

**BLUE**

Preliminary Design Phase Report

15 February 1978

Contract MDA903-77-C-0324

## Table of Contents

1. INTRODUCTION .....	1-1
1.1. IRONMAN ANALYSIS .....	1-1
1.1.1. General Design Criteria .....	1-1
1.1.2. General Syntax .....	1-1
1.1.3. Types .....	1-2
1.1.3.1. Numeric Types .....	1-3
1.1.3.2. Enumeration Types .....	1-4
1.1.3.3. Composite Types .....	1-5
1.1.3.4. Set Types .....	1-6
1.1.3.5. Encapsulated Definitions .....	1-6
1.1.4. Expressions .....	1-7
1.1.5. Constants, Variables, and Declarations .....	1-7
1.1.6. Control Structures .....	1-9
1.1.7. Functions and Procedures .....	1-10
1.1.8. Input-Output Facilities .....	1-11
1.1.9. Parallel Processing .....	1-12
1.1.10. Handling .....	1-13
1.1.11. Object Representation .....	1-14
1.1.12. Libraries, etc. ....	1-15
1.1.13. Support for the Language .....	1-16
1.2. SUGGESTED CHANGES .....	1-17
1.2.1. Aliasing .....	1-17
1.2.2. Nested Routines and Modules .....	1-18
1.2.3. Segments .....	1-18
1.2.4. Variant Records .....	1-18
1.2.5. Functions with Side-Effects .....	1-19
1.2.6. Fixed Point .....	1-19
2. LEXICAL STRUCTURE OF THE LANGUAGE .....	2-1
2.1. OVERVIEW .....	2-1
2.1.1. Character Set .....	2-1
2.1.2. Tokens .....	2-1
2.1.3. Identifiers .....	2-1
2.1.4. Literals .....	2-1
2.1.5. Other Symbols .....	2-1
2.1.6. Reserved Words .....	2-1
2.1.7. Key Words .....	2-1
2.1.8. Comments .....	2-1
3. TYPES AND OPERATORS .....	3-1
3.1. OVERVIEW .....	3-1
3.1.1. Declarations and Attributes .....	3-1
3.1.1.1. Type Attributes .....	3-1
3.1.1.2. Constant Declarations .....	3-1
3.1.1.3. Variable Declarations .....	3-1
3.1.1.4. Type Specifications .....	3-2
3.1.2. Literals and Constructors .....	3-3
3.1.3. Representation Specifications .....	3-3
3.1.4. Predefined Operators .....	3-3
3.1.4.1. Operators and Expressions .....	3-3
3.1.4.2. Operator Overview .....	3-5
3.1.5. Assignment .....	3-5
3.1.6. Predefined Functions .....	3-6

## Table of Contents

3.2. FIXED POINT .....	3-7
3.2.1. Attributes and Type Specification .....	3-8
3.2.2. Literals .....	3-10
3.2.3. Representation Specification .....	3-11
3.2.4. Operations and Expressions .....	3-12
3.2.4.1. Rounding Control .....	3-12
3.2.4.2. Infix Operators .....	3-13
3.2.4.3. Prefix Operators .....	3-13
3.2.4.4. Scaling Infix Operators .....	3-13
3.2.4.5. Machine-Dependent Infix Operators .....	3-14
3.2.4.6. Relational Infix Operators .....	3-15
3.2.5. Assignment .....	3-15
3.2.6. Predefined Functions .....	3-15
3.2.6.1. TO_FIXED .....	3-15
3.2.6.2. ABS .....	3-16
3.2.6.3. FIXED_MIN and FIXED_MAX .....	3-16
 3.3. FLOATING POINT .....	3-17
3.3.1. Attributes and Type Specification .....	3-17
3.3.2. Literals .....	3-17
3.3.3. Representation Specifications .....	3-17
3.3.4. Operators and Expressions .....	3-18
3.3.4.1. Rounding Control .....	3-18
3.3.4.2. Infix Operators .....	3-19
3.3.4.3. Prefix Operators .....	3-20
3.3.4.4. Precision Control .....	3-20
3.3.4.5. Machine-Dependent Infix Operators .....	3-21
3.3.4.6. Relational Infix Operators .....	3-21
3.3.5. Assignment .....	3-21
3.3.6. Predefined Functions .....	3-21
3.3.6.1. TO_FLOAT .....	3-21
3.3.6.2. ABS .....	3-22
3.3.6.3. FLOAT_MIN and FLOAT_MAX .....	3-22
3.3.6.4. EPSILON .....	3-22
3.3.6.5. IMP_PRECISION .....	3-22
 3.4. ENUMERATED TYPES .....	3-23
3.4.1. Attributes and Type Specifications .....	3-23
3.4.2. Literals .....	3-24
3.4.3. Representation Specification .....	3-24
3.4.4. Operators and Expressions .....	3-24
3.4.4.1. Infix Operators .....	3-24
3.4.4.2. Prefix Operators .....	3-24
3.4.5. Assignment .....	3-24
3.4.6. Predefined Functions .....	3-24
3.4.6.1. SUCC and PRED .....	3-24
 3.5. BOOLEAN .....	3-25
3.5.1. Attributes and Type Specification .....	3-25
3.5.2. Literals .....	3-25
3.5.3. Representation Specifications .....	3-25
3.5.4. Operators and Expressions .....	3-25
3.5.4.1. Infix Operators .....	3-25
3.5.4.2. Prefix Operators .....	3-25
3.5.5. Assignment .....	3-26
3.5.6. Predefined Functions .....	3-26
3.5.6.1. HANDLER_EXISTS .....	3-26
3.5.6.2. IS_MANIFEST .....	3-26

## Table of Contents

3.6. BIT STRINGS .....	3-27
3.6.1. Attributes and Type Specification .....	3-27
3.6.2. Literals .....	3-27
3.6.3. Representation Specification .....	3-27
3.6.4. Operators and Expressions .....	3-27
3.6.4.1. Infix Operators .....	3-27
3.6.4.2. Prefix Operators .....	3-28
3.6.5. Assignment .....	3-28
3.6.6. Predefined Functions .....	3-28
3.6.6.1. TO_BIT .....	3-28
3.6.6.2. DUP .....	3-29
3.6.6.3. ZEROS .....	3-29
3.6.6.4. BIN_OF, OCT_OF, and HEX_OF .....	3-29
3.7. CHARACTER STRINGS .....	3-30
3.7.1. Attributes and Type Specification .....	3-30
3.7.2. Literals .....	3-30
3.7.3. Representation Attributes .....	3-31
3.7.4. Operators and Expressions .....	3-31
3.7.4.1. Infix Operators .....	3-31
3.7.5. Assignment .....	3-31
3.7.6. Predefined Functions .....	3-31
3.7.6.1. DUP .....	3-31
3.7.6.2. BLANKS .....	3-32
3.8. ARRAYS .....	3-33
3.8.1. Attributes and Type Specification .....	3-33
3.8.2. Array Constructors .....	3-34
3.8.3. Representation Specifications .....	3-35
3.8.4. Operators and Expressions .....	3-35
3.8.4.1. Infix Operators .....	3-36
3.8.5. Assignment .....	3-36
3.9. RECORD TYPES .....	3-37
3.9.1. Attributes and Type Specification .....	3-37
3.9.2. Record Constructors .....	3-41
3.9.3. Representation Specification .....	3-41
3.9.4. Operators and Expressions .....	3-41
3.9.4.1. Infix Operators .....	3-41
3.9.5. Assignment .....	3-41
3.10. POINTER TYPES .....	3-43
3.10.1. Attributes and Type Specification .....	3-44
3.10.2. Literals and Constructors .....	3-44
3.10.3. Representation Specification .....	3-44
3.10.4. Operators and Expressions .....	3-44
3.10.4.1. Infix Operators .....	3-44
3.10.4.2. Other Operators .....	3-44
3.10.5. Assignment .....	3-44
3.11. SEMAPHORES .....	3-45
3.11.1. Attributes and Type Specification .....	3-45
3.11.2. Constructors .....	3-45
3.11.3. Representation Specifications .....	3-45
3.11.4. Operators and Expressions .....	3-46
3.11.5. Assignment .....	3-46
3.11.6. Predefined Routines .....	3-46

## Table of Contents

3.12. USER-DEFINED TYPES .....	3-47
3.12.1. Type Declarations .....	3-47
3.12.2. Type Specifications .....	3-47
3.12.3. Representation Declarations .....	3-48
3.12.4. Representation Specifications .....	3-48
3.12.5. Abstract Types .....	3-48
4. PROGRAM STRUCTURE .....	4-1
4.1. SCOPE RULES .....	4-1
4.1.1. Importing .....	4-1
4.1.2. Exporting .....	4-2
4.2. SEGMENT DECLARATION .....	4-3
4.3. ROUTINES .....	4-4
4.3.1. Routine Declarations .....	4-4
4.3.2. Routine Invocation .....	4-6
4.3.3. Aliasing .....	4-7
4.3.4. Side Effects .....	4-7
4.4. MODULE DECLARATION .....	4-9
4.5. CONTROL STATEMENTS .....	4-10
4.5.1. Statements .....	4-10
4.5.2. Simple Statements .....	4-10
4.5.3. Structured Statements .....	4-10
4.5.4. With Clauses .....	4-10
4.5.5. Conditional Statements .....	4-11
4.5.5.1. If Statements .....	4-11
4.5.5.2. Case Statements .....	4-11
4.5.6. Repetitive Statements .....	4-12
4.5.6.1. Indefinite Loop Statement .....	4-12
4.5.6.2. EXIT Statement .....	4-12
4.5.6.3. Definite Loop Statement .....	4-13
4.5.7. Exception Handling .....	4-13
4.5.8. ASSERT Statement .....	4-15
4.6. PARALLEL PROCESSING .....	4-17
4.6.1. Parallel Blocks .....	4-17
4.6.2. WAIT Statement .....	4-18
4.6.3. TERMINATE Statement .....	4-18
4.6.4. PRIORITY Statement .....	4-18
4.6.5. CLOCK Function .....	4-18
4.6.6. REQUEST Statement .....	4-18
4.6.7. RELEASE Statement .....	4-18
4.6.8. Examples .....	4-18
4.6.9. Simulation .....	4-18
4.7. LIBRARIES .....	4-19
4.8. MACHINE DEPENDENT PROGRAMMING .....	4-20
4.8.1. Low-Level Input-Output .....	4-20
4.8.2. Target Routine Declarations .....	4-20
5. GENERIC DEFINITIONS .....	5-1

## Table of Contents

5.1. ROUTINE PARAMETERS .....	5-3
5.2. WHERE CLAUSE .....	5-3
A. IMPLEMENTATION ASSESSMENT .....	A-1
A.1. DEFINES/USES IMPACT .....	A-1
A.2. DATA TYPE OPTIMIZATION .....	A-2
A.3. ALIASING AND SIDE-EFFECT RULES ENFORCEMENT .....	A-2
A.3.1. Enforcement .....	A-2
A.3.2. Implementation Problems and Impact .....	A-3
A.3.2.1. Aliasing and Related Restrictions .....	A-3
A.3.2.2. Maintaining Program Information .....	A-4
A.3.2.3. Impact on IRONMAN Goals .....	A-5
A.3.3. Implementation Approach .....	A-5
A.3.3.1. Interprocedural Data Flow Analysis .....	A-5
A.3.3.2. Impact on Optimization .....	A-6
A.4. CONCURRENT PROCESSING EFFICIENCY .....	A-6
A.5. DATA SPACE MANAGEMENT .....	A-7
A.6. DEFINITION OF COMPILE TIME EXPRESSION RESTRICTIONS ....	A-9
A.6.1. Introduction .....	A-9
A.6.2. Approaches to Arithmetic .....	A-9
A.6.3. User Supplied Functions .....	A-10
A.6.4. Built-In Functions .....	A-10
A.7. IMPLEMENTATION ESTIMATES .....	A-11

## 1. INTRODUCTION

In this report, we discuss the design decisions reflected in the accompanying language specification. In Section 1.1, we analyze the specific IRONMAN requirements and how our language satisfies them. In Section 1.2, we discuss possible simplifications to the language that would result from modifying certain IRONMAN requirements. In Sections 2 through 5, we discuss specific design decisions, as referenced in the Specification. In Appendix A, we present an assessment of the implementation difficulties posed by the language. Throughout this and subsequent Sections, we use the notation Sx.y.z to refer to a Section of the Language Specification and Jx.y.z to refer to a Section of this "Justification" document.

### 1.1. IRONMAN ANALYSIS

#### 1.1.1. General Design Criteria

We have used the General Design Criteria as constant guides in interpreting the specific requirements. A detailed analysis of how we satisfy these general criteria is not provided since the requirements are non-specific. We feel that we satisfy the General Design Criteria to the extent permitted by the specific requirements presented in subsequent sections.

#### 1.1.2. General Syntax

##### 2A. Character Set (SATISFIED)

The language is limited to the 64 character ASCII subset, as Section 2.1.1 shows.

##### 2B. Grammar (SATISFIED)

The syntax is free form and uses notations derived from PASCAL, where appropriate.

##### 2C. Syntactic Extensions (SATISFIED)

Precedence rules and syntactic forms of the language are fixed by the definition.

##### 2D. Other Syntactic Issues (SATISFIED)

S2.1.2 shows that program line boundaries are treated as blanks. Keyword forms containing declarations or statements are bracketed.

2E. Mnemonic Identifiers (SATISFIED)

There is no upper limit on identifier length. The underscore is used as a break character (S2.1.3). No abbreviation method is provided for identifiers or reserved words.

2F. Reserved Words (SATISFIED)

Reserved words are listed in S2.1.6. They are limited to those introducing and terminating syntactic constructs, e.g., IF, LOOP, DO, END, etc., those serving as infix operators, e.g., AND, OR, MOD, and those for which redefinition by programmers would cause confusion in reading programs, e.g., UNTIL, WHEN, FALSE, etc.

2G. Numeric Literals (SATISFIED)

Fixed point literals are described in S3.2.3 and floating point literals in S3.3.3. The definition of literal values ensures literals have the same values in programs as data; see the discussion of TO\_FIXED in S3.2.4.1 and TO\_FLOAT in S3.3.4.1.

2H. String Literals (INTENT SATISFIED)

As we discuss in J3.7, character strings are treated as predefined types in our language, not as one-dimensional arrays of characters. Given that the intent of 2H is to require string literals, this is not a deviation from 2H. As S3.7 shows, strings are fixed length. The syntax of string literals (S3.7.3) shows that string literals are not allowed to cross program line boundaries (see also the answer to Question 36).

2I. Comments (SATISFIED)

Comment syntax is described in S2.1.8. It would be a reasonable simplification of comment capabilities to remove the requirement for embedded comments, since such comments are difficult to format so they are readable. Eliminating embedded comments would be an insignificant loss of capability and remove one construct of the language to be taught and remembered.

1.1.3. Types3A. Strong Typing (SATISFIED)

The declaration of variables shows clearly that the type of a variable is always known at translation time. Constraints on function definitions (S4.3) and generic definitions (S5) ensure that the type returned by a function is always either constant over all calls to the function or is determined by the types of the function's arguments. (See further discussion with respect to requirements 7D and 3-3E, however.)

### 3B. Implicit Type Conversions (SATISFIED)

The way this requirement is satisfied is discussed in the justification for S3.1.

### 3C. Type Definitions (SATISFIED)

New data types can be introduced as described in S3.11. The scope of a type definition is the same as the scope of an identifier declaration (S4.1). Restrictions on defined types are discussed in conjunction with S3.11.

#### 1.1.3.1. Numeric Types

##### 3-1A. Numeric Values (SATISFIED)

Integer is a special case of a fixed point type, as is described in S3.2.1. Exceptions associated with numeric operations are discussed in Sections S3.2.4 and S3.2.5 for fixed point and S3.3.4 and S3.2.5 for floating point.

##### 3-1B. Numeric Operations (SATISFIED)

Conversion from fixed point to floating point is performed by the TO\_FLOAT function, described in S3.3.6.1. Conversion from floating point to fixed point is performed by the TO\_FIXED function, described in S3.2.6.1. The other predefined operations are discussed in S3.2.4 and S3.3.4. A symbol for exponentiation is provided but its semantics are not predefined (see S3.1.4, S3.2.4.1 and S3.3.4.1).

##### 3-1C. Numeric Variables (PARTIALLY SATISFIED)

The issue of default ranges for numeric variables is discussed in J3.2.1. Range specifications are, however, interpreted as the minimum range to be implemented, as S3.2.3 and S3.3.3 show (by virtue of the default size occupied by fixed and floating point values). Explicit conversion between ranges is not required, as the discussion of fixed and floating point assignment shows (S3.2.5 and S3.3.5).

##### 3-1D. Floating Point Precision (SATISFIED)

Precision of variables is specified in S3.3.1. Precision of expressions is discussed in S3.3.4.4. Implicit rounding of floating point results is discussed in S3.3.4.1. No precision conversions are required, as the discussion of infix operators (S3.3.4.2) and assignment (S3.3.5) shows.

##### 3-1E. Floating Point Implementation (SATISFIED)

The required abilities to inquire about implemented precision, exponent range, etc. are specified in S3.3.1.

3-1F. Integer and Fixed Point Numbers (SATISFIED)

Fixed point numbers are treated as exact numeric values (S3.2). No implicit truncation or rounding is permitted (S3.2.4, S3.2.5, and S3.2.6).

3-1G. Fixed Point Scale (SATISFIED)

The issue of fixed point scale support is discussed in J3.2. We recommend as a language simplification that fixed point step sizes be limited to powers of 2, since other step sizes contribute to language complexity without really adding significant capability for embedded computer applications. This point is discussed further in J3.2.

3-1H. Integer and Fixed Point Operations (SATISFIED)

Fixed point division is provided by the // operator; remainders by the MOD operator (S3.2.4.2). Conversion between step sizes is provided by the TO\_FIXED function, S3.2.6.1. Implicit step size conversion is permitted just where the IRONMAN requires it, as the discussion in S3.2.4 and S3.2.5 shows.

1.1.3.2. Enumeration Types3-2A. Enumeration Type Definitions

The enumeration type is discussed in S3.4. Equality and inequality are automatically defined (S3.4.4.1). Subranges of enumeration types are permitted only for ordered types. The rationale for this decision is given in J3.4.1.

3-2B. Ordered Enumeration Types (SATISFIED)

Ordered types are marked in their definition (S3.4.1). Ordering operations are automatically defined for such types (S3.4.4).

3-2C. Boolean Type (INTENT SATISFIED)

The boolean type is described in S3.5. Conjunction, inclusive disjunction, and negation are described in S3.5.4.

3-2D. Character Types (SATISFIED)

The definition of new character sets is supported, as an example in S3.11 shows and as examples in S3.4.2 show. An example of an ASCII character set definition is given in S3.7.2.

### 1.1.3.3. Composite Types

#### 3-3A. Composite Type Definitions (SATISFIED)

Arrays are described in S3.8, records in S3.9.

#### 3-3B. Component Specifications (SATISFIED)

Definitions of component types follow the same syntax as declarations of simple variables of the component type, as S3.8.1, S3.9.1, and S3.1.1 show.

#### 3-3C. Operations on Composite Types (SATISFIED)

Array subscription is discussed in S3.8.4. Record field accessing is discussed in S3.9.4. Assignment operations for arrays and records are discussed in S3.8.5 and S3.9.5. Array constructors are discussed in S3.8.2; record constructors in S3.9.2. Assignable components of such objects can be used like variables of the type except when the array or record has non-standard representation and the component is to be used as an inout routine parameter. This is consistent with 11B.

#### 3-3D. Array Specifications (SATISFIED)

The declaration of arrays is specified in S3.8.1. Array indices of unordered enumeration types are not permitted, for reasons discussed in conjunction with S3.4.1.

#### 3-3E. Operations on Subarrays

Concatenation and substring access and assignment are defined for bitstrings and character strings, which can be considered defined types using array# for their representation (see our discussion of this point in J3.6). Restrictions on the use of substring access and string concatenation are discussed in S3.6.4. No predefined operations for arrays per se are provided, but such operations can easily be developed by extension. Our argument in favor of this position is given in J3.8.4.

#### 3-3F. Nonassignable Record Components

Constant record components are provided (S3.9.1). Components whose values are the dynamic values of expressions are not provided, for reasons discussed in J3.9.1.

#### 3-3G. Variant Types (SATISFIED)

Record variants are presented in S3.9.1. Their use is discussed in S3.9.4 and S3.9.5.

3-3H. Tag Fields (SATISFIED)

Tag fields are discussed in S3.9.1. Support for untagged records (as required by the answer to Question 45) is provided by the type-specifier construct applied to bitstrings, as is discussed in S3.9.3. This language capability is used in our example of an I/O support package in Appendix C.

3-3I. Definitions of Dynamic Types (SATISFIED)

Pointers are discussed in S3.10. A discussion of language design issues involving pointers is given in J3.10 and for the discriminating SELECT statement, J4.5.5.2.

3-3J. Constructor Operations (SATISFIED)

Allocation operations are given in S3.10.2. Garbage collected storage allocation semantics are supported.

1.1.3.4. Set Types3-4A. Bit Strings (SATISFIED)

Bit strings are supported as a separate data type that is in principle definable within the language as an array of boolean values. This issue is discussed further in J3.6.

3-4B. Bit String Operations (SATISFIED)

The required operations are described in S3.6.4. In addition, the concatenation and substring operations (required for arrays in general) are defined in S3.6.4 and S3.6.5.

1.1.3.5. Encapsulated Definitions3-5A. Encapsulated Definitions (SATISFIED)

Encapsulations are called modules in our language. Modules are defined in S4.4 and their use is illustrated in S3.11 and S5.

3-5B. Effect of Encapsulation (SATISFIED)

The requirements for inhibiting access to definitions in encapsulations are satisfied by the module capability, as described in S4.4 (Modules) and S4.1 (Scope Rules).

3-5C. Own Variables (SATISFIED)

Own data of a module is described in S4.4. Initialization of own data is discussed in S4.1.

3-5D. Operations Between Types (SATISFIED)

As the IRONMAN notes, satisfaction of 3-5B implies satisfaction of this requirement.

1.1.4. Expressions4A. Form of Expressions (SATISFIED)

A type-free syntax of expressions is given in S3.1.4.

4B. Type of Expressions (SATISFIED)

Since expressions containing operators are always interpreted (in principle) as calls on functions, the constraints on function definition in S4.3 and S5 show that the type of value a function returns is always determinable by a translator at the point of the call, either because the type of the returned value is the same over all calls to the function or because the type depends on the type of the function's arguments. Ability to specify the type of an expression explicitly is given by the type-specifier form of expression (S3.1.4).

4C. Side Effects

We provide a different side effect constraint that is more useful from an axiomatic definition viewpoint and also from a programming viewpoint. The constraint and its rationale is discussed in J4.1.

4D. Allowed Usage (SATISFIED)

The cross-reference listing for expressions, variables, and constants, shows that this requirement is satisfied.

4E. Constant Valued Expressions (SATISFIED)

A discussion of constant valued expressions is given in J3.1.4.

4F. Operator Precedence Levels (SATISFIED)

Operator precedence levels are discussed in J3.1.4. The syntax of expressions (S3.1.4) shows that the precedence of operators does not depend on the types of the operands.

4G. Effect of Parentheses (NOT SATISFIED)

We give arguments in J3.1.4 showing why this requirement should be changed.

### 1.1.5. Constants, Variables, and Declarations

#### 5A. Declarations of Constants (SATISFIED)

The declaration of constant-names is described in S3.1.1. Such constants need not be manifest constants.

#### 5B. Declarations of Variables (SATISFIED)

There are no default declarations of variables (S3.1.1). Variables of any type are clearly allowed by the syntax for variable-declaration. The allocation rules for variables are described in S4.1.

#### 5C. Scope of declarations (SATISFIED)

Scope rules are described in S4.1. Lexically embedded scopes are permitted. The ability to inhibit access to non-local variables is limited to closed scopes, i.e., procedures, functions, and paths.

#### 5D. Restrictions on Values (SATISFIED)

Procedures, functions, types, labels, and exception situations are not permitted as values of variables or computable as values of expressions since the syntax for type-spec (S3.1.1) does not permit such specifications. Types and routines appear to be parameters of procedures or functions, but this is actually only the case when the procedure or function is a generic definition (see S5).

#### 5E. Initial Values (SATISFIED)

No default initial values are provided, even for pointer variables, although in implementations, such variables must be initialized to NIL as far as the garbage collector is concerned. This, however, we consider an implementation matter; it is not a part of the language's semantics. The syntactic form for declaring constants resembles that for initializing variables, but only because the assignment conversions permitted in defining declared constants are the same as those permitted in initializing variables. We do not, however, use a form like INIT (expression) to both initialize variables and to declare constants. We interpret 5E's prohibition to forbid such a language form. On the other hand, it would be perfectly reasonable to replace := in CONST declarations with a keyword, e.g., IS.

#### 5F. Operations on Variables (SATISFIED)

The intent of this requirement is satisfied, since assignment and value access operations are automatically defined for all variables of built-in types. In our language, user-defined types specify whether an assignment operation is provided for variables of the type (i.e., whether the := operator can be used) or whether such variables are modified only as a result of invoking other operations on the type (e.g., PUSH for stacks, REQUEST and RELEASE for semaphores, etc.)

### 1.1.6. Control Structures

#### 6A. Basic Control Facility (SATISFIED)

A simple set of control structures is provided (S4.5), as is discussed further for subsequent requirements. Each structure has a distinguishing syntax. Nesting of control structure is allowed (as the use of open-scope-body in the cross-reference listing shows). Local scopes (i.e., open scopes) are allowed within control statement bodies. Control structures are single entry and exit except when the loop exit statement (S4.5.6.2) or the signal-stmt (S4.5.7.2) is used. These are specialized forms of the goto-stmt.

#### 6B. Sequential Control (SATISFIED)

We have provided the semi-colon as a statement delimiter. It serves as a statement terminator, not a statement separator, as in PASCAL. Studies have shown that the use of statement terminators is less error-prone than statement separators. In addition, we have not taken the IRONMAN suggestion to design the language so delimiters could be considered either terminators or separators because, from a maintainability viewpoint, it is best if all programs are written in the same style, (i.e., as separators or terminators). In general, when notation is more a matter of taste than substance, we have designed the language to restrict the options open to programmers because of the emphasis on maintainability as opposed to writability.

"It is a good principle in language design not to give more than one way of doing something. If you do, each programmer will choose one of the ways at random, and use it consistently; and he will find great difficulty in understanding programs written using the other convention." [Hoare, C.A.R "Critique of Standard Computer Language JOVIAL (J73), p. 38]

We have followed this principle throughout the language design. We have followed this principle throughout the language.

#### 6C. Conditional Control (SATISFIED)

The conditional control structures are the IF statement (S4.5.5.1) and the SELECT statement (S4.5.5.2). The ORIF construct (S4.5.5.1) satisfies the requirement for choosing the true condition in a set of conditions. When all alternatives are not specified, our language says that the effect of selecting an unspecified alternative is the same as the effect of executing a null statement. The requirement for compiling only selected branches of a conditional statement is satisfied and the need for this requirement is justified in our discussion (J4.5.5.1).

#### 6D. Short Circuit Evaluation (SATISFIED)

All boolean expressions are evaluated in short circuit mode in our language. This choice is justified in J3.5.4.

6E. Iterative Control (SATISFIED)

The indefinite and definite loop statements satisfy this requirement (S4.5.6). The iterative control structure for iterating over subranges of ordered types is limited to an iteration step of +1, since Wichman's studies of ALGOL programs has shown that this form of iteration occurs 90% of the time.

6F. Control Variables (SATISFIED)

Loop control variables are local to the loop (S4.5.6.3). Control variables associated with parallel paths are also local to the path (S4.6.1). There are no other such control variables.

6G. Explicit Control Transfer (SATISFIED)

A goto-stmt is provided (S4.5.1). It does not permit control transfer out of closed scopes, i.e., routines and parallel paths, since labels outside such control structures are not made known inside the structure (S4.1). Control into narrower access scopes is not permitted, e.g., into the THEN part of an IF statement, since such structures form scopes, and labels defined inside such scopes are not known outside the scope. There are no switches, designational expression, label variables, label parameters, or alter statements. Transfers between branches of a conditional are not permitted, since each branch forms a parallel open scope. See further discussion in S4.5.5.2 and J4.5.5.2.

1.1.7. Functions and Procedures7A. Function and Procedure Definitions (SATISFIED)

Functions and procedures can be overloaded, including those associated with infix operators, as S4.1, S4.3, and S5 show.

7B. Recursion and Nesting (SATISFIED)

Procedures and functions are automatically recursive (see J4.4). Nested routine definitions are permitted (S4.1). No display is needed (see J4.3.1).

7C. Scope Rules (SATISFIED)

Scope rules are lexical, as required (S4.1).

7D. Function Declarations (SATISFIED)

The result type of functions is explicitly specified (S4.4.1) and can be determined at translation time. Our definition of what is "determinable at translation time" is discussed in J4.3.1. We do not permit functions associated with infix operators to be associative, since our modification to 4G makes this requirement unnecessary (see J3.1.4.1).

7E. Restrictions on Functions (MODIFIED)

We have proposed a modification to this requirement that appears to better satisfy the intent of restricting side-effects (see J4.5.3).

7F. Formal Parameter Classes (SATISFIED)

The three parameter classes are provided, as discussed in S4.3.1, and S4.1. The parameters are evaluated at the point of the call (S4.3.4).

7G. Parameter Specifications (SATISFIED)

Parameter types are fully specified (S4.3.1). It is an error if the actual parameters are not compatible with the formal parameters (S4.3.4).

7H. Formal Array Parameters (SATISFIED)

Dimensions of array parameters must be specified (S4.3.1), although determination of the subscript range can be deferred until run time and may vary from call to call by use of the Generic definition capability (S5). This use of generic definitions satisfies the array parameter requirement in a uniform way, rather than as a special case capability (see J5 and J3.8.1 and J3.8.5).

7I. Restrictions to Prevent Aliasing (SATISFIED)

The required aliasing restrictions are described in S4.3.2. Their impact on pointer semantics is discussed in J3.10.

1.1.8. Input-Output Facilities8A. Low Level Input-Output Operations (SATISFIED)

Low level operations are defined in S4.8. An illustration of their use is given in Appendix C, where illustrative application-level I/O packages are shown.

8B. Application Level Input-Output Operations (SATISFIABLE)

Our design document does not make any firm proposals in this area. We have instead limited ourselves to showing that a variety of application level I/O capabilities can be defined within the language. This demonstration is given in Appendix C, which exercises both the most advanced features of the language (generic definitions) and the lower level I/O capabilities. The determination of a minimal I/O standard package we defer to Phase 2 of the effort.

8C. Input Restrictions (SATISFIED)

Appendix C's illustration of binary I/O shows that this requirement can be satisfied within the language.

8D. Operating System Independence (SATISFIED)

A minimal set of run-time support required by the language is discussed in Appendix A of this document. However, aside from the support required for parallel control, the language is independent of any operating system. In fact, we have shown in Appendix D of the language specification how the parallel control facilities of the language can be redefined to meet the specialized needs of an application by redefining the routines associated with the activation and control of parallel paths.

8E. Configuration Control (SATISFIED)

The low level I/O facilities (S4.8) can be used to satisfy this requirement. Other implementation dependent queries such as the amount of unused stack space, etc. have not been defined in the language specification; their definition has been deferred to Phase 2.

1.1.9. Parallel Processing9A. Parallel Control Structure (SATISFIED)

The parallel control path capability is described in S4.6.

9B. Parallel Path Implementation (SATISFIED)

The parallel path semantics permit multi-processor implementation. In addition, the underlying semantics of paths is to call routines implementing path activation, scheduling, etc. Since embedded computer application executives are typically special purpose executives with unique scheduling policies and disciplines, the language provides a means of realizing different policies by redefining the module that (in principle) implements the parallel path constructs. This is discussed further in Appendix D of the Specification.

9C. Mutual Exclusion and Synchronization (SATISFIED)

The required capabilities are provided via semaphores (S3.11), request and release operations (S4.6.6 and S4.6.7). An example of a monitor implemented using these features is given in S4.6.8.

9D. Scheduling (SATISFIED)

The specified scheduling discipline is given as the built-in semantics of the parallel control structure (S4.6.1 and S4.6.4).

9E. Real Time Clock (SATISFIED)

The real time clock capabilities are described in S4.6.2, S4.6.5, and S4.6.5.

**9F. Simulated Time Clock (SATISFIED)**

The required capability can be realized by defining an appropriate simulation module, much as SIMULA 67 does (see S4.9).

**9G. Catastrophic Failures (SATISFIED)**

The TERMINATE statement is described in S4.6.3. Additional discussion of possible modifications to this requirement is in J4.6.3.

**1.1.10. Handling****10A. Exception Handling Facility (SATISFIED)**

The exception handling features of the language are described in S4.5.7. Errors detected (potentially) by hardware include OVERFLOW and UNDERFLOW, but not memory failure, etc. which are asynchronous errors (i.e., their occurrence is not associated with the invocation of any particular operations). The method of detecting and dealing with hardware failures is highly machine and application dependent, and so is not part of the language.

**10B. Error Situations (INTENT SATISFIED)**

Those error situations associated with predefined operations are defined in the language together with the circumstances under which an exception is raised. Dynamic aliasing of array components is not permitted (see J4.4.3). Attempting to access a field of a variant that is not present is either an error detectable at translation time (see S4.5.5.2 for variant records) or is an error caused by the free union capability, and is inherently undetectable. Discussion of path termination is in J4.6.3. Failure to satisfy a program specified assertion is discussed in J4.5.8.

**10C. Enabling Exceptions (SATISFIED)**

We use the term "raising" an exception instead of the IRONMAN's "enabling" an exception. The signal-stmt (S4.5.7.2) raises exceptions. It does not cause transfer out of a procedure, function, or parallel path unless there is no local handler for the exception and the routine or path has been declared to be able to raise the signaled exception. Exceptions that can be raised by built-in operations are defined in S3.x.4 for each built-in type.

**10D. Processing Exceptions**

The exceptions-clause serves to discriminate among exceptions that can be raised (S4.5.7.1). The remainder of this requirement is treated differently in our design, as is discussed in J4.5.7.

**10E. Order of Exceptions (SATISFIED)**

Order of exceptions in expressions is not guaranteed (S3.1.4.).

10F. Assertions (SATISFIED)

Assertions are permitted (S4.5.8). No side effects are permitted in the assertion expression, and this constraint can be enforced by the translator (see S4.4.4).

10G. Suppressing Exceptions

For reasons discussed in J4.5.7, exceptions in our language are normally suppressed. Providing a handler for the exception ensures that the exception situation is checked for.

1.1.11. Object Representation11A. Data Represetion (SATISFIED)

Representation specifications are described for each built-in data type in S3.x.3. For record structures, the specification of field order and filler fields is described in S3.9.3. The object representation of atomic data is currently limited to specification of the number of bits occupied by values. The ability to specify ones-complement or twos-complement representation, for example, is probably possible (see S3.11.3 and J3.11.3), but raises problems best resolved in Phase 2.

11B. Multiple Representations (SATISFIED)

The representation specification capability as described in S3.x.3 for each predefined type and in S3.11.3 for user-defined types satisfies this requirement. The restriction on inout parameters is described in S4.4. Representation changes can be accomplished implicitly by assignment or by invoking some function requiring a different representation for its arguments. Explicit representation conversions are provided by the explicit-rep-converter form (S3.1.4.1).

11C. Translation Time Constants and Functions (SATISFIED)

The specification of object machine configuration constants has not been defined in detail. However, it is discussed in S4.8. The ability to determine properties of program components is supported in part through attribute-queries (S3.1.1). For a variable V, the specified range is determined by writing V.MIN (for the lower bound) and V.MAX (for the upper bound). The implemented range depends on the number of bits used to represent the value, and this is V.SIZE, which is automatically defined for every built-in or user-defined type. The HANDLER\_EXISTS and MANIFEST\_CONSTANT function is described in S3.5.6. The current optimization criteria can be queried through constants to be defined in S4.8. Machine-dependent programmins is discussed in S4.8.

11E. Code Insertions

In S4.8 we describe a method of integrating procedures written in assembly code with statements of the language. Since the procedures may be inline, the overhead of a procedure call can be avoided but the method of using assembly code clearly separates it from the HOL code. The need for object code insertions is minimized by virtue of the start-to-stmt (S4.8) and the ability to specify that bitstrings are of some type (in a machine-dependent section of code) (see S4.8).

11F. Optimization Specifications

The specification of optimization criteria we consider more an implementation issue than a language design issue. Rather than provide for different degrees of optimization within the same compilation unit, we require that a given unit be compiled with the same degree of optimization throughout. This is consistent with current practice and removes what would otherwise be an additional language capability to be described and supported by implementations. Our general philosophy has been to provide explicit directives that affect time and space usage in clearly specified ways rather than provide general directives that depend on the intelligence of a particular translator implementation to carry out correctly. For example, local time/space optimizations can have an adverse affect on global time/space efficiency. It is better to depend on a programmer to make such tradeoffs by appropriate use of inline routines and data packing directives than to depend on the intelligence of a translator.

1.1.12. Libraries, etc.12A. Library Entries (SATISFIED)

The use of libraries is discussed in S4.7.

12B. Separately Compiled Segments (SATISFIED)

Separate compilation is discussed in S4.2. Exporting and importing of definitions and declarations from separately compiled segments is discussed in S4.1. All language constraints are enforced across separate compilation interfaces, but as we note in J4.2, this imposes a considerable burden on both the users of the language and the implementation; the implications of this requirement on the implementation of a translator and the language design need to be explored further in Phase 2.

12C. Restrictions on Separate Compilation (SATISFIED)

This requirement is satisfied, as S4.2 shows and J4.2 notes. However, its implications on program development practices and translator complexity (and interaction with other development support software) needs considerable further study in Phase 2 to determine whether "integrity of object code" can be reasonably enforced, as opposed to aided by the language and support system.

12D. Generic Definitions (SATISFIED)

The generic definition capability is satisfied in principle (see S5) but no method of passing statements directly as generic parameters is provided.

1.1.13. Support for the Language13A. Defining Documents (NOT APPLICABLE)

The language definition we have provided is intended to be a first approximation to a reference manual for the language.

13B. Standards (NOT APPLICABLE)

This requirement does not apply to the language design.

13C. Subsets and Supersets (SATISFIED)

We have defined the language so it provides a basic capability that can be supported by all implementations, and at the same time, the definitional facilities of the language permit extending the language in standardized ways to support special capabilities such as multiple precision fixed point, matrix arithmetic, varying length character strings, etc. Although these extensions are definable using the language's definitional facilities, we expect that often the extensions will be directly supported by some translators, so the extensions can be implemented efficiently. Strictly speaking, such extensions are not supersets because they are definable directly within the language.

13D. Translator Diagnostics (SATISFIED)

All language imposed restrictions can be enforced by the translator except for restrictions involving the use of variant records accessed by pointers (see J4.5.5.2). Diagnostic messages have not been provided in the language definition at this time, however.

13E. Translator Characteristics (SATISFIED)

The language does not dictate the internal characteristics of the translator. For example, the definition of manifest-expression does not require that A + 5 in the following case:

```
A := 6;  
... A + 5 ...
```

be considered a manifest-expression, since this would require that a translator perform constant propagation optimizations.

13F. Translation and Execution restrictions (SATISFIED)

Arbitrary restrictions are avoided in general. Certain language imposed restrictions for each of the data types are discussed in J3.x.1 sections of this document.

### 13G. Software Tools and Application Packages (SATISFIED)

Standard application libraries can be provided, as Appendix C of the specification illustrates.

#### 1.2. SUGGESTED CHANGES

In this language design effort, we have attempted to satisfy all IRONMAN requirements as fully as possible, even though it was clear from the beginning that a complex language would be the probable result. However, we felt that only by doing a thorough analysis of language features to support the requirements would we truly understand the cost of the requirements in language and/or implementation complexity. In this Section, we discuss some changes that we feel would lead to useful simplification of the language. We also mention some areas where more analysis is needed.

##### 1.2.1. Aliasing

Prevention of aliasing is a significant source of language and implementation complexity. Preventing aliases requires that every routine header contain information about all the non-local data used (either directly or indirectly, by calling another routine). In principle, listing such information is a good idea. After all, a routine performs a mapping from inputs to outputs; listing all its inputs and outputs in its header just makes this mapping more evident.

But the effect on the translator of enforcing aliasing checks is unpleasant. A change in the implementation of a routine may necessitate recompilation of all using routines, all routines using these routines, etc., in order to redo the aliasing checks.

The most important practical problem with aliasing detection appears to be that it conflicts badly with separate compilation. Separate compilation is a concept of proven worth in embedded computer system development. Aliasing detection is a new and as yet untested idea, developed to make proofs of correctness easier and without consideration of its effect on program development and modification. Even its effect on program proofs is somewhat debatable, since many routines work perfectly well in the presence of aliasing, and proofs in the presence of aliasing are currently a matter of research. Note that aliasing detection was invented for EUCLID (although suggested by earlier work on the axiomatic definition of PASCAL); the designers of EUCLID explicitly stated that separate compilation was not a goal, however.

We recommend that aliasing detection be removed as a requirement. The IRONMAN is inconsistent here anyway, since aliasing effects due to the use of pointers (dynamic variables) must nonetheless be dealt with (see J4.5.4).

#### 1.2.2. Nested Routines and Modules

Our current design permits routines and module declarations to be nested inside routine declarations. Such nesting is probably not beneficial to good program structure, since it encourages the use of global variables in place of parameters. Modules provide a mechanism for limiting the accessibility of data; the mechanism is probably enough. Therefore we recommend that such nesting not be permitted.

One ability that is lost if modules cannot be declared inside routines is that of having different copies of the own data of the module. We believe that if such an ability is needed, the necessity of achieving it through programming will not be an undue hardship.

Note that if modules and routines cannot be declared in other routines, a display is not needed. Also rules constraining what data recursive routines can import become identical to the rules for non-recursive routines.

Clearly it is necessary to declare routines inside modules. While it is unclear how useful it is to declare modules inside modules, we recommend that this kind of nesting be permitted; note that a display is not needed for this.

#### 1.2.3. Segments

The rules about program structure would be simplified if segments were not permitted to hide declarations, but instead simply represented a piece of the total program. This simplifies the language because where we currently have two hiding mechanisms (segments and modules), we would only have one. Such a change would encourage a secondary use of modules as a repository for a part of the system data together with a set of routines to manipulate that data. Such a module supports a single abstract object, rather than an abstract type.

#### 1.2.4. Variant Records

Variant records are quite complicated. One way of simplifying them is to remove the constant components. This feature seems to be of minimal utility. More interesting and useful would be a feature that permitted access control of record components (i.e., the ability to specify that certain fields of a record are read-only in some scope). Whether the complexity of such a feature is really justified is unclear, however.

Another way of simplifying variant records might be to divide them into two types: (non-variant) records and a tagged, discriminated union. This approach should be studied in Phase 2.

### 1.2.5. Functions with Side-Effects

One simplification would be to require that functions not have any side-effects. Such a restriction would ensure that expressions invoking functions would be side-effect free and could be freely optimized.

If functions cannot have side effects, it would probably be desirable to permit procedures to return values. Such procedures could appear in expressions, but many optimizations would not be possible and an order of operand evaluation would have to be specified.

Our current rule about optimization in the presence of functions with side effects is complicated. Furthermore, it depends on the use of aliasing information discussed earlier. In the absence of such information, the side effect rule would not be worth enforcing, and very little optimization of expressions would be possible when invoked functions have side effects.

### 1.2.6. Fixed Point

The fixed point data type would be considerably simplified if only radix 2 step sizes are permitted. Since this is the step size supported in current embedded computer languages and is adequate to meet the needs of embedded computer applications, it seems worthwhile to impose this restriction.

## 2. LEXICAL STRUCTURE OF THE LANGUAGE

### 2.1. OVERVIEW

#### 2.1.1. Character Set

##### J2.1-1.

We show display characters in the lexical syntax in order to explain their effect in comments, string literals, and in determining what sequences of character codes form tokens.

#### 2.1.2. Tokens

##### J2.1-2.

Having the new-line character sequence separate tokens is required by the IRONMAN.

#### 2.1.3. Identifiers

#### 2.1.4. Literals

#### 2.1.5. Other Symbols

#### 2.1.6. Reserved Words

#### 2.1.7. Key Words

#### 2.1.8. Comments

##### J2.1-3.

A reasonable simplification of the comment capability would be to delete embedded comments. They pose difficulties for program reformatters, and programs are generally clearer if embedded comments are not used. The programs in Appendix C use no embedded comments, and their absence is not missed, from a readability viewpoint.

2/15/78

**Comments**

**Comments**

2-2

**Section 2.1.8.**

### 3. TYPES AND OPERATORS

#### 3.1. OVERVIEW

##### J3.1-1.

The view of type that we take in this language is different from the view presented in the IRONMAN in that we define "type" to include all abstract and representational properties of a value. We then specify which of these properties must match in assignments and formal/actual parameter correspondence. The IRONMAN's use of the term "type" corresponds to our term "type class". The kinds of implicit type conversions we permit are precisely those the IRONMAN requires, e.g., implicit conversions between floating point ranges and precisions, conversions between fixed point scales, etc. We do not, however, permit implicit conversions between type classes, thus satisfying 3C.

##### 3.1.1. Declarations and Attributes

###### 3.1.1.1. Type Attributes

##### J3.1-2.

The reason for not evaluating the expression in attribute-queries is that in queries of the form (FIXED.SCALE).MIN, the value of FIXED.SCALE is not defined, but since the type of the scale attribute is defined, (FIXED.SCALE).MIN gives the minimum permitted value for the fixed point SCALE attribute. In general, since the type of an expression is defined at translation time, there is no need to evaluate the expression in an attribute-query. The purpose of the query is to determine some attribute of the type of the expression, not the expression's value.

###### 3.1.1.2. Constant Declarations

###### 3.1.1.3. Variable Declarations

##### J3.1-3.

The "uninitialized variable" exception will, in general, be expensive to check for, since it applies to individual elements of arrays and fields of record as well as scalar variables. In general, if it cannot be determined at translation time that initialization has been performed for certain variables or their components, a special bit vector must be kept to detect use of an uninitialized variable. Since exceptions are normally suppressed (see

S4.5.7), the cost of uninitialized variable checking will usually be avoided, but of course, an implementation must be prepared to support uninitialized variable detection at execution time.

It should be noted that no exception handlers can be attached directly to declaration statements, even though evaluation of expressions in declarations can raise exceptions that are not detectable until run-time. We have not permitted such handlers because it is not clear what the effect should be on the declaration raising the exception (e.g., is it possible to provide a new declaration within the handler? If no replacement declaration is provided, what is the interpretation, etc.) If necessary, a handler can be attached to a block enclosing the scope in which exceptions from declarations are expected to arise. For example,

```
BLOCK
  VAR A: ... := F(X);
  VAR B: ... := 3;
  VAR C: ... := F(Y);
  ...
END BLOCK; EXCEPT
  RANGE -->      % range handler
  INVALID_REP --> % rep handler
END;
```

If F(X) yields an out of range value, then the RANGE handler will be executed and evaluation of the block terminated. None of the variables declared in the block are accessible in the handler. (This is the normal scope rule for handlers.) Since declarations are processed in the order they occur in program text, any side-effects associated with declaration processing are well defined, e.g., if F is a function and F(Y) returns an out of range value, the RANGE handler will be executed, but F(X) will have been evaluated first.

Exception handling issues are discussed further in J4.5.7.

#### 3.1.1.4. Type Specifications

##### J3.1-4.

Array-specs have a different form because their retype-specs do not appear at the end of the type-spec, e.g.,

```
ARRAY [1:10] REP ARRAY_REP(DENSE) OF BOOLEAN;
```

Having the retype-spec appear at the end of the array declaration would make it too easily confused with the retype-spec of the array's element type.

### 3.1.2. Literals and Constructors

### 3.1.3. Representation Specifications

### 3.1.4. Predefined Operators

#### 3.1.4.1. Operators and Expressions

##### J3.1-5.

The IRONMAN requires "few" levels of precedence in expressions. We consider the six levels of precedence (for infix operators) that we have provided to be the minimal number that is appropriate for reliable use of the language. Fewer levels could have been provided by following PASCAL, placing AND at the precedence level of \* and OR at the level of +. Wirth, however, has noted ["An Assessment of PASCAL", IEEE Trans. on Software Engineering, June 1975, p. 194] that "in retrospect, however, the decision to break with a widely used tradition [with respect to the precedence of AND and OR] seems ill-advised, particularly with the growing significance of complicated Boolean expressions in connection with program verification." So our deviation from PASCAL is accepted by PASCAL's designer.

An additional reason for providing traditional levels of precedence is that in a language where infix operators can be overloaded, intuitive precedence rules provide greater safety against misunderstanding. For example, if - is extended to bit strings to mean the set difference operator of PASCAL, then A - B AND C will parse as A - (B AND C) using PASCAL's precedence levels, but as (A - B) AND C using our (and conventional) precedences. (Note that requirement 4G does not apply in this case, since - and AND have different precedences, so parentheses are not required.) We believe that traditional precedence levels will be less prone to misinterpretation when infix operators are overloaded.

If it were not for 4F, we would put the implication operator at the lowest level of precedence, since in its use in assertions, this is its appropriate precedence level. Despite 4F, we have placed \*\* at a separate level of precedence, since to do otherwise would be too confusing a break with tradition, causing A \* B \*\* C to be parsed as (A \* B) \*\* C.

##### J3.1-6.

Our grammar does not satisfy 4G, which requires that A - B - C be parenthesized as (A - B) - C. We feel that 4G should be dropped, for several reasons. First, it does not force parenthesization in all circumstances where operand grouping makes a difference in the semantics of expression evaluation. For example, (A OR B) XOR C is not equivalent to A OR (B XOR C), but 4G does not require parenthesization in this case since OR and XOR are both associative in the normal sense of the term. Consequently, the

parenthesization rule proposed in 4G will be difficult to explain to programmers and difficult for them to use correctly, since it requires parentheses in situations where programmers are not used to providing them and does not require parentheses where the parentheses would have a semantic effect. In addition, in a language with operator overloading, it is impossible to ensure that overloaded operators assumed to be "associative" are actually associative. All in all, 4G should be abandoned in favor of a simpler, traditional rule, namely, the left to right association rule we have proposed.

An advantage of defining how operands are associated with operators is that it fixes the abstract value of an expression and also fixes the set of exceptions that can be raised when the expression is evaluated for specific operand values. It does not, however, determine which exception will be raised first, e.g., if  $A*B$  raises OVERFLOW and  $C*D$  raises UNDERFLOW, the language does not define which exception will be raised when the whole expression is evaluated. Note that once an exception has been raised, evaluation of an expression is terminated (see Section 4.5.7) and so no more than one exception is actually raised for any given expression evaluation.

Fixing the value of an expression and the exceptions it can raise is important since it leads to simpler proof rules and programs whose behavior is more predictable when optimized. In particular, we do not believe the answer to question 79 can be considered correct, since an optimization that removes an exception must be considered erroneous if the statement raising the exception is within the scope of a handler for that exception. OVERFLOW is not necessarily a programmer error, e.g., it is not an error when implementing a multiple precision arithmetic package. Consequently, it appears to us as important for program proofs and correctness to fix the exceptions a statement can raise as it is to fix the value produced by the expression.

A final argument in favor of our position is that it is the position supported by most languages and implementations, so programmers will find it easy to remember and use correctly.

### J3.1-7.

The reason for providing this form of interval specification is that in some embedded computer applications, in particular, flight training simulators, variables are partitioned into subranges and computations are conditioned on the subrange a particular variable is in. For example, the equations of motion for an aircraft will use a particular value if the aircraft's rate of climb is in the range [0.:750.), another value if it is in the range [750.:1200.), etc. Thus, providing a full range of partitioning capabilities is in direct support of embedded computer applications. Moreover, as will be seen in examining the rest of the language, this notation is readily adapted for other purposes as well, in particular, for specifying the range of values associated with a numeric or enumerated variable and in iteration statements.

It should be noted that it is not appropriate to use PASCAL's form of range specification, e.g., 1..10 because the decimal points in fixed and floating point literals are too easily confused with the periods separating the lower and upper bound values. This is partly the reason we use a colon. The other reason is that [1:10] is a traditional notation for expressing array bounds, so extending the notation for use in specifying ranges in general should be natural and easy for programmers to use correctly.

### J3.1-8.

The need for defining manifest-expressions occurs because manifest values are required in certain contexts in the language, e.g., as case-labels in case-stmts, in routine-headers to determine which overloaded routine is to be called, as attributes of certain predefined types (e.g., the scale of a fixed point value, etc.). If the language does not define which expressions are manifest, some implementations will permit certain expressions in these contexts where other implementations do not. The consequence will be that some programs will be considered legal by one implementation and not by the other. Such a consequence is not consistent with the goals of a common language.

### J3.1-9.

Restricting lower and upper range-spec bounds to the same type class is consistent with the view that the range-spec form is to be used to specify ranges of values within a given type.

#### 3.1.4.2. Operator Overview

#### 3.1.5. Assignment

### J3.1-10.

We permit the assignment operator to be overloaded, in apparent contradiction to the answer to question 35 because we do not find the answer convincing. First, although assignment is not a function, it is a procedure, and 7A permits procedures to be overloaded. Second, the fact that 5A provides an automatic assignment operator does not mean the operator cannot be overridden. In fact, we have taken a more conservative position: for types defined inside modules and exported from the module, only those operations explicitly defined by a user are also exported. This is a simple rule that provides for complete user control over the behavior of a user-defined type, as is illustrated by several examples in the language specification. Finally, redefinition of assignment need not violate the intent of 3B. We require that the assignment procedure accept only arguments of the same type class; this constraint is entirely consistent with 3B.

In addition, there are several cases where the automatically defined assignment operation (which we assume means to copy the bits of the representation) is not an acceptable definition. For example, the assignment of semaphores should not be defined to copy the queues associated with the semaphores. Similarly, in assigning one varying length string to another, bit-copy is not appropriate when the maximum lengths of the strings are not identical, as is often the case in varying length string assignments. If users cannot define the assignment operator, then a useful type like varying length strings cannot be defined appropriately within the language.

It should be noted that our constraint on assignment (which is motivated by 3B) is not always completely natural. For example, the constraint prevents assignments like  $A := 0$  where  $A$  is an array, since 0 is not an array value. Similarly, if  $VS$  is a varying length string variable,  $VS := 'STRING'$  is not permitted since 'STRING' is a fixed length string, and fixed length strings are not in the same type class as varying length strings. Consequently, our varying length string definition provides an operator for converting fixed length values to varying length values, namely, `TO_VAR_CHAR` (see S5). The inconvenience caused by this restriction is offset by the virtue of ensuring against implicit type class conversions in assignments; such conversions would violate a basic tenent of the language design and the IRONMAN.

### 3.1.6. Predefined Functions

#### J3.1-11.

The only predefined functions that need be provided are:

1. those requiring translator dependent knowledge to implement properly;
2. those needed in manifest constant expression evaluation to ensure conditional compilation;
3. those whose absence would make programming in the language difficult or inconvenient for many users of the language;
4. those for which it is clear what definition should be provided.

The last criterion is applied only if one of the first three is satisfied. For example, we have consciously not provided functions for converting from the built-in types to a character string representation, since such conversions are best supported as part of an input-output package. Depending on the capabilities supported by such a package, different conversion functions will be appropriate.

### 3.2. FIXED POINT

#### J3.2-1.

The fixed point capabilities we provide are the minimal capabilities satisfying 3-1G. In particular, by restricting step sizes to powers of integers (both positive and negative) we provide a less general capability than that described in the Fisher and Wetherall paper referenced in Question 34. The ability to specify step sizes that are any rational number, e.g.,  $2/3$  or  $15/2$ , is simply unneeded generality in our opinion. Every embedded computer language to date has supported only fixed point computations expressed in terms of powers of two. Virtually every embedded computer system only efficiently supports fixed point computations in which values are represented with step sizes that are powers of two. Since one of the primary reasons for using fixed point arithmetic in embedded computer applications is time and space efficiency, we believe it is unneeded generality to support computations for step sizes other than powers of two. However, since the IRONMAN does not currently take this strong a position, we have adopted a compromise position in which step sizes that are other than powers of two can, in principle, be specified by extending the language. Even this limited additional capability leads to unattractive complexity in the language design (e.g., constraints that the radices of operands must be the same, an additional argument to be considered in the TO\_FIXED function for specifying the radix of the result, etc.). In view of the complexity elsewhere in the language, we strongly recommend that the fixed point requirement be restated to limit step sizes to powers of two. This has proved adequate in the past, and there is no reason to believe it will not be adequate in the future.

#### J3.2-2.

We have not defined multiple precision fixed point arithmetic for several reasons: 1) although multiple precision fixed point is needed in some embedded computer applications, it is not needed in all; requiring all implementations to support it is probably not reasonable; 2) if multiple precision arithmetic is to be predefined, how much precision is appropriate and how should a cutoff point be decided? Whatever the cutoff, some application might require more; 3) single precision is adequate for integer computations (which are used to index arrays, substrings, etc.). All in all, it seems best to require that only one level of fixed point precision be supported, and then to ensure that the extension capabilities of the language permit additional levels of precision to be defined for those application areas needing them.

This approach does not violate 13C (Subset and Superset Implementations) since the extensions are definable in the language. In practice, however, we would expect such extensions to be directly supported by an implementation (so multiple precision arithmetic can be effectively optimized.)

## Fixed Point

3.2.1. Attributes and Type SpecificationJ3.2-3.

We permit default ranges, which is a deviation from requirement 3-1C and Question 39. However, we believe this deviation to be consistent with 1C's requirement that defaults be stated in the language definition, always be meaningful, reflect common usage, and can be explicitly overridden. Moreover, we believe the provision of default ranges yields a language that is more uniform in its syntax, more readable, and merely provides what a user can get with a type definition anyway. We discuss these points in detail below.

First, a default range for numeric variables certainly reflects common usage. Moreover, it is commonly the case that input parameters of procedures and functions accept any representable value of a fixed or floating point type, i.e., any value in the maximum range.

Second, the specification of maximal ranges for fixed point values is necessarily complex, since the maximal range depends on the scale, radix, and precision of the representation. We have provided functions `FIXED_MIN` and `FIXED_MAX` that yield these values, although we do not believe that use of these functions contributes to readability, e.g.,

```
VAR A: FIXED(0, 2, STD) [FIXED_MIN(0,2,STD) : FIXED_MAX(0,2,STD)]
```

Providing a default that replaces such necessarily cumbersome notation contributes to readability while providing a meaningful definition under all circumstances.

Third, for programmers to specify non-default representations, e.g., `UNSIGNED` or a representation size that is smaller than the standard representation, the language requires explicit specification of a range so the translator can verify that the representation specification does not change the abstract behavior of a program. Given that embedded computer programmers are often quite concerned about storage economy, we believe this requirement provides adequate incentive to use non-default ranges when they are meaningful.

J3.2-4.

Limiting the scale attribute to a range of [-1000:1000] imposes no significant limitation on users of the language, but does permit compiler implementers to ignore the possibility of OVERFLOW for scale factors. Some limitation should be imposed, and rather than have different implementations impose different limitations, it is better to have the language impose a standard one.

J3.2-5.

We limit the radix attribute to a small range for the same reason we limit the scale attribute's range, namely, to eliminate pathological implementation problems that arise if no limit is imposed. The limit does not reduce the useful power of the language.

Since fixed point values are only predefined for radix 2, a FIXED\_BIN (scale, precision) type might also be predefined to be equivalent to FIXED (scale, 2, precision). Such auxiliary type definitions can be specified in the language using the type declaration statement, described in Section 3.12.

J3.2-6.

The rationale for requiring only one level of precision has been discussed previously.

J3.2-7.

It should be noted that the precision of a fixed point variable cannot be determined by the range of the variable, since ranges are not necessarily specified with manifest expressions.

The representation size associated with STD precision is usually the number of bits in an object computer word. The number of bits in a STD precision value affects the number of values permitted in an enumerated type's value set and the range of subscript values that can be conveniently specified.

We have made the precision attribute for fixed point values an enumerated value to emphasize that discrete levels of precision are expected in the behavior of fixed point data. In particular, the representation of each FIXED\_PRECISIONS value tells how many bits are needed to represent the value. For example, on a computer with a 16-bit word length, we would expect STD fixed point values to be represented with 16 bits; this is encoded in the representation of STD so that TO\_FIXED(TO\_BIT(STD)) = 16. Users of the language need not be aware of this relation, but those extending the language to multiple-precision will need to take advantage of this method of coding information.

3.2.2. LiteralsJ3.2-8.

Literals preceded with a + or - are also, syntactically, manifest-fixed-expressions or manifest-float-expressions. This is an inconsequential syntactic ambiguity that is readily removed by writing more productions. Permitting + and - in the syntax for literals makes it more straightforward to explain conversions from character string values to fixed and float values. We need only say the character string has the form and value of a fixed or float literal.

J3.2-9.

Although the language definition supports the implementation of literals represented with radices other than 2, we doubt that there is really much need for such radices.

J3.2-10.

There are two options in defining the range of a literal: either the range is the range of the precision of the literal or the range is the value of the literal. We have chosen the second alternative because it works out best when invoking overloaded functions such as TO\_FIXED and functions associated with infix operators, e.g., +. The rules for choosing which overloaded routine to invoke (see S4.3.2) look for exact matches to the argument types first. Since the match is with respect to the type rather than the value of the argument, treating literals as having the range [L:L] provides different types for each literal value (since range is a part of a fixed point type). Thus the fixed point ADD\_ operator can have one definition for arguments with STD precision and a separate definition for arguments with LONG precision. This means that to extend an implementation to accept non-standard precisions, the infix, prefix, and predefined functions can simply be overloaded to accept additional argument types.

J3.2-11.

If .900. were permitted as both a fixed and floating point literal, context would have to be used to determine whether fixed or floating representation should be used. The rule resolving the ambiguity would be an exception to the rules used elsewhere in the language. Moreover, embedded computer system programmers are accustomed to the use of the decimal point to distinguish fixed and floating point values.

J3.2-12.

Limiting the number of digits to 1000 certainly is no real restriction. Imposing some upper limit is a convenience to a translator implementer and not a burden to a programmer. In fact, a lower limit, e.g., 100, would undoubtedly be quite acceptable.

### 3.2.3. Representation Specification

J3.2-13.

UNSIGNED representation is not permitted unless a range specification states that only non-negative values are expected, thereby enforcing our (and the IRONMAN's) view that representation specifications cannot change the abstract properties of a type. Representing values without a sign bit only makes sense if the values are non-negative.

It should be noted also that UNSIGNED does not extend the range of values representable with a given precision, e.g., FIXED\_MAX(S, R, STD) is the same for SIGNED as well as UNSIGNED representations. This once again is consistent with our position that representation attributes cannot affect abstract properties. Note that if an implementation supports a precision level greater than standard, say, LONG, such that STD values are represented in 16 bits and LONG values in 32 bits, then

```
VAR X: FIXED(0,2,LONG) [0:2*FIXED(0,2,STD)] REP FIXED_REP (16, UNSIGNED);
```

declares X to be a 16-bit unsigned integer value. Note that we would not want UNSIGNED to be an abstract attribute, since this would then give us two ways of declaring unsigned integer values of 16-bits. However, our design does imply that if support for unsigned integers of one word in length is really wanted, the set of FIXED\_PRECISIONS must be extended.

J3.2-14.

It is reasonable to permit a specification of zero for a size as a means of getting rid of unneeded record components or variables. For example, a 32 bit representation of fixed point values on a 16 bit object computer might represent the values as two signed 16 bit values. When a double precision value's range is sufficiently small that it can be represented in 16 bits, then the size of one component should be set to zero.

There does not seem to be any reason for permitting values to be represented in more than the default number of bits for a given precision.

### 3.2.4. Operations and Expressions

#### J3.2-15.

We follow the IRONMAN suggestion (3-1B) of providing some infix operators with no language-specified definition.  $**$  is one of these operators. Exponentiation is traditionally a difficult operator to define satisfactorily in a programming language. We follow the IRONMAN by permitting an appropriate definition to be provided for each application area, rather than building a specific definition into the language.

#### J3.2-16.

It is reasonable to specify that integer values must have a step size of one, and this is the effect of requiring a scale of zero. One design option is to permit values of any radix and a scale of zero to be considered an integer-expression, but this is not fully consistent with constraints elsewhere in the language that radices be identical for operands of infix operators and in assignments. (These constraints are consistent with not providing unneeded generality.) It seems simplest and most consistent with our decision to make radix 2 the standard to also constrain integer values to this radix. Any other decision seems to provide just non-useful generality.

#### 3.2.4.1. Rounding Control

#### J3.2-17.

As the paper by Fisher and Wetherall points out, there are basically two choices for truncation rules -- toward zero or toward minus infinity. For fixed point arithmetic, choice of a truncation rule only has a significant effect on the result of integer division. Fisher and Wetherall recommend truncation toward minus infinity because this rule yields more useful regularities than the truncation toward zero rule. In particular, the property  $(X + nY)/Y = X/Y + n$  holds for truncation towards minus infinity but not for truncation toward zero. With truncation toward minus infinity, the sign of the remainder is the sign of the divisor, whereas with truncation toward zero, the sign of the remainder is the sign of the dividend. For most computers, integer division truncates toward zero; consequently, this truncation rule can in general be implemented more efficiently on current hardware. Although the abstract properties of truncation toward minus infinity are somewhat more pleasant, these properties are apparent only when the dividend and divisor are opposite in sign. We consider the efficiency considerations of greater importance in embedded computer applications than the regularities of the truncation toward minus infinity definition, and so have chosen truncation toward zero as our truncation rule.

J3.2-18.

The round to even rule is unbiased and is the rule recommended for floating point, so we use it for fixed point as well, for uniformity.

#### 3.2.4.2. Infix Operators

J3.2-19.

Radix conversions are expensive and unlikely in practice, so it is reasonable to require that the radices of the operands be identical, and that the radix of the result be the radix of the operands. The scaling rules for addition, subtraction, and multiplication are appropriate for "exact" computations, i.e., the rules preserve fraction bits, at the possible expense of magnitude bits (i.e., at the cost of increased possibility of OVERFLOW exceptions being raised). The scale of the result for MOD follows directly from the definition of the MOD operator. The precision rule is appropriate given our desire to limit standard implementations to a single precision level. Consequently, the product of two single precision values should not be defined to yield a double precision result. Moreover, our precision rule conforms to custom, e.g., the product of two (single precision) integers customarily yields a single precision result.

J3.2-20.

It should be noted that although we define the semantics of integer division in terms of converting exact results, this does not imply implementation difficulties, since the default scaling rules conform exactly to what is provided by most computer division instructions, e.g., the exact result of 5/3 truncated to an integer is 1.

#### 3.2.4.3. Prefix Operators

#### 3.2.4.4. Scaling Infix Operators

J3.2-21.

Default fixed point scaling rules are never exactly what is required for applications. Every default choice has some disadvantage that is sometimes significant. Consequently, there is a need for programmers to explicitly control the scaling of results. In particular, in a language with a finite set of precisions, there are always cases in which results of the desired step size cannot be obtained unless scaling operators are built into the language. For example, if A and B have a scale of S and precision P and it is desired

that the product of A and B also have scale S and precision P (i.e., fraction digits of the product are to be truncated or rounded off) then A\*B with precision P and scale S+S may well overflow, i.e., the desired effect cannot be obtained by first computing A\*B and then rescaling the result. It could be obtained by computing A\*B in double precision, rescaling to the desired scale, and then converting to single precision, but this is possible only if double precision is supported. Consequently, an operator is needed for specifying the desired result scale as a parameter of the operator. These are the scaling operators defined here.

It should be noted that the same kinds of arguments apply for addition and subtraction. If A and B have the same precision and A is scale S1 and B, scale S2, with S1 > S2, if the desired result scale is S2 (i.e., the scale with fewer fraction digits), then A + B can overflow at scale S2 (the default scale) whereas A !+S2! B will not overflow.

In short, the behavior provided by the scaling operators cannot be provided in any other uniform way.

Another notational alternative is to use function notation when an explicitly specified result scale is desired. This is PL/I's approach. However, arithmetic expressions written with functional notation rapidly become unreadable. Of course, the readability of !+5! is not admirable either.

In Phase 2, we might choose another bracketting symbol than !. However, no really good symbols are available. Parentheses and brackets are easily confused with function invocation and subscription. The caret ^ is available, but it is a rather small mark to serve this purpose; its main advantage is that it avoids confusion with the conditional or (!) and concatenation (!! ) operators.

#### 3.2.4.5. Machine-Dependent Infix Operators

##### J3.2-22.

Machine-dependent infix operators are needed to provide access to results when overflow or underflow exceptions occur and to permit multiple precision arithmetic functions to be defined within the language. The use of these operators for extension purposes is perhaps less significant, since INLINE assembly language procedures (S4.8) can be defined for the same purpose. Generally the only need to access OVERFLOW results is when implementing multiple precision extensions, so in the interests of simplicity these operators could perhaps be deleted. We have provided them primarily to limit the need to write assembly language routines.

### 3.2.4.6. Relational infix Operators

### 3.2.5. Assignment

### 3.2.6. Predefined Functions

#### 3.2.6.1. TO FIXED

##### J3.2-23.

There are at least two options for the default radix and precision of the TO\_FIXED operator: the option we have chosen (radix 2 and precision STD) or having the default radix and precision be the radix and precision of the to-fixed-value. We have discarded the latter alternative in favor of having radix 2 and precision STD be the consistent defaults throughout the language. When it comes to defining defaults, uniformity is essential if confusion is to be avoided, even though the defaults in this case are probably not as desirable if the language is extended to support additional radices and precisions.

##### J3.2-24.

To ensure that the conversion is as close as possible to the exact result, the literal is passed to TO\_FIXED as a character string. This also ensures that the conversion is the same for literals and for data, as the IRONMAN requires.

##### J3.2-25.

Treating bitstring values as unsigned integers is the most convenient interpretation in most contexts. In particular, we take advantage of this interpretation in various operations on representations of enumerated values. The effect of converting a fixed point value to a bitstring and then back to a fixed point value is performed via a type-specifier, not the TO\_FIXED function. Type-specifiers are used to specify that a certain bitstring is to be considered to represent a value of a stated type. For example [FIXED(4,2,STD)]\$\$[TO\_BIT(.1\S4)] = .1\S4. Such conversions can be performed only in machine-dependent sections of code (see S3.1.4.1).

##### J3.2-26.

This satisfies IRONMAN requirement 2G.

2/15/78

Fixed Point

3.2.6.2. ABS

3.2.6.3. FIXED\_MIN and FIXED\_MAX

### 3.3. FLOATING POINT

#### J3.3-1.

The reasons for supporting multiple precision floating point by extension rather than directly are the same as for supporting multiple precision fixed point by extension.

#### 3.3.1. Attributes and Type Specification

#### J3.3-2.

Some computers may have different exponent ranges for different precisions, although this is not generally the case. Consequently, our definitions of EXPONENT\_MIN and EXPONENT\_MAX do not assume the same value for all supported levels of precision.

#### J3.3-3.

If default ranges are supported for fixed point data and for enumerated data, they should be supported for floating point data as well.

#### 3.3.2. Literals

#### 3.3.3. Representation Specifications

#### J3.3-4.

The amount of space occupied by a floating point value is not necessarily the sum of MANTISSA\_SIZE and EXPONENT\_SIZE. For example, in the DELCO M362F computer, a double precision value is stored in 64 bits, although the mantissa size is 48 bits and the exponent size is 8 bits. (The unused bits are in the middle of the mantissa.)

#### J3.3-5.

Any change in the mantissa size potentially affects the accuracy of the abstract value being stored, and representation attributes can never change the abstract behavior of a program. So control over MANTISSA\_SIZE cannot be provided. Control on EXPONENT\_SIZE does not seem useful and would seem non-uniform to programmers if mantissa sizes cannot be specified.

### 3.3.4. Operators and Expressions

#### J3.3-6.

An alternative position could be that the floating point type has no predefined operators so an implementation on a machine that had no floating point hardware would not have to provide an interpretive floating point package. On the other hand, this would effectively impair the portability goals of the language as a standard, since programs using floating point could not be executed on such a computer until a floating point package was provided. In view of the language's goals as a common language, it is perhaps better to bear the expense of providing floating point packages as a minimum implementation standard. Moreover, the IRONMAN explicitly requires floating point types and operators to be supported, so we have defined a minimum set of operators. If the IRONMAN requirement is modified to require only the ability to define floating point types and operators as an extension, our definitions here could be used in Phase 2 to specify a standard for floating point capabilities that are not necessarily part of the basic language, and hence, are not necessarily supported by all implementations (13C).

#### 3.3.4.1. Rounding Control

#### J3.3-7.

Truncating toward minus infinity is another option, but most programmers are accustomed to the "truncation toward zero" rule. Most computers do not truncate toward zero or toward minus infinity since for most computers,  $E - F = E$  if  $F$  is small enough. Our truncation rule, however, requires that  $E - F = E$  if and only if  $F$  is zero. The reason for requiring this form of truncation rule is that it is the only form that provides a useful invariant about the true value of an expression, namely, if truncation toward minus infinity is the rule, the true value is always greater than or equal to the represented value. Otherwise, the absolute value of the true value is always greater than or equal to the absolute value of the represented value. This kind of truncation rule provides useful information about approximation errors in floating point computations, whereas the truncation rule implemented by most hardware guarantees nothing about the relation between the true and represented value.

#### J3.3-8.

Two native modes are supported since many computers support both rounded and truncated floating point operations. We provide the native modes as a compromise between rules that are numerically desirable and rules that are efficient to implement. This is consistent with the general philosophy of the IRONMAN that machine-dependent computations are to be highlighted. We address this point in more detail later.

### 3.3.4.2. Infix Operators

#### J3.3-9.

The rule determining result precision is the rule customarily adopted in languages that support multiple precisions. It is a simple rule that leads to readily predictable results. Fisher and Wetherall in their paper propose a quite different position in which result precision is determined by the context of the operation. For example, if A, B, and C all have the same nominal precision, the proposed rules require that in computing  $D := A*B/C$ ,  $A*B$  be computed in double precision and the quotient be computed in single precision, but in the context:

$D := A*A + B*B - 2.*A*B;$

$A*B$  should be computed as a single precision result (even though when A and B are expected to be approximately equal in value, the products and sums should all be computed in double precision before converting to single precision to maximize the result accuracy).

This is a point that could be argued at some length, but the basic principle is that a translator should not try to outsmart a programmer. In this case, providing complicated rules to determine the precision of subexpression results makes it difficult for programmers to determine what the result precisions will be. This is acceptable if the rules always yield the intuitively correct result, but as our example of  $A*A + B*B - 2.*A*B$  shows, the rules Fisher and Wetherall propose do not in practice always produce the numerically most accurate result.

We believe that the simplest and most understandable language results when the precision of arithmetic results are not determined by how the results are being used, but instead, are determined by simple, context free rules whose effects can be readily predicted by programmers. In our language, for example, we would write:

$D := A*A + B*B - 2.*A*B \text{ WITH PREC DOUBLE};$

to ensure that products and sums are computed in double precision before converting to the nominal precision of D.

#### J3.3-10.

In specifying the rounding and truncation rules for the language, it is difficult to define rules that are both numerically acceptable and that permit efficient use of the currently ill-designed (from a numerical viewpoint) floating point hardware. The ROUNDED rule is the most numerically sound, but no existing computer implements it efficiently. It is perhaps a hopeful sign that the proposed INTEL floating point standard [Palmer, John F. "The INTEL standard for floating-point arithmetic" Proceedings CompSac '77, IEEE Catalogue No. 77CH1291-4C, Nov. 1977, pp. 107-112] specifies exactly the

correct rule and argues that it can be efficiently and economically implemented in hardware. Since there is little question that the INTEL standard is numerically superior to any implemented machine capability today in hardware, the advantages of this rule are sufficiently great that it is sensible to use it as the DoD Common Language's rule as a prod to its universal implementation.

The truncation rule is also not efficiently supported except by the proposed INTEL standard, since  $E-F = E$  for most computers if the non-zero value of F is small enough relative to E's value.

Because of this inefficiency, we have defined a NATIVE\_T and NATIVE\_R mode of computation that permits the hardware floating point implementation to be used directly. This mode can be used only in portions of programs that discriminate on the object machine (i.e., machine-dependent portions of programs). The use of NATIVE\_T or NATIVE\_R will permit efficient computation when the numerical accuracy supported by the ROUNDing and TRUNCATION rules is not needed.

#### 3.3.4.3. Prefix Operators

#### 3.3.4.4. Precision Control

##### J3.3-11.

Our form of precision control does not permit nested subexpressions with different precisions specified, e.g.,

(A\*B WITH PREC 10)/C WITH PREC 5

because we believe such expressions will be to difficult to read and that the requirement for computing subexpressions with different precisions does not in practice arise very often. When it does, programmers will have to write two expressions, in which the first assigns to a temporary variable declared by the programmer.

We do not define the WITH PREC form for fixed point operands because 1) there is no IRONMAN requirement for it; and 2) the fixed point rules forbidding implicit scale conversions make the concept of precision control less useful. For example,

D := A\*A + B\*B - 2\*A\*B WITH PREC LONG;

is not a valid fixed point expression if D is of STD precision and if A, B, and D all have the same scale; the scale of A\*A is twice the scale of D, and implicit scaling approximations are not permitted from larger to smaller scales (i.e., where fraction digits are potentially lost). To be a valid fixed point expression, a programmer would have to write:

D := TO\_FIXED(A\*A + B\*B - 2\*A\*B WITH PREC LONG, D.SCALE);

This expression is currently not permitted by the language, since the position occupied by LONG must be a manifest-integer-expression, and LONG is a manifest-enumerated-expression. However, if the IRONMAN were modified to permit or require such control, there is in principle no difficulty in defining WITH PREC for fixed as well as floating point expressions.

#### 3.3.4.5. Machine-Dependent Infix Operators

##### J3.3-12.

The machine-dependent floating point operators are intended for use in extending the precisions supported by the language without writing assembly code. Although minimizing the need for assembly code insertions is an IRONMAN goal, it is probably simpler to admit that in practice, extended precision packages will be written with assembly code insertions if they are not directly supported by an implementation, so the machine-dependent infix operators can be eliminated.

#### 3.3.4.6. Relational Infix Operators

#### 3.3.5. Assignment

#### 3.3.6. Predefined Functions

##### 3.3.6.1. TO FLOAT

##### J3.3-13.

This rule makes the implemented precision levels potentially apparent to a programmer, since TO\_FLOAT(E,P) ≠ E if the implemented precision of TO\_FLOAT(E,P) is less than the implemented precision of E. The alternative is to say that TO\_FLOAT(E,P) rounds or truncates to the specified precision, P, and then converts to the implemented precision. This hides the effect of implemented precision levels, but encourages programmers to specify precisions that match the implemented precision levels to avoid extra computations. Since the IRONMAN has already required that the effect of implemented precision levels be apparent to programmers (by requiring that computations be performed using the implemented precision levels rather than the specified precisions), our proposed TO\_FLOAT definition is more consistent with the IRONMAN than a definition that tries to hide implemented precision effects.

J3.3-14.

Note that the rule for converting character strings to floating point values does not say that the effect is the same as if the character string were written as a literal in place of TO\_FLOAT(C,P). The reason is that we want the number of significant digits in a float-literal to affect the precision with which the literal is represented. Since the number of significant digits is not known when TO\_FLOAT is applied to a charstring variable, the implemented precision of the result cannot be determined at compile-time. Consequently, we specify that the second argument of P determines the implemented precision of the result. This ensures that float-literals have the same values in programs as data when the nominal precision is explicitly specified.

3.3.6.2. ABS3.3.6.3. FLOAT MIN and FLOAT MAX3.3.6.4. EPSILONJ3.3-15.

The EPSILON function is recommended by numerical analysts, e.g., [Aird, T. J. et. al. Name standardization and value specification for machine dependent constants. SIGNUM Newsletter 9, 4 (Oct. 1974), pp. 11-13]. It is used in determining when two floating point values are "approximately" equal. It is provided as a predefined function, since if it is not provided by an implementation, it is cumbersome to define within the language.

3.3.6.5. IMP PRECISION

### 3.4. ENUMERATED TYPES

#### 3.4.1. Attributes and Type Specifications

##### J3.4-1.

Range-specs are permitted in enumerated-specs for uniformity and because when type declarations are used (as is illustrated by Example 1 in the Specification), it is useful to restrict the default range associated with the type name.

##### J3.4-2.

Although we do not really expect programmers to define enumerated types with the same set of values and different reptypes for the values, it is permitted for uniformity. Note that saying the types differ if the reptypes differ is just the usual rule, since representation properties are part of a type in our view. The current specification does not state whether reptype conversions are automatically supported in this case, but since the case is expected to be rare, it would probably be best to not require automatically defined reptype conversions. As we point out later in our discussion of User-Defined Types (S3.11), the whole area of representation specifications and operations requires further study in Phase 2. The issue of automatic reptype conversions for enumerated types can best be resolved when this further analysis is performed.

##### J3.4-3.

We believe that our approach to unordered enumerated types provides a more uniform capability than that suggested by the IRONMAN requirement for specifying subranges of "unordered" enumerated types. For example, although the IRONMAN requires iteration over unordered enumeration subranges (6E), it is not clear what such an iteration form should mean. One possibility is that the selection of values is nondeterministic, but this would simply be an unenforceable fiction; in practice, the values will be selected deterministically. If the selection is deterministic, then the so-called "unordered" types seem to have an ordering. We believe it is clearer to make a clean break between the ordered and unordered enumerated types. Making this break avoids the implementation difficulties of ensuring that even when enumerated types are given programmer-defined representations, the "ordering" implied by the E1, E2, ..., En sequence is preserved in subrange specifications.

### 3.4.2. Literals

### 3.4.3. Representation Specification

#### J3.4-4.

To permit sizes larger than that required for single precision integer values is unneeded generality, since ordered enumerated values can be used as array indices and we do not want to require all implementations to support more than one precision of fixed point arithmetic. But if an implementation has been extended to support multiple precisions, the SIZE limitation can be extended as well, as our definition states.

#### J3.4-5.

Although it is not significantly harmful to permit representation specifications use bitstring-literals whose length is less than or equal to the explicitly specified representation length (we have already stated that the representations are right-justified when just a size is specified), it is more consistent with the treatment of bitstrings elsewhere in the language to require that the lengths be identical.

### 3.4.4. Operators and Expressions

#### 3.4.4.1. Infix Operators

#### 3.4.5. Assignment

#### 3.4.6. Predefined Functions

##### 3.4.6.1. SUCC and PRED

### 3.5. BOOLEAN

#### J3.5-1.

We do not provide predefined operators for logical conjunction and disjunction of boolean values. This is a minor deviation from the IRONMAN since such operators can easily be defined by extending the definition of AND and OR to apply to boolean operands. We did not provide such a definition because we felt that the "short-circuit" (i.e., conditional) forms are the more generally useful boolean operators in programs. Providing both logical and conditional operators we felt would be confusing to programmers. In addition, providing only conditional operators provides an error detection capability that would otherwise be lacking, namely, the mistaken use of AND and OR as the conditional boolean operators. We do not think the lack of logical AND and OR will be missed, even when writing assertions, since assertions are supposed to be free of side-effects.

#### 3.5.1. Attributes and Type Specification

#### 3.5.2. Literals

#### 3.5.3. Representation Specifications

#### 3.5.4. Operators and Expressions

##### 3.5.4.1. Infix Operators

##### J3.5-2.

When compiling INLINE procedures, it is too restrictive to require that both E and F be manifest-expressions for conditional compilation to occur. Hence we define E & F and E ! F to be manifest under a wider set of circumstances. See further discussion in J4.5.5.1.

##### 3.5.4.2. Prefix Operators

### 3.5.5. Assignment

### 3.5.6. Predefined Functions

#### 3.5.6.1. HANDLER EXISTS

J3.5-3.

This function is required by the IRONMAN. Its effective use is probably limited to INLINE routines, to explicitly control the conditional compilation of code checking for an exception situation. The semantics of the construct should be modified to explicitly state that the handler-exists-function is considered a manifest expression in such cases.

#### 3.5.6.2. IS MANIFEST

J3.5-4.

This function is required by the IRONMAN. It was probably intended to be used to control conditional compilation for INLINE routines.

In S3.1.4.1, we formulated the definition of manifest-expression to ensure manifest arguments of INLINE routines would be considered manifest when the INLINE routine is invoked. Consequently, this function will return TRUE when applied to formal parameters of such routines when the corresponding arguments are manifest.

### 3.6. BIT STRINGS

We have defined bit and character strings as separate data types rather than as arrays of boolean and arrays of enumerated types. Although strings can be defined in terms of one-dimensional arrays, it is most understandable to discuss them as separate, abstract data types. For example, special literal forms must be provided for string types; the array-constructor notation would be just too unwieldy. In addition, operators such as concatenation make sense primarily for string operands; concatenation of one dimensional arrays of integers is not all that useful. For these reasons, we have chosen to treat bitstrings and character strings as separate types.

#### 3.6.1. Attributes and Type Specification

##### J3.6-1.

Zero originated indexing for bitstrings is traditional in embedded computer languages.

#### 3.6.2. Literals

##### J3.6-2.

The use of bitstring literals in programs often requires fairly long literals whose understandability is considerably enhanced by the use of blanks within the literal.

#### 3.6.3. Representation Specification

#### 3.6.4. Operators and Expressions

##### 3.6.4.1. Infix Operators

##### J3.6-3.

Not providing padding defaults for bitstring operators that require equal length operands is consistent with the IRONMAN philosophy of making explicit potentially expensive or significant type conversions. If zero padding is explicitly programmed in source code, there can be no misunderstanding on a reader's part as to what the intent is. We later provide a function that can be used to make bitstring padding easier and more understandable.

J3.6-4.

The use of a range-spec to specify a substring is consistent with its use elsewhere in the language. However, substring extraction is the single exception to what is otherwise an important uniformity in the language, namely, the length of the returned value cannot, in general, be determined until the point of call and can vary from call to call. This is the sole exception to the rule that the type of a returned value can be determined at translation time. This exception poses a difficult problem, because in principle, selection of an overloaded routine can depend on the length of the returned value. The IRONMAN's proposed solution to this problem is to limit the application of substring extraction severely (namely, extracted substrings can only be used for concatenation or assignment), thereby imposing a non-uniformity on the use of an operation in the language. We do not find the IRONMAN solution consistent with the rest of the language, and so are reluctant to adopt it. Such special-case patches to problems always cause unexpected difficulties later in the use of a language. Instead, we prefer to give additional attention to this problem in Phase 2 so we can find a uniform solution.

3.6.4.2. Prefix Operators3.6.5. AssignmentJ3.6-5.

Zero length strings are often useful degenerate cases, so it is reasonable to permit extraction of null bitstrings. An alternative solution is to raise a ZERO\_LENGTH exception in the case where the upper bound of a range-spec is less than the lower bound. The advantage here is that if the exception is suppressed, more efficient substring extraction code can be generated, since the zero-length bitstring case need not be checked for. The disadvantage is that null strings become a special case rather than a degenerate case.

3.6.6. Predefined Functions3.6.6.1. TO BITJ3.6-6.

We permit the TO\_BIT function to be applied to enumerated-expression values outside of machine-dependent sections of code because either the programmer has explicitly defined the bitstring representation of the enumerated value, or the default representation has been defined by the

language. In either case, the value of TO\_BIT is both well-defined and machine-independent. In all other cases, however, the bitstring representation of a value is potentially object-computer dependent.

### 3.6.6.2. DUP

#### J3.6-7.

The DUP function is provided primarily to give bitstrings the properties they would have if they were defined as arrays of booleans. In the array of boolean case, presumably some function would be provided to initialize the array with by setting all element values to TRUE or all values to FALSE. DUP is the analogue of that array-based function.

### 3.6.6.3. ZEROS

### 3.6.6.4. BIN\_OF, OCT\_OF, and HEX\_OF

#### J3.6-8.

The reason for excluding blanks from the permitted charstring-expression values (even though they are permitted in literal values) is that permitting blanks makes the definition of the function more complicated. If blanks are permitted, then the length of the returned result cannot in general be the same as the length of the corresponding literal, e.g., BIN\_OF ('100 001') would have to return a length 7 result because if the argument were not a manifest value, it could not be determined at translation time how many blanks are in the value. Hence the most pessimistic assumption must be made. It would be too confusing to have different types returned depending on whether the argument was a manifest expression or not.

Note that if a programmer wants to convert a string that contains blanks, he can write a routine that takes the blanks out and then apply the appropriate conversion function.

An alternative, of course, is to simply not permit blanks in bitstring literals. Then the length of the returned value can be determined based on the length of the charstring value. In practice, we do not think our definition of BIN\_OF, etc., will cause problems, however.

### 3.7. CHARACTER STRINGS

#### 3.7.1. Attributes and Type Specification

#### 3.7.2. Literals

##### J3.7-1.

Although it is reasonable for the language to be based on the ASCII character set, it is unreasonable to expect that ASCII will be the appropriate character string encoding for all application programs. Depending on the object computer and its peripheral equipment, various non-ASCII character codes might be a more appropriate default. Programs written for such non-ASCII systems will not have their understandability enhanced by explicitly overriding the ASCII default in every string literal written for the programs. Consequently, we have provided a way of replacing what is assumed to be the default representation for character string literals. Example 10 in the Specification shows how the default string encoding can be replaced. We emphasize that we do not expect the default to be replaced by individual application programmers. Instead, the illustrated method of providing a different default will be used in an application library accessed by all application programs for a particular system.

##### J3.7-2.

Whether an element of a character set should be considered a string of length one or not has been decided differently in various languages. The language C uses "C" and 'C'; one form represents a character and the other a string of length one. PASCAL treats 'C' as a character and has no strings of length one; we did not take this approach because treating 'C' as a string makes sense when varying length strings are defined (by extension).

Another possibility is to treat 'C' as a string but \HT, for example, as either a string or a character. This possibility is discussed further in conjunction with example 8.

##### J3.7-3.

An alternative design possibility to forbidding C := \HT when C is of type ASCII\_STR(1) is to consider \HT ambiguous as to whether it is a one character string or an enumerated-literal. We permit implicit type-specifiers in assignment contexts, so C := \HT could be considered equivalent to:

```
C := [ASCII_STR(1)]$$\HT;
```

if the type of \HT were defined to ambiguous. However, the need for such assignments is sufficiently rare that the "trick" of using "'\HT to resolve

the situation is probably adequate.

J3.7-4.

The whole purpose of distinguishing "charset-names" is to define how strings composed of the 64 character ASCII subset are to be encoded. The form '\C' defines explicitly which character symbols are available for direct use as basic-chars in charstring-literals. Consequently, it doesn't make sense to consider any enumerated type a character set -- only those types containing printing-char-ids should be considered charset-names.

3.7.3. Representation Attributes

3.7.4. Operators and Expressions

3.7.4.1. Infix Operators

J3.7-5.

The rule that both operands of the !! operator must have the same packing is simply a special case of the general rule (described in S4.3.2) determining which overloaded routine is to be invoked. We assume that a concatenate function has been defined to accept PACKED and DENSE charstring operands, but not mixtures of the two. Then if !! is invoked for operands with mixed packing modes, it will be ambiguous whether the DENSE or PACKED concatenate function is to be invoked, and consequently, an error will be reported at translation time.

3.7.5. Assignment

3.7.6. Predefined Functions

3.7.6.1. DUP

**Charstring**

**2/15/78**

**3.7.6.2. BLANKS**

### 3.8. ARRAYS

#### 3.8.1. Attributes and Type Specification

##### J3.8-1.

By automatically converting enumerated values to integer values when they are used for array subscripts, we satisfy in a simple and uniform way the IRONMAN requirement for routines accepting array parameters independent of the index type of the array.

##### J3.8-2.

An implementer's burden is somewhat lessened if upper bounds are placed on certain language parameters. Certainly arrays of more than ten dimensions are extremely rare in practice, especially in embedded computer applications, so an upper bound of 100 is more than adequate.

##### J3.8-3.

Since zero origin indexing is used for substring extraction, it seems best to preserve a zero origin default wherever such choices must be made.

##### J3.8-4.

An EXTENT attribute is provided since assignment compatibility is defined in terms of extents rather than bounds.

##### J3.8-5.

We permit zero extents primarily because the IRONMAN specifies that bitstrings and character strings be considered arrays of boolean and enumerated type, respectively. We have already argued that making bitstrings and charstrings a separate type is a better design decision, but to avoid deviating from the IRONMAN, we have argued that these types can be defined in terms of arrays as abstract types exported from modules. However, since strings of length zero are a useful capability, this implies that arrays with extents of length zero must also be supported. A better long-term solution might be to forbid zero length dimensions and treat BIT and CHAR\_STR as true primitives, i.e., as types not definable within the language.

J3.8-6.

Requiring that the ELEMENT\_TYPE of arrays not be ARRAY implies that if T is a type defined as:

```
TYPE T = ARRAY [0:N] OF INTEGER;
```

it is not legal to declare

```
VAR X: ARRAY [0:M] OF T;
```

unless T is a type exported from a module and is used outside the exporting module (see 3.12). The reason is that if T is not exported from a module, then T is an abbreviation for its definition and ARRAY [0:M] OF T is equivalent to writing ARRAY [0:M] OF ARRAY [0:N] OF INTEGER. But if T is exported, then T is not equivalent to its definition. An example of an exported type that can be defined in terms of arrays is the bitstring type:

```
TYPE BIT(LENGTH: FIXED(0, 2, FIXED_PRECISIONS.MAX)) =
    ARRAY [0:LENGTH] REP ARRAY REP(DENSE) OF BOOLEAN;
```

Hence, ARRAY [0:N] OF BIT(M) is legal. PASCAL, of course, permits ARRAY [0:N] OF ARRAY [0:M] OF INTEGER and defines this to be equivalent to ARRAY [0:N, 0:M] OF INTEGER, thus providing two notations for declaring the same type. (Several notations for the same concept is contrary to 1E.) Of course, we could define the two forms to be different in type, e.g., ARRAY [0:N] OF ARRAY ... is an array whose ELEMENT\_TYPE is ARRAY and ARRAY [0:N, 0:M] OF INTEGER has an INTEGER ELEMENT\_TYPE, but this seems unnecessary generality (1A).

J3.8-7.

Although permitting implicit radix conversions for integers does not seem harmful at first glance, in fact, the maximum representable values might differ from one radix to another, since fixed point values of radix 10, for example, might be represented differently in some object computers from radix 2 values. Even if this is not the case for a given implementation, it seems non-uniform to permit implicit radix conversions in special cases.

### 3.8.2. Array Constructors

J3.8-8.

We noted earlier in our discussion of fixed-literals that it is useful to define the range of a literal to be its value.

3.8.3. Representation SpecificationsJ3.8-9.

An inout parameter is always a simple-variable, i.e., it is assumed to be stored as a simple-variable would be stored. Consequently, elements of PACKED or DENSE arrays do not match the representation of inout parameters. In accordance with 11B, the representations of formal and actual inout parameters must match (so they can be passed by reference).

3.8.4. Operators and ExpressionsJ3.8-10.

We intend in Phase 2 to use brackets to enclose subscript lists instead of parentheses, but the use of parentheses was too scattered through the current document to change reliably before printing. The use of brackets is consistent with PASCAL and can be argued to make programs more understandable by clearly distinguishing function calls from subscripting.

#### 3.8.4.1. Infix Operators

#### 3.8.5. Assignment

##### J3.8-11.

Restricting assignment of arrays to arrays with matching bounds seems overly restrictive. It seems reasonable to require only that the number of dimensions and extents match.

### 3.9. RECORD TYPES

#### J3.9-1.

The record type capabilities defined here are a superset of PASCAL's capabilities in that records containing more than one tag field can be defined directly and records with constant fields can be defined. The capabilities are essentially those of EUCLID.

Requiring that records with constant and tag fields be declared as parameterized types is justified partly because such records are expected to be declared most frequently with type-names and partly because type names are needed when constructing record objects to clearly indicate what type of record is being constructed.

The ability to supply constant fields in record variables is not supported directly by any embedded computer language in current use, but provides additional notational clarity, since the values of these fields generate different record types that are not assignable one to the other. Nonetheless, the language can be simplified without significant loss of capability if the constant field capability is deleted.

#### 3.9.1. Attributes and Type Specification

#### J3.9-2.

The use of VAR to declare variable record fields is consistent with its use elsewhere in the language and serves to emphasize the difference between the constant and variable record fields. An alternative notation would be to treat the field declarations as a kind of parameter list enclosed in parentheses. Since this notation is somewhat more traditional, it perhaps should be preferred, but only further experience in using the language will indicate whether deleting VAR in record declarations is worthwhile.

We do not provide "components whose values are the dynamic values of expressions" as 3-3F requires. The primary purpose of these components, as explained in the answer to Question 45, is to support non-discriminating unions. We provide the required free union capability in another way, namely, via the type-specifier applied to bitstrings. A programmer can determine what type record a particular bitstring is assumed to be, and can then specify that the bitstring be assumed to be that type. For example, if the type is T(\RED), he can write:

```
IF ... THEN ... [T(\RED)]$$bitstring...
```

The selection of the type specifier can be conditioned on any predicate, or the argument of T can be replaced with some variable or expression computing the tag field value. The use of type-specifiers in this way is illustrated in the illustrative input-output package provided in Appendix C of the Specification.

The other capability specified in the answer to Question 45 is the ability to have components whose values are dependent on the values of other fields. We have not yet seen a convincing example of the need for this capability. Until one (or several) are presented, we feel it is best to avoid the complexity of introducing such a novel feature into the language, especially since records are already so complex (in order to satisfy the other requirements).

J3.9-3.

We have not provided an ELSE variant-spec because it does not seem to mesh well with ELSE-elements in discriminating case-statements and with ELSE-packing-specs in record-reotypes. In particular, if an ELSE variant-spec is permitted and used, then the ELSE element of a discriminating case statement or packing specification is, in a sense, ambiguous. Does the ELSE refer to the ELSE element of the variant-spec, or does it refer to all unmentioned values of a tag-name in the given case statement or packing specification? Although a language definition can resolve the ambiguity, it still presents a potential trap for unwary programmers, and so we have avoided the problem by prohibiting ELSE in a variant-spec.

J3.9-4.

The ability to restrict a variable to a specific variant is a useful and uniform generalization that follows naturally from treating variant records as parameterized types.

J3.9-5.

Although requiring the same order is not strictly necessary (the field-names and variant-names are sufficient), having the same order aids program readability and maintainability. It is also compatible with most other languages that have record types.

J3.9-6.

The rule concerning attributes of earlier field-names in a record-spec is uniform with the general principle in the language that declarations are processed in the order in which they appear in a program, and once processed, the declared identifiers are immediately available for use in subsequent declarations.

J3.9-7.

The primary advantage of forbidding variable references in record-specs is readability. When the attributes of a type can vary from one declaration to the next, it is best to highlight this variability by specifying the attributes as parameters of the type. Note that ranges, array bounds, etc. for fields can still be specified directly with non-manifest constant names (and expressions composed of such constant names), but the value of such names (and expressions) is fixed throughout the scope of the record-spec.

A second advantage of forbidding the use of variables is that an actual parameter is evaluated only once, whereupon its value is fixed for that use of the parameterized type. If variables and expressions were permitted directly in record-specs, different evaluations of the same expression could yield different values because functions are not side-effect free. Hence, the type of fields defined with such expressions would be implementation-dependent or else a special rule would have to be defined to prohibit the use of such expressions. No such special rule is needed for expressions serving as actual parameters, since we already have a general rule that the evaluation of one expression is not permitted to affect the value of another expression in the same parameter list. This rule is explained and justified in conjunction with routine invocations (Section 4.3.2).

J3.9-8.

Requiring tag-names to be parameters of a type highlights their use in specifying the properties of a type. Moreover, some specific value for the tag field must be specified in record constructors, and since this value indicates what field names can be accessed, it is important for readability and understandability that the value be specified toward the beginning of the constructor. This is ensured by requiring the value to be specified as a type parameter.

In addition, if it were optional whether tag fields were parameters of a type declaration or not (as in EUCLID), some programmers would choose one style and some another. Forcing a uniform style helps to make programs more maintainable.

J3.9-9.

Permitting a tag-name to be used in more than one variant-part does not provide additional expressive power that is likely to be used and would create additional implementation complexity. For example, consider the declaration:

```
TYPE T(TAG COLOR: (\RED, \GREEN, \BLUE)) = RECORD
  VAR A: ...;
  VAR E: ... ;
  SELECT COLOR FROM
    \RED,\GREEN -->
      VAR B: ... ;
      VAR C: ... ;
      END;
    \BLUE -->
      VAR D: ... ;
      END;
    END SELECT;
  SELECT COLOR FROM
    \RED,\BLUE -->
      VAR F: ...;
    \GREEN -->
      VAR G: ...;
      END;
    END SELECT;
  END RECORD;
```

Collecting together the variant parts for COLOR and minimizing the amount of space required is an extra complexity neither the reader nor the compiler implementer should have to cope with.

Of course, permitting multiple tag fields imposes some implementation complexity, since the compiler must check that the field names are distinct for all combinations of tag-name values. This can be done, however, by specifying for each variable field name in a given record-spec, what tag-names and tag-name values it is associated with. As long as no field-name is associated with more than one tag-name (as opposed to tag-name value), the field-names are sufficiently distinct to avoid ambiguity.

#### J3.9-10.

Integer, boolean, and enumerated are by far the most useful types for tag fields. Allowing other types such a noninteger fixed or character string is unneeded generality. Also, the constraints on these additional types are more complicated, especially when range-specs are used for variant-names.

#### J3.9-11.

A rule permitting unused argument values to be omitted would be complicated to state and understand and would not be consistent with our view that the parameters of a type determine which member of a type class is being specified. This view is predicated on all attributes having specified values.

### 3.9.2. Record Constructors

#### J3.9-12.

The usual view of constructors is that they construct a complete object. Hence, record constructors are required to define initial values for all accessible field-names.

### 3.9.3. Representation Specification

#### J3.9-13.

Representation specifications for variant records are complex because variant records are complicated. As we work out the details of representation specifications in Phase 2, it is likely that some of the retype specification forms for variant records will also need to be modified.

### 3.9.4. Operators and Expressions

#### 3.9.4.1. Infix Operators

#### J3.9-14.

Allowing two T(ANY) variables to be compared requires a runtime test of the tag fields for equality, followed by a discriminating case statement to allow the particular variant fields to be compared. It is more in keeping with the IRONMAN to require the programmer to explicitly write these runtime tests and discriminations, as our language requires.

### 3.9.5. Assignment

#### J3.9-15.

The semantics of assigning composite variables is complicated by the possibility of raising an exception in the course of performing the assignment. The proposed semantics implies that if the range or other abstract attributes of expression and variable components do not agree exactly, then if the expression and variable components are assignment compatible, some check must be performed to determine if the expression value satisfies the variable component's abstract type attributes. If this check is performed after some components have already been assigned, then the state of the variable is potentially inconsistent (or at least implementation-dependent). To avoid this, we have specified, in effect, that the RANGE and

other checks be performed before the assignment takes place. In practice, this may mean assigning the expression value to a temporary on a component-by-component basis so checks of the abstract attributes can be made for each component.

In practice, we expect expression and variable components to agree exactly with respect to abstract attributes, so no such checks need be performed, even if the representation attributes are not identical, since records will be specified with type declarations and variables will be declared with identical type parameters.

Consequently, implementations need not efficiently support record assignment between compatible, but not identical, record types, and in particular, need not efficiently support record assignments that may cause exceptions to be raised. To reduce implementation effort in supporting an operation that will seldom be needed, we could require that the types be identical, but this would be non-uniform with the rest of the language.

## 3.10. POINTER TYPES

J3.10-1.

The pointer mechanism that we provide is quite conventional and does not appear to satisfy the IRONMAN requirements very well. Before making our design decision on pointers, we investigated an alternative approach that appears to be more consistent with the IRONMAN requirements. This approach distinguished two classes of variables: regular variables (whose values could not be shared with other variables) and dynamic variables (whose values could be shared and which hence were implemented with pointers). The reason for distinguishing the two types of variables was to prevent certain errors from arising. These errors can be illustrated using the pointer semantics of the current design. Assume a procedure declared as follows:

```
PROCEDURE P(A,B: T # C,D: T # E,F: T); ...
```

where T is some non-dynamic (i.e., unshared) type, e.g., INTEGER. In the current design, it is possible to call P with the following argument list, assuming the declaration VAR PT: PTR(T);

```
P(PT@, PT@, # PT@, PT@ # PT@, PT@)
```

i.e., the same dereferenced pointer value can be passed in all argument positions. The following unpleasant effects can then occur:

- 1) unless the dereferenced values corresponding to the input arguments are copied into A and B, the value of these input parameters will be modified by assignments to C and D. Note that in general, for safety, copying will be required whenever a sharable value (i.e., a value accessed through a pointer) is passed to an input parameter that expects an unshared value. Such copying must be performed (even when T is a record or array type) whenever there is a possibility that the shared value could be modified.
- 2) the ordering of copying results into output parameters E and F will be significant. The simplest way to eliminate this problem is to prohibit dereferenced pointers as output parameters, although this is a stronger restriction than is really necessary. We will discuss weaker restrictions later.
- 3) If the procedure terminates by raising an exception, some output actual parameter may have been modified if it shares with an inout actual. This problem can similarly be resolved by prohibiting dereferenced pointers as inout parameters.
- 4) the procedure may not behave as expected if two inout actuals share, e.g., the code generated for an expression under the assumption of no sharing may not be correct. Such a code generation assumption would seem reasonable when an inout parameter is declared to be of type T instead of PTR(T), but the use of dereferenced inout actuals can violate this assumption. This is a more serious problem than the other three, since it potentially impairs optimization for all

procedures or else it subjects programmers to optimization-dependent errors if they use a procedure in a way that violates assumptions of the language.

In essence, the conventional pointer approach permits all the aliasing problems that are forbidden by the IRONMAN for non-dynamic variables.

These problems can be resolved, but only at a cost in language and implementation complexity that may outweigh the dangers inherent in the conventional pointer approach. For example, the weakest safe rule is to prohibit dereferenced pointer arguments that share the same object. This is a simple rule to enforce when there is only one dereferenced pointer argument and the routine does not import any pointer variables, but in general, a run-time check must be performed to ensure that dereferenced pointers do not point to overlapping objects, e.g., P1 that points to a record R and P1@.C, a component of R. If the forbidden sharing is found at run-time, an "ALIASING" exception should be raised. We would expect, however, that programs would generally operate with this exception suppressed, since its occurrence is a programming error rather than an exception situation that is permitted to arise in the correct execution of an algorithm. Running with this exception suppressed is equivalent to our current design.

### 3.10.1. Attributes and Type Specification

### 3.10.2. Literals and Constructors

### 3.10.3. Representation Specification

### 3.10.4. Operators and Expressions

#### 3.10.4.1. Infix Operators

#### 3.10.4.2. Other Operators

### 3.10.5. Assignment

### 3.11. SEMAPHORES

#### 3.11.1. Attributes and Type Specification

##### J3.11-1.

We have chosen to provide two kinds of semaphores, REGION semaphores for mutual exclusion and SYNCH semaphores for synchronization. Although one kind of semaphore would appear to be adequate for both synchronization and mutual exclusion, the reason we chose to provide two kinds has to do with the interaction between semaphores and abnormal termination. When a path that is in a critical region abnormally terminates, it is mandatory that the critical region be terminated in order to prevent system deadlock. There is, however, no way that an exception handler for the path can know for certain that the path was in a critical region. Even if the path set a flag upon entry to the critical region, the act of entry (i.e., the REQUEST on the semaphore) and the setting of the flag are not indivisible, so that it is possible for termination to occur after entry to the region but before the flag is set. Recording the fact that the path has successfully requested the semaphore can only be done within the REQUEST operation (which is indivisible). Abnormal termination can then automatically release the semaphore. If only one kind of semaphore were provided, abnormal termination of a path would automatically release all semaphores that the path had successfully requested and not released, including those being used for synchronization. This would have the effect of (possibly) sending false synchronization signals to other paths in the system. Consequently, we have chosen to provide two kinds of semaphores, those for synchronization (SYNCH) and those for mutual exclusion (REGION). Abnormal termination will only release REGION semaphores.

It has been suggested to us that undesirable consequences can arise from automatically terminating a critical region when the data on which the region operates is in an inconsistent state. While this may be true, it is far less desirable (in fact, unacceptable) in embedded systems to not terminate such regions and have the system deadlock. In many embedded systems, in fact, it will be the case that the data will only remain inconsistent for a very short time until it is "refreshed" with a new set of data from an input sensor.

#### 3.11.2. Constructors

#### 3.11.3. Representation Specifications

3.11.4. Operators and Expressions

3.11.5. Assignment

3.11.6. Predefined Routines

### 3.12. USER-DEFINED TYPES

#### 3.12.1. Type Declarations

##### J3.12-1.

New data types are introduced by defining modules that implement both the values of the new type and a set of primitive operations available to manipulate those values. If type-declarations were also considered to introduce new types, then the language would provide two different ways of achieving similar purposes. Furthermore, treating type-declarations as abbreviations simplifies the programming of modules defining new types (because inside the module, the new type is declared using a type-declaration, and it is most convenient there to treat it simply as an abbreviation -- see Section 4.4).

##### J3.12-2.

If variables are imported into a type-declaration, they are in effect additional implicit parameters of the type-name. When the type-name is used in several declarations, these implicit parameters can have different actual values, resulting in different types, even though the type-specs may be textually identical. We feel that programs will be clearer if such parameters are forced to be explicit, by using type-formals and type-arguments.

#### 3.12.2. Type Specifications

##### J3.12-3.

It appears that it will be possible to provide multiple representations for abstract types also, but the mechanism requires further analysis. If it proves feasible to provide such a mechanism, then one representation of a STACK type, for example, might use a list representation and another representation an array. By exporting different retype-names from a module, programmers could choose which STACK implementation they wish. However, since this capability is novel, it deserves more study than was possible in Phase 1. We propose to give it additional attention in Phase 2.

### 3.12.3. Representation Declarations

#### J3.12-4.

The value of retype parameters remains to be demonstrated. We provide the capability primarily for uniformity with other type declarations. The ability of programmers to specify new names for retypes is currently of unknown utility. If our study of retypes for abstract data types (i.e., types exported from modules) proves successful, then the capability will be more useful.

### 3.12.4. Representation Specifications

#### 3.12.5. Abstract Types

#### J3.12-5.

Enumerated types are treated specially so that non-ASCII character sets (especially those with unusual collating sequences) can be efficiently defined within the language using modules, and yet such character sets can be treated in the same way as the built-in character set. An example (BCD) is given in the specification.

#### 4. PROGRAM STRUCTURE

##### 4.1. SCOPE RULES

###### J4.1-1.

The reason that variables must be explicitly imported into closed scopes, whereas constants, types, modules, and routines need not be, is that variables are potentially alterable within the scope. Explicit importing highlights this fact (and, in addition, facilitates checking for undesirable side-effects and aliasing). The importing also makes optimization across routine invocations possible when the invoked routines have been separately compiled. This can be a significant source of object code efficiency.

###### 4.1.1. Importing

###### J4.1-2.

Although it may appear that aliasing checking of own variables of a module need only be done within the module, this is not the case. Consider the following example:

```

PROCEDURE P(# X: INTEGER) IMPORTS (ROUTINES: (G));
  ...
  G(# X);
  ...
END PROCEDURE P;

MODULE M EXPORTS G;

  VAR V: INTEGER;

  PROCEDURE F IMPORTS (DATA: (# V));
    ...
    P (# V);
    ...
  END PROCEDURE F;

  PROCEDURE G (# A: INTEGER) IMPORTS (DATA: (# V));
    ...
  END PROCEDURE G;
END MODULE M;

```

In this example, procedure F inside the module imports own variable V and invokes procedure P outside the module, passing it V as an argument. Procedure P invokes procedure G inside the module, passing it the argument it received from F (i.e., V). Because G also imports V, there is an aliasing violation, i.e., G can modify V directly or by modifying its formal parameter A. This aliasing violation will be detected at the invocation P(# V) inside F only

because P is required to import G (which it is only required to do since G imports V and P cannot import V directly).

It should be noted that this particular aliasing problem (involving own data of a module) cannot occur across separate compilation units. If procedure P and module M were in different segments (S1 and S2), S1 would have to import G from S2 (for P to invoke it) and S2 would have to import P from S1 (for F to invoke it), but this kind of circular importing is prohibited (see the constraints in S4.2). Other kinds of aliasing violations involving local data of segments can, however, occur across separate compilation units.

#### 4.1.2. Exporting

#### 4.2. SEGMENT DECLARATION

##### J4.2-1.

The uses-clause and defines-clause on segments satisfy the separate compilation requirements, 12B and 12C. In addition, as noted in S4.7, they facilitate the building of libraries (requirement 12A).

However, we feel that the mechanism will require further analysis in Phase 2, particularly with regard to usability in system construction. The parenthetical note in 12C states that "this suggests that a segment cannot be compiled until all segments from which it imports definitions and declarations, are defined." Not only will this be true, but the situation appears to be far worse. It appears that even small changes made to a segment will necessitate recompilation of all segments that directly or indirectly import that segment. This will often be the case merely to re-perform aliasing and side-effect checking. Such continual recompilation, particularly when the existing object code turns out to be valid, seems unacceptable in large system construction. It would be far more acceptable for the linker to check compatibility of object modules and indicate those that really do need recompilation, but it seems unlikely that the linker can perform all the aliasing and side effect checking required.

These problems stem from the severe restrictions on separate compilation in 12C in combination with aliasing and side-effect checking. Requiring the capability of separate compilation but with all the security and checking attainable in a single monolithic compilation appears to all but nullify the benefits of separate compilation.

In Phase 2 we will investigate modifications to the language and/or requirements that will provide a usable separate compilation facility consistent, to the extent possible, with the goal of security. (As noted elsewhere, we recommend changes to the aliasing and side-effect rules. Such changes will clearly simplify the design of the separate compilation facility.).

##### J4.2-2.

Importing a variable for input means that the value of the variable will not be changed. Consequently, a routine which has the potential for changing the value of the variable (i.e., a routine which imports it inout) cannot be invoked and hence cannot be imported. This would be equivalent to importing the variable both for input and inout.

### 4.3. ROUTINES

#### 4.3.1. Routine Declarations

##### J4.3-1.

The use of a delimiter (i.e., "#") to separate the three classes of formal parameters is not customary and will seem unnatural to anyone who is not familiar with JOVIAL. In JOVIAL (which supports only input and inout parameters) a colon is used to separate the two parameter classes. JOVIAL programmers have not only become accustomed to this usage, but find it very helpful in detecting programming errors and in making programs more understandable to others. Hoare, in a critique of JOVIAL J73, noted this feature of JOVIAL with approval.

"This [the use of the colon separator] is one of the most excellent features of JOVIAL. It makes a clear conceptual and notational distinction between input and output parameters, both at the declaration and at the call. ... It is quite the best notation and mechanism for parameters that I have seen proposed in a high level language." [Hoare, C.A.R. Critique of Standard Computer Programming Language JOVIAL (J73), p.63].

The principal advantage of the notation is that it makes clear at the point of invocation whether certain parameters may have their values altered. In JOVIAL usage, this has been found to significantly aid readability and maintainability of programs.

The principal disadvantage of the capability is in defining abstract data types, where the ordering of the arguments and their treatment as inout or output parameters might be reasonable to hide from users. The effect in this sense can be seen in our syntax for the REQUEST and RELEASE statements, which indicate that the argument is an inout parameter. Since it is certainly unclear at this time whether abstract data types will in practice have a strong use in embedded computer applications, and since it is even less clear that the # notation will deleteriously affect abstract type definition and use, we feel that the notation's demonstrated advantages significantly outweigh its problematical disadvantages.

##### J4.3-2.

Requirement 7B of the IRONMAN prohibits the use of an execution time display or equivalent mechanism. To meet this requirement, we have provided the constraint that a recursive routine may only import statically allocated data, where statically allocated data is defined to be data local to a segment scope (which includes own data of modules local to the segment scope). We have defined a recursive routine to be any routine which directly or indirectly through a sequence of calls invokes itself.

The reason that non-recursive nested routines can import the local data of their containing routines without requiring a display is that the local data of a nested routine can be preallocated with the local data of its containing routine (in fact when it can be determined that two nested routines at the same level will never be active simultaneously, their (preallocated) local data areas can be overlayed on one another for storage efficiency).

Our rule to avoid the need for a display is not minimal but was chosen for simplicity. The minimal rule, which we have derived, is that parent recursive routines may only import statically allocated data, but child recursive routines may import both statically allocated data and local data of enclosing routines up to and including the innermost parent recursive routine (parent and child recursive routines will be defined below). This rule and the definitions of parent and child recursive routines are sufficiently complex that we opted for the simpler rule. Furthermore, the likelihood that the slight additional capability provided by the minimal rule will ever be required in practice is much too small to warrant its adoption.

A parent recursive routine is a recursive routine that can directly or indirectly invoke itself without having to directly or indirectly invoke any routine within which it is lexically nested. A child recursive routine is a recursive routine that must directly or indirectly invoke a routine within which it is lexically nested in order to invoke itself. For example:

```

PROCEDURE P;
...
P1;
...
PROCEDURE P1;
...
P;
...
END PROCEDURE P1;
END PROCEDURE P;

```

In this example, procedure P is parent recursive (because it indirectly invokes itself by invoking P1), but P1 is child recursive because it only invokes itself by invoking a routine in which it is lexically nested, P. If the minimal rule were adopted, P1 could import local data of P without the need for a display. (Since each invocation of P1 is always preceded by an invocation of P, local data of each activation of P1 can be preallocated with the local data of the activation of P which preceded it). Under our simpler rule both P and P1 are recursive and hence P1 cannot import local data of P.

It should also be noted that we always stack allocate the local data of procedures at the outermost level of a compilation unit. This enables such routines to be invoked reentrantly by several paths of a parallel block (even if it is in a separate compilation unit). Requiring a REENTRANT attribute would serve no purpose since such a declaration must be checked if it can affect code generation, and checking it across separate compilation is too cumbersome. Similarly a RECURSIVE attribute is unnecessary. Within a compilation unit recursion must be determined by the compiler to enforce the importing restrictions, and recursion cannot occur across separate compilation units because of our restrictions on circularities in the uses-clauses in

segments.

#### J4.3-3.

There are four points in time at which it is possible to require the return-type of a function invocation to be fixed: compile-time, after initiating the scope containing the invocation (i.e., after all declarations in the scope have been processed), when the function is invoked, and when the function returns.

For uniformity and implementability, we have chosen constraints that require the return-type of a function invocation to be fixed after initiating the scope containing the invocation. This rule is uniform with the rule for variables, i.e., the type of a variable is fixed after scope initiation (e.g., the bounds of an array need not be known at compile-time). We are consistent with IRONMAN requirement 7D in that the type and size of a value returned by a function is determinable at translation time to the same extent that the type and size of any variable is determinable at translation time.

One reason for requiring this uniformity is that the return-value of a function (just like a variable) may be used as an argument to another routine (e.g., P(F(X)). At compile-time we must determine from the types of the arguments which routine (if it is overloaded) to invoke. Of course, if this cannot be uniquely determined at compile-time (because the value of some attribute of the type is not manifest), then the invocation is in error (i.e., choosing which (overloaded) routine to invoke cannot be decided at run-time).

Another reason for requiring this uniformity is to ensure that all stack storage needed by a scope, including the storage for temporaries, can be determined when the scope is initiated. If the return-type of a function invocation were not fixed after scope initiation, a problem would arise if an invocation occurred in a loop written with a goto statement, i.e., each execution of the function invocation might return something of a different size. (Note that a function invocation in a loop-statement (as opposed to a loop constructed with a goto) would not cause this problem because the body of the loop is an open scope that is reinitiated on each iteration of the loop.)

#### 4.3.2. Routine Invocation

##### J4.3-4.

This rule provides a point in the program where the detected lack of a return statement can be dealt with. UNINITIALIZED\_VARIABLE is raised because the return value is conceptually stored in a variable and this value is accessed on returning from a function, but it has no value in this case. Functions are expected to execute a return-stmt, but if they do not, an exception is raised. This exception will normally be suppressed, and if so, the effect is undefined if a return-stmt is not executed. However, the implementer need not attempt to determine if a return-stmt will be executed.

A warning should be issued, however, since if a simple-minded examination of the program cannot detect whether exiting by raising an exception or by returning will occur, the program will probably be difficult to maintain and understand.

#### 4.3.3. Aliasing

##### J4.3-5.

The rule preventing components of simple-variables from being used as more than one inout argument prevents two elements of arrays or two fields of a record from being passed as inout arguments to a procedure. We have taken this position because we believe the requirement to pass such components in inout position does not arise hardly at all in embedded computer applications. The rule could be relaxed to permit different fields of a record to be passed as inout arguments, as long as the fields are independent of each other (i.e., one field is not a subfield of another; this can be checked at translation time), but our rule is simpler.

##### J4.3-6.

For optimization purposes, it is necessary to prevent a variable from being used as an inout argument if it has been imported for input but not if it is passed as an input argument. The reason is that although imported variables are like implicit parameters, the restriction always enables variables imported for input to be passed by reference. If the restriction were also applied to input arguments, they, too, could always be passed by reference, but this would lead to the following undesirable (PL/I-like) consequence:  $P(X \# X)$  would be illegal but  $P((X) \# X)$  and  $P(+X \# X)$  would be legal since  $(X)$  and  $+X$  are expressions, not variables.

#### 4.3.4. Side Effects

##### J4.3-7.

The side effect rules we have provided in the language vary from those required by the IRONMAN (requirements 4C and 7E). The intent of the IRONMAN rules appears to be to prevent side effects that change the value of an expression depending on the order of evaluation of the operands. The IRONMAN attempts to do this by limiting functions to having side effects only on own data of modules that are not directly accessible at the point of function invocation. The IRONMAN rules, however, do not work successfully as the following example will show.

Consider a module in which the own data is a stack and which exports PUSH and POP (where POP is a function that has side effects on the stack). Outside the module (where the own stack is inaccessible), the following expressions would be legal (according to the IRONMAN rules): POP-POP and POP/POP. The value of these expressions would be different depending on the order of evaluation of the operands.

Consequently, we have provided different side effect rules that guarantee unique values of expressions (and also argument lists and subscript lists) regardless of the order of evaluation of operands. Our rules (which are described in the Side Effect section of the language specification) are more restrictive than the IRONMAN in that the expressions in the above example would be illegal, but are less restrictive than the IRONMAN in that we do not limit functions to only having side effects on own data of modules. It appears to us that limiting functions in that way may be too restrictive for the needs of embedded computer systems. In particular, it seems desirable to allow functions to have side effects on global system data (e.g., data declared in a globally accessible segment). Since our side effect rules guarantee unique values of expressions, argument lists, and subscript lists (regardless of whether side effects occur on data local to a module or on globally accessible data), there is no danger introduced by loosening the IRONMAN restrictions.

## 4.4. MODULE DECLARATION

J4.4-1.

The reason for distinguishing certain routines in a module as operations of a type with a T\$ prefix is due to the ability to overload routine names. Within the module, invoking an operation P with an object of type T, will invoke the P operation of the underlying type of T (which is normally what is desired). Sometimes, however, it is necessary to invoke the P operation of abstract type T within the module defining abstract type T. It is in this case that T\$P must be used. Outside the module, the T\$ form will not usually be required since most operations of abstract type T take an object of type T as an argument (and the underlying type of T is not available, so there is no ambiguity). There are, however, some operations of an abstract type that do not take as arguments any objects of that type, in particular, constructor operators (which return an object of the type). For example consider the invocation: NEW (5). Which NEW routine to invoke will not be determinable from the type of its argument if there are several NEW routines, all of which take an integer argument. In some contexts, namely data declarations and assignment statements, which NEW function to invoke can be determined. Consider, though, the following invocation: P(NEW(5)) where P is an overloaded procedure. In this case it is not possible to determine which NEW function to invoke and thus the invocation must be written P(T\$NEW(5)). Incidentally, it is precisely because of this problem that the return-type of a function cannot be considered part of its signature.

#### 4.5. CONTROL STATEMENTS

##### 4.5.1. Statements

##### 4.5.2. Simple Statements

##### 4.5.3. Structured Statements

###### J4.5-1.

Blocks are provided primarily to permit an exceptions-clause to be attached to open-scope-bodies that cannot otherwise be associated with exception handlers, e.g., the then-element of an if-stmt or the body of a routine.

##### 4.5.4. With Clauses

###### J4.5-2.

The WITH clause defined here does not have the same effect as in PASCAL, i.e., it does not permit field names of records to be referenced without stating the record variable name. It does however serve to bind a name to a variable or value of an expression so the bound-var name can be used in discriminating which variant a record variable is, or so a complicated addressing calculation need only be done once before entering the scope of the WITH clause.

###### J4.5-3.

When the bound-var of a with-clause is accessed via pointers, it is impossible to enforce the restriction that the variable's value not be changed via any of the alternate paths. In particular, it is impossible, within a case-stmt, to guarantee that the value of a variant record variable will not change to a different variant unless the variable's value is copied into the bound-var before the case-stmt is executed. Requiring such copying is safe, but usually unnecessarily inefficient, since modification of the variant record's type will usually not occur.

Another solution is to impose novel restrictions on the use of pointers, namely, to forbid  $P@ :=$  where P is a pointer. Such a restriction means that P can be changed to point to a different object, but any object P points to cannot have its type changed. The main effect of this rule is that the space occupied by the object P points to cannot be reused to hold an object that is a different variant. In addition, we must impose the restriction that a with-clause of the form WITH BV NAMING P@.R is illegal if R is a variant

record, since the restriction against assigning to P@ does not apply to assignments to field P@.R. Although these restrictions make the with clause constraints enforceable, their effect on programming style is difficult to predict. The most conservative solution is not to impose novel restrictions whose ramifications are unclear, but rather to live with the known defects of normal pointer capabilities.

#### 4.5.5. Conditional Statements

##### 4.5.5.1. If Statements

###### J4.5-4.

The rule for conditional compilation of if-stmt elements is like the rule for evaluating boolean-expressions, in that certain invalid expressions do not raise exceptions, e.g.,

```
IF A <> 0 & B/A > C THEN
```

should not cause an error at translation time even if A is a constant set to zero. This is true even if the expression is written:

```
IF A <> 0 THEN
    IF B/A > C THEN ... END IF;
END IF;
```

and A is a constant with value zero.

##### 4.5.5.2. Case Statements

###### J4.5-5.

Permitting case-stmts for any types having a defined equality operator is perhaps overly permissive, since precision conversions will in general be required. However, if no such conversions are required, quite efficient code can be generated.

###### J4.5-6.

As we have already noted in discussing the with-clause, the aliasing constraints of the with-clause cannot, in general, be readily enforced, and design alternatives that make them enforceable have other undesirable effects on execution efficiency or language complexity.

#### 4.5.6. Repetitive Statements

##### 4.5.6.1. Indefinite Loop Statement

J4.5-7.

We use the UNTIL and WHEN forms so the sense of the predicate is the same in both positions, i.e., in both cases, the loop is exited when the predicate is TRUE. We feel this is more uniform than the DO WHILE ... REPEAT UNTIL forms in which the effect of the predicate's value on loop exiting depends on whether it precedes or follows the loop body. We use the UNTIL rather than the WHILE form because END LOOP WHILE does not read well, and we have established a general style for control constructs, XXX ... END XXX, that should be maintained. LOOP is a good opening word, so END LOOP is required for the closing construct; we must then use the UNTIL/WHEN terms to keep the senses of the predicates consistent.

J4.5-8.

Permitting both the UNTIL and WHEN forms in the same loop-stmt seems to contribute only to obscure programming style. It is very rarely the case that both forms are really needed in the same loop construct.

##### 4.5.6.2. EXIT Statement

J4.5-9.

Other possibilities are to provide an EXIT WHEN form and an EXIT TO form. The EXIT WHEN form would have the same semantics as IF predicate THEN EXIT;. Rather than add another control structure alternative to the language, we have required use of the IF statement form. The IF form is required in any case, so providing the EXIT WHEN form seems to just cater to stylistic preferences on the part of programmers.

An EXIT TO form would be used in place of a signal-stmt when the exception being raised is to be handled within the routine containing the signal-stmt. If EXIT TO were provided, then the signal-stmt would be limited to raising exceptions that cause execution of a routine (or parallel path) to be terminated. There are good arguments in favor of having two control structures in this case and good arguments in favor of having only one. The arguments in favor of distinguishing EXIT TO and SIGNAL are that terminating execution of a routine is more serious than exiting a loop and readers of programs should be alerted to this by having a special control construct for the purpose. On the other hand, whether leaving a loop or leaving a routine, in both cases an exception situation has been encountered and execution is being terminated abnormally, so the same control construct should be used. We

do not feel a decision is clearcut. Our choice was, on balance, motivated by a desire to minimize the number of different constructs in the language when there was not a clear argument in favor of having separate constructs.

#### 4.5.6.3. Definite Loop Statement

J4.5-10.

Studies by Wichman [Wichman, B.A. Some Statistics from ALGOL Programs. National Physical Laboratory, CCU Report No. 11, August 1970] show that 85% of ALGOL iteration statements specify a step size of +1. Rather than provide an alternative form to handle only 10% of the cases (a step increment of -1), we limit the iteration statement to the common case.

#### 4.5.7. Exception Handling

J4.5-11.

Programmers are required to specify all the exceptions a routine can raise in its signals-clause for the same methodological reasons they are required to document their use of global data in an imports list, namely, to specify the procedural interface completely. Requiring such documentation for exceptions is consistent with requiring imports-clauses elsewhere in the language.

J4.5-12.

The treatment of exception conditions poses a significant problem in any language design. Our solution to the IRONMAN rerequirements reflects our experience with embedded computer applications programming. This experience indicates that applications programs are normally written with the assumption that OVERFLOW, UNDERFLOW, RANGE, UNINITIALIZED\_VARIABLE, etc. exceptions will not arise, and so no checking for these conditions is necessary. It is neither expected nor desired that compilers generate checking code in these cases. For example, JOVIAL, a language used in many embedded computer applications, has no way of checking for OVERFLOW at all. The lack of such a capability has not been a problem noted by JOVIAL's users. Quite to the contrary.

Consequently, if explicit suppression of exceptions is required by a language design, we would expect the language form for suppressing language-defined exceptions to appear in virtually every embedded computer application program. There is no reason to think that requiring programmers to write such a statement will make them think harder about the possibility of such exceptions arising, nor will reading such a statement provide useful information to a reader, since it will appear in virtually every program.

Consequently, we have taken the position that exceptions are normally suppressed. We expect that academically oriented reviewers of the language will find this position unattractive, but that those familiar with embedded computer programming requirements will find the position one that meets their needs directly.

It should be kept in mind, however, that by "normally suppressed" we are referring to the production version of programs. We would expect that as a compilation option, it would be possible to check for suppressed exceptions, and if they occur, the support software environment would provide either an informative diagnostic message or would invoke a debugging support tool.

In our design, we take the uniform, but somewhat extreme position that all exceptions are normally suppressed, even those raised by user-defined procedures and functions that are invoked directly (rather than indirectly by using some infix or prefix operator). The alternative of treating some exceptions (those associated with infix and prefix operators) as normally suppressed and all others as normally unsuppressed is more realistic in matching the expectations of programmers, but seems unduly non-uniform and introduces some additional complexity if a method is to be provided for suppressing the normally unsuppressed exceptions.

Several alternative design decisions were considered. First is the stated IRONMAN position, i.e., that exceptions must be suppressed explicitly but that suppression is not to be considered an assertion that the exception situation will not occur. We feel this position is untenable primarily because it eliminates many optimization possibilities. This is particularly true for RANGE exceptions. For example, global data (i.e., data referenced from several compilation units) either cannot have range checks suppressed or no optimizations based on the range of the data can be performed; the possibility that range checks might be suppressed in one of the compilation units referencing the data implies that all units must assume the maximum range of values the variable can hold. Otherwise, compiling routines separately might cause differences in behavior. Similarly, within a called procedure, range exceptions must not be suppressed for inout formal parameters or else no optimization can assume that an inout actual parameter is in range after a call. Once again, the problem is posed primarily by separate compilation -- optimizations based on ranges must be suppressed for variables passed as inout parameters to separately compiled procedures. The net effect will be to greatly limit the optimization possibilities afforded by range specifications.

The line between stating that suppressing an exception is an assertion that it will not occur and that suppression only implies removal of checking code, but no assertion is an exceedingly fine one. It seems conceptually much simpler to say that suppression is equivalent to an assertion that the situation will not arise, and that this assertion can be treated like any other assertion in guiding program optimization.

J4.5-13.

We do not provide an OTHERS handler, i.e., a handler for processing exceptions whose names are implicit. We believe that exception handling control flow should be highlighted for maintainability purposes. It may be that this position is too extreme, but until examples drawn from actual embedded computer applications show otherwise (and thereby provide guidance as to what the proper solution is), it seems prudent to follow the general rule used elsewhere in the language, namely, highlight important behavior by making it explicit. We think that the exceptions raised and handled within a routine are sufficiently important to the proper operation of the routine that they deserve to be highlighted by being explicitly named.

Our exception handling mechanism does not permit exceptions to have parameters, even though such parameters are often very useful, e.g., when exiting from a loop by raising an exception. We have not provided parameters because of their complex interaction with other features of the language. For example, it seems reasonable that identically named exceptions with different argument lists should not be permitted in the same handler-clause, i.e., exception names should not be overloaded, but it is less clear how other parameter issues should be treated. For example, we currently allow several exception names to be associated with a single open-scope-body. Should this also be allowed when the exceptions have parameters? Must the parameters be identical in number and type in this case? Deciding what is a useful and reasonably simple capability is not easy. Since the IRONMAN does not require parameters and since the issues are complex, our decision not to provide them seems well justified.

J4.5-14.

A program is not permitted to SIGNAL TERMINATED because the TERMINATE statement exists for this purpose. The TERMINATE statement cannot be replaced by the signal-stmt, however, because TERMINATE can raise the TERMINATED exception in more than one path and in a path other than the one executing the statement. SIGNAL is limited to raising an exception in the path that is currently executing.

#### 4.5.8. ASSERT Statement

J4.5-15.

The utility of making assertions executable is questionable. For example, an assertion that is made on entry to a binary search routine is that the table to be searched is sorted. Checking this assertion by executing it at run time obviates the whole purpose of the search routine. But since the ASSERT statement form is easily provided and does not interact with other language features, we have nonetheless provided it.

J4.5-16.

The constraint that the argument of ASSERT be free of side-effects is enforceable in our language, since the translator will have sufficient information available to see whether or not the expression is side-effect free.

## 4.6. PARALLEL PROCESSING

### 4.6.1. Parallel Blocks

#### J4.6-1.

The IRONMAN requires that parallel paths be scheduled on a first-in/first-out basis within priorities. This is the scheduling discipline we provide as the built-in semantics of the language. This scheduling discipline is quite general and flexible and allows more specialized scheduling disciplines to be built on top of it. It is our experience, however, that certain embedded computer applications requiring more specialized scheduling disciplines also have extreme efficiency requirements. Consequently, we have provided the flexibility for applications with such efficiency requirements to modify the built-in scheduling discipline. The parallel processing facilities of the language are defined in terms of calls on routines in a built-in parallel support module (see Appendix D of the Language Specification). Each implementation will have to provide such a run-time support module. Those applications requiring a particular scheduling discipline to be built in can substitute a module containing the appropriate routines but with their own scheduling discipline, for the built-in module.

#### J4.6-2.

The IRONMAN requires, and we do permit, nesting of parallel blocks. Because most embedded computer applications, however, will not require such a capability, we have placed certain restrictions on it in order to avoid complexity and thus keep its use and implementation simple without providing unneeded generality in the language.

Parallel blocks may be used within the body of a main segment (i.e., in a main program) or lexically nested (to any depth) within a path of an outer parallel block. This provides a very general capability. The structure of such a system is immediately apparent from its textual form (i.e., the lexical nesting structure corresponds exactly to the dynamic activation structure.) What we do not permit is the use of a parallel block within a routine. Such a capability would permit arbitrary dynamic activation structures including recursive activation of parallel blocks. In addition a path would not have the ability to raise a TERMINATED exception in a path belonging to an outer parallel control structure (as required by question 18) when such a path was "outer" on the dynamic activation sequence rather than statically outer, since the name of such an "outer" path would not be known by the lexical scope rules of the language (IRONMAN 5C). (Allowing such an "outer" path name to be passed in as a parameter would require an additional mechanism, i.e., path parameters, to be added to the language.)

Consequently, we decided to keep the nesting capabilities simple by allowing parallel blocks to be lexically nested only within paths of an outer parallel block. It should be emphasized that our restriction is only that parallel blocks may not be used within routines; there is no restriction on

routine invocation within a parallel path (i.e., a parallel path may invoke any routine, including a recursive one).

#### 4.6.2. WAIT Statement

##### J4.6-3.

The IRONMAN requires translation time constants to convert between implementation units and program units for real time. We have met this requirement with existing facilities in the language without having to explicitly provide translation time constants. The WAIT routine, which is used to delay a path for a specified real-time interval, is generic in the scale of its (fixed point) argument. Consequently, the user is free to invoke WAIT with any scale (i.e., program unit) desired. The CLOCK function, which is used to query the cumulative processing time of a path, returns a (fixed point) value with an implementation-dependent scale. The user of the CLOCK function can determine the scale with an attribute query or, instead, can simply convert the return value (using the TO\_FIXED function) to any scale (program unit) desired.

#### 4.6.3. TERMINATE Statement

#### 4.6.4. PRIORITY Statement

#### 4.6.5. CLOCK Function

#### 4.6.6. REQUEST Statement

#### 4.6.7. RELEASE Statement

#### 4.6.8. Examples

#### 4.6.9. Simulation

2/15/78

Libraries

4.7. LIBRARIES

**4.8. MACHINE DEPENDENT PROGRAMMING**

**4.8.1. Low-Level Input-Output**

**4.8.2. Target Routine Declarations**

## 5. GENERIC DEFINITIONS

### J5.0-1.

The generic definition capabilities provided in the language essentially satisfy requirement 12D of the IRONMAN. This section will first discuss the variations that exist and the reasons for the variations and will then discuss the reasons for the particular design decisions made in the generic facilities.

One minor variation with the IRONMAN is that 12D requires encapsulations to be defined with generic parameters. The reason we have chosen not to provide such a capability is that we provide nearly all the power of generic encapsulations with other features of the language, namely, parameterized type definitions and generic routines. Parameterized type definitions, which are capable of accepting TYPE parameters, may be defined within a module and exported. Outside the module any number of variables may be declared with different values of the parameters. Operations on these types may be declared within the module as generic routine definitions. Two examples of this are given in S5.2, namely the STACK and MATRIX examples.

The only capability lacking in our language that would be provided by generic encapsulations has to do with own data of the encapsulation. In our language, there will only be one copy of the own data of the module, regardless of how many variables are declared of an exported type. With generic encapsulations, however, each instantiation of the encapsulation (module) would have a new copy of the own data. Whether this capability is necessary or even desirable is not at all obvious. Only experience with writing generic definitions, abstract types, and modules can provide the answer. Note that the stack and matrix examples did not require any own data at all.

Providing generic encapsulations seems likely to add a great deal of complexity to the language. In particular, likely interactions with parameterized types and generic routines would have to be resolved (after appropriate analysis had been performed). Also it appears that a mechanism would have to be added to the language to permit explicit instantiation of generic modules. In the language as it currently stands, no explicit instantiation mechanism is needed; generic routines are implicitly instantiated when they are invoked and parameterized types are implicitly instantiated when variables are declared to be of such types. (However, an explicit instantiation mechanism may be desirable for generic routines, to facilitate the compilation process. This issue will have to be analyzed further in Phase 2.)

Consequently, we decided that to meet the requirements of simplicity (1E) and to avoid unneeded generality (1A), we would not provide generic encapsulations. Such a facility would duplicate capabilities already provided by other features while adding insignificant (and not necessarily desirable) power.

Another minor variation with the IRONMAN is that 12D requires that expressions and statements be allowed as generic parameters. The reason we do not provide such a capability explicitly is that we effectively provide the capability in another way.

In effect, a statement can be viewed as a nameless, parameterless procedure, and an expression can be viewed as a nameless, parameterless function. Since our language permits passing procedures and functions as arguments to generic routine definitions, a statement can be passed to a generic routine only by first making it into a procedure and then passing the procedure. Similarly, to pass an expression requires first making it into a function and then passing the function. This slight potential inconvenience to the programmer in the cases when he really needs to pass a statement or expression to a generic, seems more justified than complicating the language with an additional mechanism.

The basic philosophy in designing the generic definition facilities was to make generic routine definitions appear as similar to normal routine definitions as possible. The reason is to encourage the use of generic definitions, where applicable, by making them easy to use and making them appear as a natural part of the language rather than requiring programmers to learn new rules for writing generic definitions.

To meet these goals, we decided to introduce derived generic parameters based on Alphard [ref.?] notation. It is our feeling that many generic routines can be written using only derived generic parameters, i.e., where the values of the generic parameters can all be derived from the types and attributes of the arguments passed (see for example the STACK and MATRIX examples). The routine headers for such routine declarations look just like the headers for normal routines except that the type-specs for some of the formal parameters contain identifiers preceded by question marks instead of actual types or values of some of the attributes. An invocation of such a routine is indistinguishable from the invocation of a regular routine (instantiation of the generic being done implicitly based on the types of the arguments, although, as noted before, the possibility of allowing explicit instantiation needs further analysis).

For the cases where derived generic parameters are not sufficient (e.g., the TRUNCATE function example), explicit generic parameters are used. Again, routine headers for generic definitions with explicit generic parameters appear just like normal routine declarations, the generic parameters merely being additional input parameters. Also, the invocation of such a routine is indistinguishable from the invocation of a normal routine.

The only generic parameters that require somewhat special notation are procedure and function formal parameters and type formal parameters. These require somewhat special notation because procedures, functions and types are not data objects in the language (e.g., variables of such types cannot be declared).

Again, notation was chosen to make such parameters seem a natural part of the language. Procedure and function formal parameter specifications look very much like procedure and function declaration headers (they may even contain derived generic parameters, although subject to certain constraints to

prevent circular definitions). Requiring such full specification of routine formal parameters facilitates type-checking and enhances readability. The question of whether a generic definition with a routine formal parameter is instantiated with a separate body for each invocation with a different actual routine argument, or instead is instantiated as a single body with the actual routine argument passed in (at run-time) is an implementation decision (even when the actual routine has the `INLINE` attribute). In particular, our constraint that a routine passed as an actual argument may not import any data, is sufficient to permit the latter implementation method without requiring a display and without causing any other implementation problem often associated with passing routines as parameters.

Type formal parameters are either derived (i.e., a question mark identifier) or explicit (an input formal parameter of type `TYPE`). In both cases, we require routines containing such parameters to have a `WHERE` clause. The `WHERE` clause serves several purposes. First, it places constraints on which types are permissible as arguments by listing those operations that acceptable types must provide. Secondly, it serves to import those operations into the generic definition. Finally, it enhances readability and maintainability by stating in the header of the generic routine definition all required operations that must be provided by types that are acceptable as arguments.

#### 5.1. ROUTINE PARAMETERS

#### 5.2. WHERE CLAUSE

Appendix AImplementation Assessment

The difficulty and cost of a compiler implementation stems from the efficiency required in the compiler (both speed and space), the thoroughness with which it examines the source program for errors, and the quality of code it is to produce. These factors interact and conflict with each other to increase the cost. mentioned language characteristics. However, because of the intended widespread use of the language for military applications and the ultimate goal of enforced language compatibility, much of the implementation cost can be incurred during the one-time development of a common compiler front end. This front end can embody the syntax and semantic analysis, the error checking, and the global machine independent optimizations. Additional costs will be incurred each time the compiler is retargeted for a new computer. To minimize these costs and alleviate the dependency of future government programs on the retargeting schedule, major emphasis should be placed on the ease of this retargeting. Language size directly contributes to code generator size and thereby increases the number of special cases for machine specific expansion which is so important to code quality.

This effect then on both the initial and retargeting costs and code quality should be considered while reviewing the following implementation assessments.

**A.1. DEFINES/USES IMPACT**

Although the DEFINES/USES clauses on SEGMENTS directly satisfies 12B and 12C, the clauses have more impact on language support software design than any other feature in the language. The requirement to obtain attributes from independent SEGMENTs to permit compilation may radically change current system development and maintenance practices and therefore poses problems that are difficult to assess at this time. As the language exists (and as 12C notes), the compiler must have access to the SEGMENT header named in a USES clause and all contained routine headers as well as data and type declarations that may be potentially referenced. Since any declarations might use information contained in additional SEGMENTs, finding this information is not a trivial process. Moreover, these restrictions impose an ordering of program development, processing and maintenance that seems unwieldy. The compiler's job can be simplified somewhat by storing DEFINED information in an "interface file" when a segment is compiled. When that segment is USED by another segment, compilation of this segment can be done using the interface file without requiring access to the USED segment's source file. Changes to a segment that affect the interface, however, will still necessitate recompilation of all segments that directly or indirectly USE the changed segment. Some of this recompilation may be eliminated, though, through the use of an interface checker, to be described later. Version compatibility will have to be checked by this interface checker (or in its absence, by a linker).

### A.2. DATA TYPE OPTIMIZATION

The optimizations applicable to expressions in the language do not differ from those in other languages. Those optimizations are based on an examination of special cases in code generation, paying particular attention to operand values that are constant. The language additionally requires a range specification for numeric variables, which may give rise to special cases not present in other languages.

The major difficulty in generating good code for the language stems from the multiplicity of arithmetic modes--floating with specified precision and rounding/truncation/native modes, and fixed with specified result scales and machine dependent operations. In addition, floating-point operations are defined with a rounded mode of operation that is not directly implemented on most computers. Some ingenuity will have to be expended by the implementors of the code selection algorithms in determining the cases in which they may safely use the native instructions of the target machine and when a call to a run-time subroutine is required to implement an operation.

### A.3. ALIASING AND SIDE-EFFECT RULES ENFORCEMENT

No unsolvable problems of implementation arise. That is, the side-effects and aliasing rules can be enforced provided some degree of compile-time or run-time efficiency is traded off. The run-time efficiency lost, if any, may be regained by the greatly enhanced level of optimization possible when the rules are enforced strictly (as the IRONMAN requires). The major loss, other than compile-time efficiency (which we see as a real problem), is in user system maintainability (updating the interfaces, see A.3.2), and to some extent, in compiler system implementability.

#### A.3.1. Enforcement

IRONMAN (1F) states that there shall be no non-enforceable language restrictions. Thus, every restriction identified in the language is reflected in the increased complexity of the compiling system. This complexity can be mitigated depending on the level and time of enforcement.

The language specification indicates that enforcement is to be applied even in the presence of information indicating that a violation cannot occur. For example INOUT parameters are assumed to be modified at each call, whereas a compiling system may detect that no such modification is possible. Nevertheless, the language requires notification of, e.g., an aliasing violation. An advantage of this approach can be found in increased maintainability of a system. That is, by enforcing merely possible violations, a system component may be changed to modify a variable it had declared modified previously but which was in fact unmodified. The new version of the component can then be integrated into the system with no mismatched interfaces.

Enforcement may occur at several times, though it is clearly preferable to detect violations as soon as possible. Possible times of enforcement are:

- 1) At compile time, prior to optimization and code generation.
- 2) As part of an optimization phase following syntax analysis.
- 3) Subsequent to compile time as part of linking or system integration.
- 4) Independent of compilation and linking, by means of a separate interface checker-validator-maintainer.
- 5) At execution time.

In general, the later the time of enforcement occurs, the more information is available, the less analysis is required to produce a given level of enforcement, and the smaller the probability of producing spurious violation messages.

#### A.3.2. Implementation Problems and Impact

This section discusses the problems of implementing the aliasing and side-effect restrictions, and comments on the impact of that enforcement on the general language goals as stated in the IRONMAN. It begins by classifying the restrictions according to the amount of information necessary to enforce them, and then considers the problem of maintaining correct program information in the presence of multiple versions of programs and incremental updates.

##### A.3.2.1. Aliasing and Related Restrictions

In disallowing aliasing in the presence of potential modification of the aliased variable and in disallowing side-effects which cause the value of an expression to depend on the evaluation order of its operands, the language defines several contexts in which violations can be detected. These contexts may be placed in three categories:

- 1) Local to an expression or statement. Examples are:
  - . Passing the same variable twice as an INOUT argument in the same call.
  - . Passing a record and one of its components twice as OUTPUT arguments in the same call.
- 2) Local to a segment or routine. Examples are:
  - . Ensuring every global variable is explicitly imported in the using scope and known in the immediately enclosing scope.
  - . Ensuring a variable whose use is declared INPUT appears only in contexts in which constants are permitted.

3) Local to a system of independently compiled routines. Examples are:

- Ensuring a variable passed to an INOUT formal parameter of a routine is not also imported implicitly by that routine.
- Ensuring a variable explicitly imported INPUT is not implicitly imported INOUT in the same routine.
- Ensuring that function side-effects do not affect the value of an expression.
- Ensuring that every explicitly exported name in a segment is either local or imported.
- Ensuring that recursive routines import only statically allocated data.

Restrictions in categories (1) and (2) are clearly enforceable by a traditional compiler which has available to it only the text of the segment being compiled. These restrictions present no problems of implementation. The restrictions in category (3) are a different matter, however. Two kinds of problems arise if these restrictions are to be enforced at compile time. The first problem is that in compiling the routines in any given segment, A, all other segments referenced by A, directly or indirectly, must be defined and available either in original source text or as system-validated interface information specifying complete import lists, etc. That is, the system interfaces must be completely defined before any component is compiled. This problem may be termed the ab initio problem: How are programs initially compiled? The second problem arises in the maintenance of a self-consistent set of programs. If one program in the set is updated and re-compiled, all the routines having that program in their call chain are potentially in error if, e.g., the import list of that program is changed. This may be termed the update problem.

#### A.3.2.2. Maintaining Program Information

The solution to these two problems is to defer the enforcement of category (3) aliasing and side-effect restrictions to a system-level interface checker interposed between segment compilation and system integration. The compiler must generate the information derivable from independent segment compilations, which is input to the checker, but need not otherwise concern itself with these restrictions. The information to be made available by the compiler will include, for each routine:

- A list of all routines called.
- A list of all global variables directly referenced, and whether set or used.
- A list of variables, routines, modules, and segments directly imported, and the import mode of the variables.

- A list of variables passed as INOUT arguments, paired with the routines called.
- For every expression, argument-list, and subscript-list subject to side-effects, a list of functions called, variables referenced, etc.

With this information, the interface-checker can create the call graph of the system, expand the import lists for each routine, and perform the necessary validations, using the transitive closure of either the programmer-specified import lists, or the more exact information based on the program text.

The advantage of using the program text information is that fewer spurious violations will be called out, assuming the enforcement criterion is based on the possibility of a violation. On the other hand, if the less exact programmer-specified lists are used, the validation will be less subject to multiple version and sub-system update incompatibilities. From a user viewpoint it would be convenient to provide both options, but this conflicts with the IRONMAN commonality goals, unless the options are supported for all implementations.

#### A.3.2.3. Impact on IRONMAN Goals

The approach to aliasing and side-effects enforcement outlined above sacrifices compile-time (and system-integration time) efficiency to the reliability gained by system-wide interface validation.

#### A.3.3. Implementation Approach

This section discusses the optimization known as Interprocedural Data Flow Analysis (IDFA), comments on reports of two implementations, and points out how IDFA will be useful in enforcing the language restrictions.

##### A.3.3.1. Interprocedural Data Flow Analysis

The collection of set-use information on variables used as "by REF" arguments and on variables known in more than one routine is termed interprocedural data flow analysis (IDFA). This subject is at the root of our discussion on aliasing and side-effect rules enforcement, since the information required for enforcement is exactly that gathered by IDFA.

Several reports on IDFA have appeared in the literature recently but two are of especial interest in that they demonstrate a feasible implementation.

The first report, "An Interprocedural Data Flow Analysis Algorithm," by Jeffrey M. Barth in 4th ACM Symposium on Principles of Programming Languages, illustrates a method using bit vectors which produces accurate information in the presence of recursive routines and of "by REF" parameter semantics. It was applied to a PASCAL compiler consisting of 700 variables and 130 subroutines and required approximately the same amount of time to complete as the PASCAL compiler requires to compile itself. The data base used was entirely in-core and consisted of about 10,000 60-bit words. The time

required is of cubic order in both the number of variables and the number of procedures. Thus, though a PASCAL compiler is admittedly a small system, this approach appears feasible.

A second report occurs in a book by Mathew Hecht, ("Flow Analysis of Computer Programs," North-Holland, New York 1977) and discusses the IDFA method used by the SIMPL-T compiler. Of particular interest is a technique for computing the transitive closure of the routine call graph using depth-first search, a computation which is linear in the number of routine-to-routine edges in the call graph. The implementation is restricted to simple segment compilations and makes worst-case assumptions about external routines referenced from the segment.

Both of these reports, though they represent applications that are small in scale, present solutions to the IDFA problem that are applicable to the rules enforcement of the language.

#### A.3.3.2. Impact on Optimization

The use of techniques for interprocedural data flow analysis for implementing the aliasing and side-effect rules enforcement makes possible the construction of optimizers and code generators which have available to them full information about the effect of procedure calls on global variables and parameters. The value of this information will largely be felt in global register assignment and in the generation of specialized routine calling sequences which pass arguments in machine registers.

Optimization makes great demands on the absolute accuracy of the information furnished, however, and the difficulty of updating individual routines without invalidating previously compiled modules will limit the use of optimization to highly stable systems undergoing little or no maintenance.

### A.4. CONCURRENT PROCESSING EFFICIENCY

The concurrent processing capability described in the language provides the primitive facility to express parallel process invocation/termination, synchronization, and queueing. The efficiency resulting from this capability will depend largely upon the process creation, scheduling and queueing software which will in general be a part of the parallel support module. No undue overhead should result from the language primitives and the efficiency should be comparable to any high order language implementation. All mechanisms necessary for parallel processing are present; however, since the request and release primitives are not scope entrance/exit related there is a greater chance of a semaphore being left locked unintentionally and creating a permanently waiting path or a deadly embrace.

#### A.5. DATA SPACE MANAGEMENT

The proposed language requires an underlying mechanism to perform data space management. This space management is separated into two distinct functions although at some level they may conflict in their request for space.

The first function is used to provide the local storage for routines (data contained in a module nested within a routine may be considered local to the routine for storage allocation purposes). The algorithms are quite simple; space is acquired upon entrance to a routine and released upon exit or termination of the routine. Since this space acquisition/release is well-behaved, any of the usualy stacking mechanisms accommodate this space and are quite efficient. There is little compiler or run-time library impact associated with this space management except in connection with parallel paths (addressed below).

The second form of space management is much more complex. This space is used in connection with pointer variables and is acquired from what is commonly known as a "heap". No explicit release feature exists in the language; this space may be released only when no pointer exists which will locate the space. A mechanism must be provided then to free unattached space and/or compact the storage to consolidate discontiguous holes to satisfy requests for larger chunks. Significant overhead is usually associated with the management of this space. This overhead may be handled differently depending upon the constraints on and characteristics of the using embedded computer system. Some alternatives and related comments are presented below:

- Pointers may be linked together with like valued pointers to permit delinking and release of unattached data at pointer reassignment. This requires a significant data space overhead for large lists.
- Counts of pointer copies may be kept with the space to determine unreferenced space. A limited count field minimizes the data overhead and accommodates most lists but not multiply threaded lists. A maximum value inhibits further incrementation and, of course, decrementation.
- Garbage collection may be performed at space exhaustion to trace all pointers, mark all referenced space and free released space and/or compact referenced space. A severe response time penalty occurs at space exhaustion that is unacceptable in a real time system.
- Incremental tracing and marking may be done to permit continual free space release. Current studies limit the overhead for incremental collection to twice that of collection at exhaustion. The tradeoff between total CPU usage for the former and the delay associated with the latter must be evaluated for the application.
- Pointer data may be accessed through a descriptor. Explicit releases could be permitted by nullifying the descriptor. Although this may leave a pointer hanging, a run time exception could be signaled at reference to a nil descriptor.

- A transaction file may be maintained which permits incremental updating of reference counts for determining attachment. A compiler may optimize out many of the transactions by flow analysis but the overhead of the file operations are still significant.

All of these algorithms involve varying degrees of code space execution overhead and data space that in total may not be ignored. Most of these algorithms have been developed to support well defined data structures controlled totally by an underlying language mechanism. Imposing a mechanism such as this on real time computer systems with program overlays, a high percentage of space allocated to list structure, and list structures that may be moved partially or entirely to secondary store at the application discretion (such as with a store and forward message switching system) dictates that some facility must be available within the language to permit (and require) explicit space release and forego automatic reclamation by garbage collection. We recognize the potential use of dangling pointers is undesirable but system constraints may require such treatment of this space.

A complication derives from space management with concurrent processing paths. In addition to the requirement for multiple stacks (or acquisition of stack space from the heap), certain mutual exclusion areas are necessary within the space management routines. Note, however, that these may already be present for an incremental collection scheme. If stack space and heap space are acquired from the same space pool, this stack space in the heap need not enter into the garbage collection process and should be distinguished as requiring a release. This release may be prompted invisibly when exiting the closed scope which had requested it by inserting a return to a space interface routine between the caller and callee requesting the space extension.

If heap space compaction is required, all paths using the heap must be suspended to pack the used space and adjust the pointer values including those in registers. This either places a burden upon optimization or the space management mechanism.

## A.6. DEFINITION OF COMPILE TIME EXPRESSION RESTRICTIONS

### A.6.1. Introduction

This section discusses compile time expression restrictions and implementation considerations. An expression is manifest if its value is known at compile time. We distinguish between those expressions which are required to be manifest by the syntax specification and those expressions which are manifest but not required to be so by the syntax. The following syntax forms are the only ones required to be manifest by the syntax specification.

- Precision - for fixed and floating point types
- Scale - for fixed point type
- Radix - for fixed point type
- Size - of a fixed point representation
- Case-label - to specify case-element selection values
- Values in representation specifications
- Fixed point scaling operators
- Boolean-expressions - in conditional compilation of IF statements

It is reasonable for a compiler to evaluate constant expressions at compile time in all contexts in the interest of improved optimization and compile time error checking. However, restricting the generality of compile time expressions processed only constrains the syntax forms listed above.

In order to discuss the problems of compile time expression processing, the nature of a compiler for the language must be considered. We assume the compiler has the following characteristics. It is coded in itself. It produces code for multiple target computers which may differ in word size and number representation. Large portions of the compiler (e.g., syntax analysis and optimization) are designed to be target computer independent, but produce target specific output through the use of parameterization. Since some of the syntax forms which require manifest expressions appear in type specifications, compile time arithmetic must be performed during syntax analysis.

### A.6.2. Approaches to Arithmetic

The language requires that constant values be maintained in target computer form. An arithmetic package must be developed for each target computer. The compiler must use the appropriate arithmetic package for the selected target computer to perform all compile time arithmetic.

This approach ensures that expression results will be identical whether they are computed at compile time or object time. However, several arithmetic packages (one per target) must be developed.

#### A.6.3. User Supplied Functions

User supplied functions in compile time expressions pose the following implementation difficulties:

##### Host/Target Incompatibility

Suppose that the host and target computers are different. In order for the user routine to be executed on the host by the compiler, the routine must have been compiled for the host. But a routine compiled for the host may perform arithmetic that is incompatible with the target arithmetic. So in general, routines to be executed at compile time must be interpreted rather than compiled.

##### In-Line Functions

The problem described above could be alleviated by restricting user functions to be in-line in compile time expressions. Then the in-line substitution for the function call could be evaluated with the rest of the expression. However, such functions would have to be restricted to those whose in-line substitution reduces to a single expression. Otherwise, the compiler would have to interpret several different statement types in order to evaluate constant expressions.

##### Error Recovery

If the compiler executes pre-compiled user functions an additional error recovery problem is introduced. The compiler must protect itself and produce intelligible diagnostic messages when the user function misperforms.

#### A.6.4. Built-In Functions

There are no technical reasons, other than implementation cost, why built-in functions which are allowed in compile time expressions should be restricted. There are 68 built-in operators, counting each operator which supports n types as n operators, counting separately for rounding and truncating modes, and not including machine dependent operators. For each target computer to be supported, all of these operators would have to be implemented in a target arithmetic package which is callable by the syntax analyzer.

#### A.7. IMPLEMENTATION ESTIMATES

The following estimates assume use of a suitable HOL for implementation, that at least half of the personnel are experienced compiler writers and half of those are very senior design level with experience in global optimizers and implementation of compilers for complex languages. Implementation is assumed to be on a computer with conversational job submission, text editing, listing scanning and interactive execution. The estimates are based upon our experience with JOVIAL J3/J73/J3B, FORTRAN, PL/I, ALGOL 60/68, PASCAL, and COBOL.

The language/compiler system estimated is assumed to generate code for the host machine but is designed and developed to permit rehosting and retargeting. USES attributes are acquired from a SEGMENTS "interface file". This file will be produced as part of a compiler's output to be used by subsequent compilation, by a side effect enforcer (which will verify version compatibility, i.e., that the SEGMENT object module to be linked with a SEGMENT using it, is compatible with the version used for compiling the using SEGMENT), and ultimately by a language level debugging/data reduction package. An interface file for a SEGMENT must exist to compile a using SEGMENT. It should be noted that the optimization will only be as thorough as the USES clauses are specific. For example the language rule which only allows importing of entire variables (i.e., record fields cannot be separately imported) will degrade optimization. The language support system consists then of the compiler, the enforcer, and a set of library routines. A special linker is not expected to be required.

The estimates are as follows:

Compiler Exec	6 man months
sequences phases of compiler and interfaces with host operating system	
Analysis Phases	48 man months
lexical scan, statement driver. expression parser, declaration processor, allocator, direct code	
Optimizer	24 man months
Code Generator	24 man months
Assembler	12 man months
object module formatter and listings output	
Test Cases	12 man months
Integration and QA	24 man months
Enforcer	12 man months

**2/15/78**

**Project Manager                    24 man months**

**Library                          30 man months**

**I/O, record/string store and  
relational, space management,  
garbage collection, parallel  
path, locking/queueing, floating  
point routines, conversion routines**

**The total effort requires 18 man years and is performed in 24 months.**

**Language Specification**

**15 February 1978**

**Contract MDA903-77-C-0324**

## Table of Contents

1. INTRODUCTION .....	1-1
1.1. BACKGROUND .....	1-1
1.2. LANGUAGE OVERVIEW .....	1-1
1.2.1. Data Types .....	1-3
1.2.1.1. Fixed Point .....	1-3
1.2.1.2. Floating Point .....	1-4
1.2.1.3. Enumeration Types .....	1-4
1.2.1.4. Boolean .....	1-4
1.2.1.5. Bit Strings .....	1-5
1.2.1.6. Character Strings .....	1-5
1.2.1.7. Arrays .....	1-5
1.2.1.8. Records .....	1-5
1.2.1.9. Pointers .....	1-5
1.2.1.10. Semaphores .....	1-5
1.2.1.11. User Defined Types .....	1-6
1.2.2. Program Structure .....	1-6
1.2.2.1. Segments .....	1-6
1.2.2.2. Routines .....	1-6
1.2.2.3. Modules .....	1-6
1.2.2.4. Control Structures .....	1-6
1.2.2.5. Exception Handling .....	1-7
1.2.2.6. Parallel Processing .....	1-7
1.2.2.7. Machine-Dependent Capabilities .....	1-7
1.2.2.8. Generic Definitions .....	1-7
1.3. ORGANIZATION OF THE SPECIFICATION .....	1-8
1.4. SYNTAX NOTATION .....	1-10
2. LEXICAL STRUCTURE OF THE LANGUAGE .....	2-1
2.1. OVERVIEW .....	2-1
2.1.1. Character Set .....	2-1
2.1.2. Tokens .....	2-2
2.1.3. Identifiers .....	2-3
2.1.4. Literals .....	2-4
2.1.5. Other Symbols .....	2-4
2.1.6. Reserved Words .....	2-5
2.1.7. Key Words .....	2-5
2.1.8. Comments .....	2-7
3. TYPES AND OPERATORS .....	3-1
3.1. OVERVIEW .....	3-1
3.1.1. Declarations and Attributes .....	3-3
3.1.1.1. Type Attributes .....	3-3
3.1.1.2. Constant Declarations .....	3-6
3.1.1.3. Variable Declarations .....	3-7
3.1.1.4. Type Specifications .....	3-9
3.1.2. Literals and Constructors .....	3-11
3.1.3. Representation Specifications .....	3-12
3.1.4. Predefined Operators .....	3-13
3.1.4.1. Operators and Expressions .....	3-13
3.1.4.2. Operator Overview .....	3-19
3.1.5. Assignment .....	3-20

## Table of Contents

3.1.6. Predefined Functions .....	3-22
3.2. FIXED POINT .....	3-23
3.2.1. Attributes and Type Specification .....	3-23
3.2.2. Literals .....	3-26
3.2.3. Representation Specification .....	3-28
3.2.4. Operations and Expressions .....	3-30
3.2.4.1. Rounding Control .....	3-31
3.2.4.2. Infix Operators .....	3-32
3.2.4.3. Prefix Operators .....	3-35
3.2.4.4. Scaling Infix Operators .....	3-35
3.2.4.5. Machine-Dependent Infix Operators .....	3-37
3.2.4.6. Relational Infix Operators .....	3-38
3.2.5. Assignment .....	3-39
3.2.6. Predefined Functions .....	3-41
3.2.6.1. TO_FIXED .....	3-41
3.2.6.2. ABS .....	3-44
3.2.6.3. FIXED_MIN and FIXED_MAX .....	3-45
3.3. FLOATING POINT .....	3-46
3.3.1. Attributes and Type Specification .....	3-47
3.3.2. Literals .....	3-49
3.3.3. Representation Specifications .....	3-50
3.3.4. Operators and Expressions .....	3-51
3.3.4.1. Rounding Control .....	3-51
3.3.4.2. Infix Operators .....	3-52
3.3.4.3. Prefix Operators .....	3-53
3.3.4.4. Precision Control .....	3-54
3.3.4.5. Machine-Dependent Infix Operators .....	3-55
3.3.4.6. Relational Infix Operators .....	3-56
3.3.5. Assignment .....	3-57
3.3.6. Predefined Functions .....	3-59
3.3.6.1. TO_FLOAT .....	3-59
3.3.6.2. ABS .....	3-62
3.3.6.3. FLOAT_MIN and FLOAT_MAX .....	3-62
3.3.6.4. EPSILON .....	3-63
3.3.6.5. IMP_PRECISION .....	3-63
3.4. ENUMERATED TYPES .....	3-65
3.4.1. Attributes and Type Specifications .....	3-65
3.4.2. Literals .....	3-68
3.4.3. Representation Specification .....	3-69
3.4.4. Operators and Expressions .....	3-71
3.4.4.1. Infix Operators .....	3-71
3.4.5. Assignment .....	3-73
3.4.6. Predefined Functions .....	3-74
3.4.6.1. SUCC and PRED .....	3-74
3.5. BOOLEAN .....	3-75
3.5.1. Attributes and Type Specification .....	3-75
3.5.2. Literals .....	3-75
3.5.3. Representation Specifications .....	3-75
3.5.4. Operators and Expressions .....	3-76
3.5.4.1. Infix Operators .....	3-76
3.5.4.2. Prefix Operators .....	3-79
3.5.5. Assignment .....	3-79
3.5.6. Predefined Functions .....	3-79

## Table of Contents

3.5.6.1. HANDLER_EXISTS .....	3-79
3.5.6.2. IS_MANIFEST .....	3-80
3.6. BIT STRINGS .....	3-81
3.6.1. Attributes and Type Specification .....	3-81
3.6.2. Literals .....	3-82
3.6.3. Representation Specification .....	3-83
3.6.4. Operators and Expressions .....	3-84
3.6.4.1. Infix Operators .....	3-84
3.6.4.2. Prefix Operators .....	3-85
3.6.5. Assignment .....	3-86
3.6.6. Predefined Functions .....	3-87
3.6.6.1. TO_BIT .....	3-87
3.6.6.2. DUP .....	3-87
3.6.6.3. ZEROS .....	3-88
3.6.6.4. BIN_OF, OCT_OF, and HEX_OF .....	3-88
3.7. CHARACTER STRINGS .....	3-90
3.7.1. Attributes and Type Specification .....	3-90
3.7.2. Literals .....	3-92
3.7.3. Representation Attributes .....	3-97
3.7.4. Operators and Expressions .....	3-99
3.7.4.1. Infix Operators .....	3-99
3.7.5. Assignment .....	3-100
3.7.6. Predefined Functions .....	3-102
3.7.6.1. DUP .....	3-102
3.7.6.2. BLANKS .....	3-102
3.8. ARRAYS .....	3-104
3.8.1. Attributes and Type Specification .....	3-104
3.8.2. Array Constructors .....	3-107
3.8.3. Representation Specifications .....	3-109
3.8.4. Operators and Expressions .....	3-111
3.8.4.1. Infix Operators .....	3-112
3.8.5. Assignment .....	3-113
3.9. RECORD TYPES .....	3-114
3.9.1. Attributes and Type Specification .....	3-114
3.9.2. Record Constructors .....	3-124
3.9.3. Representation Specification .....	3-127
3.9.4. Operators and Expressions .....	3-133
3.9.4.1. Infix Operators .....	3-135
3.9.5. Assignment .....	3-136
3.10. POINTER TYPES .....	3-138
3.10.1. Attributes and Type Specification .....	3-138
3.10.2. Literals and Constructors .....	3-139
3.10.3. Representation Specification .....	3-139
3.10.4. Operators and Expressions .....	3-140
3.10.4.1. Infix Operators .....	3-140
3.10.4.2. Other Operators .....	3-140
3.10.5. Assignment .....	3-142
3.11. SEMAPHORES .....	3-143
3.11.1. Attributes and Type Specification .....	3-143
3.11.2. Constructors .....	3-143
3.11.3. Representation Specifications .....	3-144

## Table of Contents

3.11.4. Operators and Expressions .....	3-144
3.11.5. Assignment .....	3-145
3.11.6. Predefined Routines .....	3-145
3.12. USER-DEFINED TYPES .....	3-146
3.12.1. Type Declarations .....	3-147
3.12.2. Type Specifications .....	3-150
3.12.3. Representation Declarations .....	3-153
3.12.4. Representation Specifications .....	3-155
3.12.5. Abstract Types .....	3-157
4. PROGRAM STRUCTURE .....	4-1
4.1. SCOPE RULES .....	4-3
4.1.1. Importing .....	4-6
4.1.2. Exporting .....	4-6
4.2. SEGMENT DECLARATION .....	4-7
4.3. ROUTINES .....	4-11
4.3.1. Routine Declarations .....	4-11
4.3.2. Routine Invocation .....	4-15
4.3.3. Aliasing .....	4-19
4.3.4. Side Effects .....	4-20
4.4. MODULE DECLARATION .....	4-23
4.5. CONTROL STATEMENTS .....	4-29
4.5.1. Statements .....	4-29
4.5.2. Simple Statements .....	4-31
4.5.3. Structured Statements .....	4-32
4.5.4. With Clauses .....	4-33
4.5.5. Conditional Statements .....	4-35
4.5.5.1. If Statements .....	4-35
4.5.5.2. Case Statements .....	4-36
4.5.6. Repetitive Statements .....	4-40
4.5.6.1. Indefinite Loop Statement .....	4-40
4.5.6.2. EXIT Statement .....	4-42
4.5.6.3. Definite Loop Statement .....	4-42
4.5.7. Exception Handling .....	4-45
4.5.8. ASSERT Statement .....	4-51
4.6. PARALLEL PROCESSING .....	4-52
4.6.1. Parallel Blocks .....	4-54
4.6.2. WAIT Statement .....	4-57
4.6.3. TERMINATE Statement .....	4-59
4.6.4. PRIORITY Statement .....	4-61
4.6.5. CLOCK Function .....	4-62
4.6.6. REQUEST Statement .....	4-63
4.6.7. RELEASE Statement .....	4-64
4.6.8. Examples .....	4-65
4.6.9. Simulation .....	4-68
4.7. LIBRARIES .....	4-70
4.8. MACHINE DEPENDENT PROGRAMMING .....	4-71
4.8.1. Low-Level Input-Output .....	4-71

## Table of Contents

4.8.2. Target Routine Declarations .....	4-74
5. GENERIC DEFINITIONS .....	5-1
5.1. ROUTINE PARAMETERS .....	5-5
5.2. WHERE CLAUSE .....	5-8
Appendices	
A. Consolidated Syntax .....	A-1
B. Cross-Reference Index .....	B-1
C. Input-Output .....	C-1
D. The Basic Semantic Framework .....	D-1
D.1 Values and Variables .....	D-1
D.2 Procedures .....	D-2
D.3 Types and Type Classes .....	D-3
D.4 Modules .....	D-4
D.5 Operator Extension .....	D-5
D.6 Standard Prelude Operations .....	D-8
D.7 Parallelism .....	D-12

## 1. INTRODUCTION

### 1.1. BACKGROUND

This preliminary language design specification was developed to satisfy the DoD IRONMAN Requirements for High Order Computer Programming Languages, July 1977. Our design reflects an intensive exploration of how to satisfy all the IRONMAN requirements in a uniform and minimally complex way. We have resisted introducing novel concepts into the language except where interaction with various requirements made novel solutions necessary. The resulting language design is a good basis for proceeding to a final design and for evaluating where some modifications of the IRONMAN's specific requirements might be appropriate.

### 1.2. LANGUAGE OVERVIEW

This language contains many capabilities, but none that are not required by the IRONMAN. In fact, our few deviations from the IRONMAN have been motivated by the desire to reduce the capabilities of the language to the minimum needed for embedded computer applications and to reduce the difficulty of producing a production compiler. Further simplifications can be made, but only by relaxing some IRONMAN requirements more significantly than we felt appropriate in the preliminary design phase. (The purpose of this phase,, after all, was to explore the feasibility of satisfying all the requirements in a single language.) In our analysis of the language with respect to the IRONMAN (in the accompanying Final Report), we point out areas where IRONMAN requirements can be modified to significantly simplify the language without necessarily reducing the language's ability to meet DoD needs.

Our design has been strongly influenced by the maintainability requirements of embedded computer programs. We have not attempted to minimize

the key strokes needed to code programs in the language. Instead our goal has been to require explicit specification of a programmer's intent. Features that make programs easier to write have been excluded if they would adversely affect readability or maintainability. This is a decision well justified by Requirement 1C, but it is worth emphasizing here because this design goal has received so much more emphasis in our design than has previously been the case in language designs. Abbreviations, defaults, and stylistic options are minimized, even though they can make programs easier to write.

Another basic premise of the language design has been to provide an orderly means of satisfying different needs of different application areas. Not all embedded computer applications require double precision fixed (or floating) point arithmetic. In some application areas, use of matrix arithmetic notation would make programs easier to understand, but other areas need no support for matrix operations at all. Varying length character strings are used in some but not all areas. In short, although the different application areas have a great commonality of programming requirements, there are enough "minor" variations that some means of adapting the language to these special needs is necessary. This is why the IRONMAN requires support for generic definitions and "overloading" of infix operators (7A). Our design reflects this situation by providing means (within the language) of extending the definition of built-in operators and types to meet the special needs of each application, without requiring that every application pay a price for unneeded capabilities. For example, fixed and floating point arithmetic is required only to support a single level of precision, although some applications will need double precision (or greater) fixed point support and some will need double precision floating point support. Few applications will require both.

The basic set of data types defined in the language are fixed point, floating point, enumerated, boolean, bitstring, character string, array, record, pointer, and semaphore. Since understanding a data type means understanding what operations can be performed on values of the type and what the effect of these operations is, we have organized our discussion of data types so all information relevant to the use of a particular data type is found in one section of the language specification.

The data types and operations specified in this document are considered "predefined" for the language, in the sense that every implementation must support at least the capabilities specified here. The language permits additional capabilities to be defined using the language's type definition features. It is likely, however, that certain basic extensions (e.g., multiple precision fixed or floating point arithmetic) will be directly supported by some implementations, since this will permit more efficient object code to be generated. Since the extensions are defined within the language, however, all implementations can provide the same set of extended capabilities, if this is necessary (e.g., to use a program originally written for another application area whose compiler had certain extensions built-in).

### 1.2.1. Data Types

#### 1.2.1.1. Fixed Point

The fixed point data type in this language is, because of the IRONMAN requirements, somewhat different in its generality than is found in other languages. Integers are provided as a special case of fixed point. The basic fixed point type provided by the language is quite similar to that provided in other embedded computer languages (e.g., JOVIAL, TACPOL, and CORAL-66) in that it assumes fixed point values are binary numbers with an implicit binary

point. The language provides a means of specifying the number of fraction digits in the fixed point value and default rules for adjusting the implied binary point when values are combined arithmetically. To fully meet the IRONMAN requirements, this basic fixed point type can be extended in various ways.

#### 1.2.1.2. Floating Point

The floating point data type is similar to floating point types in other languages, although the floating point type is "predefined" only for a single precision level.

#### 1.2.1.3. Enumeration Types

The enumeration types of PASCAL are supported in this language, but additional capabilities are provided as well. In particular, the representation of character strings can be specified by appropriate definitions of different enumeration types. Thus, although ASCII is the standard character string representation, EBCDIC, BCD, BAUDOT, etc. character representations can be specified in the language. Thus an application whose equipment uses EBCDIC codes can use EBCDIC strings directly in the language.

#### 1.2.1.4. Boolean

The boolean type is provided in a way that is customary for most languages. The only unusual capability here is that boolean expressions are required to be evaluated in "short-circuit" fashion, i.e., in evaluating an expression of the form

$I \leq N \& A(I) = 0$

if  $I \leq N$  is FALSE,  $A(I)$  is not evaluated.

#### 1.2.1.5. Bit Strings

Strings of bits are provided as a predefined type in the language, although it is possible to define bit strings as arrays of booleans. Only fixed length bitstrings are directly supported.

#### 1.2.1.6. Character Strings

Fixed length character strings are also provided as a predefined type in the language, although these too could be defined in terms of arrays.

#### 1.2.1.7. Arrays

The array capabilities of the language are very similar to those provided for PASCAL.

#### 1.2.1.8. Records

Record capabilities provided in the language are a superset of those provided in PASCAL and are modelled, to a large extent, after the capabilities provided in EUCLID. The ability to directly define records with several variant parts and with constant fields is supported in the language.

#### 1.2.1.9. Pointers

Pointers are provided in the language to permit the construction and manipulation of dynamically allocated objects. The pointer capability is essentially that of PASCAL.

#### 1.2.1.10. Semaphores

Semaphores are provided for mutual exclusion and synchronization of parallel paths.

#### 1.2.1.11. User Defined Types

A type definition facility is provided in the language. This facility can be used both to define abbreviated names for types (this is the capability provided in PASCAL), and, in conjunction with the module capabilities of the language, to define new data types and operations on them.

#### 1.2.2. Program Structure

##### 1.2.2.1. Segments

Segments are used to define separately compilable program units along with their interfaces to other segments. Segments can also be used to build libraries of routines and declarations.

##### 1.2.2.2. Routines

Routines are portions of programs that may be invoked from many different places, possibly with parameter values. Both procedure and function routines are provided. Routines must explicitly state which global variables they use to facilitate alias and side-effect checking.

##### 1.2.2.3. Modules

Modules are used for packaging definitions for the purpose of controlling access to them. Modules can be used in conjunction with the type definition facilities to define new data types with their operations.

##### 1.2.2.4. Control Structures

A variety of control structures is provided including capabilities for repetition (definite and indefinite looping), conditional execution (if-then-else and case), exiting loops, and branching to labels.

#### 1.2.2.5. Exception Handling

Capabilities are provided for signalling and handling exceptional situations. This includes both built-in and user-defined exceptions.

#### 1.2.2.6. Parallel Processing

Facilities are provided for defining and activating parallel paths. Capabilities are included for synchronization, mutual exclusion, termination, alteration of priorities, and access to the real-time clock.

#### 1.2.2.7. Machine-Dependent Capabilities

Capabilities are provided for including machine dependencies in programs in a restricted fashion. Low level I/O operations and machine-code routines can be used.

#### 1.2.2.8. Generic Definitions

Generic definition facilities are included that provide a powerful and flexible means of writing general-purpose reusable routines. The generic facilities complement the type definition and module capabilities, thus facilitating the definition of abstract data types. The generic definition facilities were designed to be easy to use and to appear as a natural part of the language. An extensive illustration of their use is in conjunction with application level I/O definition (see Appendix C).

### 1.3. ORGANIZATION OF THE SPECIFICATION

The language is defined in five main sections. Section 2 describes the lexical structure of the language, i.e., the rules for forming identifiers, etc. of the language. The structure of literals of each predefined type is described together with other information about each type in Section 3.

In Section 3, the language's data types are defined, including all infix, prefix, and other operators associated with the type. For example, the rules for adding and multiplying fixed point values are given in Section 3.2, which discusses all aspects of the fixed point type. The rules for assigning to variables are also given in Section 3 for each data type.

In Section 4, we describe the structure of programs in the language, namely, the separate compilation form, scope rules, functions and procedures, modules, statements and control structures of the language, the parallel processing language capabilities, and the treatment of object computer dependent code.

In Section 5, we describe the concept of generic definitions, an important capability of the language.

Various Appendixes are devoted to other topics. Appendix A contains a consolidated listing of all the syntax productions given in the main body of the report. Appendix B is a cross reference listing of the elements comprising these productions. This cross reference listing can be used to find where in the report the various constructs are defined, used, and explained. Appendix C contains an extended example of the use of the facilities of the language, and shows how standard I/O packages can be defined using the language. Appendix D describes the basic semantic structure of the language and the information that is needed to extend the infix and prefix operators of the language for user-defined data types such as matrices and varying length

strings. Appendix D in the final version of this report will describe the operations for each predefined data type in detail, but in the current version, we only give examples of the kind of information that will ultimately be presented in the Appendix.

A justification of the language design decisions reflected in this report is given in an accompanying report. References to justification notes have the form [See Jx.y-n], where x and y are section numbers and n uniquely identifies each justification note within a subsection. The organization of the justification notes parallels the organization of the language specification.

The language specification has been written informally in the sense that we have not been legalistic in specifying the language. We depend more on the good will of the reader than a formal specification should. Nonetheless, our specification is more precise than the PASCAL specification, which is well known to leave undefined several important details of the language (e.g., the type matching rules for parameters; the effect of a case statement when one of the alternatives is not selected, etc.). We in fact go into quite a bit of detail in describing the semantics of well-formed constructs in the language, and this contributes to the size of this report. This specification provides a good basis for a formal definition.

Our specification of each construct of the language has the following structure:

- . an informal summary of the purpose of the construct being defined;
- . a syntactic specification of the construct;
- . the semantics of the construct, with individual semantic rules given in separate paragraphs.
- . examples of the construct's correct use;

- . a specification of well-formation constraints not expressed by the syntax; each of these constraints separately specifies an error condition to be detected at translation time or at execution time;
- . exception conditions associated with error conditions detected at execution time; the lack of a specification of exception conditions implies violation of constraints can be determined at translation time;

The semantics descriptions are always written with the assumption that the program is well-formed, i.e., that the constraints are satisfied. Some readers may wish to read the constraints subsections before the semantics subsections.

The syntactic specification has been formulated for understandability, not for use in automatically generating a parser for the language. Consequently, we use many more syntax productions than are actually needed in a translator. Many productions merely serve to rename non-terminals of the language, so we can use a suggestive term (e.g., fixed-scale) instead of a syntactic equivalent, e.g., expression. Moreover, semantic constraints are associated with syntactic non-terminals such as manifest-integer-expression that would not in a translator actually be checked by the parser. In short, a reader should not take the number of syntax productions as a suggestive measure of the language's complexity, particularly in comparison with PASCAL.

#### 1.4. SYNTAX NOTATION

The notation we use for describing the syntax of the language is a variant of BNF. The differences from BNF are:

- . instead of the form

<nonterminal symbol>

we write

nonterminal-symbol

i.e., a sequence of lower case letters optionally separated with hyphens.

- . the notation

non-terminal...

means one or more repetitions of non-terminal, i.e., it is equivalent to

`<nonterminal list>`

where `<nonterminal list>` is defined as

```
<nonterminal list> ::= <nonterminal list> <nonterminal>
                      | <nonterminal>
```

- . the notation

non-terminal,...

indicates a list of non-terminals separated with commas.  
For example, C,... is equivalent to C or C,C or C,C,C etc.

- . the notation

[symbol]

means the symbol is optional.

- . curly braces are used to group symbols, e.g.,

{a b}...

stands for

```
a b ;
a b a b ;
a b a b a b ;
etc.
```

- . sequences of upper case letters are terminals symbols of the language. Brackets are underlined when they are used as symbols of the language rather than as meta symbols, e.g.,

[ b ]

represents b enclosed in brackets.

- . In some productions, the right side is replaced with

note: comment ...

This provides a description of a concept that cannot be readily described otherwise.

## 2. LEXICAL STRUCTURE OF THE LANGUAGE

### 2.1. OVERVIEW

In this section, we define the character set and lexical structure of the language. The meaning of various lexical symbols is discussed in more detail in Section 3.

#### 2.1.1. Character Set

##### Purpose

All programs may be represented using just the 64 character ASCII subset. This set of characters is subdivided into letters, digits, other, and display-chars. [See J2.1-1.]

##### Syntax

[2-1]	character	::= letter   digit   blank   other
[2-2]	letter	::= A B C D E F G H I J K L M N O  P Q R S T U V W X Y Z
[2-3]	digit	::= 0 1 2 3 4 5 6 7 8 9
[2-4]	other	::= ! "#\$%& ^()!*+,-.,/  & : ; < = > ? @ _ ^ _
[2-5]	display-char	::= blank   new-line
[2-6]	blank	::= note: the blank (space) character
[2-7]	new-line	::= note: the new line character sequence

### 2.1.2. Tokens

#### Purpose

The tokens comprising a program are defined here.

#### Syntax

[2-8] lexical-program ::= [display-char]...{lexical-token...  
[display-char...]})...

[2-9] lexical-token ::= identifier  
| literal  
| other-symbols  
| reserved-word  
| key-word  
| comment

#### Semantics

From a lexical viewpoint, a program is a sequence of lexical-tokens, optionally preceded and followed by display-chars. To determine whether a sequence of characters forms a single lexical-token or not, the following rule is used:

Characters not separated by one or more display-chars are considered to belong to the same lexical-token unless the rules defining the form of lexical-tokens (see following sections) forbid such an interpretation.

Hence blanks separating lexical-tokens may be omitted if their omission does not cause two lexical-tokens to be interpreted as a single lexical-token or as two different lexical-tokens. [See J2.1-2.]

Note also that the new-line character (or character sequence) serves to separate lexical-tokens.

#### Constraints

The sequence of lexical-tokens must form a valid program, according to the rules in Sections 3 and 4.

### 2.1.3. Identifiers

#### Purpose

Identifiers serve as names of user-defined and language-defined variables, routines, constants, types, labels, etc.

#### Syntax

[2-10] identifier ::= {letter|\!?}  
                          [letter|digit|break]...

[2-11] break ::= \_

#### Semantics

The significance of an identifier depends on the context in which it appears in a program. This significance is defined in Sections 3 and 4.

#### Constraints

An identifier may contain any number of characters. However, since the new-line sequence delimits lexical-tokens, an identifier cannot be longer than the longest line length supported by an implementation's input medium.

An identifier beginning with a ? has a special significance in formal parameter lists (see Section 5).

An identifier beginning with a \ is intended to be used as an enumerated-id, i.e., an identifier for an enumerated value. However, this is just a recommended notational convention, not a language requirement.

### 2.1.4. Literals

#### Purpose

Literals provide constant values of language-defined value types. The form and meaning of literals is defined in Section 3 for each value type.

### 2.1.5. Other Symbols

#### Purpose

Lexical-tokens not having the form of identifiers are defined here for reference. Their significance is specified in Sections 3 and 4. Some of these symbols have no language-defined meaning; they may be given a meaning by language users.

#### Syntax

[2-12] other-symbols

::=	!	!!	!*	!+	!-	!//	!MOD	
	!*	!+!	!-!	!/!	!//!	!MOD!		
	#	\$	\$\$	%	&	'	"	
	(	)	*	**	+	,	-	
	->	-->	.	/	//	\	:	
	:=	;	<	<=	=	>	>=	
	<>	?	@	_	l			

### 2.1.6. Reserved Words

#### Purpose

Reserved words have predefined meanings and may not be used for any other purpose.

#### Syntax

[2-13] reserved-word	::= AND   ANY   ARRAY   ASSERT   BIN   BLOCK   CONST   DO   ELSE   END   EXCEPT   EXIT   FALSE   FOR   FROM   FUNCTION   GO   HEX   IF   IN   INLINE   LOOP   MAIN   MOD   MODULE   NAMING   NEEDS   NIL   NOT   OCT   OF   OR   ORIF   PAR_BLOCK   PATH   PROCEDURE   RECORD   REP   RETURN   SEGMENT   SELECT   SIGNAL   TAG   TERMINATE   THEN   TO   TRUE   TYPE   UNTIL   VAR   WHEN   WHERE   WITH   XOR
----------------------	---

### 2.1.7. Key Words

#### Purpose

Key words have predefined meanings, but they may be used for other purposes as well. In general, redefining the meaning of a key word makes its language-defined meaning inaccessible in the scope containing the new definition. The keywords listed here are those used in the language definition in Sections 3, 4, and 5.

#### Syntax

[2-14] key-word	::= ABS   ALL   ARRAY REP   ARRAY STD   AS   BIT   BIT STD   BLANKS   BOOLEAN   BOOLEAN STD   BOUNDS   CHARSET   CHAR STR   CHAR STR REP   CHAR STR STD   CLOCK   CONNECT   DATA   DEFINES   DIMENSIONS   DUP   DYN SIZE   ELEMENT TYPE   ENUM   ENUM REP   ENUM STD   ENUM VAL REP   EPSILON   EXPONENT MIN   EXPONENT MAX   EXPONENT SIZE   EXPORTS   EXTENT   FILLER   MIN   FIXED   FIXED MAX   FIXED MIN   FIXED REP   FIXED STD   FLOAT   FLOAT MAX   FLOAT MIN   FLOAT STD   HANDLER EXISTS   IMPORTS   IMP PRECISION   INTEGER   MAX   LENGTH   IS MANIFEST   MANTISSA SIZE   NEW   NOM PRECISION   OBJECT MACHINE   OMIT   PACKING   PREC   PRECISION   PRED   PRIORITY   PTR   PTR STD
-----------------	--

RADIX	RECORD REP	REGION	RELEASE
REPTYPE	REQUEST	RETURNS	ROUTINES
SCALE	SEMAPHORE	SIGNALS	SIGNING
SIZE	SUCC	SYNCH	TARGET
TO_FIXED	TO_FLOAT	TYPES	UNORDERED
USAGE	USES	VALUES	VARIANTS
ZEROS			

### 2.1.8. Comments

#### Purpose

A comment is used to convey programmer intent to a reader and has no effect on a program.

#### Syntax

- [2-15] comment ::= embedded-comment  
| terminating-comment
- [2-16] embedded-comment ::= % character... %
- [2-17] terminating-comment ::= % character... new-line

#### Semantics

A comment is considered equivalent to a single blank character, and hence, serves to separate lexical-tokens, but otherwise has no effect on the interpretation of the program.

It is intended that if a terminating-comment begins a line, a reformatter will keep it at the left-most margin of a program listing. Otherwise a terminating comment will be placed at a comment margin, normally somewhere in the middle of a page. An embedded comment will be left where it is unless it is followed on its line only by blanks or tabs, in which case, it will be treated as a terminating-comment by a formatter. [See J2.1-3.]

#### Examples

1. C := D; %THIS IS A  
%MULTI-LINE COMMENT  
%THIS IS A STAND-ALONE COMMENT

type is a set of operators (procedures and functions) that manipulate the type's values. These operators give the values their meaning because they determine how the information represented by the values can be interpreted and manipulated.

In describing each type, we specify the predefined operators for the type and the attributes of the type. Attributes are a convenient way of summarizing certain useful information about properties of the type. (For example, floating point values are distinguished (in part) by their nominal precision, so NOM\_PRECISION is an attribute of floating point values. NOM\_PRECISION would have the value ten for a value of type FLOAT (10).)

We distinguish abstract and representation attributes. Abstract attributes, e.g., NOM\_PRECISION, describe properties that can affect the values produced by computations. Representation attributes, on the other hand, affect only the efficiency (time and space) with which values of a type are stored and manipulated. When we describe each type of the language, we will describe the abstract and representation attributes separately.

An important reason for distinguishing abstract and representation attributes is to define when implicit type conversions are permitted. The language permits implicit representation conversions (i.e., conversions that change representation attributes but not abstract attributes). Because of the way representation attributes are defined, representation conversions never affect the values produced by computations; such conversions only affect the time and space required to perform computations. Implicit conversions that change the abstract attributes of a value are permitted only under special circumstances, in essence, only if the conversion changes just attributes of a value and not the value itself. For example, a FLOAT (10) value can be implicitly converted to a FLOAT (11) value. Implicit conversion between type

### 3. TYPES AND OPERATORS

In this Section, we present complete definitions of the predefined types of the language. The structure of each subsection follows the same pattern.

- 0 An informal overview of the type
- 1 Major properties of the type -- its specification form in declarations, its attributes, and its subtypes
- 2 Methods of creating values of the type (literals and constructors)
- 3 Representations of the type -- the choices available and their properties
- 4 The operators predefined for a type, e.g.,
  - . infix operators
  - . prefix operators
- 5 assignment semantics and constraints
- 6 predefined functions

A type is defined by its attributes, its set of possible values, and the operators that can be applied to these values. Each subsection gives this information for a particular type and so defines that type completely.

In Section 3.1 we give an informal overview of language properties common to all types. The details of specific types are then explained in Sections 3.2-3.12.

#### 3.1. OVERVIEW

In this Section, we introduce some of the basic concepts of the language. Scope rules are defined separately in Section 4.1. Information specific to individual predefined types is given in later Sections devoted to each predefined type.

The universe of values (or objects) on which programs operate is grouped into subsets called types, and types are grouped into related types called type classes. (An example of a type is FLOAT (10), a floating point number with ten digits of precision; its type class is FLOAT.) Associated with each

### 3. TYPES AND OPERATORS

In this Section, we present complete definitions of the predefined types of the language. The structure of each subsection follows the same pattern.

- 0 An informal overview of the type
- 1 Major properties of the type -- its specification form in declarations, its attributes, and its subtypes
- 2 Methods of creating values of the type (literals and constructors)
- 3 Representations of the type -- the choices available and their properties
- 4 The operators predefined for a type, e.g.,
  - . infix operators
  - . prefix operators
- 5 assignment semantics and constraints
- 6 predefined functions

A type is defined by its attributes, its set of possible values, and the operators that can be applied to these values. Each subsection gives this information for a particular type and so defines that type completely.

In Section 3.1 we give an informal overview of language properties common to all types. The details of specific types are then explained in Sections 3.2-3.12.

#### 3.1. OVERVIEW

In this Section, we introduce some of the basic concepts of the language. Scope rules are defined separately in Section 4.1. Information specific to individual predefined types is given in later Sections devoted to each predefined type.

The universe of values (or objects) on which programs operate is grouped into subsets called types, and types are grouped into related types called type classes. (An example of a type is FLOAT (10), a floating point number with ten digits of precision; its type class is FLOAT.) Associated with each

classes is never permitted, e.g., implicit conversion between a fixed and floating point value is not permitted, since such conversions can introduce unexpected changes in the value being converted. [See J3.1-1.]

The predefined types in the language are of two classes: composite and scalar. Objects of composite types have components that can be individually accessed and possibly updated. Scalar objects do not have such components.

The predefined scalar types are fixed, float, enumerated, boolean, pointer, and semaphore. The predefined composite types are bitstring, charstring, array, and record.

### 3.1.1. Declarations and Attributes

#### Purpose

The methods of specifying the type of an identifier are described here. In particular, declaration forms for variables and constants are specified. We also present the basic principles and definitions concerning attributes of types.

#### Syntax

```
[3-1] declaration-stmt      ::= constant-declaration  
                           | variable-declaration  
                           | type-declaration  
                           | retype-declaration
```

#### Semantics

Variable and constant declarations are discussed later in this Section. Type and retype declarations are discussed in Section 3.12.

### 3.1.1.1. Type Attributes

#### Purpose

The attributes of a type define the set of values associated with the type, the representation properties of stored values, and other properties affecting the use of values of the type. An attribute-query is used to access

the values of these attributes for any variable, named constant, or expression of the type.

### Syntax

[3-2]    attribute-query	::= variable . attribute-name   constant . attribute-name   (type-name . attribute-name) . attribute-name   (expression) . attribute-name
[3-3]    attribute-name	::= fixed-attr   fixed-rep-attr   float-attr   float-rep-attr   enumerated-attr   enumerated-rep-attr   boolean-attr   boolean-rep-attr   bitstring-attr   bitstring-rep-attr   charstring-attr   charstring-rep-attr   array-attr   array-rep-attr   record-attr   record-rep-attr   pointer-attr   pointer-rep-attr   semaphore-attr   semaphore-rep-attr   defined-attr   defined-rep-attr

### Semantics

When a type class is being defined, one of the things which must be specified are its attribute names. For predefined types, names and their significance are explained in subsequent 3.x.1 discussions of each type. Values of these attributes are defined when types are instantiated, e.g., in declarations of identifiers or when the type of an expression is determined.

Each attribute itself has a type. The form (T.A).B where T is a type-name provides access to the value of B, an attribute of A's type, where A is an attribute of T. For example, (FLOAT.NOM\_PRECISION).MAX gives the value of the maximum supported floating point precision.

The value of an attribute-query is the value of the corresponding attribute for the type used in the query.

The expression in an attribute-query is not evaluated, since the type of an expression is always determinable at translation time without evaluating the expression. This means that even if an expression's value is not well-defined, the type of the expression is. [See J3.1-2.]

Attribute queries may be used to access the value of either abstract or representation attributes.

#### Examples

##### 1. V.NOM\_PRECISION

(If V is a floating point variable declared as FLOAT(10), its NOM\_PRECISION attribute is 10, and so the value of V.NOM\_PRECISION is 10.)

##### 2. (U+V).NOM\_PRECISION

(This attribute-query gives the value of the NOM\_PRECISION attribute for U + V.)

##### 3. VAR V: FLOAT(10); VAR U: FLOAT(2\*V.NOM\_PRECISION);

(This shows how attribute-queries can be used to preserve relationships between variables. Here U is declared to be a variable with double the nominal precision of V. This relation will hold even if V's nominal precision is changed during program maintenance.)

##### 4. (A/0.).NOM\_PRECISION

(Since the expression in this attribute-query is not evaluated, division by zero is not considered an error. The value of the query is determined by the NOM\_PRECISION attributes of A and 0. and the rules determining the precision of quotients.)

#### Constraints

The attribute-name used in an attribute-query must have a defined meaning for the type of the identifier or expression in the query.

### 3.1.1.2. Constant Declarations

#### Purpose

A constant declaration is used to give an identifier a value that is unchangeable throughout the scope of the declaration. Constants are called manifest constants if their values are known at translation time.

#### Syntax

[3-4]    constant-declaration   ::= CONST simple-constant,... : type-spec  
                                      ::= expression;

[3-5]    simple-constant        ::= identifier

#### Semantics

The type-spec determines the type (i.e., the type class and attributes) of the constant being declared.

The expression in a constant declaration is evaluated on each scope entry (see Section 4.1) when the declaration is evaluated. If the type of the expression is not exactly the same as the type of the constant, the expression's value is (conceptually) assigned to a variable of the constant's type (thereby invoking implicit representation or other conversions permitted in assignments), and then the value of the variable is defined as the value of each constant-name in the declaration. Declarations are evaluated in the order in which they appear in a program (see Section 4.1), so the current value of any identifier appearing in the expression is used in determining the value of the constant-name.

A constant-name may be used in any context where a constant of that type is permitted or required. A constant-name that is an array or record may have its components accessed just as if the constant-name were a variable-name, but its components can't be modified.

The type-spec is evaluated before the expression is evaluated.

Examples

1. CONST PI: FLOAT(6) := 3.14159;

(The identifier PI can be used in place of 3.14159. PI is a manifest constant.)

2. CONST A: INTEGER := B + C;

(If the values of B or C are not manifest constants, the value of A will not be determined until the scope in which A is declared is entered at run time; then the current values of B and C will be used to compute the value of A.)

Constraints

The value and type of the expression must satisfy the constraints for assigning to a variable of the type specified by type-spec.

Efficiency Considerations

It should be noted that an array or record constant must generally occupy storage at run time if it is passed as a parameter or its components are accessed by a subscript whose value is non-manifest. Of course, an array whose value is manifest can be allocated at compile time rather than at scope entry. In addition, all exported constants that are arrays or records may also occupy storage since they must be accessed by separately compiled program segments. However, if routines are compiled in-line, then it may not be necessary to store a composite constant used as a parameter to the in-line routine. Some scalar constants may also occupy space at run time.

3.1.1.3. Variable DeclarationsPurpose

A variable declaration is used to specify the type (and optionally, an initial value) of a variable. Depending on the type of the variable, it may designate scalar values (i.e., values having no components) or composite values (e.g., arrays and records).

Syntax

[3-6] variable-declaration ::= VAR simple-variable,... : type-spec  
[ := expression];

[3-7] simple-variable ::= identifier

Semantics

The type-spec specifies the type class and attributes of a simple-variable.

The type-spec is evaluated on each scope entry when the declaration is evaluated, as is the initializing expression if present. The value of the initializing expression is assigned to each of the variables, using the normal assignment rules (as specified in Section 3.x.4 for each value type). Since declarations are evaluated in the order in which they appear in a program (see Section 4.1), the current value of any identifier is used in determining the value of the variable-name.

If no (initializing) expression is present, the variable has a special inaccessible "uninitialized" value, unless the UNINITIALIZED\_VARIABLE exception is suppressed. If suppressed, the value of the variable is undefined until it is explicitly assigned a value. [See J3.1-3.]

The type-spec or pointer-spec is evaluated before the expression is evaluated.

Examples

1. CONST PI: FLOAT(6) := 3.14159;  
VAR A: FLOAT(6) := PI;

(A is given an initial value equal to the value of PI.)

2. VAR B,C: INTEGER [1:10];

(B and C are both declared to be integer variables with the range 1 through 10.)

Constraints

The value and type of the expression, if present, must satisfy the constraints for assigning to a variable of the type specified by type-spec.

3.1.1.4. Type SpecificationsPurpose

Type specifications are used to define the type of a variable, constant, or parameter. The type-spec forms are defined in subsequent 3.x.1 sections.

Syntax

[3-8]      type-spec	::= type-reference [REP retype-spec]   predefined-type [REP retype-spec]   array-spec
[3-9]      predefined-type	::= fixed-spec   float-spec   enumerated-spec   boolean-spec   bitstring-spec   charstring-spec   record-spec   pointer-spec   semaphore-spec

Semantics

The significance of a type-reference depends on the declaration of the type-class. Type-declarations and type-references are discussed in Section 3.12. (All type-specs (except array-specs) have the form of a type-reference.) [See J3.1-4.]

If REP is used in a type-spec, the default representation properties of the type class are replaced with the representation properties of the retype-spec. Examples showing this usage are given for each predefined type in later Sections. If REP is not used, the default representation properties of the type are implied.

Constraints

Where predefined types require parameters, the arguments used must correspond in number, type, and position to the specifications given in subsequent sections and in Appendix D.

If retype-specs are given, they must satisfy the constraints for the type class of the type-spec.

### 3.1.2. Literals and Constructors

#### Purpose

Literals and constructors are used to specify values of a given type. The term "literal" is used for fixed, float, enumerated, boolean, bitstring, and charstring values, while the term "constructor" is used for other value types.

#### Syntax

[3-10] literal	::= fixed-literal   float-literal   enumerated-literal   boolean-literal   bitstring-literal   charstring-literal   pointer-literal
[3-11] constructor	::= array-constructor   record-constructor   pointer-constructor   semaphore-constructor

#### Semantics

The form and meaning of each predefined literal and constructor are given in subsequent 3.x.2 Sections.

#### Constraints

Literals are considered lexical-tokens of the language, and hence the syntax productions for literals explicitly note where blanks and other display-chars are permitted. Constructors are composed of lexical-tokens, and so no explicit specification of display-chars is needed or provided.

### 3.1.3. Representation Specifications

#### Purpose

A reptype-spec permits programmers to specify non-standard representations for variables, constants, and types. Typically such specifications affect only the size and packing of stored values.

#### Syntax

```
[3-12] reptype-spec      ::= reptype-reference  
                      | predefined-reptype  
  
[3-13] predefined-reptype ::= fixed-reptype  
                           | float-reptype  
                           | enumerated-reptype  
                           | boolean-reptype  
                           | bitstring-reptype  
                           | charstring-reptype  
                           | array-reptype  
                           | record-reptype  
                           | pointer-reptype  
                           | semaphore-reptype
```

#### Semantics

When a reptype-spec is given in the REP part of a type-spec in a declaration, each of the names being declared is given the representation specified by the reptype-spec, i.e., the standard representation properties of the type are replaced with the properties specified in the reptype-spec.

The form of each kind of predefined-reptype is given in subsequent 3.x.3 Sections. Reptype-references are discussed in section 3.12.

### 3.1.4. Predefined Operators

#### Purpose

In this Section, we introduce the infix and prefix operators that are predefined in the language as well as operators for type conversion and other purposes.

#### 3.1.4.1. Operators and Expressions

Infix and prefix operators can be extended to apply to user-defined types as well as language-defined types. For this reason, the syntax for expressions (i.e., the relative precedence of infix and prefix operators, the effect of parentheses, and how operators are associated with operands) is defined independently of the types of the operands. Of course, whether an expression is well-defined or not does depend on the types of the operands, since operators are generally defined only to accept certain operand types. These constraints are specified individually in later 3.x.4 sections.

The relative precedence of infix and prefix operators is specified in the formal syntax. The relative precedence is also, however, given below, where operators on the same line are equal in precedence, and operators on earlier lines are higher in precedence than operators on later lines. (The "s" below stands for a manifest integer value. Operators containing such values are delimited with exclamation points and are used in fixed point expressions to specify the scale of the result. Operators without "s" and delimited by exclamation points are machine-dependent primitives used in defining multiple precision fixed and floating point operators by extension.)

```

substring subscription @ field-reference routine-invocation
unary + - NOT
** !**s!
* / // MOD !*s! !//s! !*! !/! !//! !MOD s!
+ - !+s! !-s! !+! !-! !!
= <> <> <= >= IN
& AND
! OR XOR ->

```

There are no !MOD!, !\*\*!, or !/s! operators. [See J3.1-5.]

### Syntax

- [3-14] manifest-expression ::= expression
- [3-15] expression ::= simple-expression [rounding-mode]  
[WITH PREC manifest-integer-expression]
- [3-16] rounding-mode ::= enumerated-literal
- [3-17] simple-expression ::= conjunction  
| simple-expression simple-exp-ops conjunction
- [3-18] simple-exp-ops ::= OR!!!XOR!->
- [3-19] conjunction ::= relation  
| conjunction conj-ops relation
- [3-20] conj-ops ::= AND|&
- [3-21] relation ::= sum  
| relation rel-ops sum  
| sum IN range-spec
- [3-22] rel-ops ::= =|<!>|<=|>|=|<>|IN
- [3-23] sum ::= term  
| sum sum-ops term
- [3-24] sum-ops ::= +|-||!  
| !+!  
| !-!  
| {!+||-} manifest-integer-expression!
- [3-25] term ::= factor  
| term term-ops factor
- [3-26] term-ops ::= \*|/|//|MOD  
| {!\*||//} manifest-integer-expression!  
| !MOD manifest-integer-expression!  
| !/  
| !\*!  
| !//!

[3-27]	<b>factor</b>	::= secondary   factor factor-ops secondary
[3-28]	<b>factor-ops</b>	::= **   !** manifest-integer-expression!
[3-29]	<b>secondary</b>	::= primary   unary-ops primary
[3-30]	<b>unary-ops</b>	::= + -!NOT
[3-31]	<b>primary</b>	::= variable   constant literal constructor substring-reference predefined-function function-invocation (simple-expression) attribute-query type-specifier   explicit-rep-converter
[3-32]	<b>variable</b>	::= simple-variable   subscripted-variable   field-reference   primary @
[3-33]	<b>constant</b>	::= simple-constant   subscripted-constant   field-constant
[3-34]	<b>substring-reference</b>	::= variable closed-range   constant closed-range
[3-35]	<b>range-spec</b>	::= [!{} lower-bound : upper-bound {}!]
[3-36]	<b>closed-range</b>	::= [lower-bound : upper-bound]
[3-37]	<b>lower-bound</b>	::= expression
[3-38]	<b>upper-bound</b>	::= expression
[3-39]	<b>manifest-range-spec</b>	::= range-spec
[3-40]	<b>type-specifier</b>	::= [type-spec]\$\$(expression)
[3-41]	<b>explicit-rep-converter</b>	::= reptype-spec (expression)

Semantics

The formal syntax indicates that if successive operators in an expression have the same precedence, the operands are associated with operators in left

to right order, e.g.,  $A-B-C$  is equivalent to  $(A-B)-C$ .

Although the association of operands with operators is fixed by the language, the evaluation order of operands is not fixed except for the & and ! operators, which represent conditional conjunction and disjunction, respectively. For example, in evaluating  $A*B + C*D$ ,  $C*D$  may be evaluated before  $A*B$ , and  $D$  may be evaluated before  $C$ . [See J3.1-6.]

It should also be noted that the side-effect rule (Section 4.3.4) ensures that for all expressions, even those containing explicit function calls, the value of the expression and the set of exceptions raised does not depend on the order of operand evaluation.

The operators in expressions (with the exception of & and !; see Section 3.5.4.1) are considered to invoke functions defined for the type of the operands. Predefined operator functions are described for each predefined type of the language. The type of an expression is determined by the type of result returned by functions. If an expression is of the form variable, constant, literal, constructor, attribute-query, type-specifier, or any of these forms enclosed in parentheses, its type is the type of the form. All other forms of expression invoke a function whose result type is the type of the expression.

The rounding-mode and WITH PREC qualifiers of an expression apply to the expression but not to syntactic constituents of the expression that are (nested) expressions, in particular routine-arguments and array-subscripts. These nested constituents may have their own qualifiers specified. Rounding-mode is discussed in Sections 3.2.4.1 and 3.3.4.1. WITH PREC is discussed in Section 3.3.4.4.

The range-spec specifies a set of ordered values whose forms have the usual mathematical interpretation for the specification of open and closed

intervals of values, i.e., [See J3.1-7.]

- . the form [E:F] specifies a closed interval, i.e., all X such that E ≤ X and X ≤ F.
- . the form (E:F] specifies a semi-open interval, i.e., all X such that E < X and X ≤ F.
- . the form [E:F) also specifies a semi-open interval, namely, all X such that E ≤ X and X < F.
- . the form (E:F) specifies an open interval, namely, all X such that E < X and X < F.

The type-specifier states that the type of the expression is that given by the type-ref. This form is used to resolve ambiguities among literals in certain cases (see Section 3.4) and to specify that a bitstring is to be considered to represent a value of a specific type. No executable action is associated with the evaluation of a type-specifier except that the expression is evaluated.

The explicit-rep-converter states that the representation of the expression is to be converted to the representation of the retype-spec.

#### Constraints

The expression in a type-specifier must either be of the specified type, or one of the possible types it can have must be the specified type, or the expression must be a bitstring whose length is the length required for the specified type. (For examples, see Sections 3.4.5 and 3.9.2.)

When the expression is a bitstring value, the type-specifier must be in a machine-dependent section of code (see Section 4.5.5.2) since knowledge of how values of a type are represented is being used in this case. For example, to treat a bitstring as a floating point value, the positions of the sign bits of mantissa and exponent, the position of the exponent, and the position of the mantissa bits must all be known, and these are machine-dependent.

In an explicit-rep-converter, the value of the expression must satisfy the constraints for the specified representation.

The rounding-mode constituent of an expression must be of the enumerated type ROUNDINGS, which is predefined to have the values TRUNCATED, ROUNDED, NATIVE\_R, and NATIVE\_L. The meaning of rounding-mode is given in Sections 3.2.4.1 and 3.3.4.1. The ROUNDINGS type may be given additional values (see Appendix D) whose significance is programmer-defined. A manifest-expression is an expression whose value is computable at translation time. Such expressions are defined in more detail for each of the predefined types, but in general, if an expression consists of only manifest-constant primaries and function-invocations evaluable at translation-time, the expression is considered a manifest expression. In addition if an argument of an inline function (see Section 4.3.1) is a manifest-expression, the corresponding formal parameter is considered a manifest-expression within the function body for that invocation. [See J3.1-8.]

The lower-bound and upper-bound of a manifest-range-spec must be manifest-expressions.

The lower-bound and upper-bound of a range-spec must be of the same type class. [See J3.1-9.]

#### Exceptions

The INVALID\_REP exception is raised if the value of the expression in an explicit-rep-converter or type-specifier does not satisfy the constraints for the specified representation.

The INVALID\_TYPE exception is raised if the value of the expression in a type-specifier does not satisfy the constraints for the specified type.

### 3.1.4.2. Operator Overview

Not all the infix operators are predefined for use with all types. This table indicates where definitions of the operators are discussed. In general, any operator's definition can be extended to accept other types as well.

substring	bitstring	3.6
	charstring	3.7
subscription	array	3.8
field-reference	record	3.9
@	pointer	3.10
unary + -	fixed	3.2
	float	3.3
NOT	boolean	3.5
	bitstring	3.6
== !==s!	fixed	3.2*
==	float	3.3*
* != !s!	fixed	3.2
* !=	float	3.3
/ !/	float	3.3
// !// !//s!	fixed	3.2
MOD !MOD s!	fixed	3.2
+ !+! !+s!	fixed	3.2
+ !+!	float	3.3
- !-! !-s!	fixed	3.2
- !-!	float	3.3
!!	bitstring	3.6
	charstring	3.7
= <>	all types	
> < <= >=	fixed	3.2
	float	3.3
	enumerated	3.4
IN	fixed	3.2
	float	3.3
	enumerated	3.4
&	boolean	3.5
AND	bitstring	3.6
!	boolean	3.5
OR	bitstring	3.6
XOR	bitstring	3.6
->	boolean	3.5*
WITH PREC	float	3.3
rounding-mode	fixed	3.2
rounding-mode	float	3.3

\* These operators have no language-specified definition, but their intended use is discussed in these sections.

Associated with each operator is a function whose definition can be extended by users to accept other than the predefined argument types. The

information needed to extend the definitions of these functions is given in Appendix D.

substring	SUBSTRING_
unary +	MON_PLUS_
unary -	NEGATE_
NOT	COMPLEMENT_
** !**s!	EXPONENTIATE_
* !*s!	MULTIPLY_
!*!	MD_MULTIPLY_
/ // !//s!	DIVIDE_
! / ! ! / !	MD_DIVIDE_
MOD !MOD s!	MODULUS_
+ !+s!	ADD_
- !-s!	SUBTRACT_
!+!	MD_ADD_
!-!	MD_SUBTRACT_
!!	CONCATENATE_
=	EQUAL_
<>	UNEQUAL_
>	GREATER_THAN_
<	LESS_THAN_
>=	GREATER_EQUAL_
<=	LESS_EQUAL_
AND	AND_
OR	OR_
XOR	XOR_
->	IMPLIES_

### 3.1.5. Assignment

#### Purpose

The semantics of the := operator are discussed in Section 3.x.4 for each value type.

In general, the assignment operators are viewed as invoking a procedure that actually performs the assignment. This procedure is language-defined for predefined types (and representations) but is user-defined for user-defined types. [See J3.1-10.]

The rules defining how the operands of the assignment operator must match are defined for each predefined type.

Syntax

[3-42] assignment-stmt ::= general-assignment  
| fixed-assignment  
| float-assignment  
| enumerated-assignment  
| boolean-assignment  
| bitstring-assignment  
| charstring-assignment  
| array-assignment  
| record-assignment  
| pointer-assignment  
| semaphore-assignment

[3-43] general-assignment ::= variable [closed-range] := expression;

Semantics

The semantics of the various types of assignment are described in subsequent 3.x.5 Sections. They all have the form of the general-assignment statement. The general-assignment form is provided for variables of user-defined types (see Appendix D).

Constraints

The type classes of the variable and expression in assignment statements must be identical.

### 3.1.6. Predefined Functions

For each predefined type, certain predefined functions are specified for operating on values of the type or for converting to values of the type. These functions are defined with syntax productions so the place where their use is explained can be found by consulting the syntax cross-reference index (Appendix F). [See J3.1-11.]

#### Syntax

```
[3-44] predefined-function ::= fixed-abs-function
                           | to-fixed-function
                           | fixed-min-function
                           | fixed-max-function
                           | float-abs-function
                           | to-float-function
                           | float-imp-prec-function
                           | float-min-function
                           | float-max-function
                           | float-epsilon-function
                           | enumerated-pred-function
                           | enumerated-succ-function
                           | handler-exists-function
                           | manifest-function
                           | bitstring-bin-function
                           | bitstring-dup-function
                           | bitstring-hex-function
                           | bitstring-oct-function
                           | bitstring-to-bit-function
                           | bitstring-zeros-function
                           | charstring-blanks-function
                           | charstring-dup-function
                           | clock-function
```

#### Semantics

The semantics of these functions are described in subsequent Sections

#### 3.x.6.

### 3.2. FIXED POINT

Fixed point values are scaled integer values used in numeric computations. The set of values associated with a particular fixed point type is the set determined by the following function:

$$F = I * R-S$$

where R is an integer (the radix), S is an integer (the scale), R-S is the step size (the minimum value separating successive fixed point values), and I is a set of integral values. For example, the value 2 with step size 1/2, (i.e., radix 2 and scale 1) is represented as 4\*(1/2). The value 2.5 is represented as 5\*(1/2). The value 1.1 in step size 1/10 is represented as 11\*(1/10). [See J3.2-1.]

Although the standard language does not define operators for multiple precision fixed point values, the declaration syntax and other operators are able to support such types by extension. It is anticipated that for applications requiring multiple precision fixed point representations, such extensions will be implemented directly in some translators even though the extensions are definable in the language. [See J3.2-2.]

#### 3.2.1. Attributes and Type Specification

##### Syntax

[3-45]    fixed-spec	::= FIXED (fixed-scale, radix, fixed-prec) [range-spec] [REP fixed-reptype]   INTEGER [range-spec] [REP fixed-reptype]
[3-46]    fixed-scale	::= manifest-integer-expression
[3-47]    radix	::= manifest-integer-expression
[3-48]    fixed-prec	::= manifest-enumerated-expression
[3-49]    fixed-attr	::= SCALE   RADIX   PRECISION   MIN   MAX

Semantics

A fixed-spec of the form FIXED (S, R, P) [L:U] defines, respectively, the values of the SCALE, RADIX, PRECISION, MIN, and MAX attributes of the type. The INTEGER form is used when integer values are desired. INTEGER is predefined to be equivalent to FIXED (0, 2, STD), i.e., a fixed point type with scale zero, radix 2, and standard precision.

If the range-spec is omitted from a fixed-spec, the default range for the type is implied. The default range for a FIXED (S, R, P) type is [FIXED\_MIN (S,R,P) : FIXED\_MAX (S,R,P)], where FIXED\_MIN and FIXED\_MAX are predefined functions (see Section 3.2.6.3). [See J3.2-3.]

The significance and type of the FIXED attributes is as follows:

- . SCALE -- the scale attribute can be thought of as specifying the number of fraction digits (in the specified radix). More specifically, if the radix of a fixed point type is R and the scale is S, the only fixed point values in the value set are integral multiples of R<sup>-S</sup>. R<sup>-S</sup> is called the step size of a fixed point type. The type of the SCALE attribute is INTEGER [-1000:1000]. [See J3.2-4.]
- . RADIX -- the radix attribute is used in determining the step size of a fixed point abstract type. FIXED is predefined only for RADIX 2. Other radix values can be used only if the fixed point type class has been extended to permit them. The type of the RADIX attribute is INTEGER [2:1000]. [See J3.2-5.]
- . PRECISION -- the precision attribute defines when computed values cannot be represented (i.e., PRECISION determines when OVERFLOW occurs). The default value of the precision attribute is STD, a predefined enumerated type value. Other precision values can be specified only if the fixed point abstract type has been extended to permit them. [See J3.2-6.]

The type of the PRECISION attribute is the enumerated type FIXED\_PRECISIONS. FIXED\_PRECISIONS is predefined to have a single element, STD. If the language is extended to support additional fixed point precisions, FIXED\_PRECISIONS may be redefined to include the elements SHORT, LONG, LONG2, LONG3, etc. (For details on how to extend the language to support additional precisions, see Appendix D, "Operational Semantics".) [See J3.2-7.]

- . MIN -- the minimum valid value in the type's value set. The type of the MIN attribute is FIXED (S,R,P), i.e., MIN has the same scale, radix, and precision attributes as the type being specified by fixed-spec, and the range of MIN is the default range for FIXED (S, R, P),

2/15/78

## Fixed Point

i.e., [FIXED\_MIN (S,R,P) : FIXED\_MAX (S,R,P)]. FIXED\_MIN and FIXED\_MAX are predefined functions (see Section 3.2.6).

- MAX -- the maximum valid value in the type's value set. The type of MAX is the same as the type of MIN.

If the type of the lower-bound or upper-bound of the range-spec is not the same as the type of the fixed-spec, the values are implicitly converted, if possible, to the fixed-spec type. (Note: Such implicit conversion is permitted only in certain cases; see Constraints).

### Examples

1. VAR A: FIXED (0, 2, STD) [0:10];

(A is an integer variable permitted to have values 0 through 10, inclusive.  
This is equivalent to VAR A: INTEGER [0:10].)

### Constraints

The lower bound and upper bound in a range-spec must both be fixed-expressions that are assignment compatible with the fixed-spec type (i.e., it must be possible to assign these values to a variable of the fixed-spec type without invoking the TO\_FIXED conversion function; see Section 3.2.6.1).

### 3.2.2. Literals

#### Syntax [See J3.2-8.]

[3-50]	fixed-literal	::= decimal-literal scale-spec   [+ -] number [exponent-spec]
[3-51]	decimal-literal	::= [+ -] number . [number] [exponent-spec]   [number] . number [exponent-spec]
[3-52]	exponent-spec	::= E [+ -] number
[3-53]	scale-spec	::= \S [+ -] number
[3-54]	number	::= digit...

#### Semantics

The value of number is the decimal value.

The exponent-spec is considered to represent a power of ten. The decimal value preceding the exponent-spec is multiplied by the specified power of ten. If no exponent-spec is present, the effect is the same as if E0 had been written.

The absence of a + or - character in a scale-spec or exponent-spec is equivalent to the presence of a +.

A fixed literal with an explicit scale-spec, e.g., ddd\Ssss, is considered to be explicitly converted to a rounded radix 2 fixed point value with scale sss and STD precision (see Section 3.2.6.1).

The RADIX of a fixed-literal is 2. The language can be extended to accept fixed point literals of other radices (e.g., 10) by providing additional suitable definitions of the TO\_FIXED function and using the function explicitly.

[See J3.2-9.]

The precision of a fixed-literal is STD. Although only one fixed point precision is predefined, the language can be extended to accept multiple precision literals by modifying the TO\_FIXED function definition and then

using the function explicitly. The range of a fixed-literal L is [L:L]. [See J3.2-10.]

#### Examples

1. 900

(This literal has the same value as 9E2, 900.\\\$0, 900\\\$0, 9E2\\\$0, .9E3\\\$0 etc. It has a scale of zero. 900. is not permitted nor is .9E3, because there is no scale-spec.) [See J3.2-11.]

2. .25\\\$2

(The step size for scale 2 is 1/4, i.e., 2-2. Consequently, the value .25 can be represented exactly as 1\*(1/4). This shows that fixed point values have an implied mantissa and exponent, just like floating point values. The difference is that for fixed point values, the exponent is a manifest constant, and so does not need to be physically represented in the stored value. The "mantissa" value of .25\\\$2 is 1 and the implicit exponent value is -2. Similarly, the mantissa value of .5\\\$2 is 2, i.e., 2\*(1/4).)

3. 10E-5

(This is equivalent to zero, since omission of a scale-spec implies a scale of zero.)

#### Constraints

The value of number in an exponent-spec or scale-spec must not exceed 1000.

The number of digits in a decimal-literal must not exceed 1000. The value of a literal, L, must be in the range [FIXED\_MIN (S,2,STD) : FIXED\_MAX (S,2,STD)], where S is the specified or implied scale of the literal. [See J3.2-12.]

### 3.2.3. Representation Specification

#### Syntax

```
[3-55] fixed-reptype ::= FIXED REP (size, signing)
                  | FIXED STD

[3-56] size       ::= manifest-integer-expression

[3-57] signing    ::= manifest-enumerated-expression

[3-58] fixed-rep-attr ::= SIZE
                  | SIGNING
```

#### Semantics

A fixed-reptype of the form FIXED REP (S, U) defines, respectively, the values of the SIZE and SIGNING representation attributes. The significance and type of these attributes is as follows:

- . SIZE -- the value of this attribute specifies the number of bits in the stored representation of a fixed point value, including sign bits, if any. The type is INTEGER [1:MAXSIZE], where MAXSIZE is the value of the largest supported fixed point precision.
- . SIGNING -- this attribute specifies whether or not a fixed point value is stored with its sign bit explicitly represented. A stored value with the UNSIGNED attribute is considered to represent a non-negative number and no sign bit is part of its representation. The type is SIGNING, an unordered enumerated type with the values SIGNED and UNSIGNED.

Specification of SIZE affects only the space used to store a value; it does not affect a value's use in expressions.

The default representation of fixed point values is FIXED STD. For a variable, V, FIXED STD is equivalent to FIXED REP(S, SIGNED), where S is the number of bits associated with V.PRECISION.

#### Examples

1. VAR A: INTEGER [0:10] REP FIXED REP (4, UNSIGNED);

(The variable A has a non-standard representation, namely, 4 bits with no sign bit.)

## 2. (1).SIZE

(This gives the SIZE attribute of the literal one. Since such literals have STD precision, the SIZE attribute specifies the default number of bits used to represent STD precision values. For most implementations, (1).SIZE will be equal to the word length of the object computer. Note that FIXED.SIZE is not well-defined, since SIZE will differ for different precisions and representations. In addition, queries of the form type-name.attribute are not permitted (see 3.1.1.1).)

### Constraints

The value of signing must be a value of the type SIGNING.

If the specified SIZE is N, then the specified range of values of the corresponding variable must not exceed the range of values that can be represented in N bits, namely  $[-(2^N-1):2^N-1]$  for a two's-complement object computer and  $[-(2^{N-1}-1):2^{N-1}-1]$  for a ones-complement or sign-magnitude computer.

If UNSIGNED is specified, the corresponding variable must have an explicitly specified range whose lower-bound is non-negative. [See J3.2-13.]

If the specified SIZE is N and the default size value is P, then N must be in the range  $[0:P]$ , i.e., it is not possible to specify a representation size that is bigger than the default representation size as determined by the precision. [See J3.2-14.]

### Exceptions

The INVALID\_REP exception is raised if a range-spec contains non-manifest values (i.e., values not known at translation time) and the specified SIZE attribute value is too small to represent the specified range, or the lower-bound is negative and UNSIGNED has been specified.

### 3.2.4. Operations and Expressions

#### Syntax

[3-59] fixed-expression ::= expression  
[3-60] integer-expression ::= fixed-expression  
[3-61] manifest-integer-expression ::= manifest-expression

#### Semantics

Fixed point operators fall in five classes:

- . the standard arithmetic infix operators: addition, subtraction, multiplication and division; in addition an operator symbol is available for exponentiation although no language definition is provided for this symbol. [See J3.2-15.]
- . the standard arithmetic prefix operators, + and -.
- . scaling operators, in which the scale of the result is explicitly specified by the programmer, overriding the default scaling rules given in Section 3.2.4.2.
- . machine-dependent arithmetic operators for use in implementing multiple precision arithmetic and for accessing the machine-dependent value produced when overflow occurs.
- . relational operators producing a boolean result.

Each of these operator types is discussed in separate sections later. But first, constraints common to all the operator types are specified.

#### Constraints

Both operands must have the same radix.

A fixed-expression must be an expression yielding a result of type class FIXED.

An integer-expression or manifest-integer-expression must be a fixed-expression with a scale of zero and radix 2. [See J3.2-16.]

### 3.2.4.1. Rounding Control

#### Purpose

The ROUNDED and TRUNCATED modifiers (in expressions; see 3.1.4.1) specify that all arithmetic operations in an expression are performed in ROUNDED or TRUNCATED mode. The default mode for fixed point is TRUNCATED. No operators are predefined for NATIVE\_T or NATIVE\_R rounding modes (these are intended for use with floating point expressions).

#### Semantics

Fixed point expressions are evaluated in either ROUNDED or TRUNCATED mode. (Other modes may be supplied by adding them to the ROUNDINGS enumerated type and providing definitions for such modes, e.g., FLOOR, truncation toward minus infinity, might be added by extension.) Which mode is in effect depends on whether the ROUNDED or TRUNCATED modifier is applied to an expression. These modifiers may be applied to initializing expressions of declarations (Section 3.1.1.3), to arguments of routines (Section 4.3.2), subscript lists (Section 3.8.5), and in assignment statements (Section 3.2.5). If no modifier is explicitly present, the evaluation mode is TRUNCATED.

When infix operators are invoked in TRUNCATED mode, the mathematically defined result is represented exactly, if possible, and otherwise is truncated toward zero to the closest exactly representable value. [See J3.2-17.]

In ROUNDED mode, the mathematically defined result of an infix operation is converted to the closest exactly representable value. If the exactly representable values are equally far away from the true value, the value whose last digit is even (in the radix of the representation) is chosen. [See J3.2-18.]

Examples

1.  $A//D + TO_FIXED(E, A.SCALE-D.SCALE, ROUNDED) ROUNDED$

(The result of the division is rounded and the value of E is rounded to the specified scale. The sum is also computed in rounded mode, but since no rescaling of the sum is specified, the rounded result will be the same as the truncated result, since sums are computed exactly.)

3.2.4.2. Infix OperatorsSemantics

The following infix operators have a predefined meaning for single precision fixed point operands (i.e., operands whose PRECISION attribute is STD; the operator definitions may be extended to apply to other than STD precisions):

OP	NAME	RESULT	EXCEPTIONS
//	divide	fixed	ZERO_DIVIDE OVERFLOW
MOD	modulus	fixed	ZERO_DIVIDE OVERFLOW
*	multiply	fixed	OVERFLOW
+	add	fixed	
-	subtract	fixed	

Let the precision of the first and second operands be P1 and P2 respectively, the radix be R, and the scale of the first and second operands be S1 and S2. Then the following rules apply: [See J3.2-19.]

- . The radix of the result is R.
- . The scale of the result, SR, is defined according to the following rules:

operation	scale
//	$SR = S1-S2$
MOD	$SR = S1$
*	$SR = S1+S2$
+	$SR = \text{Max}(S1, S2)$
-	$SR = \text{Max}(S1, S2)$

- . The precision of the result is the maximum of the precisions of the operands, i.e., PR = Max(P1,P2). This rule applies to the \*\* operator as well when its operands are fixed point values.

2/15/78

The value of  $E * F$ ,  $E + F$ , and  $E - F$  is the mathematically defined value of the product, sum, and difference, respectively, if OVERFLOW is not raised. In evaluating the sum and difference, if the scales of the operands are different the operand with the smaller scale is first converted to the result scale by invoking the TO\_FIXED function (see Section 3.2.6.1).

For an expression of the form  $E // F$ , the mathematically defined exact result is converted to the result scale and precision. [See J3.2-20.] More specifically let EI and FI be defined so the following relationships are satisfied, where SSE and SSF are the step size of E and F respectively:

$$\begin{aligned} E &= EI * SSE \\ F &= FI * SSF \end{aligned}$$

Then the value of  $E // F$  is the value of

$$\text{Sign}(E) * \text{Sign}(F) * \text{Floor}(\text{Abs}(EI) / \text{Abs}(FI))$$

where  $\text{Sign}(x) = -1$  if  $x$  is less than zero and otherwise equals +1,  $\text{Floor}(x)$  is the largest integer value not exceeding the value of  $x$ , and  $\text{Abs}$  is the absolute value function. This definition ensures that the following identity is satisfied if no exceptions are raised:

$$\text{ABS}(E // F) = \text{ABS}(-E // F) = \text{ABS}(E // -F) = \text{ABS}(-E // -F)$$

The above definitions for \*, +, and - apply whether an expression is evaluated in ROUNDED or TRUNCATED mode. The definition for // applies for evaluations in TRUNCATED mode. If // is evaluated in ROUNDED mode, then the rounded result, RR, is computed by rounding the exact result symmetrically away from zero or to the nearest even value when the remainder is exactly half the divisor. This is the same rounding definition that is used for floating point rounding.

The MOD operator is defined as:

$$\begin{aligned} E \text{ MOD } F \text{ TRUNCATED} &= E - (E // F) * F \text{ TRUNCATED}; \\ E \text{ MOD } F \text{ ROUNDED} &= E - (E // F) * F \text{ ROUNDED}; \end{aligned}$$

i.e., the scale of E MOD F is the scale of E and the precision of E MOD F is the maximum of the precisions of E and F.

The \*\* operator is intended for exponentiation, although such a use requires a programmer to define the function invoked when this operator is applied to fixed point operands. The normal fixed point operator constraints apply to this operator, however. That is, its operands must have the same radix, and the precision of the result is the maximum of the precision of the operands. The value and default scale of the result are defined by the function that is associated with the \*\* operator.

#### Examples

1. 7//2

(This yields the result 3; 7 MOD 2 = 1.)

2. 7//-2

(This yields the result -3; 7 MOD -2 = 1.)

3. -7//-2

(This yields the result 3; -7 MOD -2 = -1.)

4. -7//2

(This yields the result -3; -7 MOD 2 = -1.)

5. 7//2 ROUNDED

(This yields the result 4, since the true remainder is exactly half the divisor. Consequently, 7 MOD 2 ROUNDED is -1. 9//2 ROUNDED would equal 4 also, since the unbiased rounding rule is round to even when the remainder is exactly half the divisor.)

6. 2\\$0 // .75\\$2

(The scale of the result is -2 (i.e., the step size is four). The true result is two and two-thirds. When this result is truncated to a step size of four, the result is 0. Any specific division scaling rule gives what might appear to be unexpected results in some cases. However, fixed point division of integers or with an explicitly specified result scale are the most common cases to be expected, and these cases cause no surprises.)

7.  $2\backslash S2 // .75\backslash S2$

(This value is truncated to the nearest multiple of the result step size, i.e., 1, and hence the result is 2.)

#### Exceptions

The OVERFLOW exception is raised whenever the mathematically defined value of an operator would lie outside the range of representable values for a given precision, radix, and scale.

The ZERO\_DIVIDE exception is raised if the divisor has the value zero.

#### 3.2.4.3. Prefix Operators

The following prefix operators have a predefined meaning for single precision fixed point operands (i.e., operands whose PRECISION attribute is STD; the operator definitions may be extended to apply to other than STD precisions):

OP	NAME	RESULT	EXCEPTIONS
-	negate	fixed	OVERFLOW
+	mon plus	fixed	

The effect of -E is equivalent to TO\_FIXED(0, E.SCALE, E.RADIX, E.PRECISION) - E.

The value of +E is E.

#### 3.2.4.4. Scaling Infix Operators

##### Purpose

Scaling operators permit a programmer to specify the scale of a fixed point arithmetic result, i.e., to override the default scaling rules. [See J3.2-21.]

##### Semantics

The following scaling operators have predefined meaning for single precision operands, and the operator definitions can be extended to provide meanings for other precisions as well.

OP	NAME	RESULT	EXCEPTIONS
!//s!	scaled divide	fixed	ZERO_DIVIDE OVERFLOW
!MOD s!	scaled mod	fixed	ZERO_DIVIDE OVERFLOW
*s!	scaled multiply	fixed	OVERFLOW
+s!	scaled add	fixed	
-s!	scaled subtract	fixed	

For the above operators, "s" is a manifest-integer-expression. The value of "s" specifies the scale of the result. In computing results, the exact product, quotient, modulus, sum, or difference is computed and then converted to the specified scale and default precision. The conversion is performed, in effect, by applying the TO\_FIXED operator to the exact result in ROUNDED or TRUNCATED mode, depending on the rounding mode of the expression. In all other respects, the rules for the standard operators apply, i.e., the radix of the result is the radix of the operands and the precision of the result is the maximum of the precision of the operands. Unless the operator definitions are extended, the only result precision supported is STD and the only radix supported is 2.

#### Examples

Assume the following declarations are in effect:

```
VAR E,F: FIXED (2, 2, STD);
```

1. E !#2! F

(The scale of E#F would be four, but the scale of this result is two. If the value of E and F were .25 (i.e., 1#(1/4)), the value of E # F would be .0625 (i.e., 1#(1/16)). The value of E !#2! F, however, is zero, since the smallest representable positive value with a scale of 2 is .25, and the exact result is less than half of .25, so the result is zero in either ROUNDED or TRUNCATED mode.)

2. 2 !//2! .75\\$2

(The scale of the result is 2, i.e., the step size is 1/4. The true result is 2 + 2/3; this is truncated to a multiple of 1/4, yielding the result 2.50\\$2.)

### 3.2.4.5. Machine-Dependent Infix Operators

#### Purpose

The operators defined here are intended for use in implementing multiple precision arithmetic not directly supported by an implementation. In addition, the operators provide access to machine dependent results produced when the OVERFLOW exception is raised. [See J3.2-22.]

#### Semantics

OP	NAME	RESULT	EXCEPTIONS
!//!	!md divide	!bitstring	
!MOD!	!md modulus	!bitstring	
!*!	!md multiply	!bitstring	
!+!	!md add	!bitstring	
!-!	!md subtract	!bitstring	

Each of these operators produces a bitstring result whose length and format is object computer dependent. None of the operators raise exceptions. The bitstring result represents the maximum amount of information derivable from the arithmetic operation, e.g., !\*! applied to two STD precision operands will produce the machine-dependent form of double precision product. This form can then be manipulated to whatever format LONG fixed point values are to have. Similarly, the !+! operator can be used to access the results of a fixed point addition in which OVERFLOW occurred.

#### Constraints

These operators may only be used in machine-dependent portions of a program (see Sections 4.5.5.2 and 4.8).

### 3.2.4.6. Relational Infix Operators

#### Purpose

Relational operators are used to test fixed point values for equality, for an ordering relationship, or to see if the value is in a specified range.

#### Semantics

The following relational operators have a predefined meaning for single precision operands. The operator definitions can be extended to provide meanings for other precisions and radices as well.

OP	NAME	RESULT	EXCEPTIONS
=	equality	boolean	
<>	inequality	boolean	
>	greater than	boolean	
<	less than	boolean	
>=	greater/equal	boolean	
<=	less/equal	boolean	
IN	in-range	boolean	

For all operators except IN, the values of the operands are compared, giving the value TRUE as the result if the relation is satisfied. Otherwise the value FALSE is produced.

For E IN [L:U], the value is the value of

$$L \leq E' \& E' \leq U$$

where E' is the value of E. Note that E is evaluated only once.

For E IN [L:U), the value is the value of

$$L \leq E' \& E' < U$$

where E' is the value of E.

For E IN (L:U], the value is the value of

$$L < E' \& E' \leq U$$

where E' is the value of E.

For E IN (L:U), the value is the value of

$$L < E' \& E' < U$$

where E' is the value of E.

### Efficiency Considerations

If the scales of the operands are not equal, the operand with the smaller scale is converted to the larger scale before the comparison is made. If the scale conversion raises the OVERFLOW, then clearly the operand values are not equal and the operand being rescaled has the larger absolute value. Hence, there is no need to raise the OVERFLOW exception. From an optimization point of view, a programmer should be guaranteed that if the scales of the operands are the same, no overflow check will be performed for an equality or ordering relation.

Although the language provides predefined semantics only for comparisons of STD precision operands, the semantics require that if an implementation is extended to support multiple precision operands, and if the operands are not of the same precision, the operand with the smaller precision is converted to the precision of the other operand and then the operand values are converted to the same scale.

#### 3.2.5. Assignment

##### Syntax

[3-62] fixed-assignment        ::= variable := fixed-expression;

##### Semantics

The value of the fixed-expression becomes the value of the variable unless the value of the source is outside the specified range of the variable (see RANGE exception below).

If the scale and/or precision of the expression is not the same as the scale and/or precision of the variable, then the expression's value is converted to the variable's scale and precision, using the rounding mode that is in effect for the expression. The conversion is equivalent to that performed by the TO\_FIXED function, except that if applying the TO\_FIXED

function explicitly would cause OVERFLOW to be raised, the RANGE exception is raised instead.

(Note: since precisions other than STD are not defined unless the language is extended, the above rule constrains the semantics of the extended language.)

If the RANGE exception is raised, the value of the variable is not changed.

#### Constraints

The variable and the expression must have the same radix.

The expression's value must not be outside the variable's range (see RANGE exception).

The scale of the expression's value must not exceed the scale of the variable (i.e., no implicit approximation of the assigned value is permitted.)

#### Exceptions

The RANGE exception is raised if the value of the expression exceeds the specified range for the variable.

### 3.2.6. Predefined Functions

#### 3.2.6.1. TO\_FIXED

##### Purpose

The TO\_FIXED function is used to convert fixed point values to different scales, radices, and precisions and other types to fixed point.

##### Syntax

```
[3-63] to-fixed-function ::= TO_FIXED (to-fixed-value, fixed-scale
                                [, radix] [,fixed-prec]
                                [rounding-mode])

[3-64] to-fixed-value    ::= fixed-expression
                           | float-expression
                           | enumerated-expression
                           | boolean-expression
                           | bitstring-expression
                           | charstring-expression
```

##### Semantics

TO\_FIXED is a function that converts a value to a specified or implied scale, precision, and radix. The most general form, TO\_FIXED (E,S,R,P,T) yields a fixed point value with scale S, radix R, and precision P. If the conversion is not exact, T specifies what rounding mode is applied.

The radix, precision, and rounding mode arguments are optional. The default radix is 2, the default precision is STD, and the default rounding-mode is TRUNCATED. [See J3.2-23.]

When the to-fixed-value has the form of a fixed-literal, if the mathematically defined value of the literal can be exactly represented in the result scale, radix, and precision, it is exactly represented. Otherwise, the closest exactly representable value is produced, where "closest" is determined by the value of the rounding-mode argument. Note that the literal is not first converted to default radix and precision, so when TO\_FIXED is applied to literals, the result is as close as possible to the exact result. [See

## J3.2-24.]

When the to-fixed-value is not a fixed-literal but is a fixed-expression or float-expression, if the value of the expression can be exactly represented in the result scale, radix, and precision, it is exactly represented. Otherwise, the closest exactly representable value is used, where "closest" is determined by the value of the rounding-mode argument.

When applied to an enumerated-expression, E, the value returned is the ordinal value of E. The ordinal value, N, for an ordered enumerated type T is defined such that the Nth successor of T.MIN is the value E. TO\_FIXED is automatically defined only for ordered enumerated values with default representation. (The definition of TO\_FIXED can be extended by programmers to apply to non-standard representations if this is necessary, but the function must be defined explicitly for each non-standard representation; see Section 3.4).

When applied to a boolean-expression, TO\_FIXED has the value zero if the boolean-expression is FALSE; otherwise it has the value one.

When applied to bitstring-expressions, the bitstring value is considered to represent an unsigned radix 2 integer value. This value is converted to the result scale, radix, and precision. [See J3.2-25.]

When applied to charstring-expressions, the value of the expression is required to have the form of a fixed-literal optionally preceded or followed by blanks. The resulting fixed point value is the same as the value that would have been the result if TO\_FIXED had been applied to a fixed-literal identical in form to the charstring-expression's value. TO\_FIXED is predefined only for ASCII charstring values. [See J3.2-26.]

Examples

1. TO\_FIXED(A, 0, LONG)

(If TO\_FIXED has been extended for LONG precision, this expression converts A to an integral value of LONG precision. If A is not an exact integral value, the value is truncated (toward zero)).

2. TO\_FIXED(1E6, 0, LONG)

(This produces a double precision integer with value 1.)

3. TO\_FIXED (1.1\S1, 1, 10)

(This produces the value 1.1 exactly represented as a radix 10 value with scale 1. This example requires that the TO\_FIXED function be extended to process radix 10 fixed point values. As the operational definition in Appendix D shows, when TO\_FIXED is applied to a literal value, the literal is passed to TO\_FIXED as a character string to ensure no spurious approximations are produced. From a user's viewpoint however, TO\_FIXED applied to literal values produces exactly the result specified for fixed point values. Note that TO\_FIXED(1.1, 1, 10) would be equivalent to TO\_FIXED (FLOAT(1.1, 2), 1, 10) since 1.1 is a float-literal.)

Constraints

The abstract value being converted must not exceed the range permitted for values of the result scale, radix, and precision.

The values of the fixed-scale, radix, and fixed-prec must lie in the range of values permitted for the fixed point SCALE, RADIX, and PRECISION attributes.

Exceptions

The OVERFLOW exception is raised if the value being converted lies outside the range permitted for values of the result radix, scale, and precision.

The CONVERSION exception is raised if the to-fixed-value is a charstring-expression and its form does not satisfy the syntax for a fixed-literal optionally preceded or followed by blanks.

### 3.2.6.2. ABS

#### Syntax

[3-65] fixed-abs-function ::= ABS (fixed-expression)

#### Semantics

The function ABS(E) yields the absolute value of E with the scale, radix, and precision of E. The OVERFLOW exception is raised if the mathematically defined absolute value of E exceeds the maximum value permitted for fixed point values having the precision and scale of E.

#### Examples

Assume the following declarations are in effect:

```
VAR E: FIXED (0,2,STD);
VAR F: FIXED (0,2,LONG);
```

1. F := ABS (TO\_FIXED(E, E.SCALE, E.RADIX, LONG)) ;

(Note that for some object computers, the complement of the most negative value is larger than the largest positive value. Hence, the ABS function can raise the OVERFLOW exception on such computers. This example shows a way of avoiding OVERFLOW, since the result of the TO\_FIXED conversion will be a LONG value, and the absolute value of a single precision value can always be represented as a double precision value. Note that TO\_FIXED(ABS(E), E.SCALE, E.RADIX, LONG) will not have the desired effect, since ABS always returns a result with the precision of its argument. Note also that if E.RADIX = 2, the argument can be omitted, TO\_FIXED (ABS(E), E.SCALE, LONG)).

### 3.2.6.3. FIXED\_MIN and FIXED\_MAX

#### Purpose

The FIXED\_MIN and FIXED\_MAX functions yield the minimum and maximum permitted values for a given scale, radix, and precision.

#### Syntax

- [3-66] fixed-min-function ::= FIXED\_MIN (fixed-scale, radix, fixed-prec)
- [3-67] fixed-max-function ::= FIXED\_MAX (fixed-scale, radix, fixed-prec)

#### Semantics

These functions are predefined for STD precision and any permitted scale and radix. They yield the minimum (FIXED\_MIN) and maximum (FIXED\_MAX) values permitted for the specified scale, radix, and precision.

#### Examples

1. FIXED\_MAX (0,2,STD)

(This gives the maximum standard precision integer value. For a computer whose STD precision size is 16 bits, FIXED\_MAX (0,2,STD) will normally equal 2<sup>15</sup>-1. For a computer with twos-complement representation of negative numbers, FIXED\_MIN (0,2,STD) will equal -2<sup>15</sup>.)

#### Constraints

The values of fixed-scale, radix, and fixed-prec must lie in the range of values permitted for the fixed point SCALE, RADIX, and PRECISION attributes.

### 3.3. FLOATING POINT

Floating point values are suitable for use in computations where implicit approximations to the mathematically defined result are acceptable and where a wide range of values is needed.

The set of values associated with a particular floating point value type is the set determined by the following function:

$$F = I \cdot R^E$$

where  $R$  is an integer (the radix),  $E$  is an integer (the exponent), and  $I$  is a set of integral values. In contrast to fixed point values and expressions where the value of  $E$  must be a manifest constant for a given fixed point type, the value of  $E$  need not be manifest for floating point types. Nonetheless, the set of values associated with a floating point type is defined in a manner similar to that for fixed point types.

Although the standard language does not define operators for multiple precision floating point values, the declaration syntax and other operators are able to support such types by extension. It is anticipated that for applications requiring multiple precision floating point representations, such extensions will be implemented directly by some translators, even though the extensions are definable in the language. [See J3.3-1.]

The radix of floating point values is fixed at a single value for a given implementation, unlike the radix of fixed point values.

### 3.3.1. Attributes and Type Specification

#### Syntax

```
[3-68] float-spec ::= FLOAT (float-precision)
                  | [range-spec]

[3-69] float-precision ::= manifest-integer-expression

[3-70] float-attr ::= NOM_PRECISION
                  | IMP_PRECISION
                  | RADIX
                  | MIN
                  | MAX
                  | EXPONENT_MIN
                  | EXPONENT_MAX
```

#### Semantics

A float-spec of the form FLOAT (P) [L:U] defines, respectively, the values of the NOM\_PRECISION, MIN, and MAX attributes of the type. The significance and type of these attributes is as follows:

- . NOM\_PRECISION -- the minimum number of decimal digits of precision specified by a programmer for a FLOAT type class. The type of this attribute is INTEGER [1:M] where the value of M is implementation defined.
- . MIN -- the minimum value in the type's value set. The type of MIN is FLOAT (IMP\_PRECISION) [LL:UU] where the value of LL and UU are the smallest (most negative) and largest representable floating point values, respectively, for the specified precision.
- . MAX -- the maximum value in the type's value set. The type of MAX is the same as the type of MIN.

In addition to these explicitly specified attributes, the following implied attributes have implementation-defined values.

- . IMP\_PRECISION -- the number of decimal digits of precision actually used when computing with values of the type. This value is implementation defined and is always greater than or equal to the NOM\_PRECISION attribute for a valid FLOAT type. The type of this attribute is INTEGER [NOM\_PRECISION:M], where the value of M is implementation defined.
- . RADIX -- the radix of the floating point mantissa and exponent representation. This value is fixed for a given implementation. Its type is INTEGER [2:1000].

- EXPONENT\_MIN -- the minimum exponent value for a floating point value of the implemented precision. The type of EXPONENT\_MIN is INTEGER [L:U] where L and U are the minimum and maximum values for a STD precision integer, (i.e., FIXED\_MIN (0,2,STD) and FIXED\_MAX (0,2,STD); see Section 3.2.6.3.) [See J3.3-2.]
- EXPONENT\_MAX -- the maximum exponent value for a floating point value of the implemented precision. The type of EXPONENT\_MAX is the same as the type of EXPONENT\_MIN.

The value of NOM\_PRECISION specifies the minimum number of decimal digits of accuracy required in the sense that if the value of NOM\_PRECISION is N, then any integer value in the range [0:10<sup>N-1</sup>] can be represented without error. In practice, the implemented precision associated with a given NOM\_PRECISION will be whatever precision the object computer supports efficiently and which is at least as large as the NOM\_PRECISION specified. It should be noted that when OVERFLOW and UNDERFLOW occur depends on the implemented precision and exponent range, and the value of these attributes depends directly on the specified nominal precision. (Hence these attributes are not representation attributes, because they affect what results occur when operating with floating point values.)

If the range-spec is omitted from a float-spec, the default range for the type is implied. The default range for a FLOAT(P) type is [FLOAT\_MIN (P) : FLOAT\_MAX (P)]. [See J3.3-3.]

If the NOM\_PRECISION of the lower-bound or upper-bound of a range-spec is not the same as the NOM\_PRECISION of the float-spec, the values are implicitly converted to the specified nominal precision.

#### Constraints

The lower-bound and upper-bound of range-spec must both be float-expressions.

The specified precision must be non-negative and not exceed IMP\_PRECISION.MAX, the maximum precision supported by an implementation.

### 3.3.2. Literals

#### Syntax

[3-71] float-literal ::= decimal-literal

#### Semantics

The value of the float-literal is an approximation to the decimal value, where the extent of the approximation depends on the set of (exactly representable) values in the value set for a given precision. An acceptable approximation will be given in detail in the formal language specification, but the essence of the rules are given here:

- . values of float-literals that can be represented exactly using the object computer's representation of floating point values must be represented exactly as long as the number of significant digits in the literal does not exceed a maximum value defined for each floating point precision that is supported. If the number of significant digits in the literal is too large, the literal value is rounded or truncated (depending on the mode of the expression containing the literal) to S significant digits, (where S is the maximum supported NOM\_PRECISION value), and then this value is converted to an exactly representable value.
- . given two float-literals, R1 and R2, whose value in object computer representation is OR1 and OR2, respectively, then if R1 > R2, OR1 must be greater than or equal to OR2. (This rule guarantees that if the true value of the literal lies between two representable values, one of the representable values is chosen as the value of the literal, and the choice does not ever invert the ordering of the true values.)

The precision of a float-literal is the number of digits in the float-literal, exclusive of the digits in the exponent-spec, if present. If there are more digits than the largest number of digits supported (i.e., IMP\_PRECISION.MAX), the literal's value is rounded to this number of digits, and its specified precision attribute is equal to IMP\_PRECISION.MAX.

The range of a literal L is [L:L].

### 3.3.3. Representation Specifications

#### Syntax

[3-72] float-reptype ::= FLOAT\_STD  
[3-73] float-rep-attr ::= SIZE  
| MANTISSA\_SIZE  
| EXPONENT\_SIZE

#### Semantics

The standard representation of a floating point value of NOM\_PRECISION P defines values for the following attributes:

- . MANTISSA\_SIZE -- the number of bits, including sign, used to represent a stored floating point value's mantissa. This attribute's type is INTEGER [1:FIXED\_MAX(0, 2, STD)], i.e., the upper limit is the maximum for a STD precision INTEGER value.
- . EXPONENT\_SIZE -- the number of bits, including sign, used to represent a stored floating point value's exponent. The type is the type of MANTISSA\_SIZE.
- . SIZE -- the number of bits used to store a floating point value. The type is the type of MANTISSA\_SIZE. [See J3.3-4.]

No representation control is provided for floating point types. FLOAT\_STD is the only predefined float-reptype. [See J3.3-5.]

### 3.3.4. Operators and Expressions

#### Syntax

[3-74] float-expression ::= expression

#### Semantics

Floating point operators fall in four classes:

- . the standard infix arithmetic operators: addition, subtraction, multiplication and division; in addition an operator symbol is available for exponentiation although no language definition exists for this symbol;
- . the standard arithmetic prefix operators, + and -.
- . machine-dependent arithmetic operators for use in implementing multiple precision floating point arithmetic and for accessing the machine-dependent value produced when OVERFLOW or UNDERFLOW occurs;
- . relational operators producing a boolean result.

Each of these operator types is discussed in separate sections. [See J3.3-6.]

#### 3.3.4.1. Rounding Control

##### Purpose

The ROUNDED, TRUNCATED, NATIVE\_R and NATIVE\_T modifiers specify whether all infix and prefix operators in an expression are invoked in ROUNDED, TRUNCATED, or machine-dependent mode. The default mode for float-expressions is ROUNDED. The default mode is overridden by explicitly specifying a different mode in an expression.

##### Semantics

Operators invoked in ROUNDED mode yield results that are rounded to the nearest representable floating point value in the result representation precision. If the true result is exactly between two exactly representable values, the value whose low order mantissa digit represents an even number (in the radix of the representation) is always chosen (hence values are rounded up or down in an unbiased manner). It is equally acceptable to always choose the

value whose low order mantissa digit is odd, but whether the implemented rule is "round to even" or "round to odd", only one rule is used by a given implementation in computing ROUNDED values.

Operators invoked in TRUNCATED mode yield results that are truncated toward zero, i.e., toward the nearest representable value that is closest to zero. (This rule is consistent with the TRUNCATION rule for fixed point arithmetic.) [See J3.3-7.]

Operators invoked in NATIVE\_R mode yield results using the object computer's instructions for rounded floating point arithmetic of the required result precision. These results are subject to whatever behavior the instructions imply. Use of the NATIVE\_T mode yields truncated results, using the object computer instruction set's definition of truncation. In particular, E - F NATIVE\_T may well yield the value E if F is small enough. [See J3.3-8.]

#### Constraints

Expressions may be evaluated in NATIVE\_T or NATIVE\_R modes only in machine-dependent portions of programs (see Sections 4.5.5.2 and 4.8).

#### 3.3.4.2. Infix Operators

The following infix operators have a predefined meaning for floating point operands.

OP	NAME	RESULT	EXCEPTIONS
/	divide	float	ZERO_DIVIDE OVERFLOW UNDERFLOW
*	multiply	float	OVERFLOW
+	add	float	UNDERFLOW
-	subtract	float	

Let the NOM\_PRECISION attribute of the first and second operands be P1 and P2, respectively. Then the NOM\_PRECISION attribute of the result is the maximum of P1 and P2. (This rule also applies to the \*\* operator.) [See

## J3.3-9.]

The value of E/F, E\*F, E+F, and E-F is computed by taking the abstract value of E and F (i.e., the value exactly specified by the representation of E and F), computing the mathematically defined quotient, product, sum, and difference, respectively, and then rounding or truncating to the "closest" exactly representable value for the precision of the result. Whether the result is rounded or truncated depends on whether the ROUNDED, TRUNCATED, NATIVE\_T or NATIVE\_R modifier is applied to the expression.

If ROUNDED is specified, then the "round to even" rule is used. If TRUNCATED is specified, then the result is truncated toward zero. If NATIVE\_T or NATIVE\_R is specified, the result is computed in a machine-dependent manner, using the actual object machine hardware instructions. (NATIVE\_T or NATIVE\_R may be specified only in machine-dependent portions of programs). If neither ROUNDED nor TRUNCATED is specified, ROUNDED is assumed. [See J3.3-10.]

Exceptions

The OVERFLOW exception is raised if the value produced is outside the range of representable values of the result's precision.

The UNDERFLOW exception is raised if the value produced is non-zero but its representation requires an exponent value that is less than EXPONENT\_MIN.

The ZERO\_DIVIDE exception is raised if division by zero is attempted.

3.3.4.3. Prefix Operators

The following prefix operators have a predefined meaning for floating point operands.

OP	NAME	RESULT	EXCEPTIONS
-	negate	float	OVERFLOW
+	mon plus	float	

The effect of -E is equivalent to 0. - E.

The effect of +E is equivalent to E.

#### 3.3.4.4. Precision Control

The phrase "WITH PREC p" (where p is a manifest-integer-expression; see Section 3.1.4.1) may be added at the end of any floating point expression, following an optional rounding-mode, as the syntax in Section 3.1.4.1 shows. This modifier specifies that the result of each individual float-literal or floating point infix or prefix operator in the expression is computed to at least the specified number, p, of decimal digits. In essence, with a suitable value of p, a programmer can specify that all operations be performed in single, double, etc. floating point precision, depending on the relation between the value of p and the representation precision levels. [See J3.3-11.]

##### Examples

1. A\*B + C\*D WITH PREC 20;

(If A, B, C, and D were all declared to have precision 10, then the products and sum in this expression will be computed in double the default precision. Note that it is possible to define a constant

```
CONST DOUBLE: INTEGER := 20;
```

A programmer could then write

```
A*B + C*D WITH PREC DOUBLE
```

2. A\*B/C WITH PREC 20

(A double precision product and quotient are computed. If no WITH PREC phrase were written, then the precision of the quotient would be determined by the precision of the product and C, since the result precision of an expression is determined by the result precisions of the individual operations performed. Note that if it were desired to compute a double precision product and then a single precision quotient, one would write:

```
VAR TEMP: FLOAT (20);
TEMP := A * B WITH PREC 20;
TEMP/C WITH PREC 10
```

Note that since floating point results are defined in terms of true mathematical results, TEMP is not first reduced to ten digits of precision. Instead, the entire 20 digits are used in computing the single precision quotient. Note that if WITH PREC 10 had not been specified, TEMP/C would

produce a result with 20 digits of precision, since the default precision rules yield results with the maximum precision of the operands.)

3.  $X := 1.0 \text{ WITH PREC } 15;$

(This is equivalent to writing 1. followed by 14 zeroes, or writing `TO_FLOAT(1.0, 15)` in place of 1.0.)

4.  $X := F(3.0, 7.5) + A \text{ WITH PREC } 15;$

(The WITH clause does not affect the precision with which 3.0 and 7.5 are represented, since these are not syntactic constructs of the form:

expression WITH PREC manifest-integer-expression

### 3.3.4.5. Machine-Dependent Infix Operators

#### Purpose

The operators defined here are intended for use in implementing multiple precision arithmetic not directly supported by an implementation. In addition, the operators provide access to machine dependent results when OVERFLOW or UNDERFLOW exceptions are raised. [See J3.3-12.]

#### Semantics

OP	NAME	RESULT	EXCEPTIONS
!*/!	divide	bitstring	
!*!	multiply	bitstring	
!+!	add	bitstring	
!-!	subtract	bitstring	

Each of these operators produces a bitstring result whose length and format is object computer dependent. None of the operators raise exceptions. The bitstring result represents the maximum amount of information derivable from the arithmetic operation, e.g., `!*!` applied to two single precision floating point operands will produce the machine-dependent form of a double precision product. This form can then be manipulated to whatever format double precision floating point values are to have. Similarly, the `!*` and `!*/` operators can be used to access the product or quotient when UNDERFLOW has been raised.

Constraints

These operators must be used only in machine-dependent portions of programs.

3.3.4.6. Relational Infix OperatorsPurpose

Relational operators are used to test floating point values for equality, for an ordering relationship, or to see if a value is in a specified range.

Semantics

The following relational operators have a predefined meaning for floating point operands.

OP	NAME	RESULT	EXCEPTIONS
=	equality	boolean	
<>	inequality	boolean	
>	greater than	boolean	
<	less than	boolean	
>=	greater/equal	boolean	
<=	less/equal	boolean	
IN	in-range	boolean	

For all operators except IN, the values (as determined by the representation) are compared, giving the value TRUE if the relation is satisfied. Otherwise the value is FALSE.

For  $E \text{ IN } [L:U]$ , the value is the value of

$$L \leq E' \& E' \leq U$$

where  $E'$  is the value of  $E$ . Note that  $E$  is evaluated only once.

For  $E \text{ IN } [L:U]$ , the value is the value of

$$L \leq E' \& E' < U$$

where  $E'$  is the value of  $E$ .

For  $E \text{ IN } (L:U)$ , the value is the value of

$$L < E' \& E' \leq U$$

where  $E'$  is the value of  $E$ .

For E IN (L:U), the value is the value of

L < E' & E' < U

where E' is the value of E.

### 3.3.5. Assignment

#### Syntax

[3-75] float-assignment ::= variable := float-expression;

#### Semantics

If the implemented precision of the float-expression is the same as the implemented precision of the variable, the value of the expression becomes the value of the variable.

If the implemented precisions are not the same, the value assigned the variable is the same as the value produced by

TO\_FLOAT(E,PV,R)

where E is the value of float-expression, PV is the implemented precision of the variable, and R is the rounding mode of the expression.

If the value to be assigned lies outside the range specified for the variable, the RANGE exception is raised, and the value of the variable is not changed.

#### Constraints

The variable must be of type class FLOAT.

The float-expression's value, after conversion to the variable's implemented precision, must not lie outside the range specified for the variable.

#### Exceptions

The RANGE exception is raised if the value to be assigned the variable lies outside the range specified for the variable. Note that this implies if explicitly converting the expression value to the variable's implemented

**Floating Point**

2/15/78

precision would raise the OVERFLOW exception, the RANGE exception is raised instead.

### 3.3.6. Predefined Functions

#### 3.3.6.1. TO\_FLOAT

##### Purpose

The TO\_FLOAT function is used to convert from different types to a floating point value of specified precision. The converted value may be rounded or truncated.

##### Syntax

```
[3-76] to-float-function ::= TO_FLOAT (to-float-value, float-precision  
[ , rounding-mode])  
  
[3-77] to-float-value ::= fixed-expression  
| float-expression  
| charstring-expression
```

##### Semantics

The value of the second argument (which must be a manifest constant) specifies the nominal precision of the value returned by the function. The implemented precision is the smallest implemented precision not less than the specified precision. Rounding and truncation are performed with respect to the implemented precision. The third argument, if present, specifies whether the abstract value of the to-float-value is to be rounded or truncated depending on whether the value of the third argument is ROUNDED or TRUNCATED, respectively. If the third argument is omitted, the value will be rounded (the normal default for floating point operations). TO\_FLOAT is not predefined for NATIVE\_T and NATIVE\_R rounding modes.

If TO\_FLOAT is applied to a fixed point value, a floating point value closest to the abstract fixed point value is produced, where the definition of "closest" depends on whether result values are to be rounded or truncated. TO\_FLOAT is predefined only for fixed point values of STD precision.

In an expression of the form TO\_FLOAT(E,P) where E is a float-expression, if  $P \geq E.\text{IMP\_PRECISION}$ , the abstract value of E is not changed and the implemented precision is increased to the next supported implementation precision  $P'' = \text{IMP\_PRECISION}(p)$  (see 3.3.6.4). Otherwise,  $P < E.\text{IMP\_PRECISION}$ . If there exists an implemented precision value  $P'$  such that  $P' = \text{IMP\_PRECISION}(P)$  and  $P' < E.\text{IMP\_PRECISION}$ , then the value of E is converted to the closest exactly representable  $P'$  precision value. [See J3.3-13.]

If L in TO\_FLOAT(L,P) is a float-literal and L has fewer than P significant digits, the effect is the same as if the mathematically defined value of L were written with P significant digits and then processed as a float-literal. If L has P or more digits and the implemented precision for variables of type TO\_FLOAT(P) is  $P' \geq P$ , then the mathematically defined value of L is represented as accurately as possible with implemented precision  $P'$ .

#### Examples

1.  $X := \text{TO\_FLOAT}(.1,5) \text{ ROUNDED};$

(The effect is the same as if the programmer had written .10000 assuming  $\text{NOM\_PRECISION}.\text{MAX} > 4$ , i.e., the maximum supported precision is greater than 4 decimal digits.)

2.  $\text{TO\_FLOAT}(.1,10)$

(Note that  $\text{TO\_FLOAT}(.1,10)$  need not equal .1 (which does equal  $\text{TO\_FLOAT}(.1,1)$ ), if the implemented precision of  $\text{TO\_FLOAT}(.1,10)$  is greater than the implemented precision of  $\text{TO\_FLOAT}(.1,1)$  and the radix of the floating point representation does not permit .1 to be represented exactly.)

3.  $X := Y + Z \text{ ROUNDED};$

(If the implemented precision of X is less than the implemented precision of  $Y + Z$ , the value of  $Y + Z$  is rounded to the implemented precision of X. The effect is the same as writing

```
X := \text{TO\_FLOAT}(Y + Z, X.\text{IMP\_PRECISION}, \text{ROUNDED});
```

except that if  $\text{TO\_FLOAT}$  would raise OVERFLOW, RANGE is raised instead. Note that if the expression is evaluated in NATIVE\_R or NATIVE\_T rounding mode and the implemented precision of  $Y + Z$  is not the same as the implemented

precision of X, then TO\_FLOAT would have to have been defined to accept the NATIVE\_R or NATIVE\_T rounding modes, since it is not predefined for these modes.)

If C is a charstring value, TO\_FLOAT(C,P) has the same value as if the value of C were written as a literal, C', and TO\_FLOAT(C',P) evaluated. [See J3.3-14.]

TO\_FLOAT is predefined only for float-precision values not exceeding a single implementation-dependent implemented-precision value. TO\_FLOAT is also predefined only for ASCII charstring values. The definition of TO\_FLOAT can be extended (see Appendix D) to return values of more than one implemented precision or to deal with charstring values of different character sets.

TO\_FLOAT is not predefined for enumerated, boolean, or bitstring values. (However, a bitstring can be treated as a floating point value by using a type-specifier, e.g., [FLOAT(10)]\$\$(bitstring), where the bitstring must be of the correct length to represent a floating value whose specified precision is 10. This type-specifier form can only be used in machine dependent portions of programs (see Section 3.1.4)).

#### Constraints

The float-precision value must not exceed the maximum supported precision value, i.e., NOM\_PRECISION.MAX.

TO\_FLOAT is predefined only for rounding-modes ROUNDED and TRUNCATED. If definitions for other rounding-modes are provided by extension, TO\_FLOAT must be used in a machine-dependent portion of a program for these rounding-modes.

The value produced by TO\_FLOAT must not lie outside the range of representable values for the implemented precision of the value, otherwise, OVERFLOW is raised.

Exceptions

TO\_FLOAT raises the OVERFLOW exception if the returned value would lie outside the range of representable values for the implemented precision of the TO\_FLOAT result.

3.3.6.2. ABSSyntax

[3-78] float-abs-function ::= ABS (float-expression)

Semantics

The function ABS(E) yields the absolute value of E with the implemented precision of E. The OVERFLOW exception is raised if the mathematically defined absolute value of E exceeds the maximum value permitted for floating point values having E's implemented precision.

3.3.6.3. FLOAT\_MIN and FLOAT\_MAXPurpose

The FLOAT\_MIN and FLOAT\_MAX functions yield the minimum and maximum permitted values for a given nominal precision.

Syntax

[3-79] float-min-function ::= FLOAT\_MIN (float-precision)

[3-80] float-max-function ::= FLOAT\_MAX (float-precision)

Semantics

- These functions are predefined for all predefined float-precision values. They yield the minimum (FLOAT\_MIN) and maximum (FLOAT\_MAX) values permitted for the specified nominal precision. Note that since different nominal precision values may yield the same implemented precision value, and since the maximum range is a function of the implemented precision, different argument values for these functions can yield the same values.

Examples

1. FLOAT\_MAX (5)

(This gives a value of type FLOAT(5) that is the maximum floating point value permitted for a float value with NOM\_PRECISION equal to five.

Constraints

The value of float-precision must be greater than zero and not exceed the maximum permitted nominal precision value, i.e., (FLOAT.NOM\_PRECISION).MAX.

### 3.3.6.4. EPSILON

Purpose

The EPSILON function is used in deciding when two floating point values are "essentially" equal. [See J3.3-15.]

Syntax

[3-81] float-epsilon-function ::= EPSILON (float-precision)

Semantics

The argument of EPSILON specifies a nominal precision. The value of EPSILON (P) is defined as the smallest positive number such that

$$\text{FLOAT}(1.0, P) + \text{EPSILON}(P) \text{ TRUNCATED} > \text{FLOAT}(1.0, P).$$

Examples

1. ABS(A-B) > 3. \*EPSILON (A-B).NOM\_PRECISION \* A

(This will check whether A is equal to B within 3 low-order bits if floating point values are represented in binary notation.)

### 3.3.6.5. IMP\_PRECISION

Purpose

This function gives the implemented precision value for a specified nominal precision. By modifying this function, the range of implemented precision values can be extended. Of course, float operators must then also be extended to accept the extended range of implemented precision values.

Syntax

[3-82] float-imp-prec-function ::= IMP\_PRECISION(float-precision)

Semantics

The value of IMP\_PRECISION(N) is the same as the value of the IMP\_PRECISION attribute for a variable or constant specified to have nominal precision N.

Examples

1. IMP\_PRECISION(5)

(This has the same value as (TO\_FLOAT(1, 5)).IMP\_PRECISION. Note that this is a form of attribute-query using an expression as the argument.)

Constraints

The argument of IMP\_PRECISION must lie in the range [1:(FLOAT.NOM\_PRECISION).MAX].

### 3.4. ENUMERATED TYPES

Enumerated types provide a set of values whose names are unique within the type. The value sets may be ordered or unordered.

Enumerated types are usually used to provide a set of named states a variable can designate (e.g., the nose position of a simulated aircraft: UP, DOWN, LEVEL). They are also used to define different character sets for use with charstrings (see Section 3.7) and to provide a mnemonic notation for representing "unprintable" characters such as new-line, horizontal tab, form feed, etc.

The language provides a default representation for enumerated values, but programmers can specify another representation if they wish. Operators and functions that are predefined for the default representation must be programmer-defined if enumerated values are given a non-default representation.

#### 3.4.1. Attributes and Type Specifications

##### Syntax

[3-83]    enumerated-spec                    ::= ENUM (enumerated-literal,...) [range-spec]	
	UNORDERED (enumerated-literal,...)
[3-84]    enumerated-attr                    ::= MIN	
	MAX
	EXTENT

##### Semantics

An enumerated-spec of the form ENUM (E1, E2,..., En) [L:U] or UNORDERED (E1, E2, ..., En) implicitly defines values for the following attributes: [See J3.4-1.]

- . MIN -- The value of MIN is the value of L if a range-spec is present; otherwise MIN's value is E1. The type of MIN is ENUM (E1, E2, ..., En) or UNORDERED (E1, E2, ..., En), depending on the form of the enumerated-spec, i.e., the type of MIN includes all the values given in the enumerated-spec.

- . MAX -- The value of MAX is the value of U if a range-spec is given; otherwise, MAX's value is En. The type of MAX is the same as the type of MIN.
- . EXTENT -- The value of EXTENT is the number of enumerated-literals listed in the enumerated-spec if no range-spec is present; otherwise, it is the number of enumerated-literals between L and U inclusive. The type of EXTENT is INTEGER [0:FIXED\_MAX(0,2,STD)].

Enumerated types are divided into type classes depending on the enumerated-literals listed in the enumerated-spec, the order in which the enumerated-literals appear, and on whether the type name is ENUM or UNORDERED. Thus there is a type class associated with the form ENUM (E1,E2,...,En) for each unique sequence E1, E2, ..., En. The types in such a class are different if their MIN or MAX attributes differ, or if their retypes differ. Similarly, there is a type class associated with the form UNORDERED (E1,E2,...,En) for each unique sequence E1, E2, ..., En. Types in such a class are different if their retypes differ. [See J3.4-2.]

The elements of an enumerated type declared with the ENUM form are considered fully ordered, i.e., the ordering operators (<, >, <=, >=, IN), the SUCC and PRED functions, and the TO\_FIXED function are automatically defined for such a type. An enumerated type declared with the UNORDERED form has no intrinsic order. That is, the SUCC and PRED functions are not automatically defined for such a type, nor are any of the ordering operators or TO\_FIXED. A programmer may impose an ordering on such a type by providing definitions for those functions and operators. See section 3.12 for details. [See J3.4-3.]

If an enumerated type is exported from a module (see Section 3.12), its enumerated-literal values are also exported.

#### Constraints

Each enumerated-literal must be unique within the list of enumerated-literals.

The lower-bound and upper-bound of a range-spec must be of the enumerated-spec's type class.

The number of enumerated-literals in an enumerated-spec must not exceed FIXED\_MAX(0,2,STD), the largest integer fixed point value for STD precision.

#### Examples

1. VAR X: ENUM (\RED, \BLUE, \GREEN, \VIOLET);  
 VAR Y: ENUM (\RED, \BLUE, \GREEN, \VIOLET) [\RED:\BLUE];  
 VAR Z: ENUM (\RED, \BLUE, \GREEN, \VIOLET) [\GREEN:\VIOLET];  
 VAR B: ENUM (\GREEN, \VIOLET);

Normally ENUM (\RED, \BLUE, \GREEN, \VIOLET) would be defined with a type declaration (Section 3.12), so an identifier (e.g., COLOR) could be used in place of the enumerated-spec. When such a declaration is made, X, Y, Z could be declared as

```
TYPE COLOR = ENUM(//RED, //BLUE, //GREEN, //VIOLET);
VAR X: COLOR;
VAR Y: COLOR [\RED: \BLUE];
VAR Z: COLOR [\GREEN: \VIOLET];
```

Whether X, Y, and Z are declared in either way, the following observations hold. Variables X, Y, and Z are all in the same type class but have different types because their MIN and MAX attributes are not the same. Note that Z.MIN = \GREEN, but (Z.MIN).MIN = \RED (because the type of Z.MIN is ENUM(\RED, \BLUE, \GREEN, \VIOLET)). Variable B is in a different type class from the other variables (even though the literal names overlap) because the values of the type are different. Variable B would have a different type class even if it were defined as ENUM (\RED,\BLUE).

2. TO\_FIXED([COLOR]\$\$(\GREEN))

(Note that since \GREEN is a value of two enumerated type-classes, we must distinguish which type class the literal argument belongs to. For the type-class represented by COLOR, the value of TO\_FIXED is 2.)

### 3.4.2. Literals

#### Syntax

```
[3-85] enumerated-literal ::= enumerated-id
                           | printing-char-id
                           | non-printing-char-id

[3-86] enumerated-id ::= identifier

[3-87] printing-char-id ::= '\'character-no-apos'
                           | \''

[3-88] non-printing-char-id ::= note: an enumerated-id whose first character
                                is \

[3-89] character-no-apos ::= note: all elements of character except
                                the apostrophe
```

#### Semantics

The printing-char-id form is used in defining character sets. The details are explained in Section 3.7. However, an enumerated type class containing at least one printing-char-id is called a charset enumerated type.

#### Constraints

To help distinguish enumerated-literals from identifiers used for other purposes, we recommend that the first character of an enumerated-literal be a \. However, this is simply a naming convention; it is not a language restriction, since it is sometimes convenient to have enumerated-ids of other forms (e.g., ROUNDED; see Section 3.2.4).

### 3.4.3. Representation Specification

#### Syntax

```
[3-90] enumerated-reptype ::= ENUM REP (size)
          | ENUM VAL REP (size,
                           VALUES (value-spec,...));
          | ENUM STD

[3-91] value-spec      ::= enumerated-literal AS
                           manifest-bitstring-expression

[3-92] enumerated-rep-attr ::= SIZE
```

#### Semantics

An enumerated-reptype of the form ENUM REP (N) specifies the value of the SIZE attribute:

- SIZE -- the value of this attribute specifies the number of bits used to represent values of the type. The values are stored right-justified within the specified number of bits. The type of SIZE is INTEGER [0: (FIXED.SIZE).MAX]. [See J3.4-4.]

An enumerated-reptype of the form ENUM VAL REP (N, VALUES(...)) specifies the SIZE attribute to be N. The VALUES argument specifies the actual representation of enumerated literals, namely, each enumerated literal has the representation specified by the manifest-bitstring-expression.

If the ENUM REP form is used, the enumerated-literals have a default representation, namely, the bitstring representations corresponding to successive unsigned integer values beginning with zero. The enumerated literals are assigned values in the order in which they appear in the associated enumerated-spec.

If no enumerated reptype is explicitly specified in a type-spec, the reptype is assumed to be ENUM STD, which is equivalent to ENUM REP (S). The value of S is the smallest number of bits needed to store the value N-1 as an unsigned integer, where N is the number of enumerated literals in the type class.

Constraints

If SIZE is explicitly specified, the value must be sufficiently large to represent each enumerated-literal, whether each enumerated value is given a default or programmer-specified representation.

In a VALUES list, all enumerated-literals of the type class must be given an explicit representation.

Two enumerated-literals in the same type class must not be specified to have the same representation.

The length of the bitstring-literals must equal the explicitly specified number of bits in the representation. [See J3.4-5.]

The ENUM\_VAL REP retype can only be specified for unordered enumerated types. (In Section 3.12, we show how a character set with a non-standard representation can be made to appear like an ordered enumerated type.)

Examples

```
1. VAR X: UNORDERED (\EXCELLENT, \GOOD, \FAIR, \POOR)
   REP ENUM_VAL REP (ASCII.SIZE, VALUES (
      \EXCELLENT AS TO_BIT ('E'),
      \GOOD AS TO_BIT ('G'),
      \FAIR AS TO_BIT('F'),
      \POOR AS TO_BIT ('P')));
```

(This example shows an enumerated variable having four values whose representation is non-standard. Each value is represented by the ASCII character code of its initial letter. ASCII.SIZE is the number of bits used to represent an ASCII character.

```
2. Y.SIZE
```

(If the definition of the variable Y is the one given earlier in 3.4.1, the value of Y.SIZE is 2, since there are four values in Y's type class.)

### 3.4.4. Operators and Expressions

#### Syntax

[3-93] enumerated-expression ::= expression

[3-94] manifest-enumerated-expression ::= manifest-expression

#### Constraints

An enumerated-expression or manifest-enumerated-expression must be an expression yielding a value of some enumerated type.

#### 3.4.4.1. Infix Operators

##### Semantics

Infix operators that can be applied to enumerated values are:

OP	NAME	RESULT	EXCEPTION
=	equality	boolean	
<>	inequality	boolean	
>	greater than	boolean	
<	less than	boolean	
>=	greater/equal	boolean	
<=	less/equal	boolean	
IN	range	boolean	

The four ordering operators (i.e., >, <, >=, <=) produce the value TRUE if the ordering relation is satisfied when the representations are compared as unsigned integer values.

All enumerated types, whether ordered or unordered, and whether they have default representation or not, have predefined equality and inequality relational operators. These operators produce the value TRUE if the relation is satisfied when the representations of the operands are compared.

The IN infix operator is defined to accept an ordered enumerated value as its first operand and a range-spec as its second operand. An expression of the form E IN range-spec is evaluated as follows:

For  $E \text{ IN } [L:U]$ , the value is the value of

$L \leq E' \& E' \leq U$

where  $E'$  is the value of  $E$ . Note that  $E$  is evaluated only once.

For  $E \text{ IN } [L:U]$ , the value is the value of

$L \leq E' \& E' < U$

where  $E'$  is the value of  $E$ .

For  $E \text{ IN } (L:U)$ , the value is the value of

$L < E' \& E' \leq U$

where  $E'$  is the value of  $E$ .

For  $E \text{ IN } (L:U)$ , the value is the value of

$L < E' \& E' < U$

where  $E'$  is the value of  $E$ .

The four ordering and the IN operator are not automatically defined for unordered enumerated values. (If ordering operators have been programmer-defined for the enumerated type, then these operators determine the truth value of the ordering relation, see Section 3.12.)

#### Constraints

The ordering operators can only be applied to enumerated values from the same enumerated type class. These operators are not automatically defined for UNORDERED enumerated types.

- In an expression of the form  $E \text{ IN range-spec}$ ,  $E$ , the lower-bound value, and the upper-bound value must all have the same enumerated type class. IN cannot be used unless  $<$ ,  $>$ ,  $\leq$ , and  $\geq$  are defined for the type class.

### 3.4.5. Assignment

#### Syntax

[3-95] enumerated-assignment ::= variable := enumerated-expression;

#### Semantics

The value of the expression replaces the value of the variable if the value of the expression is within the specified (or default) range for the variable. If not within range, the RANGE exception is raised.

If the type-class of the enumerated-expression is ambiguous, it is automatically considered to be the type class of the variable. (Such ambiguities can arise only when the expression is an enumerated-literal.)

#### Constraints

The variable must be of an enumerated type.

The expression and variable must be of the same type class, and, if the type is unordered, and the expression or variable has an enumerated-reptype of the ENUM\_VAL REP form, the other must have the same enumerated-reptype.

#### Examples

1. B := \GREEN;

(This assignment statement can also be written using a type-specifier:

B := [ENUM (\GREEN, \VIOLET)]\$\$(\GREEN);

The shorter form can be used because there is no ambiguity given the declaration of B and COLOR in Section 3.4.1.)

### 3.4.6. Predefined Functions

#### 3.4.6.1. SUCC and PRED

##### Purpose

The successor (SUCC) and predecessor (PRED) functions generate the next and preceding elements of an ordered enumerated type class, respectively.

##### Syntax

[3-96] enumerated-succ-function ::= SUCC (enumerated-expression)

[3-97] enumerated-pred-function ::= PRED (enumerated-expression)

##### Semantics

SUCC and PRED are functions automatically defined for only ordered enumerated types.

SUCC returns the next enumerated value in the type class of its argument. If the value of its argument, E, is the last value of the argument's type class (i.e., (E.MAX).MAX), the RANGE exception is raised.

PRED returns the preceding enumerated value in the type class of its argument. If the value of its argument, E, is the first value of the argument's type class (i.e., (E.MIN).MIN), the RANGE exception is raised.

##### Examples

###### 1. SUCC(Y)

(Using the declarations of 3.4.1, if Y has the value \BLUE, i.e., Y = Y.MAX, then SUCC(Y) = \GREEN, the next value in Y's type class.)

.

### 3.5. BOOLEAN

The boolean values, TRUE and FALSE, are produced by the relational operators (e.g., equality) and by the boolean operators for complement, conditional conjunction, and conditional disjunction. [See J3.5-1.]

#### 3.5.1. Attributes and Type Specification

##### Syntax

- [3-98] boolean-spec ::= BOOLEAN
- [3-99] boolean-attr ::= note: there are none

##### Semantics

The value set for the BOOLEAN type-spec contains two values, represented by the literals TRUE and FALSE. The BOOLEAN type has no attributes that can be queried.

#### 3.5.2. Literals

##### Syntax

- [3-100] boolean-literal ::= TRUE  
| FALSE

##### Semantics

These literals represent the only possible BOOLEAN values.

#### 3.5.3. Representation Specifications

##### Syntax

- [3-101] boolean-reptype ::= BOOLEAN\_STD
- [3-102] boolean-rep-attr ::= SIZE

##### Semantics

A boolean-reptype defines the value of the SIZE attribute. The value of SIZE is one.

### 3.5.4. Operators and Expressions

#### Syntax

[3-103] boolean-expression ::= expression

#### Constraints

A boolean-expression or manifest-boolean-expression must be an expression producing a BOOLEAN value.

#### 3.5.4.1. Infix Operators

##### Syntax

[3-104] boolean-manifest-expression ::= manifest-expression

##### Semantics

The following infix operators have a predefined meaning for BOOLEAN operands.

OP	NAME	RESULT	EXCEPTIONS
&	conditional and boolean		
!	conditional or boolean		
=	equality boolean		
<>	inequality boolean		

In addition, an operator symbol " $\rightarrow$ " is available for defining as the "logical implication" operator in assertions (see Section 4.5.8).

An expression of the form E & F has the value TRUE if and only if both E and F have the value TRUE; otherwise it has the value FALSE. In addition, E is evaluated before F, and if the value of E is FALSE, F is not evaluated.

An expression of the form E ! F has the value FALSE if and only if both E and F have the value FALSE; otherwise it has the value TRUE. In addition, E is evaluated before F, and if the value of E is TRUE, F is not evaluated.

An expression of the form E = F (E <> F) has the value TRUE (FALSE) if and only if both E and F have the same values and has the value FALSE (TRUE) if and only if E and F have different values.

(The operators AND, OR, and XOR are predefined only for bitstring values, but these definitions can be extended to boolean values if a user so chooses (see Appendix D). The difference between an extended definition of AND and the & definition is that for the AND operator, both operands are evaluated whereas for the & operator, only the first operand will be evaluated in certain cases. In addition, the order of operand evaluation is arbitrary for AND but it is left-to-right for &. An extended definition of OR has the same properties with respect to the definition of !.) Note that & and ! cannot be functions since both operands of & and ! are not always evaluated and arguments of functions are always evaluated before the function is invoked.

#### Constraints

A boolean-expression is a manifest-boolean-expression if it is a boolean-literal or an expression of the form E < F, E > F, E <= F, or E >= F where E and F are manifest-expressions. Since a boolean-expression of the form E IN range-spec is considered an abbreviation for B1 & B2, where the form of B1 and B2 depends on the form of the range-spec (see 3.1.4.1, 3.2.4.6, 3.3.4.6, and 3.4.4.1), the next rule applies when the IN operator is used.

An expression of the form E & F is a manifest-boolean-expression if the value of E or the value of F is manifestly FALSE or both E and F are manifest-boolean-expressions.

An expression of the form E ! F is a manifest-boolean-expression if the value of E or the value of F is manifestly TRUE or both E and F are manifest-boolean-expressions.

An expression of the form NOT E is a manifest-boolean-expression if E is a manifest-boolean-expression. [See J3.5-2.]

Efficiency Considerations

Note that for manifest-boolean-expressions of the form E & FALSE, E must be evaluated unless E's evaluation is known to have no side-effects, since optimization cannot change the semantics of the language. Similarly, for E ! TRUE, E must be evaluated if the evaluation would have side-effects.

Examples

1. A & 3 > 5

(Even if A is a variable, this is a manifest-boolean-expression. If A is a function with side-effects, it must still be invoked however.)

2. INLINE FUNCTION F (ARG: INTEGER) RETURNS...;

  VAR A: BOOLEAN;

  ...

  IF A & 3 > ARG THEN

  ...

  ELSE

  ...

  END IF;

END FUNCTION F;

(F(5) will cause conditional compilation of the else branch of the if statement; F(2) will cause no conditional compilation.)

### 3.5.4.2. Prefix Operators

#### Semantics

The following prefix operator has a predefined meaning for a BOOLEAN operand:

OP	!NAME!	RESULT	EXCEPTIONS
NOT !not !boolean!			

An expression of the form NOT E has the value FALSE if the value of E is TRUE and the value TRUE if the value of E is FALSE.

### 3.5.5. Assignment

#### Syntax

[3-105] boolean-assignment ::= variable := boolean-expression;

#### Semantics

The value of the expression replaces the value of the variable.

#### Constraints

The type of the variable must be BOOLEAN.

### 3.5.6. Predefined Functions

#### 3.5.6.1. HANDLER\_EXISTS

##### Syntax

[3-106] handler-exists-function ::= HANDLER\_EXISTS (exception-name)

##### Semantics

The use of exception-names is described in 4.5.7. This function returns the value TRUE if an exception handler exists for the specified exception-name, i.e., if signaling the specified exception-name (in the statement containing the handler-exists-function) would not be an error. [See J3.5-3.]

### 3.5.6.2. IS\_MANIFEST

#### Syntax

[3-107] manifest-function ::= IS\_MANIFEST(expression)

#### Semantics

This function returns the value TRUE if the expression is a manifest-expression; otherwise it returns FALSE. [See J3.5-4.]

This function is a manifest-expression.

The argument of the function is not evaluated.

### 3.6. BIT STRINGS

Bit strings are fixed length sequences of binary digits. They are intended primarily for use in dealing with representations of values, e.g., for specifying the representation of character sets, for writing conversion routines from numeric values to character set codes, for implementing multiple precision arithmetic by extension, etc. A secondary purpose of bitstrings is to manipulate sets of values, as in PASCAL. However, the full range of set operators is not predefined for this purpose.

Only fixed length bit strings are supported directly in the language.

#### 3.6.1. Attributes and Type Specification

##### Syntax

[3-108]	bitstring-spec	::= BIT (length-expression)
[3-109]	length-expression	::= integer-expression
[3-110]	bitstring-attr	::= LENGTH

##### Semantics

A bitstring-spec of the form BIT(L) defines the value of the LENGTH attribute to be the value of L. This value determines the number of bit positions in the bitstring. The type of LENGTH is INTEGER [0:max-length], where max-length is not larger than the largest supported fixed point integer value and is otherwise as large as possible for the object computer.

The bits in the bitstring are numbered left to right, starting with zero.

[See J3.6-1.]

##### Exceptions

The SIZE exception is raised if the value of the length-expression exceeds (BIT.LENGTH).MAX, i.e., the maximum permitted bitstring length.

### 3.6.2. Literals

#### Purpose

Depending on the application and on the object computer, it will be clearest to specify bitstring literals in binary, octal, or hexadecimal. All three literal formats are supported.

#### Syntax

[3-111] bitstring-literal	::= BIN [display-char]... {bin-string [display-char]...} ...   OCT [display-char]... {oct-string [display-char]...} ...   HEX [display-char]... {hex-string [display-char]...} ...
[3-112] bin-string	::= '[bin-char blank]...'
[3-113] oct-string	::= '[oct-char blank]...'
[3-114] hex-string	::= '[hex-char blank]...'
[3-115] bin-char	::= 0 1
[3-116] oct-char	::= 0 1 2 3 4 5 6 7
[3-117] hex-char	::= 0 1 2 3 4 5 6 7 8 9 A B C D E F

#### Semantics

The type of a bitstring-literal is BIT(L), where L is a function of N, the number of bin-chars, oct-chars, or hex-chars in a sequence of strings. L = N for a sequence of bin-strings, L = 3\*N for a sequence of oct-strings, and L = 4\*N for a sequence of hex-strings.

The value of a bitstring-literal is the value of the corresponding binary, octal, or hexadecimal number; embedded blanks and display characters are ignored. [See J3.6-2.]

#### Examples

1. BIN '101 110'

(This literal has the same value as OCT '56' or BIN '101110'. Its type class is BIT and LENGTH is 6.)

## 2. HEX 'OFF'

(This literal has the same value as BIN '0000 1111 1111'. Its type class is BIT and LENGTH is 12.)

### 3.6.3. Representation Specification

#### Syntax

[3-118] bitstring-reptype ::= BIT\_STD

[3-119] bitstring-rep-attr ::= SIZE

#### Semantics

The value of the SIZE attribute is equal to the number of bits in the representation of the bitstring value. For standard representation, the value of SIZE equals the value of LENGTH.

If no bitstring-reptype is specified, the default representation is BIT\_STD. This is the only predefined reptype.

Successive bits of a value are adjacent in the representation of the value for the BIT\_STD representation.

### 3.6.4. Operators and Expressions

#### Syntax

[3-120] bitstring-expression ::= expression

[3-121] manifest-bitstring-expression ::= manifest-expression

#### Constraints

A bitstring-expression or manifest-bitstring-expression must be an expression yielding a value in the BIT type-class.

#### 3.6.4.1. Infix Operators

##### Semantics

The following infix operators have a predefined meaning for bitstring-expression operands:

OP	NAME	RESULT	EXCEPTIONS
AND	conjunction	bitstring	INVALID_LENGTH
OR	disjunction	bitstring	
XOR	exclusive disj.	bitstring	
!!	concatenate	bitstring	INVALID_SIZE
[L:U]	substring	bitstring	INVALID_SUBSTRING
=	equality	boolean	INVALID_LENGTH
<>	inequality	boolean	INVALID_LENGTH

The AND, OR, and XOR operators accept only bitstring operands of the same length and produce a result by performing the logical and, logical or, and logical exclusive-or, respectively, on each bit of the operands. The result is a bitstring of the same length as each of the operands. [See J3.6-3.]

An expression of the form E !! F produces a bitstring whose value is the value of E followed by the value of F. Concatenation can be used to shift bitstring values, e.g., BIN '00' !! E[0:E.LENGTH-2] shifts the value of E right two places.

An expression of the form V[L:U] when V is a bitstring, has the value of bits L through U of V. If L is equal to zero then the extracted substring begins with the first bit of V's value. The LENGTH attribute of the value is U

- L + 1, except that if the value of L exceeds the value of U, the value of LENGTH is zero, i.e., substrings of length zero can be specified. The bits in V[L:U] are renumbered so bit L of V is bit 0 of V[L:U]. [See J3.6-4.]

The equality and inequality operators accept as operands only strings of the same length. The equality operator produces the result TRUE if the operands have the same value, and otherwise, the value FALSE. The inequality operator produces the result TRUE if the operands have different values; otherwise the result is FALSE.

#### Constraints

Operands of AND, OR, XOR, =, and <> must have the same length (see INVALID\_LENGTH exception).

The value of L must not be less than zero and the value of U must not exceed LENGTH - 1 (see SUBSTRING exception).

The length of the result of concatenation must not exceed LENGTH.MAX, i.e., the maximum permitted bitstring length.

#### Exceptions

INVALID\_SUBSTRING is raised if the value of L is negative or the value of U exceeds LENGTH-1.

INVALID\_LENGTH is raised if strings of unequal length are evaluated as operands of AND, OR, XOR, =, or <>.

INVALID\_SIZE is raised if the result of concatenation exceeds the maximum permitted bitstring length.

#### 3.6.4.2. Prefix Operators

##### Semantics

The only predefined prefix operator is NOT, which takes a single bitstring-expression argument and produces a bitstring result of the same length as the argument. Zero bits of the argument are changed to one bits in

the result and vice versa.

### 3.6.5. Assignment

#### Syntax

[3-122] bitstring-assignment ::= bit-target := bitstring-expression;

[3-123] bit-target ::= variable  
| variable closed-range

#### Semantics

The value of the expression becomes the value of the target.

If a target is of the form V[L:U], the positions of V's value indicated by the values of L through U are replaced with the value of the expression; the value of the remaining positions is unchanged. If the lower bound of the closed-range is zero, replacement begins with the first bit of V.

The LENGTH attribute of a target of the form V[L:U] is U-L+1, except that if the value of L exceeds the value of U, the value of LENGTH is zero.

#### Constraints

The LENGTH attributes must be the same for the expression and target.

The target must be of type class BIT.

#### Exceptions

The SIZE exception is raised if the expression and target are not the same length.

The INVALID\_SUBSTRING exception is raised if the lower bound of the closed-range is negative or the upper bound is greater than the length of the variable. (Note that this permits closed ranges of the form [0:-1].) [See J3.6-5.]

### 3.6.6. Predefined Functions

#### 3.6.6.1. TO\_BIT

##### Syntax

[3-124] bitstring-to-bit-function ::= TO\_BIT (to-bit-value)

[3-125] to-bit-value ::= expression

##### Semantics

TO\_BIT returns the bitstring representation of the expression's value.

The type of the result is BIT ((expression).SIZE).

##### Constraints

Unless the expression is an enumerated-expression or a bitstring-expression, the to-bit-function must be invoked in a machine-dependent section of code (see Sections 4.5.5.2 and 4.8). [See J3.6-6.]

#### 3.6.6.2. DUP

##### Syntax

[3-126] bitstring-dup-function ::= DUP (bitstring-expression,  
integer-expression)

##### Semantics

DUP returns a bitstring result whose length is the value of the integer-expression and each bit has the value of the first argument. [See J3.6-7.]

##### Constraints

The first argument must be a bitstring of length one.

The second argument must not be negative or exceed the maximum permitted bitstring length, i.e., (BIT.LENGTH).MAX.

##### Exceptions

The INVALID\_SIZE exception is raised if the value of the second argument exceeds the maximum permitted bitstring length.

The INVALID\_LENGTH exception is raised if the value of the second argument is negative.

### 3.6.6.3. ZEROS

#### Syntax

[3-127] bitstring-zeros-function ::= ZEROS (integer-expression)

#### Semantics

ZEROS (N) is equivalent to DUP(BIN'0',N). This function is used in record packing and to pad bitstrings in assignments. Although DUP is more general, ZEROS is more convenient.

### 3.6.6.4. BIN\_OF, OCT\_OF, and HEX\_OF

#### Purpose

The BIN\_OF, OCT\_OF, and HEX\_OF functions convert character strings representing bitstring-literals to bitstring values. The functions are applied to strings representing binary, octal, or hexadecimal literals.

#### Syntax

[3-128] bitstring-bin-function ::= BIN\_OF (charstring-expression)

[3-129] bitstring-oct-function ::= OCT\_OF (charstring-expression)

[3-130] bitstring-hex-function ::= HEX\_OF (charstring-expression)

#### Semantics

For an invocation of the form BIN\_OF(E), OCT\_OF(E), or HEX\_OF(E), the value of E is considered to represent a string of bin-chars, oct-chars, or hex-chars, respectively. The value of E is converted to a bitstring value of length E.LENGTH, 3\*E.LENGTH, and 4\*E.LENGTH, respectively.

The BIN\_OF, OCT\_OF, and HEX\_OF functions are predefined only for ASCII charstring values. Programmers must supply definitions if the functions are to be applied to other character set values.

Examples

1. VAR S: ASCII\_STR(2): % S is an ASCII character string of length 2

...  
S := '10';

(The value of BIN\_OF(S) is the same as the value of BIN '10'. The value of OCT\_OF (S) is OCT '10'.) [See J3.6-8.]

Constraints

The value of the charstring argument must contain only bin-chars, oct-chars, or hex-chars for the BIN\_OF, OCT\_OF, and HEX\_OF functions, respectively (see CONVERSION exception).

The length of the returned result must not exceed (BIT.LENGTH).MAX, i.e., the maximum permitted bitstring length (see INVALID\_SIZE exception).

Exceptions

The INVALID\_SIZE exception is raised if the length of the returned value would exceed (BIT.LENGTH).MAX.

The CONVERSION exception is raised if the value of the charstring argument does not contain just bin-chars, oct-chars, or hex-chars.

### 3.7. CHARACTER STRINGS

Character strings are fixed length sequences of character codes. The ASCII character set is predefined for use with programs. Other character sets (i.e., character encodings) may be defined by users of the language. Concatenation and substring extraction and assignment are pre-defined operators for character strings.

Although only fixed length character strings are directly supported by the language, varying length strings (with a programmer-specified maximum length) can be supported using the type extension facilities of the language.

#### 3.7.1. Attributes and Type Specification

##### Syntax

[3-131]	charstring-spec	::= CHAR_STR (length-expression, charset-name)
[3-132]	charset-name	::= enumerated-type
[3-133]	enumerated-type	::= type-reference
[3-134]	charstring-attr	::= LENGTH   CHARSET

##### Semantics

A charstring-spec of the form CHAR\_STR (N, C) defines, respectively, the values of the LENGTH and CHARSET attributes. The type and significance of these attributes is as follows:

- . LENGTH -- the value of this attribute defines the number of characters in strings of this type. The type of the LENGTH attribute is INTEGER [0:max-length], where max-length is not larger than the largest supported fixed point integer value (i.e., FIXED\_MAX (0, 2, FIXED\_PRECISIONS.MAX)) and is otherwise as large as possible for the object computer.
- . CHARSET -- the value of this attribute is an ordered enumerated type. The type of CHARSET is TYPE.

For convenience, the type ASCII\_STR (length) is predefined to be equivalent to CHAR\_STR (length, ASCII), where ASCII represents the

128-character ASCII character set. (This character set is defined in 3.7.2.)

The characters in a charstring are numbered left to right, starting with zero.

Exceptions

The INVALID\_SIZE exception is raised if the value of the length-expression exceeds (CHAR\_STR.LENGTH).MAX, i.e., the maximum permitted charstring length.

### 3.7.2. Literals

#### Purpose

The format for character string literals permits non-printing characters such as horizontal tab, back-space, etc. to be visibly represented in string literals. Enumerated types are used to define sets of characters and their representations, as the examples below show.

#### Syntax

```
[3-135] charstring-literal ::= [charset-name] [display-char]...
                                {basic-string [display-char]...} ...

[3-136] basic-string      ::= '['basic-char]...
                                | enumerated-literal

[3-137] basic-char       ::= ''
                                | printing-char

[3-138] printing-char    ::= note: any character, C, appearing in a
                                printing-char-id, i.e.,
                                a literal of the form '\C'
```

#### Semantics

The character set associated with a particular charstring-literal depends on what character sets are available for use in the scope containing the literal. In essence, if only one character set is available, then all literals are assumed to be of that type. If more than one is available, then a programmer must explicitly specify which set is meant by using an explicit charset-name as part of the charstring-literal.

The abstract type of a charstring-literal is CHAR\_STR(L,C), where L is the number of basic-chars and enumerated-literals in the literal and C is the charset-name preceding the first basic-string (if one is present); otherwise, C is the default character set. The default character set is the charset enumerated-type known in the scope containing the literal. (This rule provides a way of changing default character sets; see example below.) A charset type is an enumerated type class containing at least one printing-char-id. [See

## J3.7-1.]

The ASCII charset-name is predefined for each compilation unit, i.e., unless another charset-name is explicitly made available, charstring-literals are ASCII.

A basic-char having the form '' is considered to represent a single apostrophe character.

A basic-string having the form of a single enumerated-literal is not considered a charstring-literal; it is considered an enumerated-literal. [See

## J3.7-2.]

Examples

```
1. TYPE ASCII = ENUM (
  \NUL, \SOH, \STX, \EXT, \EOT, \ENQ, \ACK, \BEL,
  \BS, \HT, \LF, \VT, \FF, \CR, \SO, \SI,
  \DLE, \DC1, \DC2, \DC3, \DC4, \NAK, \SYN, \ETB,
  \CAN, \EM, \SUB, \ESC, \FS, \GS, \RS, \US,
  '\ ', '\!', '\!', '\$', '\$', '\$', '\$',
  '\(', '\)', '\+', '\+', '\-', '\.', '\/',
  '\0', '\1', '\2', '\3', '\4', '\5', '\6', '\7',
  '\8', '\9', '\:', '\;', '\<', '\=', '\>', '\?',
  '\@', '\A', '\B', '\C', '\D', '\E', '\F', '\G',
  '\H', '\I', '\J', '\K', '\L', '\M', '\N', '\O',
  '\P', '\Q', '\R', '\S', '\T', '\U', '\V', '\W',
  '\X', '\Y', '\Z', '\[', '\]', '\^', '\_',
  \ACCENT, \A, \B, \C, \D, \E, \F, \G,
  \H, \I, \J, \K, \L, \M, \N, \O,
  \P, \Q, \R, \S, \T, \U, \V, \W,
  \X, \Y, \Z, \[, \BAR, \], \TILDE, \DEL);
```

(This is an example of how the full 128 ASCII character set can be defined using only characters in the 64 character subset. The definition exploits the fact that the default representation starts with zero. An example of a character set definition for which this is not true is given in 3.11.

This enumerated type is considered to be a charset type by virtue of the fact that it contains at least one printing-char-id (e.g., '\A', '\B', etc.). Note that the enumerated-literals of the form '\x' specify which characters can be used directly in string literals, e.g., because there is an enumerated-literal of the form '\A', the character A can be used in a string literal. Since no lower case characters appear in this form, the character set definition shows that lower case letters cannot be used directly. However, the position of \A in the ASCII definition corresponds to the position of lower case A in the collating sequence, and so \A can be used as enumerated-literals in strings (see examples below).

In a system like MULTICS which uses 9 bits to represent a character, this definition would include REP ENUM<sub>REP(9)</sub>.

A definition for full ASCII for an implementation which has the lower case letters and some other symbols as printing characters would be given as shown in the following example.)

```
2. TYPE ASCII = ENUM (
  \NUL, \SOH, \STX, \EXT, \EOT, \ENQ, \ACK, \BEL,
  \BS, \HT, \LF, \VT, \FF, \CR, \SO, \SI,
  \DLE, \DC1, \DC2, \DC3, \DC4, \NAK, \SYN, \ETB,
  \CAN, \EM, \SUB, \ESC, \FS, \GS, \RS, \US,
  '\ ', '\"', '\"$', '\"$', '\"$', '\"&', '\"''',
  '\"(', '\"')', '\"+', '\"-', '\".', '\"/',
  '\"0', '\"1', '\"2', '\"3', '\"4', '\"5', '\"6', '\"7',
  '\"8', '\"9', '\":', '\";', '\"<', '\"=', '\">', '\"?',
  '\"@', '\"A', '\"B', '\"C', '\"D', '\"E', '\"F', '\"G',
  '\"H', '\"J', '\"I', '\"K', '\"L', '\"M', '\"N', '\"O',
  '\"P', '\"Q', '\"R', '\"S', '\"T', '\"U', '\"V', '\"W',
  '\"X', '\"Y', '\"Z', '\"[', '\"\\', '\"]', '\"^', '\"_',
  '\"`', '\"a', '\"b', '\"c', '\"d', '\"e', '\"f', '\"g',
  '\"h', '\"i', '\"j', '\"k', '\"l', '\"m', '\"n', '\"o',
  '\"p', '\"q', '\"r', '\"s', '\"t', '\"u', '\"v', '\"w',
  '\"x', '\"y', '\"z', '\"{', '\"|', '\"}', '\"-', \DEL);
```

### 3. \HT 'RING THE BELL' \BEL \CR \LF

(This is an ASCII string consisting of a horizontal tab character followed by "RING THE BELL" followed by a BEL character and a newline sequence. Note that the spaces outside the apostrophe are permitted but not required. The string could also have been written as

\HT'RING THE BELL'\BEL\CR\LF

### 4. 'THIS IS A' 'MULTI-LINE LITERAL'

(Note that since any display-char can separate basic-strings, literals too long for a single line can be split across two lines by creating two or more basic-strings. This is the same as the value of 'THIS IS A MULTI-LINE LITERAL'.)

### 5. 'THIS SPACE' ' IS NECESSARY'

(This is equivalent to 'THIS SPACE IS NECESSARY'. If the blank were omitted, as in 'THIS SPACE' ' IS NECESSARY', the pair of apostrophes would form a basic-char and represent a single apostrophe, giving the result THIS SPACE' IS NECESSARY.)

## 6. 'L'\O\W\E\R' C'\A\S\E

(The value is "Lower Case" if the default ASCII character set (example 1 above) is used. If the other character set definition is used (example 2 above), the string could be entered directly as 'Lower Case', because the lower-case characters are given in the form of print-char-ids.)

## 7. EBCDIC 'THIS IS A STRING'

(Assuming two character sets are available, EBCDIC and ASCII, this charstring-literal represents a sequence of EBCDIC character codes.)

## 8. C := 'A';

(C must be a charstring-variable with length 1. Note that C := \HT; would be illegal, since \HT is an enumerated-literal, not a string; see next example.)  
[See J3.7-3.]

## 9. C := ''\HT

(This assigns the string consisting of a single horizontal tab character to the variable C. Note that since ''\HT consists of two basic-strings (the null string followed by the horizontal tab character), it is a charstring-literal and not an enumerated-id. Note that '''\HT would be a two-character string consisting of an apostrophe followed by a Horizontal tab character.)

## 10. SEGMENT S USES (CHARSET (TYPES: (EBCDIC)));

...  
CONST ASCII: INTEGER := 0;

...  
C := 'EBCDIC STRING';

A SEGMENT is a compilation unit. The USES clause identifies a SEGMENT called CHARSET and makes the name EBCDIC available for use inside S. We assume here that EBCDIC defines the EBCDIC character set. Next, by redefining the automatically imported ASCII charset-name, the charstring-literal is automatically considered to be a sequence of EBCDIC characters if no other character sets are imported or defined. Normally this redefinition of the default character set would be in an application library imported into all compiled programs, so this example illustrates how it can be done rather than what applications programmers would be expected to do.)

Constraints

A charset-name must be an enumerated-type whose elements contain at least one enumerated-literal of the form printing-char-id (i.e., the form '\C'; see Section 3.4.2). [See J3.7-4.]

The form with an explicit charset-name must be used if more than one charset-name exists in the scope in which the literal appears.

(Note that it is an error for a new-line to be used as a basic-char, i.e., a basic-string cannot cross lines.)

(An implementation may choose to redefine the ASCII charset-name so lower case letters are directly representable in charstring-literals.

### 3.7.3. Representation Attributes

#### Syntax

```
[3-139] charstring-reptype ::= CHAR_STR REP (packing-mode)
          | CHAR_STR STD

[3-140] packing-mode ::= manifest-enumerated-expression

[3-141] charstring-rep-attr ::= PACKING
          | SIZE
```

#### Semantics

A charstring-reptype of the form CHAR\_STR REP (P) defines the value of the PACKING attribute to be the value of P and the SIZE attribute implicitly.

- PACKING -- the value of this attribute implies how individual characters in a string are organized with respect to each other. The type of PACKING is UNORDERED (DENSE, PACKED, UNPACKED).

If the value of PACKING is DENSE, characters are stored without regard to word boundaries in the object machine; adjacent characters are never separated by filler bits. (This is the same representation as a DENSE array of enumerated values.)

If the value is PACKED, characters are stored so they do not cross word boundaries but are otherwise stored with as many characters per word as possible. The position of filler bits (which are needed if the number of bits per character is not an exact divisor of the word length) is implementation-dependent. (This is the same representation as a PACKED array of enumerated values.)

If the value of PACKING is UNPACKED, the representation is the same as if each character were a simple-variable of enumerated type. (This is the same as the representation for an UNPACKED array of enumerated values.)

- SIZE -- the value of this attribute is the number of bits occupied by the string. The number of bits is counted beginning with the first bit of the first character of the string and ending with the last bit of the last character. The value of SIZE is a function of the number of characters in the string and the packing mode.

The default representation of character string values is CHAR\_STR STD, which is defined to be equivalent to CHAR\_STR REP (PACKED). Hence, character strings are PACKED by default.

Examples

If character codes are seven bits in length, 5 characters can fit in a 36-bit word. Whether the extra bit is at the left or right in the word (or somewhere else, for that matter) is implementation-dependent. Such a storage allocation strategy is specified by the PACKED attribute. If DENSE were specified, the first bit of the sixth character in a string would occupy the rightmost bit of the word and the six remaining bits would be at the left of the next word.

### 3.7.4. Operators and Expressions

#### Syntax

[3-142] charstring-expression ::= expression

#### Constraints

A charstring-expression is one yielding a value of type class CHAR\_STR.

#### 3.7.4.1. Infix Operators

##### Semantics

The following infix operators have a predefined meaning for charstring-expression operands of the same character set.

OP	NAME	RESULT	EXCEPTIONS
!!	concatenate	charstring	INVALID_SIZE
[L:U]	substring	charstring	INVALID_SUBSTRING
=	equality	boolean	INVALID_LENGTH
<>	inequality	boolean	

The concatenation operator (!! ) produces a string whose value is the value of the first operand followed by the value of the second operand. The representation of the result is the representation of the operands. (Note: if the packing of the operands is not the same, one of the operands must be converted to the packing of the other unless the definition of !! has been explicitly extended to define what the representation of the result will be in this case.) [See J3.7-5.]

An expression of the form V[L:U] has the value of positions L through U of the value of V. The packing of the result is the packing of V. If L is zero, the first character of V is identified. The LENGTH of the result is U - L + 1 except that if L > V, the value of LENGTH is zero.

The equality and inequality operators accept as operands only strings of the same length (the packing attributes may be different). The equality operator produces the result TRUE if the operands have the same value; the inequality operator produces the result TRUE if the operands have different

values; if the value is not TRUE, it is FALSE.

#### Constraints

The value L must not be less than zero and the value of U must not exceed LENGTH - 1.

Strings being compared for equality or inequality must be of equal length (see INVALID\_LENGTH exception).

#### Exceptions

INVALID\_SUBSTRING is raised if the value of L is negative or the value of U exceeds LENGTH - 1.

INVALID\_LENGTH is raised when strings of unequal length are compared for equality or inequality.

The INVALID\_SIZE exception is raised if for E !! F, E.LENGTH + F.LENGTH exceeds the maximum permitted charstring length, (E.LENGTH).MAX.

### 3.7.5. Assignment

#### Syntax

[3-143] charstring-assignment ::= char-target := charstring-expression;

[3-144] char-target ::= variable  
                  | variable closed-range

#### Semantics

The value of the expression becomes the value of the target.

If a target is of the form V[L:U], the positions of V's value indicated by the values of L through U are replaced with the value of the expression; the value of the remaining positions is not changed. If the lower bound of the closed-range is zero, replacement begins with the first character of V.

The LENGTH attribute of a target of the form V[L:U] is U-L+1, except that if the value of L exceeds the value of U, the value of LENGTH is zero.

Constraints

The LENGTH and CHARSET attributes must be the same for the expression and target.

The target must be of type class CHAR\_STR.

Exceptions

The SIZE exception is raised if the expression and target are not the same length.

The INVALID\_SUBSTRING exception is raised if the lower bound of the closed-range is negative or the upper bound is greater than LENGTH - 1. (Note that this permits closed ranges of the form [0:=1].)

### 3.7.6. Predefined Functions

#### 3.7.6.1. DUP

##### Syntax

[3-145] charstring-dup-function ::= DUP (charstring-expression,  
integer-expression)

##### Semantics

DUP returns a charstring result whose length is the value of the integer-expression. Each character of the result is the value of the charstring-expression. The CHARSET attribute of the result is the same as that of the charstring-expression.

##### Constraints

The first argument must be a charstring of length one.

The second argument must not be negative or exceed the maximum permitted charstring length, ((CHAR\_STR).LENGTH).MAX.

##### Exceptions

The SIZE exception is raised if the value of the second argument exceeds the maximum permitted charstring length.

The INVALID\_LENGTH exception is raised if the value of the second argument is negative.

#### 3.7.6.2. BLANKS

##### Syntax

[3-146] charstring-blanks-function ::= BLANKS (integer-expression)

##### Semantics

BLANKS(N) is equivalent to DUP(' ', N). This function is used in record packing and to explicitly pad strings in assignment. Although DUP is more general, BLANKS is more convenient.

Examples

```
1. VAR S1: ASCII_STR(N);
   VAR S2: ASCII_STR(1);
   ...
   S1[0:0] := S2;
   S2 := S1[M:M];
```

The first assignment illustrates an assignment to the first character of S1. In the second example, the assignment is valid if M IN [0:N-1] is TRUE. Depending on the declaration of M, run-time checks of M's range may or may not be needed.)

```
2. VAR S1: ASCII_STR(4);
   VAR S2: ASCII_STR(6);
   ...
   S1 := S2 [0:3];
   S2 := S1 !! BLANKS (2);
```

(This shows how length adjustments must be done explicitly in assignments.)

### 3.8. ARRAYS

Arrays are composite objects whose elements all have the same type (as opposed to records (see 3.9), whose elements may have different types). In addition, individual array elements (as well as the arrays themselves) have the properties of a variable, i.e., their value can be accessed, assigned, and used as INOUT parameters of routines.

There are no predefined infix or prefix operators for arrays nor are there predefined functions that accept array arguments. Subscription is the only predefined operation.

#### 3.8.1. Attributes and Type Specification

##### Syntax

[3-147] array-spec	::= ARRAY [array-bound,...] [REP array-reptype] OF type-spec
[3-148] array-bound	::= integer-expression : integer-expression   enumerated-expression : enumerated-expression
[3-149] array-attr	::= DIMENSIONS   BOUNDS [(index) [. {MIN   MAX}]]   EXTENT [(index)]   ELEMENT_TYPE
[3-150] index	::= integer-expression

##### Semantics

An array-spec of the form ARRAY [L0:U0, L1:U1, ..., Ln:Un] OF T defines the DIMENSIONS, BOUNDS, EXTENT, and ELEMENT\_TYPE attributes of the array type. If Li and Ui are enumerated-expressions, they are implicitly converted to a fixed point integer value by invoking TO\_FIXED (Li, 0, 2, STD) and TO\_FIXED (Ui, 0, 2, STD). (Hence, when enumerated types are used as array indices, they are always treated as integers. This means enumerated types with non-standard representations or that are unordered cannot be used as array indices unless a user-defined TO\_FIXED function has been created for the type, since

TO\_FIXED is not automatically defined for such types.) [See J3.8-1.]

The value and meaning of the array-spec attributes are as follows:

- . DIMENSIONS -- the value of this attribute is n+1, the number of array-bounds in the array-spec. The type of the DIMENSIONS attribute is INTEGER [1:100] (i.e., no more than 100 dimensions can be used). [See J3.8-2.]
- . BOUNDS -- this attribute is a one-dimensional array defining the lower and upper bound values for each dimension. If MaxPrec is the maximum precision of the Li and Ui values, then the BOUNDS array is composed of records of the following type:

```
TYPE BOUNDS_DEF (MAX_P: FIXED_PRECISIONS) =
  RECORD
    VAR MIN, MAX: FIXED (0, 2, MAX_P);
  END RECORD;
```

The type of BOUNDS is ARRAY [0:n] OF BOUNDS\_DEF(MaxPrec). The value of BOUNDS(1).MIN is the value of TO\_FIXED(Li, 0, 2, MaxPrec). The value of BOUNDS(1).MAX is the value of TO\_FIXED(Ui, 0, 2, Maxprec). [See J3.8-3.]

- . EXTENT -- this attribute has the type ARRAY [0:n] OF FIXED (0, 2, MaxPrec). The value of EXTENT(i) is BOUNDS(i).MAX - BOUNDS(i).MIN + 1, i.e., it is the number of elements in the ith dimension of the array being declared. [See J3.8-4.]
- . ELEMENT\_TYPE -- the value of ELEMENT\_TYPE is the type of the array's elements, i.e., the value of ELEMENT\_TYPE is T in ARRAY ... OF T. The value of ELEMENT\_TYPE is needed to describe when arrays are assignment compatible as well as other abstract properties of an array.

#### Examples

1. VAR A: ARRAY [0:99] OF INTEGER [0:10000];

(This declares A to be a one-dimensional array containing 100 integer-valued elements in the range 0 to 10,000. A.DIMENSIONS = 1, A.BOUNDS(0).MIN = 0, A.BOUNDS(0).MAX = 99, A.EXTENT(0) = 100, and A.ELEMENT\_TYPE is INTEGER [0:10000].

If the value of the second expression in an array-bound is less than the value of the first expression, the array-bound has an EXTENT value of zero. [See J3.8-5.]

#### Constraints

The type classes of the enumerated-expressions in an array-bound must be identical.

The type-spec in an array-spec must not itself be an array-spec, i.e.,  
ARRAY ... OF ARRAY ... OF T is forbidden. [See J3.8-6.]

The radix of the integer-expressions in an array-bound must be identical.  
[See J3.8-1.]

#### Exceptions

If the number of bits required to represent the array type is larger than  
the maximum supported value, the ARRAY\_TOO\_BIG exception is raised. (The  
maximum value is discussed in Section 3.8.3.)

### 3.8.2. Array Constructors

#### Purpose

An array constructor is used to create an array whose elements all have defined values.

#### Syntax

[3-151] array-constructor ::= (non-array-expression,...)  
| (array-constructor,...)

[3-152] non-array-expression ::= expression

#### Semantics

The DIMENSIONS attribute has the value 1 for an array-constructor of the form  $(T, \dots)$  where  $T$  is a non-array-expression. The value of EXTENT(0) in this case is  $N$ , the number of  $T$ 's enclosed in parentheses. The BOUNDS(0).MIN and BOUNDS(0).MAX attribute values are 0 and  $N-1$ , respectively. Successive non-array-expressions are associated with successive index values of the rightmost (i.e., highest) dimension of an array.

Consider an array-constructor  $L$  of the form  $(T, \dots)$  where each  $T$  is an array-constructor. This form defines an array with one dimension more than  $T$ ; the first  $T$  is the value of the first element of the first dimension of  $L$ , i.e.,  $L$  has the form of an array of arrays. More precisely,  $L$  has one more dimension than  $T$ , i.e.,  $L.DIMENSIONS$  has the value  $T.DIMENSIONS + 1$ ,  $L.EXTENT(0)$  equals  $N$ , the number of  $T$ 's, and for  $I$  in  $[1:L.DIMENSIONS-1]$ ,  $L.EXTENT(I)$  has the value of  $T.EXTENT(I-1)$  and  $L.BOUNDS(I)$  has the value of  $T.BOUNDS(I-1)$ .  $L.BOUNDS(0).MIN = 0$  and  $L.BOUNDS(0).MAX = N-1$ .

#### Examples

1.  $((1,2,3,4),$   
 $(5,6,7,8),$   
 $(9,10,11,12))$

(This constructor represents a  $3 \times 4$  array, i.e., an array with  $DIMENSIONS = 2$ ,  $EXTENT(0) = 3$ ,  $EXTENT(1) = 4$ , the bounds of the first dimensions are  $[0:2]$  and the bounds of the second are  $[0:3]$ . The ELEMENT\_TYPE of the constructor is

INTEGER [1:12] (see below). If the value of the constructor is assigned to A, the value of A(2,1) is 5.)

The ELEMENT\_TYPE of the array-constructor is the type of its non-array-expression components. If these components have range attributes, the MIN attribute of the ELEMENT\_TYPE is the minimum of the MIN attributes of each component and the MAX attribute is the maximum of the MAX attributes of each component. (This implies that if there is an expression component that invokes some function or operator, the range of ELEMENT\_TYPE is, for predefined functions and operators, the default range for the ELEMENT\_TYPE, see 3.1.4.1).

#### Examples

2. ((A, B<sup>2</sup>),  
(3, D+B),  
(E, F))

(This is a constructor composed of non-literal values. Its ELEMENT\_TYPE is INTEGER because 3 is of type INTEGER [3:3] and D+B must therefore be of type INTEGER with default range. The other attributes of this constructor are the same as for the previous example.) [See J3.8-8.]

#### Constraints

For an array-constructor (T,...), the type of each T must be the same, except that if T is a non-array-expression with range attributes (i.e., if T is a fixed-expression, float-expression, or enumerated-expression for predefined types), the range attributes of the expressions need not be identical. (Note that this implies array-constructor components of an array constructor must each have the same shape, i.e., the same dimensions, bounds, and extent values, i.e., a value must be assigned to every element of an array defined by the constructor.)

The type class of a non-array-expression must not be ARRAY.

### 3.8.3. Representation Specifications

#### Syntax

[3-153] array-reptype ::= ARRAY\_STD  
| ARRAY REP (packing-mode)

[3-154] array-rep-attr ::= SIZE  
| PACKING

#### Semantics

An array-reptype of the form ARRAY REP(P) defines the value of the PACKING attribute to be P and the value of the SIZE attribute implicitly.

- . PACKING -- the value of this attribute implies how array components are organized with respect to each other. The type of PACKING is UNORDERED (DENSE, PACKED, UNPACKED).

If the value of PACKING is DENSE, array elements are stored without regard to word boundaries in the object computer and successive elements are packed together without intervening filler bits.

If the value is PACKED, elements are stored so they can be accessed and assigned reasonably quickly, but not necessarily as quickly as if the array was UNPACKED. Moreover, elements are stored so they do not cross word boundaries unless an element is too long to fit in a single word, and if the element size is such that more than one element can be packed in a single word, this is done.

If the value of PACKING is UNPACKED, each element is stored as if each element were a simple-variable of the ELEMENT\_TYPE.

- . SIZE -- the value of this attribute is the number of bits occupied by the array. The number of bits is counted beginning with the first bit of the first array element and ending with the last bit of the last element. The value of SIZE is a function of the ELEMENT\_TYPE and its representation, the number of elements in the array, and the PACKING mode of the array.

The default representation for arrays is ARRAY\_STD, which is defined to be equivalent to ARRAY REP(UNPACKED). Only elements of ARRAY\_STD arrays can be used as inout parameters. [See J3.8-9.]

Examples

1. VAR X: ARRAY [-10:10] REP ARRAYREP(DENSE) OF  
INTEGER [0:63] REP FIXEDREP (6, UNSIGNED);

(This example declares an array of 21 elements. Each element is specified to be stored in 6 bits as an unsigned number. The packing of the array is DENSE. Hence the value of SIZE for this array is  $6 \cdot 21 = 126$  bits. If the object computer were an IBM 370, specifying PACKED for the array would probably store each element in successive bytes, but if the FIXEDREP specified a size of 9 bits per element, the elements would probably be stored in half-words of 16 bits if PACKED representation were specified for the 370.)

### 3.8.4. Operators and Expressions

#### Syntax

- [3-155] array-expression ::= expression
- [3-156] subscripted-variable ::= variable (subscript,...)
- [3-157] subscripted-constant ::= constant (subscript,...)
- [3-158] subscript ::= expression

#### Semantics

The type of the subscripted-variable or subscripted-constant is the ELEMENT\_TYPE of the variable or constant. (The variable or constant must have an array value; see Constraints.) The array element identified by a subscripted-variable or subscripted-constant depends on the value of the subscripts; each array element is identified by a unique combination of subscript values. [See J3.8-10.]

For a subscripted variable or constant of the form A(...,E,...), if E is the ith subscript and E is an enumerated-expression, its value is implicitly converted to an integer fixed point value by invoking TO\_FIXED(E, 0, 2, STD). If the type of E is not identical to the type of the corresponding array dimension (e.g., if the scales or precisions differ), the subscript value is (in principle) implicitly assigned to a variable of the dimension's type, thereby invoking whatever implicit range, scale, precision, etc. conversions are defined for such assignments.

#### Constraints

An array-expression must be of type class ARRAY.

For a subscripted-variable or constant of the form A(S0, S1, ..., Sn), the value of n+1 must equal A.DIMENSIONS and the type of the ith subscript, Si, must be assignment compatible with the type of the corresponding array dimension, i.e., it must be possible to assign the subscript value to a

variable of the dimension's type. (This implies that the subscript's value must be in the range [A.BOUNDS(1).MIN : A.BOUNDS(1).MAX].)

#### Exceptions

If the value of a subscript lies outside the range of permitted values for the corresponding dimension, the SUBSCRIPT\_RANGE exception is raised.

##### 3.8.4.1. Infix Operators

The only predefined infix operators for array values are equality and inequality:

OP	NAME	RESULT	EXCEPTIONS
=	equality	boolean	
<>	inequality	boolean	

The equality operator returns the result TRUE if all corresponding elements of the array operands are equal (i.e., if when corresponding elements are evaluated for equality, the result is TRUE for each element pair; this implies that if the array elements are of a user-defined type, the user-defined type's definition of equality is used in comparing corresponding array elements.) Otherwise, the result is FALSE.

The value of  $A <> B$  is the value of NOT ( $A = B$ ).

#### Constraints

The two operands must be of the ARRAY type class and have the same DIMENSIONS and EXTENT attributes. (The BOUNDS attributes may differ.)

#### Efficiency Considerations

If a user-defined equality operator is not free of side-effects, then all element pairs must be tested for equality, even if some pair is found to be unequal before all pairs have been compared. Of course, the equality operator is normally free of side-effects, so the comparison can be stopped as soon as one pair is found unequal.

### 3.8.5. Assignment

#### Syntax

[3-159] array-assignment        ::= variable := array-expression;

#### Semantics

The value of the array-expression becomes the value of the variable. The assignment is performed on an element-by-element basis, i.e., an element of the array-expression's value is assigned to the corresponding element of the variable.

#### Constraints

The ELEMENT\_TYPE of the variable and expression must be assignment compatible.

The DIMENSIONS and EXTENT attributes of the variable and expression must be identical. (The BOUNDS attributes are permitted to differ.) [See J3.8-11.]

The value of an element of the expression must be assignable to the corresponding element of the variable (e.g., the value of the element must be in the range permitted by the variable's ELEMENT\_TYPE).

#### Exceptions

If an exception is raised when assigning an element of the expression to an element of the variable, the value of the variable is not changed.

### 3.9. RECORD TYPES

A record is an object composed of a fixed number of components, each of a possibly different type. Each component is called a field.

Record variables can have three kinds of fields: variable fields, constant fields, and tag fields. Variable fields are those components whose values can change throughout the lifetime of the record variable. Constant fields are those components whose value is fixed when the record variable is created. In addition, the values of the constant fields form part of the type of the record object (e.g., two record objects that are identical except for the value of a constant field may not be assigned to one another).

Tag fields are used in variant records. A variant record is one in which a portion of the record can be specified to have alternate sets of fields. The value of a tag field determines which particular set of fields a record actually has. Furthermore, a record may have several variant parts, each having its own tag field.

Record specifications having only variable fields may be used directly in variable (or constant) declarations. Record specifications with constant fields or variant parts can only be defined using parameterized record-type-declarations. [See J3.9-1.]

#### 3.9.1. Attributes and Type Specification

##### Syntax

[3-160] record-spec ::= RECORD [field-declarations]...  
END RECORD

[3-161] field-declarations ::= [var-field-spec]... [variant-part]...

[3-162] var-field-spec ::= VAR field-name,... : type-spec ;

[3-163] field-name ::= identifier

```

[3-164] variant-part      ::= SELECT tag-name FROM
                           [variant-spec]...
                           END SELECT;

[3-165] variant-spec     ::= variant-name,... -->
                           [field-declarations]...
                           END;

[3-166] variant-name     ::= manifest-expression
                           | manifest-range-spec

[3-167] record-type-declaration ::= TYPE record-type-name
                           [(record-parameter,...)] =
                           record-type-definition;

[3-168] record-type-name   ::= identifier

[3-169] record-parameter   ::= tag-field-spec
                           | const-field-spec
                           | type-parameter

[3-170] tag-field-spec     ::= TAG tag-name,... : type-spec

[3-171] tag-name           ::= identifier

[3-172] const-field-spec    ::= field-name,... : type-spec

[3-173] type-parameter      ::= identifier,... : TYPE

[3-174] record-type-definition ::= record-spec

[3-175] record-type-reference ::= record-type-name [(record-argument,...)]

[3-176] record-argument      ::= expression
                           | ANY
                           | type-spec

[3-177] record-attr          ::= note: only record fields have attributes

```

Semantics

The attributes characterizing the abstract properties of a record object are determined by the fields comprising the record. The fields are of several kinds: 1) variable fields, whose value is not part of the object's type; 2) constant fields, whose value is part of the object's type, and which consequently can never be changed during the object's existence; and 3) tag fields whose value determines which variant is available for access and assignment; a tag field's value can only be changed by assigning to the record

variable as a whole; a tag field's type may only be integer, boolean, or an enumerated type. [See J3.9-2.]

Each of the var-field-spec declarations specifies one or more field-names that act as component variables (via a field-reference) if the record-spec is used in declaring a record variable. If the record-spec is used to declare a record constant, no assignment to the record constant or any of its fields is permitted.

Record types composed only of variable fields may be specified with record-specs. To specify record types that contain constant fields or tag fields (variant records), one must use a parameterized record-type-declaration to declare a record-type-name. This parameterized type-name can then be used to specify a particular record type. (See Section 3.12 for additional information about type-declarations.)

When a record-type-declaration is specified, its tag-field-spec and constant-field-spec record-parameters are considered to be additional fields of the record-type-definition, and consequently their values can be accessed using a field-reference form (see examples below). Such parameters are non-assignable fields of the record (i.e., they cannot be assigned to or used as an INOUT or OUT parameter). Moreover, except for TAG parameters, the values of such fields are constant. When the record-type-name is used in a record-type-reference (e.g., in declaring a record variable), the values of the record-arguments become the values of the constant and tag fields, and also become part of the type being declared. However, TYPE parameters are not additional fields of the record type. Instead, they may be used within the record-type-definition to specify the types of variable fields.

Examples

1. VAR DATE: RECORD

```
    VAR DAY: INTEGER [1:31];
    VAR MONTH: ENUM (JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP,
                      OCT, NOV, DEC);
    VAR YEAR: INTEGER [1900:2100];
END RECORD;
```

(This is an example of a variable DATE which is declared to be a record containing three variable fields, namely DAY, MONTH, and YEAR).

2. TYPE T (C: INTEGER) = RECORD

```
    VAR A: INTEGER;
    VAR B: BOOLEAN;
END RECORD;
```

```
VAR V: T(3);
```

(This is an example of a variable V which is declared to be a record containing two variable fields, A and B, and a constant field C whose value is 3. Unlike the previous example, a type-declaration must be used because the record has a constant field.)

3. TYPE R (C: INTEGER, D: TYPE) = RECORD

```
    VAR A: INTEGER;
    VAR B: D;
END RECORD;
```

```
VAR Y: R (3, BIT(5));
```

(This is an example of a variable Y which is declared to be a record containing two variable fields, A and B, and a constant field C. Note that D is not a constant field of the record because it is of the special type TYPE. In this example D is used to determine the type of field B. Thus the type of Y.B is BIT(5).)

A variant record-type-definition has alternate sets of fields depending on the value of a tag-name parameter. A variant-part of a record-type-definition defines which fields are accessible for each value of the tag-name. The list of variant-names in each variant-spec is the set of values of the tag-name for which the field-declarations in that particular variant-spec will be accessible. When a variant-name has the form of a manifest-range-spec, this is considered equivalent to listing each value in the specified range in order from lowest to highest (e.g., 1,2,3 --> is equivalent to [1:3] --> in a variant-spec). If all values of a tag-name are not specified explicitly as

values of variant-names, the unused values denote a null variant-spec. [See J3.9-3.]

When specifying a variant record type (e.g., by using a record-type-reference in a variable-declaration), the record-arguments for the tag parameters must be either manifest-expressions or the reserved word ANY. If a variable V is declared to be of type T(expr), where T is a variant record-type-name with a tag parameter, then V is of type T(e), where e is the value of expr and is a valid value for the type of the tag parameter. Variable V can only be assigned values of type T(e). If, however, V is declared to be of type T(ANY), then V may be assigned values of type T(X) where X can be any value of the tag's type. When the value most recently assigned to V is of type T(e), the value of V.t (where t is the tag-name) will be e. [See J3.9-4.]

For two record-specs to specify the same type, they must have the same field-names in the same order, the same variant-names in the same order, and the types of corresponding field-names must be identical. If a record-spec is used as a record-type-definition in a parameterized record-type-declaration, then the constant-field-specs and tag-field-specs are considered to be additional fields of the record-spec (following RECORD) and hence are part of the record type. In addition, when a record-type-name is used in a record-type-reference, the values of the record-arguments become part of the record type, and except for tag-name parameters whose actual value is specified as ANY, the value of these parameters cannot be changed by assignment or any other operation. (The values of these parameters are used in determining when a record variable and a record object are assignment-compatible.) [See J3.9-5.]

```

4.  TYPE STREAM (TAG DEV: ENUM (\DISPLAY, \TAPE, \DISK, \KEYBOARD)) = RECORD
    VAR UNIT: INTEGER [0:7];
    SELECT DEV FROM
        \DISPLAY -->
            VAR FIRST, LAST: DISPLAY_CONTROL_BLOCK; { user-defined type
            VAR HEIGHT: SCREEN_POSITION;           { user-defined type
            VAR N_LINES: INTEGER [0:(SCREEN_POSITION.LAST)//8];
            END;
        \TAPE, \DISK -->
            VAR FILE: FILE_HANDLE;                { user-defined type
            VAR POSITION: INTEGER [0:4095];
            VAR BUFFER: ARRAY [0:255] OF ASCII;
            END;
        \KEYBOARD -->
            VAR BUFFER: ASCII_STR(20);
            VAR BUF_FIRST, BUF_LAST: INTEGER [0:BUFFER.LENGTH];
            END;
    END SELECT;
    END RECORD;

    VAR X: STREAM (\KEYBOARD);
    VAR Y: STREAM (ANY);
    VAR Z: STREAM (\KEYBOARD);
    VAR A: STREAM (\TAPE);
    VAR B: STREAM (\DISK);

```

(This is an example of a variant record-type-declaration, STREAM. DEV is the tag whose value is used to determine which particular variant a record object contains. Field UNIT will occur in all STREAM records regardless of the value of the tag DEV. The occurrence of any other field in a particular STREAM record will depend on the value of DEV. Five variables, namely X, Y, Z, A and B, are declared to be STREAM records. Variables X, Z, A and B are declared with specific values of the tag field DEV and thus can only hold record objects with these specific values of DEV (i.e., X can only hold STREAM records with the \KEYBOARD variant.) Variable Y is declared with the special value ANY so that it may hold stream objects with any variant.

Of the five variables, only X and Z are of the same type (because the value of the tag field is part of the type). Variable Y, however, is assignment compatible with all four of the other variables, i.e., Y may be assigned the values of the other variables.

Note that X.BUFFER.LENGTH gives the length of BUFFER, i.e., 20. Note also that X.DEV = \KEYBOARD. It is intended as an implementation standard that the amount of space allocated for X be just enough to hold STREAM (\KEYBOARD) records, since X can only be assigned values of this type.

The value of Y.DEV is not yet defined in this example, since Y has not been assigned an initial value. If X is given an initial value and then X is assigned to Y, Y.DEV will have the value \KEYBOARD.)

```

5.  TYPE STRING(LEN: INTEGER [1:N]) = RECORD
    VAR LENGTH: INTEGER [1:LEN];
    VAR VALUE: ASCII_STR (LEN);
    END RECORD;

    VAR X: INTEGER [1:50] := 50;
    VAR STRING50: STRING(50);
    VAR STRING_X: STRING(X);
    VAR STRING_Y: STRING(STRING50.LEN);
    VAR STRING_Z: STRING(10);

```

(This is an example of a record-type-declaration (for a varying length character string) with a constant field, LEN. The variables STRING50, STRING\_X and STRING\_Y are all of the same type because the value of LEN (which is part of the type) is the same (i.e., 50). STRING\_Z, however, is not of the same type as any of the other variables because the value of its LEN field is 10. The reference STRING50.LEN refers to the LEN field (i.e., constant field) of the variable STRING50. It would be equally valid to write STRING50.LENGTH.LAST to get the upper bound of the LENGTH component of STRING50. Note that STRING\_X.LEN = STRING50.LEN = 50.)

```

6.  TYPE T(TAG X: ENUM (\RED, \GREEN)) = RECORD
    SELECT X FROM
        \RED -->
            VAR Y: INTEGER [1:100];
            END;
        \GREEN -->
            VAR Z: BOOLEAN;
            END;
    END SELECT;
    END RECORD;
TYPE T_RED = RECORD
    VAR Y: INTEGER [1:100];
    END RECORD;
VAR V_RED: T_RED;
VAR VT_RED: T(\RED);

```

(Note that V\_RED and VT\_RED do not have the same type, even though each contains a single variable field with field name Y and the type of Y is the same for both variables. The reason is that T(\RED) has a field T.X whose value is \RED, but T\_RED has no such field.)

A var-field-spec may refer to the attributes of a variable field-name declared earlier in the record-spec provided that every set of tag arguments that makes the later field-name accessible also makes the earlier field-name accessible. Record-parameter names are usable throughout the record-type-definition. [See J3.9-6.]

A record-spec defines a closed scope, i.e., no expression in a record-spec can refer to a variable declared outside the record-spec, and field-names declared within a record type may be identical to names declared outside the record-spec. (Note that saying variables are not known inside a record-spec implies that no function importing a variable can be invoked inside the record-spec.)

(If some attribute of a field, e.g., an array bound, is to depend on the value of some variable outside the record, then the variable must be passed in as an argument to a record-parameter that is used to specify the array bound. Note, however, that constants outside the record may be referenced directly without being passed in as arguments.) [See J3.9-7.]

#### Efficiency Considerations

Note that if a parameter of a type is given a manifest value when the type-name is used in a declaration, the value need not be stored as part of the representation.

For tag-fields as well as other parameter values, we suggest that as an implementation standard, manifest actual parameter values always be stored explicitly in a record's default representation. If a T(\RED) record is passed as an actual input parameter to a T(ANY) formal parameter, this implies that the T(\RED) record must be (implicitly) converted to the T(ANY) representation, i.e., a tag-field value must be explicitly provided. Since these implicit representation conversions can be expensive, and since a parameter field usually must be included in the representation when its value is not manifest, it is reasonable to include parameter fields in the default representation. However, we provide a representation directive (see Section 3.9.3) for specifying that a parameter field may be omitted from a record's representation when its value is specified as a manifest-expression. In this

way, a programmer can save the space required for these fields if this saving is considered significant.

#### Constraints

Tag fields can only be specified as formal parameters of a type-name, i.e., the tag-name in a variant-part must have been declared as a tag-field record-parameter. [See J3.9-8.]

The field-names (including tag field and constant field names) of a record-spec or record-type-declaration must all be distinct, except that field-names in different variant-specs of a variant-part may be the same. (See the previous STREAM example.)

Although a record may have more than one variant-part, two variant-parts must not discriminate on the same tag-name. [See J3.9-9.]

The values specified for variant-names must be values of the tag-name type being used.

The manifest-range-spec must specify a valid range of values of the tag-name type.

Each value of the tag-name must be specified not more than once as the value of a variant-name of a variant-part (e.g., overlapping manifest-ranges are forbidden as well as manifest-expressions yielding the same value for more than one variant-name of a variant-part.)

The type of a tag field must be either integer (i.e., FIXED with a scale of zero), boolean, or an enumerated type. [See J3.9-10.]

In a record-type-reference, the record-argument corresponding to a tag-field parameter must be a manifest-expression (of the parameter's type) or the reserved-word ANY.

In a record-type-declaration, the attributes of the type-specs of the record-parameters must be manifest.

2/15/78

Record

A record-type-reference must include argument values for all the record parameters even when the record has variant-parts and all the record parameters are not used in the specific variant being selected. [See J3.9-11.]

### 3.9.2. Record Constructors

#### Purpose

Record constructors are used to create record objects with initialized values for all fields of the record.

#### Syntax

```
[3-178] record-constructor ::= [record-type-specifier]
                  ([var-field-init,...])

[3-179] record-type-specifier ::= [type-spec]$$

[3-180] var-field-init      ::= field-name: expression
```

#### Semantics

A record object is created with the type specified by the type-spec. Constant and tag fields are given the corresponding values of the type-spec arguments. Each variable field is assigned the value of the expression in the corresponding field-init.

Note that in general, a record-constructor will be ambiguous with respect to its type if the record-type-specifier is omitted. Hence, the record-type-specifier may be omitted only in the following syntactic positions: the initializing expression of a constant or variable declaration, and the source expression of a record-assignment (but see Constraints below). In these cases, the record type is that of the constant or variable.

#### Examples

```
1. VAR Y: STREAM (ANY) := [STREAM(\KEYBOARD)]$$ (UNIT: 5,
                                              BUFFER: BLANKS(20),
                                              BUF_FIRST: 0,
                                              BUF_LAST: 0);
```

(This example declares a variable Y to be of type STREAM with ANY variant allowed. Y is initialized with a record constructor to contain an object with the \KEYBOARD variant. In this case the record-type-specifier is required in the record-constructor because the type of Y is not completely specified (i.e., it contains ANY).)

```
2. VAR X: STREAM (\KEYBOARD) := (UNIT: 5,
                                BUFFER: BLANKS(20),
                                BUF_FIRST: 0,
                                BUF_LAST: 0);
```

(In this example, a record-type-specifier is not required in the record-constructor because the type of X is completely specified.)

#### Constraints

The type-spec must specify a record type. Furthermore, it must be a specific record type, i.e., arguments for tag formal parameters must be manifest-expressions and not ANY. The record-type-specifier may be omitted only if the target constant or variable is declared to be a specific record type, i.e., is not of type T(...,ANY,...), where T is a type-name.

The type-spec must include argument values for all the record parameters, even when constructing a variant record and all the record parameters are not used in the specific variant being constructed.

If the record-type-name has a tag parameter, the only field-names permitted in the field-inits are the field-names valid (accessible) for the specified variant, i.e., the field-names common to all variants and the field-names associated with the specified variant-spec.

All accessible field-names in the record being created must be given initial values by the constructor. [See J3.9-12.]

The value of the expression in a var-field-init must be assignment compatible with the type of the field-name. If not, an appropriate exception is raised.

The values of the arguments of the type-spec must be assignment compatible with the types of the corresponding parameters. If not, an appropriate exception is raised.

The expressions and arguments of a constructor must be interference-free, i.e., the order of evaluation of the expressions and arguments must not be

Record

2/15/78

potentially able to affect their values. (The interference-free property is defined in Section 4.3.4).

Exceptions

If a field-init expression raises an exception or if the value of an argument or expression is not assignment compatible with the field being defined, an appropriate exception will be raised. If a constructor is potentially able to raise more than one exception, it is undefined which exception is actually raised.

### 3.9.3. Representation Specification

#### Purpose

A representation specification for a record is used to specify the ordering of the fields of the record, how tightly packed the fields are, and the presence of padding fields whose values cannot be accessed (but which might be used later in the evolution of a program). [See J3.9-13.]

#### Syntax

[3-181] record-reptype	::= RECORD_REP ([common-packing-spec] [variant-packing]...)   RECORD_STD
[3-182] common-packing-spec	::= packing-spec,...
[3-183] packing-spec	::= [packing-mode] [(element-spec,...)]
[3-184] element-spec	::= field-name   tag-name   filler-spec   VARIANTS (tag-name)   OMIT ({field-name : tag-name})
[3-185] filler-spec	::= FILLER (manifest-integer-expression)   ZEROS (manifest-integer-expression)   BLANKS (manifest-integer-expression)
[3-186] variant-packing	::= SELECT tag-name FROM [variant-packing-spec]... [else-packing-spec] END SELECT;
[3-187] variant-packing-spec	::= variant-name,... --> [packing-spec,...] END;
[3-188] else-packing-spec	::= ELSE --> [packing-mode] END;
[3-189] record-rep-attr	::= SIZE

#### Semantics

A packing-mode must yield a value of the enumerated type PACKING, namely, DENSE, PACKED, or UNPACKED. These values have their usual meanings, i.e., DENSE means the specified record fields are packed adjacent to each other without respect to word boundaries and beginning with the first available bit.

UNPACKED means the fields are packed in the same manner as variables of the field's type would be packed. PACKED means fields are stored such that fields do not extend across word boundaries unless the field is too long to fit in a single word and that otherwise, the field is stored in a position that is a reasonable compromise between space occupied and access ease. If a packing-mode is omitted, the default is UNPACKED.

If a packing-spec contains a list of element-specs, the fields specified in the list are stored in the order in which they appear in the list and with the packing strategy indicated by the packing-mode. A field-name may be either a variable field-name or a constant field-name (i.e., a formal parameter name). A tag-name denotes the tag field, not the corresponding variant-part fields.

If a common-packing-spec or variant-packing-spec contains several packing-specs, the order of these packing-specs is used to determine the order of the fields (see example 3 below).

A common-packing-spec or variant-packing-spec without any element-specs is equivalent to a packing-spec in which all the field-names accessible to the packing-spec are listed in an order determined by the translator. The field-names accessible to the common-packing-spec are the constant and tag parameter names and the variable field-names not declared in variant-parts. The accessible field-names of a variant-packing-spec are the variable field-names declared for the specified variant-name in a variant-spec (not including field-names of nested variant-parts).

#### Examples

##### 1. RECORD REP (DENSE)

(This indicates that the entire record is to be densely packed. The translator determines the order of the fields.)

## 2. RECORD REP (DENSE (A, B, C))

(This indicates that the fields A, B, and C are to be densely packed in the order A, B, C. DENSE(C,B,A) would specify the reverse order.)

## 3. RECORD REP (DENSE (A, B), PACKED (C, D), E)

(This indicates that fields C and D are to be stored in PACKED mode, but A and B are to be densely packed. If the field length of A and B is such that both fields can be packed within a single word but not in a single half word, and fields C and D can be packed in half words, then A and B will be packed in the first word of the record, C will occupy the first half of the second word and D, the second half. E will occupy the third word. Note that even though A and B are densely packed, there may be unused space in the first word because C cannot be packed in the first word and still satisfy the PACKED packing strategy (assuming the object machine can efficiently access half words). If A and B together occupy less than a full word, then A and B occupy the left part of the first word in this case. (This is an arbitrary choice, but it is best for all translators to make the same arbitrary choice. Note that E needs to be mentioned so all fields are included in the packing specification.)

The VARIANTS form in an element-spec is used to indicate the relative positioning of a particular variant-part of a record. All variant-specs of the variant-part identified by the tag-name are located together at the point indicated by the VARIANTS directive. The packing-mode preceding the VARIANTS element-spec is ignored since the packing for the variant-part is specified in a variant-packing.

The OMIT directive indicates that a constant or tag field (i.e., a formal parameter) of the record type is to be omitted from the record representation if the actual parameter value is a manifest-expression. If the actual value is not manifest or is ANY, then the field is included in the representation at the position of the OMIT directive.

The filler-spec permits padding fields to be specified. The length of a don't care field (in bits) is indicated by FILLER (N). FILLER is used when it doesn't matter what values are stored in the field, so the translator can assume complete control over the values. (For example, if a FILLER field is specified appropriately, a translator might be able to use full word store and

fetch instructions to access partial word values, rather than generate instructions just to access the field.)

The form ZEROS(N) indicates a padding field consisting of N zero bits.

The form BLANKS(N) indicates a padding field consisting of a character string of N blank characters. The representation of the blank character is determined by the current default character set (see Section 3.4.2). The packing of the string is the packing-mode of the packing-spec, e.g., PACKED(A, BLANKS(5), B) where A and B are charstring fields indicates that field A is separated from field B by a string of 5 ASCII blanks, stored in PACKED representation form.

The variant-packing form provides a means of individually specifying, for each tag-name value, how the fields of a variant-spec of a variant-part are to be packed. The else-packing-spec applies to all variant-part tag values for which an explicit variant-packing-spec has not been provided. Otherwise, variants for which no explicit packing-spec is provided have default representation, i.e., UNPACKED.

The default representation of record types is RECORD\_STD, which is defined to be equivalent to RECORD REP (UNPACKED).

The value of the SIZE attribute is of type INTEGER and specifies the number of bits occupied by a record value. There are no other attributes of records as a whole, although each field has attributes.

#### Constraints

If any packing-spec in a record-retype contains an element-spec, then the positions of all fields and variant-parts in the record must be explicitly specified.

No field or variant-part may be positioned more than once in a record-retype.

The field-names and tag-names in a record-reptype must be field-names and tag-names of the record type whose representation is being specified.

If field-names or tag-names are specified in a variant-packing-spec then they must be accessible for each tag value specified by the variant-names.

The variant-names in a variant-packing-spec must be values of the tag-name type.

The list of variant-names in a particular variant-packing-spec must have exactly the same values as the list of variant-names in one variant-spec of the variant-part whose representation is being specified.

Filler-specs are permitted only when DENSE packing mode is specified.

The argument of FILLER must not be negative.

Examples

```

4.  TYPE STREAM (TAG DEV: ENUM (\DISPLAY, \TAPE, \DISK, \KEYBOARD)) = RECORD
    VAR UNIT: INTEGER [0:7];
    SELECT DEV FROM
        \DISPLAY -->
            VAR FIRST, LAST: DISPLAY_CONTROL_BLOCK; % user-defined type
            VAR HEIGHT: SCREEN_POSITION; % user-defined type
            VAR N_LINES: INTEGER [0:(SCREEN_POSITION.LAST)//8];
            END;
        \TAPE, \DISK -->
            VAR FILE: FILE_HANDLE; % user-defined type
            VAR POSITION: INTEGER [0:4095];
            VAR BUFFER: ARRAY [0:255] OF ASCII;
            END;
        \KEYBOARD -->
            VAR BUFFER: ASCII_STR(20);
            VAR BUF_FIRST, BUF_LAST: INTEGER [0:BUFFER.LENGTH];
            END;
    END SELECT;
END RECORD REP RECORDREP (
    PACKED (UNIT, DEV, VARIANTS (DEV))
    SELECT DEV FROM
        \DISPLAY -->
            DENSE (FIRST, N_LINES, HEIGHT, LAST)
            END;
        \TAPE, \DISK -->
            DENSE(POSITION, FILE), PACKED(BUFFER)
            END;
        \KEYBOARD -->
            UNPACKED (BUF_FIRST, BUF_LAST, BUFFER)
            END;
    END SELECT;);


```

(This example is an extension of the type-definition of STREAM with an explicit representation specification. The first line of the representation specification, i.e., PACKED (UNIT, DEV, VARIANTS (DEV)), is the common-packing-spec which gives the overall layout of the entire record. It states that the record is to be PACKED with the UNIT field first, followed by the tag field DEV, followed by the variant portion of the record of which DEV is the tag. The remaining portion of the specification gives the explicit representation for each of the alternative variants.)

### 3.9.4. Operators and Expressions

#### Syntax

- [3-190] record-expression ::= expression
- [3-191] field-reference ::= variable . field-id
- [3-192] field-constant ::= constant . field-id
- [3-193] field-id ::= field-name  
| tag-name

#### Semantics

A field-reference or field-constant selects a field of the record variable or constant. The field chosen is identified by the field-id. The type of the resulting variable or constant is the type of the selected field.

If the field-id is a tag-name, then a field-reference or field-constant yields the current value of the tag field. For record variables of type T(ANY), the tag field value may be changed only by assigning an entire record to the variable.

A field-constant is a constant. A field-reference is a variable if the field-id names a variable field, a constant if the field-id names a constant field, and a value if the field-id names a tag field.

A field is said to be accessible at a particular point in a program if its name may legally be used as a field-id in a field-reference or field-constant at that point (see Constraints below which limit the accessibility of fields of variant-parts).

#### Constraints

A record-expression must be of type class RECORD.

In a field-reference or field-constant, the variable or constant must be of a record type and the field-id must be a field-name or tag-name of the record type.

If a variable or constant of a field-reference or field-constant is of type T(...,ANY,...), then the field-id may be a field-name of a variable field that is declared in a variant-spec only when the field-reference or field-constant occurs in a corresponding case-element of a case-stmt that discriminates on the tag field of the record value (see section 4.5.5.2), i.e., the variant-part fields of a T(...,ANY,...) record object are not accessible except within a discriminating case statement.

If a variable or constant of a field-reference or field-constant is of a specific variant type, say T(...,e,...) then the field-id may be a field-name of a variable field that is declared in a variant-spec only if the variant-spec has a variant-name with value e, i.e., the other variant-spec fields of the variant-part are not accessible.

#### Exceptions

A field-reference whose variable is of type T(ANY) and whose field-id is the tag-name raises the UNINITIALIZED\_VARIABLE exception if the tag field has not been initialized (by assigning to the whole variable).

### 3.9.4.1. Infix Operators

The only predefined infix operators for record values are equality and inequality:

OP	NAME	RESULT	EXCEPTIONS
=	equality	boolean	
<>	inequality	boolean	

The equality operator returns the result TRUE if all corresponding accessible fields of the record operands are equal (i.e., if when corresponding fields are evaluated for equality, the result is TRUE for each pair; this implies that if the fields are of a user-defined type, the user-defined type's definition of equality is used in comparing corresponding fields). Otherwise, the result is FALSE.

The value of  $A <> B$  is the value of NOT ( $A = B$ ).

#### Efficiency Considerations

If a user-defined equality operator is not free of side-effects, then all field pairs must be tested for equality, even if some pair is found to be unequal before all pairs have been compared. Of course, the equality operator is normally free of side-effects, so the comparison can be stopped as soon as one pair is found unequal.

#### Constraints

The two operands must be of the same record type (see 3.9.1).

An operand cannot have been declared to be of type  $T(\dots, \text{ANY}, \dots)$ . It must be of a nonvariant record type or have been declared to be of a specific variant type, i.e.,  $T(\dots, \text{expr}, \dots)$ . [See J3.9-14.]

#### Examples

1. VAR X,Y: STREAM (ANY) := ...;  
 VAR W,Z: STREAM (\DISK) := ...;

(The comparisons  $X = Y$  and  $X = W$  are prohibited because  $X$  is of type STREAM (ANY). However,  $W = Z$  is allowed because both operands are declared to be of a specific variant type, namely STREAM (\DISK). The fields compared are the

common fields (DEV,UNIT) and the specific variant fields (FILE, POSITION, BUFFER). (Of course, the DEV field comparison is manifestly TRUE and so is skipped.)

### 3.9.5. Assignment

#### Syntax

[3-194] record-assignment        ::= variable := record-expression ;

#### Semantics

The assignment operation for nonvariant record variables and specific variant record variables (i.e., those with no ANY tag arguments) assigns the accessible VAR fields of the expression to the corresponding fields of the variable, using the assignment semantics of the component field types.

Assignment to a variant record variable declared with ANY as the value of one or more of its tag field parameters assigns the values of the corresponding expression value's tag fields to the variable's tag fields. The values of the other fields are assigned as specified for specific variant record variables.

If the expression and variable representations are not identical, the expression's representation is first converted to the variable's representation, and then the assignment is performed. A warning is issued if this conversion occurs. The warning can be suppressed by expressing the conversion explicitly with an explicit-rep-converter.

If an exception is raised while assigning a record value (e.g., the RANGE exception), the value of the variable is unchanged. [See J3.9-15.]

#### Constraints

The type of the record-expression and the variable must be the same (see Sec. 3.9.1) except that a record-expression with a particular value of a tag field may be assigned to a variable whose declaration has ANY specified as the value of that tag field. (It should be noted that, as mentioned in Sec.

2/15/78

Record

3.9.1, two record objects containing constant fields are of the same type if and only if the value of the constant fields are the same).

### 3.10. POINTER TYPES

The purpose of pointers is to make use of dynamically allocated storage. The use of such storage permits the construction of objects (e.g., lists) whose number of elements can vary during program execution. In addition, the lifetime of the objects is not controlled by procedure activation; instead, the objects are explicitly created by the programmer and continue to exist until the program can no longer refer to them.

A value of a pointer is a reference to a variable of some other type; the type of this variable is specified in the pointer type-spec. Only dynamically allocated variables can be referred to by pointer values. [See J3.10-1.]

#### 3.10.1. Attributes and Type Specification

##### Syntax

- [3-195] pointer-spec                    ::= PTR (type-spec)  
[3-196] pointer-attr                    ::= ELEMENT\_TYPE

##### Semantics

A pointer-spec of the form PTR(T) defines the value of the ELEMENT\_TYPE attribute to be T. The ELEMENT\_TYPE attribute shows what type of variable the pointer value can refer to.

##### Examples

1.    TYPE NODE = RECORD  
          VAR VALUE: INTEGER;  
          VAR NEXT: PTR(NODE);  
      END RECORD;  
      VAR LIST: PTR(NODE) := NIL;

(The variable LIST is a pointer variable. The type NODE can be used to represent a linked list.)

### 3.10.2. Literals and Constructors

#### Purpose

The pointer literal, NIL, represents the pointer that does not point to any variable. All other pointer values are created by using a pointer-constructor.

#### Syntax

- [3-197] pointer-literal ::= NIL
- [3-198] pointer-constructor ::= [PTR\$] NEW (type-spec)

#### Semantics

The purpose of the pointer-constructor is to provide a reference to an uninitialized variable in dynamic storage. The value of the constructor is a pointer value suitable for accessing the variable. The type-spec gives the type of the pointed-to variable, so the type of the pointer value is PTR(type-spec).

#### Constraints

The PTR\$ prefix is required except when the constructor is the initial value expression in a variable or constant declaration or the source expression in an assignment statement.

#### Exceptions

The SPACE\_EXHAUSTED exception is raised if no space is available for allocating the requested type even after garbage collection.

### 3.10.3. Representation Specification

Only one built-in representation, PTR\_STD, is provided for pointers.

#### Syntax

- [3-199] pointer-reptype ::= PTR\_STD
- [3-200] pointer-rep-attr ::= SIZE

Semantics

The value of the SIZE attribute is the number of bits required to represent a pointer value.

### 3.10.4. Operators and Expressions

Syntax

[3-201] pointer-expression ::= expression

Constraints

A pointer-expression must be an expression yielding a value in some pointer type.

#### 3.10.4.1. Infix Operators

Semantics

Infix operators that can be applied to pointer values are:

OP	NAME	RESULT	EXCEPTIONS
=	Equality	boolean	
<>	Inequality	boolean	

Two pointer values are equal if they refer to the same variable.

If one of the operands is the literal NIL, it is automatically considered to be of the type of the other operand. NIL = NIL is always TRUE.

Constraints

The ELEMENT\_TYPE type class of the pointer operands must be the same.

#### 3.10.4.2. Other Operators

Purpose

There is a single operator for pointer values that performs dereferencing.

OP	NAME	RESULT	EXCEPTIONS
e	dereference type-spec	INVALID REF	

Semantics

The dereferencing operator, `e`, returns the variable referred to by the pointer value. The variable can now be used like any other variable, i.e., it may appear on the left-hand side of an assignment, or be passed as a parameter to a procedure or function.

If the pointer-expression evaluates to NIL, the exception NULLREF is raised.

Dereferencing must always be done explicitly.

Examples

```
1. VAR X: PTR(NODE) := NEW(NODE);
   X@ := (VALUE: 3, NEXT: NIL);
   X@.VALUE := 4;
```

Here `X@` produces a record variable with a component named VALUE which may be assigned to.

### 3.10.5. Assignment

#### Purpose

Assignment of pointers is different from ordinary assignment since it may introduce sharing, i.e., cause two variables to refer to the same dynamically allocated variable.

#### Syntax

[3-202] pointer-assignment ::= variable := pointer-expression;

#### Semantics

The pointer value resulting from evaluating the pointer-expression is stored in the variable.

#### Examples

1. VAR Y: PTR(NODE) := X;

(Here the NODE declarations given earlier are being used. After the assignment, X and Y both refer to the same node in a linked list.)

#### Constraints

The variable must be of pointer type and the ELEMENT\_TYPE of the expression must be identical with the ELEMENT\_TYPE of the variable.

#### Exceptions

The TYPE\_MISMATCH exception is raised if the type of the expression and variable do not match.

### 3.11. SEMAPHORES

#### Purpose

Semaphores are used for synchronization and mutual exclusion. The REQUEST and RELEASE statements (Section 4.6.6 and 4.6.7) are used for these purposes.

#### 3.11.1. Attributes and Type Specification

##### Syntax

- [3-203] semaphore-spec ::= SEMAPHORE (use-mode)
- [3-204] use-mode ::= manifest-enumerated-expression
- [3-205] semaphore-attr ::= USAGE

##### Semantics

A semaphore is an object with a value component containing a non-negative integer value and a list of path activations blocked on the semaphore.

A semaphore-spec of the form SEMAPHORE(U) defines the value of the USAGE attribute to be U. The type of USAGE is UNORDERED(SYNCH, REGION). If USAGE has the value SYNCH, the semaphore is to be used for synchronization; if USAGE is REGION, the semaphore is to be used for defining critical regions (mutual exclusion). The value of the USAGE attribute only affects program behavior if a parallel path is abnormally terminated (see Section 4.6.1), namely, when termination occurs, all REGION semaphores on which the terminated path activation had successfully performed a REQUEST operation without having performed a subsequent RELEASE are automatically RELEASEd. [See J3.11-1.]

#### 3.11.2. Constructors

##### Syntax

- [3-206] semaphore-constructor ::= [SEMAPHORE\$] NEW (integer-expression)

Semantics

A semaphore-constructor returns a semaphore with its value component initialized to the integer-expression value and an empty list of blocked path activations. It is intended that semaphore-constructors be used only in initializing semaphore variables. Its use in other contexts produces well-defined results that are not likely to be useful.

Constraints

The SEMAPHORE\$ prefix is required except when the constructor is the initial value expression in a variable or constant declaration or the source expression in an assignment statement.

The value of the integer-expression must be non-negative.

Exceptions

The RANGE exception is raised if the value of the integer-expression is negative.

### 3.11.3. Representation Specifications

Syntax

[3-207] semaphore-reptype ::= SEMAPHORE\_STD

[3-208] semaphore-rep-attr ::= SIZE

Semantics

The value of the SIZE attribute is implementation-defined and is the number of bits used in the representation of a semaphore.

### 3.11.4. Operators and Expressions

Syntax

[3-209] semaphore-expression ::= expression

Semantics

There are no infix or prefix operators predefined for semaphores. In particular, equality is not predefined.

Constraints

A semaphore-expression must yield a SEMAPHORE value.

### 3.11.5. Assignment

Syntax

[3-210] semaphore-assignment ::= variable := semaphore-expression;

Semantics

Assignment of semaphores is defined in order to permit their initialization. However, assignment is defined to assign only the value component of the semaphore-expression to the variable and to assign an empty list of blocked path activations to the variable (i.e., the list of blocked path activations of the semaphore-expression is not assigned to the variable.) Assignment to semaphore variables, except for the purpose of initialization, can produce unexpected (but well-defined) effects, and is thus not recommended.

### 3.11.6. Predefined Routines

The only predefined routines are REQUEST and RELEASE. These are defined in Sections 4.6.6 and 4.6.7.

### 3.12. USER-DEFINED TYPES

#### Purpose

Users will define types for several purposes.

- . to define abbreviations for types, e.g., INTEGER was defined to be an abbreviation for FIXED(0,2,STD);
- . to define new data types and operations on them, e.g., matrices and matrix arithmetic operations. Such new types will be referred to as abstract data types.
- . to declare record types.

The declaration and use of records is described in Section 3.9. In section 3.12.1, we discuss the use of type-declarations to provide type abbreviations; the use of type-declarations to create abstract data types is introduced in 3.12.5 and is discussed further in the sections describing modules (Section 4.4) and Generic Definitions (Section 5).

The definition and use of types is based on the syntactic form "type-spec" (e.g., "FLOAT (12)", "CHARSTR (15,ASCII) REP CHAR\_STRREP(DENSE)"). Type-specs were introduced in Section 3.1.1; their definition and use is further explained here.

The basic method of defining new types is the type-declaration. A type-declaration, by itself, merely declares a new type-name to be an abbreviation for a type-spec. Thus, the operations and attributes associated with the type-name are those of the type-spec. For example, the type declaration:

```
TYPE ASCII_STR (N:INTEGER) = CHAR_STR(N,ASCII);
```

defines a new type ASCII\_STR. A declaration of the form:

```
VAR X ASCII_STR (12);
```

is considered to be an abbreviation for the declaration:

```
VAR X CHAR_STR (12,ASCII);
```

However, when a type-declaration is written inside a module and the type-name is exported from the module, then outside the module the type-name stands for

a new abstract type class. This means the type-spec (i.e., the definition of the abstract type class) and its associated operations and attributes are not known outside the module, but only within it. For example, if the following type declaration appears within a module:

```
TYPE VAR_CHAR_STR (MAX_LENGTH: INTEGER, CHAR_SET: TYPE) =
  RECORD
    VAR CUR_LENGTH: INTEGER;
    VAR STRING: CHAR_STR (MAX_LENGTH, CHAR_SET);
  END RECORD;
```

then within that module, variables declared to be of type "VAR\_CHAR\_STR" are merely viewed as records with a CUR\_LENGTH component and a STRING component. If a variable is declared outside the module to be of type "VAR\_CHAR\_STR" then its "internal structure" is not accessible; it is viewed as an atomic entity. Outside the module which defines it, the only operations which can be performed on an abstract data type are those defined in and exported from that module, as later examples will show.

A reptype-spec is used to specify the representation of a predefined type and hence of a variable, constant, or parameter. Reptype-specs are used in the REP part of a type-spec. A reptype-declaration allows the programmer to declare a reptype-name to be an abbreviation for a reptype-spec.

### 3.12.1. Type Declarations

#### Purpose

A type-declaration declares a type-name to be an abbreviation for (equivalent to) a type-spec. The type-name may be parameterized, in which case it stands for a set of types whose member types are distinguished by the actual parameter values supplied in a type-spec that uses the type-name.

#### Syntax

```
[3-211] type-declaration ::= TYPE type-name [(type-formal,...)]
                           .type-definition;
                           | record-type-declaration
```

- [3-212] type-name ::= identifier
- [3-213] type-formal ::= identifier,... : {type-spec | TYPE}  
| TAG tag-name,... : type-spec
- [3-214] type-definition ::= type-spec

#### Semantics

A type-declaration associates a type-name and a set of formal parameters with a type-definition. When the type-name is used (e.g., to declare a variable to be of that type), arguments corresponding to the type-formals must be provided. These arguments are evaluated and substituted for the formals in the type-definition. Type-formals are input parameters and hence are constants within the type-definition. In the ASCII\_STR example above the type-name is "ASCII\_STR", the type-formal is "N:INTEGER", the type-definition is "CHAR\_STR(N,ASCII)" and the value to be substituted for the type-formal (in the declaration of S) is "12".

The new type-name introduced by the type-declaration is simply an abbreviation for the type-definition. This means that when the type-name is used in a type-spec, the effect is equivalent to direct use of the type definition (with values of the actuals used in place of the formals). The operations and attributes available for use are those provided by the underlying type class, i.e., the type class named in the type-definition either directly or indirectly (in the case where the type-definition uses a type-name declared in another type-declaration). Thus the underlying type class of the type-definition becomes the type class of the new type-name. [See J3.12-1.]

A type-declaration is a closed-scope, i.e., no expression in a type-declaration can refer to a variable declared outside the type-declaration. The scope of the type-formals is the type-definition. Constants and functions (that don't import variables) are imported implicitly into a type-declaration.

The meaning of such names is determined according to the usual lexical scope rules at the point where the type-declaration is written, not where the declared type-name is used in type-specs. [See J3.12-2.]

#### Examples

1. TYPE COLOR = ENUM(\RED, \BLUE, \GREEN, \VIOLET);
2. TYPE INTEGER = FIXED(0, 2, STD);
3. TYPE FIXED\_BIN (S: INTEGER [-1000:1000]) = FIXED (S, 2, STD);
4. TYPE ARRAY100 (T:TYPE) = ARRAY [1:100] OF T;  
      · · ·

#### Constraints

Recursive and circular type-declarations are forbidden except when declaring pointers (see Section 3.10). That is, type-declarations such as

```
TYPE T = RECORD ... VAR X: PTR(T); ... END RECORD;
```

are permitted while type-declarations such as

```
TYPE T = T;
TYPE T = ARRAY[1:2] OF T;
TYPE T = RECORD ... VAR X:T;... END RECORD;
```

are forbidden.

A TAG formal identifier may only be used in the type-definition as an argument to another type-spec's TAG formal or as a tag-name in a variant-part of a record-spec.

Type-arguments (see Section 3.12.2) of type-specs used in declaring type-formals must be manifest. For example,

```
TYPE T (A: ARRAY [0:N] OF INTEGER) = ... ;
```

is illegal if N is not a manifest-expression.

### 3.12.2. Type Specifications

#### Purpose

Type specifications are used to define the type of a variable, constant, parameter, or type-name.

Type-specs were discussed briefly in Section 3.1.1.4, where type-specs for predefined types were introduced. Type-specs for these types have been described in the 3.x.1 sections. In this section, we discuss type-references, i.e., type-specs whose type-ids are user defined. For convenience, we reproduce the definition of type-spec here.

```
type-spec      ::= predefined-type [REP retype-spec]
                  | array-spec
                  | type-reference [REP retype-spec]
```

#### Syntax

[3-215] type-reference	::= type-instantiation [range-spec]   record-type-reference
[3-216] type-instantiation	::= type-id [(type-argument,...)]
[3-217] type-id	::= abstract-type-name   abbrev-type-name
[3-218] abstract-type-name	::= type-name
[3-219] abbrev-type-name	::= type-name
[3-220] type-argument	::= type-spec   expression   ANY

#### Semantics

A type-instantiation specifies a type, i.e., a particular member of a type class. If the type-id of a type-instantiation is exported from a module and is being used outside the module, then the type-id is an abstract-type-name and denotes a type class that is distinct from its underlying definition. The type-arguments specify the values of the abstract type's attributes, and hence a particular abstract type of the class. The underlying definition is hidden. The operations are those exported along with the abstract-type-name.

Abstract types are further discussed in sections 3.12.5, 4.4, and 5.

If the type-id of the type-instantiation is being used inside a module in which it is declared, (or if it is not declared in a module at all), then the type-id is an abbrev-type-name and the type-instantiation is an abbreviation for the type specified by the type-definition of the type-declaration (after substituting type-arguments for type-formals, as discussed in 3.12.1). Thus, the type class, attributes, representation, and operations of the type-instantiation are those of the underlying type-definition. In the ASCII\_STR example above, the type-definition is "CHAR\_STR (N,ASCII)" and the type-instantiation is "ASCII\_STR (12)" which is an abbreviation for "CHAR\_STR (12,ASCII)".

If a range-spec is present in a type-reference, then it overrides the default range attributes (MIN and MAX) of the type. Similarly, if a REP clause is present in a type-spec, then it overrides the default representation for the type-instantiation.

#### Examples

1. VAR X: COLOR [\RED:\BLUE];

2. VAR I: INTEGER [1:10];

(This is equivalent to VAR I: FIXED (0,2,STD) [1:10].)

3. VAR Y: FIXED\_BIN(10) [0:1] REP FIXEDREP(10,UNSIGNED);

(This declares Y to be of type FIXED(10,2,STD) with a range from 0 through 1 - 2-10. Y is given a non-standard representation of 10 bits.)

4. VAR A: ARRAY100(INTEGER);

(This is equivalent to VAR A: ARRAY [1:100] OF INTEGER;)

#### Constraints

The type-arguments provided in a type-reference must match in number, type, and position with the type-formals. Furthermore, if a formal is used in

a position requiring a manifest-expression (e.g., the scale of a fixed point type), then the actual must be a manifest-expression. The argument ANY may be used only if the formal is a TAG formal.

A range-spec may be given in a type-reference only if the type-instantiation is an abbreviation for a member of type-class FIXED, FLOAT or ENUM, and the expressions in the range-spec must satisfy the requirements for these type classes as discussed in 3.2.1, 3.3.1 and 3.4.1.

A REP clause may be given in a type-spec only if the type-instantiation is a member of (or an abbreviation for a member of) one of the pre-defined type classes. The retype-spec must satisfy the constraints of the type class as defined in Sections 3.x.3. [See J3.12-3.]

### 3.12.3. Representation Declarations

#### Purpose

A retype-declaration declares a retype-name to be an abbreviation for (equivalent to) a retype-spec. The retype-name may be parameterized, in which case it stands for a set of representations whose members are distinguished by the actual parameter values supplied in a retype-spec that uses the retype-name. [See J3.12-4.]

#### Syntax

- [3-221] retype-declaration ::= REPTYPE retype-name [(type-formal,...)]  
[OF base-formal-id] = retype-definition;
- [3-222] retype-name ::= identifier
- [3-223] base-formal-id ::= identifier
- [3-224] retype-definition ::= retype-spec

#### Semantics

A retype-declaration associates a retype-name and a set of formal parameters with a retype-definition. When the retype-name is used (e.g., to declare a variable to be of that representation), values for the type-formals must be provided. These values are substituted for the formals in the retype-definition. The base-formal-id is also a formal parameter, of type TYPE. When the retype-name is used in a type-spec, the base-formal-id is replaced with the type of the type-reference used in the type-spec.

#### Examples

1. REPTYPE FRACTION OF T = FIXED\_REP (T.SCALE, UNSIGNED);  
VAR Y: FIXED\_BIN (10) [0:1] REP FRACTION;

(This declares Y to be of type FIXED(10, 2, STD) with a range from 0 through 1 - 2-10. The representation is FIXED\_REP (10,UNSIGNED), which is a nonstandard representation of 10 bits with no sign bit.)

The type-formals are input parameters and hence are constants within the retype-definition (as the above example with T.SCALE shows).

The attributes (both abstract and representational) of the base-formal-id may be queried (3.1.1.1) within the retype-definition. The values of such queries are the values of the corresponding attributes of the type-reference supplied as an actual parameter for the base-formal-id.

As with type-declarations, the new retype-name introduced by a retype-declaration is an abbreviation for the retype-definition (with formals replaced by the values of actuals).

A retype-declaration is a closed-scope. i.e., no expression in a retype-declaration can refer to a variable declared outside the retype-declaration. The scope of the type-formals and the base-formal-id is the retype-definition. Constants and functions (that don't import variables) are imported implicitly into a retype-declaration. The meaning of such names is determined according to the usual lexical scope rules at the point where the retype-declaration is written, not where the declared retype-name is used in retype-specs.

#### Constraints

Recursive and circular retype-declarations are forbidden. (For example, REPTYPE R = R; is circular and forbidden.)

Type-arguments of type-specs used in declaring type formals must be manifest.

### 3.12.4. Representation Specifications

#### Purpose

Reptype-specs permit programmers to specify non-standard representations for variables, constants, and types. Typically such specifications affect only the size and packing of stored values. The form of reptype-spec was given in Section 3.1.3 and is repeated here for convenience.

```
reptype-spec      ::= reptype-reference  
                   | predefined-reptype
```

#### Syntax

[3-225] reptype-reference ::= reptype-name [(type-argument,...)]

#### Semantics

Predefined-reptypes have been discussed in previous 3.x.3 sections. In this section we discuss user defined reptype-references. Note that predefined-reptypes have the same syntactic form and semantics as reptype-references.

When a reptype-spec is given in the REP part of a type-spec in a declaration, each of the names being declared is given the representation specified by the reptype-spec, i.e., the representation properties of the type specified by the type-reference part of a type-spec (preceding the REP part) are replaced with the properties specified in the reptype-spec.

A reptype-reference specifies a representation for a type, and is an abbreviation for the representation specified by the reptype-definition of the reptype-declaration that declares the reptype-name. The type-arguments of the reptype-reference are substituted for the type-formals of the reptype-declaration as discussed in section 3.12.3.

#### Constraints

The type-arguments provided in a reptype-reference must match in number, type, and position with the type-formals. Furthermore, if a formal is used in

a position requiring a manifest-expression (e.g., the size argument of a fixed-reptype), then the actual must be a manifest-expression.

The representation specified by a reptype-spec used in the REP part of a type-spec must satisfy the constraints of the type specified by the type-spec. The constraints for the predefined type classes are described in Sections 3.x.3.

#### Examples

1. REPTYPE NOSIGN (S: INTEGER [0:1000]) = FIXEDREP (S,UNSIGNED);  
VAR W: INTEGER [0:255] REP NOSIGN (8);

(This declares W to be an integer variable in the range 0 through 255. W has a nonstandard representation of 8 bits with no sign bit.)

### 3.12.5. Abstract Types

#### Syntax

- [3-226] defined-attr                ::= type-formal-id
- [3-227] type-formal-id              ::= identifier
- [3-228] defined-rep-attr            ::= SIZE

#### Semantics

Abstract types are defined by providing a module that implements the values of the type and a set of primitive operations to manipulate the values. The module also defines the attributes of the type. Abstract types are discussed further in the Sections describing modules (Section 4.4) and Generic Definitions (Section 5). Here we give some general properties of abstract types.

Within the module, the abstract type is defined by giving a type-declaration. If module M exports type T (i.e., makes T available for use outside the module), inside of M there will be a type-declaration of T in the usual form. This type-declaration serves to define the representation of values of the abstract type and the attributes of the abstract type. However, the actual representation chosen (i.e., the type-definition in the type-declaration) is visible only within the module, and not to programs outside the module that use the abstract type. (The type-definition is available to the translator when compiling programs that use the abstract type, and is used, e.g., to allocate space for declared variables.)

The type-declaration in the module may have parameters (type-formals). If it does have parameters, then whenever a variable of the abstract type is declared, actual values must be provided (in the type-arguments of a type-spec) for those parameters. Furthermore, the formal names of the parameters become the names of the attributes (defined-attr) of the abstract type, the

types of these attributes are the types given in the type-formals, and the values of these attributes are the values of the type-arguments. These attribute names may be used in attribute-queries (section 3.1.1.1) outside of the module. If the type-declaration has no parameters, then no parameter actuals are needed (in type-specs) for variable declaration, and the type has no attributes.

When the type-declaration has parameters, the module is defining an abstract type class (rather than a single abstract type). In this case, many of the operations will be generic (see Section 5).

If the type-definition in the type-declaration in the module is an enumerated type, then the enumerated-literals are exported from the module along with the type, and furthermore, the exported abstract type is considered to be an enumerated type. The exported type will be an ordered enumerated type only if the four ordering operations ( $<$ ,  $>$ ,  $\leq$ ,  $\geq$ ) and the SUCC and PRED functions have been explicitly provided for the abstract type. (Note that these operations are automatically provided for ENUM enumerated types within the module, but must be explicitly provided for the abstract type if they are to be available outside the module.) Values of the abstract type can be used as array indices outside the module only if TO\_FIXED is provided as well. [See J3.12-5.]

The only automatically defined representation attribute for abstract types is SIZE. Its value is that of the SIZE attribute of the underlying representation.

Examples

```

1. MODULE M EXPORTS (STACK);
    TYPE STACK (ELEMENT_TYPE: TYPE,
                 LENGTH: INTEGER [1:FIXED_MAX(0,2,STD)]) =
    RECORD
        VAR ELEMENTS: ARRAY [1:LENGTH] OF ELEMENT_TYPE;
        VAR TOP: INTEGER [0:LENGTH];
    END RECORD;
    ...
END MODULE M;

```

(Stack is a type class containing stacks of different sizes and element types. It has two attributes, LENGTH and ELEMENT\_TYPE, of types INTEGER [1:FIXED\_MAX(0,2,STD)] and TYPE, respectively. An example variable declaration is

```
VAR S: STACK(INTEGER,100);
```

This example is discussed further in Section 5.)

```

2. MODULE CHARSET EXPORTS (BCD);
    TYPE BCD = UNORDERED (
        \\'A\', \\'B\', \\'C\', \\'D\', \\'E\', \\'F\', \\'G\', \\'H\',
        \\'I\', \\'J\', \\'K\', \\'L\', \\'M\', \\'N\', \\'O\',
        \\'P\', \\'Q\', \\'R\', \\'S\', \\'T\', \\'U\', \\'V\', \\'W\', \\'X\', \\'Y\', \\'Z\')
    REP ENUM_VALREP ($SIZE% 6, VALUES (
        \\'A\' AS OCT '61', \\'B\' AS OCT '62', \\'C\' AS OCT '63',
        \\'D\' AS OCT '64', \\'E\' AS OCT '65', \\'F\' AS OCT '66',
        \\'G\' AS OCT '67', \\'H\' AS OCT '70', \\'I\' AS OCT '71',
        \\'J\' AS OCT '41', \\'K\' AS OCT '42', \\'L\' AS OCT '43',
        \\'M\' AS OCT '44', \\'N\' AS OCT '45', \\'O\' AS OCT '46',
        \\'P\' AS OCT '47', \\'Q\' AS OCT '50', \\'R\' AS OCT '51',
        \\'S\' AS OCT '22', \\'T\' AS OCT '23', \\'U\' AS OCT '24',
        \\'V\' AS OCT '25', \\'W\' AS OCT '26', \\'X\' AS OCT '27',
        \\'Y\' AS OCT '30', \\'Z\' AS OCT '31'));

    INLINE FUNCTION REORDER (OLD_CODE: BIT(6)) RETURNS (BIT(6));
        RETURN (OLD_CODE XOR OCT '60');
    END FUNCTION REORDER;

    INLINE FUNCTION INT_OF (ARG: BCD) RETURNS (INTEGER);
        RETURN (TO_FIXED(REORDER(TO_BIT(ARG)), 0));
    END FUNCTION INT_OF;

    INLINE FUNCTION BCD$LESS_THAN (OP1,OP2: BCD) RETURNS (BOOLEAN);
        RETURN (INT_OF(OP1) < INT_OF(OP2));
    END FUNCTION BCD$LESS_THAN;

    INLINE FUNCTION BCD$LESS_EQUAL (OP1,OP2: BCD) RETURNS (BOOLEAN);
        RETURN (INT_OF(OP1) <= INT_OF(OP2));
    END FUNCTION BCD$LESS_EQUAL;

```

```

INLINE FUNCTION BCD$GREATER_THAN (OP1,OP2: BCD) RETURNS (BOOLEAN);
  RETURN (INT_OF(OP1) > INT_OF(OP2));
END FUNCTION BCD$GREATER_THAN;

INLINE FUNCTION BCD$GREATER_EQUAL (OP1, OP2: BCD) RETURNS (BOOLEAN);
  RETURN (INT_OF(OP1) >= INT_OF(OP2));
END FUNCTION BCD$GREATER_EQUAL;

FUNCTION BCD$SUCC(OP: BCD) RETURNS (BCD) SIGNALS (RANGE);
  SELECT OP FROM
    \\'I\' --> RETURN(\\'J\'); END;
    \\'R\' --> RETURN(\\'S\'); END;
    \\'Z\' --> SIGNAL RANGE; END;
    ELSE --> CONST NEXT: INTEGER := TO_FIXED(TO_BIT(OP), 0) + 1;
      CONST BITS: BIT(NEXT.SIZE) := TO_BIT(NEXT);
      RETURN [BCD]$(BITS[BITS.SIZE-BCD.SIZE : BITS.SIZE-1]);
        \$ extracts low order bits of integer value
        \$ remember that bitstrings are indexed from zero
    END;
  END SELECT;
END FUNCTION BCD$SUCC;

FUNCTION BCD$PRED (OP: BCD) RETURNS (BCD) SIGNALS (RANGE);
  SELECT OP FROM
    \\'J\' --> RETURN (\\'I\'); END;
    \\'S\' --> RETURN (\\'R\'); END;
    \\'A\' --> SIGNAL RANGE; END;
    ELSE --> CONST NEXT: INTEGER := TO_FIXED(TO_BIT(OP), 0) - 1;
      CONST BITS: BIT(NEXT.SIZE) := TO_BIT(NEXT);
      RETURN [BCD]$(BITS[BITS.SIZE-BCD.SIZE : BITS.SIZE-1]);
        \$ extracts low order bits of integer value
        \$ remember that bitstrings are indexed from zero
    END;
  END SELECT;
END FUNCTION BCD$PRED;

FUNCTION BCD$TO_FIXED (OP: BCD) RETURNS(INTEGER [0:25]);
  CONST INT_REP INTEGER [0:63] := TO_FIXED(TO_BIT(OP),0);
  SELECT INT_REP FROM
    [49:57] --> RETURN (INT_REP-49); \$ \\'A\' = OCT '61' = 49
    [33:41] --> RETURN (INT_REP-24); \$ \\'J\' = OCT '41' = 33
    [18:25] --> RETURN (INT_REP); \$ \\'S\' = OCT '22' = 18
  END SELECT;
END FUNCTION BCD$TO_FIXED;
END MODULE CHARSET;

```

(This is an example of the definition of a character set in which the representation is given explicitly. In particular, the bit representation order does not correspond to the lexical ordering (and consequently the collating sequence) of the characters. As a result the four ordering operators (<, <=, >, >=) and SUCC and PRED are explicitly provided. Exporting type BCD from the module exports all of its operators (i.e., those routines prefixed with BCD\$) and, because it is an enumerated type, all of its

2/15/78

## User-defined Types

enumerated-literals. Since all the necessary operations are provided, BCD is treated as an ordered enumerated type outside the module and values of the type can be used as array indices.

Notice that within the module, BCD is simply treated as an abbreviation.)

#### 4. PROGRAM STRUCTURE

The basic facilities provided in the language for structuring programs are segments, modules, and routines. A segment is used to define a separately-compilable program unit along with its interface to other segments. The interface specification is used to 1) explicitly state what names defined within the segment are available to other segments (this is called exporting) and 2) explicitly state what names defined in other segments are needed within this segment (this is called importing).

Exporting and importing names replaces what have been called external declarations in other languages, e.g., PL/I. Names of variables, constants, routines (procedures and functions), modules, and types can be defined in one segment and used in another.

Separately compiled program units are combined using these interface specifications. In fact, libraries can be built up using this mechanism. Exhibit 4-1 shows a system built up of separately-compiled program units and libraries.

A module is used for packaging definitions for the purpose of controlling access to them. Definitions within a module are only accessible outside the module when explicitly exported and then only in a restricted fashion. One main use of modules is for defining abstract data types, e.g., matrices and matrix operations.

Routines are portions of programs that may be invoked from many different places, possibly with different parameter values. Routines are either functions, which may be invoked within expressions to obtain a value, or procedures, which are invoked with procedure statements.

This section defines the various program structuring features of the language. The first sub-section defines the scope rules of the language.

These are the rules that determine the availability of access to named items within programs and when explicit import and export are required. Following this are sub-sections on segments, routines, and modules. Finally, other program structuring capabilities are dealt with, namely, control statements, parallel processing features, and libraries.

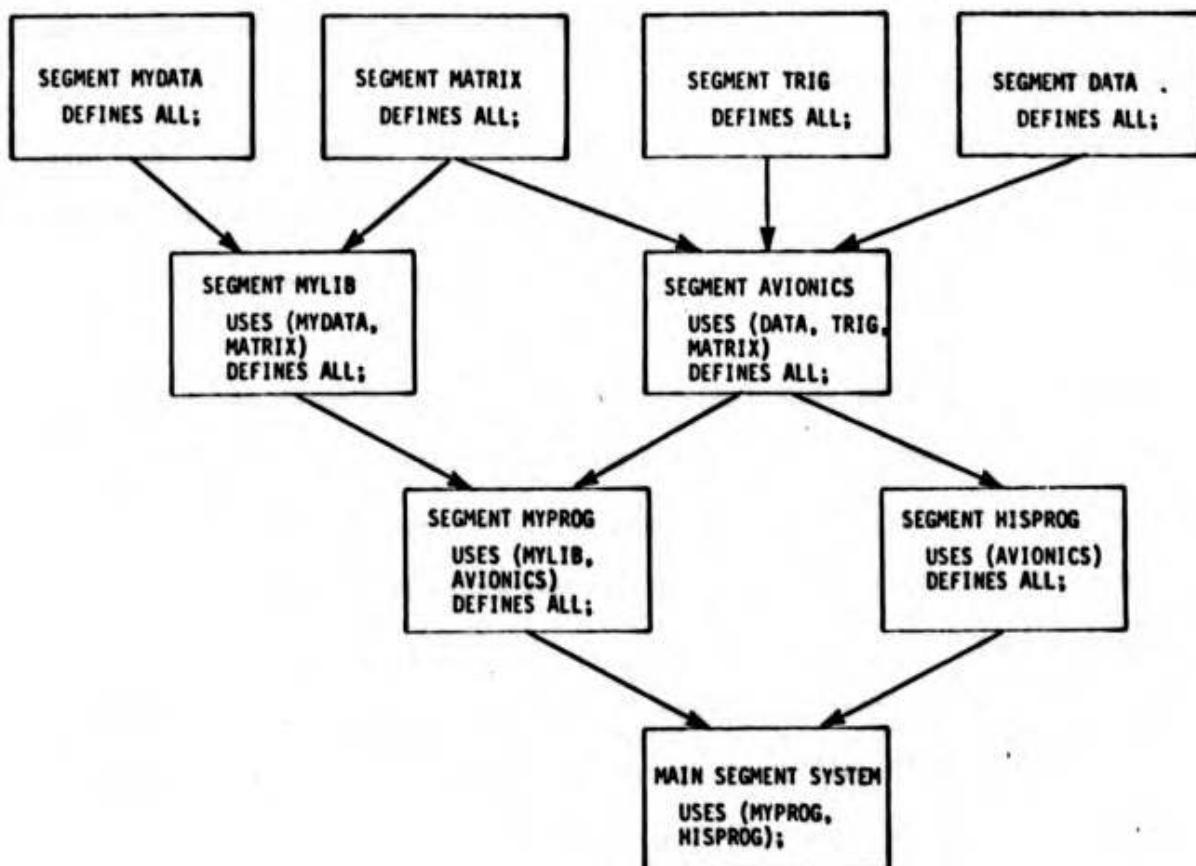


Exhibit 4-1

## 4.1. SCOPE RULES

Syntax

- [4-1] open-scope-body ::= [declaration-stmt]...  
[executable-stmt]...
- [4-2] closed-scope-body ::= [declaration-stmt]...  
[executable-stmt]...  
[routines-and-modules]...
- [4-3] routines-and-modules ::= routine-declaration  
| module-declaration

Semantics

A scope is a region of a program in which a name is known. Scopes may be nested within scopes. There are three kinds of scopes, open, closed and segment scopes.

Open scopes are blocks, parallel blocks, and the bodies of other control structures (see Sec. 4.5). They follow the customary scope rules, i.e., all names known in the immediately enclosing scope are automatically known unless redefined within the scope.

Closed scopes are routines (procedures and functions), parallel paths, and modules. The rules for closed scopes are very similar to those for open scopes except for variables and labels. Variables are treated specially because they are potentially alterable within the scope. A closed scope must explicitly specify which variables known in the immediately enclosing scope it intends to use directly or indirectly (by invoking a routine that uses the variable). Specifying such variables by name is called importing. A closed scope can only import from the immediately enclosing scope. Names of constants, modules, routines, and types need not be explicitly imported except under certain circumstances (see Importing below). [See J4.1-1.]

Labels are treated specially in that they may not be imported implicitly or explicitly into closed scopes. This is so goto statements cannot transfer

control out of routines or parallel paths. (Exceptions are used for this purpose; see Section 4.5.7.)

A segment scope is the outermost scope of a compilation unit (i.e., the body of a segment). Segments may use definitions from other segments but must explicitly import all such items, including routines, types, modules, constants, and variables. Since segments are separately-compilable, it is not possible to know what items are required from other compilation units unless they are named explicitly.

Names are either local or global to a scope. Local names are either names declared explicitly within the scope or names exported from a module declared within the scope (see Exporting below). Global names are either names known from an outer scope or names imported into a segment scope from another segment. Names local to a scope may not be redeclared within the scope. If a name global to a scope conflicts with a local name, the local name is considered to be a redeclaration of the name, and the global name (even if it was explicitly imported) is unknown within the scope.

Routine names are somewhat special in that multiple definitions (overloading) are permitted. Consequently, routines have both a name and a signature. The signature is the routine name appended with the types of all its formal parameters. The signature is needed to uniquely determine which routine is being invoked when a routine invocation occurs. The following rules apply to routines:

- . A routine name local to a scope may have additional routine declarations in the scope if the routine signatures are different. All such routines are known in the scope.
- . A routine name local to a scope can only be redeclared as a routine having a different signature (e.g., it is illegal to declare both a routine and a variable with the same name in a scope).

- . If a routine name is global to a scope, local declarations of routines with the same name are permitted. if the signatures of a local and global routine are the same, the global routine will not be known in the scope. If none of the local routines have the same signature as a global routine, the global routine will be known in the scope.
- . If a routine name is global to a scope, a local declaration of the name as other than a routine will make the global routine unknown (e.g., a local declaration of a variable will make all global routines with that name unknown in that scope).

Routine, module, and type names local to a scope do not have to be declared before they are used (i.e., they are known throughout the scope). Variable and constant names local to a scope, however, must be declared before they are used and their declarations are evaluated in the order in which they occur. Note that the syntax requires the placement of routine and modules declarations at the end of a scope, and that such declarations cannot appear in an open scope.

Named items within a scope become available for use (and, in the case of variable and constants, initialization occurs) when the scope is "entered" during execution. Modules are considered to be entered when their containing scope is entered. The local data of a module is initialized after the local data of the containing scope. In the case of segments, all segments linked together via USES and DEFINES clauses (see Section 4.2) are considered to be entered simultaneously when the MAIN segment is entered, subject to the constraint that the local data of a segment must be initialized before the local data of any other segment which imports that segment (i.e., names it in its USES clause). This ensures that all such data will be initialized before it can be referred to.

#### 4.1.1. Importing

Importing is used for two purposes. In a closed scope, all global variables used within the scope, either directly or indirectly (by invoking a routine that uses the variable), must be imported. Two forms are provided for importing such variables: the variable may be explicitly imported, or, in the case of a variable used only indirectly, the routine that uses the variable may be imported. Importing the routine is sufficient to permit it to be invoked in the scope, but does not make the variable directly accessible. A routine must be imported when the following conditions all hold: 1) the routine is exported from a module; 2) the variable the routine uses is an own variable of the module; and 3) the invoking scope is outside the module. Such an own variable is inaccessible outside the module. (The information that these own variables are imported into the routine is needed for aliasing and side-effect checking.) [See J4.1-2.]

The second use of importing is to support separate compilation of segments. Segment scopes must explicitly import all names that are needed from other segments (including constant, types, routines, etc.)

#### 4.1.2. Exporting

Modules and segments are special kinds of scopes in that they may make externally known (i.e., export) names defined within them. All such names must be explicitly exported.

In the case of a module, its routines, types, and modules (but not data) may be exported. The representations of types exported from a module are inaccessible outside the module. All names exported by a module are considered local to the scope containing the module.

In the case of a segment, names of routines, types, modules, and data local to the segment scope may be exported.

## 4.2. SEGMENT DECLARATION

Purpose

A segment declaration is used to define a separately-compilable program unit. Segments may contain declarations of data, routines, types, and modules. A main segment may, in addition, contain executable statements (these executable statements constitute the main program). Segments may import data, routines, modules, and types from other segments with a USES clause and may export such items with a DEFINES clause. The USES and DEFINES clauses are the mechanisms for combining separately-compiled portions of a program and for building libraries (see Section 4.7).

Syntax

- [4-4] segment-declaration ::= segment-header  
                              segment-body  
                              segment-trailer;
- [4-5] segment-header     ::= [MAIN] SEGMENT segment-name  
                             [uses-clause]  
                             [defines-clause];
- [4-6] uses-clause      ::= USES ({segment-name[item-list]},...)
- [4-7] item-list        ::= ([DATA : ([input-items][#inout-items])]  
                             [ROUTINES : (routine-name,...)]  
                             [TYPES : (type-name,...)]  
                             [MODULES : (module-name,...)])
- [4-8] input-items     ::= {simple-variable | simple-constant},...
- [4-9] inout-items     ::= simple-variable,...
- [4-10] routine-name   ::= proc-name  
                          | func-name
- [4-11] defines-clause ::= DEFINES {(name,...) | ALL}
- [4-12] name           ::= identifier
- [4-13] segment-body   ::= closed-scope-body
- [4-14] segment-trailer ::= END SEGMENT segment-name
- [4-15] segment-name   ::= identifier

Semantics

A segment declaration associates a name with a separately-compilable body of code. The segment-name becomes externally available so that the segment's data, etc. may be imported into other segments.

The uses-clause is used to name data, routines, types, and modules exported from other segments that are to be available within this segment. If a segment-name in the uses-clause is not followed by an item list, then all the names exported by that segment in its defines-clause will be available (exported variables will all be available inout). If a segment-name is followed by an item-list, then only those names included in the item-list will be explicitly available within this segment. The meaning of each type of item in the item-list is as follows:

- . A DATA item names variables and constants. Variables may be imported either for input or inout. Constants may only be imported for input.
- . A ROUTINES item names routines. A single routine-name may cause several routines to be imported if the segment from which the routines are being imported had several routine declarations for that name.
- . A TYPES item names types. Such a type may be either abstract (if it is exported from some module in the segment) or it may simply be a type-declaration appearing in the segment. If the type is abstract, then importing it imports all the type's operations as well; if the type is not abstract, then importing it provides the type-declaration.
- . A MODULES item names a module. All routines, abstract types, and abstract type operations exported by that module will be automatically imported.

Imported names are global, i.e., they may be redefined.

The defines-clause is used to make names defined within the segment externally available. If the keyword ALL is used, all names local to the segment scope and those imported into the segment with the uses-clause are exported. Otherwise, only those names listed are exported. Naming a routine exports all routines with that name. Naming an abstract type exports all the type's operations, and naming a module exports all routines, abstract types,

and abstract type operations exported by that module.

The segment containing the uses-clause, all segments named in its uses-clause, plus all segments named in their uses-clauses, etc. may be thought of as being merged together into a larger unit in which all declarations from all the segments are in the same scope. The defines-clause makes some of those names inaccessible to other segments and the uses-clause specifies which accessible names will be used, but all names are known to the translator.

[See J4.2-1.]

The MAIN attribute indicates that the executable-statements in the segment-body are the main program that is to be executed when this segment (possibly combined with other separately compiled segments) is executed. Execution proceeds from the first executable statement following normal flow of control rules.

Segments without the MAIN attribute are used to separately compile data, types, modules, and routines which may then be used in other segments.

Data local to the scope defined by a segment is considered to be statically allocated. (This includes own data of modules local to the segment and own data of modules nested in such modules. See Section 4.4.) Data local to routines declared in the segment, however, is not statically allocated, but is considered to be reallocated at each invocation of the routine.

#### Constraints

The segment-name in the segment-trailer must be the same as the segment-name in the segment-header.

Input-items may only appear in contexts in which constants are permitted within the body.

A name must only appear once explicitly in an uses-clause. (A name can be imported implicitly more than once, however, e.g., by importing routines

that both import the same global data.)

A segment that imports a variable for input may not import a routine that imports the same variable inout. [See J4.2-2.]

Only a segment with a MAIN attribute may contain executable-statements.

Only names that are local to the scope defined by the segment or imported into the segment may appear in the defines-clause.

Only one segment with a MAIN attribute may be combined with other segments (via uses-clauses) to form an executable program.

A segment S is not permitted to import from a segment T if T directly or indirectly imports from S (e.g., S cannot import from T if T imports from U and U imports from S).

### 4.3. ROUTINES

#### 4.3.1. Routine Declarations

A routine is either a procedure or a function. A procedure declaration is used to associate a name with a (possibly parameterized) portion of a program which may then be invoked with a procedure statement.

A function declaration is used to associate a name with a (possibly parameterized) portion of a program that returns a value. Such a function may then be invoked with a function invocation. [See J4.3-1.]

#### Syntax

- [4-16] routine-declaration ::= procedure-declaration  
| function-declaration  
| target-routine-declaration
- [4-17] procedure-declaration ::= procedure-header  
procedure-body  
procedure-trailer;
- [4-18] procedure-header ::= [INLINE] PROCEDURE proc-name  
proc-formal-parm-list  
[signals-clause]  
[imports-clause]  
[where-clause];
- [4-19] procedure-body ::= closed-scope-body
- [4-20] procedure-trailer ::= END PROCEDURE proc-name
- [4-21] proc-formal-parm-list ::= ([input-parameters] [#inout-parameters] [#output-parameters])
- [4-22] input-parameters ::= formal-parameter,...
- [4-23] inout-parameters ::= formal-parameter,...
- [4-24] output-parameters ::= formal-parameter,...
- [4-25] formal-parameter ::= formal-name,... :  
(type-spec | routine-spec | TYPE)
- [4-26] proc-name ::= [type-name\$] identifier
- [4-27] formal-name ::= identifier
- [4-28] imports-clause ::= IMPORTS item-list

- [4-29] signals-clause ::= SIGNALS (exception-name,...)
- [4-30] function-declaration ::= function-header  
                              function-body  
                              function-trailer;
- [4-31] function-header ::= [INLINE] FUNCTION func-name  
                              func-formal-parm-list  
                              returns-clause  
                              [signals-clause]  
                              [imports-clause]  
                              [where-clause];
- [4-32] function-body ::= closed-scope-body
- [4-33] function-trailer ::= END FUNCTION func-name
- [4-34] func-formal-parm-list ::= ([input-parameters])
- [4-35] returns-clause ::= RETURNS (type-spec)
- [4-36] func-name ::= [type-name\$] identifier

#### Semantics

A routine declaration associates a name with a body of code. The scope of the name is the scope containing the routine declaration.

The formal-parameters are declarations. The scope of the formal parameters is the scope defined by the routine excluding the formal-parm-list, (i.e., a reference to a formal-parameter may not occur in the type-spec of another formal-parameter). Note that the standard representation is assumed for formal parameter types unless a non-standard representation is explicitly specified.

Formal parameters to procedures may be of one of three classes: input, inout, or output. Formal parameters to functions may only be input. The class of the parameter determines how and when correspondence between the formal parameter and an argument is established (see routine invocation).

Formal-parameters whose type is a routine-spec or TYPE are generic parameters (see Section 5).

The returns-clause in a function-declaration declares the type of the value returned by the function.

The INLINE attribute in a routine declaration indicates that the body of the routine must be expanded at each invocation point. When such an invocation contains manifest constant arguments, conditional compilation of the routine body (where applicable) will occur.

The imports-clause is used to explicitly or implicitly (by naming a routine, module, or abstract type) name global data that can be accessed within the routine body (see constraints below, routine invocation, and scope rules). (The form of item-list was explained in the section on segments.)

When invoked, execution of the routine body proceeds from the first statement, following normal flow of control rules. Normal completion of the routine occurs when a return-statement is executed or when execution reaches the trailer.

The signals-clause declares what exceptions a routine (or parallel path) is permitted to raise to its invokers. The TERMINATED exception (which is raised when execution of a parallel path is terminated with a TERMINATE statement) is implicitly declared as an exception for every routine declaration.

Abnormal termination occurs when an exception is explicitly signaled by a signal-stmt (see Section 4.5.7).

#### Constraints

The name in the trailer must be the same as the name in the header.

A name must be listed only once in an imports-clause.

A recursive routine (one which directly or indirectly through a sequence of calls invokes itself) can only import statically allocated data (variables and constants). [See J4.3-2.]

Input formal parameters and input-items (in an imports-clause) may only appear in contexts in which constants are permitted within the body (i.e., they may not be assigned to or used as inout or output parameters).

A routine with the INLINE attribute must not be recursively invoked.

A routine that imports a variable for input must not import a routine that imports the same variable inout.

Execution of a routine may be terminated by raising an exception only if the exception name has been specified in the routine's signals-clause and the exception is raised by a signal-stmt local to the routine.

Attributes in type-specs of formal parameters must be manifest expressions (or generic parameters see Section 5).

A '#' must not be the last symbol in a formal-parm-list, e.g., (#) and (A:INTEGER#) are forbidden.

Attributes in the type-spec in the returns-clause of a function header must be expressions in which all the operands are constants or generic parameters (see Section 5) and all the functions are pure. A pure function is one which does not import any variables. [See J4.3-3.]

For additional constraints see Scope Rules, Side Effects, and Aliasing.

### 4.3.2. Routine Invocation

#### Purpose

A routine invocation is used to execute a routine with the specified parameters.

#### Syntax

- [4-37] procedure-stmt ::= proc-name proc-argument-list;
- [4-38] proc-argument-list ::= ([input-arguments] [#inout-arguments] [#output-arguments]))
- [4-39] input-arguments ::= {expression | routine-argument | type-spec},...
- [4-40] inout-arguments ::= variable,...
- [4-41] output-arguments ::= variable,...
- [4-42] function-invocation ::= func-name func-argument-list
- [4-43] func-argument-list ::= ([input-arguments])
- [4-44] return-stmt ::= RETURN [expression];

#### Semantics

A procedure-stmt or function-invocation initiates execution of the referenced routine after evaluating the arguments and establishing the appropriate correspondence between the arguments and formal parameters. (Note: When there exists more than one routine with the referenced name (i.e., the routine is overloaded), which routine to invoke must first be determined. How this is accomplished will be described at the end of this section.)

Correspondence between the arguments and the formal parameters is carried out as follows:

- . Each input argument's value is copied into the formal parameter after representational conversions, if required, are performed.
- . Each inout argument is passed by reference to the formal parameter. That is, the argument is bound to the formal parameter such that all references to the formal parameter have their effect on the argument.

- . Output arguments are bound to their formals when the procedure is invoked. If the procedure completes execution normally, the value of each output formal parameter is copied into the argument after representational conversions, if required, are performed. If the procedure terminates abnormally (i.e., because an exception is raised), the copy is not performed.

(Note: The order in which arguments are evaluated is not defined, and because of the side-effect rules, any order of evaluation will produce the same effect.)

Input arguments that are either routine-arguments or type-specs are generic arguments (see Section 5).

A function is completed when its body is terminated by executing a return-stmt. If execution of the body would be complete otherwise (i.e., if no return-stmt has been executed), no value exists to be returned, and so, the last executed statement raises the UNINITIALIZED\_VARIABLE exception. [See J4.3-4.]

If a function completes normally, the value returned by the function is the value of the expression in the return-stmt. This value becomes the value of the function invocation (for use in the expression containing the function invocation).

Execution of a procedure is completed when execution of its body is complete or when a return-stmt in the body is executed.

Execution of a return statement completes the innermost enclosing routine (or parallel path, see Section 4.6.1) after terminating all intervening open scopes.

Data declared locally in a routine become accessible when the routine is entered and remain in existence until the routine completes or terminates. Routines global to a parallel block are reentrant with regard to that block, i.e., if such a routine is invoked in several paths of the parallel block, the local data of the routine is not shared among the paths.

When a routine is invoked using an infix operator (e.g., A\*B), this is converted to a regular function-invocation using the operator names listed in Section 3.1 (e.g., the above example would be converted to MULTIPLY\_(A,B)).

When the routine to be invoked is overloaded, determination of which routine to invoke is accomplished as follows:

- The types of all the arguments are determined. The types are then combined with the routine name to construct the invocation signature (a definition of signature is contained in Section 4.1).
- All routines with the given name and the proper number of arguments that are known in the scope of the routine invocation are then considered as candidates for invocation. If any of the candidate routines are generic, their headers are instantiated, if possible, from the types of the arguments (see Section 5). Those that cannot be instantiated are discarded as candidates.
- If the invocation signature matches exactly one candidate routine signature, that is the routine to be invoked. If it matches more than one, the invocation is in error.
- If there is no exact match, all the candidates are reconsidered to determine if a match exists such that the types of the inout arguments match the types of the formals exactly, and the types of the input and output arguments are assignment compatible. If one match exists, that is the routine to be invoked. If no match or more than one match exists, the invocation is in error. Assignment compatible for an input argument means that the STORE-operation of the argument's type permits an assignment from the type of the actual to the formal argument type; for an output argument, the formal and actual are assignment compatible if the STORE-operation permits an assignment from the formal argument type to the type of the actual argument. Assignment compatibility is only possible if both the formal and the actual are in the same type class.

#### Exceptions

During argument evaluation, any exception may be raised that may be raised during expression evaluation. During argument correspondence, any exception that may be raised by assignment may be raised.

#### Constraints

In the case of inout arguments, the attributes of the arguments (e.g., range, scale, array bounds, precision) and the representation must match the attributes and representation of the formal parameters exactly.

In the case of input and output arguments, the attributes and representation of the arguments need not match the attributes of the formals exactly, but the arguments and formal parameters must be assignment compatible.

A return-statement must contain an expression if (and only if) it is in a function-body, and in this case the type of that expression must be assignment compatible with the type in the returns-clause in the function-header.

A '#' must not be the last symbol in an argument-list (e.g., (#) and (A#) are both illegal).

For additional constraints see Scope Rules, Side Effects, and Aliasing.

#### Efficiency Considerations

Although the basic semantics of input parameters state that the argument is copied into the formal parameter, it is desirable that when such a formal parameter is an array, record, bit string, or character string, the routine be compiled to accept the argument by reference. At each invocation of the routine, if the actual argument corresponding to such a formal is not also passed as an inout argument nor imported inout into the routine, then the argument may be passed by reference. If the actual argument is also passed as an inout argument or imported inout into the routine, then the argument must be copied at the point of invocation and the copy passed by reference.

#### 4.3.3. Aliasing

Aliasing is the ability to refer to the same simple variable (or its components) with two or more different names in the same scope. Aliasing is particularly undesirable when the value of the variable can be changed via one of its names. In particular, the rule to guarantee unique values of expressions, subscript lists, and arguments lists by ensuring interference-free function calls (see section on Side Effects), can only be enforced in the absence of this kind of aliasing. (Aliasing in which the variable can be read but not modified under any of its names is not prohibited.)

In order to prevent undesirable aliasing, the following constraints on routine invocation are needed:

- . A simple-variable or any of its components must not be used as more than one inout argument to a procedure. [See J4.3-5.]
- . A simple-variable or any of its components must not be used as more than one output argument to a procedure.
- . A simple-variable or any of its components must not be used as an inout argument to a procedure if it has been explicitly or implicitly imported inout or for input into that procedure. [See J4.3-6.]

#### Examples

```
1. PROCEDURE P (IN1, IN2: INTEGER # INOUT1, INOUT2: INTEGER
   # OUT1, OUT2: INTEGER)
      IMPORTS (DATA: (A, B # C, D));
      ...
END PROCEDURE P;
```

(By the aliasing rules, the following calls on P are illegal. M and D are assumed to be one-dimensional arrays. The variables involved in aliasing are underlined:

P (E, F # G, <u>G</u> # H, L);	\$ same inout variables
P (E, F # <u>M(I)</u> , <u>M(J)</u> # H, L);	% components of same variable
P (E, F # <u>A</u> , G # H, L);	% A is also imported
P (E, F # G, H # <u>L</u> , L);	% same output variables
P (E, F # <u>C</u> , G # H, L);	% C is also imported inout
P (E, F # <u>D(I)</u> , G # H, L);	% D is imported inout

The following calls, however, would be legal:

```
P (E, E # G, H # L, N);      % input parameters can be the same
P (E, F # E, H # L, N);      % E must be copied in this case
P (E, C # G, H # L, N);      % even though C is imported
P (E, F # G, H # E, L);      % E can be both input and output
P (M(I), M(J) # G, H # L, N); % even though A is imported also
P (A, E # G, H # L, N);
```

#### 4.3.4. Side Effects

The language does not define the order of evaluation of operands in an expression (except for certain Boolean expressions, see Section 3.5.4.1), arguments in a routine invocation, or subscripts in an array reference. It is, however, required that the value of expressions and the value of arguments and subscripts not be affected by the order of evaluation when pointers are not involved. This will only be the case if the kinds of side effects that functions can have are suitably restricted. Uses of functions having such restricted side effects will be called interference-free uses of the functions.

The basic rule is that any variable that is modified at some point in the evaluation of an expression, subscript-list or argument-list, cannot be accessed (read or modified) at any other point in the expression, subscript-list, or argument-list. (Note: A variable is considered to be modified during expression evaluation if and only if it is imported inout into an invoked function. A variable is considered to be accessed for reading if and only if it is named directly in the expression or is imported for input into an invoked function.) [See J4.3-7.]

In order to enforce this rule, the following constraint on function invocation is provided:

If a function imports a variable inout, an invocation of the function must not occur in an expression, subscript-list or argument-list that names the variable directly or that contains an invocation of a function that imports the variable for input or inout. (Note: This rule only works in the absence of aliasing, see section on Aliasing.)

Note that assignment statements are covered by this rule since the left-hand-side and right-hand-side of an assignment statement are actually both arguments to the STORE\_ routine. Note: Although the aliasing and side effect restrictions apply to all variables (including pointer variables), undesirable aliasing and hence non-interference-free side-effects can still occur when pointer variables are involved because distinct pointer variables are capable of pointing to the same object.

#### Examples

```

1. PROCEDURE P;
   VAR X,Y: INTEGER [1:10];

   FUNCTION F (A: INTEGER [1:10]) RETURNS (INTEGER[1:10])
      IMPORTS (DATA:(#X));
      ...
   END FUNCTION F;

   FUNCTION G (A: INTEGER[1:10]) RETURNS (INTEGER[1:10])
      IMPORTS (DATA: (#X));
      ...
   END FUNCTION G;

   FUNCTION H (A: INTEGER[1:10]) RETURNS (INTEGER[1:10])
      IMPORTS (DATA: (#Y));
      ...
   END FUNCTION H;

   PROCEDURE Q IMPORTS (ROUTINES: (F,G,H))
      VAR L,M,R: INTEGER[1:10];
      ...
      L := F(R) + H(R);
      M := G(R) + H(R);
      ...
   END PROCEDURE Q;
END PROCEDURE P;

```

(This example shows a procedure P which contains within it variables X and Y, procedure Q, and three functions F, G, and H.

Procedure Q, which invokes F, G, and H, explicitly imports them. Alternatively procedure Q could have imported variables X and Y, the data imported by F, G, and H.

Although functions F, G, and H all potentially have side effects (they import variables inout), the expressions within procedure Q,  $F(R) + H(R)$  and  $G(R) + H(R)$  are both legal because they are guaranteed to produce unique values regardless of the order of evaluation of operands (i.e., F and G can only modify X, and H can only modify Y). The expression  $F(R) + G(R)$ , however, would not be interference-free.)

#### 4.4. MODULE DECLARATION

##### Purpose

A module is a portion of a program that contains declarations of data, types, routines, and other modules. The purpose of putting such declarations in a module is to control access to them.

Declarations inside a module are accessible outside the module only when specifically exported from the module. Data local to the module may not be exported, nor may the structure and representation of types declared in the module. Consequently, when a type is exported from a module, objects of that type may only be operated on by using routines exported from the module.

Modules are used to provide abstract data types, e.g., a varying length string type could be specified using a module. Several abstract data types can be defined in the same module.

##### Syntax

- [4-45] module-declaration ::= module-header  
                                  module-body  
                                  module-trailer;
- [4-46] module-header        ::= MODULE module-name  
                                  [ imports-clause ]  
                                  [ exports-clause ];
- [4-47] module-body         ::= closed-scope-body
- [4-48] module-trailer      ::= END MODULE module-name
- [4-49] module-name         ::= identifier
- [4-50] exports-clause      ::= EXPORTS ({name[WITH(routine-name,...)]},...)  
                                  | EXPORTS ALL

##### Semantics

A module declaration associates a name with a group of declarations. The scope of the module-name is the scope containing the module-declaration. The module itself defines a closed scope.

The imports-clause is used to explicitly or implicitly (by naming a routine, module, or abstract type) name global data that can be accessed within the module-body (see constraints below, and scope rules).

The exports-clause is used to make names defined within the module known within the scope containing the module. If the reserved word ALL is used, all type and routine names local to the module scope are exported. Otherwise, only those names listed are exported (see scope rules). Only abstract types (with their operations) and routines may be exported.

Types declared within a module and exported from the module behave differently inside and outside the module. Inside the module, the type-name T is merely an abbreviation for the underlying type specified in a type-definition (this is the normal rule for type-declarations, see Section 3.12.). Outside the module, however, T is an abstract type (or abstract type class if T is parameterized). This means that the underlying type of T is unavailable outside the module and that objects of type T declared outside the module may only be operated on using abstract operations of the type. Abstract operations of type (or type-class) T are those routines (procedures and functions) declared within the module whose names begin with the prefix T\$. When type-name T occurs in the exports-clause of the module, all abstract operations of type T are automatically exported unless T is followed by a WITH clause. In this case, only those operations listed in the WITH clause are exported. In the WITH clause, the routine-name should not include the T\$ prefix.

In addition to operations, the type also has abstract attributes; these are discussed in Section 3.12.

The choice of what abstract operations to provide is up to the definer of the module. However, certain operations will almost always be provided. For example, almost every type should have a STORE\_ operation (which defines the

meaning of the assignment operator for the type). In addition every type should have an operation that either creates or initializes objects of the type.

Outside the module, invocation of an abstract operation of type T need not include the T\$ prefix if the routine signature without the prefix is unique. If not, the T\$ prefix must be included. (Such routines can also be invoked by use of the assignment operator and infix (or prefix) operators of expressions. Constraints on operation definitions that are intended to be used in this way are discussed in Appendix D).

Inside the module, operations on objects of type T will normally invoke operations of the underlying type of T. The abstract operations of T may be invoked within the module only by naming them explicitly, including the T\$ prefix, in the routine invocation. Inside the module, the abstract operations cannot be invoked using infix or prefix operators; such operators will always invoke the operations of the underlying type, not the abstract type. [See J4.4-1.]

When an exported abstract type is declared inside a module to be an enumerated type, then its enumerated-literal values are exported from the module and the abstract type is considered to be an enumerated type outside the module. It is considered to be an unordered enumerated type outside unless the module makes available operations for <, <=, >, >=, SUCC, and PRED.

Data (i.e., variables and constants) local to the module (as opposed to data local to routines declared in the module) are known as own data of the module. Such data become accessible and are initialized upon entry to the scope containing the module (i.e., if the module is in a segment, the own data are statically allocated, while if the module is in a procedure, the own data is allocated on the stack). Own data remain in existence throughout the

lifetime of the scope containing the module and their existence are unaffected by entry to and exit from routines in the module body. Own data cannot be made accessible outside the module and can only be operated on using routines of the module.

When a module that is global to a parallel block is imported into several paths, the own data of the module (as is the case with any global data) is shared among the paths. However, when a module is local to a routine that is global to a parallel block, and the routine is invoked in several of the paths, the own data of the module (as is the case with other local data of the routine) is not shared among the paths (i.e., the routine is re-entrant).

#### Constraints

The module-name in the module-trailer must be the same as the module-name in the module-header.

A name must only appear once explicitly in an imports-clause. (A name can be imported implicitly more than once, however, e.g., by importing routines that both import the same global data.)

Input-items (in an imports-clause) may only appear in contexts in which constants are permitted within the module-body.

A module that imports a variable for input must not import a routine that imports the same variable inout.

The only kind of names permitted in the exports-clause are type and routine names local to the scope defined by the module. (Note: This rule permits names exported by an embedded module to be exported.)

The only kind of name in an exports-clause that may be followed by a WITH clause is a type-name; in this case, the names listed must all be operations of that type.

The module-body must not contain executable-stmts, (i.e., it may only contain declaration-stmts, routine-declarations, and module-declarations).

#### Examples

1. MODULE M EXPORTS (STACK);

```

TYPE STACK = RECORD
    VAR ELEMENTS: ARRAY [1:100] OF INTEGER;
    VAR TOP: INTEGER [0:100];
END RECORD;

PROCEDURE STACK$CLEAR (#S: STACK);
    S.TOP := 0;
END PROCEDURE STACK$CLEAR;

PROCEDURE STACK$PUSH (E: INTEGER # S: STACK)
    SIGNALS (STACK_OVERFLOW);
    IF S.TOP < 100 THEN
        S.TOP := S.TOP + 1;
        S.ELEMENTS(S.TOP) := E;
    ELSE
        SIGNAL STACK_OVERFLOW;
    END IF;
END PROCEDURE STACK$PUSH;

PROCEDURE STACK$POP (# S: STACK # E: INTEGER)
    SIGNALS (STACK_EMPTY);
    IF S.TOP > 0 THEN
        E := S.ELEMENTS(S.TOP);
        S.TOP := S.TOP - 1
    ELSE
        SIGNAL STACK_EMPTY;
    END IF;
END PROCEDURE STACK$POP;

PROCEDURE STACK$STORE_ (S1: STACK ## S2: STACK);
    LOOP FOR I IN [1:S1.TOP];
        S2.ELEMENTS(I) := S1.ELEMENTS(I);
    END LOOP I;
    S2.TOP := S1.TOP;
END PROCEDURE STACK$STORE_;

END MODULE M;
```

(This examples defines an abstract type STACK. Stack objects of this type are defined to hold a maximum of 100 integer elements. The CLEAR operation initializes a stack. The PUSH operation puts an element on a stack and the POP operation removes an element from a stack. The STORE\_ operation defines assignment for stacks (outside the module this STORE\_ operation can be invoked using :=). A constructor (e.g., NEW) operation is not defined for stacks. Section 5, Generic Definitions, contains a generalization of this example which defines stacks of any maximum length and any element type.)

```
2. MAIN SEGMENT Z;
...
VAR D,E: T;
...
P(#D);
...
Q(#E);
...
MODULE M EXPORTS (T);
  VAR P_COUNT: INTEGER [1:1000] := 0;
  VAR Q_COUNT: INTEGER [1:1000] := 0;
  TYPE T = RECORD
    VAR A: INTEGER [1:10];
    VAR B: BOOLEAN;
  END RECORD;
  PROCEDURE T$P (#X: T) IMPORTS (DATA: (#P_COUNT));
    ...
    P_COUNT := P_COUNT +1;
    X.B := FALSE;
    X.A := 2 * X.A;
    ...
  END PROCEDURE T$P;
  PROCEDURE T$Q (#Y: T) IMPORTS (DATA: (#Q_COUNT));
    ...
    Q_COUNT := Q_COUNT +1;
    Y.B := TRUE;
    Y.A := 5 * Y.A;
  END PROCEDURE T$Q;
END MODULE M;
END SEGMENT Z;
```

This example contains a module M which exports a type T with its two operations P and Q. The module also contains some own data, the variables P\_COUNT and Q\_COUNT, that are accessible only within the module. Outside the module, two variables, D and E, of type T are created. The components of these variables (i.e., the record fields A and B) are not accessible, but the variables can be passed as arguments to the routines P and Q. Within the module (i.e., within routines P and Q), the component fields A and B of objects of type T are accessible.

#### 4.5. CONTROL STATEMENTS

In discussing the flow of control, several terms are used: construct, initiation, completion, and termination. A construct is a syntactically defined program part that is under discussion. Initiation means that execution of a construct begins. Completion means a control construct has finished execution and the next construct to be executed is determined by the standard rules for the context in which the construct is embedded. For example, (normal) completion of a loop-body implies that the loop control predicate will be tested to see if the loop-body should be executed again. Termination means execution of a control construct cannot be completed (normally) because a goto-stmt, exit-stmt, return-stmt, or raised exception is directing control out of the construct. Termination implies that execution of the terminated construct stops and selection of the next construct to be executed depends on the reason for termination. Termination is always considered abnormal in the sense that the normal control flow rules do not apply; completion is always considered normal.

##### 4.5.1. Statements

###### Purpose

Statements are used to specify actions and specify the sequencing of actions.

###### Syntax

- [4-51] executable-stmt ::= [label :]... unlabeled-stmt  
[exceptions-clause]
- [4-52] unlabeled-stmt ::= simple-stmt  
| structured-stmt
- [4-53] label ::= identifier

Semantics

Statement labels are local to the statement sequence of the innermost enclosing open-scope-body and are not importable into closed scopes. This implies that a closed scope (e.g., a routine) cannot be terminated with a goto-stmt, since labels outside the closed scope are not known inside the scope.

Execution of a statement consists of executing actions appropriate to the particular kind of statement. Completion of all the statement's actions completes the statement. Termination of an action terminates the statement.

The sequence in which statements are executed is defined for each kind of structured-stmt.

#### 4.5.2. Simple Statements

##### Purpose

A simple-stmt has no statement or open-scope-body components.

##### Syntax

```
[4-54] simple-stmt ::= assignment-stmt
                  | procedure-stmt
                  | return-stmt
                  | exit-stmt
                  | signal-stmt
                  | go-to-stmt
                  | assert-stmt
                  | null-stmt
                  | parallel-control-stmt
                  | start-io-stmt

[4-55] null-stmt ::= ;

[4-56] go-to-stmt ::= GO TO label;

[4-57] parallel-control-stmt ::= wait-stmt
                               | terminate-stmt
                               | priority-stmt
                               | request-stmt
                               | release-stmt
```

##### Semantics

The assignment statement is discussed in Sections 3.x.5 since its semantics depend on the type of value being assigned.

The go-to statement terminates execution of itself and all enclosing constructs out to (but not including) the innermost enclosing open-scope-body that has a statement with the same label. It then initiates execution of the labeled statement.

The go-to statement cannot terminate a closed scope, because both the go-to and the labeled statement must be contained in the same closed scopes (See Section SCOPE).

The null statement performs no action.

The procedure and return statements are discussed in Section 4.3.2, the exit statement in 4.5.6.2, the signal statement in 4.5.7, the assert statement

in 4.5.8, the parallel control statements in 4.6, and the start-io-stmt in 4.8.1.

#### 4.5.3. Structured Statements

##### Purpose

Structured statements are composed of component statements or open-scope-bodies that are to be executed sequentially, conditionally, repetitively, or in parallel.

##### Syntax

[4-58] structured-stmt        ::= block  
                                  | conditional-stmt  
                                  | repetitive-stmt  
                                  | parallel-block  
                                  | with-clause structured-stmt

[4-59] block                    ::= BLOCK open-scope-body END BLOCK;

##### Semantics

A block has the same semantics as its open-scope-body. [See J4.5-1.]

Conditional statements are discussed in Section 4.5.5, repetitive statements in 4.5.6, and parallel blocks in 4.6.

#### 4.5.4. With Clauses

##### Purpose

A with-clause names a variable or an expression. It is used to abbreviate a complex name or to ensure the type of a record variant is fixed while its fields are subject to change (see case-stmt). [See J4.5-2.]

##### Syntax

[4-60] with-clause                    ::= WITH bound-var := expression  
    | WITH bound-var NAMING variable

[4-61] bound-var                    ::= identifier

##### Semantics

A with-clause may prefix any structured-stmt. It forms an inner open-scope-body consisting of a declaration of the bound-var followed by the structured statement.

If the ":=" form is used, the expression is evaluated, a local variable of the same type is declared, and the value of the expression is assigned to the local variable. Then the structured statement is executed.

If the "NAMING" form is used, then within the structured-stmt the bound-var is a reference to the specified variable. Access to that variable (other than through the bound-var) is prohibited within the structured-stmt. (As a consequence, routines that import the specified variable or its containing aggregates may not be invoked directly or indirectly in the structured-stmt.) The type of the bound-var is the same as the variable (in particular, the bound-var is type T(...,ANY,...) if the variable is type T(...,ANY,...)).

##### Constraints

If the "NAMING" form is used, then there are two cases to consider. (1) If the variable is not obtained by dereferencing a pointer, then the name of the variable (or its containing aggregate) is not imported into the structured statement (and hence there is no aliasing). (2) If the variable is obtained

by dereferencing a pointer, then alternate access paths (from other pointers) to the variable may exist. The variable's value must not be accessed or changed via these alternate paths. [See J4.5-3.]

Examples

1. VAR V: ARRAY[1:5] OF INTEGER;

...  
WITH W NAMING V(I+1)  
BLOCK ... END BLOCK;

(In this example the name V is not known inside the block, i.e., writing V(2) would be an error. Inside the block, W references element I+1 of V, where the value of I+1 is fixed at the time the with-clause is evaluated. A more elaborate example of a with-clause combined with a discriminating case statement is given in example 2 of section 4.5.5.2.)

2. VAR P: PTR(ARRAY[1:5] OF INTEGER);

...  
WITH W NAMING P@(I+1)  
LOOP ... END LOOP;

(In this example, P is dereferenced and so P@(I+1) must not be changed by dereferencing some other pointer to the array referenced by P.)

#### 4.5.5. Conditional Statements

##### Purpose

A conditional statement selects and executes a single one of its component open-scope-bodies.

##### Syntax

```
[4-62] conditional-stmt ::= if-stmt
                           | case-stmt
```

#### 4.5.5.1. If Statements

##### Purpose

The if statement provides for selective execution based on the values of predicates (i.e., boolean expressions).

##### Syntax

```
[4-63] if-stmt ::= if-element [orif-element]...
                           [if-else-element] END IF;
[4-64] if-element ::= IF boolean-expression THEN open-scope-body
[4-65] orif-element ::= ORIF boolean-expression THEN open-scope-body
[4-66] if-else-element ::= ELSE open-scope-body
```

##### Semantics

An omitted else-element is an abbreviation for ELSE; i.e., ELSE with a null-stmt. The statement

IF boolean-expression THEN open-scope-body ORIF .... END IF;

is an abbreviation for

IF boolean-expression THEN open-scope-body ELSE IF .... END IF; END IF;

The semantics for the statement

if-element if-else-element END IF;

are as follows. The boolean-expression of the if-element is evaluated. If it yields a true value, the open-scope-body of the if-element is initiated. Otherwise, the open-scope-body of the else-element is initiated. Completion

of the selected element completes the if statement. Termination of the boolean-expression or of the selected element terminates the if statement.

If the boolean-expression is a manifest-boolean-expression, then only the element selected by the expression's value is used to generate object code. Any statements that would raise exceptions (if executed) in the unselected elements are considered null statements, i.e., no translation-time exceptions are raised. [See J4.5-4.]

#### Examples

```
1. VAR X: FLOAT(8);
   VAR RC: FLOAT(8);
   ...
   IF RC IN [0.:500.) THEN
      X := 2.79 + 3.85 * RC;
   ORIF RC IN [500.:1500.) THEN
      X := 1927.79 + 4.76 * (RC - 500.);
   ORIF RC IN [1500.:3000.) THEN
      X := 6687.79 + 6.25 * (RC - 1500.);
   ELSE X := 16062.79 + 8.37 * (RC - 3000.);
   END IF;
```

(This illustrates the kind of piecewise approximation that occurs in aircraft aerodynamics simulation.)

#### 4.5.5.2. Case Statements

##### Purpose

The case statement provides for selective execution based on the value of an expression. In particular, it provides access to the variant parts of a variant record and for the designation of machine-dependent code segments.

##### Syntax

- [4-67] case-stmt ::= SELECT choice FROM [case-body] END SELECT;
- [4-68] choice ::= expression  
| bound-var . tag-name  
| OBJECT\_MACHINE
- [4-69] case-body ::= case-element... [case-else-element]
- [4-70] case-element ::= choice-label --> open-scope-body  
END; [exceptions-clause]

- [4-71] case-else-element ::= ELSE --> open-scope-body END;  
[exceptions-clause]
- [4-72] choice-label ::= case-label,...  
| variant-name,...  
| machine-config,...
- [4-73] case-label ::= manifest-expression  
| manifest-range-spec
- [4-74] machine-config ::= identifier

#### Semantics

The choice is evaluated to yield a value. If there is a case-element whose choice-label includes a value equal to the choice, then that case-element is selected. Otherwise, the else-element is selected if it is present (if not, the case-stmt is completed, i.e., the effect is like that of selecting an else-element containing a null-stmt). The selected element's open-scope-body is initiated. Completion of the open-scope-body completes the case statement. Termination of the expression or of the selected scope terminates the case statement.

A manifest-range-spec case-label is an abbreviation for all values in the range.

If the choice is of form bound-var.tag-name, the case statement must be in the inner scope of the with-clause that binds the bound-var, and the tag-name must be a formal tag-name of the variant record type of that bound-var. Within the selected case-element, the type of the bound-var is T(...,e,...) rather than T(...,ANY,...), where e is the current value of the selected tag-name. Hence, the field-names of the selected variant-spec are accessible in the case-element (via field-references). However, if the else-element is selected, the type of the bound-var remains as in the with-clause, i.e., usually T(...,ANY,...), so that no additional field-names are accessible.

If the choice is the constant-name OBJECT\_MACHINE, the value used for selecting a case-element is its value and each case-element is considered machine-dependent. Use of machine-dependent language capabilities is permitted only in the open-scope-body of a machine-dependent case-element.

#### Constraints

The type of the choice must be a type for which the equality operator is defined (i.e., any predefined type). [See J4.5-5.]

The type of the choice-labels and the choice must satisfy the constraints for equality comparison of values of the type.

The choice-label values must be distinct.

If the choice is of the form bound-var.tag-name, the tag-name must be a tag formal parameter of the variant record type of the bound-var; in addition, if more than one variant-name is used in a choice-label in a single case-element, the names must be associated with the same variant-part of the record (see example 2). (As a result, assignment to the entire bound-var as well as to individual fields is safe and permitted, provided that the aliasing constraints of the with-clause are observed (section 4.5.4).) [See J4.5-6.]

#### Examples

```
1. TYPE COLOR = (\RED, \ORANGE, \YELLOW, \GREEN, \BLUE, \VIOLET);
   VAR I: INTEGER [1:10];
   VAR C: COLOR;
   ...
   SELECT I FROM
      1 -->      C := \RED; END;
      [2:4], 9 --> C := \YELLOW; END;
      [5:7] -->   C := \GREEN; END;
      8,10 -->   C := \VIOLET; END;
   END SELECT;
```

(This example is an ordinary case statement.)

```
2.  TYPE T(TAG B: BOOLEAN) =
      RECORD
        SELECT B FROM
          TRUE --> VAR X: INTEGER; END;
          FALSE --> VAR Y: FLOAT(8); END;
        END SELECT;
      END RECORD;
      VAR V: ARRAY [1:5] OF T(ANY);
      ...
      WITH W NAMING V(3)
        SELECT W.B FROM
          TRUE --> W.X := W.X + 2; END;
          FALSE --> W.Y := 3.5; END;
      END SELECT;
```

(This example discriminates a variant record. Note that V is not imported into (accessible in) the case statement.)

#### 4.5.6. Repetitive Statements

##### Purpose

A repetitive statement executes its body repeatedly until the statement completes or terminates.

##### Syntax

[4-75] repetitive-stmt ::= indefinite-loop-stmt  
| definite-loop-stmt

#### 4.5.6.1. Indefinite Loop Statement

##### Purpose

This statement provides for indefinite repetitive execution of its body until the body terminates via an exit statement or until a controlling boolean expression is true. [See J4.5-7.]

##### Syntax

[4-76] indefinite-loop-stmt ::= LOOP [UNTIL predicate DO]  
open-scope-body  
END LOOP [WHEN predicate];

[4-77] predicate ::= boolean-expression

##### Semantics

The phrase

LOOP UNTIL predicate DO

is an abbreviation for (equivalent to) the phrase

LOOP IF predicate THEN EXIT; END IF;

The phrase

END LOOP WHEN predicate;

is an abbreviation for (equivalent to) the phrase

IF predicate THEN EXIT; END IF; END LOOP;

The semantics for

LOOP open-scope-body END LOOP;

are as follows. The open-scope-body is initiated. Normal completion of the

body is followed by another initiation of the body. Termination of the body by a (textually contained) exit statement completes the indefinite loop statement. Termination of the body by any other construct terminates the loop statement.

#### Constraints

Only one UNTIL or WHEN form is permitted in a given indefinite-loop-stmt.  
[See J4.5-8.]

#### Examples

```
1. TYPE NODE = RECORD
    VAR DATA: some type;
    VAR NEXT: PTR(NODE);
  END RECORD;
  VAR P: PTR(NODE);
  ...
  P := list of nodes;
  LOOP UNTIL P = NIL DO
    R(P^.DATA);
    P := P^.NEXT;
  END LOOP;
```

(This example applies a routine R to each data field in a list of nodes.)

```
2. LOOP
  read next record;
  IF end of file THEN EXIT; END IF;
  process the record;
END LOOP;
```

(This example shows a general algorithm for independently processing each record in a file.)

```
3. VAR N: INTEGER;
  ...
  N := a positive integral value;
  LOOP
    CONST M: INTEGER := some expression using N;
    ... using M;
    N := N - 1;
  END LOOP WHEN N <= 0;
```

(In this example, M is local to the loop body and is initialized on each execution of the loop body.)

#### 4.5.6.2. EXIT Statement

##### Purpose

The exit statement is used to exit from a loop.

##### Syntax

[4-78] exit-stmt ::= EXIT;

##### Semantics

Execution of an exit statement terminates itself and all enclosing constructs out to (but not including) the innermost enclosing repetitive statement, which is then completed. [See J4.5-9.]

##### Constraints

An exit statement must be textually contained in a repetitive statement. Consequently, an exit statement cannot terminate a closed scope.

#### 4.5.6.3. Definite Loop Statement

##### Purpose

This statement provides for repetitive execution with a progression of values assigned to a local constant identifier. The set of values is determined prior to the first execution, and so there is a definite upper bound on the number of repetitions.

##### Syntax

[4-79] definite-loop-stmt ::= LOOP FOR [identifier IN] range-spec DO  
open-scope-body END LOOP [identifier];

##### Semantics

The range specification is evaluated and yields an ordered set of values. The open-scope-body is initiated once for each value in the set, taken in ascending order. After normal completion of the open-scope-body for a value, the open-scope-body is re-initiated using the next value. After the open-scope-body completes execution using the final value, the loop statement is

completed. [See J4.5-10.]

The range specification may yield an empty set of values (e.g., when its upper-bound is less than its lower bound), in which case the open-scope-body is not executed and the loop statement is completed.

Termination of the open-scope-body by a (textually contained) exit statement completes the definite loop statement.

Termination of the open-scope-body by any other construct (e.g., a raised exception), termination during evaluation of the range specification, or termination during sequencing (from one value to the next) all terminate the definite loop statement.

When an identifier is specified (following FOR), it is declared as a local constant and is initialized to each successive value in the set as described above. The declaration is considered to be part of the open-scope-body.

#### Constraints

The range specification element value type must be integer (FIXED) or an ordered enumerated type (<= and SUCC defined).

The identifier following END LOOP must be present if an identifier is used following FOR, and the two identifiers must be identical.

#### Examples

1. LOOP FOR I IN [1:N] DO  
    open-scope-body  
END LOOP I;

This is equivalent to the block

```
BLOCK
  VAR V: INTEGER [1:N] := 1;
  IF V <= V.MAX THEN
    LOOP
      CONST I: INTEGER [1:N] := V;
      open-scope-body
      IF V = V.MAX THEN
        EXIT;
      END IF;
      V := V + 1;
    END LOOP;
  END IF;
END BLOCK;
```

```
2.  VAR A: ARRAY[1:100] OF FLOAT(8);
  VAR SUM: FLOAT(8);
  ...
  SUM := 0.;
  LOOP FOR I IN [A.BOUNDS(1).MIN : A.BOUNDS(1).MAX] DO
    SUM := SUM + A(I);
  END LOOP I;
```

(This example sums the elements of an array A of numbers.)

#### 4.5.7. Exception Handling

##### Purpose

The exception-handling mechanism provides a means for signaling and handling exceptional situations. The mechanism has two components: a signal statement for indicating (dynamically) that an exception situation has occurred and an exceptions clause for providing handlers to process the exceptions. In addition, a signals clause is used to specify the exceptions that a routine or parallel path is permitted to raise. [See J4.5-1.]

The exception-handling mechanism provides well-structured ways of exiting loops and other structured statements as well as of informing the invoker of a routine or the initiator of a parallel path about some situation the closed scope is not prepared to handle.

The exceptions-clause defines handlers to process some or all of the exceptional situations that may arise during execution of the unlabeled-stmt or case-element to which it is attached. To facilitate discussion of its semantics, we reproduce here the syntax productions that permit an exceptions-clause:

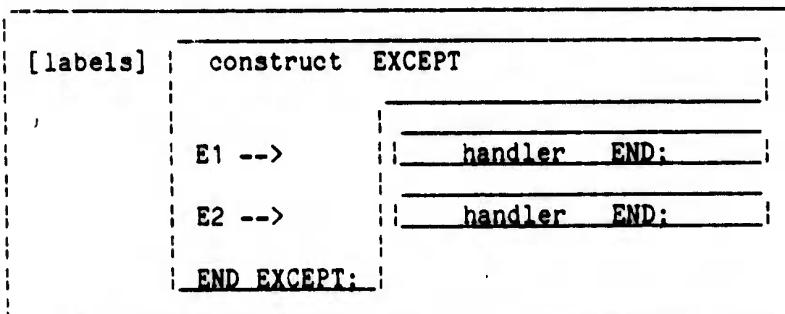
executable-stmt	::= [label :]... unlabeled-stmt [exceptions-clause]
case-element	::= choice-label --> open-scope-body END; [exceptions-clause]
case-else-element	::= ELSE --> open-scope-body END; [exceptions-clause]

##### Syntax

[4-80] exceptions-clause	::= EXCEPT [named-handler]... END EXCEPT;
[4-81] named-handler	::= exception-name,... --> [open-scope-body] END;
[4-82] exception-name	::= identifier
[4-83] signal-stmt	::= SIGNAL [exception-name];

Semantics

The open-scope-bodies of the handlers are considered to be at the same scope level as the control construct (unlabeled-stmt or open-scope-body of a case-element or case-else-element) to which the exceptions-clause is attached. The exception-names are not imported into the open-scope-body of the named-handlers. The scoping of names is as illustrated below for an exceptions-clause containing a handler for exception E1 and one for E2. The lines indicate the scope of names, i.e., E1 and E2 are not imported into the handlers and are not known outside the construct to which the exceptions clause is attached. The labels, if any, are known in the construct and in the handlers, since labels are automatically imported into open-scope-bodies.



Note in particular that labels attached to the unlabeled statement are imported into the handlers of the attached exceptions-clause. A goto statement therefore can be used to exit from a handler (or the construct), and reinstantiate execution of the statement to which the handler is attached.

Exceptions can be raised in two ways: 1) by executing a signal-stmt, or 2) by invoking some routine (including routines associated with infix and prefix operators) that terminates by raising an exception. (Exceptions can also be raised by parallel paths. In the following discussion, unless we explicitly say otherwise, the semantics that apply to routines also apply to parallel paths.)

In discussing exceptions, two considerations must be kept in mind. First, a routine (or a parallel path) can be terminated by raising an exception only by explicitly executing a signal-stmt within the routine's body, and a routine can only signal those exceptions specified in its signals-clause. Second, exceptions are suppressed unless a handler for the exception is provided (see below for details). When an exception is suppressed, it is implied that the condition that would cause the exception to be raised will not occur, e.g., suppressing OVERFLOW exceptions means the correct behavior of a program is predicated on OVERFLOW not occurring; this means the exception situation must not arise. (If it does nonetheless arise, the program is in error and its behavior is undefined.) [See §4.5-2.]

An exceptions-clause is attached to unlabeled statements or elements of case statements, as the syntax productions show. The effect of such a clause is to enable all exception conditions for which the exceptions-clause provides handlers. Enabling an exception means it is no longer suppressed, i.e., if the exception is raised, the program is not in error and its behavior is well-defined.

Within a given closed scope (i.e., a routine body exclusive of any nested routines or modules, or a parallel path body exclusive of nested paths, routines, or modules), when an exception is raised, the control construct that raised it is terminated and a handler is selected as follows:

- . if the raised exception matches a name in an exceptions-clause attached directly to the control construct that raised the exception, then the handler with that name is executed.
- . if the handler completes, then the executable-stmt or case element is completed.
- . if the handler terminates (e.g., with a goto-stmt or by raising an exception) then the executable-stmt or case element is terminated (and the next statement to be executed is determined by the manner in which the construct was terminated (i.e., the usual effect of termination)).

- . if no handler for the raised exception exists in that exceptions-clause or if there is no exceptions-clause, the immediately enclosing control construct is terminated and the exception is raised for that construct.
- . if the enclosing control construct is the closed-scope-body of a routine, the exception is raised for the statement from which the routine was invoked. (In this case, the exception must have been raised explicitly with a signal-stmt and the signals-clause of the routine must name the exception.)
- . for the case when the enclosing control construct is a parallel path, see Section 4.6.1.

Execution of a signal-stmt raises the specified exception and initiates the search for a handler. The exception-name may be omitted only when the signal statement is local to a handler, in which case the exception that initiated the handler's execution is raised. (Note that a different handler will be selected to handle this signal-stmt because of the scope of exception names (see example below).) [See J4.5-3.]

#### Examples

The following example defines a function SUM that sums the integer data in an input stream. The function signals OVERFLOW if the sum is too large and signals INVALID\_FORMAT if the stream contains incorrectly formatted data. The function uses another function GET\_ITEM that returns the next data item in the stream. GET\_ITEM signals EOF when the stream is exhausted and signals INVALID\_ITEM when the next item has an invalid format or is too large.

```

FUNCTION SUM(S: STREAM) RETURNS (INTEGER)
    SIGNALS (OVERFLOW, INVALID_FORMAT)
    IMPORTS (ROUTINES: (GET_ITEM));
    VAR T: INTEGER := 0;
    LOOP
        T := T + GET_ITEM(S);
    END LOOP;
    EXCEPT
        EOF --> RETURN T; END;
        OVERFLOW --> SIGNAL; END; % same as SIGNAL OVERFLOW;
        INVALID_ITEM --> SIGNAL INVALID_FORMAT; END;
    END EXCEPT;
END FUNCTION SUM;

```

Note that the SIGNAL statement in the OVERFLOW handler terminates function SUM and raises OVERFLOW at the invocation of SUM. Similarly, the SIGNAL INVALID\_FORMAT terminates function SUM. If the OVERFLOW handler is removed, then the OVERFLOW exception is suppressed within SUM. In particular, +'s

raising OVERFLOW does not cause SUM to raise OVERFLOW, but instead is an error yielding undefined behavior.

The next example illustrates table searching with hashing. Array A is to be searched for a value X of type T. If the value is not in A then it is to be inserted. Another array B of integers is to contain the number of searches performed for each value. An error message is to be printed if the table is full when inserting a new value. The code below uses a hash function H, that takes a value and returns a nonnegative integer.

```

TYPE T = value type;
CONST M: INTEGER := max number of table entries;
VAR A: ARRAY [1:M] OF T;
VAR B: ARRAY [1:M] OF INTEGER;
CONST EMPTY: T := empty table slot value;
VAR X: T;
LOOP FOR I IN [1:M] DO
  A(I) := EMPTY;
  B(I) := 0;
END LOOP I;
...
X := value to search for;
BLOCK
  VAR I: INTEGER [1:M];
  BLOCK
    CONST J: INTEGER [1:M] := (H(X) MOD M) + 1;
    I := J;
    LOOP
      IF A(I) = X THEN SIGNAL FOUND; END IF;
      IF A(I) = EMPTY THEN
        A(I) := X;
        SIGNAL FOUND;
      END IF;
      IF I = M
        THEN I := 1;
        ELSE I := I + 1;
      END IF;
      IF I = J THEN SIGNAL FULL; END IF;
    END LOOP;
  END BLOCK;
  EXCEPT
    FOUND --> B(I) := B(I) + 1; END;
    FULL --> print error message; END;
  END EXCEPT;
END BLOCK;

```

### Constraints

The handler names of an exceptions-clause must be distinct.

The exception-name in a signal-stmt must either have a handler (attached to a control construct containing the signal-stmt) in the innermost closed

scope containing the signal-stmt, or the closed-scope must have a signals-clause declaring the exception-name (i.e., the closed-scope is permitted to raise the signaled exception).

The exception-name in a signal-stmt must not be TERMINATED (the exception raised when a parallel path's execution is to be terminated (see the TERMINATE statement, Section 4.6.3)). [See J4.5-4.]

#### 4.5.8. ASSERT Statement

##### Purpose

The ASSERT statement permits specification of a boolean expression that can be evaluated at execution time. If the expression is not satisfied, an exception is raised. The ASSERT statement can be used to specify predicates useful in proving programs correct. [See J4.5-5.]

##### Syntax

[4-84] assert-stmt ::= ASSERT boolean-expression;

##### Semantics

The boolean-expression is evaluated. If its value is TRUE, the statement is complete. If its value is FALSE, an exception is raised.

##### Constraints

The boolean-expression must be free of side-effects. [See J4.5-6.]

##### Efficiency Considerations

Since exceptions are normally suppressed, if there is no handler for the exception raised by the assert-stmt, there is no need to evaluate the boolean-expression. We expect, however, that if a program is compiled in a special "debugging" mode, all exceptions will be checked for, and if an unhandled exception is detected, an informative indication of the circumstances will be given. In this case, assert-stmts will be evaluated as well and their violation noted to programmers.

##### Exceptions

The ASSERTION\_VIOLATED exception is raised if the boolean-expression evaluates to FALSE.

#### 4.6. PARALLEL PROCESSING

This section describes the features in the language that permit the parallel execution of portions of a program known as paths. The features are described in terms of their built-in semantics, including such aspects as the built-in scheduling discipline. Although the scheduling discipline is extremely general and flexible, it is anticipated that some applications will require more specialized scheduling disciplines to be built-in, mainly for efficiency purposes. Consequently, it is anticipated that the language features will be implemented as calls to operations in an underlying parallel support module. A definition of the language features in terms of such calls is given in Appendix D. Should it be necessary in a particular implementation to change the built-in semantics (e.g. the scheduling discipline), a user-defined module containing the appropriate operations can be substituted for the built-in module.

The primary parallel language feature is the parallel-block. This construct is used to define and activate parallel-paths. A distinction is made between a path, which consists of the textual code, and a path activation, which consists of the execution of that code. This distinction is needed because a single path may have multiple concurrent activations. Such a path is indicated with a loop-prefix in its definition.

A path is considered to be in one of two states: activated or deactivated. A path is activated if there exists one or more path activations of the path. Otherwise the path is deactivated. A path activation may be in one of the following states: executing, ready, blocked, terminated. An executing path activation is one that is currently assigned a processor. A ready path activation is one that could be executing if a processor were available to be assigned to it. A blocked path activation is one that cannot

be assigned a processor because it is waiting for a semaphore it has requested or it is waiting for a certain time interval to elapse. A terminated path activation is one that has completed execution, either normally or abnormally, but has not yet been deactivated.

Execution of a parallel-block consists of activating all the paths (with multiple activations for those with loop prefixes), and then simultaneously putting them in the ready state. When all the path activations are in the terminated state, all the paths are deactivated and execution of the parallel-block is complete.

Synchronization of path activations and mutual exclusion is accomplished using semaphores with the REQUEST and RELEASE operations. REQUEST is essentially Dijkstra's P operation; RELEASE is essentially Dijkstra's V operation.

Asynchronous hardware interrupts are handled synchronously in the language. An interrupt is essentially a synchronization indication between two parallel path activations, a hardware one and a software one. Consequently semaphores can be used for handling interrupts. The occurrence of an interrupt is treated as a RELEASE operation by a hardware path performed on a special implementation defined semaphore (there will be one such semaphore for each kind of interrupt that can occur in an implementation). Consequently, such an interrupt can be handled by a path activation issuing a REQUEST on the semaphore. Such a path activation is in effect a logical handler for the interrupt and must declare that fact with a CONNECT clause in its prefix.

The CONNECT clause makes the name of the semaphore known within the path and establishes a connection between the path activation and the interrupt, thus preventing several different path activations from waiting for the same interrupt.

#### 4.6.1. Parallel Blocks

##### Purpose

A parallel-block is used to define and activate parallel-paths.

##### Syntax

[4-85]	parallel-block	::= PAR_BLOCK parallel-path... END PAR_BLOCK;
[4-86]	parallel-path	::= path-header path-body path-trailer;
[4-87]	path-header	::= PATH path-name [loop-prefix] [PRIORITY priority-expr] [CONNECT (interrupt)] [signals-clause] [imports-clause];
[4-88]	path-body	::= closed-scope-body
[4-89]	path-trailer	::= END PATH path-name
[4-90]	loop-prefix	::= FOR identifier IN range-spec
[4-91]	path-name	::= identifier
[4-92]	priority-expr	::= integer-expression
[4-93]	interrupt	::= semaphore-name
[4-94]	semaphore-name	::= identifier

##### Semantics

A parallel-block is an open scope in which only paths can be declared.

Each parallel-path defines a closed scope.

The scope of the identifier declared in a loop-prefix is the scope defined by the path (including the path-header), not the parallel-block.

The names of all paths declared in this parallel-block, and any other parallel-blocks in which this one is nested, are automatically known in all the paths in this parallel block. Data declared in outer nested scopes are not automatically known, but must be explicitly imported by including them in the

imports-clause (see scope rules).

The purpose of the CONNECT clause is to allow a path to act as a logical interrupt handler for the interrupt specified in the CONNECT clause. The CONNECT clause makes the name of the interrupt semaphore known within the path. When the path is activated, a connection is established between the path activation and the interrupt.

For each path in the parallel-block with a regular prefix, a single activation of the path is created and put in the blocked state. The path activation is initially assigned the execution priority specified. If no priority is specified, a default priority of MIN\_PRIORITY is assigned.

For each path in the parallel-block with a loop-prefix, an activation of the path is created for each value of the loop identifier in the specified range. This value is accessible in the path activation as the (constant) value of the loop identifier. Each activation is initially put in the blocked state. Each activation is assigned the execution priority specified. If no priority is specified, a default priority of MIN\_PRIORITY is assigned. (Note: MIN\_PRIORITY and MAX\_PRIORITY (see Constraints below) are implementation defined constants).

When all paths in the parallel-block have been activated, the activations are all simultaneously put in the ready state.

Ready path activations must be assigned to available processors on a first-in-first-out basis within priorities. Consequently, the priority of any executing path activation will always be greater than or equal to the priority of any ready (non-executing) path activation (see also 4.6.4). [See J4.6-1.]

Execution of a path activation completes normally when the END P, "H suffix is reached or when a RETURN statement is executed. The path is placed in the terminated state. Execution of a path activation is abnormally

terminated when an exception that the path is permitted to raise is raised (signaled) and the exception is not processed within the path body. The path is placed in the terminated state along with the raised exception. The exceptions that a path is permitted to raise are those named in its signals-clause and the TERMINATED exception (which is implicitly declared for all paths).

When a path is abnormally terminated, a RELEASE is automatically performed on all REGION SEMAPHOREs on which the path had successfully performed a REQUEST without having performed a subsequent RELEASE. When a path completes normally, such RELEASEs are not performed.

The execution of the path activation (or main program) containing the parallel-block is blocked until all path activations within the parallel-block are in the terminated state. When all such path activations are in the terminated state, they are deactivated, and the path activation (or main program) containing the parallel-block is put in the ready state. If at least one of the paths has a raised exception pending, then the last of these exceptions is raised by the parallel-block and terminates the block. Otherwise, execution of the parallel-block is complete.

#### Constraints

The value of priority-expr must be in the range MIN\_PRIORITY through MAX\_PRIORITY (see range exception).

The interrupt in a CONNECT clause must refer to an implementation defined semaphore.

Only one activated path may be connected to a particular interrupt semaphore. (See interrupt exception).

The path-names must be unique within the scope defined by the parallel-block.

The identifier in a loop-prefix may only be referenced as a constant.

A parallel-block may only occur in a main segment or nested within a path of an outer parallel-block. It must not occur within a routine body. [See J4.6-2.]

#### Exceptions

The RANGE exception is raised if the value of the priority-expr exceeds the range MIN\_PRIORITY through MAX\_PRIORITY.

The INTERRUPT exception is raised if a path activation attempts to connect to an interrupt semaphore that another path activation is already connected to.

#### 4.6.2. WAIT Statement

##### Purpose

The WAIT statement is used to specify a delay for a specified real time interval.

##### Syntax

[4-95] wait-stmt ::= WAIT (time) ;

[4-96] time ::= fixed-expression

##### Semantics

The wait-statement puts the path activation issuing the statement into the blocked state for at least the number of seconds (or fraction thereof) specified by time. A negative value of time is treated as equivalent to zero. WAIT accepts an argument with any built-in fixed point value type and may be extended (by a user) to accept non-standard (i.e., extended) fixed point value types supported by an implementation. [See J4.6-3.]

##### Examples

1. WAIT (.001\S10);

(This statement blocks the path activation for at least one millisecond.)

2. WAIT (5);

(This statement blocks the path activation for at least 5 seconds.)

3. WAIT (1.0E-6\S20)

(This statement blocks the path activation for at least 1 microsecond.)

#### 4.6.3. TERMINATE Statement

##### Purpose

The terminate statement is used to gain control of a path activation that has gone awry.

##### Syntax

[4-97] terminate-stmt        ::= TERMINATE [(path-reference)];

[4-98] path-reference        ::= path-name [(subscript)]

##### Semantics

The effect of the terminate-stmt is to raise a TERMINATED exception in the designated path activation. If the path activation is blocked, it is put in a ready state. If no path-reference is specified, the path activation issuing the statement is the designated path activation. If the designated path activation has any descendent path activations, the TERMINATED exception is only raised in all descendent path activations (i.e., the TERMINATED exception is only raised in the "leaves" of the activation subtree of which the designated path activation is the "root").

If the designated path activation is already in the terminated state, the statement has no effect.

The terminate-stmt is complete as soon as the activity of raising the TERMINATED exception in the designated path has been initiated.

(Note: The terminate-stmt does not actually terminate the designated path, it merely raises a TERMINATED exception which allows the path to possibly recover. As with any exception, a handler for the TERMINATED exception within the path body may take one of the following actions: 1) It may continue execution of the path by executing a goto-statement to some label, 2) It may re-raise the TERMINATED exception to an outer level by issuing a terminate-statement without a path-reference (this is equivalent to

a signal statement without an exception-name within a handler for that exception, for all exceptions except TERMINATED). If re-raising the TERMINATED exception actually causes the path to abnormally terminate (which will be the case if there is no outer-level TERMINATED handler within the path), then the path's REGION SEMAPHOREs will be released as discussed in Section 4.6.1. 3) The TERMINATED handler may complete normally. In this case, the construct to which the handler was attached completes normally. If the handler was attached to the entire body of the path, the path will complete normally (and its REGION SEMAPHOREs will not be RELEASED)..)

(It should also be noted that should a handler for the TERMINATED exception within a path activation go awry, another terminate-statement applied to the path activation will terminate the handler by raising the TERMINATED exception, causing execution of an outer-level handler (if any) for the exception. Thus repeated application of the terminate-statement to a path activation can be used to ensure that the path actually terminates.)

#### Constraints

A subscript may only be applied to a path-name that was declared with a loop-prefix.

The value of the subscript expression must be within the range of values of the loop identifier (see subscriptrange exception).

#### Exceptions

The SUBSCRIPTRANGE exception is raised if the value of the subscript expression in a path-reference exceeds the range of values of the loop identifier in the path's loop-prefix.

#### 4.6.4. PRIORITY Statement

##### Purpose

The priority statement is used to dynamically alter the priority of a path activation.

##### Syntax

[4-99] priority-stmt ::= PRIORITY (priority-expr [, path-reference]);

##### Semantics

The priority-statement changes the priority of the designated path activation to the value of the designated priority-expr. If a path activation is not specified, the path activation issuing the statement is the designated path activation.

If the designated path activation has already terminated, the statement has no effect.

(As mentioned in the section on parallel-blocks, path activations are assigned processors on a first-in-first-out basis within priorities. Consequently, the priority-statement may have the effect of changing the designated path activation from ready to executing and another path activation from executing to ready, or it may have the effect of changing the designated path activation from executing to ready and another path activation from ready to executing, or it may have no effect on path activation states).

##### Constraints

The value of priority-expr must be in the range MIN\_PRIORITY through MAX\_PRIORITY.

##### Exceptions

The RANGE exception is raised if the value of the priority-expr exceeds the range MIN\_PRIORITY through MAX\_PRIORITY.

#### 4.6.5. CLOCK Function

##### Purpose

The clock-function is used to access the cumulative processing time of a path activation.

##### Syntax

[4-100] clock-function ::= CLOCK [(path-reference)]

##### Semantics

The clock-function returns a fixed-point value equal to the cumulative processing time (i.e., time in executing state) in seconds (or fraction thereof) of the designated path activation. If a path activation is not specified, the path activation executing the function is the designated path activation. The scale of the returned value is implementation-dependent but may be determined using an attribute-query, e.g., (CLOCK(P1)).SCALE. Alternatively, the returned value may be converted to any desired form using the TO\_FIXED operator.

(Note: Even if the designated path activation has terminated, this function will return the correct value since all path activations (including terminated ones) whose name is known in the scope of the function reference will still be activated.)

#### 4.6.6. REQUEST Statement

##### Purpose

The request statement is used for synchronization and mutual exclusion of parallel path activations.

##### Syntax

[4-101] request-stmt        ::= REQUEST (#semaphore-variable);  
[4-102] semaphore-variable    ::= variable

##### Semantics

If the value of the referenced semaphore is positive, the request-statement has the effect of decrementing the value by one.

If the value of the semaphore is zero, the request-statement puts the path activation issuing the statement into the blocked state until the path activation is unblocked (i.e. put into the ready or executing state) by another path activation issuing a release-statement on the same semaphore.

Execution of a request-statement is an indivisible operation.

(Note: The '#' preceding the semaphore-variable indicates that the semaphore is being passed inout to the REQUEST routine.)

#### 4.6.7. RELEASE Statement

##### Purpose

The release-statement is used for synchronization and mutual exclusion of parallel path activations.

##### Syntax

[4-103] release-stmt                    ::= RELEASE (#semaphore-variable);

##### Semantics

If any path activations are blocked on the referenced semaphore, the highest priority path activation (the one that has been blocked the longest if there are several at the same priority level) is unblocked (i.e., put in a ready or executing state, depending on its priority).

If no paths are blocked on the referenced semaphore, the value of the semaphore is incremented by one.

Execution of a release statement is an indivisible operation.

(Note: The '#' preceding the semaphore-variable indicates that the semaphore is being passed inout to the RELEASE routine.)

#### 4.6.8. Examples

```

1. PAR_BLOCK
    PATH A;
        VAR S: SEMAPHORE(SYNCH) := NEW(0);
        ...
        PAR_BLOCK
            PATH B PRIORITY (3) IMPORTS (DATA:(#S));
            ...
            REQUEST (#S);
            ...
            END PATH B;
            PATH C IMPORTS (DATA: (#S));
            ...
            RELEASE (#S);
            IF CLOCK (DISK_DRIVER(3)) > MAX THEN
                TERMINATE (DISK_DRIVER(3));
            END IF;
            ...
            END PATH C;
        END PAR_BLOCK;
    END PATH A;
    PATH DISK_DRIVER FOR I IN [1:N] CONNECT (DISK(I));
    ...
    REQUEST (#DISK(I));
    ...
    END PATH DISK_DRIVER;
END PAR_BLOCK;

```

This is an example of a parallel-block with a second parallel-block nested in one of the paths. The outer parallel-block contains two paths, A (with a regular prefix) and DISK\_DRIVER (with a loop prefix). Execution of the parallel-block will create one activation of path A, and N activations of DISK\_DRIVER (where N is determined on entry to the parallel-block).

Each activation of DISK\_DRIVER serves as a logical interrupt handler for an interrupt from a particular DISK (as indicated in the CONNECT clause). The REQUEST statement is used to wait for an interrupt.

Path A contains a nested parallel-block with two paths, B and C. It also contains a semaphore, S, which is used for synchronizing paths B and C (with the request and release statements). S must be explicitly imported (in an INOUT position) into paths B and C.

Path C uses the CLOCK function to test the cumulative processing time of the third activation of DISK\_DRIVER and if it has exceeded the value MAX, abnormally terminates the activation. (Note that DISK\_DRIVER cannot terminate path B or C but can terminate path A.)

Path B is given an explicit priority of 3. Paths A, C and DISK\_DRIVER get a default priority of MIN\_PRIORITY.

```

2. MAIN SEGMENT PRODUCER_CONSUMER;

    VAR BUF: BUFFER(100);

    CLEAR(BUF);
    PAR_BLOCK
        PATH PRODUCER IMPORTS (DATA: (# BUF));
        VAR ITEM: BIT(8);
        ...
        APPEND (ITEM # BUFF);
        ...
    END PATH PRODUCER;

    PATH CONSUMER IMPORTS (DATA: (# BUF));
    VAR ITEM: BIT(8);
    ...
    REMOVE (# BUF # ITEM);
    ...
END PATH CONSUMER;
END PAR_BLOCK;

MODULE BOUNDED_BUFFER_MONITOR EXPORTS (BUFFER);

TYPE BUFFER (LENGTH: INTEGER) = RECORD
    VAR SLOTS: ARRAY [0:LENGTH-1] OF BIT(8);
    VAR LASTPOINTER, FIRSTPOINTER: INTEGER [0:LENGTH-1];
    VAR EMPTY_SLOT,FULL_SLOT: SEMAPHORE (SYNCH);
    VAR MUTEX: SEMAPHORE (REGION);
END RECORD;

PROCEDURE BUFFER$CLEAR (# B: BUFFER(?N));
    B.LASTPOINTER := 0
    B.FIRSTPOINTER := 0
    B.EMPTY_SLOT := SEMAPHORE$NEW(N);
    B.FULL_SLOT := SEMAPHORE$NEW (0);
    B.MUTEX := SEMAPHORE$NEW (1);
END PROCEDURE BUFFER$CLEAR;

PROCEDURE BUFFER$APPEND (X: BIT(8) # B:BUFFER (?N));
    REQUEST (# B.EMPTY_SLOT);
    REQUEST (# B.MUTEX);
        B.SLOTS (B.LASTPOINTER) := X;
        IF B.LASTPOINTER = N-1 THEN
            B.LASTPOINTER := 0;
        ELSE
            B.LASTPOINTER := B.LASTPOINTER +1;
        END IF;
        RELEASE (# B.FULL_SLOT);
    RELEASE (# B.MUTEX);
END PROCEDURE BUFFER$APPEND;

PROCEDURE BUFFER$REMOVE (# B: BUFFER (?N) # X: BIT(8));
    REQUEST (# B.FULL_SLOT);
    REQUEST (# B.MUTEX);

```

```

X := B.SLOTS (B.FIRSTPOINTER);
IF B.FIRSTPOINTER = N-1 THEN
    B.FIRSTPOINTER := 0;
ELSE
    B.FIRSTPOINTER := B.FIRSTPOINTER + 1;
END IF;
RELEASE (# B.EMPTY_SLOT);
RELEASE (# B.MUTEX);
END PROCEDURE BUFFER$REMOVE;

END MODULE BOUNDED_BUFFER_MONITOR;
END SEGMENT PRODUCER_CONSUMER;

```

(This example shows how high-level synchronization primitives (e.g., monitors) can be implemented using the low-level primitives and other facilities of the language. The main program consists of two parallel paths, PRODUCER and CONSUMER. PRODUCER generates items which CONSUMER uses. Communication between PRODUCER and CONSUMER is done via the BOUNDED\_BUFFER\_MONITOR (adapted from [Hoare 74]). This monitor, which is implemented as a module, exports a type, BUFFER. The main program declares a variable, BUF of this type. BUF is the shared data structure (i.e., the buffer) through which PRODUCER and CONSUMER communicate. The representation of the buffer is only accessible in the monitor (i.e., the module) and thus access to it can only be accomplished through the monitor procedures APPEND and REMOVE. These procedures ensure exclusive access to the buffer using the region semaphore MUTEX and the REQUEST and RELEASE primitives. Synchronization is accomplished using the synch semaphores FULL\_SLOT and EMPTY-SLOT and the REQUEST and RELEASE primitives. These ensure that when the PRODUCER tries to APPEND when the REMOVE from an empty buffer, he will be blocked until something is put in the buffer. (Note: ?N, which occurs in the headers of the APPEND and REMOVE procedures, is a generic parameter. These are described in Section 5. Essentially, use of the generic parameter enables the APPEND and REMOVE routines to accept as arguments buffers of any length.)

#### 4.6.9. Simulation

Simulation capabilities can be added to the language by extension in much the same way they were added to SIMULA67 by extension. The simulation package can be defined as a module in an external library. A program wishing to run a simulation will import the module by naming it in its USES clause.

The user program will include a parallel-block in which each path corresponds to a task in his simulation. These paths will then be able to invoke routines exported by the simulation module. (The simulation module is like a monitor; its routines execute as part of the path activation that invokes them.)

The simulation module will contain an own variable which represents the simulation clock. It will also define an abstract type RESOURCE which it will export. (Simulation paths will use RESOURCE variables to synchronize their execution in much the same way normal paths use semaphores, see below.) The module will also contain data structures (i.e., own variables) to keep track of the state of each path in the simulation. The module will export the following routines: SIM\_INIT, SIM\_WAIT, SIM\_REQUEST, and SIM\_RELEASE.

Each path in the simulation must first issue a SIM\_INIT to inform the simulation monitor of its existence. A SIM\_WAIT is issued to inform the monitor that a path wishes to delay its execution for some number of simulated time units. Synchronization of paths is accomplished by issuing SIM\_REQUEST and SIM\_RELEASE on RESOURCE variables. (Note: SIM\_REQUEST is the special seize operation required by IRONMAN 9F.)

Implementation of these operations will be accomplished within the simulation monitor through the use of semaphores. The simulation monitor will physically block and unblock paths of the simulation using semaphores. However, the monitor must keep track in its data structures of which paths are

blocked and for what reason (i.e., whether they are waiting for a resource or waiting for a simulated time interval to elapse). When the last path in a simulation is about to block (i.e., all the others are already blocked), the simulation monitor will increment simulated time sufficiently to unblock the path that is waiting for the shortest elapsed interval of simulated time. Note that it is necessary that all paths of the simulation use only the SIM\_WAIT, SIM\_RELEASE, and SIM\_REQUEST operations for blocking and unblocking. If a path were to block by directly issuing a REQUEST on a semaphore, the simulation monitor would be unaware that the path was blocked, would not increment simulated time at the appropriate point, and thus ruin the simulation. This is due to the fact that the simulation monitor routines execute as part of the path that invokes them. Consequently the monitor must ensure that at least one path is always unblocked; this is why simulated time is incremented when the last path is about to block. If a path blocks on its own without going through the simulation monitor, deadlock of the simulation can occur.

#### 4.7. LIBRARIES

The separate compilation facilities of the language (see Segments) provide a convenient means of building and structuring libraries. Segments permit separate compilation of data, types, routines, and modules. A segment can make available all of the definitions it contains or any specified subset. Furthermore, a segment can import definitions from another segment and make those available as well. It is this latter capability that facilitates the structuring of libraries. For example, suppose an existing library contains a segment (MATRIX) composed of routines for performing matrix operations and another segment (TRIG) containing trigonometric routines. Suppose now that all members of a particular avionics project need access to the matrix and trigonometric routines as well as some common data. The project manager can, for example, create a segment (DATA) containing the common data. Then instead of requiring all users to import all three segments (i.e., DATA, MATRIX, and TRIG), he can instead create a segment for the project (AVIONICS) which imports DATA, MATRIX, and TRIG and exports ALL. Members of the project need only import AVIONICS to obtain all definitions common to the project. Furthermore, as the project progresses and additional segments containing commonly needed routines and data are created, the AVIONICS segment need only be changed to import these segments as well. This structuring can be extended to as many levels as necessary. In addition the selective exporting and importing capabilities of segments allows fine control over which definitions are actually made available.

## 4.8. MACHINE DEPENDENT PROGRAMMING

### 4.8.1. Low-Level Input-Output

There are no predefined device types in the language. Instead, a set of supported device types and the detailed semantics of their operations are specified for each implementation. Nevertheless, the language does provide a standard form for initiating these various implementation-dependent device operations.

The START\_IO statement provides language-constrained access to the full range of low-level input-output capabilities provided by any object machine and any device. Obviously, due to the diverse nature of machine architectures and I/O devices, the parameters accepted by the START\_IO statement, and the implementation of the statement, will be highly machine dependent. However, in the interests of standardization, a minimal uniformity on low-level device interactions is specified by constraining the form of the START\_IO statement.

#### Syntax

```
[4-104] start-io-stmt      ::= START_IO (device-object, device-operation  
                                         [,input-parameters] [#inout-parameters]  
                                         [#output-parameters]])  
  
[4-105] device-object      ::= expression  
  
[4-106] device-operation   ::= expression
```

#### Semantics

The START\_IO statement is used to initiate an operation to a peripheral device. The argument to the device-object parameter must always have one of the implementation-dependent device types. In general, the argument's type will determine which "overloaded case" of the START\_IO operation is appropriate for a given invocation. The interpretation of the argument's value is device-dependent, but it could be used to locate a specific physical device corresponding to the object; for example, the value could be the number

(either logical, or a physical address) of a specific device in the class of devices (e.g., all teletypes) represented by the device type.

The device-operation argument must have an implementation-dependent enumerated type that enumerates the valid operations for the device indicated by the first parameter. The value of the device-operation argument will determine the specific operation to be performed by the given invocation of START\_IO. In general, the device-dependent operations will include capabilities for transferring data to and from devices, for controlling the operation of a device, and for testing the status of a device.

As an example, the reader is referred to Appendix C which defines Application-Level IO Facilities, where START\_IO capabilities for objects of type LT33\_DEVICE are described for a PDP-11 implementation. The legal operations on these teletype device-objects are defined by

```
ENUM (\READ_BYTE, \PRINT_BYTE,           $data I/O
      \ENABLE_PRINTER_INTERRUPT,
      \ENABLE_KEYBOARD_INTERRUPT,..., $control
      \READER_BUSY, \PRINTER_READY,
      \READER_INTERRUPT_ENABLED,...) $status inquiry
```

The arguments to the first two START\_IO parameters will always determine what other device-dependent arguments are required. (This information should be contained in the programmer's guide for each implementation.)

The program invoking the START\_IO operation is not (necessarily) blocked until completion of the operation. (This, of course, depends on the nature of the hardware.) If the device is such that it proceeds in parallel with program execution and issues an interrupt on completion, the program can explicitly wait for completion of the operation with a REQUEST statement in a parallel path CONNECTed to the particular device (See Section 4.6.1); examples of this usage are also included in the IO Appendix.

2/15/78

Machine Dependencies

Constraints

A START\_IO statement may only be used in a machine-dependent portion of a program.

#### 4.8.2. Target Routine Declarations

##### Purpose

Target routine declarations are used to encapsulate assembly code. A target routine is either a procedure or a function.

##### Syntax

[4-107] target-routine-declaration ::= target-procedure-declaration  
                                  | target-function-declaration

[4-108] target-procedure-declaration ::= TARGET procedure-header  
                                  target-procedure-body  
                                  procedure-trailer;

[4-109] target-procedure-body ::= DYN SIZE (stack-size);  
                                  target-scope-body

[4-110] target-scope-body     ::= note: target computer assembly code

[4-111] stack-size         ::= manifest-integer-expression

[4-112] target-function-declaration ::= TARGET function-header:  
                                  target-function-body  
                                  function-trailer;

[4-113] target-function-body ::= DYN SIZE (stack-size);  
                                  target-scope-body

##### Semantics

A target routine declaration associates a name with a body of assembly language code. The scope of the routine name and formal parameters is the same as routine declarations.

The prologue and epilogue code for the routine will be generated by the translator. The stack-size is the number of additional words of dynamic storage the target routine requires over and above the space required by procedure calling conventions.

The mechanism for accessing the stack, formal parameters, imported variables etc. is implementation-dependent and will be described in target dependent documentation that is produced for each translator.

Note that **INLINE** target procedures and functions can be declared, since **INLINE** is permitted in procedure and function headers.

**Constraints**

Target routines cannot be **INLINE**.

Target routines cannot call other routines defined in the language; of course, assembly language routines can be called.

The stacksize must be greater than or equal to zero.

## 5. GENERIC DEFINITIONS

A normal (non-generic) routine definition is one in which the types of all formal parameters, the return-value (if any) and locally declared data are determined when the routine is declared. A generic routine definition is one in which the types of some or all of its formal parameters, locally declared data, and/or return-value are only partially specified, but subject to certain constraints specified with generic parameters. In addition, generic routine definitions include routines that have routine parameters, see Section 5.1.

[See J5.0-1.]

Generic parameters are parameters that affect the types of other values manipulated by a routine. Conceptually, before a generic routine may be invoked, values must be supplied for the generic parameters. The generic routine is then instantiated with the values of the generic parameters to produce a normal routine which is then invoked.

It is an optimization question to determine how instantiation is actually done. In some cases, different routine bodies may need to be produced for different values of the generic parameters, whereas in other cases (e.g., when the generic parameters are the bounds of an array), a single routine body can be used, with the generic parameters (e.g., the array bounds) being passed in as extra hidden parameters).

Generic parameters may be either derived or explicit. Derived generic parameters are indicated as identifiers preceded by a question mark. They occur where a type-spec, a retype-spec, or some attribute of a type-spec or retype-spec would have been written. The value of the derived generic parameter is determined by the type-spec, retype-spec, or attribute value of the actual argument passed. Use of a derived generic parameter in a routine header implicitly declares that identifier. Its type is determined by

context.

Examples

1. FUNCTION CONCAT (X: ASCII\_STR(?L1), Y: ASCII\_STR(?L2))  
RETURNS (ASCII\_STR(L1 + L2));

(This function accepts two strings of any length and returns a string whose length is the sum of the two input strings. Note that the length of the return value is known at the point of call.)

2. FUNCTION EQ (X: ASCII\_STR(?L), Y: ASCII\_STR(?L))  
RETURNS (BOOLEAN);

(This function accepts two character strings but constrains them to be of the same length.)

3. PROCEDURE PRINT (X: ?T) ...;

(This procedure accepts an argument of any type, but only with a standard representation. (The ... will be occupied with a where-clause, see Section 5.2.))

4. PROCEDURE PRINT (S: ?T REP ?R)...;

(This procedure accepts an argument of any type with any representation.)

5. PROCEDURE P (X: INTEGER REP FIXED.REP (?S, SIGNED));

(This procedure accepts an integer argument whose representation can be of any size.)

As the examples indicate, when a derived generic parameter is used in place of an attribute, the parameter's type is determined by the type of the attribute. This kind of derived generic parameter is called a derived (generic) attribute.

When a derived generic parameter is used in place of a type-spec, its type is TYPE (and its use must be constrained by a where-clause, see Section 5.2). This kind of derived generic parameter is called a derived (generic) type.

When a derived generic parameter is used in place of a retype-spec, its type is REPTYPE. This kind of derived generic parameter is called a derived (generic) retype. Note that the standard representation is assumed unless a

non-standard representation is explicitly called for.

Example 2 shows that a derived generic parameter may occur more than once within a formal-parm-list. This means that the values (as determined from the actual arguments) corresponding to each use of the derived generic parameter must be the same. Because derived generic parameters preceded by question marks are in effect being declared as additional formal input parameters, they may not be used in the formal-parm-list as operands in expressions, e.g., ?L1 + ?L2 is not permitted in the formal-parm-list.

The scope of a derived generic parameter is the body of the routine, the where-clause, and the returns-clause (in the case of a function). Uses of derived generic parameters within the routine body, the where-clause, and the returns-clause are not preceded by a question mark (since the question mark indicates a declaration). Derived generic attributes must only be used where constants are permitted. Derived generic types must only be used where type-specs are permitted. Derived generic reptypes must only be used where reptype-specs are permitted.

When the value of a derived generic attribute is not used within the routine body, a question mark by itself may be used in place of the attribute in the formal-parm-list. As with derived generic parameters (i.e., a question mark followed by an identifier) a question mark by itself will match any value of the attribute, but in this case the value will not be accessible in the routine body since there is no name for it. A question mark by itself will be known as the "don't care" symbol. (See also "WHERE Clause" (Section 5.2) and "Routine Parameters" (Section 5.1)).

#### Examples

6. PROCEDURE P (X: FIXED (?, ?, ?) [?:?]);

(This procedure accepts a FIXED argument with any scale, radix, precision, and range. In the routine body, the values of the derived generic parameters are

inaccessible since they are not named. Note that the attributes of X could be obtained using an attribute-query (e.g., X.SCALE); however, this is not recommended practice.)

Explicit generic parameters are input parameters of a routine (i.e., not inout or output parameters) whose value is used to determine the type or attributes of the return-value of a function and/or the type of local data of a routine. Explicit generic parameters may be of any type that can be used as an attribute, or they may be of the special type TYPE (in which case they must be constrained by a WHERE clause). The scope of these parameters is the routine body, the where-clause, and the returns-clause (if any).

Explicit generic parameters of other than type TYPE can be distinguished from ordinary input parameters by their use in determining the attributes of the type of the return-value of a function. The actual arguments passed to such parameters must be constants or constant expressions. (A constant expression is an expression in which all the operands are constants and all the functions invoked are pure, i.e., do not import any data.)

#### Examples

7. FUNCTION TRUNCATE (C: ASCII\_STR(?L1), L: INTEGER)  
RETURNS (ASCII\_STR(L));

(This function accepts a character string argument of any length and an integer argument that is used to specify the length of the string returned by the function.

Note that a derived generic parameter cannot be used in this case because the type of the return value is not derivable from the type of any input parameter. This shows why both derived and explicit parameters are useful. Following are legal invocations of this function:

```
CONST LENGTH: INTEGER := N;  
VAR S1: ASCII_STR(10);  
VAR S2: ASCII_STR(5);  
VAR S3: ASCII_STR(2*LENGTH);  
S2 := TRUNCATE(S1,5);  
S3 := TRUNCATE (S1, 2*LENGTH);
```

Note that the second argument (which corresponds to the explicit generic parameter L) must be constant (although not necessarily manifest).

### 5.1. ROUTINE PARAMETERS

#### Syntax

- [5-1] routine-spec ::= procedure-spec | function-spec
- [5-2] procedure-spec ::= PROCEDURE proc-template
- [5-3] proc-template ::= ([type-spec,...][#[type-spec,...]  
[#type-spec,...]])  
[signals-clause]
- [5-4] function-spec ::= FUNCTION func-template
- [5-5] func-template ::= ([type-spec,...]) RETURNS (type-spec)  
[signals-clause]
- [5-6] routine-template ::= proc-template | func-template
- [5-7] routine-argument ::= procedure-argument  
| function-argument
- [5-8] procedure-argument ::= proc-name [proc-template]
- [5-9] function-argument ::= func-name [func-template]

#### Semantics

Procedure-specs and function-specs are used in declaring formal parameters whose arguments are to be actual procedures and functions, respectively. (They are also used in where-clauses, see Section 5.2.) The routine-template gives the number and types of formal parameters that the argument (i.e., the actual routine being passed) must have. In addition the func-template gives the return type that arguments (i.e., actual functions being passed) must have. The signals-clause indicates some exceptions the passed routine can raise. Only those exceptions for which handlers are to be provided in the generic routine need be listed. The passed routine must be capable of raising the listed exceptions (i.e., in its signals-clause, those exceptions must be named).

Usually, to pass a procedure or function argument as an actual, just the name of the routine being passed need be given. However, when that name has

several declarations (i.e., it has been overloaded), the exact routine intended to be passed can be specified using a routine-template.

During execution of a routine, invoking a formal procedure or function name will invoke the actual routine passed to that formal.

#### Constraints

A routine passed as an argument may not import any data.

Routine-specs must only be used when declaring input formal parameters (i.e., they may not be used to declare inout or output formal parameters).

A '#' must not appear as the last symbol in a proc-template, e.g., PROCEDURE (#) and PROCEDURE (INTEGER #) are forbidden.

The number and types of the formal parameters and return-value (if any) of a routine being passed as an argument must match exactly the number and types of the formal parameters and return-value given in the routine-template for the routine formal parameter. However, the routine-template may contain declarations of derived generic parameters (preceded by a question mark) in place of type-specs, retype-specs or attributes, and it may contain "don't care" symbols in place of attributes. The derived generic parameters will only match real types and values of attributes in the routine argument being passed to the formal (i.e., they will not match derived generic parameters in the argument nor will they match the special type TYPE). The don't care symbol, however, will match any value of the attribute in the argument (including a derived generic parameter -- see example below).

Examples

```

1. A := INTEGRATE (P,1,100);
   B := INTEGRATE (0,10\$,14,20\$,14);

   ...
   FUNCTION P (X: INTEGER [?LB: ?UB])
      RETURNS (INTEGER [LB:UB]);
   ...

END FUNCTION P;

FUNCTION Q (X: INTEGER (14,2,STD) [1:100])
   RETURNS (INTEGER (14,2,STD) [1:100]);
   ...

END FUNCTION Q;
FUNCTION INTEGRATE (FIRSTVAL,LASTVAL : FIXED(?S, ?R, ?P)[?:?],
   F: FUNCTION (FIXED(?S, ?R, ?P) [?:?])
      RETURNS (FIXED (?S,?R,?P)[?:?]))
   RETURNS (FIXED (S,R,P) [FIXED_MIN(S,R,P):FIXED_MAX(S,R,P)]);
   ...

END FUNCTION INTEGRATE;

```

This example shows a function INTEGRATE that accepts two FIXED values, FIRSTVAL and LASTVAL, of any scale, radix, precision, and range, and a function, F, provided that F accepts a FIXED argument of the same scale, radix and precision as FIRSTVAL and LASTVAL and returns a FIXED value of that same scale, radix and precision. Note that the question marks (don't care symbols) in the range-specs for F's argument and return value match anything (including the derived generic parameters ?LB and ?UB) so that both P and Q are acceptable as arguments to F.

## 5.2. WHERE CLAUSE

### Syntax

- [5-10] where-clause ::= WHERE needs-spec,...
- [5-11] needs-spec ::= type-name NEEDS (template-spec,...)
- [5-12] template-spec ::= routine-name routine-template

### Semantics

When a type is passed as a derived or explicit generic parameter to a routine, a where-clause is required in the routine header to state how objects of the type will be used in the routine body (i.e., what operations (routines) the type is expected to support). The routines specified in the where-clause, plus other generic routines as discussed below, are the only routines that may be invoked with objects of the type. Within the routine body, the routines listed in the where-clause are considered to be operations of the type (i.e., they can be invoked with a T\$ prefix or with infix or prefix operator notation). However, the actual routines may be operations of the type or any other routines (accessible to the caller) with the specified names that accept arguments of the types specified in the where-clause. An invocation of the generic routine is legal only if there exist actual routines known to the caller that correspond to the routines specified in the WHERE clause. These routines become available to be invoked within the generic routine by virtue of having been named in the WHERE clause.

The routine-templates in the WHERE clause specify the types of arguments and return-values of the routines listed in the WHERE clause.

A generic routine, A, that accepts either a derived or explicit TYPE parameter may invoke another generic routine, B, known in its scope without naming it in its where-clause. A is permitted to pass B the type, if B has an explicit generic TYPE formal-parameter, and A is also permitted to pass B an

object of the type if B has a formal-parameter whose type-spec is a derived generic-type parameter. The routines listed in B's where-clause must be a subset of the routines listed in A's where-clause. (An example of one generic calling another in this way is shown below.)

#### Constraints

The routine-templates may not contain declarations of derived generic parameters but may use generic parameters from the formal-parm-list. They may, however, contain the "don't care" symbol (i.e., a question mark by itself) in place of any attribute. The don't care symbol means that any value will match.

Routines named in where-clauses may not import any data.

#### Examples

1. PROCEDURE PRINT (X: ?T)  
WHERE T NEEDS (TO\_CHAR(T) RETURNS CHAR(?));

(This procedure accepts a value of any type if the type has an operation TO\_CHAR that accepts an object of the type and returns a character string. The length of the string returned is only indeterminate until the PRINT routine is instantiated.)

2. PROCEDURE P (# X: ?S) IMPORTS (DATA: (# M))  
WHERE S NEEDS (A (# S), C (# S));  
...  
END PROCEDURE P;  
PROCEDURE Q (# Y: ?T) IMPORTS (DATA: (# M))  
WHERE T NEEDS (A (# T), B (# T), C (# T));  
...  
P (# Y);  
...  
END PROCEDURE Q;

(The example contains two generic procedures declared in the same scope, each of which takes an argument of any type (subject to the constraints in the where-clauses). Procedure Q can invoke procedure P without naming it in its where-clause because procedure P is generic in its argument type and the operations required in P's where-clause are a subset of those required in Q's where-clause. Note that in this particular case, Q could not name P in its where-clause because P imports data).

```

3. MODULE MATRIX_ARITHMETIC EXPORTS(MATRIX);
   TYPE MATRIX (M: INTEGER, N: INTEGER, E: TYPE) = ARRAY [1:M, 1:N] OF E;

   FUNCTION MATRIX$MULTIPLY_ (M1: MATRIX(?M, ?N, ?E),
                             M2: MATRIX(?N, ?P, ?E))
      RETURNS (MATRIX(M, P, E))
      SIGNALS (OVERFLOW)
      WHERE E NEEDS (MULTIPLY_ (E, E) RETURNS (E),
                     ADD_ (E, E) RETURNS (E),
                     STORE_ (E ## E));

      VAR RESULT: MATRIX (M, P, E);

      LOOP FOR I IN [1:M] DO
         LOOP FOR J IN [1:P] DO
            RESULT(I, J) := M1(I, 1) * M1(1, J);
            & LOOP FOR K IN [2:N] DO
               RESULT(I, J) := RESULT(I, J) +
                  M1(I, K) * M2(K, J);
            END LOOP K;
         END LOOP J;
      END LOOP I; EXCEPT
         OVERFLOW --> SIGNAL OVERFLOW; END;
      END EXCEPT;
      RETURN (RESULT);
   END FUNCTION MATRIX$MULTIPLY_;

   FUNCTION MATRIX$MULTIPLY_ (C: ?E, M1: MATRIX (?M, ?N, ?E))
      RETURNS (MATRIX (M, N, E))
      SIGNALS (OVERFLOW)
      WHERE E NEEDS (MULTIPLY_ (E,E) RETURNS (E),
                     STORE_ (E ## E));

      VAR RESULT: MATRIX (M, N, E);

      LOOP FOR I IN [1:M] DO
         LOOP FOR J IN [1:N] DO
            RESULT(I, J) := C * M1(I, J);
         END LOOP J;
      END LOOP I; EXCEPT
         OVERFLOW --> SIGNAL OVERFLOW; END;
      END EXCEPT;
      RETURN (RESULT);
   END FUNCTION MATRIX$MULTIPLY_;

   FUNCTION MATRIX$ADD_ (M1, M2: MATRIX (?M, ?N, ?E))
      RETURNS (MATRIX (M, N, E))
      SIGNALS (OVERFLOW)
      WHERE E NEEDS (ADD_ (E, E) RETURNS (E),
                     STORE_ (E ## E));

      VAR RESULT: MATRIX (M, N, E);

      LOOP FOR I IN [1:M] DO

```

```

LOOP FOR J IN [1:N] DO
    RESULT(I, J) := M1(I, J) + M2(I, J);
END LOOP J;
END LOOP I; EXCEPT
    OVERFLOW --> SIGNAL OVERFLOW; END;
    END EXCEPT;
    RETURN (RESULT);
END FUNCTION MATRIX$ADD_;

FUNCTION MATRIX$EQUAL_ (M1, M2: MATRIX (?M, ?N, ?E))
RETURNS (BOOLEAN)
WHERE E NEEDS (EQUAL_ (E, E) RETURNS (BOOLEAN));

    RETURN (M1=M2);
END FUNCTION MATRIX$EQUAL_;

FUNCTION MATRIX$NEW (A:ARRAY [?L1 : ?U1, ?L2 : ?U2] OF ?T)
RETURNS MATRIX (U1 - L1 + 1, U2 - L2 + 1, T);

    RETURN (A);
END FUNCTION MATRIX$NEW;

PROCEDURE MATRIX$STORE_ (M1: MATRIX (?M, ?N, ?E) ##
                           M2: MATRIX (?M, ?N, ?E))
WHERE E NEEDS (STORE_ (E ## E));

    M2 := M1;
END PROCEDURE MATRIX$STORE_;
END MODULE MATRIX_ARITHMETIC;

```

(This is an example of a module that implements the type-class of matrices. The module defines and exports an abstract type MATRIX along with its operations.

MATRIX has two MULTIPLY\_ operations. One multiplies a matrix by a matrix and the second multiplies a scalar by a matrix. Both routines are generic over the size and element type of the matrix. The first (MATRIX \* MATRIX) requires the element type have a MULTIPLY\_, ADD\_, and STORE\_ routine. The second (scalar \* MATRIX) requires the element type have a MULTIPLY\_ and STORE\_ routine.

The ADD\_ operation adds two matrices providing that the element type has an ADD\_ and STORE\_ operation.

The EQUAL\_ and STORE\_ operations merely invoke the EQUAL\_ and STORE\_ operations of the MATRIX representation type, namely, ARRAY. However, the element type is required to provide an EQUAL\_ and STORE\_ operation.

The NEW operation constructs a MATRIX from an ARRAY, determining the size and element type of the matrix from the extents and element type of the array.

Outside the module, MULTIPLY\_, ADD\_, EQUAL\_, and STORE\_ may all be invoked using the appropriate infix operators (\*, +, =, and :=).)

```

4. MODULE M EXPORTS (STACK);
   TYPE STACK (ELEMENT_TYPE: TYPE,
                LENGTH: INTEGER [1:FIXED_MAX(0, 2, STD)]) =
      RECORD
         VAR ELEMENTS: ARRAY [1:LENGTH] OF ELEMENT_TYPE;
         VAR TOP: INTEGER [0:LENGTH];
      END RECORD;

      PROCEDURE STACK$CLEAR (# S: STACK (?, ?));
         S.TOP := 0;
      END PROCEDURE STACK$CLEAR;

      PROCEDURE STACK$PUSH (E: ?T # S: STACK (?T, ?LENGTH))
         SIGNALS (STACK_OVERFLOW)
         WHERE T NEEDS (STORE_(T ## T));

         IF S.TOP < LENGTH THEN
            S.TOP := S.TOP + 1;
            S.ELEMENTS(S.TOP) := E;
         ELSE
            SIGNAL STACK_OVERFLOW;
         END IF;
      END PROCEDURE STACK$PUSH;

      PROCEDURE STACK$POP (#S: STACK(?T, ?LENGTH) # E: ?T)
         SIGNALS (STACK_EMPTY)
         WHERE T NEEDS (STORE_(T ## T));

         IF S.TOP > 0 THEN
            E := S.ELEMENTS (S.TOP);
            S.TOP := S.TOP - 1;
         ELSE
            SIGNAL STACK_EMPTY;
         END IF;
      END PROCEDURE STACK$POP;

      PROCEDURE STACK$STORE_ (S1: STACK (?T, ?LENGTH) ##
                                S2: STACK (?T, ?LENGTH))
         WHERE T NEEDS (STORE_(T ## T));

         LOOP FOR I IN [1:S1.TOP];
            S2.ELEMENTS (I) := S1.ELEMENTS(I);
         END LOOP I;
         S2.TOP := S1.TOP;
      END PROCEDURE STACK$STORE_;

END MODULE M;

```

(This example defines an abstract type class STACK. STACK is parameterized by the type of elements the stack can contain and the length of the stack. The operations of STACK are generic over the element type and the length.

The CLEAR operation initializes a stack. The element type and length of the stack are not needed by the CLEAR routine so "don't care" symbols are used in the header. Since no NEW routine is provided, there is no constructor (i.e., NEW) operation for stacks.

The PUSH and POP operations each take in a STACK object and can use derived generic parameters to determine the element type.)

## 5. Defining Varying Length Strings by Extension

The following module defines varying length character strings as a new abstract data type called VAR\_CHAR\_STR. A varying length string S is declared as follows:

```
VAR S : VAR_CHAR_STR (M, CHAR_SET);
```

where M is the maximum length the string S can attain and CHAR\_SET is the name of the character set containing the characters which make up S.

Varying strings have a current length defined to be the length of the latest value assigned to them. The current length can have any value in the range [0:M]. Its value can be determined using the CUR\_LENGTH function.

All strings used in a varying string expression must be defined using the same character set.

The value of a fixed length string variable or literal may be used in varying string expressions and assignments by first converting the value to a varying string using the TO\_VAR\_CHAR function. This method was chosen for the sake of conciseness of presentation; the varying string routines (STORE\_, SUBSTRING\_, etc.) could have been overloaded to accept arbitrary combinations of fixed and varying length strings, consequently allowing mixtures of fixed and varying strings in assignments and expressions.

```
MODULE VSTR EXPORTS(VAR_CHAR_STR);

TYPE VAR_CHAR_STR(MAX_LENGTH: INTEGER[0:(CHAR_STR.LENGTH).MAX],
                  CHAR_SET: TYPE) =
RECORD
  CUR_LENGTH: INTEGER [0:MAX_LENGTH];
  STRING: CHAR_STR(MAX_LENGTH,CHAR_SET);
END RECORD;
```

```
PROCEDURE VAR_CHAR_STR$STORE_ % assignment operator
  (SOURCE: VAR_CHAR_STR(?,?CHAR_SET)
   #
   TARGET: VAR_CHAR_STR(?TARGET_MAX_LENGTH, ?CHAR_SET))

  SIGNALS (INVALID_SIZE);

  IF SOURCE.CUR_LENGTH > TARGET_MAX_LENGTH THEN
    SIGNAL INVALID_SIZE;
  END IF;

  TARGET.CUR_LENGTH := SOURCE.CUR_LENGTH;
  TARGET.STRING[0:SOURCE.CUR_LENGTH-1] :=
    SOURCE.STRING[0:SOURCE.CUR_LENGTH-1];

END PROCEDURE VAR_CHAR_STR$STORE_;

FUNCTION VAR_CHAR_STR$SUBSTRING_ % handles V[L:U] in exprs.
  (V: VAR_CHAR_STR(?V_MAX_LENGTH,?CHAR_SET),
   L,U: INTEGER)
  RETURNS (VAR_CHAR_STR(V_MAX_LENGTH,CHAR_SET))

  SIGNALS (INVALID_SUBSTRING);

  BLOCK
    VAR SUBV: VAR_CHAR_STR(V_MAX_LENGTH,CHAR_SET);

    IF (L<0) ! (U>V.CUR_LENGTH-1) THEN
      SIGNAL INVALID_SUBSTRING;
    END IF;

    SUBV.CUR_LENGTH := MAX(U-L+1,0);
    SUBV.STRING := V.STRING[L:U];
    RETURN (SUBV);
  END BLOCK
  EXCEPT %in case U-L+1 causes overflow
    OVERFLOW--> SIGNAL INVALID_SUBSTRING; END;
  END EXCEPT;
END FUNCTION VAR_CHAR_STR$SUBSTRING_;
```

```

PROCEDURE VAR_CHAR_STR$SUBSTRING_STORE_
  (S: VAR_CHAR_STR(?,?CHAR_SET),
   L,U: INTEGER
  ##
   V: VAR_CHAR_STR(?,?CHAR_SET))

  SIGNALS (INVALID_SIZE,INVALID_SUBSTRING);

  IF (L<0) ! (U>V.CUR_LENGTH-1) THEN
    SIGNAL INVALID_SUBSTRING;
  END IF;

  IF MAX(U-L+1)<>S.CUR_LENGTH THEN
    SIGNAL INVALID_SIZE;
  END IF;

  V.STRING[L:U] := S.STRING[0:S.CUR_LENGTH-1];

END PROCEDURE VAR_CHAR_STR$SUBSTRING_STORE_;


FUNCTION VAR_CHAR_STR$CONCATENATE_
  (S1: VAR_CHAR_STR(?S1_MAX_LENGTH,?CHAR_SET),
   S2: VAR_CHAR_STR(?S2_MAX_LENGTH,?CHAR_SET))
RETURNS (VAR_CHAR_STR(S1_MAX_LENGTH+S2_MAX_LENGTH,CHAR_SET))

  SIGNALS (INVALID_SIZE);

  BLOCK
    VAR S1S2: VAR_CHAR_STR(S1_MAX_LENGTH+S2_MAX_LENGTH,
                           CHAR_SET);

    S1S2.CUR_LENGTH := S1.CUR_LENGTH + S2.CUR_LENGTH;
    S1S2.STRING := S1.STRING[0:S1.CUR_LENGTH-1]
                  !! S2.STRING[0:S2.CUR_LENGTH-1];
    RETURN (S1S2);
  END BLOCK
  EXCEPT      % in the case of length addition overflow
    OVERFLOW --> SIGNAL INVALID_SIZE; END;
  END EXCEPT;
END FUNCTION VAR_CHAR_STR$CONCATENATE_;


FUNCTION VAR_CHAR_STR$CUR_LENGTH
  (S: VAR_CHAR_STR(?,?))
RETURNS (INTEGER);

  RETURN (S.CUR_LENGTH);

END FUNCTION VAR_CHAR_STR$CUR_LENGTH;

```

```
FUNCTION VAR_CHAR_STR$TO_VAR_CHAR  $ convert from fixed length
  (FIXED_STR:CHAR_STR(?FIXED_LENGTH,?CHAR_SET))
  RETURNS (VAR_CHAR_SET(FIXED_LENGTH,CHAR_SET));

  VAR TEMP: VAR_CHAR_STR(FIXED_LENGTH,CHAR_SET);

  TEMP.CUR_LENGTH := FIXED_LENGTH;
  TEMP.STRING := FIXED_STRING;

  RETURN (TEMP);

END FUNCTION VAR_CHAR_STR$TO_VAR_CHAR;

FUNCTION VAR_CHAR_STR$EQUAL_  $ equality operator
  (S1,S2: VAR_CHAR_STR(?,?CHAR_SET))
  RETURNS (BOOLEAN)

  SIGNALS (INVALID_LENGTH);

  IF S1.CUR_LENGTH <> S2.CUR_LENGTH THEN
    SIGNAL INVALID_LENGTH;
  END IF;

  RETURN (S1.STRING[0:S1.CUR_LENGTH-1] =
         S2.STRING[0:S2.CUR_LENGTH-1]);

END FUNCTION VAR_CHAR_STR$EQUAL_;

FUNCTION VAR_CHAR_STR$UNEQUAL_  $ inequality operator
  (S1,S2: VAR_CHAR_STR(?,?CHAR_SET))
  RETURNS (BOOLEAN)

  SIGNALS (INVALID_LENGTH);

  RETURN (NOT VAR_CHAR_STR$EQUAL_(S1,S2))
  EXCEPT  $ reissue signal
    INVALID_LENGTH--> SIGNAL; END;
  END EXCEPT;

END FUNCTION VAR_CHAR_STR$UNEQUAL_;
```

```

FUNCTION VAR_CHAR_STR$DUP
  (CHAR_TO_DUP: VAR_CHAR_STR(?,?CHAR_SET),
   NUM_DUPS: INTEGER)
  RETURNS (VAR_CHAR_STR(NUM_DUPS,CHAR_SET))

  SIGNALS (INVALID_LENGTH);

  VAR RET_STR: VAR_CHAR_STR(NUM_DUPS,CHAR_SET);

  IF (NUM_DUPS<0) ! (CHAR_TO_DUP.CUR_LENGTH<>1) THEN
    SIGNAL INVALID_LENGTH;
  END IF;

  RET_STR.CUR_LENGTH := NUM_DUPS;
  RET_STR.STRING := DUP(CHAR_TO_DUP.STRING,NUM_DUPS);

  RETURN (RET_STR);

END FUNCTION VAR_CHAR_STR$DUP;

FUNCTION VAR_CHAR_STR$BLANKS
  (NUM_BLANKS: INTEGER,
   CHAR_SET: TYPE)
  RETURNS (VAR_CHAR_STR(NUM_BLANKS,CHAR_SET))

  SIGNALS (INVALID_LENGTH);

  VAR BLANK_OF_CHAR_SET: VAR_CHAR_STR(1,CHAR_SET) :=
    VAR_CHAR_STR$TO_VAR_CHAR(CHAR_SET' ');

  RETURN (VAR_CHAR_STR$DUP(BLANK_OF_CHAR_SET,NUM_BLANKS))
  EXCEPT
    INVALID_LENGTH--> SIGNAL; END;
  END EXCEPT;

END FUNCTION VAR_CHAR_STR$BLANKS;

END MODULE VSTR;

```

## APPENDIX A Consolidated Syntax

The following is a listing of all the syntax productions in this report. The reference number appearing at the left margin serves to uniquely identify each production. The number following the / is the page on which the production appears. For example, [3-1/3] means that production 3-1 is defined on page 3-3.

[2-1/1]	character	::= letter   digit   blank   other
[2-2/1]	letter	::= A B C D E F G H I J K L M N O  P Q R S T U V W X Y Z
[2-3/1]	digit	::= 0 1 2 3 4 5 6 7 8 9
[2-4/1]	other	::= ! " # \$ % ' ( ) # + ,- .  / & : ; < = > ? @ _ \\ \ ^ _
[2-5/1]	display-char	::= blank   new-line
[2-6/1]	blank	::= note: the blank (space) character
[2-7/1]	new-line	::= note: the new line character sequence
[2-8/2]	lexical-program	::= [display-char]...{lexical-token... [display-char...]}
[2-9/2]	lexical-token	::= identifier   literal   other-symbols   reserved-word   key-word   comment
[2-10/3]	identifier	::= {letter}\ ?} [letter digit break]...
[2-11/3]	break	::= _
[2-12/4]	other-symbols	::= !   !!   !!*   !+   !-   !//   !MOD   !#!   !+!   !-!   !/!   !//!   !MOD!   #   \$   \$\$   \$   &   '   " (   )   *   **   +   ,   -   : ->   -->   .   /   //   \   : :=   ;   <   <=   =   >   >=   <>   ?   @   _   1   1

[2-13/5]	reserved-word	::= AND   ANY   ARRAY   ASSERT   BIN   BLOCK CONST   DO   ELSE   END   EXCEPT EXIT   FALSE   FOR   FROM   FUNCTION GO   HEX   IF   IN   INLINE   LOOP   MAIN MOD   MODULE   NAMING   NEEDS   NIL NOT   OCT   OF   OR   ORIF   PAR_BLOCK PATH   PROCEDURE   RECORD   REP   RETURN SEGMENT   SELECT   SIGNAL   TAG TERMINATE   THEN   TO   TRUE   TYPE UNTIL   VAR   WHEN   WHERE   WITH   XOR
[2-14/5]	key-word	::= ABS   ALL   ARRAY REP   ARRAY STD AS   BIT   BIT STD   BLANKS   BOOLEAN BOOLEAN STD   BOUNDS   CHARSET   CHAR STR CHAR STR REP   CHAR STR STD   CLOCK CONNECT   DATA   DEFINES   DIMENSIONS DUP   DYN SIZE   ELEMENT TYPE   ENUM ENUM REP   ENUM STD   ENUM VAL REP EPSILON   EXPONENT MIN   EXPONENT MAX EXPONENT SIZE   EXPORTS   EXTENT   FILLER MIN   FIXED   FIXED MAX   FIXED MIN FIXED REP   FIXED STD   FLOAT   FLOAT MAX FLOAT MIN   FLOAT STD   HANDLER EXISTS IMPORTS   IMP PRECISION   INTEGER MAX   LENGTH   IS MANIFEST MANTISSA SIZE   NEW   NOM PRECISION OBJECT MACHINE   OMIT   PACKING   PREC PRECISION   PRED   PRIORITY   PTR   PTR STD RADIX   RECORD REP   REGION   RELEASE REPTYPE   REQUEST   RETURNS   ROUTINES SCALE   SEMAPHORE   SIGNALS   SIGNING SIZE   SUCC   SYNCH   TARGET   TO BIT TO FIXED   TO FLOAT   TYPES   UNORDERED USAGE   USES   VALUES   VARIANTS   WAIT ZEROS
[2-15/7]	comment	::= embedded-comment   terminating-comment
[2-16/7]	embedded-comment	::= % character... %
[2-17/7]	terminating-comment	::= % character... new-line
[3-1/3]	declaration-stmt	::= constant-declaration   variable-declaration   type-declaration   reptype-declaration
[3-2/4]	attribute-query	::= variable . attribute-name   constant . attribute-name   (type-name . attribute-name) . attribute-name   (expression) . attribute-name

```

[3-3/4]    attribute-name   ::= fixed-attr
            | fixed-rep-attr
            | float-attr
            | float-rep-attr
            | enumerated-attr
            | enumerated-rep-attr
            | boolean-attr
            | boolean-rep-attr
            | bitstring-attr
            | bitstring-rep-attr
            | charstring-attr
            | charstring-rep-attr
            | array-attr
            | array-rep-attr
            | record-attr
            | record-rep-attr
            | pointer-attr
            | pointer-rep-attr
            | semaphore-attr
            | semaphore-rep-attr
            | defined-attr
            | defined-rep-attr

[3-4/6]    constant-declaration ::= CONST simple-constant,... : type-spec
                                    ::= expression;

[3-5/6]    simple-constant    ::= identifier

[3-6/8]    variable-declaration ::= VAR simple-variable,... : type-spec
                                    [ ::= expression];

[3-7/8]    simple-variable    ::= identifier

[3-8/9]    type-spec          ::= type-reference [REP retype-spec]
                                    | predefined-type [REP retype-spec]
                                    | array-spec

[3-9/9]    predefined-type    ::= fixed-spec
                                    | float-spec
                                    | enumerated-spec
                                    | boolean-spec
                                    | bitstring-spec
                                    | charstring-spec
                                    | record-spec
                                    | pointer-spec
                                    | semaphore-spec

[3-10/11]   literal           ::= fixed-literal
                                    | float-literal
                                    | enumerated-literal
                                    | boolean-literal
                                    | bitstring-literal
                                    | charstring-literal
                                    | pointer-literal

```

[3-11/11]	constructor	::= array-constructor   record-constructor   pointer-constructor   semaphore-constructor
[3-12/12]	reptype-spec	::= reptype-reference   predefined-reptype
[3-13/12]	predefined-reptype	::= fixed-reptype   float-reptype   enumerated-reptype   boolean-reptype   bitstring-reptype   charstring-reptype   array-reptype   record-reptype   pointer-reptype   semaphore-reptype
[3-14/14]	manifest-expression	::= expression
[3-15/14]	expression	::= simple-expression [rounding-mode] [WITH PREC manifest-integer-expression]
[3-16/14]	rounding-mode	::= enumerated-literal
[3-17/14]	simple-expression	::= conjunction   simple-expression simple-exp-ops conjunction
[3-18/14]	simple-exp-ops	::= OR!!!XOR!->
[3-19/14]	conjunction	::= relation   conjunction conj-ops relation
[3-20/14]	conj-ops	::= AND!&
[3-21/14]	relation	::= sum   relation rel-ops sum   sum IN range-spec
[3-22/14]	rel-ops	::= = < > <= > = <> IN
[3-23/14]	sum	::= term   sum sum-ops term
[3-24/14]	sum-ops	::= + -  !   !+!   !-!   (!+  -!) manifest-integer-expression!
[3-25/14]	term	::= factor   term term-ops factor

```

[3-26/14] term-ops ::= *|/|//|MOD
| {!*|!//} manifest-integer-expression!
| !MOD manifest-integer-expression!
| !/
| !*
| !//!

[3-27/15] factor ::= secondary
| factor factor-ops secondary

[3-28/15] factor-ops ::= **
| !** manifest-integer-expression!

[3-29/15] secondary ::= primary
| unary-ops primary

[3-30/15] unary-ops ::= +|-|NOT

[3-31/15] primary ::= variable
| constant
| literal
| constructor
| substring-reference
| predefined-function
| function-invocation
| (simple-expression)
| attribute-query
| type-specifier
| explicit-rep-converter

[3-32/15] variable ::= simple-variable
| subscripted-variable
| field-reference
| primary @

[3-33/15] constant ::= simple-constant
| subscripted-constant
| field-constant

[3-34/15] substring-reference ::= variable closed-range
| constant closed-range

[3-35/15] range-spec ::= [!{} lower-bound : upper-bound {}]

[3-36/15] closed-range ::= [lower-bound : upper-bound]

[3-37/15] lower-bound ::= expression

[3-38/15] upper-bound ::= expression

[3-39/15] manifest-range-spec ::= range-spec

[3-40/15] type-specifier ::= [type-spec]$$(expression)

[3-41/15] explicit-rep-converter ::= retype-spec (expression)

```

```

[3-42/21] assignment-stmt ::= general-assignment
                                | fixed-assignment
                                | float-assignment
                                | enumerated-assignment
                                | boolean-assignment
                                | bitstring-assignment
                                | charstring-assignment
                                | array-assignment
                                | record-assignment
                                | pointer-assignment
                                | semaphore-assignment

[3-43/21] general-assignment ::= variable [closed-range] := expression;

[3-44/22] predefined-function ::= fixed-abs-function
                                 | to-fixed-function
                                 | fixed-min-function
                                 | fixed-max-function
                                 | float-abs-function
                                 | to-float-function
                                 | float-imp-prec-function
                                 | float-min-function
                                 | float-max-function
                                 | float-epsilon-function
                                 | enumerated-pred-function
                                 | enumerated-succ-function
                                 | handler-exists-function
                                 | manifest-function
                                 | bitstring-bin-function
                                 | bitstring-dup-function
                                 | bitstring-hex-function
                                 | bitstring-oct-function
                                 | bitstring-to-bit-function
                                 | bitstring-zeros-function
                                 | charstring-blanks-function
                                 | charstring-dup-function
                                 | clock-function

[3-45/23] fixed-spec ::= FIXED (fixed-scale, radix, fixed-prec)
                           | [range-spec] [REP fixed-reptype]
                           | INTEGER [range-spec] [REP fixed-reptype]

[3-46/23] fixed-scale ::= manifest-integer-expression

[3-47/23] radix ::= manifest-integer-expression

[3-48/23] fixed-prec ::= manifest-enumerated-expression

[3-49/23] fixed-attr ::= SCALE
                           | RADIX
                           | PRECISION
                           | MIN
                           | MAX

```

[3-50/26]	<b>fixed-literal</b>	<code>::= decimal-literal scale-spec   [+ -] number [exponent-spec]</code>
[3-51/26]	<b>decimal-literal</b>	<code>::= [+ -] number . [number] [exponent-spec]   [number] . number [exponent-spec]</code>
[3-52/26]	<b>exponent-spec</b>	<code>::= E [+ -] number</code>
[3-53/26]	<b>scale-spec</b>	<code>::= \S [+ -] number</code>
[3-54/26]	<b>number</b>	<code>::= digit...</code>
[3-55/28]	<b>fixed-reptype</b>	<code>::= FIXED REP (size, signing)   FIXED STD</code>
[3-56/28]	<b>size</b>	<code>::= manifest-integer-expression</code>
[3-57/28]	<b>signing</b>	<code>::= manifest-enumerated-expression</code>
[3-58/28]	<b>fixed-rep-attr</b>	<code>::= SIZE   SIGNING</code>
[3-59/30]	<b>fixed-expression</b>	<code>::= expression</code>
[3-60/30]	<b>integer-expression</b>	<code>::= fixed-expression</code>
[3-61/30]	<b>manifest-integer-expression</b>	<code>::= manifest-expression</code>
[3-62/39]	<b>fixed-assignment</b>	<code>::= variable := fixed-expression;</code>
[3-63/41]	<b>to-fixed-function</b>	<code>::= TO_FIXED (to-fixed-value, fixed-scale [, radix] [, fixed-prec] [rounding-mode])</code>
[3-64/41]	<b>to-fixed-value</b>	<code>::= fixed-expression   float-expression   enumerated-expression   boolean-expression   bitstring-expression   charstring-expression</code>
[3-65/44]	<b>fixed-abs-function</b>	<code>::= ABS (fixed-expression)</code>
[3-66/45]	<b>fixed-min-function</b>	<code>::= FIXED_MIN (fixed-scale, radix, fixed-prec)</code>
[3-67/45]	<b>fixed-max-function</b>	<code>::= FIXED_MAX (fixed-scale, radix, fixed-prec)</code>
[3-68/47]	<b>float-spec</b>	<code>::= FLOAT (float-precision) [range-spec]</code>
[3-69/47]	<b>float-precision</b>	<code>::= manifest-integer-expression</code>

```

[3-70/47] float-attr ::= NOM_PRECISION
| IMP_PRECISION
| RADIX
| MIN
| MAX
| EXPONENT_MIN
| EXPONENT_MAX

[3-71/49] float-literal ::= decimal-literal

[3-72/50] float-reptype ::= FLOAT_STD

[3-73/50] float-rep-attr ::= SIZE
| MANTISSA_SIZE
| EXPONENT_SIZE

[3-74/51] float-expression ::= expression

[3-75/57] float-assignment ::= variable := float-expression;

[3-76/59] to-float-function ::= TO_FLOAT (to-float-value, float-precision
[, rounding-mode])

[3-77/59] to-float-value ::= fixed-expression
| float-expression
| charstring-expression

[3-78/62] float-abs-function ::= ABS (float-expression)

[3-79/62] float-min-function ::= FLOAT_MIN (float-precision)

[3-80/62] float-max-function ::= FLOAT_MAX (float-precision)

[3-81/63] float-epsilon-function ::= EPSILON (float-precision)

[3-82/64] float-imp-prec-function ::= IMP_PRECISION (float-precision)

[3-83/65] enumerated-spec ::= ENUM (enumerated-literal,...) [range-spec]
| UNORDERED (enumerated-literal,...)

[3-84/65] enumerated-attr ::= MIN
| MAX
| EXTENT

[3-85/68] enumerated-literal ::= enumerated-id
| printing-char-id
| non-printing-char-id

[3-86/68] enumerated-id ::= identifier

[3-87/68] printing-char-id ::= \'character-no-apos'
| ````'

[3-88/68] non-printing-char-id ::= note: an enumerated-id whose first character
is \

```

[3-89/68] character-no-apos ::= note: all elements of character except the apostrophe

[3-90/69] enumerated-reptype ::= ENUM REP (size)  
| ENUM\_VAL REP (size,  
VALUES (value-spec,...));  
| ENUM\_STD

[3-91/69] value-spec ::= enumerated-literal AS  
manifest-bitstring-expression

[3-92/69] enumerated-rep-attr ::= SIZE

[3-93/71] enumerated-expression ::= expression

[3-94/71] manifest-enumerated-expression ::= manifest-expression

[3-95/73] enumerated-assignment ::= variable := enumerated-expression;

[3-96/74] enumerated-succ-function ::= SUCC (enumerated-expression)

[3-97/74] enumerated-pred-function ::= PRED (enumerated-expression)

[3-98/75] boolean-spec ::= BOOLEAN

[3-99/75] boolean-attr ::= note: there are none

[3-100/75] boolean-literal ::= TRUE  
| FALSE

[3-101/75] boolean-reptype ::= BOOLEAN\_STD

[3-102/75] boolean-rep-attr ::= SIZE

[3-103/76] boolean-expression ::= expression

[3-104/76] boolean-manifest-expression ::= manifest-expression

[3-105/79] boolean-assignment ::= variable := boolean-expression;

[3-106/79] handler-exists-function ::= HANDLER\_EXISTS (exception-name)

[3-107/80] manifest-function ::= IS\_MANIFEST(expression)

[3-108/81] bitstring-spec ::= BIT (length-expression)

[3-109/81] length-expression ::= integer-expression

[3-110/81] bitstring-attr ::= LENGTH

```

[3-111/82] bitstring-literal ::= BIN [display-char]...
                                {bin-string [display-char]...} ...
                                | OCT [display-char]...
                                {oct-string [display-char]...} ...
                                | HEX [display-char]...
                                {hex-string [display-char]...} ...

[3-112/82] bin-string ::= '[bin-char|blank]...''

[3-113/82] oct-string ::= '[oct-char|blank]...''

[3-114/82] hex-string ::= '[hex-char|blank]''''

[3-115/82] bin-char ::= 0|1

[3-116/82] oct-char ::= 0|1|2|3|4|5|6|7

[3-117/82] hex-char ::= 0|1|2|3|4|5|6|7|8|9|A|B|C|D|E|F

[3-118/83] bitstring-reptype ::= BIT_STD

[3-119/83] bitstring-rep-attr ::= SIZE

[3-120/84] bitstring-expression ::= expression

[3-121/84] manifest-bitstring-expression ::= manifest-expression

[3-122/86] bitstring-assignment ::= bit-target := bitstring-expression;

[3-123/86] bit-target ::= variable
                           | variable closed-range

[3-124/87] bitstring-to-bit-function ::= TO_BIT (to-bit-value)

[3-125/87] to-bit-value ::= expression

[3-126/87] bitstring-dup-function ::= DUP (bitstring-expression,
                                             integer-expression)

[3-127/88] bitstring-zeros-function ::= ZEROS (integer-expression)

[3-128/88] bitstring-bin-function ::= BIN_OF (charstring-expression)

[3-129/88] bitstring-oct-function ::= OCT_OF (charstring-expression)

[3-130/88] bitstring-hex-function ::= HEX_OF (charstring-expression)

[3-131/90] charstring-spec ::= CHAR_STR (length-expression, charset-name)

[3-132/90] charset-name ::= enumerated-type

[3-133/90] enumerated-type ::= type-reference

[3-134/90] charstring-attr ::= LENGTH
                               | CHARSET

```

```

[3-135/92] charstring-literal ::= [charset-name] [display-char]...
                                {basic-string [display-char]...} ...

[3-136/92] basic-string      ::= '['basic-char]...
                                | enumerated-literal

[3-137/92] basic-char       ::= ''
                                | printing-char

[3-138/92] printing-char    ::= note: any character, C, appearing in a
                                printing-char-id, i.e.,
                                a literal of the form '\C'

[3-139/97] charstring-reptype ::= CHAR_STR REP (packing-mode)
                                | CHAR_STR STD

[3-140/97] packing-mode     ::= manifest-enumerated-expression

[3-141/97] charstring-rep-attr ::= PACKING
                                | SIZE

[3-142/99] charstring-expression ::= expression

[3-143/100] charstring-assignment ::= char-target := charstring-expression;

[3-144/100] char-target       ::= variable
                                | variable closed-range

[3-145/102] charstring-dup-function ::= DUP (charstring-expression,
                                         integer-expression)

[3-146/102] charstring-blanks-function ::= BLANKS (integer-expression)

[3-147/104] array-spec        ::= ARRAY [array-bound,...]
                                [REP array-reptype]
                                OF type-spec

[3-148/104] array-bound      ::= integer-expression : integer-expression
                                | enumerated-expression : enumerated-expression

[3-149/104] array-attr        ::= DIMENSIONS
                                | BOUNDS [(index) [. {MIN : MAX}]]
                                | EXTENT [(index)]
                                | ELEMENT_TYPE

[3-150/104] index             ::= integer-expression

[3-151/107] array-constructor ::= (non-array-expression,...)
                                | (array-constructor,...)

[3-152/107] non-array-expression ::= expression

[3-153/109] array-reptype    ::= ARRAY_STD
                                | ARRAY REP (packing-mode)

```

```

[3-154/109] array-rep-attr      ::= SIZE
                                | PACKING

[3-155/111] array-expression    ::= expression

[3-156/111] subscripted-variable ::= variable (subscript,...)

[3-157/111] subscripted-constant ::= constant (subscript,...)

[3-158/111] subscript          ::= expression

[3-159/113] array-assignment    ::= variable := array-expression;

[3-160/114] record-spec         ::= RECORD [field-declarations]...
                                         END RECORD

[3-161/114] field-declarations  ::= [var-field-spec]... [variant-part]...

[3-162/114] var-field-spec      ::= VAR field-name,... : type-spec ;

[3-163/114] field-name          ::= identifier

[3-164/115] variant-part        ::= SELECT tag-name FROM
                                         [variant-spec]...
                                         END SELECT;

[3-165/115] variant-spec        ::= variant-name,... -->
                                         [field-declarations]...
                                         END;

[3-166/115] variant-name        ::= manifest-expression
                                         | manifest-range-spec

[3-167/115] record-type-declaration ::= TYPE record-type-name
                                         [(record-parameter,...)] =
                                         record-type-definition;

[3-168/115] record-type-name     ::= identifier

[3-169/115] record-parameter     ::= tag-field-spec
                                         | const-field-spec
                                         | type-parameter

[3-170/115] tag-field-spec       ::= TAG tag-name,... : type-spec

[3-171/115] tag-name             ::= identifier

[3-172/115] const-field-spec     ::= field-name,... : type-spec

[3-173/115] type-parameter       ::= identifier,... : TYPE

[3-174/115] record-type-definition ::= record-spec

[3-175/115] record-type-reference ::= record-type-name [(record-argument,...)]

```

```

[3-176/115] record-argument      ::= expression
| ANY
| type-spec

[3-177/115] record-attr          ::= note: only record fields have attributes

[3-178/124] record-constructor   ::= [record-type-specifier]
| ([var-field-init,...])

[3-179/124] record-type-specifier ::= [type-spec]$>

[3-180/124] var-field-init        ::= field-name: expression

[3-181/127] record-reptype       ::= RECORD REP ([common-packing-spec]
| [variant-packing]...)
| RECORD STD

[3-182/127] common-packing-spec  ::= packing-spec, ...

[3-183/127] packing-spec         ::= [packing-mode] [(element-spec,...)]

[3-184/127] element-spec         ::= field-name
| tag-name
| filler-spec
| VARIANTS (tag-name)
| OMIT ({field-name | tag-name})

[3-185/127] filler-spec          ::= FILLER (manifest-integer-expression)
| ZEROS (manifest-integer-expression)
| BLANKS (manifest-integer-expression)

[3-186/127] variant-packing      ::= SELECT tag-name FROM
| [variant-packing-spec]...
| [else-packing-spec]
END SELECT;

[3-187/127] variant-packing-spec ::= variant-name,... -->
| [packing-spec,...]
END;

[3-188/127] else-packing-spec    ::= ELSE --> [packing-mode] END;

[3-189/127] record-rep-attr       ::= SIZE

[3-190/133] record-expression     ::= expression

[3-191/133] field-reference       ::= variable . field-id

[3-192/133] field-constant        ::= constant . field-id

[3-193/133] field-id              ::= field-name
| tag-name

[3-194/136] record-assignment      ::= variable := record-expression ;

```

[3-195/138] pointer-spec	::= PTR (type-spec)
[3-196/138] pointer-attr	::= ELEMENT_TYPE
[3-197/139] pointer-literal	::= NIL
[3-198/139] pointer-constructor	::= [PTR\$] NEW (type-spec)
[3-199/139] pointer-reptype	::= PTR_STD
[3-200/139] pointer-rep-attr	::= SIZE
[3-201/140] pointer-expression	::= expression
[3-202/142] pointer-assignment	::= variable := pointer-expression;
[3-203/143] semaphore-spec	::= SEMAPHORE (use-mode)
[3-204/143] use-mode	::= manifest-enumerated-expression
[3-205/143] semaphore-attr	::= USAGE
[3-206/143] semaphore-constructor	::= [SEMAPHORE\$] NEW (integer-expression)
[3-207/144] semaphore-reptype	::= SEMAPHORE_STD
[3-208/144] semaphore-rep-attr	::= SIZE
[3-209/144] semaphore-expression	::= expression
[3-210/145] semaphore-assignment	::= variable := semaphore-expression;
[3-211/147] type-declaration	::= TYPE type-name [(type-formal,...)] = type-definition;   record-type-declaration
[3-212/148] type-name	::= identifier
[3-213/148] type-formal	::= identifier,... : {type-spec   TYPE}   TAG tag-name,... : type-spec
[3-214/148] type-definition	::= type-spec
[3-215/150] type-reference	::= type-instantiation [range-spec]   record-type-reference
[3-216/150] type-instantiation	::= type-id [(type-argument,...)]
[3-217/150] type-id	::= abstract-type-name   abbrev-type-name
[3-218/150] abstract-type-name	::= type-name
[3-219/150] abbrev-type-name	::= type-name

[3-220/150] type-argument	::= type-spec   expression   ANY
[3-221/153] retype-declaration	::= REPTYPE retype-name [(type-formal,...)] [OF base-formal-id] = retype-definition;
[3-222/153] retype-name	::= identifier
[3-223/153] base-formal-id	::= identifier
[3-224/153] retype-definition	::= retype-spec
[3-225/155] retype-reference	::= retype-name [(type-argument,...)]
[3-226/157] defined-attr	::= type-formal-id
[3-227/157] type-formal-id	::= identifier
[3-228/157] defined-rep-attr	::= SIZE
[4-1/3] open-scope-body	::= [declaration-stmt]... [executable-stmt]...
[4-2/3] closed-scope-body	::= [declaration-stmt]... [executable-stmt]... [routines-and-modules]...
[4-3/3] routines-and-modules	::= routine-declaration   module-declaration
[4-4/7] segment-declaration	::= segment-header segment-body segment-trailer;
[4-5/7] segment-header	::= [MAIN] SEGMENT segment-name [uses-clause] [defines-clause];
[4-6/7] uses-clause	::= USES ({segment-name[item-list]}),...
[4-7/7] item-list	::= ([DATA : ([input-items][#inout-items])] [ROUTINES : (routine-name,...)] [TYPES : (type-name,...)] [MODULES : (module-name,...)])
[4-8/7] input-items	::= {simple-variable   simple-constant},...
[4-9/7] inout-items	::= simple-variable,...
[4-10/7] routine-name	::= proc-name   func-name
[4-11/7] defines-clause	::= DEFINES {(name,...)   ALL}

```

[4-12/7]    name          ::= identifier
[4-13/7]    segment-body   ::= closed-scope-body
[4-14/7]    segment-trailer ::= END SEGMENT segment-name
[4-15/7]    segment-name   ::= identifier
[4-16/11]   routine-declaration ::= procedure-declaration
                                | function-declaration
                                | target-routine-declaration
[4-17/11]   procedure-declaration ::= procedure-header
                                    procedure-body
                                    procedure-trailer;
[4-18/11]   procedure-header   ::= [INLINE] PROCEDURE proc-name
                                proc-formal-parm-list
                                [signals-clause]
                                [imports-clause]
                                [where-clause];
[4-19/11]   procedure-body    ::= closed-scope-body
[4-20/11]   procedure-trailer  ::= END PROCEDURE proc-name
[4-21/11]   proc-formal-parm-list ::= ([input-parameters] [#inout-parameters]
                                         [#output-parameters])
[4-22/11]   input-parameters  ::= formal-parameter, ...
[4-23/11]   inout-parameters  ::= formal-parameter, ...
[4-24/11]   output-parameters ::= formal-parameter, ...
[4-25/11]   formal-parameter   ::= formal-name, ...
                                {type-spec | routine-spec | TYPE}
[4-26/11]   proc-name         ::= [type-name$] identifier
[4-27/11]   formal-name       ::= identifier
[4-28/11]   imports-clause    ::= IMPORTS item-list
[4-29/12]   signals-clause    ::= SIGNALS (exception-name,...)
[4-30/12]   function-declaration ::= function-header
                                function-body
                                function-trailer;

```

```

[4-31/12] function-header ::= [INLINE] FUNCTION func-name
                           func-formal-parm-list
                           returns-clause
                           [signals-clause]
                           [imports-clause]
                           [where-clause];

[4-32/12] function-body ::= closed-scope-body

[4-33/12] function-trailer ::= END FUNCTION func-name

[4-34/12] func-formal-parm-list ::= ([input-parameters])

[4-35/12] returns-clause ::= RETURNS (type-spec)

[4-36/12] func-name ::= [type-name$] identifier

[4-37/15] procedure-stmt ::= proc-name proc-argument-list;

[4-38/15] proc-argument-list ::= ([input-arguments] [#inout-arguments]
                                    [#output-arguments])

[4-39/15] input-arguments ::= {expression | routine-argument | type-spec},...

[4-40/15] inout-arguments ::= variable, ...

[4-41/15] output-arguments ::= variable, ...

[4-42/15] function-invocation ::= func-name func-argument-list

[4-43/15] func-argument-list ::= ([input-arguments])

[4-44/15] return-stmt ::= RETURN [expression];

[4-45/23] module-declaration ::= module-header
                                 module-body
                                 module-trailer;

[4-46/23] module-header ::= MODULE module-name
                           [imports-clause]
                           [exports-clause];

[4-47/23] module-body ::= closed-scope-body

[4-48/23] module-trailer ::= END MODULE module-name

[4-49/23] module-name ::= identifier

[4-50/23] exports-clause ::= EXPORTS ({name[WITH(routine-name,...)]},...)
                           | EXPORTS ALL

[4-51/29] executable-stmt ::= [label :]... unlabeled-stmt
                            [exceptions-clause]

```

```

[4-52/29]  unlabeled-stmt      ::= simple-stmt
                           | structured-stmt

[4-53/29]  label             ::= identifier

[4-54/31]  simple-stmt       ::= assignment-stmt
                           | procedure-stmt
                           | return-stmt
                           | exit-stmt
                           | signal-stmt
                           | go-to-stmt
                           | assert-stmt
                           | null-stmt
                           | parallel-control-stmt
                           | start-io-stmt

[4-55/31]  null-stmt         ::= ;

[4-56/31]  go-to-stmt        ::= GO TO label;

[4-57/31]  parallel-control-stmt ::= wait-stmt
                           | terminate-stmt
                           | priority-stmt
                           | request-stmt
                           | release-stmt

[4-58/32]  structured-stmt   ::= block
                           | conditional-stmt
                           | repetitive-stmt
                           | parallel-block
                           | with-clause structured-stmt

[4-59/32]  block             ::= BLOCK open-scope-body END BLOCK;

[4-60/33]  with-clause        ::= WITH bound-var := expression
                           | WITH bound-var NAMING variable

[4-61/33]  bound-var          ::= identifier

[4-62/35]  conditional-stmt   ::= if-stmt
                           | case-stmt

[4-63/35]  if-stmt            ::= if-element [orif-element]...
                           | [if-else-element] END IF;

[4-64/35]  if-element          ::= IF boolean-expression THEN open-scope-body

[4-65/35]  orif-element        ::= ORIF boolean-expression THEN open-scope-body

[4-66/35]  if-else-element     ::= ELSE open-scope-body

[4-67/36]  case-stmt          ::= SELECT choice FROM [case-body] END SELECT;

```

[4-68/36]	choice	::= expression   bound-var . tag-name   OBJECT_MACHINE
[4-69/36]	case-body	::= case-element... [case-else-element]
[4-70/36]	case-element	::= choice-label --> open-scope-body END; [exceptions-clause]
[4-71/37]	case-else-element	::= ELSE --> open-scope-body END; [exceptions-clause]
[4-72/37]	choice-label	::= case-label,...   variant-name,...   machine-config,...
[4-73/37]	case-label	::= manifest-expression   manifest-range-spec
[4-74/37]	machine-config	::= identifier
[4-75/40]	repetitive-stmt	::= indefinite-loop-stmt   definite-loop-stmt
[4-76/40]	indefinite-loop-stmt	::= LOOP [UNTIL predicate DO] open-scope-body END LOOP [WHEN predicate];
[4-77/40]	predicate	::= boolean-expression
[4-78/42]	exit-stmt	::= EXIT;
[4-79/42]	definite-loop-stmt	::= LOOP FOR [identifier IN] range-spec DO open-scope-body END LOOP [identifier];
[4-80/45]	exceptions-clause	::= EXCEPT [named-handler]... END EXCEPT;
[4-81/45]	named-handler	::= exception-name,... --> [open-scope-body] END;
[4-82/45]	exception-name	::= identifier
[4-83/45]	signal-stmt	::= SIGNAL [exception-name];
[4-84/51]	assert-stmt	::= ASSERT boolean-expression;
[4-85/54]	parallel-block	::= PAR_BLOCK parallel-path... END PAR_BLOCK;
[4-86/54]	parallel-path	::= path-header path-body path-trailer;

[4-87/54] path-header	::= PATH path-name [loop-prefix] [PRIORITY priority-expr] [CONNECT (interrupt)] [signals-clause] [imports-clause];
[4-88/54] path-body	::= closed-scope-body
[4-89/54] path-trailer	::= END PATH path-name
[4-90/54] loop-prefix	::= FOR identifier IN range-spec
[4-91/54] path-name	::= identifier
[4-92/54] priority-expr	::= integer-expression
[4-93/54] interrupt	::= semaphore-name
[4-94/54] semaphore-name	::= identifier
[4-95/57] wait-stmt	::= WAIT (time) ;
[4-96/57] time	::= fixed-expression
[4-97/59] terminate-stmt	::= TERMINATE [(path-reference)];
[4-98/59] path-reference	::= path-name [(subscript)]
[4-99/61] priority-stmt	::= PRIORITY (priority-expr [, path-reference]);
[4-100/62] clock-function	::= CLOCK [(path-reference)]
[4-101/63] request-stmt	::= REQUEST (#semaphore-variable);
[4-102/63] semaphore-variable	::= variable
[4-103/64] release-stmt	::= RELEASE (#semaphore-variable);
[4-104/71] start-io-stmt	::= START_IO (device-object, device-operation [, input-parameters] [#inout-parameters] [#output-parameters])
[4-105/71] device-object	::= expression
[4-106/71] device-operation	::= expression
[4-107/74] target-routine-declaration	::= target-procedure-declaration   target-function-declaration
[4-108/74] target-procedure-declaration	::= TARGET procedure-header target-procedure-body procedure-trailer;
[4-109/74] target-procedure-body	::= DYNSIZE (stack-size); target-scope-body

```

[4-110/74] target-scope-body     ::= note: target computer assembly code
[4-111/74] stack-size          ::= manifest-integer-expression
[4-112/74] target-function-declaration ::= TARGET function-header
                                         target-function-body
                                         function-trailer;
[4-113/74] target-function-body  ::= DYN SIZE (stack-size);
                                         target-scope-body
[5-1/5]   routine-spec         ::= procedure-spec | function-spec
[5-2/5]   procedure-spec       ::= PROCEDURE proc-template
[5-3/5]   proc-template        ::= ([type-spec,...][#[type-spec,...]
                                         #[type-spec,...]])
                                         [signals-clause]
[5-4/5]   function-spec        ::= FUNCTION func-template
[5-5/5]   func-template        ::= ([type-spec,...]) RETURNS (type-spec)
                                         [signals-clause]
[5-6/5]   routine-template     ::= proc-template | func-template
[5-7/5]   routine-argument     ::= procedure-argument
                                         | function-argument
[5-8/5]   procedure-argument   ::= proc-name [proc-template]
[5-9/5]   function-argument    ::= func-name [func-template]
[5-10/8]  where-clause         ::= WHERE needs-spec, ...
[5-11/8]  needs-spec           ::= type-name NEEDS (template-spec, ...)
[5-12/8]  template-spec        ::= routine-name routine-template

```

**APPENDIX B**  
**Syntax Cross-Reference Index**

The following listing shows each terminal and non-terminal element of the language's syntax. The references in brackets first give the number of the syntax production in which the element was used or defined and then the page number on which the production appears. For example, [3-1/3] means that production 3-1 is found on page 3-3 of the specification.

A "d" appearing before a production reference indicates that the non-terminal element is defined in that production. For example declaration-stmt is defined in production 3-1.

Open and closing parentheses are listed together as "( )", as are open and closing brackets, [ ].

The following non-terminal elements are not defined: lexical-program, segment-declaration, and boolean-manifest-expression. A lexical-program represents the sequence of characters comprising a compilable unit in the language. A segment-declaration is the starting symbol of the syntax and represents the language's compilable units. Boolean-manifest-expression appears in the syntax merely for expository convenience in explaining the constraints defining when a boolean-expression is to be considered manifest and what the effect of using such an expression is in an if-stmt.

```

! ..... [2-12/4], [2-4/1], [3-18/14], [3-24/14],
[3-26/14], [3-28/15]
!! ..... [2-12/4], [3-24/14]
!* ..... [2-12/4], [3-26/14]
!*! ..... [2-12/4], [3-26/14]
!*# ..... [3-28/15]
!+ ..... [2-12/4], [3-24/14]
!+! ..... [2-12/4], [3-24/14]
!- ..... [2-12/4], [3-24/14]
!-! ..... [2-12/4], [3-24/14]
!/! ..... [2-12/4], [3-26/14]
!// ..... [2-12/4], [3-26/14]
!//! ..... [2-12/4], [3-26/14]
!MOD ..... [2-12/4], [3-26/14]
!MOD! ..... [2-12/4]
" ..... [2-12/4], [2-4/1]
# ..... [2-12/4], [2-4/1], [4-101/63], [4-103/64],
[4-104/71], [4-21/11], [4-38/15], [4-7/7], [5-3/5]
$ ..... [2-12/4], [2-4/1], [3-198/139], [3-206/143],
[4-26/11], [4-36/12]
$$ ..... [2-12/4], [3-179/124], [3-40/15]
% ..... [2-12/4], [2-16/7], [2-17/7], [2-4/1]
& ..... [2-12/4], [2-4/1], [3-20/14]
' ..... [2-12/4], [2-4/1], [3-112/82], [3-113/82],
[3-114/82], [3-136/92], [3-137/92], [3-87/68]
( ) ..... [2-12/4], [2-4/1], [3-106/79], [3-107/80],
[3-108/81], [3-124/87], [3-126/87], [3-127/88], [3-128/88],
[3-129/88], [3-130/88], [3-131/90], [3-139/97], [3-145/102],

```

[3-146/102], [3-149/104], [3-151/107], [3-153/109],  
[3-156/111], [3-157/111], [3-167/115], [3-175/115],  
[3-178/124], [3-181/127], [3-183/127], [3-184/127],  
[3-185/127], [3-195/138], [3-198/139], [3-2/4], [3-203/143],  
[3-206/143], [3-211/147], [3-216/150], [3-221/153],  
[3-225/155], [3-31/15], [3-35/15], [3-40/15], [3-41/15],  
[3-45/23], [3-55/28], [3-63/41], [3-65/44], [3-66/45],  
[3-67/45], [3-68/47], [3-76/59], [3-78/62], [3-79/62],  
[3-80/62], [3-81/63], [3-82/64], [3-83/65], [3-90/69],  
[3-96/74], [3-97/74], [4-100/62], [4-101/63], [4-103/64],  
[4-104/71], [4-109/74], [4-11/7], [4-113/74], [4-21/11],  
[4-29/12], [4-34/12], [4-35/12], [4-38/15], [4-43/15],  
[4-50/23], [4-6/7], [4-7/7], [4-87/54], [4-95/57], [4-97/59],  
[4-98/59], [4-99/61], [5-11/8], [5-3/5], [5-5/5]  
\* ..... [2-12/4], [2-4/1], [3-26/14]  
\*\* ..... [2-12/4], [3-28/15]  
+ ..... [2-12/4], [2-4/1], [3-24/14], [3-30/15],  
[3-50/26], [3-51/26], [3-52/26], [3-53/26]  
, ..... [2-12/4], [2-4/1], [3-126/87], [3-131/90],  
[3-145/102], [3-147/104], [3-151/107], [3-156/111],  
[3-157/111], [3-162/114], [3-165/115], [3-167/115],  
[3-170/115], [3-172/115], [3-173/115], [3-175/115],  
[3-178/124], [3-182/127], [3-183/127], [3-187/127],  
[3-211/147], [3-213/148], [3-216/150], [3-221/153],  
[3-225/155], [3-4/6], [3-45/23], [3-55/28], [3-6/8], [3-63/41],  
[3-66/45], [3-67/45], [3-76/59], [3-83/65], [3-90/69],  
[4-104/71], [4-11/7], [4-22/11], [4-23/11], [4-24/11],  
[4-25/11], [4-29/12], [4-39/15], [4-40/15], [4-41/15],  
[4-50/23], [4-6/7], [4-7/7], [4-72/37], [4-8/7], [4-81/45],  
[4-9/7], [4-99/61], [5-10/8], [5-11/8], [5-3/5], [5-5/5]  
- ..... [2-12/4], [2-4/1], [3-24/14], [3-30/15],  
[3-50/26], [3-51/26], [3-52/26], [3-53/26]  
--> ..... [2-12/4], [3-165/115], [3-187/127],  
[3-188/127], [4-70/36], [4-71/37], [4-81/45]  
-> ..... [2-12/4], [3-18/14]  
. ..... [2-12/4], [2-4/1], [3-149/104], [3-191/133],  
[3-192/133], [3-2/4], [3-51/26], [4-68/36]  
/ ..... [2-12/4], [2-4/1], [3-26/14]  
// ..... [2-12/4], [3-26/14]  
0 ..... [2-3/1], [3-115/82], [3-116/82], [3-117/82]  
1 ..... [2-3/1], [3-115/82], [3-116/82], [3-117/82]  
2 ..... [2-3/1], [3-116/82], [3-117/82]  
3 ..... [2-3/1], [3-116/82], [3-117/82]  
4 ..... [2-3/1], [3-116/82], [3-117/82]  
5 ..... [2-3/1], [3-116/82], [3-117/82]  
6 ..... [2-3/1], [3-116/82], [3-117/82]  
7 ..... [2-3/1], [3-116/82], [3-117/82]  
8 ..... [2-3/1], [3-117/82]  
9 ..... [2-3/1], [3-117/82]  
: ..... [2-12/4], [2-4/1], [3-148/104], [3-162/114],  
[3-170/115], [3-172/115], [3-173/115], [3-180/124],  
[3-213/148], [3-35/15], [3-36/15], [3-4/6], [3-6/8], [4-25/11],  
[4-51/29], [4-7/7]  
:= ..... [2-12/4], [3-105/79], [3-122/86], [3-143/100],  
[3-159/113], [3-194/136], [3-202/142], [3-210/145], [3-4/6],

[3-43/21], [3-6/8], [3-62/39], [3-75/57], [3-95/73], [4-60/33]  
 ; ..... [2-12/4], [2-4/1], [3-105/79], [3-122/86],  
   [3-143/100], [3-159/113], [3-162/114], [3-164/115],  
   [3-165/115], [3-167/115], [3-186/127], [3-187/127],  
   [3-188/127], [3-194/136], [3-202/142], [3-210/145],  
   [3-211/147], [3-221/153], [3-4/6], [3-43/21], [3-6/8],  
   [3-62/39], [3-75/57], [3-90/69], [3-95/73], [4-101/63],  
   [4-103/64], [4-108/74], [4-109/74], [4-112/74], [4-113/74],  
   [4-17/11], [4-18/11], [4-30/12], [4-31/12], [4-37/15], [4-4/7],  
   [4-44/15], [4-45/23], [4-46/23], [4-5/7], [4-55/31], [4-56/31],  
   [4-59/32], [4-63/35], [4-67/36], [4-70/36], [4-71/37],  
   [4-76/40], [4-78/42], [4-79/42], [4-80/45], [4-81/45],  
   [4-83/45], [4-84/51], [4-85/54], [4-86/54], [4-87/54],  
   [4-95/57], [4-97/59], [4-99/61]  
 < ..... [2-12/4], [2-4/1], [3-22/14]  
 <= ..... [2-12/4], [3-22/14]  
 <> ..... [2-12/4], [3-22/14]  
 = ..... [2-12/4], [2-4/1], [3-167/115], [3-211/147],  
       [3-22/14], [3-221/153]  
 > ..... [2-12/4], [2-4/1], [3-22/14]  
 >= ..... [2-12/4], [3-22/14]  
 ? ..... [2-10/3], [2-12/4], [2-4/1]  
 @ ..... [2-12/4], [2-4/1], [3-32/15]  
 A ..... [2-2/1], [3-117/82]  
 ABS ..... [2-14/5], [3-65/44], [3-78/62]  
 ALL ..... [2-14/5], [4-11/7], [4-50/23]  
 AND ..... [2-13/5], [3-20/14]  
 ANY ..... [2-13/5], [3-176/115], [3-220/150]  
 ARRAY ..... [2-13/5], [3-147/104]  
 ARRAY REP ..... [2-14/5], [3-153/109]  
 ARRAY STD ..... [2-14/5], [3-153/109]  
 AS ..... [2-14/5], [3-91/69]  
 ASSERT ..... [2-13/5], [4-84/51]  
 B ..... [2-2/1], [3-117/82]  
 BIN ..... [2-13/5], [3-111/82]  
 BIN OF ..... [3-128/88]  
 BIT ..... [2-14/5], [3-108/81]  
 BIT STD ..... [2-14/5], [3-118/83]  
 BLANKS ..... [2-14/5], [3-146/102], [3-185/127]  
 BLOCK ..... [2-13/5], [4-59/32]  
 BOOLEAN ..... [2-14/5], [3-98/75]  
 BOOLEAN STD ..... [2-14/5], [3-101/75]  
 BOUNDS ..... [2-14/5], [3-149/104]  
 C ..... [2-2/1], [3-117/82]  
 CHARSET ..... [2-14/5], [3-134/90]  
 CHAR STR ..... [2-14/5], [3-131/90]  
 CHAR STR REP ..... [2-14/5], [3-139/97]  
 CHAR STR STD ..... [2-14/5], [3-139/97]  
 CLOCK ..... [2-14/5], [4-100/62]  
 CONNECT ..... [2-14/5], [4-87/54]  
 CONST ..... [2-13/5], [3-4/6]  
 D ..... [2-2/1], [3-117/82]  
 DATA ..... [2-14/5], [4-7/7]  
 DEFINES ..... [2-14/5], [4-11/7]  
 DIMENSIONS ..... [2-14/5], [3-149/104]

DO ..... [2-13/5], [4-76/40], [4-79/42]  
 DUP ..... [2-14/5], [3-126/87], [3-145/102]  
 DYNSIZE ..... [2-14/5], [4-109/74], [4-113/74]  
 E ..... [2-2/1], [3-117/82], [3-52/26]  
 ELEMENT\_TYPE ..... [2-14/5], [3-149/104], [3-196/138]  
 ELSE ..... [2-13/5], [3-188/127], [4-66/35], [4-71/37]  
 END ..... [2-13/5], [3-160/114], [3-164/115],  
           [3-165/115], [3-186/127], [3-187/127], [3-188/127], [4-14/7],  
           [4-20/11], [4-33/12], [4-48/23], [4-59/32], [4-63/35],  
           [4-67/36], [4-70/36], [4-71/37], [4-76/40], [4-79/42],  
           [4-80/45], [4-81/45], [4-85/54], [4-89/54]  
 ENUM ..... [2-14/5], [3-83/65]  
 ENUM REP ..... [2-14/5], [3-90/69]  
 ENUM STD ..... [2-14/5], [3-90/69]  
 ENUM\_VAL REP ..... [2-14/5], [3-90/69]  
 EPSILON ..... [2-14/5], [3-81/63]  
 EXCEPT ..... [2-13/5], [4-80/45]  
 EXIT ..... [2-13/5], [4-78/42]  
 EXPONENT\_MAX ..... [2-14/5], [3-70/47]  
 EXPONENT\_MIN ..... [2-14/5], [3-70/47]  
 EXPONENT\_SIZE ..... [2-14/5], [3-73/50]  
 EXPORTS ..... [2-14/5], [4-50/23]  
 EXTENT ..... [2-14/5], [3-149/104], [3-84/65]  
 F ..... [2-2/1], [3-117/82]  
 FALSE ..... [2-13/5], [3-100/75]  
 FILLER ..... [2-14/5], [3-185/127]  
 FIXED ..... [2-14/5], [3-45/23]  
 FIXED\_MAX ..... [2-14/5], [3-67/45]  
 FIXED\_MIN ..... [2-14/5], [3-66/45]  
 FIXED REP ..... [2-14/5], [3-55/28]  
 FIXED\_STD ..... [2-14/5], [3-55/28]  
 FLOAT ..... [2-14/5], [3-68/47]  
 FLOAT\_MAX ..... [2-14/5], [3-80/62]  
 FLOAT\_MIN ..... [2-14/5], [3-79/62]  
 FLOAT\_STD ..... [2-14/5], [3-72/50]  
 FOR ..... [2-13/5], [4-79/42], [4-90/54]  
 FROM ..... [2-13/5], [3-164/115], [3-186/127], [4-67/36]  
 FUNCTION ..... [2-13/5], [4-31/12], [4-33/12], [5-4/5]  
 G ..... [2-2/1]  
 GO ..... [2-13/5], [4-56/31]  
 H ..... [2-2/1]  
 HANDLER\_EXISTS ..... [2-14/5], [3-106/79]  
 HEX ..... [2-13/5], [3-111/82]  
 HEX\_OF ..... [3-130/88]  
 I ..... [2-2/1]  
 IF ..... [2-13/5], [4-63/35], [4-64/35]  
 IMPORTS ..... [2-14/5], [4-28/11]  
 IMP\_PRECISION ..... [2-14/5], [3-70/47], [3-82/64]  
 IN ..... [2-13/5], [3-21/14], [3-22/14], [4-79/42],  
       [4-90/54]  
 INLINE ..... [2-13/5], [4-18/11], [4-31/12]  
 INTEGER ..... [2-14/5], [3-45/23]  
 IS\_MANIFEST ..... [2-14/5], [3-107/80]  
 J ..... [2-2/1]  
 K ..... [2-2/1]

L ..... [2-2/1]  
 LENGTH ..... [2-14/5], [3-110/81], [3-134/90]  
 LOOP ..... [2-13/5], [4-76/40], [4-79/42]  
 M ..... [2-2/1]  
 MAIN ..... [2-13/5], [4-5/7]  
 MANTISSA\_SIZE ..... [2-14/5], [3-73/50]  
 MAX ..... [2-14/5], [3-149/104], [3-49/23], [3-70/47],  
       [3-84/65]  
 MIN ..... [2-14/5], [3-149/104], [3-49/23], [3-70/47],  
       [3-84/65]  
 MOD ..... [2-13/5], [3-26/14]  
 MODULE ..... [2-13/5], [4-46/23], [4-48/23]  
 MODULES ..... [4-7/7]  
 N ..... [2-2/1]  
 NAMING ..... [2-13/5], [4-60/33]  
 NEEDS ..... [2-13/5], [5-11/8]  
 NEW ..... [2-14/5], [3-198/139], [3-206/143]  
 NIL ..... [2-13/5], [3-197/139]  
 NOM\_PRECISION ..... [2-14/5], [3-70/47]  
 NOT ..... [2-13/5], [3-30/15]  
 O ..... [2-2/1]  
 OBJECT\_MACHINE ..... [2-14/5], [4-68/36]  
 OCT ..... [2-13/5], [3-111/82]  
 OCT\_OF ..... [3-129/88]  
 OF ..... [2-13/5], [3-147/104], [3-221/153]  
 OMIT ..... [2-14/5], [3-184/127]  
 OR ..... [2-13/5], [3-18/14]  
 ORIF ..... [2-13/5], [4-65/35]  
 P ..... [2-2/1]  
 PACKING ..... [2-14/5], [3-141/97], [3-154/109]  
 PAR\_BLOCK ..... [2-13/5], [4-85/54]  
 PATH ..... [2-13/5], [4-87/54], [4-89/54]  
 PREC ..... [2-14/5], [3-15/14]  
 PRECISION ..... [2-14/5], [3-49/23]  
 PRED ..... [2-14/5], [3-97/74]  
 PRIORITY ..... [2-14/5], [4-87/54], [4-99/61]  
 PROCEDURE ..... [2-13/5], [4-18/11], [4-20/11], [5-2/5]  
 PTR ..... [2-14/5], [3-195/138], [3-198/139]  
 PTR\_STD ..... [2-14/5], [3-199/139]  
 Q ..... [2-2/1]  
 R ..... [2-2/1]  
 RADIX ..... [2-14/5], [3-49/23], [3-70/47]  
 RECORD ..... [2-13/5], [3-160/114]  
 RECORD REP ..... [2-14/5], [3-181/127]  
 RECORD\_STD ..... [3-181/127]  
 REGION ..... [2-14/5]  
 RELEASE ..... [2-14/5], [4-103/64]  
 REP ..... [2-13/5], [3-147/104], [3-45/23], [3-8/9]  
 REPTYPE ..... [2-14/5], [3-221/153]  
 REQUEST ..... [2-14/5], [4-101/63]  
 RETURN ..... [2-13/5], [4-44/15]  
 RETURNS ..... [2-14/5], [4-35/12], [5-5/5]  
 ROUTINES ..... [2-14/5], [4-7/7]  
 S ..... [2-2/1], [3-53/26]  
 SCALE ..... [2-14/5], [3-49/23]

SEGMENT ..... [2-13/5], [4-14/7], [4-5/7]  
 SELECT ..... [2-13/5], [3-164/115], [3-186/127], [4-67/36]  
 SEMAPHORE ..... [2-14/5], [3-203/143], [3-206/143]  
 SEMAPHORE\_STD ..... [3-207/144]  
 SIGNAL ..... [2-13/5], [4-83/45]  
 SIGNALS ..... [2-14/5], [4-29/12]  
 SIGNING ..... [2-14/5], [3-58/28]  
 SIZE ..... [2-14/5], [3-102/75], [3-119/83], [3-141/97],  
       [3-154/109], [3-189/127], [3-200/139], [3-208/144],  
       [3-228/157], [3-58/28], [3-73/50], [3-92/69]  
 START\_IO ..... [4-104/71]  
 SUCC ..... [2-14/5], [3-96/74]  
 SYNCH ..... [2-14/5]  
 T ..... [2-2/1]  
 TAG ..... [2-13/5], [3-170/115], [3-213/148]  
 TARGET ..... [2-14/5], [4-108/74], [4-112/74]  
 TERMINATE ..... [2-13/5], [4-97/59]  
 THEN ..... [2-13/5], [4-64/35], [4-65/35]  
 TO ..... [2-13/5], [4-56/31]  
 TO\_BIT ..... [2-14/5], [3-124/87]  
 TO\_FIXED ..... [2-14/5], [3-63/41]  
 TO\_FLOAT ..... [2-14/5], [3-76/59]  
 TRUE ..... [2-13/5], [3-100/75]  
 TYPE ..... [2-13/5], [3-167/115], [3-173/115],  
       [3-211/147], [3-213/148], [4-25/11]  
 TYPES ..... [2-14/5], [4-7/7]  
 U ..... [2-2/1]  
 UNORDERED ..... [2-14/5], [3-83/65]  
 UNTIL ..... [2-13/5], [4-76/40]  
 USAGE ..... [2-14/5], [3-205/143]  
 USES ..... [2-14/5], [4-6/7]  
 V ..... [2-2/1]  
 VALUES ..... [2-14/5], [3-90/69]  
 VAR ..... [2-13/5], [3-162/114], [3-6/8]  
 VARIANTS ..... [2-14/5], [3-184/127]  
 W ..... [2-2/1]  
 WAIT ..... [2-14/5], [4-95/57]  
 WHEN ..... [2-13/5], [4-76/40]  
 WHERE ..... [2-13/5], [5-10/8]  
 WITH ..... [2-13/5], [3-15/14], [4-50/23], [4-60/33]  
 X ..... [2-2/1]  
 XOR ..... [2-13/5], [3-18/14]  
 Y ..... [2-2/1]  
 Z ..... [2-2/1]  
 ZEROS ..... [2-14/5], [3-127/88], [3-185/127]  
 \_l ..... [2-12/4], [2-4/1], [3-147/104], [3-179/124],  
       [3-35/15], [3-36/15], [3-40/15]  
 \ ..... [2-10/3], [2-12/4], [2-4/1], [3-53/26],  
       [3-87/68]  
 ^ ..... [2-4/1]  
 - ..... [2-11/3], [2-4/1]  
 abbrev-type-name ..... d[3-219/150], [3-217/150]  
 abstract-type-name ..... d[3-218/150], [3-217/150]  
 array-assignment ..... d[3-159/113], [3-42/21]  
 array-attr ..... d[3-149/104], [3-3/4]

array-bound ..... d[3-148/104], [3-147/104]  
 array-constructor ..... d[3-151/107], [3-11/11], [3-151/107]  
 array-expression ..... d[3-155/111], [3-159/113]  
 array-rep-attr ..... d[3-154/109], [3-3/4]  
 array-reptype ..... d[3-153/109], [3-13/12], [3-147/104]  
 array-spec ..... d[3-147/104], [3-8/9]  
 assert-stmt ..... d[4-84/51], [4-54/31]  
 assignment-stmt ..... d[3-42/21], [4-54/31]  
 attribute-name ..... d[3-3/4], [3-2/4]  
 attribute-query ..... d[3-2/4], [3-31/15]  
 base-formal-id ..... d[3-223/153], [3-221/153]  
 basic-char ..... d[3-137/92], [3-136/92]  
 basic-string ..... d[3-136/92], [3-135/92]  
 bin-char ..... d[3-115/82], [3-112/82]  
 bin-string ..... d[3-112/82], [3-111/82]  
 bit-target ..... d[3-123/86], [3-122/86]  
 bitstring-assignment ..... d[3-122/86], [3-42/21]  
 bitstring-attr ..... d[3-110/81], [3-3/4]  
 bitstring-bin-function ..... d[3-128/88], [3-44/22]  
 bitstring-dup-function ..... d[3-126/87], [3-44/22]  
 bitstring-expression ..... d[3-120/84], [3-122/86], [3-126/87], [3-64/41]  
 bitstring-hex-function ..... d[3-130/88], [3-44/22]  
 bitstring-literal ..... d[3-111/82], [3-10/11]  
 bitstring-oct-function ..... d[3-129/88], [3-44/22]  
 bitstring-rep-attr ..... d[3-119/83], [3-3/4]  
 bitstring-reptype ..... d[3-118/83], [3-13/12]  
 bitstring-spec ..... d[3-108/81], [3-9/9]  
 bitstring-to-bit-function ... d[3-124/87], [3-44/22]  
 bitstring-zeros-function .... d[3-127/88], [3-44/22]  
 blank ..... d[2-6/1], [2-1/1], [2-5/1], [3-112/82],  
                       [3-113/82], [3-114/82]  
 block ..... d[4-59/32], [4-58/32]  
 boolean-assignment ..... d[3-105/79], [3-42/21]  
 boolean-attr ..... d[3-99/75], [3-3/4]  
 boolean-expression ..... d[3-103/76], [3-105/79], [3-64/41], [4-64/35],  
                       [4-65/35], [4-77/40], [4-84/51]  
 boolean-literal ..... d[3-100/75], [3-10/11]  
 boolean-manifest-expression . d[3-104/76]  
 boolean-rep-attr ..... d[3-102/75], [3-3/4]  
 boolean-reptype ..... d[3-101/75], [3-13/12]  
 boolean-spec ..... d[3-98/75], [3-9/9]  
 bound-var ..... d[4-61/33], [4-60/33], [4-68/36]  
 break ..... d[2-11/3], [2-10/3]  
 case-body ..... d[4-69/36], [4-67/36]  
 case-element ..... d[4-70/36], [4-69/36]  
 case-else-element ..... d[4-71/37], [4-69/36]  
 case-label ..... d[4-73/37], [4-72/37],  
 case-stmt ..... d[4-67/36], [4-62/35]  
 char-target ..... d[3-144/100], [3-143/100]  
 character ..... d[2-1/1], [2-16/7], [2-17/7]  
 character-no-apos ..... d[3-89/68], [3-87/68]  
 charset-name ..... d[3-132/90], [3-131/90], [3-135/92]  
 charstring-assignment ..... d[3-143/100], [3-42/21]  
 charstring-attr ..... d[3-134/90], [3-3/4]  
 charstring-blanks-function .. d[3-146/102], [3-44/22]

charstring-dup-function ..... d[3-145/102], [3-44/22]  
charstring-expression ..... d[3-142/99], [3-128/88], [3-129/88], [3-130/88],  
[3-143/100], [3-145/102], [3-64/41], [3-77/59]  
charstring-literal ..... d[3-135/92], [3-10/11]  
charstring-rep-attr ..... d[3-141/97], [3-3/4]  
charstring-reptype ..... d[3-139/97], [3-13/12]  
charstring-spec ..... d[3-131/90], [3-9/9]  
choice ..... d[4-68/36], [4-67/36]  
choice-label ..... d[4-72/37], [4-70/36]  
clock-function ..... d[4-100/62], [3-44/22]  
closed-range ..... d[3-36/15], [3-123/86], [3-144/100], [3-34/15],  
[3-43/21]  
closed-scope-body ..... d[4-2/3], [4-13/7], [4-19/11], [4-32/12],  
[4-47/23], [4-88/54]  
comment ..... d[2-15/7], [2-9/2]  
common-packing-spec ..... d[3-182/127], [3-181/127]  
conditional-stmt ..... d[4-62/35], [4-58/32]  
conj-ops ..... d[3-20/14], [3-19/14]  
conjunction ..... d[3-19/14], [3-17/14], [3-19/14]  
const-field-spec ..... d[3-172/115], [3-169/115]  
constant ..... d[3-33/15], [3-157/111], [3-192/133], [3-2/4],  
[3-31/15], [3-34/15]  
constant-declaration ..... d[3-4/6], [3-1/3]  
constructor ..... d[3-11/11], [3-31/15]  
decimal-literal ..... d[3-51/26], [3-50/26], [3-71/49]  
declaration-stmt ..... d[3-1/3], [4-1/3], [4-2/3]  
defined-attr ..... d[3-226/157], [3-3/4]  
defined-rep-attr ..... d[3-228/157], [3-3/4]  
defines-clause ..... d[4-11/7], [4-5/7]  
definite-loop-stmt ..... d[4-79/42], [4-75/40]  
device-object ..... d[4-105/71], [4-104/71]  
device-operation ..... d[4-106/71], [4-104/71]  
digit ..... d[2-3/1], [2-1/1], [2-10/3], [3-54/26]  
display-char ..... d[2-5/1], [2-8/2], [3-111/82], [3-135/92]  
element-spec ..... d[3-184/127], [3-183/127]  
else-packing-spec ..... d[3-188/127], [3-186/127]  
embedded-comment ..... d[2-16/7], [2-15/7]  
enumerated-assignment ..... d[3-95/73], [3-42/21]  
enumerated-attr ..... d[3-84/65], [3-3/4]  
enumerated-expression ..... d[3-93/71], [3-148/104], [3-64/41], [3-95/73],  
[3-96/74], [3-97/74]  
enumerated-id ..... d[3-86/68], [3-85/68]  
enumerated-literal ..... d[3-85/68], [3-10/11], [3-136/92], [3-16/14],  
[3-83/65], [3-91/69]  
enumerated-pred-function ..... d[3-97/74], [3-44/22]  
enumerated-rep-attr ..... d[3-92/69], [3-3/4]  
enumerated-reptype ..... d[3-90/69], [3-13/12]  
enumerated-spec ..... d[3-83/65], [3-9/9]  
enumerated-succ-function ..... d[3-96/74], [3-44/22]  
enumerated-type ..... d[3-133/90], [3-132/90]  
exception-name ..... d[4-82/45], [3-106/79], [4-29/12], [4-81/45],  
[4-83/45]  
exceptions-clause ..... d[4-80/45], [4-51/29], [4-70/36], [4-71/37]  
executable-stmt ..... d[4-51/29], [4-1/3], [4-2/3]  
exit-stmt ..... d[4-78/42], [4-54/31]

explicit-rep-converter ..... d[3-41/15], [3-31/15]  
 exponent-spec ..... d[3-52/26], [3-50/26], [3-51/26]  
 exports-clause ..... d[4-50/23], [4-46/23]  
 expression ..... d[3-15/14], [3-103/76], [3-107/80], [3-120/84],  
     [3-125/87], [3-14/14], [3-142/99], [3-152/107], [3-155/111],  
     [3-158/111], [3-176/115], [3-180/124], [3-190/133], [3-2/4],  
     [3-201/140], [3-209/144], [3-220/150], [3-37/15], [3-38/15],  
     [3-4/6], [3-40/15], [3-41/15], [3-43/21], [3-59/30], [3-6/8],  
     [3-74/51], [3-93/71], [4-105/71], [4-106/71], [4-39/15],  
     [4-44/15], [4-60/33], [4-68/36]  
 factor ..... d[3-27/15], [3-25/14], [3-27/15]  
 factor-ops ..... d[3-28/15], [3-27/15]  
 field-constant ..... d[3-192/133], [3-33/15]  
 field-declarations ..... d[3-161/114], [3-160/114], [3-165/115]  
 field-id ..... d[3-193/133], [3-191/133], [3-192/133]  
 field-name ..... d[3-163/114], [3-162/114], [3-172/115],  
     [3-180/124], [3-184/127], [3-193/133]  
 field-reference ..... d[3-191/133], [3-32/15]  
 filler-spec ..... d[3-185/127], [3-184/127]  
 fixed-abs-function ..... d[3-65/44], [3-44/22]  
 fixed-assignment ..... d[3-62/39], [3-42/21]  
 fixed-attr ..... d[3-49/23], [3-3/4]  
 fixed-expression ..... d[3-59/30], [3-60/30], [3-62/39], [3-64/41],  
     [3-65/44], [3-77/59], [4-96/57]  
 fixed-literal ..... d[3-50/26], [3-10/11]  
 fixed-max-function ..... d[3-67/45], [3-44/22]  
 fixed-min-function ..... d[3-66/45], [3-44/22]  
 fixed-prec ..... d[3-48/23], [3-45/23], [3-63/41], [3-66/45],  
     [3-67/45]  
 fixed-rep-attr ..... d[3-58/28], [3-3/4]  
 fixed-reptype ..... d[3-55/28], [3-13/12], [3-45/23]  
 fixed-scale ..... d[3-46/23], [3-45/23], [3-63/41], [3-66/45],  
     [3-67/45]  
 fixed-spec ..... d[3-45/23], [3-9/9]  
 float-abs-function ..... d[3-78/62], [3-44/22]  
 float-assignment ..... d[3-75/57], [3-42/21]  
 float-attr ..... d[3-70/47], [3-3/4]  
 float-epsilon-function ..... d[3-81/63], [3-44/22]  
 float-expression ..... d[3-74/51], [3-64/41], [3-75/57], [3-77/59],  
     [3-78/62]  
 float-imp-prec-function ..... d[3-82/64], [3-44/22]  
 float-literal ..... d[3-71/49], [3-10/11]  
 float-max-function ..... d[3-80/62], [3-44/22]  
 float-min-function ..... d[3-79/62], [3-44/22]  
 float-precision ..... d[3-69/47], [3-68/47], [3-76/59], [3-79/62],  
     [3-80/62], [3-81/63], [3-82/64]  
 float-rep-attr ..... d[3-73/50], [3-3/4]  
 float-reptype ..... d[3-72/50], [3-13/12]  
 float-spec ..... d[3-68/47], [3-9/9]  
 formal-name ..... d[4-27/11], [4-25/11]  
 formal-parameter ..... d[4-25/11], [4-22/11], [4-23/11], [4-24/11]  
 func-argument-list ..... d[4-43/15], [4-42/15]  
 func-formal-parm-list ..... d[4-34/12], [4-31/12]  
 func-name ..... d[4-36/12], [4-10/7], [4-31/12], [4-33/12],  
     [4-42/15], [5-9/5]

func-template ..... d[5-5/5], [5-4/5], [5-6/5], [5-9/5]  
 function-argument ..... d[5-9/5], [5-7/5]  
 function-body ..... d[4-32/12], [4-30/12]  
 function-declaration ..... d[4-30/12], [4-16/11]  
 function-header ..... d[4-31/12], [4-112/74], [4-30/12]  
 function-invocation ..... d[4-42/15], [3-31/15]  
 function-spec ..... d[5-4/5], [5-1/5]  
 function-trailer ..... d[4-33/12], [4-112/74], [4-30/12]  
 general-assignment ..... d[3-43/21], [3-42/21]  
 go-to-stmt ..... d[4-56/31], [4-54/31]  
 handler-exists-function ..... d[3-106/79], [3-44/22]  
 hex-char ..... d[3-117/82], [3-114/82]  
 hex-string ..... d[3-114/82], [3-111/82]  
 identifier ..... d[2-10/3], [2-9/2], [3-163/114], [3-168/115],  
     [3-171/115], [3-173/115], [3-212/148], [3-213/148],  
     [3-222/153], [3-223/153], [3-227/157], [3-5/6], [3-7/8],  
     [3-86/68], [4-12/7], [4-15/7], [4-26/11], [4-27/11], [4-36/12],  
     [4-49/23], [4-53/29], [4-61/33], [4-74/37], [4-79/42],  
     [4-82/45], [4-90/54], [4-91/54], [4-94/54]  
 if-element ..... d[4-64/35], [4-63/35]  
 if-else-element ..... d[4-66/35], [4-63/35]  
 if-stmt ..... d[4-63/35], [4-62/35]  
 imports-clause ..... d[4-28/11], [4-18/11], [4-31/12], [4-46/23],  
     [4-87/54]  
 indefinite-loop-stmt ..... d[4-76/40], [4-75/40]  
 index ..... d[3-150/104], [3-149/104]  
 inout-arguments ..... d[4-40/15], [4-38/15]  
 inout-items ..... d[4-9/7], [4-7/7]  
 inout-parameters ..... d[4-23/11], [4-104/71], [4-21/11]  
 input-arguments ..... d[4-39/15], [4-38/15], [4-43/15]  
 input-items ..... d[4-8/7], [4-7/7]  
 input-parameters ..... d[4-22/11], [4-104/71], [4-21/11], [4-34/12]  
 integer-expression ..... d[3-60/30], [3-109/81], [3-126/87], [3-127/88],  
     [3-145/102], [3-146/102], [3-148/104], [3-150/104],  
     [3-206/143], [4-92/54]  
 interrupt ..... d[4-93/54], [4-87/54]  
 item-list ..... d[4-7/7], [4-28/11], [4-6/7]  
 key-word ..... d[2-14/5], [2-9/2]  
 label ..... d[4-53/29], [4-51/29], [4-56/31]  
 length-expression ..... d[3-109/81], [3-108/81], [3-131/90]  
 letter ..... d[2-2/1], [2-1/1], [2-10/3]  
 lexical-program ..... d[2-8/2]  
 lexical-token ..... d[2-9/2], [2-8/2]  
 literal ..... d[3-10/11], [2-9/2], [3-31/15]  
 loop-prefix ..... d[4-90/54], [4-87/54]  
 lower-bound ..... d[3-37/15], [3-35/15], [3-36/15]  
 machine-config ..... d[4-74/37], [4-72/37]  
 manifest-bitstring-expression . d[3-121/84], [3-91/69]  
 manifest-enumerated-expression . d[3-94/71], [3-140/97], [3-204/143],  
     [3-48/23], [3-57/28]  
 manifest-expression ..... d[3-14/14], [3-104/76], [3-121/84], [3-166/115],  
     [3-61/30], [3-94/71], [4-73/37]  
 manifest-function ..... d[3-107/80], [3-44/22]  
 manifest-integer-expression . d[3-61/30], [3-15/14], [3-185/127], [3-24/14],  
     [3-26/14], [3-28/15], [3-46/23], [3-47/23], [3-56/28],

[3-69/47], [4-111/74]  
**manifest-range-spec** ..... d[3-39/15], [3-166/115], [4-73/37]  
**module-body** ..... d[4-47/23], [4-45/23]  
**module-declaration** ..... d[4-45/23], [4-3/3]  
**module-header** ..... d[4-46/23], [4-45/23]  
**module-name** ..... d[4-49/23], [4-46/23], [4-48/23], [4-7/7]  
**module-trailer** ..... d[4-48/23], [4-45/23]  
**name** ..... d[4-12/7], [4-11/7], [4-50/23]  
**named-handler** ..... d[4-81/45], [4-80/45]  
**needs-spec** ..... d[5-11/8], [5-10/8]  
**new-line** ..... d[2-7/1], [2-17/7], [2-5/1]  
**non-array-expression** ..... d[3-152/107], [3-151/107]  
**non-printing-char-id** ..... d[3-88/68], [3-85/68]  
**null-stmt** ..... d[4-55/31], [4-54/31]  
**number** ..... d[3-54/26], [3-50/26], [3-51/26], [3-52/26],  
[3-53/26]  
**oct-char** ..... d[3-116/82], [3-113/82]  
**oct-string** ..... d[3-113/82], [3-111/82]  
**open-scope-body** ..... d[4-1/3], [4-59/32], [4-64/35], [4-65/35],  
[4-66/35], [4-70/36], [4-71/37], [4-76/40], [4-79/42],  
[4-81/45]  
**orif-element** ..... d[4-65/35], [4-63/35]  
**other** ..... d[2-4/1], [2-1/1]  
**other-symbols** ..... d[2-12/4], [2-9/2]  
**output-arguments** ..... d[4-41/15], [4-38/15]  
**output-parameters** ..... d[4-24/11], [4-104/71], [4-21/11]  
**packing-mode** ..... d[3-140/97], [3-139/97], [3-153/109],  
[3-183/127], [3-188/127]  
**packing-spec** ..... d[3-183/127], [3-182/127], [3-187/127]  
**parallel-block** ..... d[4-85/54], [4-58/32]  
**parallel-control-stmt** ..... d[4-57/31], [4-54/31]  
**parallel-path** ..... d[4-86/54], [4-85/54]  
**path-body** ..... d[4-88/54], [4-86/54]  
**path-header** ..... d[4-87/54], [4-86/54]  
**path-name** ..... d[4-91/54], [4-87/54], [4-89/54], [4-98/59]  
**path-reference** ..... d[4-98/59], [4-100/62], [4-97/59], [4-99/61]  
**path-trailer** ..... d[4-89/54], [4-86/54]  
**pinter-assignment** ..... d[3-202/142], [3-42/21]  
**pointer-attr** ..... d[3-196/138], [3-3/4]  
**pointer-constructor** ..... d[3-198/139], [3-11/11]  
**pointer-expression** ..... d[3-201/140], [3-202/142]  
**pointer-literal** ..... d[3-197/139], [3-10/11]  
**pointer-rep-attr** ..... d[3-200/139], [3-3/4]  
**pointer-reptype** ..... d[3-199/139], [3-13/12]  
**pointer-spec** ..... d[3-195/138], [3-9/9]  
**predefined-function** ..... d[3-44/22], [3-31/15]  
**predefined-reptype** ..... d[3-13/12], [3-12/12]  
**predefined-type** ..... d[3-9/9], [3-8/9]  
**predicate** ..... d[4-77/40], [4-76/40]  
**primary** ..... d[3-31/15], [3-29/15], [3-32/15]  
**printing-char** ..... d[3-138/92], [3-137/92]  
**printing-char-id** ..... d[3-87/68], [3-85/68]  
**priority-expr** ..... d[4-92/54], [4-87/54], [4-99/61]  
**priority-stmt** ..... d[4-99/61], [4-57/31]  
**proc-argument-list** ..... d[4-38/15], [4-37/15]

proc-formal-parm-list ..... d[4-21/11], [4-18/11]  
proc-name ..... d[4-26/11], [4-10/7], [4-18/11], [4-20/11],  
[4-37/15], [5-8/5]  
proc-template ..... d[5-3/5], [5-2/5], [5-6/5], [5-8/5]  
procedure-argument ..... d[5-8/5], [5-7/5]  
procedure-body ..... d[4-19/11], [4-17/11]  
procedure-declaration ..... d[4-17/11], [4-16/11]  
procedure-header ..... d[4-18/11], [4-108/74], [4-17/11]  
procedure-spec ..... d[5-2/5], [5-1/5]  
procedure-stmt ..... d[4-37/15], [4-54/31]  
procedure-trailer ..... d[4-20/11], [4-108/74], [4-17/11]  
radix ..... d[3-47/23], [3-45/23], [3-63/41], [3-66/45],  
[3-67/45]  
range-spec ..... d[3-35/15], [3-21/14], [3-215/150], [3-39/15],  
[3-45/23], [3-68/47], [3-83/65], [4-79/42], [4-90/54]  
record-argument ..... d[3-176/115], [3-175/115]  
record-assignment ..... d[3-194/136], [3-42/21]  
record-attr ..... d[3-177/115], [3-3/4]  
record-constructor ..... d[3-178/124], [3-11/11]  
record-expression ..... d[3-190/133], [3-194/136]  
record-parameter ..... d[3-169/115], [3-167/115]  
record-rep-attr ..... d[3-189/127], [3-3/4]  
record-reptype ..... d[3-181/127], [3-13/12]  
record-spec ..... d[3-160/114], [3-174/115], [3-9/9]  
record-type-declaration ..... d[3-167/115], [3-211/147]  
record-type-definition ..... d[3-174/115], [3-167/115]  
record-type-name ..... d[3-168/115], [3-167/115], [3-175/115]  
record-type-reference ..... d[3-175/115], [3-215/150]  
record-type-specifier ..... d[3-179/124], [3-178/124]  
rel-ops ..... d[3-22/14], [3-21/14]  
relation ..... d[3-21/14], [3-19/14], [3-21/14]  
release-stmt ..... d[4-103/64], [4-57/31]  
repetitive-stmt ..... d[4-75/40], [4-58/32]  
reptype-declaration ..... d[3-221/153], [3-1/3]  
reptype-definition ..... d[3-224/153], [3-221/153]  
reptype-name ..... d[3-222/153], [3-221/153], [3-225/155]  
reptype-reference ..... d[3-225/155], [3-12/12]  
reptype-spec ..... d[3-12/12], [3-224/153], [3-41/15], [3-8/9]  
request-stmt ..... d[4-101/63], [4-57/31]  
reserved-word ..... d[2-13/5], [2-9/2]  
return-stmt ..... d[4-44/15], [4-54/31]  
returns-clause ..... d[4-35/12], [4-31/12]  
rounding-mode ..... d[3-16/14], [3-15/14], [3-63/41], [3-76/59]  
routine-argument ..... d[5-7/5], [4-39/15]  
routine-declaration ..... d[4-16/11], [4-3/3]  
routine-name ..... d[4-10/7], [4-50/23], [4-7/7], [5-12/8]  
routine-spec ..... d[5-1/5], [4-25/11]  
routine-template ..... d[5-6/5], [5-12/8]  
routines-and-modules ..... d[4-3/3], [4-2/3]  
scale-spec ..... d[3-53/26], [3-50/26]  
secondary ..... d[3-29/15], [3-27/15]  
segment-body ..... d[4-13/7], [4-4/7]  
segment-declaration ..... d[4-4/7]  
segment-header ..... d[4-5/7], [4-4/7]  
segment-name ..... d[4-15/7], [4-14/7], [4-5/7], [4-6/7]

segment-trailer ..... d[4-14/7], [4-4/7]  
 semaphore-assignment ..... d[3-210/145], [3-42/21]  
 semaphore-attr ..... d[3-205/143], [3-3/4]  
 semaphore-constructor ..... d[3-206/143], [3-11/11]  
 semaphore-expression ..... d[3-209/144], [3-210/145]  
 semaphore-name ..... d[4-94/54], [4-93/54]  
 semaphore-rep-attr ..... d[3-208/144], [3-3/4]  
 semaphore-reptype ..... d[3-207/144], [3-13/12]  
 semaphore-spec ..... d[3-203/143], [3-9/9]  
 semaphore-variable ..... d[4-102/63], [4-101/63], [4-103/64]  
 signal-stmt ..... d[4-83/45], [4-54/31]  
 signals-clause ..... d[4-29/12], [4-18/11], [4-31/12], [4-87/54],  
                        [5-3/5], [5-5/5]  
 signing ..... d[3-57/28], [3-55/28]  
 simple-constant ..... d[3-5/6], [3-33/15], [3-4/6], [4-8/7]  
 simple-exp-ops ..... d[3-18/14], [3-17/14]  
 simple-expression ..... d[3-17/14], [3-15/14], [3-17/14], [3-31/15]  
 simple-stmt ..... d[4-54/31], [4-52/29]  
 simple-variable ..... d[3-7/8], [3-32/15], [3-6/8], [4-8/7], [4-9/7]  
 size ..... d[3-56/28], [3-55/28], [3-90/69]  
 stack-size ..... d[4-111/74], [4-109/74], [4-113/74]  
 start-io-stmt ..... d[4-104/71], [4-54/31]  
 structured-stmt ..... d[4-58/32], [4-52/29], [4-58/32]  
 subscript ..... d[3-158/111], [3-156/111], [3-157/111],  
                        [4-98/59]  
 subscripted-constant ..... d[3-157/111], [3-33/15]  
 subscripted-variable ..... d[3-156/111], [3-32/15]  
 substring-reference ..... d[3-34/15], [3-31/15]  
 sum ..... d[3-23/14], [3-21/14], [3-23/14]  
 sum-ops ..... d[3-24/14], [3-23/14]  
 tag-field-spec ..... d[3-170/115], [3-169/115]  
 tag-name ..... d[3-171/115], [3-164/115], [3-170/115],  
                        [3-184/127], [3-186/127], [3-193/133], [3-213/148], [4-68/36]  
 target-function-body ..... d[4-113/74], [4-112/74]  
 target-function-declaration ..... d[4-112/74], [4-107/74]  
 target-procedure-body ..... d[4-109/74], [4-108/74]  
 target-procedure-declaration ..... d[4-108/74], [4-107/74]  
 target-routine-declaration .. d[4-107/74], [4-16/11]  
 target-scope-body ..... d[4-110/74], [4-109/74], [4-113/74]  
 template-spec ..... d[5-12/8], [5-11/8]  
 term ..... d[3-25/14], [3-23/14], [3-25/14]  
 term-ops ..... d[3-26/14], [3-25/14]  
 terminate-stmt ..... d[4-97/59], [4-57/31]  
 terminating-comment ..... d[2-17/7], [2-15/7]  
 time ..... d[4-96/57], [4-95/57]  
 to-bit-value ..... d[3-125/87], [3-124/87]  
 to-fixed-function ..... d[3-63/41], [3-44/22]  
 to-fixed-value ..... d[3-64/41], [3-63/41]  
 to-float-function ..... d[3-76/59], [3-44/22]  
 to-float-value ..... d[3-77/59], [3-76/59]  
 type-argument ..... d[3-220/150], [3-216/150], [3-225/155]  
 type-declaration ..... d[3-211/147], [3-1/3]  
 type-definition ..... d[3-214/148], [3-211/147]  
 type-formal ..... d[3-213/148], [3-211/147], [3-221/153]  
 type-formal-id ..... d[3-227/157], [3-226/157]

type-id ..... d[3-217/150], [3-216/150]  
type-instantiation ..... d[3-216/150], [3-215/150]  
type-name ..... d[3-212/148], [3-2/4], [3-211/147], [3-218/150],  
[3-219/150], [4-26/11], [4-36/12], [4-7/7], [5-11/8]  
type-parameter ..... d[3-173/115], [3-169/115]  
type-reference ..... d[3-215/150], [3-133/90], [3-8/9]  
type-spec ..... d[3-8/9], [3-147/104], [3-162/114], [3-170/115],  
[3-172/115], [3-176/115], [3-179/124], [3-195/138],  
[3-198/139], [3-213/148], [3-214/148], [3-220/150], [3-4/6],  
[3-40/15], [3-6/8], [4-25/11], [4-35/12], [4-39/15], [5-3/5],  
[5-5/5]  
type-specifier ..... d[3-40/15], [3-31/15]  
unary-ops ..... d[3-30/15], [3-29/15]  
unlabeled-stmt ..... d[4-52/29], [4-51/29]  
upper-bound ..... d[3-38/15], [3-35/15], [3-36/15]  
use-mode ..... d[3-204/143], [3-203/143]  
uses-clause ..... d[4-6/7], [4-5/7]  
value-spec ..... d[3-91/69], [3-90/69]  
var-field-init ..... d[3-180/124], [3-178/124]  
var-field-spec ..... d[3-162/114], [3-161/114]  
variable ..... d[3-32/15], [3-105/79], [3-123/86], [3-144/100],  
[3-156/111], [3-159/113], [3-191/133], [3-194/136], [3-2/4],  
[3-202/142], [3-210/145], [3-31/15], [3-34/15], [3-43/21],  
[3-62/39], [3-75/57], [3-95/73], [4-102/63], [4-40/15],  
[4-41/15], [4-60/33]  
variable-declaration ..... d[3-6/8], [3-1/3]  
variant-name ..... d[3-166/115], [3-165/115], [3-187/127],  
[4-72/37]  
variant-packing ..... d[3-186/127], [3-181/127]  
variant-packing-spec ..... d[3-187/127], [3-186/127]  
variant-part ..... d[3-164/115], [3-161/114]  
variant-spec ..... d[3-165/115], [3-164/115]  
wait-stmt ..... d[4-95/57], [4-57/31]  
where-clause ..... d[5-10/8], [4-18/11], [4-31/12]  
with-clause ..... d[4-60/33], [4-58/32]

Appendix C  
Application-Level Input/Output Facilities

The purpose of this Appendix is to illustrate how a standard library of application-level input/output facilities can be implemented via the built-in features of the language. The library is defined by the STANDARD\_IO segment; within this segment are two device-independent modules -- BINARY\_IO and TEXTUAL\_IO.

The BINARY\_IO module defines the abstract 'data-types' STREAM\_FILE and RANDOM\_FILE and a number of overloaded routines that take objects of these two types as parameters. The exported routines are:

CLEAR -- only used to initialize STREAM\_FILE and RANDOM\_FILE objects.

OPEN -- used to associate an external file or device with a STREAM\_FILE or RANDOM\_FILE object; the association can either be to an existing file or device, or to a new physical file.

CLOSE -- used to terminate an association established by OPEN; an external file can either be deleted or saved.

PUT -- a generic routine used to output binary representations of values to stream and random files.

GET -- a generic routine used to input binary representations of values from stream and random files.

REWIND -- a control operation used to rewind appropriate stream files (e.g., a tape, but not a teletype); a corresponding POSITION operation is defined for random files, but it need not be exported since it is always called indirectly via GET and PUT invocations for random files.

Except for their methods of access, stream and random files are essentially the same -- a logical sequence of objects (termed 'elements' of the file) that all have the same type. The definitions of GET and PUT insure that the type of input and output values must agree with the type of a file's elements. Although any given file cannot have elements of more than one type, this effect can be achieved, if desired, by declaring a file with a variant-record type for its elements. The physical representation of a file's elements is determined by their type, just as the representation of program variables is type-determined. Thus, as one example, the binary representation of a FIXED file's elements will be interpreted according to the representation attributes of the specific FIXED type declared for the file. (e.g., FIXED (S,R,P) [L:U] REP FIXED\_STD).

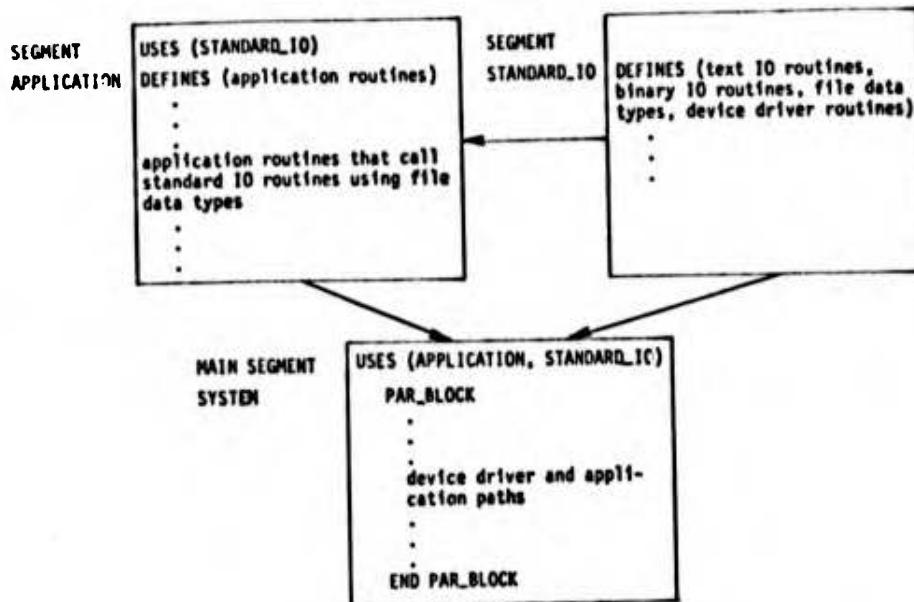
The OPEN and CLOSE routines rely on an independent set of file system operations for interpreting file 'pathnames', maintaining directories of saved files on physical devices, and so on. Such operations are not a part of the IO system per se, and are assumed to be defined in another segment.

The data transfer and file control routines eventually map down to procedures that are device-dependent. For example, an invocation of the PUT procedure will result in a call on a DEVICE\_OUT routine for some specific device. This routine will be overloaded to accept objects of various device-types. Each device-type and its overloaded cases of DEVICE\_OUT, DEVICE\_IN, and other appropriate operations will be defined in a device-specific module (i.e., a module which defines an abstract device type.) Thus the BINARY\_IO module must import these device-specific modules for all types of devices on which IO is desired.

The device-specific modules may be defined in the IO segment (as was done in the following example) or imported from another segment. Their routines will eventually map down to the low-level IO operations provided in a standard way by the language. In the following example, a device-specific module for LT33 teletype devices for a PDP-11 is defined in terms of the language's START\_IO statement. The module's definition assumes that the statement has been given a machine-dependent implementation specific to the PDP-11 and with capabilities to support LT33 teletypes. Actually two modules are defined: the SAFE\_TELETYPE module defines operations that are monitored for concurrent accesses to the same teletype printer or keyboard; this module is built on the more primitive TELETYPE module that actually uses the lowest-level START\_IO capabilities.

The control strategy by which IO is actually performed may vary slightly from device to device. For example, an output routine defined in a device-specific module may issue a START\_IO command to initiate the operation and then enter a loop to continually check the device, via another START\_IO, until the operation is completed. Alternately, the routine may use START\_IO to initiate the operation (and enable interrupts) and then wait until completion is signalled by an interrupt.

In the example provided here, the TELETYPE input and output routines transmit information to and from a set of monitored buffers, using the BUFFER type defined in Section 4.6.8. Another set of routines defined in the TELETYPE module repeatedly waits for non-full/non-empty buffers and asynchronously performs the actual IO between the buffers and the teletypes. These routines are also exported, and must be initiated in parallel with the path(s) that is (are) invoking the routines to perform teletype IO. Similar device drivers can be exported and initiated for other devices. A typical configuration for a system having such devices is illustrated in Figure C-1.



**Exhibit C-1. Typical System Configuration Using STANDARD\_IO Segment**

The second device-independent module, TEXTUAL\_IO, does not define any abstract types but collects together the definitions of the routines for the input and output of character-string representations of values. In the following example, the ASCII character set is used although this constraint can be loosened. The exported routines, which only work for character stream files, are:

**NEWLINE** -- used to output \CR \LF characters into unformatted output streams; similar routines for other special actions can also be defined.

**PRINT** -- used to output an unformatted character-string representation of a specified value.

**FORMPRINT** -- used for formatted output

**READ** -- used for unformatted input

**FORMREAD** -- used for formatted input

The type-dependent routines for converting values to and from their character-string representations are not defined as a part of the IO package, but must be defined with each type for which IO is desired (or added to the standard prelude for the built-in types). These routines are:

unformatted routines: T\$TO\_CHAR  
T\$FROM\_CHAR  
formatted routines: T\$TO\_FORM\_CHAR  
T\$FROM\_FORM\_CHAR  
T\$FORM\_CHAR\_LEN

Their expected behavior and example definitions are provided in the comments for the TEXTUAL\_IO routines that need them.

The code for the STANDARD\_IO segment is illustrated below, along with brief templates for main (i.e., executive) and application segments that use the IO capabilities. In reviewing the code, the reader is encouraged to observe the demonstrated use of the following language features and capabilities:

- . multiple segments.
- . modules for encapsulating functional capabilities and for defining abstract data types.
- . overloaded routine definitions.
- . explicit and derived generic parameters.
- . WHERE clauses for calling environment dependencies.
- . parameterized type definitions.
- . REGION semaphore requests and releases.
- . parallel paths with CONNECTS to SYNCH semaphores for hardware-dependent interrupts.
- . low-level START\_IO operations.
- . interrogation of type attributes.
- . type specifiers.
- . exception SIGNALS.
- . PAR\_BLOCK, SELECT, LOOP, IF control statements.

## MAIN SEGMENT SYSTEM

```

USES (STANDARD_IO (DATA: (NUM_OF_LT33, ...
    $ etcetera for constants
    $numbering other devices
),
ROUTINES: (INPUT_DRIVER,
    OUTPUT_DRIVER,
    NEW)), $ T$NEW for device-types T
APPLICATION);

PAR_BLOCK

LT33_PRINTERS
FOR I IN [1:NUM_OF_LT33] CONNECT (LT33_PS(I));
    OUTPUT_DRIVER (LT33_DEVICE$NEW(I) # LT33_PS(I));
END PATH LT33-PRINTERS;

LT33_KEYBOARDS
FOR I IN [1:NUM_OF_LT33] CONNECT (LT33_KS(I));
    INPUT_DRIVER (LT33_DEVICE$NEW(I) # LT33_KS(I));
END PATH LT33_KEYBOARDS;

$ etcetera for paths for other device drivers

SCHEDULE_APPLICATION
    APPLICATION_ROUTINE ( );
END PATH SCHEDULE_APPLICATION;

END PAR_BLOCK;

END SEGMENT SYSTEM;

```

## SEGMENT APPLICATION

```

USES (STANDARD_IO (MODULES: (TEXTUAL_IO,
                               BINARY_IO)),
      OTHER_SEGS ...),
DEFINES (APPLICATION_ROUTINE);

PROCEDURE APPLICATION_ROUTINE ( );
% which calls OPEN, CLOSE, PUT, PRINT, READ, and so on for STREAM_FILES
% and RANDOM_FILES; note that for textual IO, appropriate character conversion
% routines (i.e., FROM_CHAR, TO_FORM_CHAR, etc.) must be defined in the
% calling environment. (cf. the WHERE clauses in the IO routine headers).

END PROCEDURE APPLICATION_ROUTINE;

END SEGMENT APPLICATION;

```

## SEGMENT STANDARD\_IO

```

USES (MONITORED_BUFFER_SEG,
      FILE_SYSTEM_SEG)
DEFINES (TEXTUAL_IO, % modules
         BINARY_IO,
         INPUT_DRIVER, %overloaded device routines
         OUTPUT_DRIVER,
         NEW,
         NUM_OF_LT33,...); % numbers of devices

```

## MODULE SAFE\_TELETYPE

```

% This module defines procedures to input and output an arbitrary number of
% bytes to a teletype. Semaphores are used to control processes that are
% competing for the same device so that the byte streams will not be
% intermixed. The module makes use of the lower-level commands provided by
% the TELETYPE module.

```

```

IMPORTS (TYPES:(BUFFER)) % See Section 4.6.8

EXPORTS (NUM_OF_LT33,
         LT33_DEVICE WITH
         (NEW,
          INPUT_DRIVER,
          OUTPUT_DRIVER),
         DEVICE_OUT, DEVICE_IN);

VAR S: ARRAY [1:NUM_OF_LT33] OF SEMAPHORE (REGION)
    := SEM_INIT( );

CONST BYTESIZE : INTEGER := 8;

```

```

FUNCTION SEM_INIT ( )
RETURNS (ARRAY[1:NUM_OF_LT33] OF SEMAPHORE(REGION));

VAR S: ARRAY[1:NUM_OF_LT33] OF SEMAPHORE(REGION);

LOOP FOR I IN [1:NUM_OF_LT33] DO
  S(I) := SEMAPHORE$NEW(1);
END LOOP I;

RETURN (S);

END FUNCTION SEM_INIT;

PROCEDURE DEVICE_OUT
(DEV_OBJ: LT33_DEVICE,
BS: BIT(?LEN))

IMPORTS (DATA:(#S),
ROUTINES:(BYTE_OUT))
SIGNALS (IO_ERROR);

VAR I: INTEGER [0:LEN] := 0;

IF LEN = 0 THEN
  RETURN;
END IF;

IF REM(LEN, BYTESIZE) <> 0 THEN
  SIGNAL IO_ERROR;           $only output whole bytes
  END IF;

REQUEST (#S(NUMBER(DEV_OBJ)));

LOOP
  BYTE_OUT(DEV_OBJ, BS[I:I+BYTESIZE-1]);
  I := I + BYTESIZE;
END LOOP WHEN I = LEN;

RELEASE (#S(NUMBER(DEV_OBJ)));

END PROCEDURE DEVICE_OUT;

PROCEDURE DEVICE_IN
(DEV_OBJ: LT33_DEVICE
##
BS: BIT (?LEN))

IMPORTS (DATA:(#S),
ROUTINES:(BYTE_IN))
SIGNALS (IO_ERROR);

VAR I: INTEGER [0: LEN] := 0;

IF LEN=0 THEN

```

```
BS := BIN('');
RETURN;
END IF;

IF REM (LEN, BYTESIZE) <> 0 THEN
    SIGNAL IO_ERROR; $ only input whole bytes
    END IF;

REQUEST (#S(NUMBER(DEV_OBJ)));

LOOP
    BYTE_IN(DEV_OBJ ## BS[I:I+BYTESIZE-1]);
    I := I + BYTESIZE;
    END LOOP WHEN I = LEN;

RELEASE (#S(NUMBER(DEV_OBJ)));

END PROCEDURE DEVICE_IN;
```

## MODULE TELETYPE

This module defines procedures to input and output single bytes via a buffering scheme to LT33 teletype devices. Routines called from parallel application paths place information in, and retrieve information from, a set of monitored buffers. Independent device-driver paths transmit the information between the buffers and the actual devices. This module requires overloaded cases of the low-level START\_IO command that take objects of type LT33\_DEVICE\_ as their first parameters -- in particular:

```

START_IO (D: LT33_DEVICE_,
          C: LT33_OP_ [\READ_BYTE:\READ_BYTE]
          ##
          BS: BIT (BYTESIZE));
which uses the device object D to determine the storage location of the input data register and copies its contents to BS -- e.g.,
      MOVB ADDR_TABLE(NUMBER(D)) + 2, address (BS)

START_IO (D: LT33_DEVICE_,
          C: LT33_OP_ [\PRINT_BYTE:\PRINT_BYTE],
          BS: BIT (BYTESIZE));
which copies the contents of BS into the appropriate output data register -- e.g.,
      MOVB address (BS), ADDR_TABLE(NUMBER(D)) + 6

START_IO (D: LT33_DEVICE_,
          C: LT33_OP_ [\ENABLE_PRINTER_INTERRUPT:
                      \ENABLE_PUNCH_MAINTENANCE]);
which sets control bits in the appropriate status register -- e.g.,
      at ADDR_TABLE(NUMBER(D))
      or ADDR_TABLE(NUMBER(D)) + 4

START_IO (D: LT33_DEVICE_,
          C: LT33_OP_ [\READER_BUSY:
                      \PUNCH_MAINTENANCE_ENABLED]
          ##
          F: BOOLEAN);
which inquires about bits in the appropriate status register.

Just as the various cases of START_IO are built into a given implementation,
so must the types of the parameters for those cases. Thus the TELETYPE
  
```

```
% module assumes the existence of LT33_DEVICE_, whose only operation is a NEW
% construction, and LT33_OP_, whose declaration would be:
%
% TYPE LT33_OP_ = ENUM (\READ_BYTE,
%                      \PRINT_BYTE,
%                      \ENABLE_PRINTER_INTERRUPT,
%                      \ENABLE_KEYBOARD_INTERRUPT,
%                      \ENABLE_TAPE_READER,
%                      \ENABLE_PUNCH_MAINTENANCE,
%                      \READER_BUSY,
%                      \READER_DONE,
%                      \KEYBOARD_INTERRUPT_ENABLED,
%                      \TAPE_READER_ENABLED,
%                      \PRINTER_READY,
%                      \PRINTER_INTERRUPT_ENABLED,
%                      \PUNCH_MAINTENANCE_ENABLED);
%
% * * * *
%
IMPORTS (TYPES: (BUFFER))

EXPORTS (NUM_OF_LT33,
         LT33_DEVICE);

CONST NUM_OF_LT33 : INTEGER := ...;

TYPE LT33_DEVICE = INTEGER [1:NUM_OF_LT33];

CONST BYTESIZE: INTEGER := 8;
CONST BUF_SIZE: INTEGER := 64;

VAR IN_BUF: ARRAY [1:NUM_OF_LT33] OF BUFFER(BUF_SIZE)
     := BUF_INIT( );
VAR OUT_BUF: ARRAY [1:NUM_OF_LT33] OF BUFFER(BUF_SIZE)
     := BUF_INIT( );
```

```

FUNCTION BUF_INIT ( )
RETURNS (ARRAY [1:NUM_OF_LT33] OF BUFFER (BUF_SIZE));

VAR B: ARRAY [1:NUM_OF_LT33] OF BUFFER (BUF_SIZE);

LOOP FOR I IN [1:NUM_OF_LT33] DO
CLEAR (# B(I));
END LOOP I;

RETURN (B);

END FUNCTION BUF_INIT;

FUNCTION LT33_DEVICE$NEW
(DEV_NUM : INTEGER [1:NUM_OF_LT33])
RETURNS (LT33_DEVICE);

RETURN (DEV_NUM);

END FUNCTION LT33_DEVICE$NEW;

FUNCTION LT33_DEVICE$NUMBER
(DEV_OBJ : LT33_DEVICE)
RETURNS (INTEGER [1:NUM_OF_LT33]);

RETURN (DEV_OBJ);

END FUNCTION LT33_DEVICE$NUMBER;

PROCEDURE LT33_DEVICE$BYTE_OUT
(DEV_OBJ: LT33_DEVICE,
BS: BIT (BYTESIZE));
IMPORTS (DATA: (#OUT_BUF));

APPEND (BS # OUT_BUF(DEV_OBJ));

END PROCEDURE LT33_DEVICE$BYTE_OUT;

PROCEDURE LT33_DEVICE$BYTE_IN
(DEV_OBJ: LT33_DEVICE
##
BS: BIT (BYTESIZE))
IMPORTS (DATA:(#IN_BUF));

REMOVE (#IN_BUF(DEV_OBJ) # BS);

END PROCEDURE LT33_DEVICE$BYTE_IN;

```

```
PROCEDURE LT33_DEVICE$OUTPUT_DRIVER
    (DEV_OBJ: LT33_DEVICE
    #
    READY: SEMAPHORE (SYNCH))
    IMPORTS (DATA:(#OUT_BUF));

    VAR BS: BIT (BYTESIZE);

    SELECT OBJECT_MACHINE FROM

PDP_11 -->
    START_IO (LT33_DEVICE$_NEW(DEV_OBJ),
        %map to built-in type
        \ENABLE_PRINTER_INTERRUPT);
    LOOP
        REMOVE (#OUT_BUF(DEV_OBJ) # BS);
        REQUEST (#READY);
        START_IO (LT33_DEVICE$_NEW(DEV_OBJ),
            \PRINT_BYTE,
            BS);
    END LOOP;
END;

END SELECT;

END PROCEDURE LT33_DEVICE$OUTPUT_DRIVER;
```

```

PROCEDURE LT33_DEVICE$INPUT_DRIVER
  (DEV_OBJ: LT33_DEVICE
  #
  DONE: SEMAPHORE (SYNCH))
  IMPORTS (DATA: (#IN_BUF));

  VAR BS: BIT (BYTESIZE);

  SELECT OBJECT_MACHINE FROM

  PDP_11 -->
    START_IO (LT33_DEVICE$_NEW(DEV_OBJ),
              \ENABLE_KEYBOARD_INTERRUPT);
    LOOP
      REQUEST (# DONE);
      START_IO (LT33_DEVICE$_NEW(DEV_OBJ),
                \READ_BYTE
                ##
                BS);
      APPEND (BS # IN_BUF(DEV_OBJ));
    END LOOP;
  END;

  END SELECT;

  END PROCEDURE LT33_DEVICE$INPUT_DRIVER;

END MODULE TELETYPE;

END MODULE SAFE_TELETYPE;

```

% etcetera for other device-specific modules

## MODULE BINARY\_IO

% This module defines the STREAM\_FILE and RANDOM\_FILE data types, and their  
 % corresponding operations. These operations provide capabilities for  
 % dynamically linking external devices and physical files to logical file  
 % objects, for creating, deleting, and positioning files, and for transferring  
 % information to and from the files. The routines also form the basis for  
 % other procedures that handle textual IO.

```

IMPORTS (DATA:(PATHNAME_SIZE), % from FILE_SYSTEM_SEG
         ROUTINES:(FSYS_CREATE,FSYS_DELETE,
                    FSYS_UPDATE,FSYS_CHECK),
         MODULES:(SAFE_TELETYPED,...))
         %etcetera for other device-specific modules.

EXPORTS (STREAM_FILE, RANDOM_FILE,
         CLEAR, REWIND,
         GET, PUT,
         OPEN, CLOSE);

% Objects for stream and random file types have the same underlying
% representation, which is defined by the FILE type and hidden (i.e., not
% exported) by the module.

TYPE DEVICE_NAME = ENUM (\LT33, \CR11,\RP03,...);

TYPE FILE (ELEMENT_TYPE : TYPE) =
  RECORD
    VAR OPEN: BOOLEAN,
        PATHNAME: ASCII_STR(PATHNAME_SIZE),
        DEV_NAME: DEVICE_NAME,
        DEV_NUMBER: INTEGER [1:INTEGER_MAX],
        FILE_ADDRESS: INTEGER [0:INTEGER_MAX],
        OPERATIONS: ENUM(\READ, \WRITE, \READ_WRITE),
        SIZE: INTEGER [0:INTEGER_MAX],
        VAR LOCKED: BOOLEAN
  END RECORD;

TYPE STREAM_FILE (ELEMENT_TYPE : TYPE) =
  RECORD
    VAR BODY: FILE (ELEMENT_TYPE)
  END RECORD;

TYPE RANDOM_FILE (ELEMENT_TYPE : TYPE,
                  NUM_OF_ELEM: INTEGER [0:INTEGER_MAX]) =
  %Random file elements are accessed via logical
  %addresses in the range [0:NUM_OF_ELEM-1].
  RECORD
    VAR BODY: FILE (ELEMENT_TYPE)
  END RECORD;

```

```

PROCEDURE CLEAR (#SF: STREAM_FILE(?));
  $CLEAR must be called prior to OPEN and CLOSE
  SF.BODY.OPEN := FALSE;
END PROCEDURE CLEAR;

PROCEDURE CLEAR (# RF: RANDOM_FILE (?,?));
  $Overloaded case of CLEAR for random files.
  RF.BODY.OPEN := FALSE;
END PROCEDURE CLEAR;

$ The overloaded cases of the OPEN and CLOSE procedures for stream and random
$ files utilize corresponding FOPEN and FCLOSE routines for the underlying
$ FILE objects. These procedures rely heavily on functional capabilities
$ imported from an independent file-system segment. Observe that a 'standard'
$ file system could also be defined for the language, but that is beyond the
$ scope of this IO illustration.

PROCEDURE FOPEN (PATHNAME: ASCII_STR(PATHNAME_SIZE),
                 VINTAGE: (\NEW, \OLD),
                 OPERATIONS: (\READ, \WRITE, \READ_WRITE),
                 SIZE: INTEGER [0:INTEGER_MAX],
                 LOCKED: BOOLEAN
                 #
                 F:FILE(?))
  SIGNALS (IO_ERROR);

  IF F.OPEN THEN          $field must be set
    SIGNAL IO-ERROR;      $by CLEAR prior to OPEN
  END IF;

  IF VINTAGE = \NEW THEN
    FSYS_CREATE( PATHNAME,
                  SIZE,
                  OPERATIONS,           $write onto file header
                  LOCKED,              $write onto file header
                  #
                  F.DEV_NAME,           $names a device class
                  F.DEV_NUMBER,         $number within a class
                  F.FILE_ADDRESS);    $file address on device
  $ FSYS_CREATE creates a new file, with an appropriate 'header', in the
  $ external file system -- exceptions are signalled for invalid pathnames,
  $ pathnames for existing files or devices, SIZE amount of storage not
  $ available, OPERATIONS inconsistent with the device, and so on.

```

```

    ELSE          $VINTAGE = \OLD
        FYS_CHECK (PATHNAME,
                    SIZE,
                    OPERATIONS,
                    LOCKED
                    ##
                    F.DEV_NAME,
                    F.DEV_NUMBER,
                    F.FILE_ADDRESS);

    % FYS_CHECK checks an old file or device specified by a PATHNAME and returns
    % its device name (e.g.\LT33), its number within that class of devices, and,
    % for appropriate devices, a file address on the device -- exceptions are
    % signalled for invalid pathnames, syntactically valid pathnames with no
    % corresponding file or device, mismatched SIZE, incompatible OPERATIONS,
    % LOCKED = TRUE and an external entity currently linked to another open file
    % with \WRITE or \READ_WRITE permission, OPERATIONS = \WRITE or \READ_WRITE
    % and an external entity already linked with a lock to another open file, and
    % so on.

        END IF;

        F.PATHNAME := PATHNAME;
        F.OPERATIONS := OPERATIONS;
        F.SIZE := SIZE;
        F.LOCKED := LOCKED;
        F.OPEN := TRUE;

    END PROCEDURE FOPEN;

    PROCEDURE FCLOSE (DISPOSITION: (\SAVE, \DELETE),
                      OPERATIONS: (\READ, \WRITE, \READ_WRITE)
                      #
                      F: FILE(?))
                      SIGNALS (IO_ERROR);

        IF NOT F.OPEN THEN
            SIGNAL IO_ERROR;
        END IF;

        IF DISPOSITION = \SAVE THEN
            FSYS_UPDATE (F.PATHNAME,
                        OPERATIONS,
                        F.LOCKED);

    % FSYS_UPDATE updates the valid operations recorded in the file's header and
    % resets the lock flag on the external entity if that flag was set for this
    % file -- exceptions are signalled for OPERATIONS inconsistent with device,
    % user not allowed to change OPERATIONS permissions, new OPERATIONS
    % inconsistent with those of other files currently linked to the external
    % object, and so on.

```

```

ELSE                      $DISPOSITION = \DELETE
    FSYS_DELETE (F.PATHNAME);
; FSYS_DELETE deletes the entity from the external file system -- exceptions
; are signalled for a user without DELETE permission, an external entity still
; linked to other open files, and so on.
    END IF;

    F.OPEN := FALSE;

    END PROCEDURE FCLOSE;

PROCEDURE OPEN (PATHNAME: ASCII_STR(PATHNAME_SIZE),
               VINTAGE : (\NEW,\OLD),
               OPERATIONS: (\READ, \WRITE, \READ_WRITE),
               LOCKED: BOOLEAN
#
# RF: RANDOM_FILE (?T REP ?R, ?S));

FOPEN (PATHNAME, VINTAGE, OPERATIONS,
       S * R.SIZE, LOCKED
#
# RF.BODY);

END PROCEDURE OPEN;

PROCEDURE OPEN (PATHNAME: ASCII_STR(PATHNAME_SIZE),
               VINTAGE: (\NEW, \OLD),
               OPERATIONS: (\READ, \WRITE, \READ_WRITE),
               SIZE: INTEGER [0:INTEGER_MAX],
               LOCKED: BOOLEAN
#
# SF: STREAM_FILE (?T REP ?R));

FOPEN (PATHNAME, VINTAGE, OPERATIONS,
       SIZE * R.SIZE, LOCKED
#
# SF.BODY);

END PROCEDURE OPEN;

PROCEDURE CLOSE (DISPOSITION: (\SAVE, \DELETE),
                 OPERATIONS: (\READ, \WRITE, \READ_WRITE)
#
# RF: RANDOM_FILE (?,?));

FCLOSE (DISPOSITION, OPERATIONS # RF.BODY);

END PROCEDURE CLOSE;

```

```
PROCEDURE CLOSE (DISPOSITION: (\SAVE, \DELETE),
                 OPERATIONS: (\READ, \WRITE, \READ_WRITE)
                 #
                 SF: STREAM_FILE (?));
FCLOSE (DISPOSITION, OPERATIONS # SF.BODY);
END PROCEDURE CLOSE;

% The GET procedure also has two overloaded cases which are promoted -- one
% for stream files and one for random files. The fact that they map to the
% same underlying IO primitives is hidden by the module.

PROCEDURE GET (SF: STREAM_FILE (?T REP ?R)
               #
               VAL: ?T REP ?R)
IMPORTS (ROUTINES:(GET_IN));
GET_IN (SF.BODY ## VAL);

END PROCEDURE GET;

PROCEDURE GET (RF: RANDOM_FILE (?T REP ?R, ?),
               IND: INTEGER [0:INTEGER_MAX]
               #
               VAL: ?T REP ?R)
IMPORTS (ROUTINES: (GET_IN, POSITION));

POSITION (RF, IND);
GET_IN (RF.BODY ## VAL);

END PROCEDURE GET;

% The overloaded PUT procedure is similar to GET, except that the value is
% output.

PROCEDURE PUT (SF: STREAM_FILE (?T REP ?R),
               VAL: ?T REP ?R)
IMPORTS (ROUTINES: (PUT_OUT));
PUT_OUT(SF.BODY, VAL);

END PROCEDURE PUT;
```

```

PROCEDURE PUT(RF: RANDOM_FILE (?T REP ?R, ?),
              IND: INTEGER [0:INTEGER_MAX],
              VAL: ?T REP ?R)
IMPORTS (ROUTINES: (PUT_OUT, POSITION));

POSITION (RF, IND);
PUT_OUT(RF.BODY, VAL);

END PROCEDURE PUT;

PROCEDURE GET_IN (F: FILE(?T REP ?R)
                  ##
                  VAL: ?T REP ?R)

$ The GET_IN procedure provides the interface between the lower-level IN
$ procedure and the overloaded cases of GET for stream and random files.
$ Observe that the types of objects read in must agree with the type of
$ elements in the file.

WHERE T NEEDS (STORE_(T##T))
IMPORTS (ROUTINES:(IN));

VAR BS: BIT (R.SIZE);

IN (F ## BS);           $read in the bit-string rep

SELECT OBJECT_MACHINE FROM      %interpret machine-dependent
PDP_11 -->                   %bit representation according to T
    VAL := [T REP R]$$$(BS);
    END;
UYK_99 -->
    VAL := [T REP R]$$$(BS);      %different BS length check
    END;
END SELECT;

END PROCEDURE GET_IN;

```

```
PROCEDURE PUT_OUT (F: FILE(?T REP ?R),
                   VAL: ?T REP ?R)
```

% The PUT\_OUT procedure is similar to GET\_IN except that it provides an  
% interface between the lower-level OUT procedure and the overloaded cases of  
% PUT.

```
WHERE T NEEDS (TO_BIT (T REP R) RETURNS (BIT(?)));
% which is provided for built-in types; overloaded cases of TO_BIT must be
% supplied for user-defined types desiring output capability.
```

```
IMPORTS (ROUTINES:(OUT));
OUT (F, TO_BIT(VAL));
END PROCEDURE PUT_OUT;
```

```

PROCEDURE IN (F:FILE (?)
    ##
    BS: BIT(?LEN))
IMPORTS (ROUTINES: (DEVICE_IN))
SIGNALS (IO_ERROR);

```

% The IN procedure reads in a bit string from the physical file or device  
% associated with the FILE object. The 'current' position of the file is  
% used; random files are assumed to be positioned at the desired logical  
% address. If a stream file is positioned at the 'end-of-file', an exception  
% will be raised by the lower-level device handler. The number of bits to be  
% read is determined by the size of the bit string parameter.

```

IF NOT F.OPEN THEN
    SIGNAL IO_ERROR;           %unopened file
    END IF;
IF F.OPERATIONS <> \READ & F.OPERATIONS <> \READ_WRITE THEN
    SIGNAL IO_ERROR;           %no read permission
    END IF;

```

```
SELECT F.DEV_NAME FROM
```

```
\LT33 -->
    DEVICE_IN (LT33_DEVICE$NEW (F.DEV_NUMBER)
        ##
        BS);
END;
```

% Note: No end-in-file exception at lower levels of LT33 case -- the 'EOF'  
% character is read like any other.

```
\CR11 -->
    DEVICE_IN (CR11_DEVICE$NEW(F.DEV_NUMBER)
        ##
        BS);
END;
```

```
%etc. for other devices
```

```
ELSE --> SIGNAL IO_ERROR;           %unsupported device
END;
```

```
END SELECT;
```

```
END PROCEDURE IN;
```

```
PROCEDURE OUT (F:FILE (?),
               BS: BIT (?LEN))
IMPORTS(ROUTINES:(DEVICE_OUT))
SIGNALS (IO_ERROR);
```

% The OUT procedure is similar to IN, except that the bit string is output to  
% a physical file or device.

```
IF NOT F.OPEN THEN
  SIGNAL IO_ERROR;           %unopened file
  END IF;
IF F.OPERATIONS <> \WRITE & F.OPERATIONS <> \READ_WRITE THEN
  SIGNAL IO_ERROR;           %no write permission
  END IF;

SELECT F.DEV_NAME FROM
\LT33 --> DEVICE_OUT(LT33_DEVICE$NEW(F.DEV_NUMBER), BS);
END;

%etcetera for other devices

ELSE --> SIGNAL IO_ERROR;      %unsupported device
END;

END SELECT;

END PROCEDURE OUT;
```

```
PROCEDURE REWIND (SF: STREAM_FILE(?))
    SIGNALS(IO_ERROR);

    IF NOT SF.BODY.OPEN THEN
        SIGNAL IO_ERROR;           $unopened file
        END IF;

    ; Select an appropriate call on a device-driver to rewind the file -- this
    ; will (ultimately) result in a call on one of the overloaded cases of
    ; START_IO.

        SELECT SF.BODY.DEV_NAME FROM
            \LT33 --> SIGNAL IO_ERROR;           $operation invalid for the device
            END;

    ; etcetera for other devices, calling appropriate device routine or
    ; signalling exception

        ELSE --> SIGNAL IO_ERROR;           $unsupported device
        END;

    END SELECT;

END PROCEDURE REWIND;
```

```
PROCEDURE POSITION (RF: RANDOM_FILE (?T REP ?R, ?S),
                    IND: INTEGER [0:INTEGER_MAX])
  SIGNALS(IO_ERROR);

  IF NOT RF.BODY.OPEN THEN
    SIGNAL IO_ERROR;           %unopened file
    END IF;
  IF IND >= S THEN
    SIGNAL IO_ERROR;           %illegal element index
    END IF;

  % Select appropriate call of a device-specific routine or START_IO case
  % according to the type of device.

  SELECT RF.BODY.DEV_NAME FROM

    \LT33 --> SIGNAL IO_ERROR;           %can't position teletype
    END;

  % etcetera for other devices, calling appropriate routine to position them --
  % note, the physical address is computed from the file address
  % (RF.BODY.FILE_ADDRESS) plus some offset (e.g., R.SIZE * IND).

  ELSE --> SIGNAL IO_ERROR;           %unsupported device.
  END;

  END SELECT;

END PROCEDURE POSITION;

% Note -- unlike REWIND, POSITION is not exported from the module since its
% only meaningful use is prior to an IN or OUT call, which are also not
% directly accessible. POSITION, IN, and OUT are indirectly invoked via
% higher-level calls on the exported procedures GET and PUT.

END MODULE BINARY_IO;
```

## MODULE TEXTUAL\_IO

% This module defines procedures for transferring data to and from files of  
 % characters -- i.e., those with type STREAM\_FILE(ASCII\_STR(1)). They are  
 % built on the more 'primitive' routines for binary IO, and handle both  
 % unformatted and formatted text.

```

IMPORTS (MODULES:(BINARY_IO))

EXPORTS (NEWLINE,
         PRINT, FORMPRINT,
         READ, FORMREAD);

CONST CHAR_MAX: INTEGER := (ASCII_STR.LENGTH).MAX;

PROCEDURE NEWLINE (SF: STREAM_FILE(ASCII_STR(1)))
  IMPORTS (ROUTINES:(PUT));

    PUT(SF,''\CR);
    PUT(SF,''\LF);

  END PROCEDURE NEWLINE;

PROCEDURE STRPUT(SF: STREAM_FILE(ASCII_STR(1)),
                 STR: ASCII_STR(?S))
  IMPORTS(ROUTINES:(PUT));

  LOOP FOR I IN [0:S-1] DO
    PUT(SF,STR[I:I]);
  END LOOP I;

  END PROCEDURE STRPUT;

PROCEDURE PRINT (SF: STREAM_FILE(ASCII_STR(1)),
                VAL: ?T REP ?R)

  % PRINT assumes the output is to be ASCII and thus can use 'TO_CHAR' routines
  % that produce character strings whose bit representation reflects an ASCII
  % encoding.

  WHERE T NEEDS (TO_CHAR(T REP R)
                  RETURNS (ASCII_STR(?)))
    IMPORTS(ROUTINES:(STRPUT));

    STRPUT(SF,TO_CHAR(VAL));

  END PROCEDURE PRINT;

```

```

PROCEDURE FORMPRINT (SF: STREAM_FILE(ASCII_STR(1)),
                     FORMAT : ASCII_STR(?FS))

% This overloaded case of the FORMPRINT procedure is used for doing
% formatting that is independent of any output value -- such as issuing
% spaces, linefeeds, and so on.

IMPORTS(ROUTINES:(STRPUT))
SIGNALS(IO_ERROR);

VAR INDEX : INTEGER [0:FS] := 0;
VAR AHEAD : INTEGER [1:FS];
VAR NUM_OF_ITERATIONS : INTEGER [0:INTEGER_MAX];

LOOP UNTIL INDEX = FS DO

    AHEAD := INDEX + 1;

    LOOP UNTIL (AHEAD = FS !
                 NOT DIGIT(FORMAT[AHEAD:AHEAD])) DO
        AHEAD := AHEAD + 1;
    END LOOP;

    IF AHEAD > INDEX + 1 THEN
        NUM_OF_ITERATIONS :=
            TO_FIXED(FORMAT[INDEX+1:AHEAD-1],0);
    ELSE
        NUM_OF_ITERATIONS := 1;
    END IF;

    BLOCK
        CONST N_O_I : INTEGER := NUM_OF_ITERATIONS;

        SELECT FORMAT [INDEX:INDEX] FROM

            'S' -->                               $skip spaces
                STRPUT (SF, STRDUP ('', N_O_I));
                END;
            'L' -->                               $line feed
                STRPUT (SF, STRDUP (''LF, N_O_I));
                END;
            'N' -->                               $new line
                STRPUT (SF, STRDUP (''CR LF, N_O_I));
                END;

% etcetera for other format indicators

        ELSE --> SIGNAL IO_ERROR           $illegal format string
        END;

        END SELECT;

    END BLOCK;

```

```

INDEX := AHEAD;

END LOOP;

$ end of FORMPRINT body

FUNCTION DIGIT (C: ASCII_STR(1)) RETURNS (BOOLEAN);

    SELECT C FROM

    '0', '1', '2', '3', '4',
    '5', '6', '7', '8', '9' -->
        RETURN (TRUE);
    END;

ELSE -->
    RETURN (FALSE);
END;

END SELECT;

END FUNCTION DIGIT;

FUNCTION STRDUP (S: ASCII_STR(?L),
                  N: INTEGER [0: INTEGER_MAX])
    RETURNS (ASCII_STR(N*L));

VAR RS: ASCII_STR(N*L);
VAR I: INTEGER [0:N*L] := 0;

IF (N*L = 0) THEN
    RETURN('');
END IF;

LOOP
    RS[I:I+L-1] := S;
    I := I + L;
END LOOP WHEN I = N*L;

RETURN (RS);

END FUNCTION STRDUP;

END PROCEDURE FORMPRINT;

```

```

PROCEDURE FORMPRINT (SF: STREAM_FILE(ASCII_STR(1)),
                     FORMAT: ASCII_STR(?FS),
                     VAL : ?T REP ?R)

# This overloaded case of the FORMPRINT procedure outputs a formatted string
# representation of its VAL parameter. The FORMAT specification string
# consists of three delimited, possibly-empty substrings in the form:

#
#           [fstr #] vstr [# fstr]
#
# If present the fstr substrings are used to output value-independent format
# controls before and after the value is output. The vstr substring is passed
# to a TO_FORM_CHAR routine that must be supplied for each type on which
# formatted output is desired. These routines produce character-string
# representations for values of the types. They are similar to the TO_CHAR
# routines, except that they return character strings of differing values and
# generic lengths depending upon the specific format for the value being
# converted. Because the returned lengths must be generically determined, a
# length value must be provided as an input parameter. This length is
# computed by a FORM_CHAR_LEN function, also overloaded on the type of VAL.
# Examples of these functions' headers for one overloaded case are:
#
# FUNCTION TO_FORM_CHAR (FORMAT: ASCII_STR(?FS),
#                   LEN : INTEGER [0: CHAR_MAX],
#                   VAL: SOME_TYPE REP ?R)
#           RETURNS (ASCII_STR(LEN));
#
# and
#
# FUNCTION FORM_CHAR_LEN (FORMAT: ASCII_STR(?FS),
#                   VAL: SOME_TYPE REP ?R)
#           RETURNS (INTEGER [0:CHAR_MAX]);
#
#     * * * *
#
WHERE T NEEDS (
    TO_FORM_CHAR (ASCII_STR(?),
                  INTEGER [0: CHAR_MAX],
                  T REP R)
    RETURNS (ASCII_STR(?)),
    FORM_CHAR_LEN (ASCII_STR(?),
                  T REP R)
    RETURNS (INTEGER [0: CHAR_MAX]))


IMPORTS (ROUTINES: (FORMPRINT,  #the other overloaded case
                    STRPUT));


CONST DELIM: ASCII_STR(1) := '#';

VAR I      : INTEGER[0:FS] := 0;
VAR MARK : INTEGER[1:FS];

```

```

IF I=FS then                      $null format string
    CONVERT_AND_PUT(SF,'', VAL);
    RETURN;
END IF;

LOOP UNTIL (I=FS ! FORMAT[I:I] = DELIM) DO
    I := I + 1;                      $search for delimiter
END LOOP;

IF I = FS THEN                     %only value format string
    CONVERT_AND_PUT(SF, FORMAT[0:I-1],VAL);
    RETURN;
END IF;

% else I points to first delimiter

IF I > 0 THEN                     %format preceding value part
    FORMPRINT (SF, FORMAT[0:I-1]);
    %continue -- no return
    END IF;

    I := I + 1;
    MARK := I;

    IF I = FS THEN                 %remaining string, including
        CONVERT_AND_PUT(SF, '', VAL);      %value part, null
        RETURN;
    END IF;

    LOOP UNTIL (I = FS ! FORMAT[I:I] = DELIM) DO
        I := I + 1;                  $search for delimiter
    END LOOP;

    IF I=FS THEN                   %only value part remains
        CONVERT_AND_PUT(SF, FORMAT[MARK:I-1], VAL);
        RETURN;
    END IF;

% else I points to second delimiter

    IF I > MARK THEN             %non-null value part
        CONVERT_AND_PUT(SF, FORMAT [MARK:I-1], VAL);
    ELSE
        CONVERT_AND_PUT(SF,'',VAL);
    END IF;

    I := I + 1;

```

```

        IF I < FS THEN           $format following value part
          FORMPRINT (SF, FORMAT [I: FS-1]);
        END IF;

$ end body of FORMPRINT

PROCEDURE CONVERT_AND_PUT (SF:STREAM_FILE(ASCII_STR(1)),
                           FORMAT: ASCII_STR(?),
                           VAL : ?T REP ?R)

$ The TO_FORM_CHAR and FORM_CHAR_LEN routines used here must be taken from
$ the point of call within FORMPRINT, and thus are indirectly taken from the
$ point where FORMPRINT is called.

      WHERE T NEEDS (
        TO_FORM_CHAR (ASCII_STR(?),
                      INTEGER [0:CHAR_MAX],
                      T REP R)
        RETURNS (ASCII_STR(?)),
        FORM_CHAR_LEN (ASCII_STR(?),
                      T REP R)
        RETURNS (INTEGER [0:CHAR_MAX])))

      IMPORTS (ROUTINES:(STRPUT));

$ need constant for TO_FORM_CHAR
      CONST LEN: INTEGER
            := FORM_CHAR_LEN (FORMAT, VAL);

      STRPUT (SF, TO_FORM_CHAR(FORMAT,
                                LEN,
                                VAL));

    END PROCEDURE CONVERT_AND_PUT;

  END PROCEDURE FORMPRINT;

PROCEDURE STRGET (SF: STREAM_FILE(ASCII_STR(1))
                  ##
                  LEN: INTEGER [0:CHAR_MAX],
                  STR: ASCII_STR(CHAR_MAX))
  IMPORTS (ROUTINES: (GET));

$ This overloaded case of the STRGET routine is used to read in an arbitrary
$ number of characters, until a delimiting character is found in the input
$ stream. It returns the number and values of the characters it has read.
$ This case of STRGET is used by the READ routine for unformatted text.

```

```

VAR C: ASCII_STR(1);

LOOP
    GET (SF ## C);
    END LOOP WHEN NOT DELIMITER(C);

LEN := 0;

LOOP
    STR[LEN:LEN] := C;
    LEN := LEN +1;
    GET (SF ## C);
    END LOOP WHEN DELIMITER(C);

% end of STRGET body

FUNCTION DELIMITER (C: ASCII_STR(1)) RETURNS(BOOLEAN);

    SELECT C FROM
        ' ', ''CR, ''LF -->
            RETURN (TRUE);
            END;
    ELSE -->
            RETURN (FALSE);
            END;

    END SELECT;

END FUNCTION DELIMITER;

END PROCEDURE STRGET;

FUNCTION STRGET (SF: STREAM_FILE(ASCII_STR(1)),
                 LEN: INTEGER [0:CHAR_MAX])
    RETURNS (ASCII_STR(LEN))
    IMPORTS (ROUTINES:(GET));

% This overloaded case of the STRGET routine is used to read in a specified
% number of arbitrary characters. It is used by the FORMREAD routines for
% formatted text.

    VAR STR: ASCII_STR(LEN);

    LOOP FOR I IN [0:LEN-1] DO
        GET (SF ## STR[I:I]);
    END LOOP;

    RETURN (STR);

END FUNCTION STRGET;

```

```
PROCEDURE READ (SF: STREAM_FILE (ASCII_STR(1))
    ##  
    VAL: ?T REP ?R)
```

% The READ procedure for unformatted text is similar to PRINT, except that  
% the values are input. The size of the character-string representation is  
% inferred by delimiters in the input stream. All types for which text input  
% is desired must have a FROM\_CHAR operation (to perform the inverse of  
% TO\_CHAR). For example, a FROM\_CHAR for the FIXED type can be added to the  
% standard prelude as:

```
%  
% FUNCTION FIXED$FROM_CHAR (STR: ASCII_STR(?), T:TYPE)  
%     RETURNS (T)  
%     WHERE T NEEDS (STORE_(T##T));  
%     VAR VAL: T := TO_FIXED(STR  
%                         VAL.SCALE,  
%                         VAL.RADIX,  
%                         VAL.PRECISION);  
%  
%     RETURN (VAL);  
% END FUNCTION FIXED$FROM_CHAR;  
%
```

% Similar routines must be defined for other types.

```
% * * * *
```

```
WHERE T NEEDS (FROM_CHAR(ASCII_STR(?),TYPE)  
                RETURNS (T),  
                STORE_(T ## T))  
IMPORTS (ROUTINES: (STRGET));  
  
VAR LEN: INTEGER [0:CHAR_MAX];  
VAR STR: ASCII_STR(CHAR_MAX);  
  
STRGET (SF ## LEN, STR);  
  
VAL := T$FROM_CHAR (STR[0:LEN- .], T REP R);  
  
END PROCEDURE READ;
```

```

PROCEDURE FORMREAD (SF: STREAM_FILE (ASCII_STR(1)),
                    FORMAT: ASCII_STR(?FS))
                    IMPORTS (ROUTINES:(STRGET));

% This overloaded case of the FORMREAD procedure is similar to FORMPRINT for
% characters that are interspersed between those representing values in the
% stream file. In the current example, the format simply specifies a number
% of characters to be "gobbled up". However, a more sophisticated routine can
% be written to check the input characters for consistency with tightly-
% constrained format specifiers and signal errors when detected.

% need constant for STRGET call
CONST LEN: INTEGER [0:CHAR_MAX]
      := TO_FIXED (FORMAT,0);

VAR S: ASCII_STR(LEN) := STRGET(SF, LEN);

% STRGET accomplishes desired action

END PROCEDURE FORM_READ;

PROCEDURE FORMREAD (SF:STREAM_FILE(ASCII_STR(1)),
                     FORMAT: ASCII_STR(?FS)
                     ##
                     VAL: ?T REP ?R)

% This overloaded case of FORMREAD reads in a formatted character string
% representation for its VAL parameter. The format string is parsed for
% delimiters in the same way as the corresponding case of FORMPRINT. As with
% the unformatted READ procedure, a character-string conversion function must
% be provided for types on which formatted input is desired. A simple example
% of such a formatted conversion for FIXED is:

%
% FUNCTION FIXED$FROM_FORM_CHAR
%     (FORMAT: ASCII_STR(?FS),
%      T: TYPE,
%      STR: ASCII_STR(?LEN))
%     RETURNS (T)
%     WHERE T NEEDS (STORE_(T ## T))
%     SIGNALS (CONVERSION_ERROR,
%              FORMAT_ERROR);
%

```

```

3
3     VAR VAL: T;
3     IF (FS=0 ! FORMAT[0:0] = 'S') THEN
3         VAL := TO_FIXED(STR,0);
3     ORIF (FORMAT[0:0] = 'Q') THEN
3         IF (LEN < 3 ! STR[0:0] <> '') !
3             STR[LEN-1:LEN-1] <> '') THEN
3                 SIGNAL CONVERSION_ERROR;
3             ELSE
3                 VAL := TO_FIXED (STR[1:LEN-2],
3                                 VAL.SCALE,
3                                 VAL.RADIX,
3                                 VAL.PRECISION);
3             END IF;
3         ELSE
3             SIGNAL FORMAT_ERROR;
3             END IF;
3         RETURN (VAL);
3     END FUNCTION FIXED$FROM_FORM_CHAR;
3
3

```

% Obviously conversions with more sophisticated formats could be defined.

% Also required by FORMREAD is a type-specific FORM\_CHAR\_LEN routine that  
% computes the length of the formatted character representation to be read.  
% The computation is based upon the format specified and upon the actual type  
% (including representation information) of the value being input. Unlike  
% those overloaded FORM\_CHAR\_LEN routines used by FORMPRINT, the cases used  
% for input cannot use the value of the type. That is, for a given format and  
% type, a fixed number of characters are read independent of the inter-  
% pretation of individual characters. More complicated input routines can be  
% written to read in a variable number of characters by reading and  
% interpreting them (according to the format) one at a time. However, such  
% routines are type-dependent and are more appropriately defined with the  
% type, using the standard GET routine to retrieve the individual characters.

```

WHERE T NEEDS (
    STORE_(T ## T),
    FROM_FORM_CHAR(ASCII_STR(?),
                  TYPE,
                  ASCII_STR(?)),
    RETURNS (T),
    FORM_CHAR_LEN (ASCII_STR(?),
                  TYPE)
    RETURNS (INTEGER [0:CHAR_MAX]))
IMPORTS (ROUTINES:(STRGET));

CONST DELIM: ASCII_STR(1) := '#';

VAR I    : INTEGER [0:FS] := 0;
VAR MARK: INTEGER [1:FS];

```

```

IF I=FS THEN          %null format string
  GET_AND_CONVERT (SF, '' ## VAL);
  RETURN;
END IF;

LOOP UNTIL (I=FS ! FORMAT [I:I] = DELIM) DO
  I := I+1;           %search for delimiter
END LOOP;

IF I=FS THEN          %only value format string
  GET_AND_CONVERT(SF, FORMAT[0:I-1]## VAL);
  RETURN;
END IF;

% else I points to first delimiter

IF I > 0 THEN          %format preceding value part
  FORMREAD (SF, FORMAT [0:I-1]);
  %continue -- no return
END IF;

I := I+1;
MARK := I;

IF I=FS THEN          %remaining string, including
  GET_AND_CONVERT (SF, '' ## VAL);    %value part, null
  RETURN;
END IF;

LOOP UNTIL (I=FS ! FORMAT [I:I] = DELIM) DO
  I := I+1;           %search for delimiter
END LOOP;

IF I = FS THEN          %only value part remains
  GET_AND_CONVERT (SF, FORMAT[MARK:I-1] ## VAL);
  RETURN;
END IF;

% else I points to second delimiter

IF I > MARK THEN      %non-null value part
  GET_AND_CONVERT (SF, FORMAT [MARK:I-1] ## VAL);
ELSE
  GET_AND_CONVERT(SF, '' ## VAL);
END IF;

I := I + 1;

IF I < FS THEN          %format following value part
  FORMREAD (SF, FORMAT[I:FS-1]);
END IF;

% end of FORMREAD body

```

```

PROCEDURE GET_AND_CONVERT (SF: STREAM_FILE(ASCII_STR(1)),
                           FORMAT : ASCII_STR(?)
                           ##  

                           VAL : ?T REP ?R)

$ The FROM_FORM_CHAR and FORM_CHAR_LEN routines used here are taken from the
$ point of call within FORMREAD, and thus indirectly from the point where
$ FORMREAD is called. (cf. CONVERT_AND_PUT in FORMPRINT)

      WHERE T NEEDS (
        STORE_(T ## T),
        FROM_FORM_CHAR(ASCII_STR(?),
                      TYPE,
                      ASCII_STR(?))
        RETURNS (T),
        FORM_CHAR_LEN (ASCII_STR(?),
                      TYPE)
        RETURNS (INTEGER [0:CHAR_MAX]))
      IMPORTS (ROUTINES:(STRGET));

$ constant needed in STRGET call
      CONST LEN : INTEGER
                  := T$FORM_CHAR_LEN(FORMAT, T REP R);

      VAL := T$FROM_FORM_CHAR(FORMAT, T REP R,
                             STRGET (SF,LEN));

    END PROCEDURE GET_AND_CONVERT;

  END PROCEDURE FORMREAD;

END MODULE TEXTUAL_IO;

END SEGMENT STANDARD_IO;

```

## APPENDIX D

### The Basic Semantic Framework

#### D.1. VALUES AND VARIABLES

The universe of data values on which programs operate is divided into subsets called types. Examples of types are ARRAY [1:100] OF BOOLEAN, and INTEGER [-200 : +200]. These are examples of built-in types, defined automatically by the language; in addition, users may define new types by using the module mechanism. Associated with each type is a set of operations (procedures and functions) that manipulate the type's values. It is the operations that give the values their meaning; the operations define how the values can be interpreted and manipulated.

Values reside in variables. Variables are also typed, and the type of the variable and the contained value must match exactly (type equality is defined below) or be compatible as was defined for variant records in Section 3.9. Variables are created by declarations. Declarations state the type of the variable being created, plus some additional information, namely its storage class, and whether it is CONST or VAR. The type information in the declaration, together with the known relationship between the type of the variable and the type of the contained value, permit the compiler to do type checking. Normally a variable (and its value) is of just one type in a scope. The one exception is within a module defining the type, where the variable is considered to belong both to the abstract type being defined and to the type chosen to represent that abstract type.

A value is placed in a variable by means of a primitive operation called bit-copy. Bit-copy can only be used when both variable and value have the same type (or compatible types in the case of variant records); its effect is to copy the value bit by bit into the variable. Bit-copy is the basis of the

parameter passing mechanism, as will be discussed below. It is not the assignment operation (i.e., the operation performed to carry out `:=`).

Usually, a variable can be thought of as a generalized memory cell, large enough to hold a value of its type, but with no interior structure (no component variables). However, record, array, and pointer variables do have interior structure. Record and array variables are viewed as collections of (the component) variables, while the (heap) variable pointed at is visible through a pointer.

## D.2. PROCEDURES

Procedure and function invocation is the fundamental activity taking place in our programs. All expressions containing prefix and infix operators (with the exception of the conditional-and (`&`) and conditional-or (`!`) operators) are semantically compositions of functions that implement those operators, and even assignment is interpreted this way. For example, if `X` and `Y` are `FIXED` values, then `X < Y` invokes the `FIXED$LESS_THAN_` function, and `X := Y` invokes the `FIXED$STORE_` procedure. (Of course, such invocations will often be compiled in line.)

Procedures and functions communicate by sharing variables and by bit-copying values. There are three kinds of parameters, with the following meanings:

by value: the value resulting from evaluating the argument is bit copied into the formal parameter

by reference: the formal parameter is a new name for the argument (which must be a variable)

by result: at return, the value in the formal parameter is bit-copied into the argument (which must be a variable).

Functions return values; the returned value is bit-copied into a variable in the caller.

### D.3. TYPES AND TYPE CLASSES

Types are grouped into sets of related types called type classes. Examples of type classes are FIXED and ARRAY. The types within a type class share certain similarities, but differ in their attributes. For example, all arrays provide storage for elements, but differ in attributes including the amount of storage, the low and high bounds, the dimensions, and the element type.

Two types are equal if they belong to the same type class, and if their attributes are equal. The attributes themselves are limited to being types, or values in the following type classes: FIXED, enumerated, BOOLEAN, and FLOAT.

Each type class provides a set of operations to manipulate values of their member types. Some of these operations may be specialized to manipulate values of just one member type, but in general, the operations may accept values of different member types. Such operations are generic. Generic operations accept generic parameters, which are used to permit a single procedure body to be written that will work for many types of values. A generic is best thought of as a procedure type (like a type class) that defines a set of procedures. Each unique combination of generic actual parameters selects a different member of this class. The members are created by a process called instantiation, in which the actual values of the generic parameters are combined with its body to produce a procedure. (Instantiation was explained in Section 5.) Generic parameters are limited, just like attributes, to being types, BOOLEANS, and so on.

#### D.4. MODULES

Type classes are defined by writing modules. All built-in types are conceptually defined in a single module, the PRELUDE, that exports them all. PRELUDE provides definitions of all the operations mentioned in Section 3.1.4.1. For example, for each type class, it defines a STORE\_ operation, which is invoked whenever an assignment is required. Assignment is interpreted as a call on STORE\_ rather than bit-copy because what should be done on assignment may differ from simply a bit copy; for example, this is true of semaphores.

The language imposes the following constraint on STORE\_: both arguments must belong to the same type class. This constraint permits store to define conversions (as required by the IRONMAN), but limits implicit conversions to members of the same type class.

When the built-in types are to be extended, new operation definitions will be added to the PRELUDE module. When defining new types (as opposed to extending existing types), users must write a new module that exports the type class (e.g., see the varying length string type in Section 5). The advantage of defining an entire type class at once, with operations belonging to the type class rather than the type is that it permits operations to accept parameters of more than one type (in the class). For example, we can define the new type class, MATRIX, as shown in Section 5. Types in this class are MATRIX(M, N) where M and N are FIXED[0:MAX\_INT] and define the number of rows and columns in the matrix.

## D.5. OPERATOR EXTENSION

Each occurrence of an operator in an expression or assignment statement is equivalent to a regular invocation of a named routine (with the exception of the special "short-circuited" operators & and !). Built into the language is a one-to-one mapping between each operator and a unique name. Most of these names have corresponding routines that are predefined as a part of the language. Many of the routines are generic for arguments of arbitrary types within a specific type class; many have overloaded definitions that take arguments of different type classes. By further overloading the definitions of these named routines, the programmer can "extend" the semantics of the language's operators.

The general algorithm for mapping operator invocations to invocations of their corresponding routines is:

### A1) Assignment statements of the form

X := Y  
are mapped to the invocation  
STORE\_(Y ## X)

### A2) Assignment statement of the form

X[L:U] := Y  
are mapped to the invocation  
SUBSTRING\_STORE\_(Y,L,U # X)

### A3) Substring expressions of the form

X[L:U]  
are mapped to the invocation  
SUBSTRING\_(X,L,U)

### A4) Infix expressions of the form

X op Y  
are mapped to an invocation  
OP\_ROUTINE\_(X,Y)  
where OP\_ROUTINE\_ is the named routine corresponding to op.

A5) Unary expressions of the form

op X  
are mapped to an invocation  
OP\_ROUTINE\_(X)  
where OP\_ROUTINE\_ is the named routine for op.

B) If the operator occurs in the context of a precision specification, the specified precision is appended as an additional input argument.

C) If the operator occurs in the context of a rounding specification, the specified rounding is appended as a final additional input argument.

D) Finally, the standard language rules for routine invocation are applied (cf. Section 4.3.4). Observe that the ultimate selection of a specific routine is based solely upon its basic name and arguments; explicit 'T\$name' forms are not used in the operator mappings (cf Section 4.4 on modules).

Observe that paragraph A1 provides for the general assignment case. Paragraphs A2 and A3 provide for both assignment and evaluation of substrings (and slices of user-defined types). Paragraphs A4 and A5 provide for infix and prefix operators.

Paragraphs B and C are used for operators for which precision and rounding are important; for example, these rules are used for the FIXED and FLOAT routines which are overloaded to take two, three, and four parameters. (The cases for two and three parameters are INLINE routines that call the four-parameter routine with default precision and/or rounding specifications.) Other routines that accept these extra arguments can be defined for extended types having operators where rounding and precision are meaningful (e.g., VECTOR and MATRIX).

### Examples

- 1) Operator extensions for general assignment, substring evaluation, and substring assignment are defined for the abstract type VAR\_CHAR\_STR in the variable-length string example in Section 5.
- 2) Matrix multiplication:

```
TYPE MATRIX (ROW: INTEGER [1:INTEGER_MAX],
             COL: INTEGER [1:INTEGER_MAX],
             PREC: INTEGER [1:MAX_NOM_PREC]) =
    ARRAY [1:ROW, 1:COL] OF FLOAT (PREC);

FUNCTION MATRIX$MULTIPLY_(M1:MATRIX (?M, ?N, ?),
                           M2: MATRIX (?N, ?P, ?),
                           RES_PREC: INTEGER [1:MAX_NOM_PREC],
                           RND: ENUM (ROUNDED, TRUNCATED))
RETURNS (MATRIX(M,P,RES_PREC))
SIGNALS (OVERFLOW,UNDERFLOW);

VAR MR: MATRIX (M,P,RES_PREC);
VAR SUM: FLOAT (RES_PREC);

LOOP FOR R IN [1:M] DO
  LOOP FOR C IN [1:P] DO
    SUM := 0.0;
    LOOP FOR T IN [1:N] DO
      SUM := SUM + M1(R,T) * M2(T,C) RND WITH PREC RES_PREC;
      EXCEPT
        OVERFLOW,UNDERFLOW --> SIGNAL;
      END EXCEPT;
    END LOOP T;
    MR(R,C) := SUM;
  END LOOP C;
END LOOP R;

RETURN (MR);

END FUNCTION MATRIX$MULTIPLY_;
```

## D.6. STANDARD PRELUDE OPERATIONS

The standard PRELUDE module conceptually contains the definitions of all of the routines upon which the built-in semantics of the language are based. For example, the module contains the definitions of arithmetic operations for the built-in types. It also contains fundamental routines for implementing statements such as PAR\_BLOCK and procedures such as REQUEST.

It is important to define the signatures of the built-in routines. This will allow the programmer to avoid illegal duplicate definitions when adding additional routines to PRELUDE. Perhaps even more importantly, knowing the signatures is essential to avoid unintentional replacement of the operational semantics for the built-in types via locally-scoped definitions of their operations.

The signatures for all of the built-in routines have not been provided in the current document. However, the following examples are meant to illustrate their basic forms:

```
TYPE ROUNDINGS = ENUM (ROUNDED, TRUNCATED, NATIVE_R, NATIVE_T);

PROCEDURE FLOAT$STORE_(OPERAND_1: FLOAT(?P1) REP FLOAT_STD,
                      RND: ROUNDINGS [ROUNDED:TRUNCATED]
                      ##
                      OPERAND_2: FLOAT (?P2) REP FLOAT_STD)
SIGNALS (RANGE);

$ RND is the explicitly specified rounding from the evaluation context of
$ OPERAND_1, used in a call on TO_FLOAT (OPERAND_1, IMP_PRECISION (P2), RND)
$ when P1 <> P2.
...
```

% Observe that FLOAT\$STORE\_ can be overloaded for FLOAT operands having  
% alternate representations -- for example:

```
PROCEDURE FLOAT$STORE_ (OPERAND_1: FLOAT(?P1) REP SOME_NEWREP,  
                        RND : ROUNDINGS [ROUNDED:TRUNCATED]  
                        ##  
                        OPERAND_2: FLOAT(?P2) REP SOME_NEWREP)  
SIGNALS (RANGE);  
...  
  
INLINE PROCEDURE FLOAT$STORE_ (OPERAND_1: FLOAT(?) REP ?,  
                               ##  
                               OPERAND_2: FLOAT (?) REP ?);  
% invoked when the source expression has no explicit rounding specification;  
% it calls:  
  
  FLOAT$STORE_(OPERAND_1, ROUNDED %default rounding  
  ##  
  OPERAND_2);  
END PROCEDURE FLOAT$STORE_;  
  
CONST MAX_NOM_PREC: INTEGER := (FLOAT.NOM_PRECISION).MAX;  
  
INLINE PROCEDURE FLOAT$STORE_ (OPERAND_1: FLOAT(?) REP ?,  
                               PREC: INTEGER [1:MAX_NOM_PREC],  
                               RND: ROUNDINGS [ROUNDED: TRUNCATED]  
                               ##  
                               OPERAND_2: FLOAT(?) REP ?);  
% The eventual STORE_ routine ignores the source expression's explicitly  
% specified precision, although this may have been used by another operation  
% to determine the precision of the expression's value (i.e., OPERAND_1) which  
% is used by STORE_  
  
  FLOAT$STORE_ (OPERAND_1,RND  
  ##  
  OPERAND_2);  
END PROCEDURE FLOAT$STORE_;  
  
INLINE PROCEDURE FLOAT$STORE_(OPERAND_1: FLOAT(?) REP ?,  
                           PREC: INTEGER [1:MAX_NOM_PREC]  
                           ##  
                           OPERAND_2: FLOAT(?) REP ?);  
  FLOAT$STORE_(OPERAND_1, ROUNDED  
  ##  
  OPERAND_2);  
END PROCEDURE FLOAT$STORE_;
```

```

FUNCTION FLOAT$TO_FLOAT (OPERAND_1: FIXED(?S, ?R, STD),
                        PREC: INTEGER [1:MAX_NOM_PREC],
                        RND: ROUNDINGS [ROUNDED:TRUNCATED])
RETURNS (FLOAT (PREC) [FLOAT_MIN(PREC):
                      FLOAT_MAX(PREC)] REP FLOAT_STD)
SIGNALS (CONVERSION_ERROR);

 $\%$  built-in for STD fixed precision only; actual returned precision will be
 $\%$  IMP_PRECISION (PREC)
...
FUNCTION FLOAT$TO_FLOAT (OPERAND_1: FLOAT (?P) REP ?R,
                        PREC: INTEGER [1:MAX_NOM_PREC],
                        RND: ROUNDINGS [ROUNDED:TRUNCATED])
RETURNS (FLOAT (PREC) [FLOAT_MIN(PREC):
                      FLOAT_MAX(PREC)] REP R)
SIGNALS (CONVERSION_ERROR);
...
FUNCTION FLOAT$TO_FLOAT (OPERAND_1: ASCII_STR(?L) REP ?R,
                        PREC:INTEGER [1:MAX_NOM_PREC],
                        RND: ROUNDINGS [ROUNDED:TRUNCATED])
RETURNS (FLOAT (PREC) [FLOAT_MIN (PREC):
                      FLOAT_MAX(PREC)] REP FLOAT_STD)
SIGNALS (CONVERSION_ERROR);
...
 $\%$  etcetera for other routines; e.g.:

TYPE FIXED_PRECISIONS = ENUM (STD);

FUNCTION FIXED$TO_FIXED (OPERAND_1: ASCII_STR(?L) REP ?R,
                        SCALE: INTEGER [-1000:1000],
                        RADIX: INTEGER [2:1000],
                        PREC: FIXED_PRECISIONS,
                        RND: ROUNDINGS[ROUNDED:TRUNCATED])
RETURNS (FIXED(SCALE,RADIX,PREC)
         [FIXED_MIN (SCALE, RADIX, PREC):
          FIXED_MAX (SCALE, RADIX, PREC)] REP FIXED_STD)
SIGNALS (CONVERSION_ERROR);
...
 $\%$  Note that overloaded cases of TO_FIXED and TO_FLOAT for character strings
 $\%$  define the semantic interpretation of fixed and float literals -- that is,
 $\%$  the characters representing these literals are always implicitly passed to
 $\%$  the appropriate TO_FIXED or TO_FLOAT routine to determine their values. For
 $\%$  example, fixed literals of the form
 $\%$             $\pm$  ddd E  $\pm$  ee
 $\%$  are implicitly regarded as having the return value of
 $\%$            TO_FIXED (' $\pm$  ddd E  $\pm$  ee', 0, 2, STD_ROUNDED)

```

```

%
%
%
%
% Fixed literals which include a scale-spec, i.e., of the form
%      ± ddd.ddd E ± ee\$ ± ss
% are interpreted as the value resulting from
%      TO_FIXED ('± ddd.ddd E ± ee',
%                  TO_FIXED ('±ss', 0, 2, STD, ROUNDED),
%                  2, STD, ROUNDED)
%
% Float literals are similarly handled.

FUNCTION FLOAT$ADD_ (OPERAND_1: FLOAT (?P1) REP FLOAT_STD,
                      OPERAND_2: FLOAT (?P2) REP FLOAT_STD,
                      PREC: INTEGER [1:MAX_NOM_PREC],
                      RND: ROUNDINGS [ROUNDED: TRUNCATED])
RETURNS (FLOAT(PREC) [FLOAT_MIN(PREC): FLOAT_MAX(PREC)] REP FLOAT_STD)
SIGNALS (OVERFLOW);

% observe that ADD_ can be overloaded for
% other reps, or STORE_ can define rep conversions
...
INLINE FUNCTION FLOAT$ADD_ (OPERAND_1: FLOAT (?P1) REP ?,
                            OPERAND_2: FLOAT (?P2) REP ?)
RETURNS (FLOAT (MAX (P1,P2)) [FLOAT_MIN (MAX (P1,P2)):
                                FLOAT_MAX (MAX (P1,P2))] REP FLOAT_STD) ;

RETURN (FLOAT$ADD_ (OPERAND_1, OPERAND_2, ROUNDED, MAX (P1,P2)));

END FUNCTION FLOAT$ADD_;

% etcetera for other overloaded cases of ADD_

% etcetera for other FLOAT operations

FUNCTION FLOAT$REP_SIZE (R: REPTYPE,
                        P: INTEGER [1:MAX_NOM_PREC])
RETURNS (INTEGER [0:INTEGER_MAX]);

VAR T: FLOAT(P) REP R;
RETURN (T.SIZE);
END FUNCTION FLOAT$REP_SIZE;

FUNCTION FLOAT$MD_MULTIPLY_ (OPERAND_1: FLOAT (?P1) REP FLOAT_STD,
                            OPERAND_2: FLOAT (?P2) REP FLOAT_STD,
                            PREC: INTEGER [1:MAX_NOM_PREC],
                            RND: ROUNDINGS [NATIVE_R: NATIVE_T])
RETURNS (BIT (FLOAT$REP_SIZE (FLOAT_STD, PREC)))
SIGNALS (UNDERFLOW);
...

```

```

FUNCTION FIXED$ADD_ (OPERAND_1: FIXED (?S1, ?RX, ?P1) REP FIXED_STD,
                     OPERAND_2: FIXED (?S2, ?RX, ?P2) REP FIXED_STD,
                     PREC: FIXED_PRECISIONS,
                     RND: ROUNDINGS [ROUNDED:TRUNCATED])
RETURNS (FIXED (MAX (S1,S2), RX, MAX (P1,P2))
          [FIXED_MIN (MAX (S1,S2), RX, MAX (P1,P2)):
           FIXED_MAX (MAX (S1,S2), RX, MAX (P1,P2))] REP FIXED_STD)
SIGNALS (OVERFLOW);
% Note: only for equal radices
...
FUNCTION FIXED$MULTIPLY_ (OPERAND_1: FIXED (?S1, ?RX, ?P1) REP FIXED_STD,
                           OPERAND_2: FIXED (?S2, ?RX, ?P2) REP FIXED_STD,
                           PREC: FIXED_PRECISIONS,
                           RND: ROUNDINGS [ROUNDED:TRUNCATED])
RETURNS (FIXED (S1+S2, RX, MAX(P1,P2))
          [FIXED_MIN (S1+S2, RX, MAX (P1,P2)):
           FIXED_MAX (S1+S2, RX, MAX (P1,P2))] REP FIXED_STD)
...

```

% and so on ...

#### D.7. PARALLELISM

All of the language features for parallelism are invocations of routines exported from a PARALLEL\_SUPPORT module that is nested within the standard PRELUDE. Should a particular implementation wish to change the scheduling discipline, only the routines in the PARALLEL\_SUPPORT module need to be redefined. These routines and their parameters are listed here:

```

PROCEDURE PAR_BLOCK_ (PATH_LIST: ARRAY [1:?N] OF RECORD;
                      VAR PATH_ADDR: BIT(ADDR_SIZE);
                      VAR ACTIVATION_ID: INTEGER;
                      VAR PRIORITY_LEVEL: INTEGER
                           [MIN_PRIORITY:MAX_PRIORITY];
                      VAR CONNECT_NAME: ASCII_STR(?LEN)
END RECORD);

```

The PATH\_LIST array contains an entry for each path activation being created by the PAR\_BLOCK statement. The PATH\_ADDR is a compiler-generated implementation-independent pointer to the code for the path. The ACTIVATION\_ID is a compiler-generated unique id for each activation (incorporating the name of the path and its subscript in the case of a multiply-activated path). If

no priority is specified in the path-header, the compiler supplies a default priority of MIN\_PRIORITY. If a CONNECT clause is not present in the path-header, the compiler supplies a blank string for the CONNECT\_NAME.

```
PROCEDURE WAIT (TIME: FIXED(?SCALE,2,STD));
```

The standard WAIT procedure is generic with regard to the scale of the TIME parameter (and may be extended by an implementation to be generic with respect to the precision and radix of this parameter). This is intended to provide flexibility to the invoker of the routine who can thus supply an argument in the most convenient form.

```
PROCEDURE TERMINATE (ACTIVATION_ID: INTEGER);
```

and

```
PROCEDURE TERMINATE ();
```

As with PAR\_BLOCK, the ACTIVATION\_ID is generated by the compiler for the path-reference given in the statement. (In the case of a subscripted reference, code to generate the id at run-time may be needed.) The reason for the two forms of this routine (and also for the PRIORITY and CLOCK routines described below) is that the ACTIVATION\_ID becomes an optional parameter to the routine. When this parameter is not supplied, the parameter-less form of the routine can determine at run-time which path-activation invoked it, but the compiler cannot necessarily determine this (or generate code to determine this). For example, the TERMINATE statement may be executed in a routine which is not lexically nested in a path.

```
PROCEDURE PRIORITY (PRIORITY_LEVEL: INTEGER  
                     [MIN_PRIORITY:MAX_PRIORITY],  
                     ACTIVATION_ID: INTEGER);
```

and

```
PROCEDURE PRIORITY (PRIORITY_LEVEL: INTEGER  
                     [MIN_PRIORITY:MAX_PRIORITY]);
```

```
FUNCTION CLOCK (ACTIVATION_ID: INTEGER)  
RETURNS (FIXED(CLOCK_SCALE,2,STD)[0:FIXED_MAX(CLOCK_SCALE,2,STD)]);
```

and

```
FUNCTION CLOCK ()  
RETURNS (FIXED(CLOCK_SCALE,2,STD)[0:FIXED_MAX(CLOCK_SCALE,2,STD)]);
```

For maximum flexibility, the CLOCK function returns its value with scale equal to the implementation-defined CLOCK\_SCALE. The user of the function can determine this scale with an attribute query or merely convert it to another scale by using the TO\_FIXED function.

```
PROCEDURE REQUEST (#S: SEMAPHORE(?SEM_TYPE));
```

```
PROCEDURE RELEASE (#S: SEMAPHORE(?SEM_TYPE));
```