightarrow \text{new-next}(t, \text{key}); \\
\text{dir-cbu}(t, \text{key}) \\
\text{is-keyto-opt}(s \text{-key-keyto}(t)) \rightarrow \text{update-next}(t, \text{key}, 1); \\
\text{kbu-keyto}(t); \\
\text{next-kbu}(t); \\
\text{next-cbu}(t) \\
\end{cases}
\]

\[
T \rightarrow \text{update-next}(t, \text{key}, 1); \\
\text{next-cbu}(t)
\]
6.4.7 The Rewrite-Statement

Abstract Syntax:

\[
\text{is-rewrite-st} = (\langle \text{s-st:is-rewrite} \rangle, \\
\langle \text{s-file-opt:is-name} \rangle, \\
\langle \text{s-from-opt:(is-ref | is-Q)} \rangle, \\
\langle \text{s-key-opt:(is-expr | is-Q)} \rangle, \\
\langle \text{s-event-opt:(is-ref | is-Q)} \rangle)
\]

Definition:

\[
\text{int-rewrite-st}(t) = \\
\text{int-rewrite}(t, \text{key}); \\
\text{int-open-st}(\text{open}); \\
\text{open : m-open-rewrite}(t) \\
\text{key : int-key}(\text{s-key-opt}(t))
\]

\[
\text{m-open-rewrite}(t) = \\
\text{RETURNS :} \\
\mu_0 (\langle \text{s-file-opt:s-file-opt}(t) \rangle, \\
\langle \text{s-open-attr:} \rangle \\
\exists (\text{s-key-opt}(t)) \rightarrow \{ \text{FILE, RECORD, UPDATE, DIRECT, KEYED} \} \\
T \rightarrow \{ \text{FILE, RECORD, UPDATE} \})
\]

\[
\text{int-rewrite}(t, \text{key}) = \\
\neg \exists (\text{s-from-opt}(t)) \& \text{is-buffered}(\text{csa}(\text{fid})) \rightarrow \\
\text{update-next}(t, \text{key}, 1); \\
\text{bu-next}(t) \\
\exists (\text{s-from-opt}(t)) \& \text{is-record-ref}(\text{s-from-opt}(t)) \rightarrow \\
\text{from-rewrite}(t, \text{key})
\]
from-rewrite(t,key) =
  is-buffered(csa(fid)) → update-next(t,key,1);
                  from-hu(t)
  is-sequential(csa(fid)) → update-next(t,key,1);
              from-next(t,key)
  is-exclusive(csa(fid)) → null;
               int-unlock(t,key)
                   from-dir(t,key)
       T → from-dir(t,key)

Note: In the case where a REWRITE statement without a FROM option follows a READ statement with the INTO option the transmission of the record to the data set is redundant.

6.4.8 The Delete Statement

Abstract Syntax:

is-delete-st = (<s-st:is-delete>,
          <s-file-opt:is-name>,
          <s-key-opt:is-expr>,
          <s-event-opt:(is-ref | is-Q)>)

Definition:

int-delete-st(t) =
   int-delete(t,key);
   int-open-st(open);
       open : m-open-delete(t)
   key : int-key(s-key-opt(t))
m-open-delete(t) =
  RETURNS: \( \mu \langle s-file-opt:s-file-opt(t)\rangle, \langle s-open-attr:\{ \text{FILE, RECORD, UPDATE, DIRECT, KEYED} \} \rangle \)

int-delete(t,key) =
  null;
  int-unlock(t,key)
  del-record(t,i)
  i : keyed-access(t,key)

del-record(t,i) =
  s-es: \( \delta(ES, \text{data-elem}(fid,i)) \)

Note: It is assumed that both, record and key has to be deleted from the data set.

6.4.9 The Unlock Statement

Abstract Syntax:

\( \text{is-unlock-st} = (\langle s-st:is-unlock\rangle, \langle s-file-opt:is-name\rangle, \langle s-key-opt:is-expr\rangle) \)

Definition:

\( \text{int-unlock-st}(t) = \)
  int-unlock(t,key);
  int-open-st(open);
  open : m-open-unlock(t)
  key : int-key(s-key-opt(t))

m-open-unlock(t) =
  RETURNS: \( \mu \langle s-file-opt:s-file-opt(t)\rangle, \langle s-open-attr:\{ \text{FILE, RECORD, DIRECT, UPDATE, KEYED, EXCLUSIVE} \} \rangle \)
6.4.10 The Locate Statement

Abstract Syntax:

is-locate-st = (<s-st:is-locate>,
             <s-file-opt:is-name>,
             <s-variable:is-level-one-ref>,
             <s-set-opt:is-ref>,
             <s-keyfrom-opt:(is-expr | is= Ω )>)

Definition:

int-locate-st(t) =
   int-locate(t,keyfrom);
   int-open-st(open);
   open : m-open-locate(t)
   keyfrom : int-key(s-keyfrom-opt(t))

m-open-locate(t) =
   RETURNS :
   µ₁(<s-file-opt:s-file-opt(t)>,
       <s-open-attr:
        exists(s-keyfrom-opt(t)) →
        { FILE, RECORD, OUTPUT, SEQUENTIAL, BUFFERED, KEYED }
        T →
        { FILE, RECORD, OUTPUT, SEQUENTIAL, BUFFERED }>)

int-locate(t,keyfrom) =
   int-set-locate(t);
   int-keyfrom-locate(t,keyfrom);
   exec-locate(t)
int-keyfrom-locate(t,keyfrom) =  
exists(s-keyfrom-opt(t)) →  
null;  
keyfrom-kbu(t,keyfrom)  
put-keyed-locate(t)  
T → put-unkeyed-locate(t)

put-keyed-locate(t) =  
    s-fu : μ(FU,<locate-exec-sel(fid):KEYED>)  

put-unkeyed-locate(t) =  
    s-fu : μ(FU,<locate-exec-sel(fid):UNKEYED>)

Note: The locate-exec part of the fu-elem(fid) signals to the next
WRITE, LOCATE or CLOSE statement that the record in the buffer
must be written to the data set. This operation is done by
exec-locate.

6.4.11 The Write Statement

Abstract Syntax:

is-write-st = (<s-st:is-write>,
    <s-file-opt:is-name>,
    <s-from-opt:is-ref>,
    <s-keyfrom-opt:(is-expr | is-Ω )>,
    <s-event-opt:(is-ref | is-Ω )>)

Definition:

int-write-st(t) =  
    int-write(t,keyfrom);  
    int-open-st(open);  
    open : m-open-write(t)  
    keyfrom : int-key(s-keyfrom-opt(t))
\[\text{m-open-write}(t) = \]
\[
\text{RETURNS: } \mu_o (<s-file-opt:s-file-opt(t)>, <s-open-attr:}
\]
\[
\quad \exists(s-keyfrom-opt(t)) \mapsto
\]
\[
\quad \{ \text{FILE, RECORD, KEYED} \}
\]
\[
\quad T \mapsto
\]
\[
\quad \{ \text{FILE, RECORD} \}
\]
\[
\text{int-write}(t, \text{keyfrom}) =
\]
\[
\exists(s-from-opt(t)) \& \]
\[
\text{is-record-ref}(s-from-opt(t)) \mapsto
\]
\[
\text{int-proper-write}(t, \text{keyfrom})
\]
\[
\text{int-proper-write}(t, \text{keyfrom}) =
\]
\[
(is-direct \& (is-output \lor is-update))(\text{csa}(\text{fid})) \mapsto
\]
\[
\quad \text{update-next}(t, \text{key}, 1);
\]
\[
\quad \text{from-next}(t, \text{key})
\]
\[
\quad \text{keyfrom-next}(t, \text{keyfrom})
\]
\[
(is-buffered \& is-output)(\text{csa}(\text{fid})) \mapsto
\]
\[
\quad \text{update-next}(t, \text{key}, 1);
\]
\[
\quad \text{optk-from-bu-next}(t, \text{keyfrom})
\]
\[
\quad \text{exec-locate}(t)
\]
\[
\text{is-output}(\text{csa}(\text{fid})) \& \exists(s-keyfrom-opt(t)) \mapsto
\]
\[
\quad \text{update-next}(t, \text{key}, 1);
\]
\[
\quad \text{from-next}(t, \text{key})
\]
\[
\quad \text{keyfrom-next}(t, \text{keyfrom})
\]
\[
\text{is-output}(\text{csa}(\text{fid})) \mapsto
\]
\[
\quad \text{update-next}(t, \text{key}, 1);
\]
\[
\quad \text{from-next}(t, \text{key})
\]
\[
\text{optk-from-bu-next}(t, \text{keyfrom}) =
\]
\[
\exists(s-keyfrom-opt(t)) \mapsto
\]
\[
\quad \text{kbu-next-f}(t);
\]
\[
\quad \text{from-bu}(t),
\]
\[
\quad \text{keyfrom-kbu}(t, \text{keyfrom})
\]
\[
T \mapsto
\]
\[
\quad \text{bu-next-f}(t);
\]
\[
\quad \text{from-bu}(t)
\]
exec-locate(t) =
  is-keyed(locate-exec(fid)) →
    del-exec-locate(t);
    update-next(t,key,1);
    kbu-next-f(t)
  is-unkeyed(locate-exec(fid)) →
    del-exec-locate(t);
    update-next(t,key,1);
    bu-next-f(t)
  T → null

del-exec-locate(t) =
  s-fu : δ(FU, locate-exec-sel(fid))
6.5 Closing of a File

6.5.1 Interpretation of the close-statement

Abstract Syntax:

\[ \text{is-close-st} = \text{is-close-list} \]
\[ \text{is-close} = (\text{<s-st:is-cl>}, \text{<s-file-opt:is-name>}, \text{<s-ident-opt: (is-expr | is-Ω)}> \)

Implementation-defined Functions: see 6.2.2.

Abbreviation:

\[ \text{fid} = \text{s-file-opt}(\text{te}) \]

Definition:

\[ \text{int-close-st}(t) = \text{null}; \]
\[ \{ \text{int-close}(\text{elem}(i,t)) \mid i \leq \text{length}(t) \} \]
\[ \text{int-close}(\text{te}) = \]
\[ \text{exists(old-fa(fid))} \rightarrow \text{int-unable-close}(\text{te}) \]
\[ \text{exists(new-fa(fid))} \rightarrow \]
\[ \text{null}; \]
\[ \text{delete-fa(den(fid))}, \]
\[ \text{delete-locking(locking-sel(fid))}; \]
\[ \text{exec-locate(te)}, \]
\[ \text{int-close-label(te, s-ident-opt(te))} \]
delete-fa(den) = δ(FD, den·s-nfa)
delete-locking(sel) = δ(FU, curr-task-ident·sel)

Note: curr-task-ident is the identification of the current task;
curr-task-ident ∈ TA see 3.1.73

int-close-label(te, ident) =
  ¬ exists(ident) → null
  ¬ is-output(csa(fid)) & is-backwards(csa(fid)) &
  is-simple-ref(ident) →
    assign(gen, translate-char(header(fid)));
    gen : int-variable(ident, E)
  ¬ is-output(csa(fid)) & ¬ is-backwards(csa(fid)) &
  is-simple-ref(ident) →
    assign(gen, translate-char(trailer(fid)));
    gen : int-variable(ident, E)
  ¬ is-output(csa(fid)) → int-unable-assign(te)
  T → update-es-part(trailer-sel(fid),
    translate-user-label(label));
  label : int-opt-expr(μ_o (<s-type:CHAR>,
    <s-length:EMPTY>), ident)

int-unable-close(te) =
int-unable-assign(te) =
  implementation-defined
7. PARALLEL ACTIONS

This chapter has been prepared but it was decided to hold it for further consideration.
APPENDIX I ABSTRACT SYNTAX OF TEXT
(Text not yet handled by prepass)

1. Notations

According to the definitions in chapter 2 in this appendix the following notational conventions are used:

1. \( \text{is-pred} = (\text{is-pred}_1 \mid \text{is-pred}_2 \mid \ldots \mid \text{is-pred}_n) \)
   \[ \equiv (\text{is-pred}(x) \equiv \text{is-pred}_1(x) \land \neg(\text{is-pred}_2(x) \lor \ldots \lor \neg(\text{is-pred}_n(x) \lor \ldots \lor \text{is-pred}_n(x)) \lor \ldots \lor \text{is-pred}_n(x)) \lor \ldots \lor \text{is-pred}_n(x)) \]

2. \( \text{is-pred} = (s\text{-sel}_1: \text{is-pred}_1, \ldots, s\text{-sel}_n: \text{is-pred}_n) \)
   \[ \equiv (\text{is-pred}(x) \equiv (\exists x_1, \ldots, x_n) (\text{is-pred}_1(x_1) \land \ldots \land \text{is-pred}_n(x_n) \land x = \mu_0(s\text{-sel}_1:x_1, \ldots, s\text{-sel}_n:x_n)) \]

3. \( \text{is-pred} = (\{ y : \text{is-pred}_2 \mid \text{is-pred}_1(y) \}) \)
   \[ \equiv (\text{is-pred}(x) \equiv (\exists y_1, z_1, \ldots, y_n, z_n) (\text{is-pred}_1(y_1) \land \ldots \land \text{is-pred}_2(z_1) \land \ldots \land \text{is-pred}_2(z_n) \land x = \mu_0(y_1 : z_1, \ldots, y_n : z_n)) \]

4. \( \text{is-pred-set}(x) \equiv (\exists x_1, \ldots, x_n) (\text{is-pred}(x_1) \land \ldots \land \text{is-pred}(x_n) \land x = \{ x_1, \ldots, x_n \}) \lor x = \{ \} \)
(5) \( \text{is-pred-list}(x) \equiv (\exists x_1, \ldots, x_n)(\text{is-pred}(x_1) \& \ldots \& \text{is-pred}(x_n) \& x = \mu_0(<\text{elem}(1):x_1>, \ldots, <\text{elem}(n):x_n>)) \)

(6) \( \text{is-pred-tree}(x) \equiv (\exists x_1, \ldots, x_n)(\text{is-pred}(x_1) \& \ldots \& \text{is-pred}(x_n) \& x = \mu_0(<\text{succ}(1):x_1>, \ldots, <\text{succ}(n):x_n>)) \)

(7) \( \text{is-pred-pair}(x) \equiv (\exists x_1, x_2)(\text{is-pred}(x_1) \& \text{is-pred}(x_2) \& x = \mu_0(<\text{s-first}:x_1>, <\text{s-second}:x_2>)) \)

2. List of Non-elementary Predicate Definitions

(1) \( \text{is-program} = (\{<\text{id}:\text{is-ext-proc}> \mid \text{is-name}(\text{id})\}) \)

(2) \( \text{is-ext-proc} = (<\text{s-id}:\text{is-name}>, <\text{s-body}:\text{is-body}>) \)

(3) \( \text{is-body} = (<\text{s-param-list}:\text{is-name-list}>, <\text{s-decl-part}:\text{is-decl-part}>, <\text{s-cond-part}:\text{is-cond-part}>, <\text{s-ret-type}:(\text{is-arithm} \mid \text{is-string} \mid \text{is-pict} \mid \text{is-ptr}>, <\text{s-st-list}:\text{is-st-list}>) \)

(4) \( \text{is-decl-part} = (\{<\text{id}:\text{is-decl}> \mid \text{is-name}(\text{id})\}) \)

(5) \( \text{is-decl} = (<\text{s-id}:\text{is-name}>, <\text{s-scope}:(\text{is-int} \mid \text{is-ext} \mid \text{is-param})>, <\text{s-attr}:\text{is-attr}>, <\text{s-den}:(\text{is-body} \mid \text{is-st-list} \mid \text{is-format-elem-list} \mid \text{is-empty})>) \)
(6) \textit{is-attr} = (\textit{is-prop-variable} \mid \textit{is-data-param} \mid \textit{is-entry} \mid \textit{is-file} \mid \\
\quad \textit{is-defined} \mid \textit{is-based} \mid \textit{is-generic} \mid \textit{is-built-in} \mid \\
\quad \textit{is-label-const} \mid \textit{is-cond-name})

(7) \textit{is-prop-variable} = (\textit{<s-stg-cl:(is-static \mid is-auto \mid is-ctr)>}, \\
\quad \textit{<s-dens:(is-packed \mid is-aligned)>}, \\
\quad \textit{<s-da:is-named-da>}, \\
\quad \textit{<s-init:is-init-set>})

(8) \textit{is-named-da} = (\textit{is-named-struct} \mid \textit{is-named-array} \mid \textit{is-named-scalar})

(9) \textit{is-named-struct} = \textit{is-named-succ-tree}

(10) \textit{is-named-succ} = (\textit{<s-id:is-name>}, \\
\quad \textit{<s-da:is-named-da>})

(11) \textit{is-named-array} = (\textit{<s-lbd:(is-expr \mid is-empty)>}, \\
\quad \textit{<s-ubd:(is-expr \mid is-empty)>}, \\
\quad \textit{<s-elem:is-named-da>})

(12) \textit{is-named-scalar} = (\textit{is-arithm} \mid \textit{is-string} \mid \textit{is-pict} \mid \textit{is-label} \mid \\
\quad \textit{is-task} \mid \textit{is-event} \mid \textit{is-ptr} \mid \textit{is-named-cell} \mid \\
\quad \textit{is-area})

(13) \textit{is-arithm} = (\textit{<s-mode:(is-real \mid is-compl)>}, \\
\quad \textit{<s-base:(is-bin \mid is-dec)>}, \\
\quad \textit{<s-scale:(is-fix \mid is-flt)>}, \\
\quad \textit{<s-prec:(is-integer \mid is-integer-pair>})

(14) \textit{is-string} = (\textit{<s-base:(is-bit \mid is-char)>}, \\
\quad \textit{<s-length:(is-expr \mid is-empty)>}, \\
\quad \textit{<s-var:is-opt>})

(15) \textit{is-opt} = (\textit{is-true} \mid \textit{is-Ω})

(16) \textit{is-pict} = (\textit{<s-pict-spec:is-const>}, \\
\quad \textit{<s-mode:(is-char \mid is-real \mid is-compl)>})

(17) \textit{is-label} = (\textit{is-unrest-label} \mid \textit{is-rest-label})

(18) \textit{is-rest-label} = (\textit{<s-label-const-set:is-name-set>})

(19) \textit{is-named-cell} = \textit{is-named-succ-list}
(20) \( \text{is-area} = (\text{is-default-area} \mid \text{is-spec-area}) \)

(21) \( \text{is-spec-area} = (\langle \text{s-area-descr:is-data-descr-list} \rangle) \)

(22) \( \text{is-data-descr} = (\langle \text{s-dens:} (\text{is-packed} \mid \text{is-aligned}) \rangle,
\quad \langle \text{s-da:is-da-descr} \rangle) \)

(23) \( \text{is-da-descr} = (\text{is-struct-descr} \mid \text{is-array-descr} \mid \text{is-scalar-descr}) \)

(24) \( \text{is-struct-descr} = \text{is-succ-descr-tree} \)

(25) \( \text{is-succ-descr} = (\langle \text{s-da:is-da-descr} \rangle) \)

(26) \( \text{is-array-descr} = (\langle \text{s-lbd:} (\text{is-expr} \mid \text{is-empty}) \rangle,
\quad \langle \text{s-ubd:} (\text{is-expr} \mid \text{is-empty}) \rangle,
\quad \langle \text{s-elem:is-da-descr} \rangle) \)

(27) \( \text{is-scalar-descr} = (\text{is-arithm} \mid \text{is-string} \mid \text{is-pict} \mid \text{is-label} \mid
\quad \text{is-task} \mid \text{is-event} \mid \text{is-ptr} \mid \text{is-cell-descr} \mid
\quad \text{is-area}) \)

(28) \( \text{is-cell-descr} = \text{is-succ-descr-list} \)

(29) \( \text{is-init} = (\langle \text{s-first:is-simple-ref} \rangle,
\quad \langle \text{s-second:} (\text{is-init-item-list} \mid \text{is-call-st}) \rangle) \)

(30) \( \text{is-init-item} = (\text{is-init-const} \mid \text{is-empty} \mid \text{is-init-iter}) \)

(31) \( \text{is-init-iter} = (\langle \text{s-rep-factor:is-expr} \rangle,
\quad \langle \text{s-item-list:is-init-item-list} \rangle) \)

(32) \( \text{is-init-const} = (\text{is-const} \mid \text{is-const-expr} \mid \text{is-name} \mid \text{is-null}) \)

(33) \( \text{is-const-expr} = (\langle \text{s-operator:} (\text{is-add} \mid \text{is-subtr}) \rangle,
\quad \langle \text{s-op-1:is-const} \rangle,
\quad \langle \text{s-op-2:is-const} \rangle) \)

(34) \( \text{is-data-param} = (\langle \text{s-stg-cl:} (\text{is-ctr} \mid \text{is-empty}) \rangle,
\quad \langle \text{s-dens:} (\text{is-packed} \mid \text{is-aligned}) \rangle,
\quad \langle \text{s-da:is-named-da} \rangle) \)

(35) \( \text{is-entry} = (\langle \text{s-param-list:} (\text{is-param-descr-list} \mid \text{is-empty}) \rangle,
\quad \langle \text{s-ret-type:} (\text{is-arithm} \mid \text{is-string} \mid
\quad \text{is-pict} \mid \text{is-ptr} \mid \text{is-empty}) \rangle) \)
(36) is-param-descr = (is-data-param-descr | is-entry | is-file | is-empty)

(37) is-data-param-descr = (<s-stg-cl:(is-ctr | is-empty)>,
              <s-dens:(is-packed | is-aligned)>,
              <s-da:is-da-descr>)

(38) is-file = (<s-file-attr:is-file-attr-set>,
              <s-io-env:is-io-opt-set>,
              <s-title:is-const>)

(39) is-file-attr = (is-stream | is-record | is-input | is-output | is-update | is-sequential | is-direct |
                is-buffered | is-unbuffered | is-print | is-backwards | is-exclusive | is-keyed)

(40) is-defined = (<s-base:is-simple-ref>,
                <s-pos:(is-integer | is-Ω)>,
                <s-dens:(is-packed | is-aligned)>,
                <s-da:is-named-da>)

(41) is-based = (<s-ptr:is-ref>,
               <s-dens:(is-packed | is-aligned)>,
               <s-da:is-named-da>,
               <s-init:is-init-set>)

(42) is-generic = is-entry-decl-set

(43) is-entry-decl = (<s-id:is-name>,
              <s-scope:(is-int | is-ext | is-param)>,
              <s-attr:is-entry>,
              <s-den:(is-body | is-empty)>)

(44) is-format-elem = (is-format-item | is-format-item-iter)

(45) is-format-item-iter = (<s-rep-factor:is-expr>,
              <s-element:(is-format-item | is-format-elem-list)>)

(46) is-format-item = (is-data-format | is-control-format | is-remote-format)
is-data-format = (is-real-format | is-compl-format |
  is-string-format | is-pict-format)

is-real-format = (<s-format-type:(is-fix | is-flt)>,
  <s-w:is-expr>,
  <s-d:(is-expr | is-Ω)>,
  <s-p:(is-expr | is-Ω)>,
  <s-s:(is-expr | is-Ω)>)

is-compl-format = (<s-format-type:is-compl>,
  <s-real-part:(is-real-format |
    is-pict-format)>,
  <s-imag-part:(is-real-format |
    is-pict-format)>)

is-string-format = (<s-format-type:(is-bit | is-char)>,
  <s-w:is-expr>)

is-pict-format = (<s-format-type:is-picture>,
  <s-pict-spec:is-const>)

is-control-format = (<s-format-type:is-control-format-type>,
  <s-w:(is-expr | is-Ω)>)

is-control-format-type = (is-space | is-page | is-skip |
  is-line | is-column)

is-remote-format = (<s-format-type:is-remote>,
  <s-label-ref:is-ref>)

is-cond-part = (<s-conv:is-opt>,
  <s-fofl:is-opt>,
  <s-ofl:is-opt>,
  <s-size:is-opt>,
  <s-subrg:is-opt>,
  <s-ufl:is-opt>,
  <s-zdiv:is-opt>,
  <s-check:is-simple-ref-set>)

is-st = (is-block | is-group | is-if-st | is-on-st |
  is-simple-st)
is-block = (s-decl-part:is-decl-part,  
            s-cond-part:is-cond-part,  
            s-label-list:is-name-list,  
            s-st-list:(is-st-list|is-st-list-while |  
                       is-st-list-contr))

is-st-list-while = (s-while-expr:is-expr,  
                   s-st-list:is-st-list)

is-st-list-contr = (s-contr-var:is-ref,  
                    s-spec-list:is-do-spec-list,  
                    s-st-list:is-st-list)

is-do-spec = (s-init-expr:is-expr,  
              s-by-expr:is-expr,  
              s-to-expr:(is-expr | is-empty),  
              s-while-expr:is-expr)

is-group = (s-cond-part:is-empty,  
            s-label-list:is-name-list,  
            s-st-list:is-st-list)

is-if-st = (s-cond-part:is-cond-part,  
           s-label-list:is-name-list,  
           s-decision:is-expr,  
           s-then-st:is-st,  
           s-else-st:is-st)

is-on-st = (s-cond-part:is-cond-part,  
           s-label-list:is-name-list,  
           s-on-cond:is-cond,  
           s-snap:is-opt,  
           s-on-unit:(is-block | is-simple-st |  
                      is-system-action))

is-cond = (is-conv | is-fofl | is-ofl | is-size | is-subrg |  
           is-ufl | is-zdiv | is-check | is-named-io-cond |  
           is-area-size | is-error | is-finish |  
           is-progr-named-cond)
(65) is-check = (<s-check:is-simple-ref-set>)
(66) is-named-io-cond = (<s-cond:is-io-cond>,
    <s-file:is-name>)
(67) is-io-cond = (is-data-name | is-endfile | is-endpage |
    is-key | is-record-size | is-transmit | is-undf)
(68) is-progr-namc<l-cond = (<s-cond-name:is-name>)
(69) is-simple-st = (<s-cond-part:is-cond-part>,
    <s-label-list:is-name-list>,
    <s-p-st:is-prop-st>)
(70) is-prop-st = (is-assign-st | is-allocate-st | is-call-st |
    is-close-st | is-delay-st | is-delete-st |
    is-display-st | is-exit-st | is-free-st |
    is-get-st | is-goto-st | is-locate-st |
    is-open-st | is-put-st | is-read-st |
    is-return-st | is-revert-st | is-rewrite-st |
    is-signal-st | is-stop-st | is-unlock-st |
    is-wait-st | is-write-st | is-null-st)
(71) is-assign-st = (<s-st:is-assign>,
    <s-lp:is-ref-list>,
    <s-rp:is-expr>,
    <s-name-opt:is-opt>)
(72) is-expr = (is-infix-expr | is-prefix-expr | is-prim-expr)
(73) is-infix-expr = (<s-operator:is-infix-operator>,
    <s-op-1: is-expr>,
    <s-op-2:is-expr>)
(74) is-infix-operator = (is-cat | is-or | is-and | is-lt | is-le |
    is-eq | is-ge | is-gt | is-ne | is-add |
    is-subtr | is-mult | is-div | is-power)
(75) is-prefix-expr = (<s-operator:is-prefix-operator>,
    <s-op:is-expr>)
(76) is-prefix-operator = (is-minus | is-not)
(77) is-prim-expr = (is-ref | is-const | is-isub)
(78) is-ref = (is-ptr-qual-ref | is-simple-ref)
(79) is-ptr-qual-ref = (\textless s-ptr-qual:is-ref, \\
\textless s-based-ref:is-based-ref\textgreater \\
(80) is-simple-ref = (\textless s-id:is-name, \\
\textless s-qual-list:is-name-list, \\
\textless s-arg-list:is-arg-list\textgreater )
(81) is-arg = (is-expr | is-empty)
(82) is-const = (\textless s-da:(is-arithm | is-string), \\
\textless s-vr:(is-vr | is-vr-list)\textgreater )
(83) is-integer = (\textless s-mode0s-da:is-real, \\
\textless s-base0s-da:is-doc, \\
\textless s-scale0s-da:is-fix, \\
\textless s-prec0s-da:is-def-prec, \\
\textless s-vr:is-vr\textgreater )
(84) is-isub = (\textless s-i:is-integer\textgreater )
(85) is-allocate-st = is-allocate-list
(86) is-allocate = (is-contr-allocate | is-based-allocate \\
| is-area-allocate)
(87) is-contr-allocate = (\textless s-st:is-alloc, \\
\textless s-id:is-name, \\
\textless s-al:is-al, \\
\textless s-is:is-init-set\textgreater )
(88) is-al = (is-struct-al | is-array-al | is-string-al | is-empty)
(89) is-struct-al = is-al-tree
(90) is-array-al = (\textless s-lbd:(is-expr | is-empty), \\
\textless s-ubd:(is-expr | is-empty), \\
\textless s-elem:is-al\textgreater )
(91) is-string-al = (\textless s-base:(is-bit | is-char), \\
\textless s-length:(is-expr | is-empty)\textgreater )
(92) is-based-allocate = (\textless s-st:is-alloc, \\
\textless s-id:is-name, \\
\textless s-set-variable:is-ref\textgreater )
(93) is-area-allocate = (\textless s-st:is-alloc, \\
\textless s-id:is-name, \\
\textless s-set-variable:is-ref, \\
\textless s-area:is-ref\textgreater )
(94) \( \text{is-call-st} = (\langle \text{s-st:is-call}, \linebreak \quad \langle \text{s-id:is-name}, \linebreak \quad \langle \text{s-arg-list:is-expr-list}, \linebreak \quad \langle \text{s-opt: (is-pa-opt | is-Ω) } \rangle \rangle \) \)

(95) \( \text{is-pa-opt} = (\langle \text{s-task-var: (is-ref | is-empty) }, \linebreak \quad \langle \text{s-priority: (is-expr | is-empty) }, \linebreak \quad \langle \text{s-event-var: (is-ref | is-Ω) } \rangle \rangle \) \)

(96) \( \text{is-close-st} = \text{is-close-list} \)

(97) \( \text{is-close} = (\langle \text{s-st:is-cl}, \linebreak \quad \langle \text{s-file-opt:is-name}, \linebreak \quad \langle \text{s-ident-opt: (is-expr | is-Ω) } \rangle \rangle \) \)

(98) \( \text{is-delay-st} (\langle \text{s-st:is-delay}, \linebreak \quad \langle \text{s-time:is-expr} \rangle \) \)

(99) \( \text{is-delete-st} = (\langle \text{s-st:is-delete}, \linebreak \quad \langle \text{s-file-opt:is-name}, \linebreak \quad \langle \text{s-key-opt:is-expr}, \linebreak \quad \langle \text{s-event-opt: (is-ref | is-Ω) } \rangle \rangle \) \)

(100) \( \text{is-display-st} = (\langle \text{s-st:is-display}, \linebreak \quad \langle \text{s-display:is-expr}, \linebreak \quad \langle \text{s-reply: (is-ref | is-Ω) }, \linebreak \quad \langle \text{s-event: (is-ref | is-Ω) } \rangle \rangle \) \)

(101) \( \text{is-free-st} = (\langle \text{s-st:is-free}, \linebreak \quad \langle \text{s-free-list:is-level-one-ref-list} \rangle \) \)

(102) \( \text{is-level-one-ref} = (\text{is-simple-name | is-ptr-qual-level-one-ref}) \)

(103) \( \text{is-simple-name} = (\langle \text{s-id:is-name}, \quad \langle \text{s-qual-list:is-emptylist}, \quad \langle \text{s-arg-list:is-emptylist} \rangle \rangle \)

(104) \( \text{is-ptr-qual-level-one-ref} = (\langle \text{s-ptr-qual:is-ref}, \quad \langle \text{s-based-ref:is-simple-name} \rangle \) \)

(105) \( \text{is-get-st} = (\text{is-stream-get | is-string-get}) \)
is-stream-get = (<s-st:is-get>,
    <s-file-opt:is-name>,
    <s-spec:is-get-spec>,
    <s-copy-opt:is-opt>,
    <s-mark-opt:is-opt>)

is-string-get = (<s-st:is-get>,
    <s-string-opt:is-ref>,
    <s-spec:is-get-spec>)

is-get-spec = ((<s-type:is-data-dir>,
    <s-ref-set:is-ref-set>) |
    (<s-type:is-edit-dir>,
    <s-data-list:is-ref-elem-list>,
    <s-format-list:is-format-elem-list>) |
    (<s-type:is-list-dir>,
    <s-data-list:is-ref-elem-list>))

is-ref-elem = (is-ref | is-ref-contr)

is-ref-contr = (<s-contr-var:is-ref>,
    <s-spec-list:is-do-spec-list>,
    <s-data-list:is-ref-elem-list>)

is-goto-st = (<s-st:is-goto>,
    <s-lab-ref:is-ref>)

is-locate-st = (<s-st:is-locate>,
    <s-file-opt:is-name>,
    <s-variable:is-level-one-ref>,
    <s-set-opt:is-ref>,
    <s-keyfrom-opt:(is-expr | is-Ω)>)

is-open-st = is-open-list

is-open = (<s-st:is-op>,
    <s-file-opt:is-name>,
    <s-ident-opt:(is-expr | is-Ω)>,
    <s-title-opt:(is-expr | is-Ω)>,
    <s-pagesize-opt:(is-expr | is-Ω)>,
    <s-linesize-opt:(is-expr | is-Ω)>,
    <s-open-attr:is-file-attr-set>
(115) is-put-st = (is-stream-put | is-string-put)
(116) is-stream-put = (<s-st:is-put>,
     <s-file-opt:is-name>,
     <s-spec:(is-put-spec | is-Q)>,
     <s-page-opt:is-opt>,
     <s-line-opt:(is-expr | is-Q)>,
     <s-skip-opt:(is-expr | is-Q)>)
(117) is-string-put = (<s-st:is-put>,
     <s-string-opt:is-ref>,
     <s-spec:is-put-spec>)
(118) is-put-spec = ((<s-type:is-data-dir>,
     <s-data-list:is-ref-elem-list>) |
     (<s-type:is-edit-dir>,
     <s-data-list:is-expr-elem-list>,
     <s-format-list:is-format-elem-list>) |
     (<s-type:is-list-dir>,
     <s-data-list:is-expr-elem-list>))
(119) is-expr-elem = (is-expr | is-expr-contr)
(120) is-expr-contr = (<s-contr-var:is-ref>,
     <s-spec-list:is-do-spec-list>,
     <s-data-list:is-expr-elem-list>)
(121) is-read-st = (<s-st:is-read>,
     <s-file-opt:is-name>,
     <s-destination:(is-into-opt | is-set-opt | is-Q)>,
     <s-key-keyto:(is-key-opt | is-keyto-opt | is-Q)>,
     <s-ignore-opt:(is-expr | is-Q)>,
     <s-nolock-opt:is-opt>,
     <s-event-opt:(is-ref | is-Q)>)
(122) is-into-opt = (<s-into-opt:is-level-one-ref>)
(123) is-set-opt = (<s-set-opt:is-ref>)
(124) is-key-opt = (<s-key-opt:is-expr>)
(125) is-keyto-opt = (<s-keyto-opt:is-ref>)
(126) is-return-st = (<s-st:is-return>,
    <s-expr:(is-expr | is-Q)>)
(127) is-revert-st = (<s-st:is-revert>,
    <s-on-cond:is-cond>)
(128) is-rewrite-st = (<s-st:is-rewrite>,
    <s-file-opt:is-name>,
    <s-from-opt:(is-level-one-ref | is-Q)>,
    <s-key-opt:(is-expr | is-Q)>,
    <s-event-opt:(is-ref | is-Q)>)
(129) is-signal-st = (<s-st:is-signal>,
    <s-on-cond:is-cond>)
(130) is-unlock-st = (<s-st:is-unlock>,
    <s-file-opt:is-name>,
    <s-key-opt:is-expr>)
(131) is-wait-st = (<s-st:is-wait>,
    <s-event-set:is-ref-set>,
    <s-event-number:is-expr>)
(132) is-write-st = (<s-st:is-write>,
    <s-file-opt:is-name>,
    <s-from-opt:is-level-one-ref>,
    <s-keyfrom-opt:(is-expr | is-Q)>,
    <s-event-opt:(is-ref | is-Q)>)
3. List of Elementary Predicates

The following four classes of predicates are used in the definitions of the Abstract Syntax of text as listed in the previous section without being defined there.

3.1 is-name and is-vr

These two predicates characterize two infinite classes of distinct elementary objects, which are unique representations of the names of the language respective the value representations of numerical values, characters and bits.

3.2 is-def-prec,is-io-opt,is-system-action

These three predicates characterize classes of implementation defined objects and are left undefined by the interpretation.

3.3 is-∅

This predicate is equivalent to the predicate "¬exists", i.e. the following three statements are equivalent:

\[\text{is-∅}(x), x = ∅, ¬\text{exists}(x).\]

3.4 Predicates characterizing a unique elementary object

Each of the following predicates determines uniquely one elementary object. The name of this object is that part of the predicate name, which follows the prefix is-, transcribed to capital letters.

Examples:

\[\text{is-bit}(x) \equiv x = \text{BIT},\]

\[\text{is-label-const}(x) \equiv x = \text{LABEL-CONST}.\]
List of these predicates:

- is-add
- is-aligned
- is-alloc
- is-and
- is-area-size
- is-assign
- is-auto
- is-backwards
- is-bin.
- is-bit
- is-buffered
- is-built-in
- is-call
- is-cat
- is-char
- is-cl
- is-column
- is-compl
- is-conc
- is-cond-name
- is-ctr
- is-data-dir
- is-data-name
- is-dec
- is-default-area
- is-delay
- is-delete
- is-direct
- is-display
- is-div
- is-edit-dir
- is-empty
- is-empty-list
- is-endfile
- is-endpage
- is-eq
- is-error
- is-event
- is-exclusive
- is-exit-st
- is-ext
- is-finish
- is-fix
- is-flt
- is-fofl
- is-free
- is-ge
- is-get
- is-goto
- is-gt
- is-input
- is-int
- is-key
- is-keyed
- is-label-const
- is-le
- is-line
- is-list-dir
- is-locate
- is-lt
The predicate is-emptylist characterizes the empty list, i.e. the following three statements are equivalent:

\[
is\text{-emptylist}(x), x = \text{EMPTYLIST}, x = < >.
\]
4. Additional Conditions

A program as defined by the rules of section 2 of this appendix is correct only if the following additional syntactic conditions are satisfied:

\[(\text{is-program}(x) \lor \text{is-decl-part}(x)) \land \text{is-name}(y) \land \\
(y(x) \neq \emptyset) \lor s\text{-id}\cdot y(x) = y \\
is\text{ body}(x) \lor \\
(\forall y)((\exists i)(y = \text{elem}(i) \cdot s\text{-param-list}(x)) \\
\quad \lor \text{is-param}(s\text{-scope}\cdot y\cdot s\text{-decl-part}(x))) \\
\text{is-param}(s\text{-scope}(x)) \lor \text{is-data-param}(s\text{-attr}(x)) \lor \\
\quad \text{is-entry}(s\text{-attr}(x)) \lor \\
\quad \text{is-file}(s\text{-attr}(x)) \\
\text{is-prop-variable}(s\text{-attr}(x)) \lor \text{is-int}(s\text{-scope}(x)) \lor \\
\quad \text{is-ext}(s\text{-scope}(x)) \\
\text{is-data-param}(s\text{-attr}(x)) \lor \text{is-param}(s\text{-scope}(x)) \\
\text{is-defined}(s\text{-attr}(x)) \lor \\
\text{is-based}(s\text{-attr}(x)) \lor \\
\text{is-generic}(s\text{-attr}(x)) \lor \\
\text{is-label-const}(s\text{-attr}(x)) \lor \text{is-int}(s\text{-scope}(x)) \\
\text{is-built-in}(s\text{-attr}(x)) \lor \\
\text{is-cond-name}(s\text{-attr}(x)) \lor \text{is-ext}(s\text{-attr}(x)) \\
\text{is-auto}(s\text{-stg-cl}\cdot s\text{-attr}(x)) \lor \text{is-int}(s\text{-scope}(x))\]
is-entry(s-attr(x)) & is-int(s-scope(x)) =
is-body(s-den(x))

is-label-const(s-attr(x)) = is-st-list(s-den(x)) v
  is-format-elem-list(s-den(x))

(is-prop(variable(x)) v is-data-param(x) v is-defined(x) v
  is-based(x)) &
is-named-scalar(s-da(x)) &
  ¬is-named-cell(s-da(x)) c is-packed(s-dens(x))

(is-data-descr(x) v is-data-param-descr(x)) &
is-scalar-descr(s-da(x)) &
  ¬is-cell-descr(s-da(x)) c is-packed(s-dens(x))

is-contr-allocate(x) & (y ∈ s-is(x)) v
is-decl(x) & y ∈ s-init°s-attr(x) c
  s-id°s-first(y) = s-id(x)

is-expr(s-lbd(x)) = is-expr(s-ubd(x))

is-static(s-stg-cl(x)) v is-auto(s-stg-cl(x)) v
is-bscd(x) v is-defined(x) v
is-area(x) c (∀x)(¬is-empty(s-lbd°x(x)) &
  ¬is-empty(s-ubd°x(x)) &
  ¬is-empty(s-length°x(x)))

is-static(s-stg-cl(x)) &
(∃x')(x = s-lbd°x' v x = s-ubd°x' v x = s-length°x') &
(x(x) ⊕ O) c is-integer(x(x)), v
  is-integer(s-op°x(x)) &
  is-minus(s-operator°x(x))
is-based(x) \lor is-defined(x) \supset (\forall \kappa)(s\cdot var(x) \cdot \kappa = \Omega)

is-flt(s\cdot scale(x)) \equiv is-integer(s\cdot prec(x))

is-pict(x) \supset is-string(s\cdot da\cdot s\cdot pict\cdot spec(x))

is-area(x) \supset \neg(\exists \kappa)(is\cdot rest\cdot label(\kappa(x)))

is-const-expr(x) \supset is-real(s\cdot mode\cdot s\cdot da\cdot s\cdot op\cdot 1(x)) \land
               is-compl(s\cdot mode\cdot s\cdot da\cdot s\cdot op\cdot 2(x))

is-decl(x) \supset \neg is-empty(s\cdot ret\cdot type\cdot s\cdot attr(x))

is-file(x) \supset is-string(s\cdot da\cdot s\cdot title(x))

is-file(x) \land (y = s\cdot file\cdot attr(x)) \lor
is-open(x) \land (y = s\cdot open\cdot attr(x))
\supset
  ((STREAM \in y \mid RECORD \in y) \land
   (INPUT \in y \mid OUTPUT \in y \mid UPDATE \in y) \land
   (RECORD \in y \supset (DIRECT \in y \lor SEQUENTIAL \in y)) \land
   (DIRECT \in y \lor SEQUENTIAL \in y \lor KEYED \in y \supset RECORD \in y)) \land
   (SEQUENTIAL \in y \supset (BUFFERED \in y \lor UNBUFFERED \in y)) \land
   (BUFFERED \in y \lor UNBUFFERED \in y \supset SEQUENTIAL \in y)) \land
   (DIRECT \in y \lor KEYED \in y)) \land
   (STREAM \in y \supset \neg UPDATE \in y)) \land
   (PRINT \in y \supset STREAM \in y \land OUTPUT \in y) \land
   (EXCLUSIVE \in y \supset UPDATE \in y \land DIRECT \in y) \land
   (BACKWARDS \in y \supset INPUT \in y \land SEQUENTIAL \in y))

is-isub(x) \supset (\exists \kappa,y,i)(is\cdot defined(y) \land (s\cdot pos(y) = \Omega) \land
  x = \kappa\cdot elem(i)\cdot s\cdot arg\cdot list\cdot s\cdot base(y))
is-generic(x) & y ∈ x ⊃

¬is-empty(s-param-list·s-attr(y)) &
(∀ i)(¬is-empty(s-elem(i)·s-param-list·s-attr(y)))

is-real-format(x) ⊃
(is-fix(s-format-type(x)) ≡ s-s(x) = ∅ ) &
(is-flt(s-format-type(x)) ≡ s-p(x) = ∅ )

is-pict-format(x) ⊃ is-string(s-da·s-pict-spec(x))

is-control-format(x) ⊃ (is-page(s-format-type(x)) ≡ s-w = ∅ )

is-on-st(x) ⊃
is-emptylist(s-label-list·s-on-unit(x)) &
(∀ κ)(¬(∃ κ', κ") (κ = κ'·s-den·κ") ⊃ ¬is-return-st(κ(x)))
(is-cond-part(x) v
is-check(x)) & y ∈ s-check(x) ⊃ is-emptylist(s-arg-list(y))

is-const(x) ⊃ (is-arithm(s-da(x)) ≡ is-vr(s-vr(x)))

is-get-st(x) & is-control-format(y) & (∃ κ)(y = κ(x)) ⊃
is-space(s-format-type(y))

is-put-st(x) & is-expr(s-skip-opt(x)) ⊃
s-page-opt(x) = ∅ & s-line-opt(x) = ∅

is-put-st(x) & s-spec(x) = ∅ ⊃ s-page-opt(x) ≠ ∅ v
s-line-opt(x) ≠ ∅ v
s-skip-opt(x) ≠ ∅

is-read-st(x) & is-expr(s-ignore-opt(x)) ⊃ (s-destination(x) = ∅ ) &

s-key-keyto(x) = ∅ & s-nolock-opt(x) = ∅
is-decl(x) ⊆ ⊄ is-emptylist(s-den(x))

is-rest-label(x) ⊆ s-label-const-set ≠ { }

is-named-cell(x) ⊆ length(x) ≥ 2

is-cell-descr(x) ⊆ length(x) ≥ 2

is-spec-area(x) ⊆ ⊄ is-emptylist(s-area-descr(x))

is-init(x) ⊆ ⊄ is-emptylist(s-second(x))

is-init-iter(x) ⊆ ⊄ is-emptylist(s-item-list(x))

is-entry(x) ⊆ ⊄ is-emptylist(s-param-list(x))

is-generic(x) ⊆ x ≠ { }

is-format-item-iter(x) & is-format-elem-list(s-element(x)) ⊆ length(s-element(x)) ≥ 2

is-st-list-contr(x) ∨ is-ref-contr(x) ∨ is-expr-contr(x) ⊆ ⊄ is-emptylist(s-spec-list(x))

is-block(x) ∨ is-body(x) ∨ is-group(x) ∨ is-st-list-while(x) ∨ is-st-list-contr(x) ⊆ ⊄ is-emptylist(s-st-list(x))

is-check(x) ⊆ s-check(x) ≠ { }

is-assign-st(x) ⊆ ⊄ is-emptylist(s-lp(x))

is-const(x) ⊆ ⊄ is-emptylist(s-vr(x))
is-allocate-st(x) \lor \neg is-emptylist(x)

is-close-st(x) \lor \neg is-emptylist(x)

is-free-st(x) \lor \neg is-emptylist(s-free-list(x))

is-get-spec(x) \lor is-put-spec(x) \lor is-ref-contr(x) \lor
  \neg is-emptylist(s-data-list(x)) \land
  s-ref-set(x) \neq \{ \} \land
  \neg is-emptylist(s-format-list(x))

is-open-st(x) \lor \neg is-emptylist(x)

is-wait-st(x) \lor s-event-set(x) \neq \{ \}

is-decl(x) \land is-body(s-den(x)) \lor
  (is-empty(s-param-list\cdot s-attr(x)) \lor
   length(s-param-list\cdot s-attr(x)) =
   length(s-param-list\cdot s-den(x))) \land
   s-ret-type\cdot s-attr(x) = s-ret-type\cdot s-den(x)
5. Comments

The definitions listed in section 2 of this appendix define the Abstract Syntax of a PL/I program text as it is handled by the prepass of the interpreter. It is not to be mistaken for the program text produced by this prepass which is interpreted by all following instructions of the interpreter.

The Abstract Syntax of text describes an abstract normal form of a PL/I program. We call this normal form in the following simply a "program". This abstract program may be represented in different ways by concrete text written according to the rules of the Concrete Syntax of PL/I (e.g. the default rules for attributes of declarations imply that the same program may have different concrete representations). A concrete representation of a program may be one contingent piece of text or it may consist of different separate pieces of text (e.g. a program may contain external procedures whose concrete representations are parts of different compilations and are selected from a library).

It is assumed that a pretranslator has transformed a given representation of a PL/I program into the abstract normal form of the program. This pretranslator is described in a separate paper. The main modifications made by the pretranslator will be mentioned in the following, as far as they are not obvious.
5.1 Program

A program is a set of external procedures, including any external procedures called in the body of one of them, whether their concrete representations are contained in one compilation or not.

5.2 Block

From the sequentially ordered statement list of a block all "nonexecutable statements" - declare-, entry- and format-statements - and contained procedure blocks are extracted and separately collected in the declaration part. Then the statement list contains only executable statements, i.e. begin blocks, groups, if-statements, on-statements and simple statements.

5.3 Group

In order to avoid goto-statements leading from outside into an iterated do-group, iterated do-groups are transformed into blocks, whose statement lists are iterated.

5.4 If-statement

If an if-statement has no "else alternative" the null statement is inserted.

5.5 Declarations

Each identifier occurring in the program, whether it is declared explicitly, contextually, implicitly or by occurrence as built-in name, produces a declaration in the declaration part of that block or procedure body to which it is local. These declarations may be selected out of the declaration part by means of the identifier name. Any attributes following from default rules are inserted into the declarations.
5.6 Scope attribute

Every declaration receives one of the three attributes internal, external, or parameter. Based and defined variables, generic procedures, and label-constants receive always the attribute internal, built-in names and programmer-specified condition names the attribute external and identifiers occurring in parameter-lists the attribute parameter.

5.7 Structures

Only major structures produce declarations. Minor structures and arrays or scalars, that are successors of a structure produce no own declarations, but appear as parts of the corresponding major structure declaration. They get no own scope attribute.

The like-attribute in structure or cell declarations of the concrete text is replaced by the explicit list of successors of the declared structure or cell.

5.8 Arrays

A multi-dimensional array is decomposed into a sequence of one-dimensional arrays, which are elements of each other. So, in the abstract program, any array is one-dimensional, i.e. has one bound-pair and an element, which itself is an array (if the original array was multi-dimensional), a structure (in the case of arrays of structures) or a scalar.
5.9 Density attribute

Every level-one-variable receives one of the density attributes "packed" or "aligned". Scalars which are not parts of aggregates receive the attribute "packed".

5.10 Initial attribute

All initial attributes of parts of a data aggregate are collected into an "initial set", which contains the information about the parts to be initialized on allocation and their initial values. Included are the initial values of label array elements, which in the concrete text are given contextually by subscripted statement label prefixes. For these prefixes new names and label-constant declarations are created.

5.11 Constants

Constants (numerical constants including integers, character-constants and bit-constants) of the concrete program are transformed into the same form as used for any operand of expressions by the interpreter. This form consists of a value-representation (vr-part) and data attributes (da-part).

5.12 Precision attribute

In order to have the possibility to make use of the normal arithmetic operations defined for "full" operands (consisting of vr-part and da-part), the precision attribute of an arithmetic variable or constant is, for convenience, defined as integer or integer-pair, though only the vr-part of these integers is used. In order to avoid infinite recursivity in the definition of the predicate is-integer, one has to think of only the vr-parts of an integer-pair representing the default-precision in that definition.
5.13 Picture attribute

Picture variables are assumed to form a separate class of variables besides the classes of arithmetic variables and string variables. Their values may be converted, if possible, to one of these two classes if necessary for expression evaluation, assignment etc.

5.14 Entry declaration

Any entry name occurring in the concrete text (in a declare-statement, as entry-label or contextually in a call-statement or function call) leads to an entry declaration. If the scope of this declaration is internal (not external or parameter) the declaration has a denotation part which is the body of the corresponding procedure, restricted to the declared entry point. This body has the following parts:
1. The list of parameters which belong to the declared entry point.
2. A declaration-part, which contains all declarations of the whole procedure block to be entered through the declared entry point.
3. The condition part belonging to the procedure block to be entered by the entry point.
4. The return type declared for the individual entry point.
5. The statement-list of the procedure block starting at the declared entry point.

Note, that this is not the whole statement-list of the procedure block. Each individual entry name has its own entry declaration and especially its own body. So, in fact, different entry names leading into the same concrete procedure block are handled as if they defined individual procedures. Therefore in the abstract syntax the difference between main entries and secondary entries to a procedure does not exist.
5.15 Statement labels of executable statements

Statement labels labeling executable statements, whether they are subscripted or not (cf. 5.10), lead to declarations of label-constants. These declarations have a denotation part which is a statement-list. This statement-list is the sequence of statements, starting with the statement labeled by the declared label and finishing at the end of the local block. Thereby, if the label is situated inside one alternative of an if-statement only the statements of this alternative and not those of the other alternative, are listed and continued by the statements following the complete if-statement. If the label is situated in a not iterated do-group, the statements following the label inside the group are listed, continued by the statements following the complete group.

5.16 Statement labels of a format statement

Statement labels before format statements lead to declarations of label-constants whose denotation parts are the format lists of the corresponding format statements. A consequence of this distinct denotation part of labels of executable statements and format statements is that the interpreter will fail if a program uses labels of format statements in goto statements instead of remote formats.

5.17 Optimizing attributes

Those attributes which do not influence the results of a program, but only its effectiveness in certain implementations are neglected. These attributes are: secondary, abnormal, normal, irreducible, reducible, uses, sets. The last four, however, are used to recognize the default attribute "entry" if necessary. The same is valid for the procedure option "recursive" and other implementation-dependent procedure options.
5.18 References

Each name referencing to a part of a structure is replaced by its fully qualified name. All subscripts (which may occur distributed to the different qualification names in the concrete text) are collected into one "argument-list".

5.19 Expressions

The precedence of operators and the parenthesis hierarchy is resolved by appropriate structuring of expressions. Thereby, redundant parentheses and prefix plus-signs are removed.

5.20 Condition prefixes

The condition prefixes of blocks and statements written in a concrete text and the default condition enabling states are merged according to the structure of blocks and statements. So each block and each statement receives a condition part which denotes the complete condition enabling state valid during its execution. In order to make possible an analogous execution for all statements, groups receive a redundant condition part.

Condition prefixes of declare statements and format statements are ignored. So, when evaluating expressions occurring in declarations the condition state of the block, when evaluating expressions in format statements the condition state of the referencing get or put statement is obeyed.
5.21 Do-specifications

In the do-specifications controlling iteration of statement lists or data lists by means of a controlling variable one of two normal forms is generated by insertion of defaults. In the case of no iteration (by-clause and to-clause missing in the concrete text) the initial expression is at the same time the "to-expression" and the step is 1. If the while-clause is missing in the concrete text, the while-expression is true.

5.22 Structure allocation

In allocate statements successor names of major structures to be allocated are ignored, since these names, which have to occur in a concrete text, are either redundant or erroneous.

5.23 Array allocation

By means of the way how arrays are handled in the PL/I-model (cf. 5.8) the "dimension" (as defined in the concrete form of a program) of an array appearing in an allocate statement may be less than that appearing in the corresponding declaration.

5.24 Default file names

The default file names SYSIN and SYSPRINT, respectively, are inserted into get or put statements if necessary.
APPENDIX II : INDEX

The following list contains the names of all functions, instructions and predicates, that are defined in this document, alphabetically ordered. To the left of each name is a reference to the corresponding definition. Not contained in this document are: (1) implementation dependent functions, that are used but not formally defined in this document; (2) "abbreviations" as used in single sections of the document; (3) selector functions.

The selector functions may be recognized by the prefix s-. Since they are syntactical functions with a "local scope" and the same selector function may for different classes of argument objects have very different effects, it would have no sense to list them here. The effect of the application of a selector function to a special object may in many cases be recognized by looking at the definition of the predicate defining the corresponding object class.

3.1.1    A
3.1.1    \texttt{A}_b
3.1.1    AG
4.9.1    \texttt{allocate}(a,b,da,dens)
4.5.13   \texttt{allocate-dummy}(a,b,pa,opl)
4.3.86   \texttt{ar-ar-convert}(dat,op)
4.3.861  \texttt{ar-b-sc-conv}(dat,op,)
4.3.862  \texttt{ar-b-sc-p}(dat,op,)
4.3.863  \texttt{ar-b-sc-1}(dat,op,)
4.3.864  \texttt{ar-b-sc-2}(dat,op,)
5.8.4    \texttt{area-allocate}(a,b,ptr-gen,area,da,dens)
4.3.84   \texttt{ar-str-convert}(dat,op)
4.7     \texttt{assign}(gen,op)
5.12.8   \texttt{assign-check}(t,env,at)
5.12.8   \texttt{assign-check-list}(t,env,at)
5.7.3    \texttt{assign-list}(vl,r)
6.2.1    \texttt{assoc-complete-file-attrset(set)}
6.2.2    \texttt{associate}(te,ind,page,line,ident,new-csa,ddname)
6.2.1    \texttt{assoc-index(set)}
3.1.1    AT
6.1 attr(fid)
5.11.2 attr-const-list(adl,opl)
3.1.1 B
4.2.51 based-action(ptr,da,dens)
5.8.3 based-allocate(a,b,ptr-gen,da,dens)
4.1.11 bds-set(da)
4.3.13 BINTEGER
5.5.11 block-exit-l(ld)
6.1 bu(fid)
6.4.3 bu-assign(t,opt)
6.4.6 buffered-read(t,key,ign)
5.11.3 built-in-fct-den(id)
6.4.3 bu-into(t)
6.4.3 bu-next(t)
6.4.3 bu-next-f(t)
6.1 bu-sel(fid)
3.1.1 C
5.12.8 call-check-Cond(t)
5.12.7 c-call(t)
4.3.13 CHAR
6.1 char(fid,i)
6.1 character(fid,i)
6.1 character-sel(fid,i)
6.1 char-sel(fid,i)
4.2.45 check(da,ei)
5.9.1 check-based-level-one(gen)
5.4.21212 check-da(n,attr,dap)
5.4.21211 check-entry(n,attr,pa)
5.12.8 check-expit(env,at)
5.12.8 check-list(l-list,p-set)
6.4.1 check-multiple-key(t,key)
6.4.1 check-no-key(t,key)
5.4.213 check-ri(rt)
4.3.324 collat(v)
4.5.12 collect-aggr(t,opl)
4.5.29 collect-attr(si,opl)
4.5.26 comb-da(da,pa)
4.5.28 comb-dens(da,pa)
4.5.27 comb-dir-attr(attr,pa)
3.2.17 COMP(5)
4.2.44 compare-ei(ei1,ei2)
5.11.2 comp-ei-l(eil)
6.2.1 complete-file-attrset-list
6.4.5 con-ign(ref)
6.4.5 con-key(ref)
5.8.1 contr-initialize(insr,insd,env)
4.3.82 convert(da,op)
6.3.23 copy(rep)
4.2.524 corr-il(da,b,da0,il)
6.1 count(fid)
6.1 count-sel(fid)
3.1.1 CS
5.12.3 CSa
csa(fid)

csa-sel(fid)

4.5.23 ctr-match(attr,pa)
curr-char(fid)
curr-char-sel(fid)
curr-line(fid)
curr-line-sel(fid)
curr-page(fid)
curr-page-sel(fid)

4.5.23 da-ctr-match(da,dap)
data-elem(fid,i)
data-elem-sel(fid,i)
datum(fid,i)
datum-sel(fid,i)
ddbname(fid)
ddbname-sel(fid)

4.2.52 defined-action(gen,attr)
delete-fa(den)
delete-locking(sel)
del-exec-locate(t)
del-exec-locate(t)
del-gen(t)
del-record(t,i)
den(fid)

dim(da)
dir-bu(t,key)
dir-cbu(t,key)
direct-match(attr,pa)
dir-into(t,key)

4.5.21
4.4.3
4.4.3
dir-bu-assign(t,op)
dir-cbu-assign(t,op)
dummy-assign(<b,opl>)

elem-extent(da)
elem-index(da,rl)
end-file(fid)
end-file-sel(fid)
end-page(fid)
end-page-sel(fid)
entry-match(attr,pa)

5.2.3
4.1.2
equal(da1,da2)
equ-order(da1,da2)

4.2.511
eval-bit-1(da)
eval-da(da,env)
eval-initial(inl)

5.6

E

4.1.6
elem-extent(da)
elem-index(da,rl)

4.2.23
end-file(fid)
end-file-sel(fid)
end-page(fid)
end-page-sel(fid)
entry-match(attr,pa)

5.2.3
4.1.2
equal(da1,da2)
equ-order(da1,da2)

4.3.84
eval-bit-1(da)
eval-da(da,env)
eval-initial(inl)

5.6

E
5.6 \text{ev-ta-gen}(t)
6.4.6 \text{exclusive-read}(t, \text{key})
4.3.6.11 \text{exec-i-st}(I_{op_1}, I_{op_2}, I_{op_3})
6.4.11 \text{exec-locate}(t)
4.3.6.22 \text{exec-n-comp}(n_1, n_2, I_{op_3})
4.3.6.23 \text{exec-numeric}(I_{op_1}, I_{op_2}, I_{op_3})
4.3.6.21 \text{exec-ptr-comp}(I_{op_1}, I_{op_2}, I_{op_3})
4.3.6.22 \text{exec-single}(x_1, x_2, I_{op_3})
4.3.6.1 \text{exec-st}(I_{op_1}, I_{op_2}, I_{op_3})
6.3.14 \text{execute-iterate}(t, \text{data-list}, \text{gen}, v_2, v_3, v_4, \text{while-expr}, \text{spec-inf})
4.3.6.2 \text{exec-a-comp}(I_{op_1}, I_{op_2}, I_{op_3})
2.8.129 \text{exists}(x)
5.4.14 \text{exit}
6.3.23 \text{expand-direct}(t, \text{rep}, \text{list})
4.2.71 \text{expand-gen}(<\text{gen}, \text{ei}>)
6.3.23 \text{expand-rep}(t, \text{rep})
4.2.72 \text{expand-value}(<\text{gen}, \text{ei}>)
5.2.451 \text{ex-\&-set}(\text{id}, t_0)
6.1 \text{ext-contr}(\text{fid})
4.3.831 \text{extend}(I_{op_1}, n_1, x_1)
5.11.2 \text{extract-attr}(\text{opl})
5.2.46 \text{extr-attr}(\text{id}, t_0)
6.1 \text{ext-stor}(\text{fid})
3.1.1 F
6.1 \text{fa}(\text{fid})
5.8.2 \text{fa}(\text{al}, \text{da}, \text{dag})
3.1.1 FD
5.9.1 \text{free}(t)
4.10.2 \text{free-area}(a, i)
5.4.141 \text{free-auto}
6.4.3 \text{free-b}(t)
6.4.3 \text{free-bu}(t)
6.4.3 \text{free-Kbu}(t)
4.10.1 \text{tree-1}(\text{gen})
4.10.2 \text{tree-2}(a, i,)
6.4.2 \text{from-gen}(\text{gen})
6.4.3 \text{from-bu}(t)
6.4.2 \text{from-direct}(t, \text{key})
6.4.2 \text{from-next}(t, \text{key})
6.4.7 \text{from-rewrite}(t, \text{key})
3.1.1 \text{F}
6.1 \text{fu-elem}(\text{fid})
4.2.54 g(n)
6.3.11 \text{get-data-stream}(\text{fid})
6.3.11 \text{get-data-string}(\text{string})
6.3.11 \text{get-stream}(\text{fid}, \text{eval-format})
6.3.11 \text{get-string}(\text{fid}, \text{eval-format})
5.5.1 \text{goto-1}(\text{id})
2.8.471 \text{head}(l)
6.1 \text{header}(\text{fid})
6.1 \text{header-self}(\text{fid})
3.1.1 \( \text{ID} \)
5.2.2 \( \text{ID}_o(t_o) \)
5.4.13 \( \text{ID} \text{(dec-part)} \)
5.8.4 \( \text{in-area} \text{(area, gen)} \)
4.2.2.3 \( \text{index-set} \text{(da, rl)} \)
4.3.6 \( \text{infix-op} \text{(op}_1, \text{op}_2, \text{opor}) \)
4.2.43 \( \text{init-ei} \text{(da)} \)
4.9.24 \( \text{initial-assign} \text{(gen, op1)} \)
5.2.44 \( \text{initial-call} \text{(ep, tc)} \)
5.2 \( \text{initial-instruction} \text{(t_o, t_c)} \)
5.4.112 \( \text{initialize} \text{(is)} \)
6.2.2 \( \text{init-print} \text{(te, p, l)} \)
5.11.2 \( \text{int-aggr} \text{(t)} \)
5.8 \( \text{int-allocate} \text{(t)} \)
4.5 \( \text{int-arg} \text{(t, pa)} \)
5.11.2 \( \text{int-arg-of-built-in-fct} \text{(ad, t, ei)} \)
5.7 \( \text{int-ass-st} \text{(t)} \)
5.4.3 \( \text{int-block} \text{(t)} \)
5.4.31 \( \text{int-block-decl-part} \text{(decl-part)} \)
5.11.2 \( \text{int-built-in} \text{(t, ei)} \)
5.11.2 \( \text{int-built-in-l} \text{(adl, rf, vl, ei)} \)
5.12.7 \( \text{int-c(t)} \)
5.4.21 \( \text{int-call} \text{(attr, den, arg-list)} \)
5.4.2 \( \text{int-call-st} \text{(t)} \)
6.5.1 \( \text{int-close} \text{(te)} \)
6.5.1 \( \text{int-close-label} \text{(te, ident)} \)
6.5.1 \( \text{int-close-st(t)} \)
5.2.42 \( \text{int-contr(t)} \)
6.3.14 \( \text{int-copy} \text{(t, stream-inf)} \)
3.1.72 \( \text{int-copy-fg} \text{(new, old)} \)
6.3.14 \( \text{int-count} \text{(fid, val)} \)
5.3.32 \( \text{int-ctr-st-list} \text{(t)} \)
6.3.14 \( \text{int-data-ass} \text{(t, name, data)} \)
6.3.14 \( \text{int-data-dirt} \text{(t, spec-inf)} \)
6.3.14 \( \text{int-data-item} \text{(t, inp-inf, spec-inf)} \)
6.3.14 \( \text{int-data-list} \text{(t, data-list, spec-inf)} \)
6.3.23 \( \text{int-data-list-edit} \text{(t, elem-inf, spec-inf)} \)
6.3.14 \( \text{int-data-list-elem} \text{(t, elem, spec-inf)} \)
6.3.14 \( \text{int-data-spec} \text{(t, spec-inf)} \)
6.3.14 \( \text{int-data-stream-item} \text{(t, stream-inf, spec-inf)} \)
5.4.11 \( \text{int-decl} \text{(id, den, attr, scope)} \)
6.4.8 \( \text{int-delete} \text{(t, key)} \)
6.4.8 \( \text{int-delete-st} \text{(t)} \)
4.6.11 \( \text{integer-return} \text{(t)} \)
6.3.14 \( \text{int-elem} \text{(t, elem, spec-inf)} \)
6.3.14 \( \text{int-elem-inp} \text{(t, gen, ei, spec-inf)} \)
6.3.14 \( \text{int-end-file} \text{(t, spec-inf)} \)
6.3.23 \( \text{int-end-page} \text{(t)} \)
6.3.23 \( \text{int-end-stream} \text{(t)} \)
6.3.14 \( \text{int-end-string} \text{(spec-inf)} \)
<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.3.23</td>
<td><code>int-end-string(t, spec-inf)</code></td>
</tr>
<tr>
<td>5.2.43</td>
<td><code>int-ep(n, body)</code></td>
</tr>
<tr>
<td>6.3.14</td>
<td><code>int-expr-do(t, data-list, gen, v_2, v_3, v_4, while-expr, spec-inf)</code></td>
</tr>
<tr>
<td>6.3.14</td>
<td><code>int-expr-spec-list(t, data-list, spec-list, gen, spec-inf)</code></td>
</tr>
<tr>
<td>6.3.14</td>
<td><code>int-error(t)</code></td>
</tr>
<tr>
<td>5.6</td>
<td><code>int-ev-opt(opt, ev-g)</code></td>
</tr>
<tr>
<td>4.4.1</td>
<td><code>int-evr-op(t, ei)</code></td>
</tr>
<tr>
<td>6.2.2</td>
<td><code>int-expr-cccvv(spec, op)</code></td>
</tr>
<tr>
<td>6.3.14</td>
<td><code>int-expr-op(expr)</code></td>
</tr>
<tr>
<td>6.3.23</td>
<td><code>int-format(t, elem-inf, spec-inf, hfi)</code></td>
</tr>
<tr>
<td>6.3.14</td>
<td><code>int-format-head(rep, elem, tail)</code></td>
</tr>
<tr>
<td>5.9</td>
<td><code>int-free-st(t)</code></td>
</tr>
<tr>
<td>4.3.13</td>
<td><code>INTG</code></td>
</tr>
<tr>
<td>5.3.2</td>
<td><code>int-gen-st-list(t)</code></td>
</tr>
<tr>
<td>6.3.14</td>
<td><code>int-get-st(t)</code></td>
</tr>
<tr>
<td>5.5</td>
<td><code>int-goto-st(t)</code></td>
</tr>
<tr>
<td>6.2.2</td>
<td><code>int-ident-opt(idopt)</code></td>
</tr>
<tr>
<td>5.10</td>
<td><code>int-if(v, s_1, s_2)</code></td>
</tr>
<tr>
<td>6.4.5</td>
<td><code>int-ign(ref)</code></td>
</tr>
<tr>
<td>4.3.4</td>
<td><code>int-infix-op(r_1, r_2, opor)</code></td>
</tr>
<tr>
<td>6.3.23</td>
<td><code>int-init-tab(t, op_1, op_2)</code></td>
</tr>
<tr>
<td>4.6.1</td>
<td><code>int-integer-expr(t, env)</code></td>
</tr>
<tr>
<td>6.3.14</td>
<td><code>int-item(t, inf-inf, gen, ei, spec-inf)</code></td>
</tr>
<tr>
<td>6.4.5</td>
<td><code>int-key(ref)</code></td>
</tr>
<tr>
<td>6.4.10</td>
<td><code>int-keyfrom-locate(t, keyfrom)</code></td>
</tr>
<tr>
<td>6.4.10</td>
<td><code>int-keyfrom-locate(t, keyfrom)</code></td>
</tr>
<tr>
<td>5.3.1</td>
<td><code>int-labeled-st(t)</code></td>
</tr>
<tr>
<td>5.7.4</td>
<td><code>int-leftright(t, ei)</code></td>
</tr>
<tr>
<td>6.3.14</td>
<td><code>int-list-edit-gen(t, gen, ei, spec-inf)</code></td>
</tr>
<tr>
<td>6.4.10</td>
<td><code>int-locate(t, keyfrom)</code></td>
</tr>
<tr>
<td>6.4.10</td>
<td><code>int-locate-st(t)</code></td>
</tr>
<tr>
<td>6.4.4</td>
<td><code>int-lock(t, key)</code></td>
</tr>
<tr>
<td>4.2.1</td>
<td><code>int-name(t, env)</code></td>
</tr>
<tr>
<td>6.3.14</td>
<td><code>int-next(fid, length)</code></td>
</tr>
<tr>
<td>6.3.23</td>
<td><code>int-next-tab(t, exp-rep, tablist)</code></td>
</tr>
<tr>
<td>5.12.4</td>
<td><code>int-on-st(t)</code></td>
</tr>
<tr>
<td>6.2.2</td>
<td><code>int-open(t, te, ind)</code></td>
</tr>
<tr>
<td>6.2.2</td>
<td><code>int-open-label(te, ident)</code></td>
</tr>
<tr>
<td>6.2.2</td>
<td><code>int-open-st(t)</code></td>
</tr>
<tr>
<td>6.2.2</td>
<td><code>int-open-undf(t, ind-list)</code></td>
</tr>
<tr>
<td>6.2.2</td>
<td><code>int-opt-expr(spec, expr)</code></td>
</tr>
<tr>
<td>6.3.23</td>
<td><code>int-out(t, elem-inf, spec-inf, type, format)</code></td>
</tr>
<tr>
<td>5.6</td>
<td><code>int-pa-call(attr, den, arg-list, opt)</code></td>
</tr>
<tr>
<td>5.4.2121</td>
<td><code>int-param(id, attr)</code></td>
</tr>
<tr>
<td>4.3.7</td>
<td><code>int-prefix-op(r_1, opor)</code></td>
</tr>
<tr>
<td>6.3.23</td>
<td><code>int-print(t, ident, op)</code></td>
</tr>
<tr>
<td>6.3.23</td>
<td><code>int-printing-format(t, ident, op)</code></td>
</tr>
<tr>
<td>6.3.23</td>
<td><code>int-print-options(t, first)</code></td>
</tr>
<tr>
<td>5.4.212</td>
<td><code>int-proc-decl-part(decl-part, param-list, arg-list)</code></td>
</tr>
<tr>
<td>6.4.6</td>
<td><code>int-proper-read(t, key, ign)</code></td>
</tr>
<tr>
<td>6.4.11</td>
<td><code>int-proper-write(t, keyfrom)</code></td>
</tr>
</tbody>
</table>
5.7.4 \text{int-prop-leftpart}(t, ei)
4.8.1 \text{int-ps-leftpart}(t, ei)
6.3.23 \text{int-put-data-spec}(t, \text{spec-inf})
6.3.23 \text{int-put-element}(t, \text{elem}, \text{spec-inf})
6.3.23 \text{int-put-st}(t)
6.4.6 \text{int-read}(t, \text{key}, \text{ign})
6.4.6 \text{int-read-st}(t)
4.4.2 \text{int-ref}(t, ei)
5.4.4 \text{int-ret-st}(t)
5.12.5 \text{int-revert-st}(t)
6.4.7 \text{int-rewrite}(t, \text{key})
6.4.7 \text{int-rewrite-st}(t)
5.7.2 \text{int-sc-ass-st}(vl, \text{expr})
4.6.2 \text{int-sc-expr}(t)
6.4.3 \text{int-set-locate}(t)
5.12.6 \text{int-signal-st}(t)
6.3.23 \text{int-skip-format}(t, \text{op}, \text{new-line})
5.12.7 \text{int-snap}(t)
6.3.23 \text{int-space-format}(t, \text{elem-inf}, \text{spec-inf}, \text{blanks})
5.3.321 \text{int-spec-expr}(\text{expr})
6.3.14 \text{int-spec-part}(\text{expr})
5.3.11 \text{int-st}(t)
5.2.41 \text{int-static}(\text{attr}, b)
5.3.21 \text{int-st-list}(t)
6.3.14 \text{int-stream-get}(t)
6.3.14 \text{int-stream-item}(t, \text{stream-inf}, \text{gen, ei}, \text{spec-inf})
6.3.14 \text{int-stream-open}(t)
6.3.23 \text{int-stream-out}(t, \text{elem-inf}, \text{spec-inf})
6.3.23 \text{int-string-put}(t, \text{elem-inf}, \text{spec-inf}, \text{pair})
5.6 \text{int-task-call}(t)
6.5.1 \text{int-unable-assn}(te)
6.5.1 \text{int-unable-close}(te)
6.4.4 \text{int-unlock}(t, \text{key})
6.4.9 \text{int-unlock-st}(t)
4.2.5 \text{int-variable}(t, \text{env})
5.3.31 \text{int-while-st-list}(t)
6.4.11 \text{int-write}(t, \text{keyfrom})
6.4.11 \text{int-write-st}(t)
3.1.2 \text{is-a}
5.11.1 \text{is-a-ai}
3.1.2 \text{is-ab}
AI 3.4 \text{is-add}
AI 2 (88) \text{is-al}
5.11.1 \text{is-al-cond}
AI 3.4 \text{is-aligned}
AI 3.4 \text{is-alloc}
AI 2 (86) \text{is-allocate}
AI 2 (85) \text{is-allocate-st}
AI 3.4 \text{is-and}
AI 2 (20) \text{is-area}
AI 2 (93) is-area-allocate
4.10.2 is-area-loc(a,i,s)
AI 3.4 is-area-size
AI 2 (81) is-arg
5.11.1 is-arg-int
AI 2 (13) is-arithm
4.3.12 is-arithop
4.1.1 is-array
AI 2 (90) is-array-al
AI 2 (26) is-array-descr
AI 3.4 is-assign
AI 2 (71) is-assign-st
AI 2 (6) is-attr
AI 3.4 is-auto
3.1.2 is-b
AI 3.4 is-backwards
AI 2 (41) is-based
AI 2 (92) is-based-allocate
AI 3.4 is-bin
AI 3.4 is-bit
4.1.4 is-bit-class(da)
4.3.12 is-bitop
AI 2 (57) is-block
AI 2 (3) is-body
AI 3.4 is-buffered
AI 3.4 is-built-in
5.11.1 is-built-in-fct-den
3.2.2 is-c
5.11.1 is-c-ai
AI 3.4 is-call
AI 2 (94) is-call-st
AI 3.4 is-cat
AI 2 (28) is-cell-descr
AI 3.4 is-char
4.1.4 is-char-class(da)
AI 2 (65) is-check
AI 3.4 is-cl
AI 2 (97) is-close
AI 2 (96) is-close-st
AI 3.4 is-column
4.3.12 is-comp
AI 3.4 is-compl
AI 2 (49) is-compl-format
6.3.11 is-concr-rep
AI 2 (64) is-cond
5.11.1 is-cond-built-in-fct-den
AI 3.4 is-cond-name
AI 2 (55) is-cond-part
3.1.35 is-cond-sel
4.2.33 is-connected(gen)
AI 2 (82) is-const
AI 2 (33) is-const-expr
AI 2 (87) is-contr-allocate
AI 2 (52) is-control-format
AI 2 (53) is-control-format-type
5.11.1 is-conv-arg-fct
4.2.522 is-corresp(da_1, da_2)
3.1.35 is-cs
AI 3.4 is-ctr
3.1.31 is-d
AI 2 (23) is-da-descr
AI 2 (22) is-data-descr
AI 3.4 is-data-dir
3.1.74 is-data-elem
AI 2 (47) is-data-format
AI 3.4 is-data-name
AI 2 (34) is-data-param
AI 2 (37) is-data-param-descr
3.1.74 is-dataset
6.3.11 is-data-stream-ass
6.3.11 is-data-stream-item
6.3.11 is-data-string-ass
6.3.11 is-data-string-item
3.1.74 is-datum
AI 3.4 is-dec
AI 2 (5) is-decl
AI 2 (4) is-decl-part
AI 3.4 is-default-area
AI 2 (40) is-defined
AI 3.2 is-def-prec
AI 3.4 is-delay
AI 2 (98) is-delay-st
AI 3.4 is-delete
AI 2 (99) is-delete-st
AI 3.4 is-direct
AI 3.4 is-display
AI 2 (100) is-display-st
AI 3.4 is-div
AI 2 (60) is-do-spec
3.1.32 is-e
5.11.1 is-e-ai
AI 3.4 is-edit-dir
6.3.11 is-edit-stream
6.3.11 is-edit-stream-item
6.3.11 is-edit-string
6.3.11 is-edit-string-item
4.2.21 is-elem-part(da,r1)
AI 3.4 is-empty
AI 3.4 is-empty-list
AI 3.4 is-endfile
AI 3.4 is-endpage
AI 2 (35) is-entry
AI 2 (43) is-entry-decl
AI 3.4  is-eq
AI 3.4  is-error
3.1.74  is-es
AI 3.4  is-event
5.5.1   is-evs
AI 3.4  is-exclusive
4.2.41  is-exhaust(ei)
AI 3.4  is-exit-st
AI 2 (72) is-expr
AI 2 (120) is-expr-contr
AI 2 (119) is-expr-elem
AI 3.4  is-ext
3.1.74  is-ext-contr
AI 2 (2) is-ext-proc
3.1.74  is-ext-stor
3.1.2   is-f
3.1.72  is-fa
5.11.1  is-f-ai
3.1.72  is-fd
AI 2 (38) is-file
AI 2 (39) is-file-attr
AI 3.4  is-finish
AI 3.4  is-fix
AI 3.4  is-flt
AI 3.4  is-fofl
AI 2 (44) is-format-elem
AI 2 (46) is-format-item
AI 2 (45) is-format-item-iter
AI 3.4  is-free
AI 2 (101) is-free-st
3.1.73  is-fu
3.1.73  is-fu-elem
5.11.1  is-g-ai
AI 3.4  is-ge
3.1.6   is-gen
AI 2 (42) is-generic
AI 3.4  is-get
AI 2 (108) is-get-spec
AI 2 (105) is-get-st
AI 3.4  is-goto
AI 2 (111) is-goto-st
AI 2 (61) is-group
6.3.11  is-group-end
AI 3.4  is-gt
AI 2 (62) is-if-st
6.2.1   is-incompl-file-attrset(set,i)
AI 2 (73) is-infix-expr
AI 2 (74) is-infix-operator
AI 2 (29) is-init
AI 2 (32) is-init-const
AI 2 (30) is-init-item
AI 2 (31)  is-init-iter
AI 3.4  is-input
AI 3.4  is-int
AI 2 (83)  is-integer
AI 2 (122)  is-into-opt
AI 2 (67)  is-io-cond
AI 3.2  is-io-opt
3.1.35  is-io-sel
AI 2 (84)  is-isub
4.1.3  is-isub-def(attr)
AI 3.4  is-key
AI 3.4  is-keyed
AI 2 (124)  is-key-opt
AI 2 (125)  is-keyto-opt
AI 2 (17)  is-label
AI 3.4  is-label-const
5.5.1  is-label-denotation
AI 3.4  is-le
AI 2 (102)  is-level-one-ref
AI 3.4  is-line
AI 3.4  is-list-dir
6.3.11  is-list-stream
6.3.11  is-list-stream-item
6.3.11  is-list-string
6.3.11  is-list-string-item
AI 3.4  is-locate
AI 2 (112)  is-locate-st
6.4.4  is-locked(fid, key)
3.1.73  is-locking
AI 3.4  is-lt
AI 3.4  is-minus
2.8.122  is-mp(吸纳
AI 3.4  is-mult
3.1.2  is-n
AI 3.1  is-name
AI 2 (11)  is-named-array
AI 2 (19)  is-named-cell
AI 2 (8)  is-named-data
AI 2 (66)  is-named-io-cond
AI 2 (12)  is-named-scalar
AI 2 (9)  is-named-struct
AI 2 (10)  is-named-succ
AI 3.4  is-ne
3.1.35  is-n-io-cond
AI 3.4  is-not
AI 3.4  is-null
AI 3.4  is-null-st
5.11.1  is-o-ai
AI 3.4  is-of1
AI 2 (63)  is-on-st
AI 3.4  is-op
AI 2 (114) is-open
AI 2 (113) is-open-st
AI 2 (15) is-opt
AI 3.4 is-or
AI 3.4 is-output
4.2.521 is-overlay(gen, attr)
AI 3.4 is-packed
AI 3.4 is-page
2.8.472 is-pair(x)
AI 2 (95) is-pa-opt
AI 3.4 is-para
AI 2 (36) is-param-descr
AI 2 (16) is-pict
AI 2 (51) is-pict-format
AI 3.4 is-picture
AI 3.4 is-power
AI 2 (75) is-prefix-expr
AI 2 (76) is-prefix-operator
AI 2 (77) is-print-expr
AI 3.4 is-print
3.1.74 is-printable-datum
AI 2 (1) is-program
AI 2 (68) is-progr-named-cond
AI 2 (70) is-prop-st
AI 2 (7) is-prop-variable
4.3.811 is-ps-arithm
4.3.811 is-ps-string
AI 3.4 is-ptr
AI 2 (104) is-ptr-qual-level-one-ref
AI 2 (79) is-ptr-qual-ref
AI 3.4 is-put
AI 2 (118) is-put-spec
AI 2 (115) is-put-st
5.11.1 is-r-ai
AI 3.4 is-read
AI 2 (121) is-read-st
AI 3.4 is-real
AI 2 (48) is-real-format
AI 3.4 is-record
6.4.2 is-record-ref(ref)
AI 3.4 is-record-size
AI 2 (78) is-ref
AI 2 (110) is-ref-contr
AI 2 (109) is-ref-elem
AI 3.4 is-remote
AI 2 (54) is-remote-format
5.11.1 is-res-attr-fct
5.11.1 is-res-fct
AI 2 (18) is-rest-label
3.1.34 is-ret-point
AI 3.4 is-return
AI 2 (126) is-return-st
AI 3.4 is-revert
AI 2 (127) is-revert-st
AI 3.4 is-rewrite
AI 2 (128) is-rewrite-st
3.1.34 is-ri
5.11.1 is-s-ai
4.1.1 is-scalar
AI 2 (27) is-scalar-descr
4.2.22 is-scalar-part(da,rl)
AI 3.4 is-sequential
AI 2 (123) is-set-opt
AI 3.4 is-signal
AI 2 (129) is-signal-st
5.11.1 is-simple-built-in-fct-den
3.1.35 is-simple-check-ref
AI 2 (103) is-simple-name
AI 2 (80) is-simple-ref
AI 2 (69) is-simple-st
AI 3.4 is-size
AI 3.4 is-skip
AI 3.4 is-space
AI 2 (21) is-spec-area
6.3.12 is-spec-inf
AI 2 (56) is-st
3.1.1 is-state
AI 3.4 is-static
AI 2 (59) is-st-list-contr
AI 2 (58) is-st-list-while
AI 3.4 is-stop-st
AI 3.4 is-stream
AI 2 (106) is-stream-get
AI 2 (116) is-stream-put
AI 2 (14) is-string
AI 2 (91) is-string-al
AI 2 (50) is-string-format
AI 2 (107) is-string-get
AI 2 (117) is-string-put
4.1.1 is-struct
AI 2 (89) is-struct-al
AI 2 (24) is-struct-descr
AI 3.4 is-subrg
AI 3.4 is-subtr
AI 2 (25) is-succ-descr
AI 3.2 is-system-action
AI 3.4 is-task
AI 3.4 is-transmit
2.8.472 is-triple(x)
AI 3.4 is-true
4.2.6 is-sub-action(n,rl,ei)
4.2.62 is-sub-check(da,rl,ei)
isub-env (env, isub-list)
isub-list (da, rl, ei)
is-ufl
is-unbuffered
is-undf
is-unlock
is-unlock-st
is-unrest-label
is-update
is-v-ai
is-variable (attr)
is-vr
is-wait
is-wait-st
is-write
is-write-st
is-zdiv
iterate-ass-st (t, ei)
iterate-contr-var (t, data-list, gen, v2, v3, v4, while-expr, spec-inf)
iterate-do (t, gen, v2, v3, v4, expr, tc)
iterate-initial-assign (gen, opl, ei)
iterate-spec-list (t, spec-list, gen, tc)
iterate-while (t, v, tc)
K0 (t0)

kbu (fid)
kbu-keyto (t)
kbu-next-fi (t, key)
kbu-sel (fid)
key (fid, i)
keyed-access (t, key)
keyfrom-kbu (t, keyfrom)
keyfrom-next (t, keyfrom)
key-sel (fid, i)

length (l)
line (fid, i)
line-sel (fid, i)
linesize (fid)
linesize-sel (fid)
locate-exec (fid)
locate-exec-sel (fid)
lock (t, key)
locking (fid)
locking-sel (fid)
make-all-attr (al, da, gen)
make-csa (open-attrset, attrset)
make-rl (da, subs-list, qual-list)
make-stream-end-rep (type)
make-stream-rep (elem-inf, type, format, print)
make-string-end-rep (type)
4.9.21 \texttt{make-vl(lin,env)}
4.5.24 \texttt{match-l(dar,dap)}
4.5.25 \texttt{match-2(da,dap)}
6.4.3 \texttt{m-based-allocate(a,b,ptr-gen,da,dens,t)}
6.4.3 \texttt{m-bu(t,op)}
6.4.3 \texttt{m-bu-gen(da,dens,a,t)}
6.4.3 \texttt{m-cbu(t,op)}
4.2.512 \texttt{mda(da_1,da_2)}
4.2.63 \texttt{merge(r1,r2)}
5.5.1 \texttt{m-evs(e,d)}
4.9.12 \texttt{m-gen(da,dens,a)}
4.2.513 \texttt{mil(index,da)}
5.11.2 \texttt{mk-ad1(ai1,ai)}
6.4.3 \texttt{m-kbu(t,keyfrom)}
6.4.3 \texttt{m-kbu-gen(da,dens,a,t)}
5.6 \texttt{mk-pa(aegument)}
4.8.11 \texttt{mk-sg-gen(id,ol,il)}
5.6 \texttt{mk-\text{\$}\in\gamma(e,attr,dens,arg-list)}
6.4.8 \texttt{m-open-delete(t)}
6.4.10 \texttt{m-open-locate(t)}
6.4.6 \texttt{m-open-read(t)}
6.4.7 \texttt{m-open-rewrite(t)}
6.4.9 \texttt{m-open-unlock(t)}
6.4.11 \texttt{m-open-write(t)}
4.1.9 \texttt{m-si(da)}
3.1.1 \texttt{N}
6.1 \texttt{n(fid)}
4.2.31 \texttt{new-da(da,rl)}
6.1 \texttt{new-fa(fid)}
4.2.32 \texttt{new-il(da,il,rl)}
6.4.1 \texttt{new-next(t,key)}
6.1 \texttt{next(fid)}
6.4.1 \texttt{next(t,r,ign)}
6.4.1 \texttt{next-backwards(t,r,ign)}
6.4.3 \texttt{next-bu(t)}
6.4.3 \texttt{next-cbu(t)}
6.4.2 \texttt{next-into(t)}
6.4.3 \texttt{next-kbu(t)}
6.4.2 \texttt{next-keyto(t)}
6.1 \texttt{next-sel(fid)}
4.2.12 \texttt{nu(da,id)}
4.11.1 \texttt{null}
4.9.11 \texttt{null-allocate(b)}
4.3.322 \texttt{num-val(op)}
6.1 \texttt{old-ta(fid)}
5.12.4 \texttt{on-den(t,env)}
5.12.5 \texttt{on-revert(x,t)}
5.12.4 \texttt{on-update(x,t)}
6.4.11 \texttt{optk-from-bu-next(t,keyfrom)}
4.1.5 \texttt{order(da)}
6.1 \texttt{paget(fid,i)}
6.1 \text{page-sel}\,(\text{fid}, \text{i})
6.1 \text{pagesize}\,(\text{fid})
6.1 \text{pagesize-sel}\,(\text{fid})
4.11.2 \text{pass}\,(\text{x})
4.5.11 \text{pass-gen}\,(\text{gen}, \text{da}_{\text{g}})
5.3.32 \underline{\text{pass-gen-part}}\,(\text{r})
4.2.66 \text{pass-isub}\,(\text{gen}, \text{ei})
4.6.2 \text{pass-op}\,(\text{r})
4.2.53 \text{pass-sub-gen}\,(\text{gen}, \text{rl})
6.4.3 \text{p-by}\,(\text{a}, \text{b}, \text{da}_{\text{a}}, \text{dens}, \text{t})
6.4.3 \text{p-kbu}\,(\text{a}, \text{b}, \text{da}_{\text{a}}, \text{dens}, \text{t})
4.9.3 \text{pointer}\,(\text{gen})
5.4.142 \text{pop-up}
4.2.523 \text{post-attr}
4.3.511 \text{pref-a}_{\text{b}}\,(\text{da}_{\text{a}}, \text{da}_{\text{b}})
4.3.512 \text{pref-a}_{\text{m}}\,(\text{da}_{\text{a}}, \text{da}_{\text{m}}, \text{opor})
4.3.513 \text{pref-a}_{\text{a}}\,(\text{da}_{\text{a}}, \text{da}_{\text{b}}, \text{opor})
4.3.521 \text{prefer-s}_{\text{b}}\,(\text{da}_{\text{a}}, \text{da}_{\text{b}})
4.3.522 \text{prefer-s}_{\text{a}}\,(\text{da}_{\text{a}}, \text{da}_{\text{b}})
5.4.211 \underline{\text{pre-int-params}}\,(\text{param-list}, \text{arg-list}, \text{decl-part})
5.2.4 \text{prepass-execute}\,(\text{t}_{\text{o}}, \text{t}_{\text{c}}, \text{iv}, \text{evf}, \text{ep})
5.6 \text{priority-assign}\,(\text{expr}, \text{loc(a,}i_{\text{a}}))
5.11.2 \text{p-s-dec-int-const}\,(\text{t})
4.8.21 \text{pseudo-assign}\,(\text{ps-gen}, \text{op}) \quad \text{for REAL}
4.8.22 \text{pseudo-assign}\,(\text{ps-gen}, \text{op}) \quad \text{for IMAG}
4.8.23 \text{pseudo-assign}\,(\text{ps-gen}, \text{op}) \quad \text{for SUBSTR}
4.8.24 \text{pseudo-assign}\,(\text{ps-gen}, \text{op}) \quad \text{for EVENT}
6.4.10 \text{put-keyed-locate}\,(\text{t})
6.3.23 \text{put-stream}\,(\text{t}, \text{spec-inf}, \text{rep})
6.3.21 \text{put-string}\,(\text{pair}, \text{rep})
6.4.10 \text{put-unkeyed-locate}\,(\text{t})
5.12.8 \text{raise-endfile-cond}\,(\text{t})
5.12.8 \text{raise-endpage-cond}\,(\text{t})
5.12.8 \text{raise-key-cond}\,(\text{t})
5.12.8 \text{raise-npare-cond}\,(\text{t})
5.12.8 \text{raise-record-cond}\,(\text{t})
5.12.8 \underline{\text{raise-transmit-cond}}\,(\text{t})
5.12.8 \underline{\text{raise-undef-cond}}\,(\text{t})
6.4.6 \text{read-buffered-into}\,(\text{t}, \text{key})
6.4.6 \text{read-buffered-set}\,(\text{t}, \text{key})
6.1 \text{rec-length}\,(\text{fid}, \text{i})
6.1 \text{rec-length-sel}\,(\text{fid}, \text{i})
6.4.2 \text{record-assign}\,(\text{gen}, \text{rec})
4.2.12 \text{ref-list}\,(\text{da}, \text{subs-list}, \text{qual-list})
4.2.12 \text{ref-list-1}\,(\text{da}, \text{qual-list})
4.2.12 \text{ref-list-2}\,(\text{da}, \text{subs-list}, \text{qual-list})
4.3.321 \text{represent}\,(\text{da}, \text{v})
4.3.63 \text{res-ar-da}\,(\text{da}_{\text{a}}, \text{da}_{\text{b}}, \text{opor})
4.3.63 \text{res-ar-prec}\,(\text{da}_{\text{a}}, \text{da}_{\text{b}}, \text{opor})
4.2.64 \text{rest-list}\,(\text{da}, \text{rl}, \text{ei})
6.3.23 \text{ret-exp-rep}\,(\text{exp-rep})
5.4.41 \text{return}\,(\text{r})
6.3.14 return-op(val)
3.1.1 RI
2.8.122 rp(κ₁, κ₂)
3.1.1 S
4.1.7 scalar-extent(da)
4.2.24 scalar-index(da,rl)
4.2.43 SC-EI
6.2.2 select-alternat(impl,expl)
6.2.2 select-value(opt,ident)
6.4.1 sequential-access(t,key)
4.5.261 si-match(s₁,s₂)
4.3.841 single-a-b-conv(i,n₁)
4.3.832 single-b-c-conv(vr)
4.3.851 single-c-a-conv(op)
4.3.832 single-c-b-conv(vr)
5.12.1 spp-update(t)
5.4.12 stack(ε)
6.4.2 store(t,m,rec)
6.4.2 store-K(t,m,key)
4.3.85 str-ar-convert(da₆,op)
4.3.323 string-val(op)
4.3.83 str-str-convert(da₆,op)
4.2.3 sub-gen(gen,rl)
5.2.45 subst-den(t₀,iv,evf,ep)
3.2.2 T(ε)
5.6 ta-c-assign(opt,loc(a,i₀))
2.8.471 tail(t)
4.3.5 target(da₆,da₇,op)
6.1 trailer(fid)
6.1 trailer-sel(fid)
6.2.2 translate-char(user-label)
6.2.2 translate-ddname(op)
6.2.2 translate-user-label(op)
4.3.87 truth(op)
4.11.3 un-a
4.11.3 un-ab
4.11.3 un-b
6.4.6 unbuffered-read(t,key,ign)
4.11.3 un-f
4.4.4 unlock(t,key)
4.11.3 un-n
4.2.42 update(ei)
6.3.23 update-count(t,spec-inf)
6.3.23 update-data-elem-list(t,rep)
5.4.11 update-dn-at(id,dn,attr)
5.4.15 update-env
6.3.14 update-es-part(selector,object)
6.2.2 update-fd-copen(den,f)
6.4.1 update-fu-open(f,csa,ddname)
6.3.14 update-fu-part(selector,object)
6.4.1 update-next(t,key,ign)
6.2.2 \texttt{update-\textit{states}(f,te,page,line,ident,csa,ddname)}
6.3.14 \texttt{update-\textit{stream}(t,stream-inf,spec-inf)}
4.2.73 \texttt{value(gen,S)}
4.9.23 \texttt{v1(in1)}
2.8.127 \texttt{\delta(b, \chi_1)}
3.2.15 \texttt{\Lambda(\xi_d)}
2.8.126 \texttt{\mu(\gamma)}
3.2.2 \texttt{q(\tau)}
3.2.13 \texttt{\Phi(\xi, instr, \tau)}
3.2.14 \texttt{\Psi(\xi, \tau)}