YELLOW

PRELIMINARY

DESIGN PHASE

REPORT

AND

LANGUAGE SPECIFICATION

CONTENTS

Contents

PART ONE

INTRODUCTION

CONTENTS

I SUMMARY

This document presents a language designed for the programming of embedded computer systems within the Department of Defense. This language is also appropriate for the implementation of many components of its own support environment—editors, interpreters, diagnostic aids, etc. We have found the Ironman Requirements [10] to be a successful analysis of the language needs of the embedded system and language support domains, and believe that we have succeeded in satisfying the intent and substance (and generally the letter) of these Requirements.

Our language is simple but does not lack power because of its simplicity. Given the many requirements in Ironman, it would have been easy to design a large and complex language. Had we done so, we would not have served DoD well. Instead, we have designed a language with a coherent set of simple and powerful features. Only such a language can be easily taught and learned, and only such a language can be a sound basis for the establishment of uniform programming standards across many projects, the procurement and control of many different translators, and the preservation of compatibility among installations. Only by using such a language can one hope to develop systems that are readable, maintainable, and compilable into highly efficient object code.

In the literature of programming language design, it is easy to find many novel proposals in such difficult areas as real-time control, exception handling, and encapsulation. Most of these have not been tested in production languages and are not well enough understood to make clear their complex interactions. The lesson of programming language design is that it is wise to be wary of novelty—that the apparent benefits of an untried feature are often insufficient to compensate for its hidden costs. We have confined our design to language features whose overheads and benefits are well understood (e.g., our real-time synchronization features) and can be fully resolved during program translation (e.g., our encapsulation and generic features).

This preference for the simple has not led us to sacrifice the capability of achieving optimal solutions of embedded system programming tasks. In order to help readers judge the validity of these claims, we attempt, in this document, to demonstrate in various ways the power, flexibility, and practicality of our design. We have considered a number of embedded systems programming needs, such as real-time control, fault recovery, and the use of nonstandard peripheral devices. The usefulness of our language in satisfying these needs has been reviewed by internal staff and consultants with diverse language and applications experience. These analyses convince us that our language can readily be used to solve—with simplicity, reliability, and efficiency—the problems of programming embedded systems. If we have erred in our choices, we have erred on the side of safety. Experience shows that it is very much easier to add

features to a language with a clean and simple design, than to add quality to a language with many unrelated features.

II STRUCTURE OF THIS DOCUMENT

This document consists of eight parts, the first of which is this Introduction. Part Two is a Language Report in which we specify, in precise detail, the syntactic and semantic rules of our language proposal. We were required to design our language by modifying Pascal to comply with Ironman, and have therefore written this Language Report with a structure parallel to that of the Pascal Report [22].

Part Three consists of a Tutorial and Programming Examples. The tutorial introduces the most elementary features of the language in simple combinations. The Programming Examples include elementary examples illustrating the tutorial, and advanced examples illustrating the use of advanced language features, and language features in combination. The examples are graded in difficulty and extensively annotated. They are intended to guide the reader gently into an appreciation of the style of our language and its use to solve both elementary and advanced programming problems.

Part Four is a Design Discussion. The Design Discussion is parallel in structure to the Language Report and discusses, for each significant language facility, the alternatives that we considered and the basis for our choice.

Part Five is an Analysis of Feasibility. Here we analyze the costs of implementing and documenting our language, its support environment, and its run-time systems.

Part Six is an Analysis of Compliance. It is parallel in structure to the Ironman Requirements for High-Order Languages [10]. We explain briefly how each Requirement has been met in our language, and refer the reader, where appropriate, to more extensive treatments in the other parts.

Part Seven is a Glossary. Here the reader will find brief definitions of the most frequent technical terminology used in this document.

Part Eight is a combined list of References.

III SUGGESTIONS TO THE READER

We have provided several starting points for our readers. Most readers should probably begin by reading the first two sections of the Language Report and the first section of the Compliance Analysis. They may then proceed in parallel through the easier examples and the rest of the Language Report, with the Programming Examples serving to illustrate the application of specific language rules in meaningful programming contexts. The reader who is familiar with programming will probably find that study of the Programming Examples is the quickest means of acquiring an overall perspective on our language.

Some readers will want to understand particular aspects of the design in depth. We recommend that these readers proceed through the Language Report, omitting, so far as possible, sections that are peripheral to their interests. For those sections with which they are directly concerned, these readers should turn as well to the corresponding section of the Design Discussion, to understand why we took a particular approach or why we included or excluded a specific feature.

Some readers will want to analyze our compliance with the Ironman Requirements or some subset of the Requirements of specific interest. They should use our Compliance Analysis as a starting point, reading the entry for each requirement of interest, and then turning to the Language Report, the Design Discussion, or the Programming Examples for further discussion.

Finally, some readers will want to know how to use our language to solve a specific programming problem. Unfortunately, it is not possible in a finite document to illustrate every, or even very many, different applications, or to fully explore the flexibility that our language offers. We urge the reader interested in solving a specific problem to look through the Programming Examples for an example related to that problem. If one is found, the sections of the Language Report and Design Discussion dealing with the features used in that example can then then be consulted to explain the possibilities available to the user of these features. We apologize to the reader who does not find a desired example and can only hope that he is able to synthesize his own example or, at least, be convinced, by the variety of examples that do appear, that his problem can be solved in our language.

PART TWO

LANGUAGE REPORT

CONTENTS

1 INTRODUCTION

This Language Report presents a language designed for the programming of embedded computer systems within the Department of Defense. The language is based on the Ironman Requirements [10]—an analysis of the language needs of the embedded system and language support domains—and is intended to satisfy these needs.

The language is simple but does not lack power because of its simplicity, for it has been designed as a coherent set of simple but powerful features, based on the language Pascal. Only such a language can be easily taught and learned, and only such a language can be a sound basis for the establishment of uniform programming standards across many projects, the procurement and control of many different translators, and the preservation of compatibility among installations. Only by using such a language can one hope to develop systems that are readable, maintainable, and compilable into highly efficient object code.

Although the language is quite general, it is explicitly designed for the programming of embedded computer systems and for the programming of support software such as compilers, support tools, and diagnostic aids. Generality is present in the language only to the extent necessary to meet the needs of these domains, as expressed in the Ironman Requirements.

The language design aims to assist in the development of reliable and maintainable programs. Features such as the explicit declaration of all program variables and the absence of defaults are intended to make programs easy to read, even though they may be made longer. By reducing the risk of misunderstanding, they improve program reliability and ease program maintenance. The *type* and *module* concepts included in the language can greatly assist in structuring programs and can ease understanding and maintenance of embedded systems. It is believed that this structuring will reduce the problems of programming large embedded systems.

The language has been designed so that the greater part of the language is completely machine independent. A small number of implementation dependent features have been included in the design, so as to meet the critical need of efficiency in certain embedded applications. Such features are described in a separate section of this Language Report. Almost all parts, even of these critical applications, can be written without direct use of these features—they can be contained in specific and limited program modules.

II SUMMARY OF THE LANGUAGE

The language has the general structure of Pascal and this Language Report has the same major section numbering as that used in the Pascal Report [22].

Section III explains the Extended BNF Grammar notation, which we use to describe the context-free syntax of the language, its *alphanumeric character set*, and its *comment convention*.

Section IV describes the major *lexical units* of the language: *identifiers*, *floating-point literals*, *fixed-point literals*, *integer literals*, *enumeration literals*, *character literals*, and *string literals*.

Section V is a placeholder, present only to retain the Pascal Report section numbering.

Section VI describes the *simple* and *composite types* and *representations* of the language. We draw a careful distinction between the notions of "data type" and "machinable representation of a data type," thereby accounting for the difference between machine arithmetic and mathematical arithmetic. For each data type, we give the relevant syntax rules, the valid representations of the type, the denotations of literal values of the type, and a list of the *standard operations* applicable to objects of the type.

Section VII describes the *simple declarations* of the language. *Constant*, *variable*, and *label* declarations are quite similar to these declarations in Pascal. *Type* declarations are syntactically like those of Pascal but have slightly different semantic significance, since they describe only the abstract properties of data. *Representation* declarations serve the purpose that the type declaration served in Pascal, permitting programmers to declare the type and representational attributes of data simultaneously. Section VII also gives the rules for *agreement of types and representations*, a series of examples of the simple and most common forms of declaration, and the rules for ordering simple declarations.

Section VIII describes *expressions*. In our expressions, operators have five priority levels. There are several operators at each level, and we adopt the Ironman rule for association within priority levels. In addition to these standard operators, we provide *free* nonassociative infix operators, which are given meaning by the user program. Section VIII also explains and illustrates the form of *designators*, the basic units with which expressions are composed.

Section IX describes *statements*. There are *simple statements* for *assignment*, *procedure call*, *process activation*, *assertion*, *GOTO*, *loop exit*, *procedure return*, *process termination*, and *propagation of exceptions*.

There are *structured statements* for *conditional execution* (an *IF-THEN-ELSE* conditional and a *case-selection* conditional) and for *repetitive execution* (a *FOR* statement, a *WHILE* statement, and a *LOOP* statement). There is also a *BEGIN-END* statement, styled after Algol 60, for lexical structuring of programs and nesting of program scopes. This statement includes a *simulated time* option.

Section X describes *composite declarations*—declarations that involve executable program text. The simplest of these are *function* and *procedure* declarations, which are jointly called *routine* declarations. The semantic rules for routines are presented, designed to to minimize *aliasing errors* and to allow *overloaded* definitions (that is, definitions of functions and procedures that implicitly discriminate on the basis of argument type). We have also provided a special data type, used only in the declaration of formal parameters of routines—this is the *length-unresolved* array parameter.

The *module* declaration describes our basic unit of encapsulation and abstraction. A module is a collection of declared program entities, program text to initialize these entities, and explicit control of what entities may be referred to within the module. Modules may be parameterized using *module templates* and *instances* of these templates.

The *process* declaration is similar in form to the routine declaration but allows the designation of program text which will be executed in parallel with its caller. We provide *monitors* and *signals* for coordinating these parallel processes (as well as lower-level synchronization facilities discussed in Section XIV)

Section XI describes *standard declarations*—those that the user may take as implicit. These include declarations of several kinds—standard functions, standard procedures, etc.

Section XII describes the high-level *input* and *output* facilities. Provision is made for *opening* and *closing* of *files*, and for *putting* data into and *getting* data from files, in an implementation-independent manner.

Section XIII describes the general appearance of programs as they are presented to the compiler, the notion of a *module synopsis* and its role in *separate compilation*, and certain implications of these facilities for the organization of large projects.

Section XIV describes the low-level and implementation-dependent features of the language. Features of this kind are available for control of storage, synchronization, *trap-handling*, *configuration control*, *machine code inserts*, and certain aspects of compilation.

Appendixes A, B, and C present, respectively, the reserved words, formal syntax, and standard errors.

III NOTATION, TERMINOLOGY, AND VOCABULARY

The grammar of our language is described with a variant of Wirth's Extended BNF notation (EBNF) [17]. This notation makes use of several metacharacters: left and right curly braces ({, }), left and right square brackets ([,]), left and right parentheses ((,)), a vertical bar (|), and an equal sign (=). In this Report, we print these metacharacters in a larger than usual type font to avoid confusion with their uses as characters of the language.

The grammar is composed of punctuation marks—which are either metacharacters or terminals; other terminal symbols—spelled in uppercase (possibly including the intraidentifier break "_"); nonterminal symbols—spelled in mixed case; and star-terminals—spelled in mixed case but ending with an asterisk. The first three categories have their common grammatical use; the star-terminals stand for classes of identifiers or literals, which are described in Section IV. The star-terminals used in the grammar are Id*, FloatingLiteral*, FixedLiteral*, IntegerLiteral*, CharacterLiteral*, and StringLiteral*.

Each group of grammar rules, corresponding to a nonterminal Lhs (left-hand side) with n alternatives, has the form

```
Lhs
    = Rhs-1
    ...
    | Rhs-n
```

Each alternative Rhs-i (right-hand side) is composed of nonterminals, terminals, star-terminals, and metacharacters. The metacharacter | denotes alternation; thus

a|b|...|c

denotes either "a", or "b", ..., or "c".

The metacharacters (and) are used in pairs to group the segment contained between them. Thus

(a|b) (c|d)

denotes either "a c", "a d", "b c", or "b d".

The metacharacters [and] enclose an optional segment. Thus,

RETURN [Expression]

denotes either "RETURN" or "RETURN Expression".

The metacharacters { and } enclose a segment that is to be repeated zero or more times. Thus,

{ Id* }

denotes either the empty sentence, or "Id*", or "Id* Id*", etc. Finally, these iteration brackets are sometimes used to enclose a segment whose last element is underlined. This indicates that the underlined element is a separator for the iterated segment but does not appear at its end. Thus,

TYPE { Id* = Type ; }

denotes either the sentence "TYPE", or the sentence "TYPE Id* = Type", or the sentence "TYPE Id* = Type ; Id* = Type", etc.

In the remainder of this Language Report, we will give syntax rules in the text as we describe the corresponding semantic rules. These syntax rules are numbered and the numbering in the text is the same as the numbering in Appendix B where all the rules are given. Often, not all of the syntactic options in a rule will be relevant in a particular semantic context. In this situation, we give the appropriate options and precede the rule by the character †. If not all the options for a nonterminal are given, the nonterminal is preceded by †.

The character set of our language consists of alphanumeric characters (including the underscore character) and certain punctuation marks.

Alphanumeric
= 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|
a|b|c|d|e|f|g|h|i|j|k|l|m|
n|o|p|q|r|s|t|u|v|w|x|y|z|
0|1|2|3|4|5|6|7|8|9|_

Note that uppercase and lowercase characters are distinct.

The reserved symbols are exactly the terminals (but not star-terminals) of the grammar; the reserved symbols are the symbols listed in Appendix A and all the punctuation marks.

A comment may occur wherever a space may occur—that is, between any pair of entire lexical units. (A *lexical unit* is a star-terminal or reserved symbol.) Comments do not affect the meaning of a program A comment is a sequence of characters beginning with "(*" and ending with "*)". A comment may not cross a source program line boundary; to write a comment longer than a single line, the programmer must close the comment at each line boundary and reopen it at the beginning of each subsequent line.

IV IDENTIFIERS, NUMBERS, CHARACTERS, AND STRINGS

The grammar uses seven star-terminals: Id*, FloatingLiteral*, FixedLiteral*, IntegerLiteral*, EnumerationLiteral*, CharacterLiteral*, and StringLiteral*. These stand, respectively, for the identifiers, literal floating point constants, literal fixed-point constants, literal integer constants, literal enumeration constants, literal character constants, and literal string constants.

A. *Identifiers*

An Id* is any sequence of alphanumeric characters (see Section III) not beginning with a digit (0 through 9). However, an Id* may not be a reserved word (see Appendix A) or an enumeration constant (see below).

The following are identifiers:

```
A
A1b2
Pi_Square
END_OF_MONTH
E_2_71828
StartingPay
```

B. *Numeric Literals*

A numeric literal is a FloatingLiteral*, a FixedLiteral*, or an IntegerLiteral*.

The class FloatingLiteral* is defined by the following rules:

FloatingLiteral*
= [+ | -] Mantissa [Exponent]

Mantissa
= DecimalDigit {DecimalDigit} . {DecimalDigit}

DecimalDigit
= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Exponent
= E [+ | -] DecimalDigit {DecimalDigit}

No spaces are permitted within a FloatingLiteral*. The following are members of the class FloatingLiteral*:

1.
-3.
38.26
1.0E-10

The class FixedLiteral* is defined by the following rules:

FixedLiteral*
= [Base] [+ | -] FixedMantissa FixedExponent

Base
= BASE2 | BASE8 | BASE10 | BASE16

FixedMantissa
= Digit {Digit} [. {Digit}]

FixedExponent
= P [[+ | -] DecimalDigit {DecimalDigit}]

If the Base is present, it must be followed by a space; no other spaces are permitted in a FixedLiteral*. The absence of a Base is equivalent to the presence of the Base BASE10. The Digits that comprise the FixedMantissa must be consistent with the Base: for BASE2, the Digits "0" and "1" are permitted; for BASE8, the Digits "0", ..., "7" are permitted; for BASE10 the Digits "0", ..., "9" are permitted; and for BASE16, the Digits "0", ..., "9" and "A", ..., "F" are permitted. The value of a FixedLiteral* is its optionally signed FixedMantissa (interpreted according to the specified Base), multiplied by a scale factor. The scale factor is a number—2 for BASE2, 8 for BASE8, etc.—raised to the power given by the FixedExponent. The FixedExponent is always interpreted as decimal. Some members of this class are:

3.9P
BASE16 -A67.9BP-2

The class IntegerLiteral* is defined as follows:

IntegerLiteral*
= [Base] [+ | -] Digit {Digit}

The rules for Base, the presence of spaces, and the consistency of Base and the Digits are the same as for FixedLiteral*. Some members of the class are:

-1345
BASE2 010101
BASE8 737746

C. *Enumeration, Character, and String Literals*

An EnumerationLiteral* is a sequence of alphanumeric characters satisfying the rules given above for Id* but introduced into a program by its use in an enumeration type (see Section VI).

The class CharacterLiteral* is composed of character values denoted by sequences of printing characters. (Characters such as carriage-return, apostrophe, double quote, or escape may not be used literally: equivalent mnemonics are used in their place.) Each sequence must begin and end with an apostrophe, and may include no other apostrophes or quotation marks. Embedded spaces are permitted and are significant. Every CharacterLiteral* is a member of an ALPHABET. ALPHABETS are described in Section VI. Some examples are:

'A'
'ACK'
'ENQ'
'NUL'
'CR'
'a'
'+'
'!'

A StringLiteral* is composed of a quotation mark, a sequence of CharacterLiteral*s, and a quotation mark. (Characters such as carriage-return, tab, or escape may not be used literally: equivalent mnemonics are used in their place.) If a constituent CharacterLiteral* is exactly one character long, excluding its apostrophes, then the apostrophes may be omitted. Some StringLiteral*s are:

```
"THIS IS A STRING"
"THIS IS A STRING ENDING IN A LINE FEED'LF'"
"'ACK'"
"'DQ'This is a doubly-quoted string'DQ'"
"'B''C'"
"BC"
```

Note that the last two examples denote the same string.

The opening and closing quotation marks of a StringLiteral" must appear on the same physical line of a program. However, if two or more StringLiteral"s are separated only by spaces or new-line characters, then the StringLiteral"s are implicitly catenated. This special rule is provided, and must be used, to write a StringLiteral" that does not fit on a single physical line of the source program: the StringLiteral" begins and ends with a quotation mark and each of its intermediate physical lines also begins and ends with a quotation mark. Thus,

```
"This is a single literal"
"even though it spans sever"
"al lines"
```

is one string literal.

V CONSTANT DEFINITIONS

This section is subsumed by the subsection on CONST declarations in Section VII and is included here only to retain in this Report the section numbering of the Pascal Report [22].

VI DATA TYPE DEFINTIONS AND REPRESENTATIONS

All values that can be computed by programs are members of disjoint sets of values. These disjoint sets are called types. That is, a *type* is a set of values, and each value is a member of exactly one type. The language is strongly typed: every expression in a program has a unique type. In the language, there are both primitive and nonprimitive types. The primitive types are denoted by identifiers; INTEGER and BOOLEAN are examples. The nonprimitive types are presented below. A *type definition* may be used by programmers to associate an identifier with a type. A type may be simple—for example, a scalar, INTEGER, FIXED-point, or FLOATING-point; it may also be composite—an array, record, set, or pointer type. Each type is described below.

It is useful and often necessary to specify a subset of the values of a type or to describe how the values are to be represented in some implementation. Such a description is called a *representation* of the type. A type may have many representations; a representation has a single type. For example, a representation may specify the subrange of the values in a scalar type, the limited precision of floating-point values, or the arrangement in storage of the fields of a record type. Those attributes with a high-level and implementation-independent meaning are described below; others are described in Section XIV. To further the goal of program portability, we have avoided explicit primitive representations. A *representation definition* may be used by programmers to associate an identifier with a particular representation of a type. Type and representation definitions are a kind of declaration and are therefore presented in Section VII; the present chapter is concerned with the nature and use of types and representations.

A simple example of the complementary roles of types and representations is the type INTEGER, which consists of the infinite set of mathematical integers, and a representation INT, which consists of some finite subrange of the integers available in a particular machine. Similarly, the type FLOATING consists of the infinite set of mathematical real numbers, and a particular representation SINGLEPRECISION can specify the precision and range of reals used in a computation.

Our language provides type definitions, representation definitions, and variable and routine declarations in which representations are used (see Section VII and Section X). The notation has been designed to simplify programming in the most common cases, where the programmer works primarily with representation definitions. Because type is implicit in representation, type definitions need be used only rarely—where it is necessary to specify two different representations of the same abstract type.

The next several subsections consider each type in turn, the representations appropriate to

the type, the designation of values of the type, and the operations specifically applicable to values of the type. The final subsections consider issues common to all types, including the meaning of scope entry and manifest constants.

An implementation on a machine without hardware floating-point may omit the type FLOATING and an implementation on a machine with accurate floating-point operations may use these operations to implement the fixed-point arithmetic.

The syntax relevant to representations and types is:

```
    Representation
61      = [Type] ConcreteType
62      | RepresentationId

    Type
63      = TypeId
64      | UNORDERED ( Ids )
65      | ORDERED ( Ids )
66      | ALPHABET ( {CharacterLiteral* ,} )
67      | ARRAY ( {Representation ,} OF Type )
68      | RECORD
            {(CONST|VAR) {Ids : Type ;}} [;]
            [SELECT Id* : Type
               {CASE {ManifestConstant ,} => {(CONST|VAR) {Ids : Type ;}} [;]}
             ENDSELECT [;]] ENDRECORD
69      | FILE ( Type )
70      | POINTER ( Representation )
71      | SET ( Representation )

    ConcreteType
†72     = ConcreteType1


    ConcreteType1
73      = TypeId [SimpleQualifier]
74      | UNORDERED ( Ids ) [SimpleQualifier]
75      | ORDERED ( Ids ) [SimpleQualifier]
76      | ALPHABET ( {CharacterLiteral* ,} ) [SimpleQualifier]
77      | ARRAY ( {Representation ,} OF Representation )
78      | RECORD
            {(CONST|VAR) {Names : Representation ;}} [;]
            [SELECT Name : Representation
               {CASE {ManifestConstant ,} =>
                  {(CONST|VAR) {Names : Representation ;}} [;]}
             ENDSELECT [;]] ENDRECORD
79      | FILE ( Representation )
80      | POINTER ( Representation )
81      | SET ( Representation )

    SimpleQualifier
82      = [ ScopeEntryConstant .. ScopeEntryConstant ]
83      | [ ScopeEntryConstant .. ScopeEntryConstant ] SCALE ManifestConstant
84      | [ ScopeEntryConstant .. ScopeEntryConstant ] PRECISION ManifestConstant
```

(Note that TypeId, as used in these rules, is either a standard type identifier—INTEGER,

FIXED, FLOATING, SIGNAL, or BOOLEAN—or an identifier introduced by a type definition. A RepresentationId is an identifier introduced by a representation definition. These forms of definition are described in Section VII.)

Summary of Types and Representations

Type	*Qualifier*	*Operations*	*Denotations*
INTEGER	[i1..i2]	+ - / * DIV MOD = ≠ < <= > >= SUCC PRED	IntegerLiteral*
FIXED	[f1..f2] SCALE s	+ - / * FIXED_DIV = ≠ < <= > >=	FixedLiteral*
FLOATING	[f1..f2] PRECISION i	+ - / * = ≠ < <= > >=	FloatingLiteral*
SIGNAL	(none)	SEND WAIT AWAITED	(none)
UNORDERED	[c1..c2]	= ≠ (BOOLEAN: AND OR XOR NOT)	EnumerationLiteral* (BOOLEAN: TRUE FALSE)
ORDERED	[c1..c2]	= ≠ < <= > >= SUCC PRED FIRST LAST	EnumerationLiteral*
ALPHABET	[c1..c2]	= ≠ < <= > >= SUCC PRED FIRST LAST	CharacterLiteral*
ARRAY	(none)	subscripting MOVE HIGH LOW	TypeId(e1,...,en)
RECORD	(none)	selection	TypeId(e1,...,en)
FILE	(none)	OPEN CLOSE GET PUT CHECK_EOF	(none)
POINTER	(none)	= ≠ ↑	NIL
SET	(none)	+ - * XOR IN	TypeId(e1,...,en)

A. *Simple Types*

The simple types are those types not defined in terms of other types. The simple types are the *basic types* and the *scalar types*. The basic types are primitive and not definable in the language. In principle, the scalar types are all definable within the language; in practice, one scalar type, BOOLEAN, is a language primitive. The representational information appropriate only to simple types is called *simple qualification*.

1. *Basic Types*

The basic types are INTEGER, FIXED, FLOATING, and SIGNAL.

a. *Integers*

The identifier associated with this type is INTEGER. The values are the mathematical integers. The operations are +, -, /, *, DIV, MOD, =, ≠, <, <=, >, and >=; their semantics are described in Section VIII. The operations SUCC and PRED also apply to integers and are described in Section XI. The values are denoted by the elements of the class IntegerLiteral*, which was described in Section IV.

The only representation applicable to INTEGER is *range qualification.* An integer range qualifier specifies the range of integer values to be represented. For example,

```
INTEGER [5..9]
```

represents the range of integer values from 5 through 9. An explicit representation is always required. Projects and programmers will typically declare identifiers for commonly used representations (see Section VII).

The syntax for the INTEGER type is given by the rule

```
↑    Type
63       = TypeId
```

where TypeId is INTEGER. The representations of integers are given by the rules

```
     Representation
 61      = [Type] ConcreteType
 62      | RepresentationId

     ConcreteType
†72      = ConcreteType1

†    ConcreteType1
 73      = TypeId [SimpleQualifier]

†    SimpleQualifier
 82      = [ ScopeEntryConstant .. ScopeEntryConstant ]
```

b. *Fixed-Point Numbers*

The identifier associated with this type is FIXED. The values are the mathematical rationals. The operations are +, -, /, *, =, ≠, <, <=, >, and >=; their semantics are described in Section VIII. The values are denoted by the elements of the class FixedLiteral*, which was described in Section IV.

The only representation applicable to FIXED is scale and range qualification—both are required. A FIXED range qualifier specifies the range of rational values to be represented. A FIXED scale qualifier specifies the minimal difference between two representable values. For example,

FIXED [0P..10P2] SCALE 0.5P

represents the rational values 0.0P, 0.5P, 1.0P, 1.5P, ..., 999.5P, 1000.0P.

Implementations will almost always restrict the allowable scales. For example, implementations on binary machines may only permit powers of two.

The syntax for the FIXED type is given by the rule

```
†    Type
 63      = TypeId
```

where TypeId is FIXED. The representations of fixed-point numbers are given by the rules

```
     Representation
61       = [Type] ConcreteType
62       | RepresentationId

     ConcreteType
†72      = ConcreteType1

†    ConcreteType1
 73      = TypeId [SimpleQualifier]

†    SimpleQualifier
 83      = [ ScopeEntryConstant .. ScopeEntryConstant ] SCALE ManifestConstant
```

c. *Floating-Point Numbers*

The identifier associated with this type is FLOATING. The values are the mathematical reals. The operations are +, -, /, *, =, ≠, <, <=, >, and >=; their semantics are described in Section VIII. The values are denoted by the elements of the class FloatingLiteral*, which was described in Section IV.

The only representation applicable to FLOATING is precision and range qualification—both are required. A FLOATING range qualifier specifies the range of real values to be represented. A FLOATING precision qualifier specifies the minimal number of significant decimal digits to be represented. The precision must be an integer. For example,

FLOATING [-1.0E3..1.0E3] PRECISION 7

represents the real values -1000.000, -999.9999, -999.9998, ..., 0.000000, ..., 999.9999, 1000.000.

The syntax for the FLOATING type is given by the rule

```
†    Type
 63      = TypeId
```

where TypeId is FLOATING. The representations of floating-point numbers are given by the rules

```
     Representation
61       = [Type] ConcreteType
62       | RepresentationId

     ConcreteType
†72      = ConcreteType1

†    ConcreteType1
 73      = TypeId [SimpleQualifier]

†    SimpleQualifier
 84      = [ ScopeEntryConstant .. ScopeEntryConstant ] PRECISION ManifestConstant
```

d. *Signals*

The identifier associated with this type is SIGNAL. The values are used for coordinating the actions of parallel processes. Only the built-in procedures SEND, WAIT, and AWAITED may be applied to signals. (In particular, assignment, equality, and inequality are not defined for signals.) There are no literal signals. No explicit representations are permitted for the type SIGNAL.

The syntax for the SIGNAL type is given by the rule

```
†    Type
 63      = TypeId
```

where TypeId is SIGNAL. Since there are no explicit representations of signals, the representation is given by the rule

```
     Representation
61       = [Type] ConcreteType
62       | RepresentationId
```

2. *Scalar Types*

A scalar type is an enumeration of values. There are three kinds of scalar type: unordered enumerations, ordered enumerations, and alphabets.

a. *Unordered Enumerations*

An unordered enumeration is a list of identifiers preceded by the reserved word UNORDERED. The values of the enumeration are denoted by these identifiers. Some examples are:

UNORDERED(Red, Orange, Yellow, Green, Blue)

UNORDERED(Lemon, Blueberry, Strawberry)

There is one standard unordered enumeration associated with the identifier BOOLEAN and defined by

UNORDERED(FALSE, TRUE)

The operations = and ≠ apply to all enumeration values. The standard operations AND, OR, XOR, and NOT are applicable to BOOLEAN. The semantics of these operations is described in Section VIII.

The only representation qualification applicable to unordered enumerations is range qualification. For example, if the identifier Colors is associated with the first example given above, then

Colors [Orange..Blue]

represents the range of Colors other than Red. The range qualification is stated in terms of the lexical ordering of the identifiers in the type definition. Unordered enumerations have a standard representation that is their full range of values (thus permitting the type name to be used as a representation, without further qualification).

The syntax for an unordered enumeration type is given by the rule

```
†    Type
64        =  UNORDERED ( Ids )
```

The representations of unordered enumerations are given by the rules

```
     Representation
 61      = [Type] ConcreteType
 62      | RepresentationId

     ConcreteType
†72      = ConcreteType1

†    ConcreteType1
 74      = UNORDERED ( Ids ) [SimpleQualifier]

†    SimpleQualifier
 82      = [ ScopeEntryConstant .. ScopeEntryConstant ]
```

b. *Ordered Enumerations*

An ordered enumeration is a list of identifiers preceded by the reserved word ORDERED. The values of the enumeration are denoted by these identifiers. An example is:

ORDERED(Monday, Tuesday, Wednesday, Thursday, Friday)

The operations =, ≠, <, <=, >, and >= apply to ordered enumerations and are described in Section VIII. The operations SUCC, PRED, FIRST, and LAST also apply to ordered enumerations and are described in Section XI.

The only representation qualification applicable to ordered enumerations is range qualification. The significance of range qualification of ordered enumerations is the same as that of unordered enumerations. Ordered enumerations have a standard representation that is the full range of values.

The syntax for an ordered enumeration type is given by the rule

```
†    Type
 65      =  ORDERED ( Ids )
```

The representations of ordered enumerations are given by the rules

```
     Representation
 61      = [Type] ConcreteType
 62      | RepresentationId

     ConcreteType
†72      = ConcreteType1

†    ConcreteType1
 75      =ORDERED ( Ids ) [SimpleQualifier]

†    SimpleQualifier
 82      = [ ScopeEntryConstant .. ScopeEntryConstant ]
```

c. *Alphabets*

Alphabets are ordered enumerations of character literals. The operations and representations for alphabets are the same as those for ordered enumerations. The values are denoted by members of the class CharacterLiteral*, which was defined in Section IV.

An example is:

ALPHABET ('a', 'b', '1', 'cr', 'eot')

Each implementation will provide one or more standard alphabets—for example, an ASCII_ALPHABET enumerating the ASCII characters in proper order, or an EBCDIC_ALPHABET enumerating the EBCDIC characters in proper order.

The syntax for an alphabet type is given by the rule

```
†    Type
 66      =   ALPHABET ( {CharacterLiteral* ,} )
```

Range qualification is the only representational qualification of alphabets. The syntax rules are:

```
     Representation
61       = [Type] ConcreteType
62       | RepresentationId

     ConcreteType
†72      = ConcreteType1

†    ConcreteType1
 76      = ALPHABET ( {CharacterLiteral* ,} ) [SimpleQualifier]

†    SimpleQualifier
 82      = [ ScopeEntryConstant .. ScopeEntryConstant ]
```

3. *Disambiguation*

The rules stated so far for denoting simple constants are not always sufficient to satisfy the rule that two types must denote disjoint sets of values. For example, let Colors and Flavors be, respectively, the types

UNORDERED(Red, Orange, Yellow, Green, Blue)

UNORDERED(Orange, Lemon, Lime)

Then it is impossible to tell whether the constant Orange denotes a color or a flavor. To resolve this kind of ambiguity, we adopt the rule that whenever two literal values of two distinct enumerations, within some scope, would otherwise have the same denotation, the values must be qualified by the identifier associated with the type or representation. For example, in a scope where Colors and Flavors are defined as above, Orange is not a value—it must be either Flavors Orange or Colors Orange.

Unqualified numeric literals always have one of the language-defined types INTEGER, FLOATING, or FIXED. In a scope where Feet and Inches have been defined as different types of integers, numeric literals of these types must be qualified but integer constants need not be qualified.

When a string literal occurs in a scope in which there is more than one ALPHABET, we adopt a stronger rule and require that the literal be qualified by the identifier associated with the alphabet type (even if the particular value is unambiguous or, indeed, even if the alphabets are disjoint). Thus, in a scope containing Alphabet1 defined as

ALPHABET ('a', 'A', 'cr', 'b')

and Alphabet2 defined as

ALPHABET ('b', 'lf', 'A', 'a')

all string literals must be qualified—e.g.,

Alphabet1 "a'cr'b"
Alphabet2 "bA'lf"

Modules with controlled importation (see Section X) can often be used to ensure that scopes include only one alphabet, thus eliminating the need for this notation.

B. *Structured Types*

The structured types are those types defined by composing other types. They are arrays, records, files, pointers, and sets. Each structured type, like each simple type, can have several representations.

1. *Arrays*

An array has a fixed number of *components*, all of the same type and representation, which are *selected* by scalar or integer *indices*. The type of an array is determined by the type of its components and the representation of its indices. The number of indices is called the *dimensionality* of the array. (Recall that the only representational qualifier appropriate to scalars and integers is range qualification; this range qualification specifies the bounds of an array and is therefore properly included in its type.) The representation of an array is determined by the representation of its components and certain additional representation information qualifying the array as a whole.

Some examples of array representations are:

ARRAY (INTEGER [1..5] OF FLOATING [0.0..1.0] PRECISION 10)

ARRAY (INTEGER [1..10], INTEGER [1..20] OF INTEGER [0..99])

ARRAY (Colors OF BOOLEAN)

We describe the denotation of array values first by example. Suppose the identifier Fl_Array is associated with the first of these examples. Then Fl_Array(0.0) denotes an array value with all components equal to 0.0. Fl_Array(0.0,0.5,0.6,0.7,0.8) denotes an array value with five components equal, respectively, to the enumerated values.

In general, the array value is denoted by the identifier associated with the array type or representation, followed by a list of component values. The identifier determines the type of the array value. This list may either have length one or length exactly equal to the number of components. In the first case, all components are set to the specified value. In the second case, the components are set to the corresponding values (interpreted in row-major order). No other cases are permitted.

The operations applicable to arrays are subscripting, MOVE, HIGH, and LOW. Subscripting is described in Section VIII and MOVE, HIGH, and LOW are described in Section XI.

The representation of the component type of an array may be any representation appropriate for the component type. The additional representational qualifications applicable to the array as a whole are implementation-dependent and are described in Section XIV. These

representations taken together determine the representation of the array type.

The syntax for an array type is given by the rule

```
†    Type
 67      =   ARRAY ( {Representation ,} OF Type )
```

where the representation is that of a scalar or integer type. The representation of an array is given by the rules

```
     Representation
 61      = [Type] ConcreteType
 62      | RepresentationId

     ConcreteType
†72      = ConcreteType1

†    ConcreteType1
 77      =   ARRAY ( {Representation ,} OF Representation )
```

Any array whose component type is an alphabet is called a character string; its value may be described by a StringLiteral* as described in Section IV.

2. *Records*

A record is a structure consisting of components that may have different types. Each component has a *modifiability attribute*—CONST or VAR. A CONST component is modifiable only if the entire entire record value is changed. A VAR component may be individually modified. (If a CONST component X of a record R is itself a structured value, then modification of a component of X is construed as a modification of X and is consequently forbidden.) Each component also has a *selector*, which is an identifier. The type of a record is determined by the types, modifiability attributes, and selectors of its components. The representation of a record is determined by its type, the representations of its components, and certain other representational information qualifying the record as a whole.

A record type may have several alternative structures, called *variants*; such a record type is called a *variant record type* and must include a special component, the *case-selection component*. This component indicates which variant is assumed by a value of the record type. The case selection component is implicitly CONST: it is determined when a record value is denoted and is not individually modifiable. The case selection component must be of an integer or scalar type.

Some examples of record types are:

```
RECORD CONST day: INTEGER;
       CONST month: INTEGER;
       CONST year: INTEGER
ENDRECORD

RECORD CONST name, firstname: alfa;
       VAR age: INTEGER;
       VAR married: BOOLEAN
ENDRECORD
```

An example of a variant record type is:

```
RECORD VAR x, y: FLOATING;
       VAR area: FLOATING;
       SELECT  s: shape
               CASE RightTriangle => VAR base, height: FLOATING;
               CASE Circle => VAR diameter: FLOATING;
       ENDSELECT;
ENDRECORD
```

Examples of record representations may be obtained by replacing FLOATING with a representation of FLOATING, and INTEGER, with a representation of INTEGER in the above.

We introduce denotations of record values by example. Suppose the identifiers Date, Person, and Figure are associated, respectively, with these three examples. Then

```
Date(15, 2, 1978),

Person(alfa1, alfa2, 37, FALSE),

Figure(x1, y1, 60.0, RightTriangle, 5.0, 24.0), and

Figure(x2, y2, .25*pi*25.0*25.0, Circle, 25.0)
```

each denotes a value of the corresponding record type. An appropriate value must be provided for each component. If the record type has variants, then the components present in the variant case desired are listed, including the case-selection value.

The only specific record operation is component selection, which is described in Section VIII.

The representation of a component type of a record may be any representation appropriate for the component type. The additional representational qualifications applicable to the record as a whole are implementation-dependent and are described in Section XIV. These representations, taken together, determine the representation of the record type.

The syntax for a record type is given by the rule

```
†    Type
 68      =RECORD
             {(CONST|VAR) {Ids : Type ;}} [;]
             [SELECT Id* : Type
                {CASE {ManifestConstant ,} => {(CONST|VAR) {Ids : Type ;}} [;]}
              ENDSELECT [;]] ENDRECORD
```

The representation of a record is given by the rules

```
     Representation
 61      = [Type] ConcreteType
 62      | RepresentationId

     ConcreteType
†72      = ConcreteType1

†    ConcreteType1
 78      =RECORD
             {(CONST|VAR) {Names : Representation ;}} [;]
             [SELECT Name : Representation
                {CASE {ManifestConstant ,} =>
                   {(CONST|VAR) {Names : Representation ;}} [;]}
              ENDSELECT [;]] ENDRECORD
```

3. *File Types*

A file is a sequence of components of the same type. The length of the sequence is not fixed by the file type.

An example of a file type is

FILE(ASCII_ALPHABET)

There are no denotations for file values. The operations on files—OPEN, CLOSE, GET, PUT, and CHECK_EOF—are described in Section XII. (In particular, equality, inequality, and assignment are not defined for files.) A representation of a file type consists of the representation of the component type.

The syntax for a file type is given by the rule

```
    Type
69      =  FILE ( Type )
```

The representations of files are given by the rules

```
    Representation
61      =  [Type] ConcreteType
62      |  RepresentationId

    ConcreteType
†72     =  ConcreteType1

†   ConcreteType1
 79     =  FILE ( Representation )
```

4. *Pointer Types*

A pointer is, in essence, a reference to a dynamic and anonymous variable. Normal variables are named and have a lexical scope—the execution of the block to which they are local (e.g., the procedure body, BEGIN-END statement, or process body). Such variables are called *local variables* of the scope. (In PL/1 they would be called *automatic variables*.) In contrast, variables may also be generated dynamically. In this case, they are not named directly, but only through a pointer.

Pointers are strongly typed—that is, a value p of a pointer type PT—can refer only to dynamic variables of a specific type and representation. A pointer type thus consists of an unbounded set of values pointing to elements of the same type and representation. The type of such a PT is determined by the type and representation of the variables to which reference can be made. The representation of a pointer type is determined by the representation of the variables to which reference can be made.

An example of a pointer type is:

POINTER(BOOLEAN)

The only literal pointers are the *NIL* pointers—these are special values of a pointer type which refers to nothing at all. To preserve the strong typing rules, we require that the identifier NIL be disambiguated by a pointer-type identifier. For example, if the identifier B_POINTER is associated with the type POINTER(BOOLEAN), then B_POINTER NIL is a NIL value of this type.

The operations on pointers are $=$, $\neq$, and $\uparrow$. These operations are described in Section VIII. The operation NEW is used to generate new anonymous variables and is described in Section XI.

The syntax for the pointer types is given by the rule

```
†    Type
 70      =  POINTER ( Representation )
```

The syntax for the representations of pointer types is given by the rules

```
     Representation
 61      =  [Type] ConcreteType
 62      |  RepresentationId

     ConcreteType
†72      =  ConcreteType1

†    ConcreteType1
 80      =  POINTER ( Representation )
```

5. *Set types*

A set type defines a set of values of a base type. The base type must be an integer type or a scalar type. A set type, in the same way as an array type, is determined by the type and representation of its base type. A representation of a set type is determined by the representation of its base type and certain other information qualifying the set type as a whole. This latter category is implementation-dependent and is described in Section XIV.

Some examples of set types are:

SET(Colors)

SET(INTEGER [1..10])

The first includes as values all the subsets of the Colors, i.e., the empty set, Orange, Orange and Red, etc; recall that Colors is an enumeration type and therefore may be used as a representation. The second consists of all subsets of the set of integers 1, ..., 10 and must include its range qualifier since the type INTEGER requires representation. For example,

SET(INTEGER [1..10])

We introduce denotations of set values by example. Suppose the identifier INT_SET is associated with the example just given. Then INT_SET(3,5) denotes the subset of the integers with the two specified values. In general, any list of values of the base type may be provided. As another example, INT_SET() denotes the empty set of integers.

The operations on sets are + (set union), - (set difference and complement), * (set intersection), XOR (symmetric difference), and IN (set membership). These are described in Section VIII.

The syntax for set types is given by the rule

```
†    Type
 71       =   SET ( Representation )
```

The syntax for set representations is given by

```
     Representation
 61      = [Type] ConcreteType
 62      | RepresentationId

     ConcreteType
†72      = ConcreteType1

†    ConcreteType1
 81      =   SET ( Representation )
```

C. *The Occurrence of Constants in Types*

In the syntax rules given above, use has been made of two important classes of values: ScopeEntryConstant and ManifestConstant. Syntactically, both may be arbitrary expressions, as described in Section VIII. However, they are semantically restricted as follows.

A ManifestConstant is an expression composed entirely of literals, standard functions and operators, and other manifest constant expressions. Only manifest constant expressions may be used to enumerate the cases of a variant record and to specify the precision of a floating-point number or the scale of a fixed-point number. Also, in denoting a value of a variant record type, only a manifest constant may be used as the case selection component.

A ScopeEntryConstant is an expression whose value is determined no later than that point in program execution at which control enters the smallest block enclosing a use of the expression; the value may be determined earlier, often even as early as program translation. Only ScopeEntryConstants are used in range qualifiers. Note that a manifest constant is a scope entry constant.

VII SIMPLE DECLARATIONS

This section describes the constant, variable, label, type, and representation declarations. The remaining forms—function, procedure, module, template, instance, and process declarations—are described in Section X. Each declaration introduces a name—for a constant, a type, a variable, etc.—and establishes the *scope* in which this name has a single meaning (see Section J for details).

A. *Declaring Constants*

The form of the CONST declaration is

```
↑    Declaration
7        = CONST {Names : Representation = ScopeEntryConstant ;}
```

An example is:

```
CONST Color1, Color2: Colors = Orange;
```

The effect of this declaration is to define one or more identifiers as equivalent to a constant of the specified type. The identifier may then be used in expressions, function and procedure calls, and so forth; however, assignment to it is forbidden. As illustrated in this example, the ":" must be followed by a representation. Note that the value given to a constant in its declaration is computed with an expression that must be a ScopeEntryConstant (see Section VI).

B. *Declaring Variables*

The VAR declaration is used to declare a variable, i.e., an entity whose value may change during its lifetime. Its syntax is:

```
†    Declaration
 8       =VAR {Names : Representation [:= Expression] ;}
```

An example is:

```
VAR  C1, C2: Rectangular;
     B1, B2: BOOLEAN;
     Color: Colors
```

This declares two variables with the representation Rectangular, two BOOLEAN variables, and a Colors variable.

Variables may be initialized at the time of their declaration. For example, we might write

```
VAR  B1, B2: BOOLEAN := TRUE;
     R1: Rectangular := UnitCircle(Pi/4.0);
     R2: Rectangular := UnitCircle(Pi/5.0)
```

This declares two BOOLEAN and two Rectangular variables. The BOOLEAN variables are both initialized to TRUE. The Rectangular variables are initialized to two different values, computed by calling a function declared in an enclosing scope. Several points should be noted. First, the initializing expressions should be interpreted as occurring within the current scope and thus may cite local names. Initializations are done in the order of the declarations in which they are contained. Second, initialization is syntactically optional. If a variable's initialization is omitted, then it should be otherwise set prior to an attempt to access its value. (By contrast, we require the ":=" clause in CONST declarations since there is no subsequent way to set the value of a constant.)

Assignment is the most usual way in which the value of a variable is changed. However, there are certain types for which assignment is not defined—*nonassignable types*: these are files, signals, structured types involving files or signals, and certain *opaquely exported* user-defined types (see Section X). Assignment is not a semantically sensible notion for entities of these types; such entities are initialized and modified instead by the input and output procedures or the SEND, WAIT, and AWAITED procedures (see Section X and Section XII). An entity of a nonassignable type must be declared as a variable.

The definition of a type implicitly provides the definitions of the assignment procedure and the equality and inequality functions to work appropriately for the new type (unless this new type involves components, such as signals or files, for which assignment, equality, and inequality are not defined). However, the user may explicitly *replace* any such implicit definitions (see Section X).

C. *Declaring Labels*

The form of a LABEL declaration is

```
†    Declaration
 9       = LABEL Ids
```

An example of a LABEL declaration is

LABEL End_of_Scope, Error1

Only the cited identifiers, and no others, may be used as statement labels within the scope of the LABEL declaration.

D. *Declaring Types*

In Section VI, we described the data types that can be defined in the language. The TYPE declaration serves to associate identifiers with these definitions. The syntax is

```
†    Declaration
10       =TYPE {TypeId = Type ;}
```

An example is

```
TYPE PolarType = RECORD VAR Rho, Theta: FLOATING ENDRECORD;
     RectangularType = RECORD VAR x, y: FLOATING ENDRECORD
```

Within the scope of this declaration, the identifier is an abbreviation for the type definition. However, the rule concerning type equality, presented below, is very sensitive to the use of identifiers introduced by means of TYPE declarations.

E. *Declaring Representations*

In Section VI, we described the representations that can be defined in the language. The representation declaration associates identifiers with these definitions. The syntax is

```
†    Declaration
11       =REPRESENTATION {RepresentationId = [Type] ConcreteType ;}
```

An example is

```
REPRESENTATION
   Polar = RECORD
             VAR Rho, Theta: FLOATING [0.0..1.0] PRECISION 4
           ENDRECORD;
   Rectangular = RECORD
                   VAR x, y: FLOATING [0.0..1.0] PRECISION 4
                 ENDRECORD
```

Within the scope of this declaration, the identifier is an abbreviation for the representation.

F. *Multiple Declarations*

The syntax for the declarations described above is

```
†    Declaration
7       = CONST {Names : Representation = ScopeEntryConstant ;}
8       | VAR {Names : Representation [:= Expression] ;}
9       | LABEL Ids
10      | TYPE {TypeId = Type ;}
11      | REPRESENTATION {RepresentationId = [Type] ConcreteType ;}
```

As we remarked in Section VI, TypeId and RepresentationId are distinguished by the obvious semantic context: TypeIds are identifiers that derive from type declarations and RepresentationIds are identifiers that derive from representation declarations. Declarations occur only within blocks, whose syntax is given by

```
     Block
6       = {Declaration ;} [;] [ExceptionHandler [;]] {[Id* :] Statement ;} [;]
```

It follows that several formats are possible for a sequence of declarations. For example, the sequence

```
TYPE  R = SomeType;
      S = SomeOtherType
```

is permitted and is equivalent to

```
TYPE R = SomeType;
TYPE S = SomeOtherType
```

In the case of CONST and VAR declarations, an additional freedom is the possibility of declaring several identifiers together, as in

```
CONST x, y, z: BOOLEAN = TRUE;
VAR   a, b, c: INTEGER [1..10] := 5
```

In this example, x, y, and z have the same type and representation and a, b, and c have the same type and representation. This is a special case of a general rule, which we now explain.

G. *Agreement of Types and Representations*

There are two kinds of contexts in which it is significant whether two entities have the same type or representation—in assignment, described in Section IX, and in parameter binding, described in Section X. In one case of parameter binding—VAR parameter binding—the representations of the entities involved must be identical, and this requirement is enforced during program translation. In the other cases of parameter binding—CONST and OUT parameter binding—and in assignment, translators need only check type agreement. In these cases, if representations differ, run-time code may be required to perform further computations—e.g., the check that a numeric value is in the allowed range for a numeric representation or the conversion from a packed to an unpacked representation (see Section XIV). In the special case of range checks, the compiler may sometimes be able to verify directly that no run-time range check is required; in any case, it is possible by compiler directive to suppress run-time range checking.

In the remainder of this subsection, we state general and precise rules for drawing these distinctions. Most programmers, however, will find that it is sufficient to use only the forms of declaration listed in Section H and may therefore defer learning these rules.

We now define when two declared objects have the same type. First, if X and Y are declared together—that is, in the form

```
VAR X, Y: <Representation>
```

then X and Y have the same type (and, indeed, the same representation). Note here that <Representation> may be any form derived from the syntax for Representation:

```
    Representation
61      = [Type] ConcreteType
62      | RepresentationId
```

Next, suppose that X and Y are declared separately—that is, in the form

```
VAR  X: <Representation1>;
     Y: <Representation2>
```

There are two cases.

Case 1. If <Representation1> and <Representation2> are the same identifier, then X and Y have the same type and the same representation.

Case 2. Otherwise, if <Representation1> is an identifier, it must be an identifier introduced by a representation declaration. Replace it by the right-hand side of that representation declaration. Similarly, if <Representation2> is an identifier, it must be an identifier introduced by a representation declaration. Replace it by the

right-hand side of its representation declaration. Now consider the expanded forms. If they begin with the same type identifier, then they have the same type. (Recall that reserved words such as RECORD and ORDERED are not identifiers; therefore, if two expanded forms begin with a reserved word, then the corresponding types are different.) They have the same representation if and only if the types are the same and the expanded forms have lexically identical ConcreteTypes. In any other case, X and Y have different types (and, of course, different representations).

For example, consider the following set of declarations:

```
TYPE
  Foot_Type = INTEGER;
  Inch_Type = INTEGER;
  Polar_Type = RECORD VAR A, B: FLOATING ENDRECORD;
  Rectangular_Type = RECORD VAR A, B: FLOATING ENDRECORD;
REPRESENTATION
  Feet = Foot_Type [1..5280];
  Inches = Inch_Type [1..12];
  Polar1 =
    RECORD
      VAR A, B: FLOATING [0.0..1.0] PRECISION 4
    ENDRECORD;
  Polar2 =
    Polar_Type
    RECORD
      VAR A, B: FLOATING [0.0..1.0] PRECISION 4
    ENDRECORD;
  Polar3 =
    Polar_Type
    RECORD
      VAR A, B: FLOATING [0.0..1.0] PRECISION 6
    ENDRECORD;
VAR
  A, B: Feet;
  C, D: INTEGER [1..5280];
  E: Foot_Type [1..100];
  F: Inches;
  G: Inches;
  H: Polar1;
  I: Polar2;
  J: Polar3;
  K: Polar_Type
     RECORD
       VAR A, B: FLOATING [0.0..1.0] PRECISION 6
     ENDRECORD;
  L: RECORD
       VAR A, B: FLOATING [0.0..1.0] PRECISION 6
     ENDRECORD;
  M: Rectangular_Type
     RECORD
       VAR A, B: FLOATING [0.0..1.0] PRECISION 6
     ENDRECORD;
```

A and B have the same type and representation. C and D have the same type (INTEGER) and representation, but are different from A and B (which have type FOOT_TYPE). E is the same type as A and B—that is, Foot_Type—but E has a different representation. F and G are the same type and representation. H cites a representation Polar1 whose type is omitted and is therefore different from any separately declared entity. I, J, and K have the same type—Polar_Type; J and K have the same representation and I has a different representation. L and M are different.

Representations are used in CONST, VAR, REPRESENTATION, and formal parameter declarations (see Section X) to specify both the type of an entity and some further restrictions

on the values that it may assume. It is important to understand that these restrictions are not type distinctions: the potential value sets of the entities may have values in common, which is never the case for entities of different types. For example, A and E may both assume the value 50. The attempt to assign 101 to E is an error—not a type error, but a range error. (Some implementations will allow the suppression of range error detection.)

H. *The Common Cases*

Usually, only a single representation of a type is needed. In this case, the TYPE declaration can be avoided. Instead, a single representation definition is used to associate an identifier with an abstract type and one of its representations, and that identifier is used for all declarations of entities of this type (and its single representation). Thus,

```
REPRESENTATION R_Integer = INTEGER [1..10];
VAR x, y, z: R_Integer
```

is the most common form. In this situation, it is critical to note that a variable w declared by

```
VAR w: INTEGER [1..20]
```

has the same type as x, y, and z because all declarations derive from the same type identifier INTEGER. Similarly, with structured types, one should generally write a single representation R and refer to it in all subsequent declarations. For example, consider the declaration

```
REPRESENTATION R = RECORD X: INTEGER [1..20] ENDRECORD
```

This is based on the alternative of Representation where the Type is omitted because it is deducible from the ConcreteType. It is exactly equivalent to

```
REPRESENTATION R = RECORD X: INTEGER ENDRECORD
                   RECORD X: INTEGER [1..20] ENDRECORD
```

where the type is written explicitly. These forms, because they do not give the abstract type explicitly by means of a type identifier, yield a single representation of an "anonymous" abstract type. (These forms are rarely used.)

I. *Order of Declarations*

In general, the programmer must order declarations, of the kinds presented above, so that identifiers are declared before they are used. The single exception to this rule is that a pointer type or pointer representation may be used before it is declared. In this case, it must be subsequently declared in the same sequence of declarations. If an identifier is used in a declaration D1, and an entity with the same name is declared in a declaration D2 in the same sequence of declarations, then the use in D1 is construed as a use D2.

J. *Scopes*

We will need, in what follows, to refer to the *scope* of a declaration. This is that part of a program in which the declared entity is defined with the single meaning given by the declaration. The scope of a declaration is simply the block at the head of which it occurs (but excluding any inner scopes with a different declaration of the same name). We must also introduce the notion of the *enclosing scope* of a scope of declaration. The *lexically enclosing scope* of a block B is that block in which B is lexically nested. A *dynamically enclosing scope* of B is

- The lexically enclosing scope of B if B is the body of a BEGIN-END statement, and otherwise
- The block containing the invocation of the routine or process whose body is B.

(Blocks and BEGIN-END statements are explained in Section IX and routines and processes in Section X.)

VIII EXPRESSIONS

Expressions are rules for computing values by the application of operators to operands. The rules of expression composition specify levels of operator precedence. The operator NOT has the highest precedence, followed by the multiplying operators, the adding operators, the prefix minus operator, and the relational operators.

The syntax of expressions is given by the rules:

```
    Expression
34      = SimpleExpression [RelationalOperator SimpleExpression]

    SimpleExpression
35      = [-] Term
36      | Term {AssociativeAddingOperator Term}
37      | Term NonAssociativeAddingOperator Term

    Term
38      = Factor {AssociativeMultiplyingOperator Factor}
39      | Factor NonassociativeMultiplyingOperator Factor

    Factor
40      = {NOT} (( Expression ) | Designator | [Id°] Literal)

    Designator
41      = Id°
42      | Id° ( {Expression ,} )
43      | Designator (. Id° | [ {Expression ,} ] | ↑)

    RelationalOperator
47      = IN | ≠ | = | < | <= | > | >= | FreeRelationalOperator

    FreeRelationalOperator
48      = ≠≠ | ≠≠≠ | == | === | << | <<= | <<< | <<<= |
              >> | >>= | >>> | >>>=

    AssociativeMultiplyingOperator
49      = * | AND

    NonassociativeMultiplyingOperator
50      = / | DIV | MOD | FreeMultiplyingOperator

    FreeMultiplyingOperator
51      = ** | *** | // | ///

    AssociativeAddingOperator
52      = + | OR

    NonassociativeAddingOperator
53      = - | XOR | FreeAddingOperator

    FreeAddingOperator
54      = -- | --- | ++ | +++
```

A. *Operators*

There are up to three kinds of operators in each class: standard associative operators, standard nonassociative operators, and free operators (which are syntactically nonassociative). The free operators are not given any standard meaning by the language; they can be defined by programmers and libraries using the function definitions described in Section X. The syntax permits expressions containing adjacent associative operators of the same precedence to be written without explicit parentheses; in this case, evaluation is from left to right. The syntax requires explicit parentheses wherever a nonassociative operator is used (unless it is the only operator in an expression). Explicit parentheses, whether optional or required, specify the evaluation semantics.

Two classes of run-time exceptions may arise from the evaluation of expressions. The first class consists of those exceptions that are part of the language semantics; they are enumerated below. The second class consists of implementation-dependent errors; these are discussed in Section XIV.

Every user-defined type, named by some TypeId T, inherits all standard operators applicable to the type that defines T.

1. *Relational Operators*

An expression is either a SimpleExpression, or a pair of SimpleExpressions combined with a RelationalOperator. The RelationalOperators are IN, ≠, =, <, <=, >, >=, and the FreeRelationalOperators. All relational operators are nonassociative. All relation operators yield a BOOLEAN result.

Relational Operators

Operation	*Description*	*Operands*
e IN s	set membership	e: integer or scalar, s: set
e1 ≠ e2	inequality	e1, e2: numeric, pointer, scalar, or set
e1 = e2	equality	e1, e2: numeric, pointer, scalar, or set
e1 < e2	less than	e1, e2: ordered scalar or numeric
e1 <= e2	less than or equal	e1, e2: ordered scalar or numeric
e1 > e2	greater than	e1, e2: ordered scalar or numeric
e1 >= e2	greater than or equal	e1, e2: ordered scalar or numeric

2. *Adding Operators*

A SimpleExpression is an optionally signed Term, a sequence of Terms combined with AssociativeAddingOperators, or a pair of Terms combined with a NonAssociativeAddingOperator. The AssociativeAddingOperators are + and OR. The NonAssociativeAddingOperators are -, XOR, and the FreeAddingOperators.

The operator -, in a signed Term, -T, computes the numeric negative of T if T is numeric, and computes the set complement of T if T is a set. The evaluation semantics of OR are "short circuit"—i.e., if the left operand evaluates to TRUE, then the right operand is not evaluated.

Adding Operators

Operation	*Description*	*Operands and Result*
e1 + e2	addition	same numeric type
	set union	same set type
e1 OR e2	logical disjunction	BOOLEAN
e1 - e2	subtraction	same numeric type
	set difference	same set type
e1 XOR e2	symmetric difference	same set type
	exclusive or	BOOLEAN

3. *Multiplying Operators*

A Term is either a sequence of Factors combined with AssociativeMultiplyingOperators, or a pair of Factors combined with a NonAssociativeMultiplyingOperator. The AssociativeMultiplyingOperators are * and AND. The NonAssociativeMultiplyingOperators are /, DIV, MOD, and the FreeMultiplyingOperators. (The evaluation semantics of AND are "short circuit"—i.e., if the left operand evaluates to FALSE, then the right operand is not evaluated.)

Multiplying Operators

Operation	*Description*	*Operands and Result*
e1 * e2	multipication intersection	same numeric type same set type
e1 AND e2	conjunction	BOOLEAN
e1 / e2	division	e1, e2: same numeric type returns FLOATING
e1 DIV e2	integer division	INTEGER
e1 MOD e2	integer remainder	INTEGER

4. *The Operator NOT*

A Factor is a sequence of NOTs followed by a parenthesized Expression, a Designator, or a Literal. The operator NOT acts logically negates a BOOLEAN value yielding a BOOLEAN value.

B. *Designators*

The syntax for a Designator is given by the rule:

```
    Designator
41      = Id*
42      | Id* ( {Expression ,} )
43      | Designator (. Id* | [ {Expression ,} ] | ↑)
```

This class provides for simple variables and constants, selection of a component of a record or array, denotations of structured values, function (and value-returning procedure) invocation, and dereferencing of pointers.

Some examples are:

```
X
R1.Shape
A1[3, X]
P↑
R_Type(E,F)
A_Type(E)
F(X)
C
```

where X is a variable, R1 is a record, A1 is an array, P is a pointer, R_Type is a record type, A_Type is an array type, F is a function (or value-returning procedure), and C is a constant. (The fifth and sixth examples are denotations of structured values as described in Section VI).

Designators can denote both *variables* and pure *values*. In these examples, X denotes a variable; R1.Shape denotes a variable if and only if R1 denotes a variable and Shape is a VAR component; A1[3, X] denotes a variable if and only if A1 denotes a variable; P↑ denotes a variable; and the remaining four examples denote pure values. This distinction is significant in procedure parameter binding, discussed in Section IX.

Within a program, the designation of a variant record component must be lexically embedded in a SELECT statement that verifies whether the record value has assumed the proper variant structure (see Section IX).

These operations may be composed. Thus

```
F(X)↑.Shape
```

is also permitted. (Note that neither a component of a structured value nor a dereferenced pointer is ever a function; this is why the syntax does not allow function call to follow any of the other operations in a designator.)

Several exceptions may occur in the computation of a designator. If a function or value-returning procedure is called, various kinds of parameter mismatches constitute exceptions, discussed in Section IX. If an array subscript reference is outside the range of the array's index representation, a range exception occurs. If a program attempts to dereference NIL or use of a variable whose value is undefined, an exception occurs. (As noted in [10], "many of these checks can be done or partially done during translation, thereby reducing execution costs. Several are very expensive in execution unless aided by special hardware, and consequently will often be suppressed.")

C. *Semantics of Arithmetic*

Integer arithmetic yields an exact integer result, except for the / operator, which produces an approximate floating point result. The DIV operator yields, for i1 DIV i2, the largest integer less than or equal to the quotient of i1 and i2. The MOD operator yields, for i1 MOD i2, the difference between the i1 and the product of i2 and i1 DIV i2.

Fixed-point arithmetic yields an exact fixed-point result, except for the / operator, which produces an approximate floating-point result. (There is also a function for exact fixed point division, FIXED_DIV (see Section XI), which computes an exact fixed-point result in a program-specified scale.)

Floating-point arithmetic yields an approximate floating-point result.

Division by zero is an implementation-independent exception. All other arithmetic operations may, in an implementation-dependent manner, produce additional exceptions (e.g., floating-point overflow). The semantics of exceptions is described in Section IX, Section X, and Section XIV.

IX STATEMENTS

Statements denote algorithmic actions and are the executable units of the language. They are divided into two classes: simple statements and structured statements. The simple statements are those that do not contain other statements; the structured statements do contain component statements.

A. *Simple Statements*

The simple statements are:

```
†    Statement
19       = Designator := Expression
20       | Id* ( {Expression ,} )
21       | ASSERT Expression
22       | GOTO Id*
23       | EXIT Id*
24       | RETURN [Expression]
25       | TERMINATE_PROCESS
26       | PROPAGATE_EXCEPTION [Id*]
```

1. *Assignment*

The assignment statement is used to set the value of a variable—that is, a VAR of a block, a VAR or OUT formal parameter, or a VAR component of a *structured* variable (i.e., a variable of a structured type). The right-hand side of an assignment statement must agree in type with the left-hand side of the assignment, as described in Section VII. The two sides of an assignment statement need not agree in representation; appropriate representational checks and conversions of representation, not affecting the semantics, will be made if they disagree. If it is not possible to convert the representation—for example, if a value is out of specified range—an exception is raised.

The left-hand side of an assignment statement is a *variable designator.* That is, it is a designator, as described in Section VIII, that denotes a variable. (Note that a designator that employs the function application syntax is never a variable designator.) The assignment operation is not defined for variables of type SIGNAL or for FILE types.

2. *Calling Procedures and Activating Processes*

A procedure (but not a value-returning procedure) can be called by a procedure call statement. The procedure name is followed by a list of actual parameters, enclosed in parentheses and separated by commas. The number of actual parameters must be the same as the number of formal parameters, and each actual parameter must agree in type with the corresponding formal parameter. Parameters are classified as CONST, OUT, or VAR by the procedure declaration (see Section X).

For CONST parameters, there may be an exception in the caller if the actual and formal parameters have incompatible representations as described above. Similarly, representation-incompatible formal and actual OUT parameters may produce an exception in the caller. For VAR parameters, not only the types, but also the representations, of the actual and formal parameters must match (see Section VII and Section X).

It is an error to call a procedure with the same actual parameter corresponding to distinct OUT parameters. There are further restrictions, discussed in Section X, on what the VAR actual parameters may be in a procedure call: these are concerned primarily with the interaction between VAR parameters and aliasing.

The process activation statement is syntactically identical to the procedure call statement. In place of a procedure name, a process name is used. The process is activated with the specified actual parameters and runs in parallel with the statements succeeding the point of activation, until it terminates by means of a TERMINATE_PROCESS statement or as the result of an exception. More details of the semantics of processes are given in Section X.

3. *Assertions*

The assertion statement consists of the reserved word ASSERT and a Boolean expression. The expression is expected to be true whenever control reaches the statement. The assertion

```
ASSERT e
```

is equivalent to the statement

```
IF NOT e
   THEN PROPAGATE_EXCEPTION INVALID_ASSERTION ENDIF
```

If the detection of this exception is suppressed, then the assertion has no effect. (In the future, assertions may aid optimization and program verification, but this is outside the state of the art for production programming.) Translators shall give warning if the expression in an assertion statement might have side-effects. The IF statement and the PROPAGATE_EXCEPTION statement are described below.

4. *The GOTO Statement*

A GOTO statement indicates that further processing should continue at another part of the program text, as indicated by the specified label. (A statement label must be declared at the head of the smallest block in which it is used to label a statement; like any other program entity, a label may be referred to only within its local scope. Note that the syntax permits labels only for the top-level statements of a block—e.g., statements at the top-level of a conditional, selection, or repetition construct cannot be labeled.)

The GOTO statement exits from lexical scopes—i.e., blocks, conditionals, iterations, etc.—but cannot be used to exit from a process, function, procedure, or module.

5. *Exiting Loops*

The EXIT statement causes a transfer of control from within a loop statement (see below) to the point immediately following that loop statement. Loop statements must be named by an identifier and the EXIT statement must cite this identifier to exit from within the loop. The EXIT statement can only exit lexical scopes and cannot be used to exit a process, procedure, function, or module.

6. *Returning from Procedures and Functions*

The RETURN statement is used to complete execution of a procedure or function. In the case of a procedure that does not return a value—that is, a procedure without a RETURNS clause in the header of its declaration (see Section X)—the RETURN cannot have an argument. In the case of a function or value-returning procedure, the RETURN must have an argument, which must be of the type specified in the RETURN clause of the routine declaration. It is an error to "fall out the bottom" of a routine: the RETURN statement must be used for normal completion of procedures and functions. (There is also an abnormal completion, provoked by the occurrence of exceptions, as discussed below and in Section X.) The declaration of functions and procedures is presented in Section X.

7. *Terminating Processes*

The TERMINATE_PROCESS statement is used to complete execution of a process. It is an error to "fall out the bottom" of a process: the TERMINATE_PROCESS statement must be used for the normal completion of a process. (There is also an abnormal completion, provoked by the occurrence of exceptions, as discussed below and in Section X.) The declaration of processes is presented in Section X.

8. *Propagating Exceptions*

The effect of this statement is to terminate the current context and to invoke the named exception within the dynamically enclosing context. The optional form, which omits the name, can be used only within an exception handler to propagate the same exception as that which invoked the exception handler. The search for an exception handler is through the scopes dynamically enclosing the point where the exception is raised. (See Section VII for an explanation of *dynamically enclosing scope.*)

Some examples of the PROPAGATE_EXCEPTION statement are:

```
PROPAGATE_EXCEPTION singular_matrix

PROPAGATE_EXCEPTION inertial_navigation_failing

PROPAGATE_EXCEPTION file_error

PROPAGATE_EXCEPTION
```

B. *Structured Statements*

Structured statements are constructs composed of other statements to be executed in sequence, conditionally, or repeatedly. The structured statements are:

```
†    Statement
27       = IF Expression THEN UnlabeledStatements
             {ELSIF Expression THEN UnlabeledStatements}
             [ELSE UnlabeledStatements] ENDIF
28       | SELECT Expression
             {CASE {ManifestConstant ,} => UnlabeledStatements}
             [ELSE UnlabeledStatements] ENDSELECT
29       | WHILE Expression DO UnlabeledStatements ENDWHILE
30       | FOR Name : Representation DO UnlabeledStatements ENDFOR
31       | LOOP Id* UnlabeledStatements ENDLOOP Id*
32       | [SIMULATED_TIME] BEGIN Block END
```

where UnlabeledStatements is defined by

```
     UnlabeledStatements
33       = {Statement ;} [;]
```

1. *Conditionals*

The IF statement and the SELECT statement contain zero or more component statement sequences. At most one of these is chosen to be executed.

In general, an IF statement has the form

```
IF be1 THEN seq1
   ELSIF be2 THEN seq2
   ELSIF be3 THEN seq3
     .....
   ELSE seqN
ENDIF
```

The be's are Boolean expressions, the seq's are statement sequences, and the final ELSE clause is optional. The be's are evaluated in order until one evaluates to TRUE or all are evaluated to FALSE. In the former case, the corresponding seq is executed, completing the evaluation of the IF statement. In the latter case, if an ELSE clause is present, seqN is executed; otherwise no seq is executed.

In general, a SELECT statement has the form

```
SELECT select_expression
   CASE se1 => seq1
   CASE se2 => seq2
     .....
   ELSE seqN
ENDSELECT
```

The se's are ManifestConstants with distinct values, the seq's are statement sequences, and the final ELSE clause is optional. To execute this statement, select_expression is evaluated. If this value is equal to an se, then the execution of the corresponding seq completes the execution of the SELECT. If this value is different from all the se's, and an ELSE clause is present, then seqN is executed; otherwise, no seq is executed.

2. *Repetitions*

The LOOP, WHILE, and FOR statements contain component statements to be executed repeatedly. The condition for terminating the repetition is different for each. The LOOP statement consists of a statement sequence bracketed by the keywords LOOP and ENDLOOP. The repeated execution of this statement sequence terminates when the simple statement EXIT is executed.

The WHILE and FOR statements consist of a header that specifies the condition for terminating the repetition and a body that contains the statement sequence to be repeated, bracketed by the keywords DO and ENDWHILE, or DO and ENDFOR.

The body of the WHILE statement is executed repeatedly while the value of the Boolean expression following the WHILE is TRUE. (If this value is FALSE upon entry, no statements between DO and ENDWHILE are executed.)

The body of the FOR statement is executed for each value of the representation of its control constant, which must be an ordered scalar or integer range. Each iteration should be understood as executed in a lexical scope in which the identifier following the keyword FOR is bound, as a constant, to the corresponding value.

3. *The BEGIN-END Statement*

The BEGIN-END statement is

```
↑   Statement
32      =  [SIMULATED_TIME] BEGIN Block END
```

where Block is defined by

```
    Block
6       =  {Declaration ;} [:] [ExceptionHandler [:]] {[Id* :] Statement ;} [:]
```

The declarations provide local nomenclature, which may be cited within the block, as described in Section VII and Section X. The statements, which may have labels, are then executed in the order given (except as specified by the occurrence of GOTO statements, described above).

The ExceptionHandler clause, if present, declares an exception handler for the scope, as described in Section X.

The keyword SIMULATED_TIME specifies that the block is to be executed as a single sequential process (in which parallel processes are construed as pseudo-parallel). The effect is to introduce an artificial time within the BEGIN-END statement. All statements take zero units of that time for their execution, except the standard procedure call DELAY(n) (see Section XI), which requires n units of the artificial time for its execution. A SIMULATED_TIME prefix, within a SIMULATED_TIME statement, has no effect. (Further details can be found in the parallel processing subsections of Section X.)

X COMPOSITE DECLARATIONS

A declaration that involves subsidiary statements and declarations is called *composite.* The composite declarations are function, procedure, module, module template (and template instantiation), and process declarations. (The other declarations, the *simple declarations*, were described in Section VII.)

A. *Functions and Procedures*

We will use the word *routine* to mean function or procedure. The declaration of a routine defines and names a section of program text. It can then be activated by a procedure call statement (see Section IX) or by the appearance of a procedure or function designator within an expression (see Section VIII). The function is always used to compute and return a result and its call is therefore an expression. The procedure is used in one of two ways: to compute a value in the same way as a function (but with the possibility of causing side effects during the computation) or to execute a sequence of statements. In the former case, the procedure call has the same syntactic standing as a function—it is used as an expression or subexpression. In the latter case, the procedure is called solely for its side effect—its effect on the environment of the call—and its syntactic standing is as a statement.

The syntax for routine declarations is given by the rules:

```
†    Declaration
 12      = FUNCTION [EXTENDS | REPLACES] ExtendedId Formals [INLINE]
               Block ENDFUNCTION ExtendedId
 13      | PROCEDURE [EXTENDS | REPLACES] ExtendedId Formals [INLINE]
               Block ENDPROCEDURE ExtendedId

     ExtendedId
 56      = Id* | NOT | ↑ | := | RelationalOperator |
               AssociativeMultiplyingOperator |
               NonassociativeMultiplyingOperator |
               AssociativeAddingOperator | NonassociativeAddingOperator

     FormalRepresentation
 85      = ARRAY ( ManifestConstant DIMENSIONS OF Representation )
†86      | Representation

     Formals
 87      = [RETURNS Representation]
               ( {(CONST | OUT | VAR) {Names : FormalRepresentation ;}} [;] )
```

The declaration begins with one of the keywords FUNCTION or PROCEDURE. This is optionally followed by EXTENDS or REPLACES, which are discussed below. Next comes the identifier, which will name the routine. The identifier is denoted ExtendedId (for extended identifier) rather than Id* so that the routine declarations may be used to associate meanings with the operators as well as with normal identifiers.

The Formals section of the routine header consists of an optional RETURNS clause and a formal parameter list. The RETURNS clause specifies the representation of the result to be computed. Function declarations must include a RETURNS clause; procedure declarations include the RETURNS clause if they are value-returning, and omit it if their calls are to be used as statements. It is important to note that only a Representation—not a FormalRepresentation—may be specified for result values. This means, in particular, that if an array value is to be returned, its length is not unresolved but rather is known at the time of routine declaration.

The formal parameter list describes the parameters that will be provided in calls of the routine. Each parameter has several attributes:

- Its class—CONST, OUT, or VAR,
- Its name, and
- Its FormalRepresentation.

The significance of the class is discussed below; note, however, that all parameters of a function must have the class CONST. More than one name may appear, permitting conciseness in the header when several different formal parameters have the same class and FormalRepresentation. Finaliy, the FormalRepresentation is either a length-unresolved array type—discussed below; a Representation—discussed in Section VI; or a SPACE—discussed in Section

XIV.

In this introduction to routines, we have already described those basic aspects substantially unchanged from Pascal. The subsections to follow describe the features that are new in our language: the INLINE attribute, the EXTENDS and REPLACES attributes, the formal parameter classes, length-unresolved array parameters, aliasing and entire variable rules, and the distinction between pure and impure functions.

1. *Redeclaration of Routines*

The association of several routines with a single name is called *overloading*. For example, the standard function "+" is overloaded in most languages: one definition is used to add a pair of integers and another to add a pair of floating point numbers. In our language, this facility has not only its common use with standard routines, but is also applicable to user-defined routines.

When a routine name is overloaded, each call must be examined and a choice made among the multiple definitions. This is possible in any strongly typed language so long as the basis for discriminating among several definitions is the number and type of the formal parameters for each definition. This is exactly the facility we provide.

We avoid erroneous overloading by syntactically distinguishing two classes of declarations: those that extend a routine to apply to a new n-tuple of argument types, indicated by the keyword EXTENDS, and those that replace an existing definition for a particular n-tuple of argument types, indicated by the keyword REPLACES. The EXTENDS option is used to add a new n-tuple of argument types to those for which the routine is defined. The REPLACES option, on the other hand, applies if the existing definition for a particular type n-tuple is to be made inapplicable in some scope.

(Note that, in what follows, when we explain the meaning of a call of a routine in a particular context, we refer only to declarations of the routine within the lexical scope of the call and not hidden by a module boundary [see Section B]. If a declaration is hidden by a module boundary, the rules should be interpreted as though there were no such declaration.)

A successful REPLACES declaration will replace the routine definition corresponding to the specified formal types during the new scope, but other definitions continue in force. The EXTENDS option is used to declare an additional body for an existing routine, for the specified formal parameter types, during the scope of the declaration. For example, suppose there is an enclosing f(CONST x:INTEGER). Then an inner scope may declare:

```
PROCEDURE EXTENDS f(VAR x:FloatingRepresentation)
...
ENDPROCEDURE f
```

In this inner scope, f(7.0) is a call of the second body and f(7) is a call of the first body.

If a scope declares a routine, when there is already a routine with the same name, and does not use EXTENDS or REPLACES, this hides all other definitions for the new scope. (A warning during program translation will alert the programmer to the possibility of a conflict between the names of a system routine and the user routine.) A program can only extend a routine with formal parameter types for which no enclosing definition exists, or replace a routine with formal parameter types for which some definition exists.

If an existing definition for routine f is a function, then f can be extended or replaced

only with a function or value-returning procedure. Similarly, a procedure that is not value-returning can be extended or replaced only with another procedure that is not value-returning.

It is important to note that the types of formal parameters—and not the representations—are considered for the purpose of these rules. Thus if there is a definition for a particular representation n-tuple, then a declaration for another representation n-tuple cannot use EXTENDS if the two type n-tuples are the same.

2. *Formal Parameter Classes*

The only formal parameter class allowed for functions is CONST. For procedures, there are two additional classes: VAR and OUT. In all cases, formal and actual parameters must agree in type, as described in Section VI.

The CONST class specifies that the parameter is constant within the scope of the routine and is never modified within that scope. A CONST formal cannot be used as a VAR or OUT actual within the body. That is, if passed to another procedure, it must be as a CONST parameter.

The class VAR has the semantics of passing-by-reference. Therefore, we require that VAR formal and corresponding actual parameters must have identical representations—that is, they must represent the same abstract type in the same way. Additional restrictions, arising from aliasing, are presented below. If a procedure or process has a formal parameter of class VAR, then any corresponding actual parameter must be a variable.

The class OUT is used when the purpose of the procedure call is to set the corresponding actual. Therefore, the value of the actual is not passed in to the procedure and may be undefined at the time of the call. The procedure computes using a formal parameter allocated in its local storage and assigns the value of this formal parameter to the actual parameter at the procedure exit. The value is undefined if the procedure terminates because of an exception.

3. *Length-Unresolved Array Parameters*

A formal parameter representation is a representation or a length-unresolved array type (or, as discussed in Section XIV, a SPACE). An example of a length-unresolved array parameter is:

```
VAR Fa: ARRAY (3 DIMENSIONS OF INTEGER [1..10])
```

Within the body of a function or procedure with such a formal parameter, the actual integer dimensions of Fa are available as LOW(Fa,i) and High(Fa,i) for each dimension i. (A length-unresolved array always has integer indices; by contrast, the *length-resolved* form of array presented in Section VI may have integer or scalar indices.

4. *Value-Returning Routines*

The FUNCTION deciaration is used to declare value-returning routines that do not has side-effects. They take only CONST parameters and are permitted to assign only to local variables. They are not quite mathematical functions, since they are permitted to read the values of globals and hence may return different answers for apparently identical calls. If f and g are functions, then a compiler may freely reorder their calls within an expression.

It is also possible to declare a PROCEDURE that returns a result. This is essentially an impure function. It is not constrained as to parameter class or nonlocal assignment. A

compiler will usually not be able to reorder the calls of such procedures.

5. *INLINE Attribute*

The INLINE keyword advises the compiler, where possible, to optimize in favor of speed of call and reduction of linkage code even though, in consequence, procedure or function bodies are replicated at every call. The INLINE routine cannot be recursive; its semantics are not affected.

6. *Naming and Aliasing*

Nomenclature can be introduced in several ways. These include declaration of locals of BEGIN-END statements or modules, declaration of formals of processes, procedures, or functions, and the FOR statement. Some of these may lead to *aliasing*: the existence of scopes in which (1) some entity has two distinct names and (2) assignment using one of the names can cause a change discernible through the other name. Such scopes may arise when a variable is passed as an actual to a procedure's VAR formal parameter and is also either referred to freely by the procedure body or passed to another VAR formal of the same procedure call.

To explain the restrictions imposed to eliminate this problem, it is necessary to make clear what constitutes a variable, to list precisely some forbidden constructs, and to indicate the computation that may be required to assure no aliasing in the most general case. The reader is warned that the remainder of this section has been written as a concise set of rules for this Report; a more tutorial and motivational discussion of these issues will be found in the Design Discussion.

A variable is that entity permitted as the left-hand side of an assignment statement—that is, a designator that denotes a variable and has the form

```
†     Designator
41      = Id*
43      | Designator (. Id* | [ {Expression ,} ] | † )
```

We forbid only those instances of aliasing that arise from calls of routines and processes. To determine whether a particular routine call is forbidden, consider the list containing the actual parameters corresponding to VAR formals as well as all variables referred to freely in the body of the routine. The aliasing restrictions are:

- This combined list may have no repetitions.
- If the list contains a variable of the form R.C, then it may not also contain R. (It may contain R.D if D is different from C.)
- If the list contains a variable of the form A[i,j,...] where any of the subscripts is not a manifest constant, then it may not also contain A or any variable of the form A[k,l,...]. (However, the list may contain A[i,j,...] and A[k,l,...] if enough of the subscripts are manifest constants so that it can be assured during translation that the subscripts are different.)
- If the list contains a variable of the form P†, and Q is a pointer of the same type as P, then the list may not contain Q or Q†.

(The creation of sharing patterns through pointers and parallel processes is permissible—see Section X of the Design Discussion.)

Unfortunately, the detection of one of these forbidden cases may require examining the definition and every call of a routine P. Moreover, in looking at the definition, one must look at the definitions of all the routines Q called from within P, since an alias-producing reference to a free variable might be made by one of these inner calls. Thus a transitive closure computation may be needed to determine whether there is aliasing or not. All of this is especially costly if the P's and Q's are defined in a number of separate modules or even separate compilation units. The programmer can reduce the need for alias checking by avoiding VAR formal parameters and, in coding encapsulations, by avoiding the simultaneous exportation of a variable v and the importation or exportation of a routine P that refers freely to v (see below).

B. *Modules*

The module (based on Wirth's module [17]) is our primary mechanism for abstraction. A module is a collection of declared entities, together with program text that may initialize these entities—the objects of the module. These entities come into existence when control enters the lexical block to which the module is local. The module is, in essence, a fence around its objects: it is a boundary across which nomenclature flows, in or out, but only as explicitly specified. This facility provides an effective means of making selectively available those objects that represent an intended abstraction, and of hiding those objects that are viewed as details of implementing of the abstraction. A module encapsulates those parts that are not essential to the remainder of a program or that are to be protected from inadvertent access. Modules may be nested.

The syntax of a module declaration is

```
    Module
2       = MODULE Id* Imports Exports Block ENDMODULE Id*
3       | MONITOR Id* Imports Exports Block ENDMONITOR Id*
```

where Imports and Exports are defined by

```
    Imports
4       = {IMPORT {ExtendedId [FROM Id*] [AS ExtendedId] ,} ;}

    Exports
5       = {EXPORT [OPAQUE | TRANSPARENT | READONLY | VAR]
              {ExtendedId [FROM Id*] [AS Id*] ,} ;}
```

(Note that a monitor is a module that has special properties facilitating its use in conjunction with parallel processes, as described below.)

The mechanism for specifying the externally defined entities to be visible within a module is *importation*. The mechanism for specifying the internally defined entities to be visible outside a module is *exportation*. Any declared entity is subject to these mechanisms, i.e., constants, variables, functions and procedures, processes, types, and other modules.

1. *Module Bodies*

The block that follows the imports and exports of a module is its body. The declarations of this block define the local objects of the module and the objects exported by the module will be a subset of these locals. (It is not permissible to import and then export the same entity.) The statements of this block constitute the initialization for the module—they are executed following the declaration of the module's local objects and before the execution of the statements of the enclosing block in which the module is declared. The declarations and statements of the module may refer to local names, imported names, and the standard types, representations, functions, and procedures of the language. (The standard types and representations are those described in Section VI; the standard routines are those described in Section XI and Section XII.) Note that the local objects declared in the module body are *own* objects of the module—that is, their values are defined by the module declaration and then preserved from one call on the module to the next.

2. *Importation*

A module may have one or more import lists. These lists specify the names of the entities to be imported. These names, together with the local names defined in the module (whose lifetime is that of the scope of the module), are the only names that may be used within the module. Only entities defined in the scope containing a module may be imported into the module—these will be the names of entities declared by routines, processes, and BEGIN-END statements lexically enclosing the module, as well as the names of entities exported by other modules within the same scope. The name of a module may appear on an imports list. In this context, it is just a shorthand for the list of names exported by that module.

For example, assume that a program has two modules M and N, and we are constructing the imports list for N. Suppose we wish to import x and y, which are in the lexical scope of N, as well as all the entities exported by M (assuming that M is in this scope too). Then we can write:

```
IMPORT x, y, M;
```

An import list may optionally specify *from* which other module a variable is exported. (In importing from a separate compilation unit [see Section XIII], this specification is mandatory.) For example,

```
IMPORT x FROM M, y FROM N, P;
```

specifies that the x and y to be imported are the x and y exported, respectively, by modules M and N. In addition, all entities exported by module P are to be imported.

An import list may optionally specify the name by which an entity is to be known within the module. An example is

```
IMPORT x AS x1, y AS x2;
```

The two options may be combined, as in

```
IMPORT x FROM M AS x1, y FROM N AS x2, P;
```

In this example, the entities from P are not renamed. (The aliasing restrictions given above should be interpreted with respect to a consistent naming scheme.)

3. *Exportation*

In an export list, the programmer lists those local entities of the module to be visible outside the module. In addition, the external use of the entities may be restricted, and they may be renamed.

To illustrate the simple case, suppose the module M defines Z and W; then its structure would be:

```
MODULE M
   IMPORT ...any external entities needed...;
   EXPORT Z, W;
   ...declarations of Z, W, and other locals...
   ...module body...
ENDMODULE M
```

The shorthand of using a module name to stand for the list of entities that the module exports is available here too. Thus, if one of the local declarations of M was an inner module P, and we wished to export all of its exported entities from M, the structure used would be:

```
MODULE M
   IMPORT ...any external entities needed...;
   EXPORT Z, W, P;
   MODULE P ... ENDMODULE P
   ...declarations of Z, W, and other locals...
   ...module body...
ENDMODULE M
```

The user of this nest of modules is not aware of its inner structure and has no way of distinguishing Z from an entity that came by way of the inner module P.

We now explain the attributes TRANSPARENT, OPAQUE, READONLY, and VAR. The first two apply to exported type and representation identifiers; they control whether the structure of the type or representation is known outside the module—TRANSPARENT—or not known—OPAQUE. If neither qualifier is used for a type or representation identifier, OPAQUE is assumed. As an example, consider:

```
BEGIN
   MODULE M
      EXPORT TRANSPARENT R;
      REPRESENTATION R= RECORD VAR x: INTEGER [1..10] ENDRECORD;
   ENDMODULE M;
   VAR v: R;
   (* v.x is legal here; if the TRANSPARENT is omitted, *)
   (*      then v.x is no longer legal *)
END
```

If a representation R is exported opaquely by a module M, than program text outside the module may declare variables, constants, and formal parameters of representation R, and may pass such entities as arguments to appropriate routines exported by M. Such program text may not select components of such entities (even if they are actually records), index into such components (even if they are actually arrays), or apply the assignment, equality, and inequality operations (even if they are applicable to the representation within the module). If the implementer of M wants to provide to the user equality, inequality, assignment, or any other operation applicable to R within M, then the name of the operation must be explicitly exported from M.

The attributes VAR and READONLY apply only to identifiers for variables and constants, and indicate whether they may be treated only as constants outside the module. If neither is used, READONLY—meaning treat as a constant—is assumed. An object that is exported READONLY is subject to the same restrictions as a constant outside the module from which it is exported; therefore it can only be a CONST actual parameter.

It is important that the reader understand how modules and opaque exportation can be used in combination to provide well-structured programs. A paradigm of such use is as follows:

```
MODULE M
   EXPORT OPAQUE R;
   EXPORT :=, F, G, ... ;
   REPRESENTATION R = ... ;
   FUNCTION Init RETURNS R () ... ENDFUNCTION Init;
   FUNCTION F RETURNS BOOLEAN (VAR x: R) ... ENDFUNCTION F;
   PROCEDURE G(CONST x: R) ... ENDPROCEDURE G;
   ...
ENDMODULE M
```

The user of this module obtains a representation R whose structure is known only inside M but hidden to the user of M. The user may declare any number of variables or constants with this representation. However, because the structure of R is hidden, the only operations applicable to these declared entities will be those specifically provided by M—Init, F, and G. Thus the user of M builds on the abstract framework defined by the exported operations and is insulated from the concrete details of M's definition.

By means of these attributes, an export list can impose restrictions on the environment outside the module, but can never relax existing restrictions. Therefore, if an identifier is either CONST or exported READONLY from an inner module, an enclosing module cannot export it as VAR. (Note that a READONLY object is not constant, since it may be changed by its defining module.) Similarly, if a type is OPAQUE, it cannot be made TRANSPARENT by an enclosing exportation.

These attributes cannot be used together with the shorthand mentioned above, that is, using a module name to stand for the list of entities exported by the named module. The exportation from M of all entities exported by an inner P, as illustrated above, imposes no further restrictions on these entities. To add restrictive attributes, the programmer must explicitly list and qualify the exported entities of P.

The FROM and AS clauses of import lists may also be used in export lists. For example, consider

```
MODULE Convert
   EXPORT x FROM M AS Mx, x FROM N AS Nx;
   MODULE M
      EXPORT x;
      ...
   ENDMODULE M;

   MODULE N
      EXPORT x;
      ...
   ENDMODULE N;

   ...
ENDMODULE Convert
```

Here the FROM clauses are used to distinguish the two versions of x and the AS clauses provide an external interface of Convert that distinguishes the two versions by assigning distinct names. (The aliasing restrictions given above should be interpreted with respect to a consistent naming scheme.)

4. *Uniqueness of Names*

An entity may be referred to by name only in contexts where it can be referred to without ambiguity. In the example just given, the x's can be referred to in the export list of Convert and also outside of Convert (by the names Mx and Nx). They cannot be referred to in that part of Convert outside the inner modules M and N. Name disambiguation—using FROM, and renaming—using AS, is permitted only in import and export lists and not in any other program context.

A simple programming convention—using unique names on a program-wide basis—will avoid any need for renaming if used consistently. This convention is sometimes impractical—for example, because a program is to be constructed, in part, using previously written modules. It is then possible to disambiguate with either of two dual methods: renaming in the export lists of the modules defining ambiguously-named entities, or renaming in the import lists of each module using such entities.

For example, if P exports only X, and it is desired to use P in a context in which the name X would clash, the following structure is used:

```
BEGIN
  VAR x: BOOLEAN ...this is the source of the clash...
  MODULE NEWP
    EXPORT x AS newx;
    MODULE P ...the existing module... ENDMODULE P;
  ENDMODULE NEWP;
  ...here x and newx are used...
END
```

Note that renaming can be used together with restrictive attributes, but not with the "all exports of a module" shorthand. Thus

```
EXPORT TRANSPARENT z1 AS z, x1 AS x;
EXPORT M, P;
EXPORT y2 AS y
```

shows the legal combinations. (Here x and z are the names of types or representations, y is the name of any entity other than a module, and M and P are inner modules.)

5. *Modules and Overloaded Functions*

The rules concerning modules are based on the principle that programmers should be encouraged to adopt a style in which the user of a module does not depend on how that module implements the services that it provides. A consequence, in the case that modules are used in combination with overloaded functions, is the rule that the user of a module that exports a routine must view the exported definition or definitions as extensions (in the sense of the rules on EXTENDS and REPLACES presented above). For example, consider the following:

```
MODULE M
   EXPORT f;

   MODULE N
      EXPORT f;
      FUNCTION f RETURNS BOOLEAN (CONST x: Colors)
         ...body of f...
      ENDFUNCTION f;
   ENDMODULE N;

   FUNCTION EXTENDS f RETURNS BOOLEAN (CONST x: Flavors)
      ...body of extension...
   ENDFUNCTION f;

   ...body of M...
ENDMODULE M
```

The inner module N declares and exports (a single alternative of) a function f. The outer module M extends this function with a second alternative. When M exports f, it is exporting both alternatives. If the scope outside M contains a function f, then the two exported alternatives have the significance of extensions of that definition. If they constitute an illegal extension—for example, because there is an outer definition that acts on a single Colors argument—then the program is in error because of a name clash—and this is the same error that would arise if the outer and the two inner definitions were not separated by a module boundary. (Such a clash can be simply corrected—if M renames f to ff, then the exported ff and the outer f become completely independent.)

C. *Module Templates*

We provide a notation for declaring a *module template*: a form with explicit formal parameters. These formal parameters may then be instantiated to yield a a module. The syntax of a module template is

```
†     Declaration
15        = TEMPLATE Id* ( Ids ) Module ENDTEMPLATE Id*
```

The user simply embeds a module declaration in a header listing the template name and formals and a closing "ENDTEMPLATE Id*". An example is

```
TEMPLATE Tm(i)
  MODULE M
    EXPORT ...;
    REPRESENTATION IndexRange = INTEGER [1..i];
    REPRESENTATION BoolArray = ARRAY(IndexRange OF BOOLEAN);
    VAR A: BoolArray;
    ...more declarations and code using A...
  ENDMODULE M
ENDTEMPLATE Tm
```

Within the scope of this declaration, the user may instantiate the template (as described below) and obtain the effect of a declaration of the module M with a particular size i for BoolArray.

The formal parameters of a template are simply a list of identifiers. They may be used anywhere within the template body (i.e., the enclosed module) that an identifier is syntactically legal. For example, the identifiers may be used as variable, constant, function, procedure, type, or representation names. In particular, the name of the enclosed module may be a formal parameter of the template.

D. *Instantiating Module Templates*

A module template may be instantiated by the declaration

```
†     Declaration
16        = INSTANCE Id* ( Ids )
```

The actual parameters provided in this instantiation are identifiers. The semantics of the instantiation are exactly as though the embedded module declaration of the cited template, with the actual parameters substituted for the corresponding formal parameters throughout, were written in place of the instantiation.

As an example of this facility, suppose we have a statement S and wish to protect the execution of S with certain program-defined synchronizing primitives, Lock and Unlock. Then we declare the template

```
TEMPLATE Safe(P, SafeP, Name)
  MODULE Name
    IMPORT Lock, Unlock;
    EXPORT SafeP;
    PROCEDURE SafeP()
      Lock(); P(); Unlock();
    ENDPROCEDURE SafeP;
  ENDMODULE Name
ENDTEMPLATE Safe
```

In order to use the template, the user should declare a procedure consisting of the statement to be "safely" executed by

PROCEDURE SProcedure() ...the statement S... ENDPROCEDURE SProcedure

and then instantiate the template by the declaration

INSTANCE Safe(SProcedure, SafeSProcedure, NewName)

Following this declaration, the exported procedure SafeSProcedure is available and provides "safe" execution of the statement S.

E. *Processes and Synchronization*

1. *Processes*

A process declaration describes a sequential algorithm—including its local objects—that is intended to be executed concurrently with other processes. No assumption is made about the speed of execution of processes, except that this speed is greater than zero.

The syntax of a process declaration is:

```
†     Declaration
†17       = PROCESS Id* Formals
                Block ENDPROCESS Id*
```

A process declaration is similar in form to a procedure declaration, and the usual scoping rules are in force. Additional directives dealing with the scheduling priority, the maximum number of concurrent instances, and the amount of storage space of a process are discussed in Section XIV.

If a process is declared and activated within a scope, that scope cannot terminate until that process terminates. Processes declared and activated within a module impose the same termination constraint on the scope lexically enclosing the module. A process is terminated by a TERMINATE_PROCESS statement (or by the propagation of an exception not handled by the process).

2. *Activating a Process*

A process activation statement activates a new instance of a process. It is syntactically like a procedure call but differs in that the activated process instance runs in parallel and the calling program continues. A process declaration defines a pattern of behavior, and each corresponding activation initiates the execution according to this pattern. Thus reference to the same process declaration in several process activations will yield concurrent execution of several instances of a single process (usually with different actual parameters).

A process call cites actual parameters to the process. Define the *life scope* of a variable, formal parameter, or process as the scope given by the lexically enclosing BEGIN-END statement or routine or process declaration. It is required that the life scope of a VAR actual parameter to a process be greater than or equal to the life scope of the process.

Processes, unlike some routines, do not return results.

3. *Monitors*

The *monitor* is a special kind of module that prevents simultaneous access by several processes to a set of common objects. Variables that are to establish communication or data transfers between processes are declared local to a monitor. They are accessed by procedures local to, and exported from, the monitor—called *monitor procedures*. If a monitor procedure is called, another process calling a monitor procedure of the same monitor is delayed until the first process either returns from the procedure or waits for a local signal (see below). The call of a monitor procedure by another monitor procedure of the same monitor does not cause delay.

Processes may be declared within a monitor, but the rules for synchronizing monitor procedures apply also to processes declared within the monitor. Two processes are never permitted to execute concurrently within a monitor module.

Nesting of monitors is permitted; no additional exclusion results.

4. *Signals*

Signals are declared in a program as variables of a monitor. They are members (along with files) of that class of variables for which assignment, equality, and inequality are not defined. There are only three standard procedures that can be applied to signals:

- The procedure call WAIT(S,R) delays the process until it receives the signal s. The process is given delay rank R, where R is a positive integer. WAIT(S) is a short form for WAIT(S,1).

- The procedure call SEND(S) sends the signal S to a process waiting for S with smallest delay rank. If several processes are waiting for s with the same delay rank, the process waiting the longest receives S. If no process is waiting for S, the statement SEND(S) has no effect.

- The BOOLEAN procedure AWAITED(S) yields the value TRUE if and only if there is at least one process waiting for S.

If a process executes a wait within a monitor procedure, then other processes are allowed to execute monitor procedures of this monitor, even though the waiting process has not completed its monitor procedure. If a send is executed within a monitor procedure, and if the signal is sent to a process waiting within the same monitor, then the receiving process begins execution within the monitor and the sending process is delayed. Hence, both the wait and send operations must be considered as enclaves in the monitor exempted from the exclusion rule.

Signal variables cannot be declared outside a monitor; they may not be exported from a monitor module.

5. *Simulation*

A BEGIN-END statement may be prefixed by the directive SIMULATED_TIME, which changes the meaning of its local processes and monitors, to yield a single sequential execution. Any processes declared within are quasi-parallel rather than asynchronous.

It is by reference to a record of the number of active unblocked processes within the simulation that process invocation, the resumption of blocked processes, process termination,

and the suspension of processes are all defined. The procedure DELAY(I) provides for a simulated delay for I time units. Simulated time advances when all processes within the simulation are suspended. The procedure CLOCK_TIME() returns the current simulated time.

If a process declared within a simulation enters a monitor declared outside the simulation, the normal definitions of monitors apply and the entire simulation may be delayed in real time.

Simulations, if nested within simulations, have no further effect.

F. *Exception Handling*

Our exception-handling facility provides program error recovery without distorting program structure. Two main classes of exceptions are distinguished:

- Exceptions defined by the user to report erroneous states detected explicitly by user code.
- Exceptions defined by the implementation to report erroneous states detected by the hardware, nucleus, or compiled code—called *traps*.

We describe the definition of handlers for user exceptions. (The declaration of trap handlers and the suppression of traps is dealt with in Section XIV. In Section IX, we described the PROPAGATE_EXCEPTION statement, with which the user can generate an exception based on a user-detected error.)

1. *Defining Exception Handlers*

An exception handler can be defined within any block—that is, within any process, procedure, function, module, or BEGIN...END statement. Only one exception handler may be defined in a block; that definition syntactically precedes the statements of the block. Thus the syntax of a block is

```
    Block
6       = {Declaration ;} [;] [ExceptionHandler [;]] {[Id* :] Statement ;} [;]
```

The statements of the block invoke procedures, functions, and standard operations. If control returns to this block as the result of a PROPAGATE_EXCEPTION statement (i.e., not by a RETURN statement), then control processing continues at the block's exception handler. (There is no other way that control can reach this handler.)

The exception handler definition has the form

```
    ExceptionHandler
18      = EXCEPTION {CASE Ids = > UnlabeledStatements}
              [ELSE UnlabeledStatements] ENDEXCEPTION
```

One of the programmed cases is chosen, based on the exception cited by the PROPAGATE_EXCEPTION statement that caused transfer of control to this handler. The statement sequence of this case is executed; the ELSE clause defines the statement sequence to be executed if the exception is not explicitly named by any other case.

A statement sequence is permitted as a case if and only if it is a valid terminal statement sequence of the current block. The semantics of execution of a selected case are the same as those of the execution of its statement sequence as the terminal statement sequence of the scope without an exception handler. (Note that, independent of exception handling, different scopes are exited in different ways—e.g., RETURN for functions and TERMINATE_PROCESS for processes.)

If an exception handler definition contains no ELSE clause, the semantics are as though there were an ELSE clause of the form

ELSE PROPAGATE_EXCEPTION

—that is, the effect is to propagate the same exception. Similarly, if a block contains no exception handler definition, the semantics are as though there were a definition of the form

EXCEPTION ELSE PROPAGATE_EXCEPTION ENDEXCEPTION

G. *Order of Declarations*

If any composite declaration listed above depends on a simple declaration of the same scope, then that simple declaration must lexically precede the composite declaration. No lexical ordering restriction is placed on the various composite declarations of a scope: the body of any composite declaration of a scope may invoke any other composite declaration of the same scope (subject, of course, to the access restrictions imposed by a module).

Note that the entities declared by any single module of a scope must be ordered in accord with this rule and the ordering rule for simple declarations (see Section VII). The bodies of these entities may cite the names of the others if appropriately imported and exported. The programmer need not (and, in general, cannot) order modules so that the constituent simple declarations are ordered in accord with the rule of Section VII (declaration before use). However, the program is valid if and only if such an ordering of the constituent declarations does exist.

In general, the body of a composite declaration may involve a sequence of local declarations. These local declarations are themselves subject to the ordering rules stated here and in Section VII.

XI STANDARD OPERATIONS

Each language implementation must provide several standard operations that are needed to support various language features. These operations are provided by standard FUNCTIONs and PROCEDUREs. The required operations are described here in brief (they are also described in more detail in previous sections of this report). If a standard operation S applies to a type which is the right-hand side of a user-declaration of a TypeId T, then S is automatically extended to apply to T.

If oe_name has representation oe_repr—an integer, ordered enumeration, or alphabet representation, then the function

FUNCTION SUCC RETURNS oe_type (CONST oe_name: oe_repr)

yields the immediate successor of the given argument, while the function

FUNCTION PRED RETURNS oe_type (CONST oe_name: oe_repr)

yields the immediate predecessor of the given argument. An application of PRED to the first element of an enumeration or of SUCC to the last element raises an OUT_OF_RANGE exception.

If oe_name is either an ordered enumeration or an alphabet, the function

FUNCTION FIRST RETURNS oe_repr (CONST oe_name: oe_repr)

yields the value of the first member of the representation that is its argument, while the function

FUNCTION LAST RETURNS oe_repr (CONST oe_name: oe_repr)

yields the value of the last member of the representation that is its argument.

If array_name is an array, including a length-unresolved array, and index_repr is the index representation of this array (which will always be an integer representation for the length-unresolved case), then the functions

```
FUNCTION LOW RETURNS index_repr
                (CONST array_name: array_repr; CONST dimension: INTEGER)

FUNCTION HIGH RETURNS index_repr
                (CONST array_name: array_repr; CONST dimension: INTEGER)
```

yield the low and high values, respectively, of the range of indices of array_name in the dimension given as the second argument. The value of dimension must lie between 1 and the maximum dimension of array_name.

The procedure

```
PROCEDURE NEW(OUT p: pointer_repr)
```

where pointer_repr is a pointer representation, returns a pointer to a pointer to a new anonymous variable whose representation is the base representation of pointer_repr. The dual of NEW, the procedure FREE, is implementation-dependent and is described in Section XIV.

The procedure

```
PROCEDURE MOVE(VAR object: array_repr; i: index_repr;
                        CONST source1: array_repr1; i1,j1: index_repr1;
                                ...
                                        sourcen: array_reprn; in,jn: index_reprn)
```

assigns to the array object, beginning at its index i, the catenation of the specified source array ranges. The arrays object, source1, ..., sourcen must be one-dimensional and all component types must be the same. The source array ranges are specified as 3-tuples: the elements of source1 from its index i1 through its index j1, the elements of source2 from its index i2 through its index j2, etc.

The procedure

```
PROCEDURE DELAY(CONST number_of_ticks: INTEGER)
```

when invoked, causes a delay for at least the number of clock ticks given as its argument.

The procedures

```
PROCEDURE WAIT(VAR s: SIGNAL; p: INTEGER)

PROCEDURE SEND(VAR s: SIGNAL)

PROCEDURE AWAITED RETURNS BOOLEAN (VAR s: SIGNAL)
```

are used for process coordination. WAIT causes the issuing process to wait for another process to send the named signal. The integer parameter, p, determines the queuing priority of several waiting processes. SEND causes one process currently waiting for the named signal to resume execution while the issuing process becomes excluded from the MONITOR. If no process awaits the signal, SEND has no effect. AWAITED returns TRUE if and only if process is waiting for the named signal.

The two functions

FUNCTION AND_LONG RETURNS BOOLEAN (CONST b1, b2: BOOLEAN)

FUNCTION OR_LONG RETURNS BOOLEAN (CONST b1, b2: BOOLEAN)

are "long circuit" forms of the BOOLEAN operators AND and OR—that is, both operands are evaluated.

A number of functions are provided for numeric conversion. Their names and arguments follow. Note that we will use INT, FLOAT, and FIXED to denote contextually appropriate representations of integer, floating point number, and fixed point number, respectively.

FUNCTION FLOOR RETURNS INT (CONST n: numeric)

FUNCTION CEILING RETURNS INT (CONST n: numeric)

FUNCTION NEAR_INTEGER RETURNS INT (CONST n: numeric)

FUNCTION NEAR_FIXED RETURNS FIX (CONST n: numeric; scale: INT)

FUNCTION NEAR_FLOATING RETURNS FLOAT (CONST n: numeric)

FLOOR returns the nearest integer less than its argument, which may have any numeric representation. FLOOR returns the nearest integer greater than its argument. NEAR_INTEGER returns the nearest integer to its argument. If two integers are equidistant from the argument, that with the greatest absolute value is returned. NEAR_FIXED returns the nearest fixed point number to its argument in the specified scale, and also rounds away from zero in case of ambiguity. NEAR_FLOATING returns the nearest fixed point number to its argument in the specified scale, rounding in case of ambiguity so that the least significant bit of the representation is zero. (The last is an accepted rule in numerical analysis.)

The function

FUNCTION FIXED_DIV RETURNS FIX
(CONST numerator, denominator: FIX; result_scale: FIX)

computes the fixed-point quotient of its first two arguments, returning the largest fixed-point number in the specified result scale that is smaller than the true quotient. If the denominator is zero, a ZERO_DENOMINATOR exception is raised.

XII INPUT AND OUTPUT—HIGH LEVEL

The basis of input and output is a file that is passed as a program parameter and is also associated with some input or output device. To facilitate the handling of files, several standard procedures are introduced.

The standard procedures are used to establish association with an input or output device, to prepare a file for reading or writing and to conclude reading or writing of a file, to read or write, to position a file prior to reading or writing, and to ask whether there are more file records to be obtained.

The standard procedures are: OPEN, CLOSE, GET, PUT, and CHECK_EOF. Their operation is as follows.

A. *Declaring and Opening Files*

In this and the following subsections, assume that R is some representation, E is a variable of this representation, and that FR is the representation

```
FILE(R)
```

In order to do file input or output, it is first necessary to declare a file variable. This is just like any other local variable; a sample declaration is

```
VAR F: FR
```

Within the scope of this declaration, the first operation that must be performed on the file is an OPEN. The argument structure of OPEN is

```
PROCEDURE(VAR F: FileRepresentation;
          CONST N: ARRAY (1 DIMENSION OF STANDARD_ALPHABET);
          CONST Mode: OPEN_MODE)
```

where OPEN_MODE is the standard representation

```
REPRESENTATION OPEN_MODE
        = UNORDERED(READ, WRITE, APPEND)
```

The formal parameter N is just a length-unresolved character array and its corresponding actual parameter will usually be a string literal; this parameter gives the name by which the file is known to an external file system (and possibly other externally meaningful information) and has no special semantics within the language.

We allow the file to be opened in one of three modes, specified by the parameter Mode. The READ Mode indicates that the file is one known to the external file system, which is to be read starting at its beginning. The WRITE Mode indicates that a new file is to be entered into the external filing system, associated there with the name N, and filled with data by subsequent PUT operations. The APPEND Mode indicates that subsequent PUT operations will be used to add information to the end of an existing file.

B. *Putting, Getting, and End of File*

Files are viewed as a sequence of elements of some type and representation. If a file is opened for READ, this sequence is read, from its beginning. The first GET reads the first element, the second GET reads the second element, and so forth. If a file is opened for WRITE, a new sequence of elements is to be created: one element for each succeeding PUT. Finally, if a file is opened for APPEND, the succeeding PUTs will add elements to the existing sequence of elements.

The argument structure of GET is

```
PROCEDURE GET(VAR F:FR; OUT E:R)
```

Thus the elements obtained from the file are returned as the values of the output parameter E.

Prior to each GET, programs should check that there is an unread element in the file. This is done with the CHECK_EOF operation, whose argument structure is

```
PROCEDURE CHECK_EOF RETURNS BOOLEAN (VAR F:FR)
```

CHECK_EOF returns TRUE if and only if there are no remaining unread elements in the sequence. If the operation GET is applied to a file with no unread elements, an END_OF_FILE exception is raised. GET and CHECK_EOF are permitted only on files that have been opened for READ.

Similarly, the argument structure of PUT is

```
PROCEDURE PUT(VAR F: FR; CONST E: R)
```

The second argument is the new element to be added to the file. PUT is permitted only on files which have been opened for WRITE or APPEND

C. *Closing Files*

The CLOSE operation advises the external filing system to put the file that was open back into a consistent state and to release any resources it has allocated for the input or output operations. Thus CLOSE breaks the connection between the file variable F and the external file established by a preceding OPEN. The argument structure of CLOSE is:

PROCEDURE CLOSE(VAR F: FR)

Only an open file may be closed.

XIII PROGRAMS AND PROGRAM PARTITIONING FOR COMPILATION

A. *Segments*

A program is a collection of one or more *segments*. Each segment is a module. To form an executable whole, the segments are combined by a *linker*. The order of linking given by the programmer determines the order of execution of the segment bodies. Usually, the last segment linked is a "main" module. It is invoked after others have been initialized by execution of their bodies. It drives the primary computation of the program, making use of the facilities defined by the other segments.

A program composed by linking the segments S1, S2, ..., Sk, in that order, has exactly the same behavior as would the block obtained by concatenating the corresponding module texts between BEGIN and END brackets:

```
BEGIN
  MODULE S1 ... ENDMODULE S1;
  MODULE S2 ... ENDMODULE S2;
    ...
  MODULE Sk ... ENDMODULE Sk;
END
```

B. *Synopses*

To facilitate independent compilation, every segment is written in a form that separates the specification of its external interface from the definitions that implement that specification. The specification part of a segment is called its *synopsis*. The relevant syntactic rules are:

```
ModuleSynopsis
   = MODULE   Id* Imports Exports BlockSynopsis ENDMODULE  Id*
    |MONITOR Id* Imports Exports BlockSynopsis ENDMONITOR Id*

BlockSynopsis
   = {DeclarationSynopsis  ;} [;]

DeclarationSynopsis
   = CONST  {Names : Representation = ScopeEntryConstant ;}
    |VAR {Names : Representation [:= Expression] ;}
    |TYPE {TypeId = Type ;}
    |REPRESENTATION {RepresentationId = [Type] ConcreteType ;}
    |FUNCTION  [EXTENDS |REPLACES] ExtendedId Formals
        [INLINE  Block] ENDFUNCTION  ExtendedId
    |PROCEDURE [EXTENDS |REPLACES] ExtendedId Formals
        [INLINE  Block] ENDPROCEDURE ExtendedId
    |ModuleSynopsis
    |TEMPLATE Id* ( Ids ) ModuleSynopsis ENDTEMPLATE Id*
    |INSTANCE Id* ( Ids )
    |PROCESS Id* Formals
        ENDPROCESS  Id*
```

The synopsis is a skeleton of the module which gives just the information necessary to define its external interface. It contains the import and export declarations for the module, and all the definitions relevant to exported items, omitting only the bodies of routines and the bodies of any nested modules. (However, the bodies of INLINE routines are included.)

In the following sections rules applying to particular segment declarations are given in detail.

C. *Segment Import and Export Declarations*

All import and export declarations for a segment must appear in its synopsis. Everything exported must also be declared in the synopsis. Import declarations can, of course, only refer to items exported by other segments, and they must name the external segments. Thus, the forms

```
IMPORT M;
IMPORT X, Y FROM M
```

are each legal if M names another segment. But IMPORT V is illegal in a synopsis if V is not a segment.

D. *Segment Declarations*

Constants, types, and representations mentioned in the synopsis must either be imported or they must be fully declared in the synopsis.

E. *Segment Routine and Process Declarations*

The declaration of a routine or process in a synopsis consists of the header of the actual routine or process and the body is omitted—e.g.

```
PROCEDURE P RETURNS T1 (VAR F:T2) ENDPROCEDURE P
```

Types or representations mentioned in the header (e.g., T1 and T2) must be fully defined in the synopsis. That is, they must be imported or defined explicitly unless they are standard.

The exception to the rule that bodies are omitted is that, for routines declared INLINE and exported from a segment, the body of the routine appears with its header in the synopsis. As with any exported item, an INLINE routine may not be exported if it depends on entities that are not defined in the synopsis—e.g., if it uses variables not declared in the synopsis.

F. *Segment Declarations of Nested Modules*

A module synopsis appears nested within a synopsis when the corresponding nested module in the full segment exports items that are also exported by the segment or that are used in defining the exports of the segment.

G. *Module Templates and Their Instances in Segments*

If a module template is exported from a segment, or is depended on by an exported item, then the synopsis of that segment contains a template with just the synopsis of the module that is the body of the template. If a template instance defines an exported item, or is depended on by an exported item, then it must appear in full in the synopsis.

H. *An Example*

Consider the segmentation of a compiler written in the language. One module that might reasonably be separated for compilation is the lexical analyzer, the part responsible for breaking the input text stream into atomic units called "lexemes". The synopsis of such a segment could have the form

```
MODULE LexicalAnalyzer
  IMPORT InputFile FROM Parser;
  IMPORT Converters;
  EXPORT Lexeme, IsLiteral, ...;
  EXPORT READONLY CurrentSymbol;
  EXPORT InitLex, NextSymbol;
  MODULE LexemeType
    EXPORT Lexeme, IsLiteral, ...;
    TYPE LexemeKind = UNORDERED (ID, LITERAL, ...);
    TYPE Lexeme =
      RECORD ...
        SELECT Kind: LexemeKind
          CASE ID => ...
        ...
      ENDRECORD;
    FUNCTION IsLiteral RETURNS BOOLEAN (L: Lexeme) INLINE
      RETURN L.Kind = LITERAL
    ENDFUNCTION IsLiteral;
  ENDMODULE LexemeType;
  PROCEDURE InitLex () ENDPROCEDURE InitLex;
  PROCEDURE NextSymbol () ENDPROCEDURE NextSymbol;
ENDMODULE LexicalAnalyzer;
```

(This lexical analyzer has been deliberately oversimplified in the interest of brevity.) This synopsis reveals that the full segment is to be used in conjunction with two others: one called Parser, from which the file variable to be used for input is imported, and one called Converters, from which utility functions and tables for numeric conversion are imported.

The analyzer segment exports the opaque type Lexeme and some functions, such as IsLiteral, that are to be used in the rest of the compiler to access the properties of Lexeme values. (This scheme assures that the implementation of lexemes can be changed without affecting other parts of the compiler.) The type LexemeKind must be included in the synopsis, although it is not exported, because the exported type Lexeme depends on it. The body of the IsLiteral function appears in the synopsis only because it is declared INLINE.

Also exported are a variable—CurrentSymbol—and the procedures InitLex and NextSymbol. InitLex initializes the analyzer. Recall that the scope of a segment is global, so that the initialization statements in its body will be executed only once per instruction of the whole program. Therefore InitLex is exported for use by the caller of the lexical analyzer. CurrentSymbol holds the most recently accepted lexeme, as set by calls to NextSymbol.

I. *Segment Integration*

In order to compile a segment that imports items from other segments, the compiler is given the information that allows it to find synopses of those segments. The compiler uses the synopses of external segments to obtain declarations of imported items, and it records the designator of each synopsis in the compiled object module. Later, when a program is integrated by the linker, or when it is initialized (depending on the implementation), the consistency of the synopses used is verified. If outdated synopses have been used to compile some segments, those segments will be identified for the user so that he can recompile them. Of course, a change in a full segment need not force recompilation of related segments unless the synopsis of the full segment is affected. (See the Design Discussion for a discussion of the pragmatics of segment integration.)

J. *Effects of Segmentation on Project Organization*

The designation of that skeleton of a full segment that is the synopsis of the segment can benefit the management of large programming projects. A synopsis can be distributed to the implementers of related segments and can be used in their compilation before the segment is completed. Some segments may consist of declarations only, and in these cases the synopsis and full segment are identical. For others, the effects of an internal change in the full segment do not affect the synopsis and thus do not force recompilation of related modules. Large programs are often created and maintained as program families with different versions specialized to different resource configurations. The segmentation facilities of the language support this kind of specialization. Machine- or configuration-dependent features are encapsulated in segments whose importation or inclusion in another module serves as an explicit declaration that the module depends on the particular configuration.

XIV LOW-LEVEL AND IMPLEMENTATION-DEPENDENT FACILITIES

In this section, we describe a number of implementation-dependent facilities of the language. These facilities should, where possible, be avoided. They may make programs hard to understand, because they require readers to understand not only the language semantics but also the semantics of a particular implementation, and they are almost certain to make programs nonportable. On the other hand, it is inevitable that programs for embedded systems will have some parts that must be extremely specialized and efficient and are inherently nonportable. Even in the programming of such systems, these implementation-dependent segments should be identified, segregated from the rest of the program, and labeled as implementation dependent. In what follows, we describe how this is done.

In the remainder of this section, we adopt a subsection numbering parallel to the rest of the Language Report. Subsections VI, VII, X, XI, XII, and XIII of this section describe implementation-dependent facilities applicable to the language facilities described in the corresponding sections of the Report. There are no implementation-dependent facilities corresponding to the omitted numbers.

VI. *Types and Representations*

1. *Storage Representations*

It is sometimes necessary to specify the representation of a variable in storage. This can be done by use of the PACKED, AT, and ALIGNMENT qualifiers in a representation.

In many implementations it is possible to increase the speed with which values are manipulated by increasing the amount of storage used in representing these values. In most cases, translators will choose speed rather than storage economy. However, the qualifier PACKED states the programmer's preference for storage economy. If the PACKED qualifier includes its optional manifest constant, then no more storage than the specified value may be used and if a translator requires more storage, a translation error occurs. (The values permitted for the manifest constant are implementation-dependent.) If the PACKED qualifier omits the manifest constant, this directs the translator to favor storage economy to the extent deemed reasonable in the particular implementation. The relevant syntax is:

```
    ConcreteType
†72     = ConcreteType1 [PACKED [ManifestConstant]]
```

In defining representations of record types, it is also possible to specify, as an implementation-dependent offset, the position within the record of each component using the AT option of Name. The relevant syntax is:

```
    ConcreteType1
78      = RECORD
            {(CONST | VAR) {Names : Representation ;}} [;]
            [SELECT Name : Representation
               {CASE {ManifestConstant ,} =>
                  {(CONST | VAR) {Names : Representation ;}} [;]}
             ENDSELECT [;]] ENDRECORD

    Names
57      = {Name ,}

    Name
58      = Id* [AT ManifestConstant]
```

For example, low-level input-output programming on the Decsystem10 (see [11]) makes use of the representation

```
RECORD VAR status AT  0: HALFWORD;
           deviceName AT 36: SIXBIT;
           outBuffer AT 72: ADDRESS;
           inBuffer AT 90: ADDRESS
ENDRECORD
```

where the representation HALFWORD is

```
ARRAY(INTEGER [1..18] OF BOOLEAN) PACKED 18
```

and the representation SIXBIT is

```
ARRAY(INTEGER [1..6] OF SIXBIT_ALPHABET) PACKED 36
```

(The representation ADDRESS is explained below.)

A component of a PACKED object may not be an actual parameter corresponding to a VAR formal parameter.

It is sometimes necessary to have specify the way in which representations are aligned on storage boundaries (e.g., word, byte, double word). Such alignment can be specified for a representation using the syntax:

```
    ConcreteType
+72     = ConcreteType1 [ALIGNMENT ManifestConstant]
```

All values with this representation will begin at a position in storage constrained by the implementation-dependent manifest constant that is specified.

Packing and alignment may both be specified by a representation; the combined syntax is:

```
    ConcreteType
 72     = ConcreteType1 [PACKED [ManifestConstant]] [ALIGNMENT ManifestConstant]
```

2. *Specialized Implementation-Dependent Representations*

There are two specialized implementation-dependent representations: ADDRESS and SEMAPHORE.

The ADDRESS representation is an implementation-defined representation of integers. For example, on the Decsystem10, ADDRESS is

```
INTEGER [0..262143] PACKED 18
```

Each address corresponds to some storage location in the implementation, for example a word or byte of memory or a machine register. Two operations applicable to addresses are ADDR and MAKE_POINTER; they are described below. The implementation-dependent global variable MAIN_STORE_ARRAY with index representation ADDRESS and implementation-specified component type is provided for low-level operations involving addresses. Additional address operations may be defined by a particular implementation.

Semaphores are used to implement asynchronous interactions between processes not conveniently implementable using monitors (e.g., overlapping critical regions). Only three procedures—CLAIM, RELEASE, and CONDITIONAL_CLAIM—are defined for this type and described below. (Assignment, equality, and inequality are not defined.)

VII. *Simple Declarations*

It is possible to specify the exact position in storage of a VAR or a CONST. This is done using the AT option of Name in VAR declarations, CONST declarations, and Formals. The relevant syntax is:

```
†    Declaration
 7       = CONST {Names : Representation = ScopeEntryConstant ;}
 8       | VAR {Names : Representation [:= Expression] ;}

     Names
57       = {Name ,}

     Name
58       = Id* [AT ManifestConstant]

     Formals
87       = [RETURNS Representation]
              ( {(CONST|OUT|VAR) {Names : FormalRepresentation ;}} [;] )
```

The manifest constant is implementation-dependent. If the AT qualifier is used in a VAR declaration, the variable is positioned at the specified location. If the AT qualifier is used in a CONST declaration, the constant is positioned at the specified location. If the AT qualifier is used with a formal parameter, the corresponding actual parameter is passed in the specified location. (Note that this AT qualifier, used at the top-level of a declaration, specifies an absolute location whereas the nested AT described above specifies a relative offset within a record.)

As an example, on a PDP-11, input is done by accessing a storage location. Thus, one can write

VAR ClockStatus AT BASE8 177546: WORD

to declare a variable ClockStatus which must be allocated at octal address 177546. Following this declaration, normal reference to this variable within a program invokes a PDP-11 low-level operation. (This example is adapted from page 44 of [18].)

A device signal is a SIGNAL that can be sent directly by a processor. (Other signals are sent only when a process executes the SEND operation.) A device signal is a signal declared using an AT qualifier to associate it with a machine location known by the processor. Device signals may also be sent by software. For example, on the PDP-11, in conjunction with the variable ClockStatus described above, one can also declare

VAR ClockSignal AT BASE8 100: SIGNAL

Thereafter, in view of the semantics of interrupts of the PDP-11, if ClockStatus is set to arm a process to receive clock interrupts, these will be sent by the processor using the SIGNAL ClockSignal. (The use of device signals is explained and illustrated in the Design Discussion and in the Tutorial and Programming Examples.)

X. *Composite Declarations*

1. *Functions and Procedures*

For some applications it is necessary to treat a typed object directly in terms of its concrete machine representation. For example, to implement an output procedure, one may want to treat a typed object as a stream of bits to be transmitted to an output device. The formal representation SPACE is used for this purpose. Since SPACE is a formal representation, only formal parameters may be represented in this way. When a function or procedure with a SPACE formal parameter is invoked, the corresponding actual parameter may be of any representation. Only two functions may be applied to a SPACE: ADDR and EXTENT. They are described below. If a SPACE formal parameter is to be passed as an actual parameter to an inner routine, the inner formal parameter must be SPACE as well.

2. *Processes*

Three implementation-dependent parameters may be included in a process declaration. These are the *priority* of the process, the maximum number of concurrent *instances* of the process, and the *size* of each instance. The syntax for providing these parameters is:

```
†    Declaration
17       = PROCESS Id* Formals [PRIORITY ScopeEntryConstant]
               [INSTANCES ScopeEntryConstant] [SIZE ManifestConstant]
               Block ENDPROCESS Id*
```

The priority of a process is a scope entry constant (see Section VII). This priority is used by the implementation-provided algorithm for scheduling runnable processes. This parameter does not generally have a preemptive effect on scheduling; to the extent consistent with the particular implementation configuration, scheduling algorithms will service processes with highest priorities first.

No more than the specified number of instances will exist simultaneously. When this maximum number is reached, a subsequent attempt to activate the process will raise a TOO_MANY_PROCESS_INSTANCES exception.

If an activation of a process instance, in the course of its execution, attempts to use more local storage than specified by its size, a TOO_LITTLE_LOCAL_STORAGE exception is raised. (This exception will almost always be suppressed; to handle it properly, an implementation may have to allocate, in addition to the specified size, additional space for each instance to process this exception.)

The interpretation and units of these three parameters are implementation-dependent. The priority, instances, and size used if the parameters are omitted is also implementation-dependent.

3. *Trap Handling*

Trap handlers are routines declared in the outermost scope of a program and used to process traps generated by the hardware, the nucleus, and the compiled code (e.g., range errors). Trap handling routines must be named according to a uniform, implementation-defined, scheme. The parameters of these routines are also defined for each implementation.

A trap handler is invoked, as a procedure, at that point in the user program which provoked the trap. The trap handler can exit by propagating an exception (which is invoked in the context of the user program operation which provoked the trap), or by a RETURN (which resumes execution with the user program operation following that which provoked the trap). The semantics of resuming execution, and of any result returned, is implementation defined. Within the trap handler, any diagnostic examination or analysis of the trap context is implementation dependent. Standard trap handlers are provided for each implementation, and will propagate the standard exceptions of that implementation, which must include those in Appendix C. (Trap handlers for program development emphasize error diagnosis; those for operational systems convert traps into exceptions used for application system recovery. A user may replace a standard trap handler by some other procedure using a procedure declaration with the REPLACE option.)

XI. *Standard Routines*

1. *Semaphore Operations*

Three operations are defined for manipulating semaphores. (Our semaphores are actually binary semaphores.) Their names and arguments are as follows:

```
PROCEDURE CLAIM(VAR s: SEMAPHORE)

PROCEDURE CONDITIONAL_CLAIM RETURNS BOOLEAN (VAR s: SEMAPHORE)

PROCEDURE RELEASE(VAR s: SEMAPHORE)
```

Semaphores are either *free* or *in use*. If a semaphore is in use, the CLAIM operation suspends its calling process until the semaphore is free and then changes its status to "in use". If a semaphore is free, then the CLAIM operation changes its status to "in use".

CONDITIONAL_CLAIM is a value-returning procedure which returns a BOOLEAN, TRUE if the semaphore was free; in this case it is marked as in use by the calling process. Otherwise, CONDITIONAL_CLAIM returns FALSE and the calling process is not suspended.

RELEASE frees its argument semaphore.

A newly declared semaphore is free. The resolution of when or in what order suspended processes claim semaphores when they are freed is implementation-dependent. The simulation of a BEGIN-END statement does not affect the semantics of semaphore operations within the statement.

2. *Miscellaneous Operations*

The procedures

FUNCTION ADDR RETURNS ADDRESS (CONST x: any_repr)

PROCEDURE MAKE_POINTER(OUT p: pointer_repr; CONST a: ADDRESS)

PROCEDURE FREE_POINTER(VAR p: pointer_repr)

FUNCTION EXTENT RETURNS INT (CONST x: any_repr)

are also provided for low-level programming. ADDR returns the address in storage of its argument. MAKE_POINTER sets its OUT parameter p to be a pointer of the specified pointer representation to the value at machine address a. FREE_POINTER sets the pointer p to the NIL pointer of the this type, recovers the storage in use by the anonymous variable to which p pointed, and renders undefined the values of all other pointers that point to the same anonymous variable. EXTENT returns the amount of storage used to represent its argument. The implementation-dependency of these operations arises from the implementation-dependent interpretation of the values they return, how storage is addressed, how the values of variables are represented, and the units in which the extent of a value is expressed.

3. *Processes*

The function:

FUNCTION PROCESS_IDENTITY RETURNS INT ()

is invoked by a process to provide a value for use in the next two procedures.

The procedure:

PROCEDURE CHANGE_PRIORITY (CONST process_identity, priority: INT)

changes the priority of the process determined by the first argument. The new priority of the process is the value of the second argument. The priority of a process is initially set by the PRIORITY clause in its declaration, as described above, and may be modified by CHANGE_PRIORITY.

The procedure:

PROCEDURE RAISE_PROCESS_TERMINATION (CONST process_identity: INT)

raises a PROCESS_TERMINATION exception in the process identified by its first argument.

4. *Numeric Enquiries*

The following functions provide implementation-specific information about the representation of numbers:

```
FUNCTION REAL_PRECISION RETURNS INT (CONST p: PRECISION_DESCRIPTOR)

FUNCTION RADIX RETURNS INT ()

FUNCTION UPPER_EXPONENT RETURNS INT
                                (CONST p: PRECISION_DESCRIPTOR)

FUNCTION LOWER_EXPONENT RETURNS INT ()
                                (CONST p: PRECISION_DESCRIPTOR)
```

where the enumeration representation PRECISION_DESCRIPTOR is

```
UNORDERED(STANDARD_PRECISION, EXTENDED_PRECISION)
```

These functions provide the actual precision, radix, and exponent range with which floating-point variables and expressions are implemented. A particular implementation may provide additional inquiries of this kind and may provide a suitable definition of PRECISION_DESCRIPTOR.

5. *Code Inserts*

Each implementation may provide standard procedures whose calls direct the translator to include machine-code inserts in a program. For example, if PDP10_CODE_INSERT is such a procedure in a PDP10 implementation, then the call

```
PDP10_CODE_INSERT(arg1, ...., argn)
```

will yield, in the object program, a machine-code insert specified by arg1, ..., argn. The form and interpretation of these arguments are implementation-dependent. These arguments must be manifest constants.

XII. *Input and Output*

The facilities described above are sufficient to provide low-level input and output. In particular, machine code inserts, device signals, and the SPACE formal parameter type, used together with the other features of the language, will be used to construct the modules that provide appropriate medium and high-level input-output facilities for each implementation. Such libraries of input and output modules, at various levels of abstraction, will be created and controlled on an implementation or wider basis. (For further discussion and examples in this area, see the Design Discussion and the Tutorial and Programming Examples.)

XIII. *Program Translation*

1. *Configuration Discrimination*

In Section X we presented modules and the requirement that externally defined entities be explicitly imported into modules. The implementation-dependent routines, types, representations described above are of this kind; they will be defined in, and exported from, one or more "configuration modules" in the standard prelude of each implementation. Therefore, under the rules for modules and importation, any program or module of a program using such facilities must import them explicitly. In addition to these facilities, any other implementation-specific configuration constants (describing the machine model, memory size, special hardware options, the operating system if present, and the peripheral equipment) will be provided in the

same manner.

2. *Compiler Directives*

Some of the language features already described are compiler directives, e.g., the PACKED attribute and INLINE directive. Other information can be conveyed to and from the compiler in a specific implementation by means of procedures, variables, and constants exported from the *compiler status module* in the standard prelude of that implementation. This will permit programs, during translation, to examine, and modify as appropriate, selected properties of program components: types, specified and implemented ranges, optimization criteria, etc. Note that this facility has the same form and uniform notation as that just described for referring to configuration constants.

One major category of compiler directives is used to control the status of particular traps. These can advise that no compiled code be generated for detecting and responding to specified traps. Only traps, and not user-defined exceptions, may be inhibited. The semantics of trap inhibition is implementation-dependent.

3. *Conditional Compilation*

The language as described in this Report includes two conditional statements—the IF statement and the SELECT statement. Section VII described a class of manifest constant expressions, which must be evaluated during program translation. To the extent that a translator can determine that a branch of a conditional statement will never be selected, no code will be generated for such a branch. This is not intended as a complete conditional compilation facility; more elaborate mechanisms will be provided by the tools of the Language Environment (see [10], Requirement 13G).

APPENDIX A

RESERVED WORDS

APPENDIX A—RESERVED WORDS

The following are the reserved words of the language. They are, therefore, never identifiers or enumeration constants.

ALIGNMENT ALPHABET AND ARRAY AS ASSERT AT

BASE2 BASE8 BASE10 BASE16 BEGIN

CASE CONST

DIMENSIONS DIV DO

ELSE ELSIF END ENDEXCEPTION ENDFOR ENDFUNCTION ENDIF ENDLOOP
ENDMODULE ENDMONITOR ENDPROCEDURE ENDPROCESS ENDRECORD
ENDSELECT ENDTEMPLATE ENDWHILE EXCEPTION EXIT EXPORT EXTENDS

FILE FOR FUNCTION

GOTO

IF IMPORT IN INLINE INSTANCE INSTANCES

LABEL LOOP

MOD MODULE MONITOR

NIL NOT

OF OPAQUE OR ORDERED OUT

PACKED POINTER PRECISION PRIORITY PROCEDURE PROCESS PROPAGATE_EXCEPTION

READONLY RECORD REPLACES REPRESENTATION RETURN RETURNS

SCALE SELECT SET SIMULATED_TIME SIZE SPACE

TEMPLATE TERMINATE_PROCESS THEN TRANSPARENT TYPE

UNORDERED

VAR

WHILE

XOR

APPENDIX B

GRAMMAR

APPENDIX B—GRAMMAR

The syntax is presented with a notation that makes special use of the enlarged characters =, {, }, [,], and |; other characters are used only in spelling terminals and nonterminals. Terminals are spelled with upper case letters and punctuation marks. Nonterminals are spelled with mixed case letters. A star-terminal (see Section III and Section IV) is spelled with mixed case letters and a final asterisk.

```
      Program
1        = Module

      Module
2        = MODULE Id* Imports Exports Block ENDMODULE Id*
3        | MONITOR Id* Imports Exports Block ENDMONITOR Id*

      Imports
4        = {IMPORT {ExtendedId [FROM Id*] [AS ExtendedId] ,} ;}

      Exports
5        = {EXPORT [OPAQUE|TRANSPARENT|READONLY|VAR]
             {ExtendedId [FROM Id*] [AS Id*] ,} ;}

      Block
6        = {Declaration ;} [;] [ExceptionHandler [;]] {[Id* :] Statement ;} [;]
```

```
      Declaration
7        = CONST {Names : Representation = ScopeEntryConstant ;}
8        | VAR {Names : Representation [:= Expression] ;}
9        | LABEL Ids
10       | TYPE {TypeId = Type ;}
11       | REPRESENTATION {RepresentationId = [Type] ConcreteType ;}
12       | FUNCTION [EXTENDS | REPLACES] ExtendedId Formals [INLINE]
             Block ENDFUNCTION ExtendedId
13       | PROCEDURE [EXTENDS | REPLACES] ExtendedId Formals [INLINE]
             Block ENDPROCEDURE ExtendedId
14       | Module
15       | TEMPLATE Id* ( Ids ) Module ENDTEMPLATE Id*
16       | INSTANCE Id* ( Ids )
17       | PROCESS Id* Formals [PRIORITY ScopeEntryConstant]
             [INSTANCES ScopeEntryConstant] [SIZE ManifestConstant]
             Block ENDPROCESS Id*

      ExceptionHandler
18       = EXCEPTION {CASE Ids => UnlabeledStatements}
             [ELSE UnlabeledStatements] ENDEXCEPTION

      Statement
19       = Designator := Expression
20       | Id* ( {Expression ,} )
21       | ASSERT Expression
22       | GOTO Id*
23       | EXIT Id*
24       | RETURN [Expression]
25       | TERMINATE_PROCESS
26       | PROPAGATE_EXCEPTION [Id*]
27       | IF Expression THEN UnlabeledStatements
             {ELSIF Expression THEN UnlabeledStatements}
             [ELSE UnlabeledStatements] ENDIF
28       | SELECT Expression
             {CASE {ManifestConstant ,} => UnlabeledStatements}
             [ELSE UnlabeledStatements] ENDSELECT
29       | WHILE Expression DO UnlabeledStatements ENDWHILE
30       | FOR Name : Representation DO UnlabeledStatements ENDFOR
31       | LOOP Id* UnlabeledStatements ENDLOOP Id*
32       | [SIMULATED_TIME] BEGIN Block END

      UnlabeledStatements
33       = {Statement ;} [;]
```

```
      Expression
34       = SimpleExpression [RelationalOperator SimpleExpression]

      SimpleExpression
35       = [-] Term
36       | Term {AssociativeAddingOperator Term}
37       | Term NonAssociativeAddingOperator Term

      Term
38       = Factor {AssociativeMultiplyingOperator Factor}
39       | Factor NonassociativeMultiplyingOperator Factor

      Factor
40       = {NOT} (( Expression ) | Designator | [Id*] Literal)

      Designator
41       = Id*
42       | Id* ( {Expression ,} )
43       | Designator (. Id* | [ {Expression ,} ] | ↑)

      Literal
44       = FloatingLiteral* | FixedLiteral* | IntegerLiteral* |
             EnumerationLiteral* | CharacterLiteral* | StringLiteral* |
             NIL

      ManifestConstant
45       = Expression

      ScopeEntryConstant
46       = Expression

      RelationalOperator
47       = IN | ≠ | = | < | <= | > | >= | FreeRelationalOperator

      FreeRelationalOperator
48       = ≠≠ | ≠≠≠ | == | === | << | <<= | <<< | <<<= |
             >> | >>= | >>> | >>>=

      AssociativeMultiplyingOperator
49       = * | AND

      NonassociativeMultiplyingOperator
50       = / | DIV | MOD | FreeMultiplyingOperator
```

```
      FreeMultiplyingOperator
51       = ** | *** | // | ///

      AssociativeAddingOperator
52       = + | OR

      NonassociativeAddingOperator
53       = - | XOR | FreeAddingOperator

      FreeAddingOperator
54       = -- | --- | ++ | +++

      Ids
55       = {Id* ,}

      ExtendedId
56       = Id* | NOT | ↑ | := | RelationalOperator |
               AssociativeMultiplyingOperator |
               NonassociativeMultiplyingOperator |
               AssociativeAddingOperator | NonassociativeAddingOperator

      Names
57       = {Name ,}

      Name
58       = Id* [AT ManifestConstant]

      TypeId
59       = Id*

      RepresentationId
60       = Id*

      Representation
61       = [Type] ConcreteType
62       | RepresentationId
```

```
      Type
63       = TypeId
64       | UNORDERED ( Ids )
65       | ORDERED ( Ids )
66       | ALPHABET ( {CharacterLiteral* ,} )
67       | ARRAY ( {Representation ,} OF Type )
68       | RECORD
               {(CONST|VAR) {Ids : Type ;}} [;]
               [SELECT Id* : Type
                  {CASE {ManifestConstant ,} => {(CONST|VAR) {Ids : Type ;}} [;]}
                ENDSELECT [;]] ENDRECORD
69       | FILE ( Type )
70       | POINTER ( Representation )
71       | SET ( Representation )

      ConcreteType
72       = ConcreteType1 [PACKED [ManifestConstant]] [ALIGNMENT ManifestConstant]

      ConcreteType1
73       = TypeId [SimpleQualifier]
74       | UNORDERED ( Ids ) [SimpleQualifier]
75       | ORDERED ( Ids ) [SimpleQualifier]
76       | ALPHABET ( {CharacterLiteral* ,} ) [SimpleQualifier]
77       | ARRAY ( {Representation ,} OF Representation )
78       | RECORD
               {(CONST|VAR) {Names : Representation ;}} [;]
               [SELECT Name : Representation
                  {CASE {ManifestConstant ,} =>
                     {(CONST|VAR) {Names : Representation ;}} [;]}
                ENDSELECT [;]] ENDRECORD
79       | FILE ( Representation )
80       | POINTER ( Representation )
81       | SET ( Representation )

      SimpleQualifier
82       = [ ScopeEntryConstant .. ScopeEntryConstant ]
83       | [ ScopeEntryConstant .. ScopeEntryConstant ] SCALE ManifestConstant
84       | [ ScopeEntryConstant .. ScopeEntryConstant ] PRECISION ManifestConstant

      FormalRepresentation
85       = ARRAY ( ManifestConstant DIMENSIONS OF Representation )
86       | (Representation|SPACE)
```

Formals

87 = [RETURNS Representation]
({(CONST | OUT | VAR) {Names : FormalRepresentation ;}} [;])

APPENDIX C

STANDARD ERRORS, TRAPS, AND EXCEPTIONS

APPENDIX C—STANDARD ERRORS, TRAPS, AND EXCEPTIONS

The word *error* means translation error; a program in the language will have no errors. The words *trap* and *exception* are defined and discussed in Section X; they refer to conditions that may not be detected until a program is executed. (Some translators will optimize by detecting some traps and exceptions during translation.)

1. *Errors*

The precise list of errors will be gathered during the implementation of the prototype translator for this language during Phase 2. The following is a subset of the errors detected by the CDC6000 Pascal Compiler; these are a fair sample of the errors that a compiler for our language must detect.

Actual parameter must be a variable.
Const parameter expected.
Error in base set.
Error in factor.
Error in manifest constant.
Error in parameter list.
Error in type.
Error in variable.
Expression is not of set type.
Identifier declared twice.
Identifier not declared.
Illegal expression type.
Illegal operand(s) type.
Illegal parameter substitution.
Illegal symbol.
Index type must be scalar.
Index type not compatible with declaration.
Low bound exceeds high bound.
Manifest constant expected.
Missing corresponding variant declaration.
Missing result type in function declaration.
Multidefined case label.
Multideclared label.
Multidefined label.
No case provided for this value.

No such field in this record.
Number of parameters does not agree with declaration.
Only equality tests are allowed.
Operand expected.
Operand type conflict.
Operand type must be Boolean.
Operator expected.
Result type of parameter function does not agree with declaration.
Set element type must be scalar.
Set element types incompatible.
String literal must not exceed source line.
Subrange bounds must be scalar.
Type conflict.
Undeclared label.
Undefined label.
Value to be assigned is out of range.

2. *Traps and Exceptions*

The precise list of traps and exceptions will be gathered during the implementation of the prototype translator for this language during Phase 2. The following is a fair sample of the traps and exceptions for which compiled code and runtime systems must provide:

Too many instances of a process.
Too little space for process storage allocation.
Process exceeds storage allocation.
Invalid pointer value.
Fixed-point value not representable.
False assertion.
Division by zero.
Index expression is out of bounds.
Value out of range of variable's representation.
Scope entry range beyond implementation's representation.
Overflow.
Floating-point overflow.
Floating-point underflow.
Access to uninitialized variable.
Deadlock.
Storage parity error.
Processor error.
Power failure.
Invalid operation.
Addressing protection violation.

PART THREE

TUTORIAL AND PROGRAMMING EXAMPLES

CONTENTS

I INTRODUCTION

This document is a rapid informal introduction to a subset of our language. The language itself is defined by the accompanying Language Report. Here we wish *to describe and explain* rather than *to define*. (In case of conflict, the Language Report is to be taken as definitive.) We assume that the reader is versed in writing programs in machine-independent languages, such as Fortran, Algol 60, or PL/I.

Section II gives a *rapid* introduction to a subset of the language by means of a progression of small examples supported by nontechnical explanations. These are intended particularly for those who wish, in a short time, to become generally acquainted with the language.

Whereas Section II is primer-like, with simple examples illustrating a few features at a time, Section III demonstrates the descriptive adequacy of the language in areas such as real-time scheduling, data encapsulation, and machine-level data descriptions.

II PRIMER EXAMPLES

A. *How Many Female Employees Have Received Flu Shots?*

First we describe a file of personal medical records for an organization with N employees. Each medical record specifies a person's name, sex, birthday, and flu inoculation date. Our object is to discover how many female employees aged 60 years or older have had flu shots during the last two years, by processing a collection of such medical records. The flu inoculation record for a typical employee could be:

```
101  REPRESENTATION FluShotRecord =
102    RECORD
103      CONST LastName: Name;
104      CONST FirstName: Name;
105      VAR BirthDay: Date;
106      VAR Sex: UNORDERED(Male, Female);
107      VAR FluShotDate: Date
108    ENDRECORD
```

This text describes the format of a record with five components: LastName, FirstName, BirthDay, Sex, and FluShotDate. The line numbers in the left-hand margin are not part of our language—they are given in this section in all the examples of program text to facilitate subsequent explanations.

Actually, the description of the representation FluShotRecord is incomplete. We need to specify more information about the exact nature of the components before we know exactly what a FluShotRecord is composed of. For example, the BirthDay and FluShotDate components are both Dates. A Date is defined as follows:

```
201  REPRESENTATION Date =
202    RECORD
203      VAR Month: Months;
204      VAR Day: INTEGER [1..31];
205      VAR Year: INTEGER [1850..2050];
206    ENDRECORD
```

The Month component of a Date is an abbreviated name for a month, as specified in the following definition:

```
301  REPRESENTATION Months =
302    ORDERED (Jan, Feb, Mar, Apr, May, Jun, Jul,
                Aug, Sept, Oct, Nov, Dec)
```

The abbreviated names of the months form an *ordered enumeration*. The names of the months can be used to compose representations of Dates. Two such Dates are constructed and assigned to the variables D1 and D2 in the following example:

```
401  VAR D1, D2: Date;
402  D1 := Date(Mar, 4, 1977);
403  D2 := Date(Jul, 17, 1977);
```

Line 401 declares two Date variables D1 and D2 capable of holding values that are Date records. Lines 402 and 403 construct and assign records to D1 and D2 for the two respective dates Mar 4, 1977 and Jul 17, 1977. If we were to need to compare two such dates to see if one was later than another, and if we were first to discover that the years of the two dates were identical, then we would need to determine whether the month of one date was later than that of the other. Since the representation of Months defined in lines 301-302 is an *ordered enumeration*, it is possible to compare any two months, using a relational expression such as *Jan < Jun*. In this case, it is true that Jan is less than Jun, since Jan occurs to the left of Jun in the definition of Months.

Having constructed two Dates D1 and D2, we can access the values of the components. For instance, the Year components of these dates are accessed by the selection expressions D1.Year and D2.Year. Thus, the comparison *D1.Year = D2.Year* has the value TRUE since dates D1 and D2 were constructed with the same Year component. Similarly, the expression *D1.Month < D2.Month* has the value TRUE since the Month component of D1 is Mar and the Month component of D2 is Jul, and since Mar comes before Jul in the definition of Months.

The integer components of Dates are restricted to certain relevant ranges of integer values. In particular, the Day component of a Date is declared to have the representation expression INTEGER [1..31]. This restricts the integer value for the Day of a Date to be in the range $1 \leqslant i \leqslant 31$. Similarly, the Year component of a Date is defined by the representation expression INTEGER [1850..2050]. This restricts the integer value for the Year of a Date to be in the range $1850 \leqslant i \leqslant 2050$. At this early point in the discussion, it is worthwhile noticing the distinction between a *type* and a *representation*. The word INTEGER signifies a *type*. The *type* INTEGER stands for the mathematical concept of an integer—an unbounded set of objects with defined operations such as addition and multiplication. However, in any real computer, finiteness is everywhere—in the precision with which we can define our data and in the length of time we can spend computing results. Any actual computer uses a finite range of integer representations to model mathematical computations with integers. The concept of representation can express the bounds on integer representations usable in a particular finite computer and can provide compilers opportunities for efficient representations of small integer ranges.

This is accomplished with a *range qualifier* such as [1..31] or [1850..2050]. There are other kinds of qualifiers that specify information useful for providing other computer representations of types. We sometimes use the phrase *abstract type* to emphasize the idea of a type divorced from any representational considerations.

Let us now return to consider the other components of the FluShotRecord. On line 106, we see the declaration of a component by the form *CONST Sex: UNORDERED(Male, Female)*. The representation *UNORDERED(Male, Female)* is an *unordered enumeration*. This is similar to an ordered enumeration, such as Months, except that the only comparison operators that apply to the values of an unordered enumeration are *equal* (=) and *not equal* (≠), whereas the values in an ordered enumeration may, in addition, be compared with the operators *greater than* (>), *greater than or equal* (≥), *less than* (<), and *less than or equal* (≤). Thus, the Sex component of a FluShotRecord must be one of the two values Male or Female.

Finally, the first two components of a FluShotRecord are the LastName and FirstName of a person. These components are strings of characters specified by the following representation definition:

```
501  REPRESENTATION Name =
502      ARRAY (INTEGER [1..5] OF ASCII ['A'..'Z'])
```

This specifies that the Name components of a FluShotRecord each consist of five upper-case letters. The range qualifier [1..5] specifies that these upper-case ASCII characters form an ARRAY with indices in the range $1 \leq i \leq 5$. The expression ASCII ['A'..'Z'] designates a *range* of the ASCII alphabet consisting of the letters 'A' through 'Z'.

We can now show how to construct an actual FluShotRecord:

```
601  VAR R1, R2: FluShotRecord;
602  R1 := FluShotRecord("SMITH", "MABEL", Date(Apr,10,1916),
                         Female, Date(Aug,22,1977));
603  R2 := FluShotRecord("DAVIS", "KELLY", Date(Sept,4,1952),
                         Male, Date(Jul,16,1972))
```

Line 601 declares R1 and R2 to be variables capable of holding FluShotRecords. Lines 602 and 603 construct and assign two FluShotRecords as the values of R1 and R2 respectively. The name of the record representation FluShotRecord is applied to a list of arguments that consist of the proper types of components for the record. In a FluShotRecord, the first two components must be strings of five upper-case ASCII characters, such as "SMITH" and "MABEL". The third and last components must be dates. These are specified by *record denotations* such as Date(Apr,10,1916) and Date(Aug,22,1977). The fourth component must be either Female or Male.

Returning to lines 101-108, we observe that some of the components of a FluShotRecord are designated as CONST and some are designated as VAR. The CONST components are the so-called *constant* components. These are components that may be altered during the lifetime of the record only if the entire record is replaced by a new record value. The values of these CONST components are specified in a denotation such as line 603. By contrast, the VAR components of a record are permitted to vary. Such *variable* components may be individually changed by assignment. For instance, the assignments.

```
701  R1.BirthDay.Year := 1917;
702  R2.Sex := Female;
```

change the Year of the BirthDay component of record R1 to be 1917 instead of 1916, and change the Sex component of record R2 to be Female instead of Male, in the FluShotRecords constructed in lines 601-603.

Now suppose we have a sequence of *N* employee FluShotRecords. We wish to consider these records in turn and count the number of records describing Females aged 60 or older who have had flu shots during the last two years. For convenience, let the date on which this computation is performed be Jan 1, 1978. The following program accomplishes the intended result.

```
801  VAR a: ARRAY (INTEGER [1..N] OF FluShotRecord);
802       Count: INTEGER [0..N];
803  Count := 0 (* Initialize Count to zero *);
804  FOR i: INTEGER [1..N]
805    DO IF (a[i].Sex = Female)
806          AND (a[i].BirthDay.Year < 1918)
807          AND (a[i].FluShotDate.Year >= 1976)
808        THEN Count := Count +1 ENDIF;
809  ENDFOR
```

Line 801 declares *a* to be a variable holding an array of *N* FluShotRecords, and line 802 declares an integer variable Count. Line 803 initializes the variable Count to 0, and demonstrates the format for comments in our language. The comment *(* Initialize Count to zero *)* does not affect the meaning of the program. Comments must open with the punctuation mark *(**, close with the mark **)*, and must be given on a single line. They are permitted in the language anywhere a space is permitted. Multiline comments must be composed of a sequence of single-line comments, each surrounded by the comment brackets.

Lines 804 through 809 describe one of the three kinds of *repetitions* available in the language. In this case, the repetition is called a *FOR-repetition*, and its scope is given by the reserved words FOR-ENDFOR.

In this FOR-repetition, the controlled constant *i* is treated as a local constant whose lifetime is the scope between FOR and ENDFOR. This constant i takes on integer values 1, 2, ..., N in succession, as specified by the expression INTEGER [1..N] on line 804, and for each such value in sequence the *body* of the FOR-repetition is executed.

In this example, the body consists of a single *IF-statement* given on lines 805-808. This IF-statement is given between the words IF and ENDIF. For each *i*, it checks to see whether the FluShotRecord a[i] describes a Female, 60 years or older, whose FluShotDate is within the past two years (assuming, of course, that the values in the record are accurate). The value of Count is incremented by one for each such record.

Many of the syntactic units in our language are bounded by pairs of reserved words of the form *X* and END*X*—for instance, MODULE and ENDMODULE, LOOP and ENDLOOP, PROCEDURE and ENDPROCEDURE, and so on. (BEGIN and END are an exception to this general form.)

B. *Distance Between Points in the Plane*

Imagine that we are working with a set of data that gives coordinates of points in the two-dimensional plane either as Cartesian coordinates (x, y) or as polar coordinates (ρ, θ). Given two such points—both Cartesian, both Polar, or one of each kind—we are asked to compute the distance between them.

Since we need to deal with representations of real numbers in this example, let us begin by specifying a class of floating-point numbers:

```
901  REPRESENTATION FLOAT = FLOATING [-1.0E3..1.0E3] PRECISION 7
```

FLOATING is a type—the set of mathematical real numbers of unbounded precision. The qualifier *[-1.0E3..1.0E3] PRECISION 7* qualifies the type FLOATING by prescribing a representation. The range *[-1.0E3..1.0E3]* specifies the numbers from -1000.0 to +1000.0. Note that *1.0E3* is standard scientific notation, and means "1.0 times 10 to the power 3." The qualifier *PRECISION 7* specifies that seven (7) decimal digits of precision are to be used in the representation of the numbers between -1000 and +1000. Thus, the representation given of line 901 specifies the values -1000.000, -999.9999, -999.9998, ..., 0.000000, ..., 999.9999, 1000.000 (in the closest machinable approximation).

Let us now define what we call a *variant record*—a record whose components can differ in number and representation, depending on the value of a *case selector* field:

```
1001  REPRESENTATION Coordinate =
1002    RECORD SELECT kind: UNORDERED(Cartesian, Polar)
1003             CASE Cartesian => VAR x, y: FLOAT;
1004             CASE Polar => VAR rho: FLOAT; theta: Angle;
1005    ENDRECORD
```

Lines 1001-1005 declare Coordinate as a record representation with two alternate formats. The Cartesian format has two named components, an *x* component and a *y* component, both representations of real numbers. The Polar format has two named components, a rho component that is a representation of a real number and a theta component that is an Angle. We might imagine an Angle to be specified as, say, a seven-digit floating-point number with values in the range $-\pi$ to $+\pi$. Thus, an Angle could be defined by the following representation definition:

```
1101  CONST Pi: FLOATING [0.0..4.0] PRECISION 7 = 3.14159265;
1102  REPRESENTATION Angle = FLOATING [-Pi..Pi] PRECISION 7;
```

Line 1101 defines the constant Pi as the value 3.14159265 represented to seven decimal digits of precision, and line 1102 defines Angles to be representations of real numbers with seven decimal digits of precision lying in the range [-Pi..Pi].

To denote values of variant records, we must specify the *case selector* associated with the variant as a constant argument of the variant record denotation. For instance, we next construct two Coordinate records, one Cartesian and one Polar.

```
1201  VAR C, P: Coordinate;
1202  C := Coordinate(Cartesian, 1.2, -3.4);
1203  P := Coordinate(Polar, 256.12, -1.772);
```

When writing programs that operate on variants of these Coordinate records, we can set up case analyses based on the *case selector* fields. This is accomplished by means of the *SELECT statement* in our language. For instance, we now give a portion of a program to compute the distance between two points, based on the the diagram given as Figure 1.

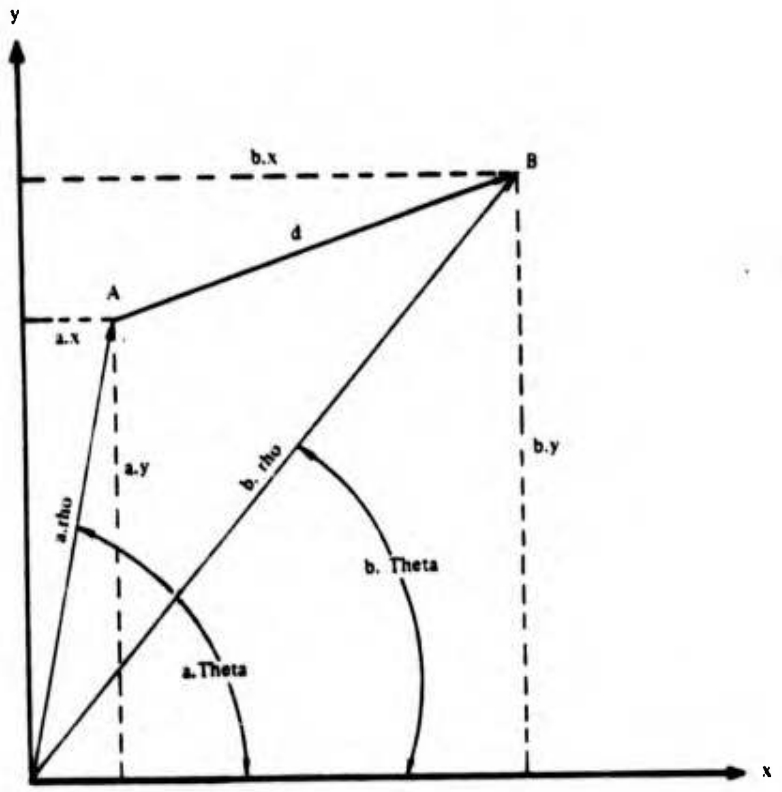

Figure 1 GEOMETRY OF CARTESIAN-POLAR CONVERSION

```
1301  MODULE ComputeDistance;
1302    IMPORT Sqrt, Sin, Cos, Coordinate, FLOAT;
1303    EXPORT Distance;

1304    FUNCTION Square RETURNS FLOAT (CONST x: FLOAT)
1305      RETURN x*x;
1306    ENDFUNCTION Square;

1307    PROCEDURE Rectify (CONST c: Coordinate, OUT x, y: FLOAT)
1308      SELECT c.kind
1309        CASE Cartesian => x := c.x; y := c.y;
1310        CASE Polar => x := c.rho*Cos(c.theta);
1311                      y := c.rho*Sin(c.theta);
1312      ENDSELECT;
1313      RETURN;
1314    ENDPROCEDURE Rectify;

1315    PROCEDURE Distance RETURNS FLOAT (CONST a, b: Coordinate)
1316      VAR ax, ay, bx, by: FLOAT;
1317      Rectify(a, ax, ay);
1318      Rectify(b, bx, by);
1319      RETURN Sqrt(Square(ax-bx)+Square(ay-by));
1320    ENDPROCEDURE Distance;

1321  ENDMODULE ComputeDistance
```

This is an instance of a *module*. The use of names across module boundaries must be explicit: names may be *imported* into the module and *exported* from the module. In this module, the functions Sqrt, Sin, and Cos and the representations Coordinate and FLOAT are imported by line 1302. (The names of standard language operations such as + and * need not be explicitly imported.) The ComputeDistance module exports the *value-returning procedure* Distance in line 1303. All the other names declared inside the module are unknown outside the module. Modules can be used to define *encapsulations* whose details of implementation are maintained privately, and whose users see only the features provided by the encapsulation. In this case, Rectify and Square are viewed as details of implementation and are not visible outside the ComputeDistance module. If they were changed, or even deleted, the user of the single exported routine Distance would be unaffected (so long as Distance continued to somehow compute the true distance between its arguments). Lines 1304-1306 declare the *function* Square. Line 1304 is called the *header* of the function declaration and states that Square *returns* a result with representation FLOAT and takes a single argument *x* that is a FLOAT. Since the argument is specified as CONST, it may not be modified by the body of the function—indeed, functions may have only constant arguments. Functions may not have *side-effects*—e.g., they may assign only to local variables. The body of this function, line 1305, consists of a single *return statement*, which says that the value x*x is the desired result.

Next, we wish to compute Cartesian coordinates of a Coordinate. This is done with the *procedure* Rectify, declared on lines 1307-1314. Rectify takes a Coordinate argument *c*, which is CONST. It also takes two OUT arguments *x* and *y*. Whereas CONST arguments are used to pass data from the caller of a routine to the body of the routine, OUT arguments are used by the body of a routine to pass data back to the caller. The body of Rectify, lines 1308-1313, consists of two statements. The first, lines 1308-1312, is a SELECT statement, which discriminates in the *case* of the variant record *c*. If *c* is Cartesian, then line 1309 sets the two OUT parameters appropriately. Lines 1310-1311 set the OUT parameters appropriately in the Polar case. In either case, the second statement of the body, line 1313, completes the execution of a

call of Rectify.

Using Square and Rectify, we can declare the *value-returning procedure* Distance. Whereas a function computes has no side effects, a procedure may have side effects. A value-returning procedure, moreover, is a procedure used syntactically to compute a value. Thus the user of the module ComputeDistance can write expressions such as Distance(a,b)+Distance(c,d). The body of Distance declares four local FLOAT variables (line 1316) and then calls Rectify twice (lines 1317 and 1318) to set these variables to the Cartesian components of the Coordinates *a* and *b*. Finally, the body of Distance uses a return statement (line 1319) to designate the value computed as the distance—the square root of the sum of the squares of the differences between the corresponding Cartesian components.

C. *Sets and the Sieve of Eratosthenes*

Suppose we are asked to compute the prime numbers between 2 and 10000. An ancient method for doing this is the *Sieve of Eratosthenes*. Let us illustrate this method with a small sieve first. We start with two sets—Sieve and Primes. Sieve contains all the integers from 2 to 15, and Primes is initially empty:

Sieve = {2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}
Primes = { }

We take the smallest number in Sieve, 2, put it in Primes, and remove all multiples of 2 from Sieve. This yields:

Sieve = {3, 5, 7, 9, 11, 13, 15}
Primes = {2}

Again, we take the smallest number in Sieve (which is now 3), put it in Primes, and strike out its multiples from Sieve. This yields:

Sieve = {5, 7, 11, 13}
Primes = {2, 3}

Continuing in this fashion, we repeatedly pick the smallest number in Sieve, transfer it to Primes, and strike out all its multiples in Sieve. When Sieve is empty, we are done and Primes contains all the primes between 2 and 15:

Sieve = { }
Primes = {2, 3, 5, 7, 11, 13}

Now we write a program to do this for the primes up to an integer *N*:

```
1401  MODULE Eratosthenes
1402    IMPORT N;
1403    EXPORT TRANSPARENT IntegerSet;
1404    EXPORT READONLY Primes;
1405    REPRESENTATION IntegerSet = SET(INTEGER [2..N]);
1406    VAR Primes: IntegerSet := IntegerSet();
1407    VAR Sieve: IntegerSet := -IntegerSet();

1408    FOR j: INTEGER [2..N] DO
1409        IF j IN Sieve
1410          THEN FOR jMultiple: INTEGER [1..N DIV j] DO
1411                    Sieve := Sieve-IntegerSet(j*jMultiple) ENDFOR;
1412               Primes := Primes+IntegerSet(j);
1413        ENDIF;
1414    ENDFOR;

1415  ENDMODULE Eratosthenes
```

This module implements the Sieve using *set representations*. Line 1405 defines an IntegerSet representation for subsets of the integers 2 through *N*. (Many compilers will choose to implement this as a bit vector N-1 bits long.) To denote a specific set value, the identifier IntegerSet, followed by a list of set members, is used. For instance, IntegerSet(6, 13) is the subset of the integers consisting of the two integers 6 and 13. IntegerSet() denotes the empty set of integers, since it has an empty list of arguments.

Line 1406 declares the variable Primes as an IntegerSet and initializes it to the empty IntegerSet. Line 1407 declares the variable Sieve as an IntegerSet and initializes it to the set of all integers 2 to N, using the set complement operator -.

The operation + on two sets denotes *set union*, and the operation - on two sets denotes *set difference*. Set union is used in line 1412 and set difference is used in line 1411. The expression *x IN S* is a test of set membership. If x is an element of the set S, the value of *x IN S* is TRUE and otherwise it is FALSE. Thus, the expression *j IN Sieve* on line 1409 is TRUE exactly when j is still in the set Sieve.

The program on lines 1408-1414 consists of two nested *FOR repetitions*. The outer FOR counts through the integers 2 to *N*. On each repetition, if the current integer is not in Sieve, we do nothing. If it is in Sieve, we use the inner repetition to delete all its multiples from Sieve, and then add it to Primes.

D. *More on Sets*

Let us look briefly at another kind of set representation. As illustrated in the Sieve example, we can represent sets of integers. In addition, our language permits the formation of sets of enumerations and alphabets. The representation of elements specified in a definition of a set representation is called the *base representation* of the set. Permissible base representations for sets are: (1) a range of integers and (2) the *scalar representations*. The *scalar representations* consist of: (a) *ordered enumerations*, as in the example Months given above (lines 301-302), (b) *unordered enumerations*, like MissileStates below, and (c) *alphabets*, as in *RomanNumChar = ALPHABET('M', 'D', 'C', 'L', 'X', 'V', 'I')*. Consider the following:

```
1501 REPRESENTATION MissileStates =
1502    UNORDERED (Inoperative, Damaged, Armed,
1503                  Fired, OnTheRail, GuidanceUpdated,
1504                  TrackingIndependently);
1505 REPRESENTATION MissileStatus = SET(MissileStates);
1506 VAR m : ARRAY(INTEGER [1..6] OF MissileStatus);
1507 m[1] := MissileStatus(OnTheRail, Armed);
1508 m[2] := MissileStatus(Fired);
1509 m[3] := MissileStatus(Damaged);
1510 m[4] := MissileStatus(Inoperative);
```

The use of set representations of unordered enumerations can lead both to efficient storage utilization—if bit vectors are chosen by the compiler to represent the sets—and also to greater clarity in programming.

E. *More on Numbers and Identifiers*

Our language permits a considerable variety of number representations beyond that seen in the examples so far in this tutorial. For instance, it is possible to give integer literals in several bases beside the decimal base assumed so far. Thus, BASE2 011011100 is a binary integer literal, BASE8 737705 is an octal integer literal, and BASE16 6A4B02 is a hexadecimal integer literal. Also, BASE10 6950 is a synonym for 6950. Binary, octal, and hexadecimal integer literals are useful when dealing with machine-level representations on some computers, and the language permits their use. It is recommended that machine-dependent portions of programs be carefully packaged in modules that do not export a machine-dependent interface.

Fixed-point numbers are one of the number representations definable in the language. For example:

```
1601 REPRESENTATION DOLLARS_AND_CENTS =
1602      FIXED [0.0P..10P5] SCALE 0.01P;
```

declares a representation DOLLARS_AND_CENTS that describes the set of fixed-point numbers 0.0P, 0.01P, 0.02P, ..., 9999.99P, 100000.00P—the range of amounts from 0 dollars to 100000 dollars graduated in steps of pennies. The word FIXED designates a *type*, which must be qualified by a *range* and a *scale* to obtain a representation. The range [0.0P..10P5] given on line 1602 has, as a lower limit, the fixed-point literal 0.0P, and, as an upper limit, the fixed-point literal 10P5, which stands for 100000 (5 being the power to which 10 is raised to yield 100000). The SCALE on 1602 is 0.01P, and this scale specifies the step size between adjacent fixed point numbers in the range [0.0P..10P5]. Fixed literals may also use nondecimal number bases, as in *BASE16 -A67.9BP-2* and *BASE8 +73.7701P-128*. Fixed literals always require the presence of the letter P and require at least one digit to the left of the optional decimal point (or radix point).

Floating literals must always have both a decimal point and at least one digit to the left of the decimal point. Any number of digits may follow the required decimal point, and an optional exponent may follow the mantissa. Thus, 0.5 and 12.E-2 are well-formed floating literals, but .77 and 12E-2 are not well-formed.

Identifiers must begin with a letter. The initial letter may be followed by a sequence of letters, digits, and the special break character '_'. Thus, T, T2G1, LAST_ONE_LEFT, and F_3 are all valid identifiers.

III ADVANCED EXAMPLES

This section consists of a collection of examples of our language as it might be employed in real applications, showing both machine-dependent and machine-independent programming tasks.

We begin with a module that provides a standard set of representations for numbers. Note that we assume the existence of certain predefined constants that are machine dependent. In particular, for this module, the constants SMALLESTINTEGER, LARGESTINTEGER, SMALLESTREAL, LARGESTREAL, and SINGLEPRECISION are imported from the *standard implementation prelude.* This module is imported by several of the subsequent examples:

```
MODULE MyStandardConventions
  IMPORT StandardImplementationPrelude;
  EXPORT TRANSPARENT INT,FIX,FLOAT;
  REPRESENTATION
    INT = INTEGER [SMALLESTINTEGER..LARGESTINTEGER];
    FIX = FIXED [-1P..1P] SCALE 0.000001P;
    FLOAT = FLOATING [SMALLESTREAL..LARGESTREAL]
                     PRECISION SINGLEPRECISION;
ENDMODULE MyStandardConventions;
```

A. *Real Time*

The following is a rendition of the device module appearing on pages 26 and 27 of the Modula Report by N. Wirth [17]. The module defines a time clock, that is incremented every 20 ms, a signal tick that is sent every 20 ms, and a procedure Pause(n) that delays the calling process by n*20 ms:

```
MONITOR RealTime
  IMPORT  MyStandardConventions;
  EXPORT  time, tick, Pause;

  REPRESENTATION word =
     ARRAY(INTEGER[1..WORDLENGTH()] OF BOOLEAN) PACKED;
  VAR time: INT;
        tick: SIGNAL;
        lcs AT BASE8 177546: word;
          (* line clock status. *)
        clockSignal AT BASE8 100: SIGNAL;
          (* this is the signal set by hardware. *)

  PROCEDURE Pause(CONST n: INT)
    VAR delay: INT := n;
    WHILE delay >0 DO
      WAIT(tick); delay := delay - 1;
    ENDWHILE;
  ENDPROCEDURE Pause;

  PROCESS Clock() PRIORITY 6
    lcs[6] := TRUE;
    LOOP forever
      WAIT(clocksignal);
      time := time + 1;
      WHILE AWAITED(tick) DO SEND(tick) ENDWHILE
    ENDLOOP forever;
  ENDPROCESS clock;

  (* body of module realtime follows. *)
  time := 0; Clock();

ENDMONITOR RealTime;
```

B. *Convex Hulls*

The following module template will instantiate to yield a module that exports facilities to determine the convex hull of a set of points in 2-space:

```
TEMPLATE ConvexHullTemplate(size)
  (* size required to be a manifest constant. *)
  MODULE ConvexHullFinder
  IMPORT MyStandardConventions, EXPT;
  EXPORT reset,addpoint,getstatus,pointstatus,convexhull;

  REPRESENTATION
     setsize = INTEGER[1..size];
     pointstatus = UNORDERED(inside,onhull,unknown);
     point = RECORD VAR x,y: FLOAT; status: pointstatus ENDRECORD;
     pointlist = ARRAY([1..(size+1)] OF point);
  VAR
     pointlistindex: INT := 0;
     points: pointlist;

  PROCEDURE reset() pointlistindex := 0; RETURN ENDPROCEDURE reset;

  PROCEDURE addpoint RETURNS BOOLEAN (CONST x,y: FLOAT)
    IF pointlistindex = size THEN RETURN FALSE
    ELSE
      pointlistindex := pointlistindex + 1;
      points[pointlistindex] := point(x,y,unknown);
      RETURN TRUE;
    ENDIF;
  ENDPROCEDURE addpoint;

  PROCEDURE getstatus RETURNS pointstatus (CONST i: setsize)
    RETURN points[i].status;
  ENDPROCEDURE getstatus;

  PROCEDURE convexhull()
    (* Similar to Jarvis' algorithm. *)
    VAR
       a,b,c,d,z,xmin,ymin,zmin,base: FLOAT;
       j,lastj,nextj: INT;
       start: point;
       done: BOOLEAN;

    FOR i: setsize DO points[i].status := unknown  ENDFOR;
    j := 1;
    xmin := points[1].x;
    ymin := points[1].y;
    FOR k : INTEGER [2..size] DO
      IF points[k].x < xmin THEN xmin := points[k].x ENDIF;
      IF points[k].y < ymin
        THEN  ymin := points[k].y; j := k;
      ENDIF
    ENDFOR;
```

```
start.x := points[j].x;
start.y := points[j].y;
points[j].status := onhull;
lastj := size+1;
points[lastj].x := xmin-1.0;
points[lastj].y := start.y;
base := EXPT((start.x-points[lastj].x),2);

LOOP grind
  done := TRUE;
  zmin := 4.01;
  a := base;
  FOR k : setsize DO
    IF points[k].status = unknown
    THEN
      done := FALSE;
      IF vorticity(points[j],points[k],start) > 0.0
        THEN
        points[k].status := inside;
        ELSE
        b := metric(points[j],points[k]);
        c := metric(points[k],points[lastj]);
        d := a+b-c;
        IF b > 0.0
          THEN
          z := d*ABS(d)/(a*b);
          IF z < zmin
            THEN  zmin := z;
              nextj := k;
              base := b  ENDIF
          ELSE points[k].status := inside  ENDIF;
      ENDIF;
    ENDIF;
  ENDFOR;
  points[nextj].status := onhull;
  IF done THEN EXIT grind ENDIF;
  lastj := j;
  j := nextj
 ENDLOOP grind
ENDPROCEDURE complexhull;

FUNCTION metric RETURNS FLOAT (CONST p,q: point)
  RETURN EXPT((p.x-q.x),2) + EXPT((p.y-q.y),2)
ENDFUNCTION metric;

FUNCTION vorticity RETURNS FLOAT (CONST p,q,r: point) INLINE
  RETURN p.x*(q.y-r.y)-p.y*(q.x-r.x)+r.y*q.x-r.x*q.y
ENDFUNCTION vorticity;

 ENDMODULE ComplexHullFinder;
ENDTEMPLATE ComplexHullMacro;
```

C. *Exception Handling*

The following fragment of code illustrates the language's exception-handling facilities. We assume that declarations for the program variables root, poly, denom, ..., etc. have been made in the scope that encloses the fragment.

```
FOR k: INTEGER [1..n] DO
  BEGIN
    VAR repcount: INTEGER [0..maximum] := 0;
    WHILE repcount ≠ maximum DO
      overflowscope:
      BEGIN
      (* Within this scope we evaluate the polynomial for  *)
      (* for the k'th root.  On overflow we divide the     *)
      (* root by 10 and repeat the evaluation, for up to   *)
      (* "maximum" times. *)

      EXCEPTION CASE overflow => 
              repcount := repcount+1;
              IF repcount < maximum
                THEN  root[k] := 0.1*root[k]
                ELSE  PROPAGATE_EXCEPTION rescale
              ENDIF
      ENDEXCEPTION;

      poly := a[n];
      FOR j : INTEGER [1..n] DO
         poly := root[k]*poly + a[n-j]
      ENDFOR;
      repcount := maximum;
      (* Evaluating the polynomial without overflow  *)
      (* forces exit from the WHILE loop. *)
      END overflowscope
    ENDWHILE
  END;

  denom := a[n];

  BEGIN
  (* If the calculation of denom causes overflow,  or if denom is zero *)
  (* causing zerodivide, then the root is perturbed and not recomputed. *)
  EXCEPTION CASE overflow,zerodevide => root[k] := root[k] + epsilon(root[k],k)
  ENDEXCEPTION;

  FOR j : INTEGER [1..n] DO
    IF j ≠ k THEN denom := denom*(root[k] - root[j]) ENDIF
  ENDFOR;
  root[k] := root[k] - poly/denom
  END

ENDFOR;
```

III

D. *File Manipulation*

This is a rendition of the modules appearing as examples on pages 19 and 20 of the Modula Report by N. Wirth [17].

```
MODULE lineinput
  IMPORT inchr,outchr,pdp11chars,MyStandardConventions;
  EXPORT read,newline,newfile,eoln,eof,lno;
  VAR lno: INT; (* line number *)
        ch: pdp11chars; (* last character read  *)
        eof,eoln: BOOLEAN := TRUE;

  PROCEDURE newfile()
    IF NOT eof THEN
      LOOP readchr
        inchr(ch);
        IF ch = 'fs' THEN EXIT readchr ENDIF
      ENDLOOP readchr;
    ENDIF;
    eof := FALSE;  lno := 0;
  ENDPROCEDURE newfile;

  PROCEDURE newline()
    IF NOT eoln THEN
      LOOP readchr
        inchr(ch);
        IF ch = 'lf' THEN EXIT readchr ENDIF
      ENDLOOP readchr;
      outchr('cr');
      outchr('lf');
    ENDIF;
    eoln := FALSE;  lno := lno + 1;
  ENDPROCEDURE newline;

  PROCEDURE read(VAR x: pdp11chars)
    (* assume NOT eoln AND NOT eof.  *)
    LOOP nibble
      inchr(ch); outchr(ch);
      IF ch >= ' ' THEN x := ch; EXIT nibble ENDIF;
      IF ch = 'lf' THEN x := ' '; eoln := TRUE; EXIT nibble ENDIF;
      IF ch = 'fs'
        THEN
          x := ' ';
          eoln := TRUE;
          eof := TRUE;
          EXIT nibble
      ENDIF;
    ENDLOOP nibble;
  ENDPROCEDURE read;
ENDMODULE lineinput;
```

E. *PDP11 Disk Track Reservation*

```
MODULE trackreservation  (* page 20 Modula Report [17]. *)
  IMPORT MyStandardConventions;
  EXPORT newtrack,returntrack;
  CONST
    m: INT = 64; w: INT = 16; (* there are m*w tracks  *)
  REPRESENTATION
    bands = INTEGER [0..(m-1)];
    bandtracks = INTEGER [0..(w-1)];
    bits = ARRAY(bandtracks OF BOOLEAN);
  VAR
    reserved: ARRAY(bands OF bits);

  PROCEDURE newtrack RETURNS INT ()
    (* reserves a new track, yields its index as the  *)
    (* returned value if a free track is found. *)
    (* otherwise, returns value -1. *)
    FOR i : bands DO
      FOR j : bandtracks DO
        IF NOT reserved[i][j] THEN
          reserved[i][j]:= TRUE;
          RETURN i*w + j
        ENDIF;
      ENDFOR;
    ENDFOR;
    RETURN -1;
  ENDPROCEDURE newtrack;

  PROCEDURE returntrack(CONST k: INT)
    (* assume  0 <= k < m*w  *)
    reserved[k DIV w][k MOD w] := FALSE;
  ENDPROCEDURE returntrack;

  (* body of module trackreservation—mark all tracks free *)

  FOR i: bands DO reserved[i] := bits(FALSE) ENDFOR

ENDMODULE trackreservation;
```

III

F. *Fault-Tolerant Avionics*

The following is a rendition of a scheduling algorithm for the management of repetitive tasks in the SIFT system. It does not include the mechanism for initializing the various data-structures describing the task allocation.

```
MODULE scheduler
  IMPORT
    MyStandardConventions,
    enable,disable,tickinterrupt,errorhelper,
    framecount,errortype,maxpriority,maxtasks;
  EXPORT
    call,schedule;
  REPRESENTATION
    levels = INTEGER [1..(maxpriority+1)];
    call = UNORDERED (normal,abnormal,clocktick);
  CONST
    period: ARRAY(levels OF INT);
  VAR
    sked: ARRAY(levels,INTEGER [1..maxtasks] OF INT);
    enabled: ARRAY(levels OF BOOLEAN);
    job: ARRAY(INTEGER [1..maxtasks] OF INT);
    entry: ARRAY(levels OF INT);
    pri,ept: INT;

(* Initially, sked[i,j] contains the absolute entry-point address   *)
(* of the jth task at priority-level i, where j is a scheduled task *)
(* on the host processor.  Other values of sked[i,j] are initially  *)
(* set to a value out of the bounds 0..memsize, (a negative integer).*)
(* Initially, entry[i] = sked[i,1], pri = priority of the diagnostic *)
(* task d, period[i] contains the number of timer ticks per frame of *)
(* tasks having priority-level i, job[pri] = the j index of d.       *)
(* All other values of job[i] = 0.  enabled[i] is initially TRUE.    *)

  PROCEDURE schedule RETURNS INT
    (CONST progcounter: INT; CONST calltype: call)
    disable(tickinterrupt);
    SELECT calltype
      CASE abnormal =>
        errorhelper(pri,job[pri],aborted);
      CASE clocktick =>
        IF framecount MOD period[pri] ≠ 0
        THEN
          entry[pri] := progcounter + 1 (* new entry-point  *)
        ELSE
          errorhelper(pri,job[pri],overrun);
          job[pri] := job[pri] + 1;  (* job overrun.      *)
          entry[pri] := sked[pri,job[pri]]
        ENDIF;
        pri := 1; (* return to top    *)
        job[1] := 0; (* level priority.  *)
    ENDSELECT;

    (* The code for the normal job termination case follows.  *)
```

```
(* We drop into this code after servicing either of the    *)
(* other calltypes.                                        *)

 job[pri] := job[pri] + 1; (* go to next job   *)
 ept := sked[pri,job[pri]]; (* get entry-point  *)
 IF ept > 0 AND enabled[pri]
 THEN RETURN ept
 ELSE
   LOOP quickly
     entry[pri] := ept;   (* all jobs done    *)
     pri := pri + 1;      (* drop one level   *)
     ept := entry[pri];   (* get saved ept    *)
     IF ept > 0 AND enabled[pri]
     THEN
       RETURN ept
     ELSE
       IF framecount MOD period[pri] = 0
       THEN               (* ready to run jobs *)
         job[pri] := 1;   (* at this level.    *)
         ept := sked[pri,1];  (* reset to 1-st job *)
         RETURN ept
     ENDIF;
   ENDLOOP quickly;

 (* The above loop cannot fail to terminate or run off the   *)
 (* bottom of the sked array because the diagnostic task     *)
 (* has the lowest priority level and never terminates       *)
 (* normally.  That is, an enabled ept will always exist     *)
 (* at that priority level.                                  *)

 ENDIF;

 enable(tickinterrupt);

ENDPROCEDURE schedule;
ENDMODULE scheduler;
```

III

G. *Pointers and Opaque Exportation*

The following module illustrates the use of pointers. An enumeration, Zodiac, is exported to the module user along with an entity Tree, which is implemented as a pointer. However, the representation and type of Tree are concealed from the user by exporting Tree opaquely. The infix operators = and ≠ are also exported so that the user can test equality of Trees. The structure of a typical tree is illustrated in Figure 2.

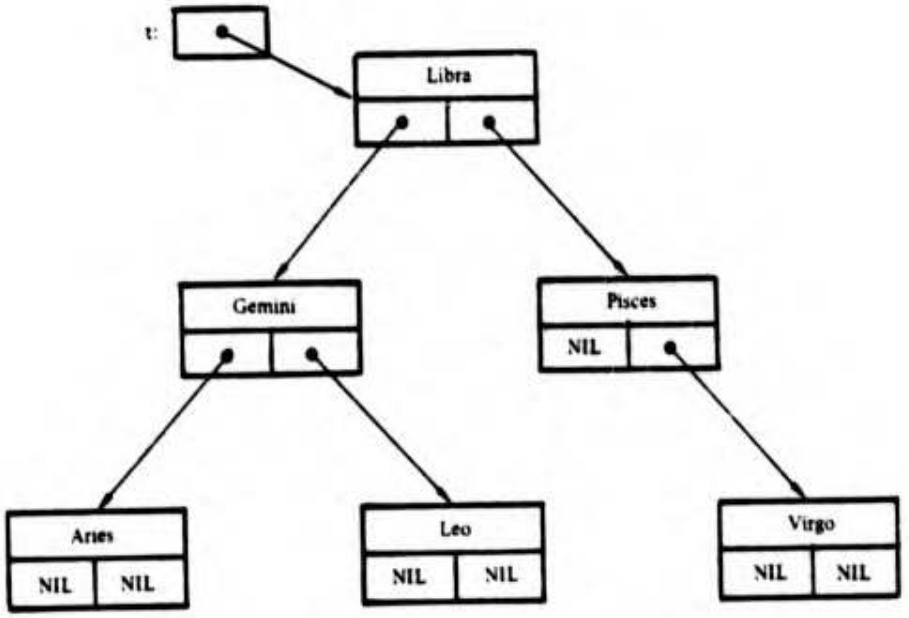

Figure 2 A BINARY TREE

```
MODULE SearchTree
  EXPORT OPAQUE Tree;
  EXPORT TRANSPARENT Zodiac;
  EXPORT NewTree,Search,=,≠;
  REPRESENTATION
    Zodiac =
      ORDERED(Aquarius,Aries,Cancer,Capricorn,Gemini,
              Leo,Libra,Pisces,Sagitarius,Scorpio,
              Taurus,Virgo);
    Tree = POINTER(Node);
    Node = RECORD
             CONST key: Zodiac;
             VAR llink,rlink: Tree;
```

```
        ENDRECORD;
CONST
  newTree: Tree = Tree NIL;
PROCEDURE Search RETURNS Tree
  (CONST SearchKey: key, VAR t: Tree)
  IF t = newTree
     THEN
       NEW(t);    (* t to a new Tree *)
       t↑ := Node(SearchKey,newKey,newKey);
       RETURN t
     ELSIF SearchKey < t↑.key
       THEN RETURN Search(SearchKey, t↑.llink)
     ELSIF SearchKey > t↑.key
       THEN RETURN Search(SearchKey, t↑.rlink)
     ELSE
       RETURN t
     ENDIF;
 ENDPROCEDURE Search;
ENDMODULE SearchTree;
```

III

H. *Hash Tables*

In the following module, NumberTables may be abstractly viewed as sets, of some limited capacity, whose elements are integers. One may test whether an element is in a set, remove an element from a set, or add a new element to a set (provided that the set is not already at its maximum size).

```
MODULE NumberTableImplementation
  IMPORT MyStandardConventions;      (* defines INT, FLOAT, etc. *)
  EXPORT NumberTable, Search, Delete, Insert;
  CONST
   tablesize: INT = 763;
  REPRESENTATION
   tableRange = INTEGER [1..tableSize];
   State = UNORDERED (fresh, full, deleted);
   TableEntry =
     RECORD
       SELECT flag:State
         CASE full => VAR key: INT;
         CASE fresh, deleted => (* no components *);
       ENDSELECT
     ENDRECORD;

   IndexSlot   = RECORD CONST index, slot: tableRange ENDRECORD;
   NumberTable = ARRAY(tableRange OF TableEntry);

  FUNCTION Scan RETURNS IndexSlot
   (CONST key: INT; table: NumberTable)
   VAR slot: tableRange := 0;
   FOR i: tableRange DO
      BEGIN
       CONST probe: tableRange = (key+i) MOD tableRange + 1;
       SELECT Tag(table[probe])
         CASE full =>
           IF table[probe].key=key
             THEN RETURN IndexSlot(probe, slot) ENDIF;
         CASE deleted => IF slot=0 THEN slot:=probe ENDIF;
         CASE fresh =>
           IF slot=0 THEN slot:=probe ENDIF;
           RETURN IndexSlot(0, slot)
       ENDSELECT
      END
   ENDFOR;
   RETURN IndexSlot(0, slot);
  ENDFUNCTION Scan;

  PROCEDURE Delete(CONST key: INT; VAR table: NumberTable)
    CONST index = Scan(key, table).index;
    IF index ≠ 0
      THEN table[index] := TableEntry(deleted) ENDIF
  ENDPROCEDURE Delete;

  PROCEDURE Insert RETURNS BOOLEAN
    (CONST key: INT; VAR table: NumberTable)
```

```
    (* Returns TRUE if and only if the key is in the table *)
    (* afterward; FALSE means the item wasn't there and    *)
    (* table was full *)
    CONST indexSlot: IndexSlot = Scan(key, table);
    IF indexSlot.index=0
      THEN
      IF indexSlot.slot≠0
        THEN table[slot]:=TableEntry(full, key); RETURN TRUE
        ELSE RETURN FALSE
      ENDIF
      ELSE RETURN TRUE
    ENDIF;
  ENDPROCEDURE Insert;

  FUNCTION Search RETURNS BOOLEAN
    (CONST key: INT; CONST table: NumberTable)
    RETURN Scan(key, table).index ≠ 0
  ENDFUNCTION Search;

ENDMODULE NumberTableImplementation;
```

III

1. *Exchange Sort*

In this subsection we present a module that exports a procedure to sort according to the ASCII collating sequence. The example illustrates the use of length-unresolved array parameters, overloading, and alphabets.

```
MODULE ExchangeSort
  IMPORTS String, ASCII;
    (* the representation String is a fixed-length array of ASCII *)
    (* characters;  ASCII is the Ascii alphabet representation    *)
  EXPORTS Sort;

  FUNCTION EXTENDS > RETURNS BOOLEAN (CONST s1, s2: String)
    VAR char: INTEGER [1..LENGTH(s1)] := 1;
    WHILE s1[char] > s2[char] AND char <= LENGTH(s1) DO
      char := char+1;
    ENDWHILE;
    IF char > LENGTH(s1) THEN RETURN TRUE ELSE RETURN FALSE ENDIF;
  ENDFUNCTION >;

  PROCEDURE Sort (VAR stringArray: ARRAY(1 DIMENSIONS OF String))
    PROCEDURE Exchange (VAR i1, i2: INTEGER [1..LENGTH(stringArray)])
      VAR temp: ASCII;
      FOR char: INTEGER [1..LENGTH(stringARRAY[1])] DO
        temp := stringArray[i1][char];
        stringArray[i1][char] := stringArray[i2][char];
        stringArray[i2][char] := temp;
      ENDFOR;
      lastSorted := i1;
    ENDPROCEDURE Exchange;
    VAR
      lastUnsorted: INTEGER [1..LENGTH(stringArray)-1]
                  := LENGTH(stringArray)-1;
      lastSorted: INTEGER [1..LENGTH(stringArray)]
                := LENGTH(stringArray);
    LOOP unsorted
      FOR i: INTEGER [1..lastUnsorted] DO
        IF lastSorted = i THEN EXIT unsorted ENDIF;
        IF stringArray[i] > stringArray[i+1]
          THEN Exchange(i, i+1)
        ENDIF;
      ENDFOR;
      IF lastSorted > lastUnsorted THEN EXIT unsorted ENDIF;
      lastUnsorted := lastUnsorted-1;
    ENDLOOP unsorted;
  ENDPROCEDURE Sort;

ENDMODULE ExchangeSort;
```

J. *Machine-Level Input-Output for Decsystem 10*

This example illustrates the use of machine-code inserts combined with high-level language code. The basis for all that follows is the Decsystem 10 Assembly Language Handbook [11]; see especially pp. 461+.

```
MODULE DECSYSTEM10
  (* this is only a fragment of the actual Decsystsem10 machine *)
  (* dependencies, provided for concreteness;  Many other routines *)
  (* used to describe machine code inserts, exception names, etc. are *)
  (* omitted for brevity but alluded to below *)
  EXPORT TRANSPARENT
    INT36, WORD, HALFWORD, ADDRESS, ONEBIT,
    SIXBIT_ALPHABET, SIXBIT, BYTE_POINTER, SIXBIT;
  REPRESENTATION
    INT36 = INTEGER [BASE8 0..BASE8 777777777777] PACKED 36;
    WORD = ARRAY (INTEGER [0..35] OF BOOLEAN) PACKED 36;
    HALFWORD = ARRAY (INTEGER [0..17] OF BOOLEAN) PACKED 18;
    ADDRESS = HALFWORD;
    ONEBIT = BOOLEAN PACKED 1;
    SIXBIT_ALPHABET =
      ALPHABET (' ','!','dq'(*double quote*),'#',
                '$','%','&','sq'(*single quote*),
                '(',')','*','+',',','-','.','/',
                '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','[','\',']','^','_');
    SIXBIT = ARRAY (INTEGER [1..6] OF SIXBIT_ALPHABET);
    BYTE_POINTER =
     RECORD VAR
       p AT 0: INTEGER [0..36] PACKED 6;  (* bits remaining *)
       s AT 6: INTEGER [0..36] PACKED 6;  (* byte size      *)
       i AT 13: ONEBIT;                   (* indirect bit   *)
       x AT 14: INTEGER [0..15] PACKED 4; (* index register *)
       y AT 18: ADDRESS (* word base address *)
     ENDRECORD;
ENDMODULE DECSYSTEM10;

TEMPLATE getBufferRing(numberOfBuffers,numberOfWords)
  MODULE bufferRing

  IMPORT DECSYSTEM10;
  EXPORT TRANSPARENT ioData, initializeBuffers;

  REPRESENTATION
    ioData =
(* this structure defines all the information needed to do io  *)
(* to a device on a specified channel:  a buffer header,       *)
(* a buffer ring, and data needed in the "open" operation      *)
     RECORD VAR
```

```
      buffers: (* see pp. 466+, paragraph 4.3 *)
       ARRAY (INTEGER [1..numberOfBuffers] OF
         RECORD VAR
            fileStatus AT 18: HALFWORD;
            useBit AT 36: ONEBIT;
            dataAreaSize AT 37: INTEGER [0..131071] PACKED 17;
            nextBuffer AT 54: ADDRESS;
            bookkeeping AT 72 : HALFWORD;
            wordCount AT 90 : INTEGER [0..262143] PACKED 18;
            data AT 108: ARRAY (INTEGER [1..numberOfWords] OF WORD)
         ENDRECORD)

      bufferHeader: (* see pp. 466+, paragraph 4.3.1 *)
       RECORD VAR
            useBit AT 0: ONEBIT;
            currentBuffer AT 18: ADDRESS,
            bufferPointer AT 36: BYTE_POINTER;
            byteCounter AT 72: INT36;
            openDescriptor: openSpec
       ENDRECORD;
    ENDRECORD;

  REPRESENTATION openSpec =
    RECORD VAR (* see pp. 462+, paragraph 4.2 *)
        status AT 0: HALFWORD;
        deviceName AT 36: SIXBIT;
        outBuffer AT 72: ADDRESS;
        inBuffer AT 90: ADDRESS
    ENDRECORD;

  PROCEDURE initializeBuffers(VAR io: ioData; CONST deviceName: SIXBIT)
     (* first, make the buffer header point to the first buffer *)
   io.bufferHeader.currentBuffer := ADDR(io.buffers[1]);
     (* next, link the buffers in a ring - each to the next...*)
   FOR i: [1..numberOfBuffers-1] DO
        io.buffers[i].nextBuffer := ADDR(io.buffers[i+1])
   ENDFOR;
     (*...and the last to the first *)
   io.buffers[numberOfBuffers] := ADDR(io.buffers[1]);
     (* the first halfword of the open descriptor is a "don't care" *)
     (* the first field, the second halfword, is the DECSYSTEM10 *)
     (* code for "ascii line mode"; the second field is filled with *)
     (* 6 SIXBIT characters; the third and fourth fields are the *)
     (* addresses of the input and/or output buffer headers; *)
     (* for us, there will only be one, and--not caring to know *)
     (* whether it is input or output--we put it in both fields *)
   io.openDescriptor :=
     openSpec(integerToHalfword(1), deviceName,
              ADDR(io.bufferHeader), ADDR(io.bufferHeader));
  ENDPROCEDURE initializeBuffers
  ENDMODULE bufferRing
ENDTEMPLATE getBufferRing;

MODULE doio
```

```
EXPORT get, put, finishIo;
IMPORT machineTypes, getBufferRing, DECSYSTEM10;

(*    Whereever DECSYSTEM10 is used, the straightforward          *)
(*    way to achieve the desired effect is to run actual machine  *)
(*    code.  For this example, we presume that the machine is that *)
(*    virtual machine presented by the DECSYSTEM10 Operating      *)
(*    System.  The procedures SOSGE, JRST, ILDB, etc. produce     *)
(*    the obvious machine instructions (see the cited machine     *)
(*    language manual);  the special 'instruction' procedures used *)
(*    are MRETURN and MTRIGGER_EXCEPTION, which are analogs of the *)
(*    high level primitives RETURN and PROPAGATE_EXCEPTION.       *)
(*    The first of these causes the enclosing DECSYSTEM10 call to  *)
(*    return whatever value is in the cited accumulator.  The     *)
(*    second triggers the named execption, so that it may be      *)
(*    handled by an enclosing exception handler.  The special     *)
(*    function THIS_ADDRESS returns the machine address at which  *)
(*    the containing instruction is to be placed.  None of these  *)
(*    are available except in a context where DECSYSTEM10 has been *)
(*    imported *)

 CONST inChannel: INT36 = 1; outChannel: INT36 = 2;
       (* thus we forbear gratuitous generality *)

 REPRESENTATION fileSpec =
(*  the mysteries of this data structure are defined by the system; *)
(*  it suffices to know that if the user fills in the file name and *)
(*  the file name extension, and sets all else to zero, as is done  *)
(*  by the initializing procedure below, then the DECSYSTEM10 will  *)
(*  fill in the rest correctly;  for further details consult        *)
(*  pp. 564+, paragraph 6.2.8.1                                     *)

   RECORD VAR
      name AT 0: SIXBIT;
      ext AT 36: ARRAY(INTEGER [1..3]  OF SIXBIT_ALPHABET) PACKED;
      hd2 AT 54: ARRAY(INTEGER [1..3]  OF BOOLEAN) PACKED 3;
      d1 AT 57: ARRAY(INTEGER [1..15] OF BOOLEAN) PACKED 15;
      hd2d1 AT 57: INTEGER [0..16777215] PACKED 24;
         (* to permit wholesale zeroing of hd2 and d1 *)
      prot AT 72: ARRAY(INTEGER [1..9]  OF BOOLEAN) PACKED 9;
      m AT 81: ARRAY(INTEGER [1..4]  OF BOOLEAN) PACKED 4;
      time AT 85: ARRAY(INTEGER [1..11] OF BOOLEAN) PACKED 11;
      ld2 AT 96: ARRAY(INTEGER [1..12] OF BOOLEAN) PACKED 12;
      word3 AT 72: INT36;
         (* to permit wholesale zeroing of prot thru ld2 *)
      ppn AT 108: WORD
   ENDRECORD;

 PROCEDURE initializeFileSpec
       (CONST name: SIXBIT; ext: ARRAY(INTEGER [1..3] OF SIXBIT);
        OUT f:  fileSpec)
   f.name := name; f.ext := ext; f.hdsd1 := 0; f.word3 := 0;
   f.ppn := WORD(FALSE)  (* specifying default directory *);
```

```
ENDPROCEDURE initializeFileSpec;

PROCEDURE finishIo() (* invoked when end-of-file is encountered *)
  PDP10(CLOSE(inChannel),
        CLOSE(outChannel),
        RELEAS(inChannel),
        RELEAS(outChannel));
  RETURN;
ENDPROCEDURE finishIo;
MODULE input

  IMPORT getBufferRing, inChannel, DECSYSTEM10;
  EXPORT get, VAR inputData;

  INSTANCE getBufferRing(3,64); (* triple buffering for output *)
  VAR inputData: ioData;
  PROCEDURE get RETURNS ASCII ()
   PDP10(
     SOSGE(ADDRESS(inputData.bufferHeader.byteCounter)),
     JRST(THIS_ADDRESS()+4),
     ILDB(AC3,ADDRESS(inputData.bufferHeader.bufferPointer)),
     JUMPE(AC3,THIS_ADDRESS()-3),
     MRETURN(AC3),
     IN(inChannel),
     JRST(THIS_ADDRESS()-6),
     STATZ(inChannel,74N8S12),
     MTRIGGER_EXCEPTION(READ_ERROR),
     MTRIGGER_EXCEPTION(EOF))
  ENDPROCEDURE get;
  initializeBuffers(inputData, SIXBIT_ALPHABET "DATA  ");
ENDMODULE input;

MODULE output

  IMPORT getBufferRing, outChannel, DECSYSTEM10;
  EXPORT put;

  VAR outputData;
  INSTANCE getBufferRing(2,64); (* double buffering for output *)
  VAR outputData: ioData;

  PROCEDURE put(CONST c AT AC3: ASCII)
   PDP10(
      SOSG(ADDRESS(outputData.bufferHeader.byteCounter)),
      JRST(THIS_ADDRESS()+3),
      IDPB(AC3,ADDRESS(outputData.bufferHeader.bufferPointer)),
      MRETURN(),
      OUT(outChannel),
      JRST(THIS_ADDRESS()-3),
      MTRIGGER_EXCEPTION(OUTPUT_ERROR))
  ENDPROCEDURE put;

  initializeBuffers(outputData, SIXBIT_ALPHABET "LIST  ");
```

```
  ENDMODULE input;

  VAR I1, e2: fileSpec;

  initializeFileSpec
    (SIXBIT_ALPHABET "INFILE", SIXBIT_ALPHABET "DAT", I1);
  initializeFileSpec
    (SIXBIT_ALPHABET "OUTFIL", SIXBIT_ALPHABET "LST", e2);
  PDP10(
    RESET(),
    OPEN(inChannel,ADDRESS(inputData.openSpec)),
    MTRIGGER_EXCEPTION(INPUT_OPEN_ERROR))
    OPEN(outChannel,ADDRESS(outputData.openSpec)),
    MTRIGGER_EXCEPTION(OUTPUT_OPEN_ERROR),
    LOOKUP(inChannel,ADDRESS(I1)),
    MTRIGGER_EXCEPTION(LOOKUP_ERROR),
    ENTER(outChannel,ADDRESS(e2)),
    MTRIGGER_EXCEPTION(ENTER_ERROR))
ENDMODULE doio;

MODULE filtrn
  (* This is the main program: like NEWCHR loop on *)
  (* pp. 486-487, paragraph 4.9.3 *)
  IMPORT doio, DECSYSTEM10;

  EXCEPTION
    CASE EOF => finishIo() ELSE DECSYSTEM10(HALT())
  ENDEXCEPTION;

  LOOP tiny put(get()) ENDLOOP tiny;

ENDMODULE filtrn;
```

PART FOUR

DESIGN DISCUSSION

CONTENTS

I INTRODUCTION

Part Four, Design Discussion, is structured similarly to Part Two, Language Report, and should be read in parallel with it. For each section of the Language Report that introduces a novel feature, or adopts a significant language facility over interesting alternatives, the reader will find a discussion of the reasons for our choice in the parallel section of this part.

III NOTATION, TERMINOLOGY, AND VOCABULARY

A. *Comments*

The language contains a single comment facility: the "(*" and "*)" brackets, which may be used to delimit a comment at any place in the program text. A comment, thus delimited, is equivalent to a space. This is exactly the comment facility of Pascal, but our language imposes, in accord with Ironman requirement 2I, the additional restriction that the opening and closing brackets of a comment must occur on the same physical line of the source program. An alternative would be to omit this additional restriction. A second alternative would be to let the new-line character be a default end-of-comment bracket. We dislike the ad hoc nature of the second alternative and would, in the absence of the Ironman requirement, have preferred to adopt the first alternative—that is, the original Pascal comment. (Note that the stand-alone comment line need not be a separate facility in any of these schemes; in our present scheme the stand-alone comment is a source program line with one properly bracketed comment.)

IV IDENTIFIERS, NUMBERS, CHARACTERS, AND STRINGS

A. *Floating-Point Literals*

We have included both fixed-point and floating-point literals in our language; the obvious alternative is to provide a single class of literals that serves both purposes. Our language design has included both fixed-point and floating-point arithmetic. In accord with Ironman, the semantics of fixed-point arithmetic are exact, and a trap (i.e., primitive run-time error) is required if the result of a fixed-point operation cannot be represented. Floating-point arithmetic, by contrast, is approximate to within a user-specified precision. It seems consistent with this design to reflect this duality in the separation of the two classes of literals. Moreover, a single-literal facility would require a host of rules governing conversion of a constant to fixed or floating point, whereas the facility provided needs no such rules.

B. *Spelling of Numeric Literals*

We chose the rules for spelling numeric literals to achieve a clean, readable, error-resistant, and uniform notation; however, the details of these rules are by no means intrinsic to our design and could easily be changed.

Of particular note is our scheme for specifying the base in which constants are interpreted. We require that any constant not in base 10 be preceded by one of the indicators BASE2, BASE8, or BASE16. (The prefix BASE10 is optional.) Thus the following integers are equal:

```
27
BASE2 11011
BASE8 33
BASE10 27
BASE16 1B
```

(Note that this disambiguation scheme uses the same notation as that for enumeration constants in overlapping enumeration types.)

C. *Character and String Literals*

Our descriptions of character and string literals must be read together with the rules for alphabets (see Section VI of the Language Report). An alphabet is a scalar type whose elements are character literals—identifiers enclosed in apostrophes. Some character literals are denoted by their actual printing forms—for example, 'A', '1', and '!'. Other character literals

have no printing forms and are denoted by appropriate mnemonics, specified by the defining alphabet—for example, 'BEL', 'SOH', 'CR', and 'LF'. In order to avoid the introduction of special overrides into this uniform notation, we require that the apostrophe and double-quote characters be treated in the second way—denoted, for example, as 'SQ' and 'DQ'.

String literals are sequences of character literals delimited by double-quote characters. However, when a constituent character literal has a single-character mnemonic, we allow its apostrophes to be omitted. This convention is concise for such common cases as "THIS IS A STRING", while making manifest the occurrences of nonprinting characters in strings, as in "THIS IS A STRING 'BEL' WITH EMBEDDED BELLS 'BEL'".

Note that we have complied with Ironman requirement 2H that string literals not be permitted to cross source program line boundaries. This requirement was intended, in part, to make new-line characters manifest in strings; note that this is achieved in our design by the character literal denotations just described. (Note that our rule for permitting a string literal to cross physical line boundaries of the source program—that it must be closed and reopened at each intermediate line-transition—is analogous to the mechanism whereby comments span physical line boundaries.)

VI DATA TYPE DEFINITIONS AND REPRESENTATIONS

In a language that is strongly typed and to be used for systems programming (among other applications), it is appropriate to classify sets of values in two ways: according to *type* and *representation*. The second classification is a refinement of the first—that is, a single type may have several representations and different types never have the same representation.

Hoare has explained [5] that types have a number of characteristics:

- A type determines the class of values that may be assumed by a variable or expression.
- Every value belongs to one and only one type.
- The type of a value denoted by any constant, variable, or expression may be deduced from its form or context, without any knowledge of its value as computed at run time.
- Each operator expects operands of some fixed type, and delivers a result of some fixed type. Where the same symbol is applied to several different types, this symbol may be regarded as ambiguous, denoting several different actual operators. The resolution of such systematic ambiguity can always be made at compile time, based on the types of the operands.

For example, the value represented by the literal 1 is an integer and not a value of any other type. The floating-point value 1.0 is distinct from the integer 1. The type of a value, variable, or constant determines the meaning of many applicable operations. The meaning of the operation "+" when applied to two integers is different from the meaning of "+" when applied to two floating-point numbers. Thus the name of an operation, together with the types of its arguments, can serve to select a particular meaning for the operation in any given linguistic context.

Although the types of the variables, constants, and parameters of an algorithm are central in its initial (and often implicit) formulation, the transformation of this formulation into an efficient computer program usually requires the qualification of types by representational information. The most common case in practice is a type with a single representation, a fact of which Pascal takes advantage by, in effect, merging the notion of type and representation. Since this case is common, it is particularly important that the programmer have available a single notion and a single notation with essentially these Pascal semantics. However, to comply

with the Ironman requirements on types, representations, and conversion, it is necessary to provide facilities in addition to those of Pascal. (Indeed, Habermann has noted that this is a source of difficulty in Pascal [4].)

Our syntax and semantics provide this dual notion of abstract type and concrete type. Most programmers will find it quite convenient to ignore the existence of abstract types for which there are several concrete types, using REPRESENTATION definitions in place of TYPE definitions in the common case that an abstract type has only one concrete realization in a program. Our syntax for types and type definitions will be seen to be parallel to that for concrete types and representation definitions. Therefore, it is quite easy, as a program becomes more sophisticated, to add additional representations of a common abstract type. This scheme has many advantages: it makes the common case easy, it retains the type structure of Pascal in this common case, and it provides a clean mathematical generalization of Pascal, as required by Ironman, to the case with multiple realizations of an abstract type. Moreover, as will be discussed below, the rules for determining type equality in our scheme (including the general case) are extremely simple and efficiently implementable.

Representations of a type are used by a programmer to gain some control over how values are represented in an implementation and as a means of error checking (in the case of numeric ranges). It is important to distinguish between types and representations. The language is strongly typed, and changing a type in a program will generally change the semantics of that program. But, except in relation to implementation-dependent facilities, representations affect only the efficiency of correct programs, and implicit conversion between representations of a type is permitted. Moreover, when overloaded operations are described below, it will be clear that the selection of the meaning of an overloaded operation cannot be made on the basis of the representations of its arguments—ambiguity would result because an abstract value can have several representations.

A. *Numbers*

Our distinction between types and representations is a consistent way of complying with the Ironman requirement 3-1C that the range of each numeric variable be specified in a program. In general, when a variable is declared, its representation must be apparent. In the case of a record with a single Boolean component, there is a sensible standard representation and thus we consider that type to be a representation as well as a type. But a standard representation of numeric variables would yield nonportable programs and would encourage lack of programmer attention to the important question of what numeric representation is needed in a particular program. Thus INTEGER, FIXED, and FLOATING are types, but variables of these types must be declared with a representation such as INTEGER [1..10] or FIXED [1P0..100P0] SCALE 1P-2.

The fixed-point number is a means of dealing with non integer numbers in an exact and efficient manner. The floating-point numbers are not exact representations for reals but approximations and therefore not suitable for certain computations. A fixed-point variable may have values that are integer multiples of a declared scale. This scale is the minimal difference between adjacent values.

Part Two, The Language Report, now states that implementations may limit the fixed-point scales that will be accepted. It is clear that on a binary machine, significant overhead is involved in calculating with a scale that is not a power of two. This overhead involves both run-time scale manipulation and complexity in compilers to deal with scales that are powers of different bases. We believe, therefore, that such calculations should not be invoked with the same notations used to invoke primitive machine arithmetic. Moreover, we believe that an implementation should be required to support only the fixed-point scales that are reasonable on

a particular machine. On most binary machines, only scales that are powers of two will be allowed, because arithmetic on numbers that are integer multiples of powers of two can be efficiently implemented. In machines with other arithmetic bases, such as decimal arithmetic, different scales will be reasonable and should be provided by implementations. This is an area where the absence of an adequate hardware cannot be overcome by elaborate software facilities. In the future, machine architectures should include floating-point arithmetic with mathematically correct scaling and no implicit rounding, thus making anachronistic any fixed-point facility in our language.

The fact that we have not described a general fixed-point facility, providing all scales, is not intrinsic to our design. If the government is willing to accept the cost of such generality, its addition to the language is straightforward.

B. *Alphabets*

In order to permit text processing within the language it is necessary to include character sets, character strings, and operations on characters and character strings. It is desirable to be able to support different character sets within the language. A character set can be represented in the language as an ordered scalar type (where the ordering is used to define the collation sequence). A character string is then simply a one-dimensional array of characters. To enhance readability, it is advantageous to make the representation of characters in a program resemble, as closely as possible, the characters themselves. To avoid the ambiguity that would arise between identifiers, character literals, and strings, all character literals and strings are enclosed in either single or double quotation marks (see Section III of the Language Report). Since a character can appear in many character sets, we have required that any character or string literal involving a nonstandard character set be disambiguated by the name of the alphabet used.

Our language therefore contains two kinds of ordered scalar types—ordered enumerations and alphabets. We considered and rejected the alternative of merging these into a single ordered type on the grounds that they are two distinct notions with rather different uses. (Moreover, enumerations, as commonly used in Pascal, include only identifiers as members, but alphabets must include characters not properly part of any identifier—e.g., '$' or '.'.)

Note that the same disambiguation notation is used here with numeric literals and with enumeration constants. In the case of strings, one could in theory require disambiguation only to the extent that the character literals comprising the string were not chosen from a single alphabet. We believe that this form of disambiguation would make programs hard to read, and therefore we require that all strings be disambiguated in all program contexts with multiple alphabets.

C. *Types and Literals*

As stated above, each value in the language belongs to exactly one type. Since literals in the language represent values, it must be clear from the literal what its type is. This becomes difficult when we consider the case of two scalar types:

```
TYPE Color  = UNORDERED (Red, Green, Brown, Orange)

TYPE Flavor = UNORDERED (Lemon, Cherry, Orange)
```

In this case the value represented by the literal Orange is ambiguous: it could be either a Color or a Flavor. Two solutions to this problem are possible. The use of the same literal in more than one scalar type could be forbidden so that each literal would represent a unique value. Alternatively, each appearance of a literal in a program could be qualified to specify a unique value. We have chosen the latter solution; the former solution would not permit overlapping alphabets, which are essential. A literal is qualified by preceding it with the name of the type—e.g., Color Orange or Flavor Orange. The qualification is necessary only when the use of the literal without qualification would be ambiguous. Note that we have made wide use of this convention: it is the basis of disambiguation of the base of numeric literals, and the type of NIL pointers.

D. *Structured Types*

It should be clear that structured types are defined recursively in terms of subtypes, and structured representations are defined recursively in terms of subrepresentations. However, when writing the definition of an array type, we require the representation of the index as part of the type of the array, since the range of the index determines the abstract structure of the array. Similarly, we view the representation of the base type of a set as part of the type of the set. Finally, the base of a pointer type must be a representation since otherwise the implementation of pointers would be burdensome—it would be necessary to carry with each pointer value the representation of the base value pointed to.

We have provided simple denotations of structured values—i.e., values of structured types. In the absence of such structured values, one must initialize structured variables by a sequence of component assignments, and we believe our facility is more concise and readable. Our structured array values are not intended to be very general. They are sufficient for array initializations where all the initial component values can be listed explicitly or where all components have the same initial value; we believe that more complex array initializations should be done by program text.

E. *Variant Records*

One of the purposes of a variant record is to allow a variable that takes on values of more than one nonvariant type. An alternative means of allowing this is the *union type*. A variable of a union type takes on values of any one of the base types of the union. However, the variant records of our language are essentially equivalent to union types. The difference is really just syntactic. The union type is somewhat more elegant mathematically because it makes the notion of variant and the notion of record independent. However, in this case the more elegant structure is also, in common cases more syntactically verbose, and therefore we have not departed from the structure of Pascal.

One generalization of the Pascal variant record would allow more than one case-selection field to cover the case where a variant record structure is naturally viewed as having two independently varying substructures. Since this case can be dealt with, although somewhat less concisely, by defining a variant record with fields that are themselves variant, we have decided not to go beyond the generality required by Ironman, which is already present in Pascal, in this respect.

There is a serious issue of type safety in dealing with variant records. Pascal is not type

safe, since it allows unrestricted modification of the case-selection component. We have made this a constant component to avoid this source of difficulty. Moreover, we have required that any reference to a variant component (i.e., a component not in the common part of the record type definition) be lexically embedded in a case statement that checks that the record value is the appropriate variant. There is, in our scheme, one remaining possibility for error. Suppose p and q are both pointers with the same variant record type as base type. Consider the statement

```
SELECT p↑.tag
   CASE Red => q↑ := newValue; ...p↑.RedComponent...
   ...
ENDSELECT
```

If p and q are equal, then despite the discrimination on the case-selection component of p↑, the selection of the RedComponent of p may be invalid. The detection of this problem, and a similar problem involving shared variables in parallel processing, would require a form of run-time alias checking which we have rejected. (The basis of this rejection is explained in Section X.)

Some illegal cases could be detected more simply. Call a SELECT statement a *simple variant-case selection* if its selection expression is the case-selection component of a non-dynamic variable v of variant record type, and it refers to variant components in its cases. The compiler could check that no assignment to v or an alias of v occurred in any case of a simple variant-case selection.

However, as noted above, this would not fully solve the type-safety problem for variants. It would certainly add complexity to compilers. We are not sure whether avoidance of the programming errors that could thereby be detected would justify this added complexity.

F. *Record Components*

Record components may be either constant (CONST) or variable (VAR). A record component declared variable may be individually changed—i.e., that particular component may be separately assigned a new value. A record component declared constant may not be individually changed and only changes its value when the value of the record of which it is a part is changed. Ironman requirement 3-3F calls for record components whose values are the dynamic values of expressions. Expressions as record components have two interesting ramifications. Since a record component is a variable, and since expressions can contain function invocations, expressions as record components effectively introduce functional variables. Even if function invocations are not permitted in these expressions, many of the problems of functional variables remain—for example, references to nonlocal variables. (Ironman requirement 5D explicitly forbids variables whose values are functions.)

In addition, the use of expressions as record components can be misleading. Programmers normally expect the simple operation of accessing a record component to be very inexpensive. However, if a component is a complex expression, an access can be extremely expensive. It is preferable that any involved computation be clearly recognizable as such from the program text. (The variant-record notation itself hides a certain amount of computation from the user, namely the variant type-checking implicitly supplied by language translators. Therefore, a combination of the variant-record selector field and the dynamic expression field should be especially avoided.) For these reasons, and because our encapsulation facility provides a functionally equivalent though notationally different mechanism, we have chosen not to include dynamic expressions as record components in the language. (See also the discussion in response to Ironman requirement 3-3F in our Analysis of Compliance.)

G. *Files*

The state of the art of language design indicates no agreement on a suitable comprehensive set of standard input and output facilities, either high-level or low-level. We have provided, in Sections VI and XII of the Language Report, a simple high-level facility that will be useful in many applications. Our encapsulation facility, and our low-level machine facilities, are sufficiently powerful to define other file types and other file-manipulation operations. However, many programmers will not be concerned with the details of facilities of this kind, and therefore we have included a particular set in the Language Report despite its theoretical redundancy. As the language enters into common use, libraries of such facilities will undoubtedly develop, and it will be appropriate to control and maintain these on a DoD-wide basis.

H. *Dynamic Types*

Pascal uses the pointer type as the means of obtaining dynamic variables. Pascal dynamic variables are strongly typed and, once allocated, may never be deleted, so they never result in dangling pointers to nonexistent values. For embedded systems, we find the inability to delete dynamic variables unacceptable because it can lead to rapid exhaustion of memory. We therefore permit the deallocation or freeing of dynamic variables. This does permit the creation of dangling pointers, but we view this as necessary in the embedded systems domain. Fortunately, those programmers who wish to use only safe pointers can define completely safe pointer types by encapsulation. For example, consider the module

```
MODULE DefineRpointer
  IMPORT R;
  EXPORT OPAQUE Rpointer;
  EXPORT =, :=, ≠, NewRpointer, DerefRpointer AS ↑;
  REPRESENTATION Rpointer = POINTER(R);
  PROCEDURE NewRpointer RETURNS Rpointer (CONST Init: R)
    VAR Answer: Rpointer;
    NEW(Answer);
    Answer↑ := Init;
    RETURN Answer;
  ENDPROCEDURE NewRpointer;

  FUNCTION DerefRpointer RETURNS R (CONST RP: Rpointer)
    RETURN RP↑;
  ENDFUNCTION DerefRpointer;
ENDMODULE DefineRpointer
```

The user of this module gets an opaque representation Rpointer which, because it comes with allocation and dereferencing operations, differs from the transparent representation POINTER(R) in that no deallocation is provided. (The programmer who wishes explicit control of the storage allocation for pointers will not use the procedure NEW but will instead allocate in specifically provided space—e.g., an array of some appropriate size whose elements ae of type R.)

An alternative to the pointer type for dynamic types is the recursive data type. Recursive data types represent a significant departure from PASCAL and are not appropriate to the types of programming found in embedded systems. In particular, recursive data structures are a way of concealing the implicit pointer that is used in such a data structure. We believe, together with the inventor of the notion of recursive data structures [6], that this kind of implicit pointer is inappropriate in an embedded system programming language.

I. *Sets*

Pascal contains a structured type called a SET. Although the operations on sets are isomorphic to the operations on Boolean arrays, and the most likely implementation of a set will be identical to the implementation of a Boolean array, the set presents a somewhat different abstract view of the objects dealt with and increases the readability of many programs. Viewing a collection of objects as a set rather than as an array is more natural in many cases. Since sets present no new problems in the language, can enhance the readability of some programs, and are already in PASCAL, we see no reason to remove them.

J. *Undefined Values*

Many programming errors would be easier to discover if for each type there were an undefined value. All variables that were not explicitly given an initial value would be set initially to the undefined value. Any function, procedure, or operation applied to an undefined value would raise an exception. Unfortunately, it is not possible to implement this feature efficiently on most existing machines because it requires a special check for the undefined value on each reference to the value of a variable. Some existing machines have hardware that implements undefined values efficiently, but by defining the undefined value in the language we would be forcing all implementations to provide it and thus render most implementations highly inefficient. (The software implementation of an undefined-value—providing an additional bit in the implementation for every variable—can, if required, be provided for program checkout.)

The issue here is that while clearly it ought to be an error to manipulate an undefined value, existing hardware makes this kind of error impossible to detect efficiently. In consequence, this becomes one of the few areas of the language where we cannot forbid undesired behavior because we cannot effectively detect it.

K. *Manifest and Scope Entry Constants*

The definition of manifest constant given in the Report involves the phrase "certain other expressions guaranteed to be evaluated during translation." We have used this wording so as not to restrict the flexibility of writing type and representation definitions that are dependent, for example, on machine configuration constants determined by means of environmental inquiries. There are many ways in which translators might be sophisticated in the ability to evaluate manifest constants. For example, a translator might be able to determine that even though an expression involves user-defined names, it is translation-time evaluable. The advantage of the definition of manifest constants given in the Report is that a particular translator can be made as powerful as appropriate for an implementation in this regard. The disadvantage is that it can lead to nonportability, since the definition of manifest constant is translator-dependent. To avoid this problem, we have included a standard definition of manifest constant in the language definition.

Note that our definition of scope entry constant covers expressions not evaluable during translation—in general, not evaluable until scope entry. Because arrays returned by functions may have scope entry ranges, it follows that we have provided greater flexibility than contemplated in Ironman requirement 7D. We do not believe that any inefficiency results from this flexibility and have therefore preferred not to adopt a special exclusion of this case.

L. *Cross-Type Routines*

The Language Report states that all standard operations are automatically extended to apply to a new type. Thus, as soon as the user declares

```
TYPE NewInteger = INTEGER;
REPRESENTATION NewInt = NewInteger [-10000..10000];
VAR a, b, c: NewInt;
```

the arithmetic operations are available for manipulating NewInts, and operations such as a+b and a-c*d are permitted. This automatic extension does not, however, suffice, to define cross-type operations—e.g., an addition operation applicable to one INT and one NewInt. Such cross-type extensions of routines must be declared using the overloading facility, but this does not suffice. The difficulty may be seen in this example by trying to write the body of the function whose header is

```
FUNCTION EXTENDS + RETURNS INT (CONST x: INT, y: NewInt)
```

There is no convenient way to turn x and y into a pair of objects of the same type, so as to add them with one of the addition functions provided by the language.

Various solutions to this problem have been proposed in the literature. Mesa [27] takes the position that no new routine needs to be declared, because the conversion between INT and NewInteger is a *free coercion* provided automatically. This solution violates Ironman Requirement 3C.

Ell [9] provides routines LIFT and LOWER in its analogous situation to convert explicitly between the types. The Ell facility requires a great deal of skill for proper use—the failure to include a required call on LIFT or LOWER is a common programming error. Moreover, this solution requires that types be used as explicit parameters of LIFT and LOWER, forbidden by Ironman Requirement 5D.

We believe that the need to define cross-type routines is rare. Indeed, the difficulty that we have been discussing only arises for standard language types of which the programmer declares "distinct" versions. In this situation, if the cross-type routine is needed, we recommend that NewInt be defined not as above but instead by

```
REPRESENTATION NewInt = RECORD VAR Value: INT ENDRECORD
```

and that the extended addition be defined as

```
FUNCTION EXTENDS + RETURNS INT (CONST x: INT, y: NewInt)
  RETURN x+y.value;
ENDFUNCTION
```

Note that no more space is required for this NewInt representation than is required to represent an INT. Nor is any more computation required to access the value y.value than is required to access the value of a simple variable. The use of the record here is simply a notational device—it solves the problem cleanly and with no addition to the language.

VII SIMPLE DECLARATIONS

A. *CONST and VAR Declarations*

Our notations for CONST and VAR declarations are quite similar. In particular, both require specifying the representation of the constant or variable as part of the declaration. The information provided in a CONST declaration might at first be thought redundant since an initializing value is present and implicitly contains needed representational information. However, some representational attributes—such as the range of a scope entry constant—cannot be deduced by a translator. Therefore, we have decided to adopt a simple uniform, but sometimes redundant, rule: representations are required for constants as well as for variables.

Since assignment to constants is forbidden, the only way to set the value is in the declaration, and we require that the value be included in a CONST declaration. However, this argument does not apply to variables, and we have made the initial value for variables optional. We considered a different design choice here: there is no possibility of uninitialized variables if initialization in the declaration is mandatory. But we believe that, more often, a more complex initialization will be needed than can be made in the declaration of the variable. In these cases, a mandatory initialization in the declaration is superfluous and hence wasteful. This possibility of waste is more important than considerations of the errors caused by uninitialized variables.

It is important to appreciate the degrees of freedom available to the programmer in specifying the representation in a constant or variable declaration (and in formal parameter declarations). The most common situation will be that the representation is a RepresentationId—that is, an identifier associated with a representation by a prior declaration (see below). In this case, the declared entity will have the same type and realization as any other entity whose declaration uses the same RepresentationId.

If the representation in a declaration is not given by a RepresentationId, then it is given by a pair: an abstract type which is optional, and a concrete realization of that abstract type (called a ConcreteType syntactically), which is mandatory. This form of representation is used in a declaration to obtain alternative representations of an abstract type. The various possibilities are illustrated in Section VII of the Language Report.

B. *Type and Representation Declarations*

We provide the possibility of defining and using abbreviations of both types and representations. We argue that this does not represent gratuitous complexity in the design, but is mandated both by the Ironman Requirements and pragmatic considerations. First, Ironman

requires in the language

- The notion of type,
- A facility for associating an identifier with a type—the type declaration,
- That two type identifiers introduced by distinct type declarations be regarded as denoting distinct types.

The first two of these points are consistent with Pascal and all of its descendants—e.g., LIS, Euclid, Mesa, and Gypsy—and clearly must be satisfied by the new language. The third point is more difficult, and different descendants of Pascal have taken different positions; the general form of these is that two distinct type identifiers may denote the same type if their two definitions satisfy some language-specified criterion (e.g., structural isomorphism). We have considered a number of schemes of this kind and conclude that all of them needlessly complicate translators and detract from program readability. Therefore, our type equality rule is exactly as prescribed by Ironman (Requirement 3C).

Ironman requires an additional degree of freedom—the possibility of defining more than one physical representation of a type and associating identifiers with such representations (Requirement 11B). This is clearly desirable as a programming convenience, and, moreover, is essentially the facility provided by the so-called type declarations of Pascal. Given the simplifying effect of the strict type equality discussed above, we believe it essential not to reintroduce complexity through the rules on physical representations of types. Moreover, we believe that the facility provided by Pascal is what is needed in most programming—that is, the possibility of associating an abstract type with a single physical representation so that equality of representations is the same as equality of types.

However, the Ironman requirement of multiple physical representations of a type, combined with the requirement of implicit conversion between two physical representations of the same abstract type, creates the issue of how, in general, to determine whether two entities with different physical representations share the same abstract type. We have proposed a design that satisfies this combination of requirements, is simple in the common case, is straightforward and easy to use in the general case, and is readable and efficiently implementable in all cases. We do this by providing two parallel syntactic structures:

- The ConcreteType, which specifies abstract and physical information together, and
- The Type, which includes only the abstract information.

Our representation declaration then permits a Type and ConcreteType to be associated, so that several ConcreteTypes can, in general, share the same Type, and this can be efficiently detected by translators and easily perceived by human readers.

C. *Recursive Declarations*

There are many kinds of simple and composite declarations. In some—e.g., procedures—recursion is meaningful. In these cases, it is not possible to require that an object be defined before it is used, since there may be an inherent circularity. Thus, one design alternative is to distinguish recursion syntactically. For example, suppose p and q are mutually recursive procedures. Then one might use a syntactic scheme such as

```
BEGIN
   PROCEDURE p RECURSIVE;
   PROCEDURE q ...formals for q here...
     ...body for q here including call of p...
   ENDPROCEDURE q;
   PROCEDURE p ...formals for p here...
     ...body for p here including call of q...
   ENDPROCEDURE p;
   ...rest of the block...
END
```

to emphasize that the mutual recursion is deliberate. It is natural, when using such a scheme, to base it on a semantics in which declarations are viewed as "executed" in the sequence they appear, in the same way as are statements. The argument for this approach is that it is easy to explain; further, the syntax emphasizes an aspect—recursion—that is somewhat expensive and hence worthy of emphasis.

The problem with this approach is the effect if the recursive attribute is omitted by mistake. In the example, the translator detects an error if there is no other definition of p. But if there is an enclosing definition of p, then the program with the mistakenly omitted attribute is semantically valid but has a different meaning: its q calls a different p.

An alternative is to view the declarations in a block as occurring in parallel. Thus the ordering of declarations in a block does not affect the semantics and, assuming p and q are mutually recursive, as well as declared together as above, the presence or absence of an outer p does not matter. We have decided that these considerations and the success of a parallel declaration semantics in Pascal and Algol 60 are persuasive. However, we wish to forbid arbitrary orderings of declarations where recursion is not meaningful and the possibility of use before definition would complicate compilation and detract from readability. We have argued above that the only recursive types appropriate in this language will introduce the recursion with pointers. Therefore we require simple declarations to precede the use of the defined entity except in the case of pointers, where a pointer type (or representation) declaration may precede the declaration of the pointer's base type. A more complex situation arises with composite declarations, where forward reference cannot always be avoided. For this case we adopt the parallel-declaration semantics and place on the translator the burden of finding a consistent ordering.

VIII EXPRESSIONS

Our syntax for expressions is that of Pascal, except as noted below. We have tried to include the specific aspects of expressions required by Ironman in a way that is consistent with and as simple as Pascal. None of the details, however, are intrinsic to our design, so that cosmetic modifications will be straightforward.

A. *Free Operators*

We have included a number of free operator symbols in the language: infix operators for which we define the precedence and parsing properties but for which we do not define the semantics. The user may then ascribe meaning to these operators by function and procedure declarations.

We have included this facility as a user convenience in the belief that, properly used, it will contribute to program clarity by allowing the infix notation with user-defined types and routines. There are several alternative possibilities. One is to permit the user to define new infix operators, but this is forbidden by Ironman. A second is to provide only those operators for which the language provides a semantics; however, in the presence of the required generic extension facilities, we believe that this would lead to user overloading of a few standard operators with many definitions, thus reducing program clarity. A third alternative is to forbid generic extension of operators, but this would be extremely restrictive—it is exactly such operators as "+" and "*" that are best explained in terms of generic extension. Thus we choose the first alternative—the provision of a limited number of "free operators" to which the user gives meaning. (In fact, it might be reasonable to forbid user overloading of standard operators, as a means of increasing program readability, but such a restriction would violate Ironman Requirement 7A.)

B. *Association of Operators*

The association of operators within precedence levels is a controversial issue, and an issue on which we cannot wholly agree with Ironman. Most conventional languages specify the association rules for the standard operators, and, in the absence of mechanisms for creating new operators, that ends the matter. (One language, APL, adopts the radical simplification of making association right to left and giving all operators the same precedence.) Some experimental extensible languages, for example Algol 68 and EL1, adopt a completely general course: the user can define new operators and specify all their parsing properties, including association. Ironman dictates an intermediate course: the user cannot create new operators (Requirement 2C), but can specify the association of existing operators (Requirement 7D); the language

translator must detect an error whenever an expression omits parentheses in a context where a nonassociative operator occurs to the left of an operator of the same precedence (Requirement 4G).

There are a number of difficulties with this combination of requirements. It is not feasible for translators to verify the mathematical correctness of user specifications of the association of overloaded operators, so the user specification of associativity can be misleading. Moreover, the notion of associativity cannot be captured by classifying individual operators as "associative" or "non associative"—for example, XOR and OR are both self associative but not jointly associative. That is, in general,

e1 XOR (e2 XOR e3) = (e1 XOR e2) XOR e3, and
e1 OR (e2 OR e3) = (e1 OR e2) OR e3, but
e1 OR (e2 XOR e3) ≠ (e1 OR e2) XOR e3

(For the inequality, consider the case where e1, e2, and e3 are all TRUE.) A mathematically sound notion of association, to comply with the spirit of the Ironman requirements, must explain the notion of "association of an n-tuple of adjacent operators." With such a notion, Ironman could meaningfully be modified to the requirement

> Explicit parentheses shall be required to resolve the operator-operand associations wherever the occurrence of an adjacent pair of operators of the same precedence makes the value of the expression ambiguous.

However, this form of the requirement could still not be feasibly enforced by translators and would not, in our view, make programming less error-prone.

Faced with these difficulties, we have adopted a compromise scheme. We classify operators as associative or nonassociative in the language definition and provide no facility for the user to specify association of operators. We classify all free operators as nonassociative and classify the standard operators according to the mathematical properties of their standard definitions (except for XOR, which is classified as nonassociative to avoid the problem cited above). This compromise does not resolve the problem of user overloading of the standard operators with definitions having nonstandard associations. This compromise does detect what is probably the most common error of this kind—the ambiguous expression e1/e2*e3 is forbidden in our syntax which permits only (e1/e2)*e3 or e1/(e2*e3).

C. *Precedence of Operators*

Ironman restricts the number of operator precedence levels that may be included in the language (see Requirement 4F). We have complied with this requirement. Our syntax provides five precedence levels, exactly as in Pascal. We note, however, that the designer of Pascal and the designers of Euclid (see [19] and [8]) have stated that it is better to have more precedence levels so that the logical and arithmetic operators can be segregated. We recommend that this requirement be reconsidered.

D. *Uniform Reference to Data Structures*

It has been noted by Geschke and Mitchell [12] that one source of the cost of changes in large software systems is that programs that access a data structure do so using notations determined by its exact representation. Our design in large measure avoids this problem by means of the notion of *opaque exportation* (see Section X of the Language Report and Section X of this Design Discussion). Geschke and Mitchell propose that uniform notations be made available more generally—e.g., by making A[i], A(i), and A.i synonymous and dropping all but one or

two of the then redundant notations. This scheme can be implemented in our strongly typed language, since the appropriateness of subscripting, function application, or record component selection can be determined without the notational clue. However, in accord with the basic principle that departures from Pascal should be avoided unless required by Ironman, we have not adopted this notational simplification.

E. *Type of Expressions*

Ironman requirement 4B states that "it shall be possible to specify the type of an expression explicitly". The requirement goes on to state that this is not intended as a mechanism for type conversion. We have provided an alternate mechanism to resolve ambiguities in the type of a literal. The programmer can, in our language, explicitly check the type of a result by embedding it in a call of a value-returning procedure such as:

```
FUNCTION CheckDesiredRepr
   RETURNS DesiredRepr (VAR a: DesiredRepr)
   RETURN a;
ENDFUNCTION CheckDesiredRepr
```

Note that, since our language is strongly typed, the call of this procedure serves only as a kind of "compile-time assertion"—it will produce a compile-time error if and only if the actual parameter has the wrong representation and need not produce run-time code.

IX STATEMENTS

A. *Assignment*

We have provided a conventional assignment statement, whose left-hand side is a single, possibly composite, variable. An alternative is to introduce a multiple assignment, an example of which is

(x, y, z) := (z, y, x)

In order to give power to this multiple assignment, one would also permit

(x, y, z) := r, or

r := (x, y, z)

where r is a composite object with the specified number of components, agreeing in type, respectively, with the list of variables. Also, one would permit functions and value-returning procedures to return multiple values, by allowing the RETURN statement to take more than one argument, and use these returned multiple values in the same ways.

The advantage of such a scheme is the convenience of the notation. If it were added to our language, one could simultaneously simplify by deleting procedure output parameters, since the present

```
VAR ay:FLOAT; az:Complex;
PROCEDURE p(CONST x:INT; OUT fy:FLOAT; fz:Complex)
  ...computation setting fy and fz...; RETURN;
ENDPROCEDURE p;
p(17, ay, az)
```

would be replaced by

```
VAR ay:FLOAT; az:Complex;
FUNCTION p RETURNS FLOAT, Complex (CONST x:INT)
  ...computation with locals...; RETURN e1, e2;
ENDPROCEDURE p;
(ay, az) := p(17)
```

On balance, we decided that the complications introduced by multiple assignment, if only in explaining this unconventional facility, were not sufficient to warrant its introduction. It remains, however, an alternative worthy of consideration in future languages.

B. *Assertions*

We have provided an assertion statement that exactly complies with the relevant Ironman Requirement (10F); in fact, it is defined as equivalent to the conditional raising of a specific exception (see Section IX of the Language Report). We considered a number of more general uses of the assertion, suitable for guiding compiler optimizations or for facilitating the use of the language with mechanical proof-of-correctness systems. However, we concluded that these generalizations, while worthy of study and promising for the future, would be inappropriate in a language intended for immediate production application.

C. *GOTO Statement*

Our language adopts the same attitude toward the GOTO statement as does Ironman: that it is generally not the best way of structuring a program; that a number of clearer control structures should be available so that the GOTO is usually avoidable; and that, in view of the intended audience for the language and the belief that there are occasional good uses for GOTO statements, it would be too radical a step to entirely prohibit them.

We have, however, restricted the GOTO somewhat more than Ironman contemplates. Clearly, it is not possible to "go" where there is not a label, and we permit labels only (at the top level) in the syntactic construct "Block" and not in the construct "UnlabeledStatements". The latter form is used for statement sequences in conditionals and repetitions—indeed, in all composite statements other than the BEGIN-END statement. It follows that it is not possible to transfer control (at the top level) within these statements. We have made this decision on the grounds that the use of the conditional or repetition statement is the programmers means of indicating, both to a language translator and to a human reader, the intent of employing a particular control discipline.

We think it unnecessary to allow a programmer to mislead readers by using one of these composite statements and then subverting the corresponding control discipline by means of a GOTO. For example, a construct such as

```
IF b1 THEN s1; GOTO mergePoint;
   ELSIF b2 THEN s2; mergePoint: s3;
   ELSE s4 ENDIF
```

can be replaced by

```
IF b1 THEN s1; s3;
   ELSIF b2 THEN s2; s3;
   ELSE s4 ENDIF
```

with an increase in program readability. (If s3 is very long, considerations of program readability suggest that it should be packaged as a procedure call, possibly inline.)

D. *EXIT Statement*

We have provided an EXIT statement for completing the execution of the LOOP repetitive statement. The EXIT may be used only to leave a loop (although it can be used to leave loops from nested inner statements), and must name the loop with a loop identifier.

One alternative is to forbid multiple-level exits—the course adopted by the designers of Euclid. They decided that the semantics of the construct were too complex to warrant its use. We disagree: we believe the semantics, including the formal semantics, are quite straightforward and, moreover, believe that the multiple-level exit arises naturally and often in practice, so that it is unreasonable to force the programmer to do without it.

Another alternative is a more general exit construct, capable of leaving any structured statement. We decided that for the other structured statements—e.g., the WHILE and FOR repetitions and the conditionals—the possibility of an explicit exit needlessly complicates the semantics and leads to obscure programs. For example, the semantics of

```
FOR i: INTEGER [1..10] DO ... ENDFOR
```

ought to be simply that when control reaches the point just after the ENDFOR, the specified statements will have been executed ten times, and not that they will have been executed ten times unless an EXIT occurred first. The loop statement has no such conflicting connotations, and we therefore restrict exits to loops.

A remaining issue is whether to label exit statements. We find the unlabeled exit confusing: one has to count "END" brackets to determine the control flow. Instead, we require all LOOP statements to be named at the lexical start and end of the loop. This is a helpful redundancy, and means that, when an EXIT statement occurs, one need only look ahead for an ENDLOOP with the appropriate name.

E. *Propagation of Exceptions*

This statement is provided in response to Ironman Requirement 10C, which is thereby satisfied. The detailed semantics of this statement are discussed, together with those of exception handling, below and in Section X of the Language Report.

F. *Selection*

An obvious generalization of the SELECT statement is to allow expressions other than manifest constants to designate the alternatives. This would prevent the compiler from guaranteeing that the cases were disjoint. It would either prevent the efficient implementation of a SELECT statement as an indexed jump, or else would introduce into the translator the complexity involved in detecting the efficiently implementable case. (It would also mean that the programmer might misunderstand the run-time complexity of a particular selection.) Therefore, we retain the Pascal restriction to manifest constants.

G. *Repetitive Constructs*

We have provided three repetitive constructs: a FOR, a WHILE-DO, and a LOOP. The last is the most general and can be used to synthesize either of the others. It can also be used to synthesize the REPEAT-UNTIL repetition, which we do not include as a primitive. The presence or absence of any one of these special cases is not critical; our choice attempts to optimize the tradeoff between including a clutter of primitive statements and forcing programmers to use overly general forms. This choice can be amended in the light of further consideration.

H. *WITH Statement*

Pascal provides a WITH statement that is used to abbreviate access to components of records. For example, if *r* is a variable of a record type with components *a*, *b*, and *c*, then the program text

```
y := r.a; z := r.b; w := r.c
```

can be abbreviated using the Pascal WITH statement to

```
WITH r DO BEGIN y := a; z := b; w := c END
```

Slightly different forms—in both syntax and semantics—are found in Euclid and Modula. We believe that all varieties suffer a common handicap—they encourage the writing of very confusing and error-prone code. Also, they complicate the machinery needed in translators to deal with aliasing (see below). The most common application of the WITH is the initialization of records which, in our language, can be done by assigning a structured denotation to a record variable (see Section VI of the Language Report). Therefore, we have omitted the WITH statement.

I. *BEGIN-END Statement*

Our BEGIN-END statement is essentially that of Algol 60. This form of the BEGIN-END, which introduces its own local variables, was not included in Pascal. However, Ironman rightly recognizes the need for such a facility, and, by including the Algol 60 BEGIN-END, we comply with Ironman Requirement 5C—that it be possible to lexically embed scopes.

Our BEGIN-END statement has a SIMULATED_TIME option. This is discussed below and in Section X of the Language Report.

X COMPOSITE DECLARATIONS

A. *Functions and Procedures*

1. *Overloading*

Ironman mentions, as a way of extending existing routines to apply to new argument types, that "functions and procedures may have the same name if they differ in number of parameters or in formal parameter types" Ironman (requirement 7A). That is, duplication of names of routines is permitted, and indicates an overloaded definition. Thus if f is already declared as a function on a single integer argument, the declaration of a function f of a real argument yields an f that is said to be overloaded. This function F may then be called with either argument type and the appropriate definition is invoked.

We have decided that this notation would be error prone: there would be no hint of accidental naming conflicts. This is especially dangerous in a language intended for programming projects that will involve many programmers and the assembly of separately written modules. Therefore, we provide explicit syntactic indicators of overloading. To add a new definition of a routine, we use a form of declaration including the keyword EXTENDS. To redefine an existing case, we use the keyword REPLACES. If neither keyword is used in a declaration, then that declaration hides all routines with the same name. (The translator can issue a diagnostic message to warning the user of a possibly unintentional conflict.)

Our facility allows overloaded cases of an overloaded routine to be distinguished by the number and types of the arguments, as required. Note, as a consequence, that if an existing routine acts on one representation of a type, then it is not possible to extend it to act on a different representation; the existing definition must be replaced. (For example, if the representations are numeric ranges, then the replacement might be a definition covering the combined range.) We feel that overloading based on representations produces a great deal of complexity in implementation, ambiguous situations, and insufficient benefit in programming. We have therefore not generalized beyond the letter of this Ironman requirement.

One can similarly conceive of functions that are overloaded not only in their argument types, but also in their result type. Thus f(1) might invoke one definition of f in a context requiring a floating-point result and another if an integer were required. We have decided that the confusion and complexity that result from this kind of overloaded definition are excessive. For example, if f is called in a context where either an integer or a floating-point number is appropriate—as may be the case in the context g(f(1)) where g is overloaded—additional rules are needed for disambiguation. We provide no such facility.

2. *Formal Parameters*

We have complied with the Ironman requirement for three formal parameter classes: input (i.e., CONST), input-output (i.e., VAR), and output (i.e., copy out). We did, however, consider several other possibilities.

The first is to use a 'copy on input' in place of the CONST rule; following Algol 60 terminology, this class would be VALUE. The merit of this class, relative to CONST, is primarily a different point of view. In the case of CONST, the language rules forbid modification and it is therefore usually correct to implement CONST parameters by reference; however, translators must check that no CONST parameters are modified, which may be quite expensive in some cases. With VALUE parameters, on the other hand, the language does permit modification of the formal parameter, and therefore implementations must, in general, copy the parameter. Note that, given CONST parameters, the user desiring to assign to the parameter could introduce an appropriate local variable, initialized to the CONST parameter, and modify that freely. We therefore decided to include CONST, rather than VALUE, parameters in our language.

A more difficult dilemma arises with VAR parameters. Here, the intention is explicitly to permit modification of the formal parameter, and the semantics are that this modification is to be reflected in the same modification being made to the corresponding actual parameter. It is clear that VAR parameters can feasibly be implemented only by reference. The difficulty is that they introduce a scope—the call of the procedure—in which there are two names for the same entity—the formal parameter and the actual parameter. Thus, deleting the VAR class would greatly simplify the enforcement of the aliasing rules. An alternative, which avoids these complications, is to replace the parameter class VAR with a class whose semantics are to copy on input, allow modifications in the procedure body, and copy the modified value into the actual parameter on exit. This class, which might be named COPYIO, does not produce any scope with two names for the same entity, and is thus semantically simpler than VAR. However, we decided that the VAR class is too useful in practice to allow us the luxury of eliminating it. Note also that the user can always achieve the effect of COPYIO by passing the same actual parameter to a CONST formal parameter and an OUT formal parameter of a procedure.

In summary, we considered replacing CONST with VALUE, and also VAR with COPYIO. Though a case can be made for either or both replacements, we have decided it is best to make neither replacement. We believe that it is very important, if these decisions are reconsidered, not to increase the complexity of the parameter mechanism. This means, in particular, that any change that adds the COPYIO or VALUE class must also delete, respectively, VAR or CONST.

We have complied with Ironman requirements so far as the provision of formal array parameters for routines. The actual array types of the language include the array length (in each dimension), in addition to the dimensionality, index types, and base type, as part of the type; therefore, if the array parameters of routines are typed as actual arrays, then the rules of strong typing make it impossible to define a routine independent of the actual array length. This would lead to an unacceptably awkward programming style, and so, in compliance with the Ironman requirements, we introduce a special array type: the length-unresolved array type, which may be used only to type formal parameters and permits the definition of routines independent of the lengths of their array parameters.

A number of issues accompany the inclusion of formal array parameters in the language. The first is the question of what should be the index type of a procedure with a formal array parameter, given that actual array types may be defined with any integer or scalar range as index types. Noting that the use of an indeterminate-length array parameter in a routine implies the possibility of any length array in a call, we have decided that the only reasonable

programming style for implementing such routines is to assume integer indices matching any integer or scalar range index type.

A second issue is whether to allow general use of formal arrays as components of other formal types—for example, a record consisting of an integer and a formal array. There would be a great deal of implementation complexity, involving access to components of general composites with indeterminate-length components, and thus our decision has been to allow only the basic formal arrays already discussed.

3. *Introduction to Naming and Aliasing*

The question of whether to forbid aliasing in the language, and if so how, is a very difficult one. On the one hand, the presence of aliasing in a language results in complex procedure call semantics and in the possibility of the very subtle programming errors that can arise from overlapping of the VAR parameters and free variables of a procedure. On the other hand, it appears that aliasing errors, while troublesome in theory, are extremely rare in practice. We are hesitant to shape the design of a production language in a way that no previous production language has been shaped, to achieve what may be only an academic nicety, since there is doubt that the exclusion of aliasing would solve a genuine problem. Moreover, many of the approaches to restricting aliasing result, as will be seen, in significant costs: descriptive inadequacies in the language, complexities in program translation, or major run-time overheads. Our task, therefore, has been to design a set of rules in this area that avoids overheads disproportionate in terms of the benefits to be achieved.

We have recently received word of new work in this area by Reynolds [13]. Reynolds introduces a formal system in which aliasing (and, more generally, *interference* between program parts) and various restrictions to avoid it can be precisely described. This work has not yet addressed such programming language constructs as arrays or recursive procedures and therefore cannot yet account for the interaction of aliasing with all of the language features required by Ironman. We are pleased, however, that insofar as references to free variables by procedures are concerned, the rules proposed by Reynolds are quite similar to those proposed below.

Names can be associated with program entities in several ways. These include the declaration of local variables of BEGIN-END statements and modules and the declaration of formal parameters of processes, procedures, or functions. Some of these can introduce *aliasing*, that is, the occurrence of program scopes in which (1) some entity has two distinct names and (2) the entity can be modified, using one of its names, to produce a change discernible through another name.

For example, if the same variable is used as an actual parameter corresponding to two distinct VAR formal parameters of a procedure, then this constitutes aliasing. Use of the same actual parameter corresponding to two CONST formal parameters is not aliasing, since neither CONST parameter will be modifiable within the procedure. The situation is less clear when the same actual parameter is passed to one CONST formal parameter and one VAR formal parameter. If the implementation suffers the expense of detecting this situation, and responds to it by passing a copy of the CONST parameter, then there is no aliasing, since a change to the formal variable is not discernible in the (copied) formal constant. But if the implementation does not try to detect this—which might involve examining separately compiled calls of a procedure during linkage loading—and also uses the efficient implementation of not copying constant parameters, then we do have aliasing.

Aliasing also arises when a variable is passed as an actual to a procedure's VAR formal parameter and is referred to freely by the procedure body: this may be understood by thinking

of an equivalent program in which each reference to a free variable is transformed into a reference to a new VAR formal parameter to which the free variable is passed explicitly at each call. Observe that combinations involving a pair of OUT formal parameters, an OUT parameter and a VAR parameter, or an OUT parameter and a free variable, do not constitute aliasing: the actual parameter corresponding to an OUT parameter is not changed within the scope of the procedure body so there is no scope in which changes to the actual and formal are linked. (Using the same actual variable corresponding to distinct OUT formal parameters is an error, but it is not aliasing.)

There are sources other than aliasing of the ills to which aliases are prone: pointers and variables shared by processes. Whatever restrictions may be imposed to limit aliasing of the kinds discussed above, it is clear that the controlled creation of sharing patterns is a major purpose of pointers and processes, and loss of descriptive adequacy arising from restrictions here would be intolerable. With aliases, an assignment to a variable y intervenes between the observation of the value of a different variable x and the observation of a change in that value. If pointers are involved, the user must realize that an assignment to a variable referred to by a pointer may also change the values of other variables referred to by pointers of the same type. And with shared variables, change may be due to an assignment in some other process and not correspond to any program text at all in the process where the change is perceived.

Several resolutions are possible. The most straightforward, and that with which there is most experience, is to ignore the issue on the grounds that no problem arises in practice. However, this possibility is explicitly rejected by Ironman.

A second alternative is to provide very strict alias checking and to guarantee that this checking is easy for a translator to do. A major source of complication in alias checking, as will be seen below, is the use of subscripted arrays as actual parameters corresponding to VAR formal parameters. A first step in making all aliasing economically detectable by translators would therefore be the prohibition of subscripted arrays (and fields of records) as VAR parameters. We considered designs along these lines and decided, on balance, that the untested benefits of alias checking did not compensate for the severe changes in programming style that would result.

A third alternative is to provide strict alias checking, unaccompanied by language restrictions such as those just mentioned. This immediately raises the problem that some aliasing situations will not be detectable by translators. For example, if i and j are program variables, it usually isn't known until program execution whether a[i] and a[j] are aliases. Similarly, if p and q are pointers, the aliases of the anonymous variables to which p and q point are often not known until execution and indeed will vary during execution. It follows that, to implement this kind of aliasing design, it is necessary to bear the cost of checking in part during program execution. We have decided that program execution costs of this kind cannot be justified—apart from the untested practical benefits, this would yield a situation where two apparently similar procedure calls p(a,b) and p(c,d) had drastically different cost because only one required execution-time alias checking.

These considerations lead to a fourth alternative, which is substantially in compliance with Ironman and which we have adopted. This alternative is an aliasing design that incurs no execution-time overheads, moderate translation-time overheads, and minor restrictions on programming style. We now describe this design.

It is clear that if v is both passed to a VAR formal and referenced freely, this must be detected as illegal. Thus the procedure call in

```
BEGIN
  VAR v:BOOLEAN;
  PROCEDURE P(VAR x:BOOLEAN)
  ...body referring to v...
  ENDPROCEDURE P;
  P(v)
END
```

creates a scope in which v and x are used to name the same entity and the procedure call is therefore illegal. A more complex, and also illegal case, is

```
BEGIN
  VAR av:ARRAY (INTEGER [1..2] OF BOOLEAN)
        := aSuitableInitialization();
  PROCEDURE P(VAR x:BOOLEAN)
    ...body referring to av[1]...
  ENDPROCEDURE P;
  P(av[1])
END
```

since it has a scope where x and av[1] are the same entity. Similarly, if av is replaced by rv—a record with a BOOLEAN component c—and av[1] is replaced by rv.c, we have a scope where x and rv.c are identical and hence a forbidden case. Replacing av by pv—a pointer to a real—and av[1] with pv↑ yields a scope with x and pv↑ identical and the same problem. In these cases, a free reference (respectively) to av, pv, or rv would be just as bad: a change is discernible in the value freely referred to even though only the formal is explicitly modified. (Note that it is not worthwhile to detect and permit the special subcase where the formal parameter is not modified—if the user wants to alias on this basis, he ought to be using a CONST formal.) Also, if we switch the roles played by the actual parameter and the free reference, we still get illegal cases: with actual parameter av and free reference to av[1], or with actual rv and free reference to rv.c, or with actual pv and free reference to pv↑. And, finally, if we have not P but a procedure Q with two appropriately typed VAR formal parameters—even one that makes no free references—then the calls Q(av, av[1]), Q(rv, rv.c), and Q(pv, pv↑) are all forbidden.

If e1 and e2 are not manifest constants, then we consider av[e1] a possible alias of av[e2]. Of course it might be determinable that, even though e1 and e2 are not manifest constants, they will be unequal at run time. However, to try to allow this case is an invitation to avoidable compiler complexity and we forbid it. (It might be worthwhile to provide a loophole through programmer assertions—one could treat "ASSUME NOT e1=e2" as permission to assume nonaliasing, and treat "ASSERT NOT e1=e2" as permission for the compiler to embark on the theorem-proving task of demonstrating nonaliasing. The ramifications of such a loophole are far from obvious, and it does not seem appropriate for the present language design.)

For many purposes, pointer dereferencing can be understood as indexing into an array or *collection* (the Euclid term) of anonymous variables. That is, if p and q can point to a common collection Z of anonymous variables, then it is helpful to consider p↑ and q↑ as abbreviations for Z[p] and Z[q]. (We hasten to point out that only the former notation is allowed and the latter is merely explanatory.) The rules for pointers and pointer dereferencing are similar to those for arrays and array dereferencing. The differences are

- That because Z never appears explicitly in our notation, the aliasing between Z and Z[p] does not arise,

- That because there is no such thing as a manifestly constant pointer, Z[p] and Z[q] are always aliases, and

- That p is an alias of Z[q]—assuming that p and q are elements of the same pointer type—even though the analogous array case with i and A[j] is perfectly legal. (The reason for this restriction is that our language considers the Z to be an implicit parameter whenever the p is an explicit parameter. Hence passing p and q↑=Z[q] as VAR actual parameters is the same as passing p, q↑, and Z—corresponding in the forbidden case with arrays of passing i, A[i], and A to VAR formals.)

Note that Euclid views pointers somewhat differently than do we: Z can be explicitly passed as a parameter, and must be available as an implicit argument to the dereferencing operation. Therefore p and q↑ need not be viewed as aliases in Euclid unless Z is present as well.

We impose no aliasing restriction on p and q, even though they are elements of the same pointer type. This is because we believe that the purpose of pointers is to permit sharing relationships and it would not be helpful to restrict this possibility. (Formal proof of the properties of programs that use such facilities is quite difficult, and will generally require that the sharing relations be demonstrated as lemmas, but that is another matter.) We impose some order on the sharing possibilities by not providing high-level primitives to produce pointers to an existing entity. There is a low-level and implementation-dependent primitive, MAKE_POINTER, that converts a machine address to a specified pointer type, but this is normally used only in specialized "allocation encapsulations" whose authors must take great care to avoid the creation of unexpected aliasing relations, e.g., sharing relations between two pointers of different pointer types or between a pointer and a nonpointer.

If c1 and c2 are different, then r.c1 and r.c2 are different—a change to one does not affect the other. Thus either may be referred to freely while the other is a VAR actual parameter, or both may be VAR actual parameters. Similarly, if e1 and e2 are expressions that are different manifest constants, then av[e1] and av[e2] are not considered aliases.

Note that to tell whether there is aliasing with the above rules, one may have to look at a procedure definition P and at every call of P. And in looking at the definition, one must look at the definitions of all the procedures Q called from within the procedure, since an alias-producing reference to a free variable might be made by one of these inner calls. Thus the computation of a transitive closure may be needed to determine whether there is aliasing or not. This is especially onerous, since the P's and Q's can be textually widely separated—e.g., in different modules or separate compilation units.

However, there is a nice compromise. Call a module M "normal" if, whenever it imports or exports a variable v, it does not either (1) export any procedure that freely refers to v, or (2) pass v as a VAR actual to any procedure that it imports. Then, we can guarantee that there is no aliasing in M with a transitive closure computation that considers only the local procedures of M. We can, mutatis mutandis, define a normal separate compilation unit and guarantee that it has no aliasing without considering any other separate compilation unit. Whether abnormal modules are permitted then becomes an administrative and not a language-design decision, to be made by the manager of a project using the language. Or, it may be appropriate to permit abnormal modules but insist on normal separate compilation units; this will complicate compilation somewhat but at least it will not complicate linkage loading. There are undoubtedly unusual cases, but, in general, the programmer who needs to export both a variable and a procedure referring to it has probably not decomposed his program into modules very well, so the normal case is not only easy to check but is also good programming style.

4. *Pure and Impure Functions*

Ironman requirement 7E states:

> During execution of a function, assignment to a variable that is nonlocal to the function shall be permitted only where the variable is own to an encapsulation that does not contain any call on the function.

We have adopted a stronger rule and forbidden assignment in a function to any nonlocal variable. With this strong rule, it is possible for compilers to optimize the ordering of function calls within an expression. The weaker version does not achieve this effect. Suppose that a function f assigns to a nonlocal a, and that a is own to an encapsulation M. Ironman requirement 3-5C defines "own to an encapsulation" as meaning "accessible only within an encapsulation." Therefore, we must assume that f is also local to M. Since f may access as well as assign to a, the effect of "f(1)+f(2)" can depend on the order of evaluation of the two calls of f. Moreover, the effect of "f(1)+g(2)" can depend on the order of evaluation of the operands of "+" if the execution of g can result in an invocation of f.

We appreciate that the need for descriptive adequacy in the programming language does demand a function-like construct that permits assignment to nonlocals. We have therefore included in our language, in addition to functions obeying our strong rule, the *value-returning procedure*. The value-returning procedure is used syntactically like a function, but is subject only to the semantic rules for procedures—e.g., it may have VAR parameters and nonlocal assignments. Optimization is not generally available within expressions that invoke value-returning procedures.

5. *Operator Declarations*

It is necessary that the user be able to associate procedure and function definitions with lexical entities other than identifiers: these are the ExtendedIds (e.g., infix operators, :=, and ↑). One can imagine a number of different ways for a language to let a programmer establish such associations. We have chosen to define the syntactic category ExtendedId which contains the normal identifiers as well as the special tokens just mentioned. We allow any ExtendedId as the declared name of a function or procedure. This means that to declare a new body for "+", perhaps corresponding to a user-defined type T, the user simply writes

```
FUNCTION EXTENDS + RETURNS T(CONST x,y: T)
  ...body...
ENDFUNCTION +
```

Note that we have deliberately included ":=" among the ExtendedIds. We believe that the implementer of a module M, exporting an opaque representation R (see Section X of the Language Report and the discussion of modules below) will find it essential, in conveying an abstraction to a user of M, to control the semantics of assignment to variables with the representation T. The very essence of opaque exportation, as employed in the successful language design Modula and retained in our own design, is that the implementer of an opaquely exported representation must have complete control over the uses of variables of that representation. At the very least, the implementer of M may want to forbid such assignment, and this can also be achieved by not exporting ":=" from M. Alternatively, the implementer can export ":=" without explicitly defining it (thus exporting the standard definition of :=) or can redefine ":=" and export the resulting nonstandard ":=".

6. *Recursion and Nesting*

We have complied with Ironman Requirement 7B by retaining the Pascal facility of functions and procedures that can be called recursively and defined within nested scopes (including function and procedure scopes). Ironman 7B goes on to require that the language "shall restrict the interaction between nested definitions and calls on functions and procedures so that the use of an execution time display or equivalent mechanism is not required." We agree and comply with the intent of this requirement—the desire to avoid the run-time overhead of maintaining a display. However, we do not believe that the language definition is the appropriate source of this restriction.

In many embedded systems applications, the demands of real-time processing will mandate primarily static allocation strategies. In our language, static allocation strategies suffice except for the local variables of recursive routines and arrays with non-manifest index ranges. (These features would, in any event, be inappropriate for the optimum programming of embedded applications.) When recursion is thus avoided and static allocation is used—including static allocation of resources for parallel processes—no run-time overheads need be incurred for reference to free variables. The language translator can generate code that accesses free variables in a single memory reference, whether such a free variable is global or is a local variable of some enclosing lexical scope. LIS provides a kind of routine—the *action*—that is guaranteed to be implementable with static allocation strategies [26]. Although this concept is compatible with our design, we do not believe that it should take the form of another language structure; rather, we recommend that production implementations of the language for embedded applications provide a compiler directive, analogous to our INLINE directive, mandating static allocation.

In less critical applications, such as the development of language-support software, the cost of some run-time mechanism for free variable access, even in the presence of recursion, is quite acceptable. For such applications, to avoid any mechanism equivalent to the display, it will be necessary to

- Include in the language a complex rule restricting the interaction of nested definitions and calls on recursive routines,
- Train programmers to properly apply this rule, and
- Enforce the rule in all compilers.

(Indeed, this is what must be done in resource-critical embedded applications.) We have designed our language to avoid special rules because they produce these consequences, and we recommend against the inclusion of a special rule in this case.

Finally, note that an implementation strategy is available, for the unrestricted case, that is quite efficient. This strategy has been used satisfactorily in Pascal compilers for the ICL 1966 computer [14] and the CDC 6000 [16]; it involves treating the computation of the address of variables in a lexically enclosing scope as a "common subexpression." Thus access to outermost and local variables requires no overhead, and the added cost of several accesses to variables in an enclosing scope is borne only once for each needed enclosing scope. We recommend the use of this implementation technique but do not mandate that translators use any specific implementation.

B. *Encapsulation Facilities*

The block structure of Algol, with its facility to declare local objects, does not adequately cover the needs of systems programming. In particular, it does not allow hiding objects (or details about them) while they are still in existence. Objects cease to exist as soon as control leaves the procedure or block to which they are local. To some degree, the *own* variables of Algol 60 incorporate the desired information-hiding property. However, the *own* concept has well-known deficiencies, and a different solution has to be found.

A very much more appropriate solution was offered by the *class* concept of Simula, as modified by Brinch Hansen and Hoare [1][5]. The class definition defines a set of procedures (operators) and a set of variables to which only these procedures have access. The important aspect is that these variables continue to exist if control leaves any of these procedures.

Why, then, did we not adopt the class structure? The primary reason is that the unit of program that has to encapsulate information from its environment—now called a *module*—appears to be of relatively large size in systems programming, and usually only one instance of such a module exists. This is in contrast with the original aim of the class concept, where there exist many such objects of identical structure. They form a class. The class embodies the idea of an abstract data type to be declared in conjunction with its applicable operators. The module, however, rather pursues the aim of an adequate facility to declare such entities as, for example, a scanner in a compiler, a disk store manager in an operating system, or a communication line handler in a data station. Hence, the module has objectives somewhat different from the class, and it would, therefore, be misleading to claim that the module concept replaces the class concept, although it can be shown that formally the module concept covers the class concept. We shall show how a class definition can be represented by a module declaration, and then discuss the advantages and disadvantages of the two structures.

Consider the following class definition (from [1]):

```
type C = class var x, y: R1;
             procedure p(U); begin...end;
             procedure entry q(V); begin...end;
             S
         end
```

This is expressed in terms of a module as follows:

```
MODULE M;
  EXPORTS P, Q, Init;
  EXPORTS OPAQUE R;
  IMPORTS R1;
  REPRESENTATION R =
    RECORD VAR x, y: R1 ENDRECORD;
  PROCEDURE P(VAR a: R; VAR u: SomeRepr);
    ...
  ENDPROCEDURE P;
  PROCEDURE Q(VAR b: R; VAR v: SomeOtherRepr);
    ...
  ENDPROCEDURE Q;
  PROCEDURE Init(VAR c: R)
    S (* same statement as above *)
  ENDPROCEDURE Init;
ENDMODULE M
```

The advantage of the class is evident, if we create instances of the data structure, each consisting of the two components x and y:

```
VAR i1, i2: C
```

The initialization statement S is implicitly invoked once for i1 and once for i2. An invocation of procedure q, applied to i1, is expressed conveniently as

```
i1.q(v)
```

where *a* appears as a parameter in a distinguished position. In the case of the module structure, the two variables are declared as

```
VAR j1, j2: R
```

and the call as

```
Q(j1, v)
```

Initialization must be stated explicitly as

```
Init(j1); Init(j2)
```

The principal advantage of the class notation appears if there are several classes, perhaps with the same identifiers for some of their individual operators: their names are entirely local, and their identity is determined by the prefixed parameter, whose type uniquely specifies one class. In our language, a similar discrimination between operators with the same names is provided by the overloading facility.

It is not our intention to belittle the advantages of classes, but the sophisticated combination of using a distinguished parameter position to identify the scope in which its operator is defined is of much less importance in the application area envisaged for modules. It seems only natural that the designer of a system choose unique names for all operators exported from different (parallel, not nested) modules. The primary advantage is then one of conceptual simplification: the module has one and only one function, namely to establish a static scope of identifiers, whose across-boundary visibility of identifiers is strictly under the programmer's

control. Export (and import) rules are not restricted to some operators (procedures), but apply identically to names of any kind of object, such as constants, types, and variables. This generality of scope rules has in practice proved to be highly valuable.

If only one instance of a class variable is to appear, then, of course, the module notation is equally simple, if not simpler. The declaration

```
MODULE M;
  EXPORT Q;
  VAR x, y;
  PROCEDURE Q(VAR v: SomeRecpr) ... ENDPROCEDURE Q;
  S
ENDMODULE M;
```

replaces both the class definition and its instantiation, and the call Q(v) replaces a.Q(v). In the case of multiple instances, the module notation is more cumbersome but also provides some additional possibilities, such as for example the introduction of variables common to all class instance operators, declared in the module heading and initialized in its body.

Implementation of the module is, considering the additional possibilities, simpler than that of the class. After all, the only conceptual innovation of the module lies in delimiting the scope of identifiers. In the compiled code, there appears no trace of the module structure itself. Nevertheless, the additional sophistication of the compiler's symbol table mechanism is rather considerable, and in any case greater than anticipated. It weighs particularly in the case of multipass compilers. Unfortunately, it appears to be almost impossible to avoid a multipass scheme. This is not only in view of the application to minicomputers with limited store size, but primarily because of the potential desirability of cross references of objects defined in different modules.

(The above is adapted from [15] with the permission of the author.)

We have made modules, together with their IMPORT and EXPORT lists, the only program scopes with controllable opacity. Modula allowed IMPORT lists for routines and processes too, and Euclid associated importation with a number of *closed scopes*. However, we have decided to rely exclusively on the elegant and quite simple module mechanism for this purpose: the programmer can easily embed a module within a routine, process, or other lexical scope to obtain a closed scope.

1. *Operations Between Types*

The observation made in Ironman 3-5D, concerning the definition of operations between encapsulated types, is valid for our design. This is illustrated by the structure

```
MODULE MandN
  EXPORT F, G, MRtoNR, NRtoMR;
  EXPORT OPAQUE MR, NR;

  MODULE M
    EXPORT TRANSPARENT MR;
    EXPORT F, F1;
    (* definitions of F, F1, MR *)
  ENDMODULE M;

  MODULE N
    EXPORT TRANSPARENT NR;
    EXPORT G, G1;
    (* definitions of G, G1, NR *)
  ENDMODULE N;

  (* definitions of MRtoNR, NRtoMR  *)
  (*           in terms of F1, G1 *)
ENDMODULE MandN;
```

Here F and G are two routines for manipulating the abstractions implemented by the representations MR and NR. F1 and G1 are used for manipulating the implementation details of these abstractions, and specifically for implementing the conversion operations between them. The combined abstraction, as seen by the user, consists of the opaque representations MR and NR, the two routines F and G, and the conversion functions MRtoNR and NRtoMR. This structure may be used not only to provide intratype operations, as in this case, but also to add mechanisms to a single encapsulation, in the style of Simula *subclasses*, illustrating the power of the module notation.

2. *Parameterization of Types and Encapsulations*

With both types and encapsulations, there is an additional mechanism that we considered adding to the proposed mechanism—the possibility of including parameters. Some experimental languages—e.g., Ell—have provided extremely flexible parameterized type facilities, resulting in run-time overheads that would be unacceptable for the present language. Other languages—e.g., Euclid—essentially merge the notions of type and encapsulation and allow limited parameterization. The combined facility is essentially a *class* facility. This facility as it appears in Euclid permits circumstances in which the program translator cannot enforce strong typing (see [20], p. 31) and therefore violates Ironman Requirement 3A. Moreover, Euclid permits only values as type parameters, and its facility therefore does not suffice to satisfy Ironman 12D which requires various other parameters.

We have designed our language to provide parameterization of encapsulations in a way that does comply with Ironman 12D, and is also very simple to implement. We do this by providing *module templates*, which may be parameterized quite generally and, when instantiated, yield modules. We argued above that one advantage of the module as an encapsulation mechanism is that all traces (and all overheads) disappear during program translation; the same argument applies to module templates, and consequently we believe that this mechanism is more appropriate in the embedded system domain than any variety of the parameterized class.

The most common application of parameterized types and encapsulations is to separate the definition of the structure and operations in an encapsulation from the specification of the size of the internal objects. This is provided for by our mechanism. For example, consider the

module template

```
TEMPLATE StackTemplate(i,R)
 MODULE Stack
  EXPORT Push, Pop, TopofStack, ... ;
  REPRESENTATION StackRepr=
      ARRAY(INTEGER [1..i] OF BOOLEAN);
  (* declarations of Push, etc. here *)
 ENDMODULE Stack
ENDTEMPLATE StackTemplate
```

Within the scope of this declaration, the user requiring a stack of up to 100 INTs need only write

```
INSTANCE StackTemplate(100,INT)
```

C. *Processes*

The simplest process structure we considered was that in which all processes are declared to be global, are fixed in number, and can be created by the compilation. This scheme would probably suffice for almost all embedded systems, particularly given compilation-time conditioning of the program, and has significant advantages of reduced overheads. It is possible to use the mechanism that we do propose in this manner, and compilers will need to recognize this and take advantage of it.

A slightly more general scheme has been demonstrated, without undesirable side effects, in the Modula language. Here, processes are declared globally and are activated by global program text, thus permitting parameterization of processes and activating of variable numbers of processes. This is particularly useful for programs that must adapt to the particular configuration within which they are running.

An alternative approach could have been based on the fork-and-join concept, best exemplified by the "cobegin-coend" proposals. This too can be used to create variable numbers of processes through the introduction of a "cofor" construct. Despite a certain formal elegance, this approach was rejected, partly because it introduced more additional mechanism into the language, partly because of arguments of process lifetime (discussed below), partly because it permits the creation of unencapsulated processes, and partly because the mechanisms of Modula appeared appropriate and already existed.

The Modula language requires that processes be declared and activated at the global level. It is accepted that this is sometimes a disadvantage, and particularly that it restricts the modularization of the program according to the logical significance of the processes. We have removed this restriction, accepting that stack-space allocation and access to nonlocal variables from within processes is more complex. We are, however, satisfied that our nonlocal variable-access mechanism already suffices. Inevitably, generality permits the creation of inefficient constructs (e.g., the declaration of processes within recursive procedures), and programmers will need to exercise some discretion in creating programs that are low in overheads and easy to understand.

One particular alternative considered was whether the life of a process should be determined by the scope of its declaration or by the scope of the statement activating it. The latter alternative would ease formal analysis of programs, and would avoid problems relating to the life of VAR parameters to processes. But the advantages of being able to activate processes from within procedures were thought to outweigh these, and resulted in the proposed rule.

1. *Interprocess Communication*

There is a very wide range of interprocess-communication mechanisms available, ranging from very-low-level primitives (such as interrupt inhibition, and test-and-set) to high-level constructs such as message passing, path expressions, or guarded commands. The disadvantage of very-low-level primitives is that they force low-level implementations, possibly lower level than might otherwise be available from the more sophisticated hardware now becoming feasible. The higher-level constructs enforce a more structured approach to system design, but may impose more overhead than is really necessary in simple situations. Moreover, many higher-level constructs, such as message passing, can be provided as encapsulations of lower-level mechanisms without significant loss.

Considerations of program structuring persuaded us to provide monitors, which encapsulate the interaction of processes in a module within which the monitor exclusions can allow safe and simple manipulation of the shared data structures. (A lower-level facility, using semaphores, is presented in Section XIV of the Language Report and also discussed below.)

The priority parameter provided with the wait operation on a signal is not strictly necessary, but was retained from the successful Modula implementation.

2. *Scheduling*

Ironman requires that the semantics of the language should be independent of the number of processors used to implement it (9B) and also requires specific scheduling disciplines (9D). We are satisfied that the scheduling discipline described in 9B is appropriate only for single-processor systems and cannot be strictly observed, on a multiprocessor system with interrupts, without excessive overhead. We have, therefore, chosen a semantics in which all processes conceptually proceed in parallel, as though there were sufficient processors for each process to have its own. (Process priorities are regarded as implementation-dependent; the semantics are presented in Section XIV of the Language Report and also discussed in Section XIV below.)

3. *Interrupt Handling*

We regarded it as self evident that the interrupt-handling facilities of the language should be based on the facilities already present in the language for the asynchronous interaction of processes. The outline of our mechanism is derived from the Modula Language, for which the feasibility of efficient implementation has already been demonstrated. Some generalization of the Modula mechanism was felt to be necessary to accommodate the wide range of interrupt mechanisms actually found in current processors, not to mention those of the future. It was also necessary to provide a mechanism that allows the release of an interrupt process waiting for a device interrupt that does not arrive.

If a program takes full advantage of these generalizations, the efficient mechanization of Modula will no longer be available and the implementation must convert all interrupts into software signals, with a consequent increase in overhead. But for each particular implementation there will be defined a set of rules, observance of which will enable the implementation of interrupts to be optimized. This set of rules may be very restrictive on processors with primitive interrupt mechanisms, such as the Data General processors, more liberal with processors such as the PDP11 or IBM 370, and possibly essentially unrestricted on future designs optimized against the DoD1 language.

It is, of course, envisaged that almost all interrupt handling will be associated with low-level input-output operations, and will be encapsulated within modules that will conceal any

implementation-dependency from their users.

4. *Simulation*

It would have been possible to design a simulation facility that was quite different from the existing facilities for asynchronism within the language. Such a design would have been both unaesthetic and unnecessary, as is shown by Hoare's Simone language [21]. On the other hand, it might have been possible to achieve a simulation facility purely by encapsulation and overloading. Unfortunately, not only must the existing types of signals and semaphores be replaced, but also the "operations" of monitor entry and exit, and of process activation and termination must be overloaded. We thus decided that simulation facility should be primitive and developed the concept of the simulation block.

Note particularly that it is essential that a program within the simulation block be able to interact with processes, monitors, etc. declared outside the simulation block, for otherwise effective input-output would be impossible. The bulk of the mechanism for simulation blocks is shared with that for monitor modules within which processes are declared, the difference being the replacement of the real-time clock within the simulation module.

D. *Exception Handling*

1. *Problems in Error Recovery*

Error recovery represents the most difficult aspect of the language design. For the rest of the language, the requirements of Ironman can be met by selecting features that have been demonstrated in prior languages, at least individually if not in aggregate. But in our opinion, no existing exception-handling mechanism is even superficially acceptable for this language. Consequently our proposals differ radically from solutions arrived at in existing languages (for example, PL/1 [38] or Mesa [27]) or in the literature [3]. Inevitably, the exception-handling facility represents the highest risk aspect of the language.

The primary problem of error recovery is the very great difficulty in programming correctly. Error recovery is undertaken when the main program has failed in an unforeseen manner and the state of the program and its data is perhaps uncertain. Further, the error-recovery programs are extremely difficult to test and get very little operational use. Thus, the rather complex and difficult-to-understand error-recovery code is liable to be extremely unreliable. We would like to keep the exception-handling mechanism as simple as possible so as to minimize the difficulties of understanding and the risks of unreliability.

Another problem with exception handling mechanisms is their obscure semantics, and the interactions of these semantics with the compilation of efficient code. The following are particularly troublesome:

- Resuming execution in the middle of an expression evaluation without benefit of the standard linkage conventions (e.g., deciding in which register to return a result)
- Repeating the "current" statement or continuing with the "next" statement
- The validity of reference parameters for exception handlers as the stack is wound up
- Inefficient compilation and run-time overheads forced by retention of the source program statement structure for run-time error recovery

- Increased subroutine linkage costs due to the need to record exception-handling scopes as a part of the linkage structure
- The need to use dynamic name resolution when searching for exception handlers rather then simple static scoping.

A further problem with exception handling is the reconciliation of the requirements of error recovery, which implies a return to an earlier "safe" state, and of diagnosis and error logging, which requires retention of and access to all state information. Still worse is the tendency, given a powerful exception-handling mechanism, to use it for fully anticipated, normal program conditions. Experience with the Mesa language indicates that exceptions can be used to provide very obscure and difficult-to-understand (and very inefficient) program structures such as:

- Loops fabricated from exceptions and exception handlers
- Loops that exit only by an exception
- Procedures that return only by an exception.

Programs may also use the exception-handling mechanism to evade scoping restrictions imposed by the language, or to fabricate features deliberately not provided by the language (such as procedure parameters).

2. *The Objectives of Error-Recovery Facilities*

The primary objective of error-recovery facilities is the provision of software-error recovery from unforeseen situations in operational embedded systems.

Secondary objectives are the provision of diagnostic and error-logging facilities during program development and commissioning, and the extension of the domain of hardware operations.

The primary objective is oriented purely towards recovery and continued processing within the partially failed program, possibly without regard to the cause of failure. The paradigm of program recovery we have used is

DO P ELSE Q

where an exception detected within a program text P causes some alternative action Q to be undertaken. If the program text P invokes a routine or operation that detects an exceptional condition preventing it from completing normally, the routine or operation generates an exception. The exception terminates the procedure or operation and returns to the context of P, seeking there an exception handler, in this case Q. The exception handler Q aims to complete the processing for which P was required and to return normally to the environment that invoked P.

When the exception occurs, some routines will be in the middle of their execution. If such routines are terminated abruptly, their own variables and global data structures may be left in a partially modified, and therefore inconsistent, state. Recovery of the system requires that such routines be allowed at least to restore their data structures to a consistent state even if they are unable to complete the processing for which they were invoked. It is this requirement that prevents the use of the GOTO statement for error recovery.

The secondary objective of error diagnosis and logging requires that the context of the error be retained for examination by the diagnostic system. Of course, any such diagnosis, involving examination of the run-time stack and context, must be inherently implementation-dependent.

Similarly, a handler that is to extend the domain of a hardware operation—for instance, that returns a floating zero after floating-point underflow—also requires that the context of the error be retained, so as to permit continuation of the program. Unfortunately, this type of handler must also be implementation-dependent, for it must examine that context and the program code to discover exactly what is required—for instance, in which register to place the floating zero.

3. *Alternative Approaches to Exception Handling*

In our search for a simpler mechanism for exception handling, we considered the use of existing language constructs such as BOOLEAN flags or error codes, the GOTO statement, and procedure calls. Schemes based on error codes or flags required additional statements (and possibly parameters) in the main-line program to test for errors, and may also suffer from ambiguous global error indications in multiple-process situations. Schemes based on the GOTO statement and procedure calls were unable satisfactorily to restore the partially updated data structures of uncompleted routines. It became evident that a special-purpose mechanism is more obvious in the program, is more specific in its intent and effect, and permits more constraints to ensure the correct use of the recovery mechanism.

Examination of the objectives shows that the problem of error recovery has two distinct aspects. Therefore we decided to separate the mechanisms for the initial handling of the error indication (by the *trap handlers*) and the subsequent recovery actions within the user program (by the *exception handlers*). The two different mechanisms can each be simpler and more specific, and also the exception handlers can be completely implementation-independent.

The design of the trap-handling mechanism is explained in Section XIV of this Design Discussion, since trap handlers are likely to be implementation-dependent. Each implementation of the language must provide a standard set of trap handlers that generate the standard set of exceptions for the user program. This standard set of exceptions is listed in Appendix C of the Language Report. However, these trap handlers can be replaced with others more appropriate to a specific purpose. During program commissioning, a set of trap handlers might be used that is oriented towards interactive program diagnosis and fix-up, with the program being resumed without any exceptions being generated. For an operational embedded system, the trap handlers will probably convert all the traps into a very small set of exceptions, which may be largely independent of the cause of the error but will be oriented towards recovery of the embedded system.

In contrast with other existing designs, the proposed mechanism for exceptions and exception handlers does not regard the exception handler as a modified procedure, called by generating an exception. There are three reasons why this approach was rejected.

First, the section of program that discovers the error condition (or the trap handler) can itself invoke a procedure directly, should it need to. That procedure call is then subject to the normal rules for scoping of names and parameters that are designed to maintain the structure, the integrity, and the clarity of the program. Existing mechanisms, using procedure-like exception handlers, weaken these rules and damage the structure of the program.

Second, the paradigm of error recovery to which we worked does not involve resumption of the routine that detects the error, nor does it involve reference to the local variables of

that routine. Consequently, it is not necessary to retain for use by the exception handler the context of the routine that detects the error, and the mechanism is simplified if that context can be removed from the stack before the exception handler is invoked.

Third, experience with the existing Mesa exception-handling mechanism suggests that procedure-like exception handlers are used to provide implicit procedure parameters. Not only are procedure parameters explicitly forbidden by Ironman 5D, but also they greatly increase the difficulty of understanding a program (particularly when implicit), and they require a more complex run-time stack management and addressing system.

Because of these considerations, the generation of an exception, by a PROPAGATE_EXCEPTION statement, more closely resembles a RETURN statement than a procedure call.

Consideration was given to the provision of parameters for exception handlers but, after much discussion, they were excluded from the design. The presence of parameters requires a much more complex declaration for the exception handler and also special rules to ensure that the formal parameter of the exception handler cannot refer to an actual parameter that no longer exists. Further, in the paradigm of error recovery given above, the exception handler Q has little need for parameters describing the exact location and cause of the error, but rather needs to know the intent and progress of P and can gain that information through local variables without need for parameters. There is, of course, no reason why the body of the exception handler should not contain a call on a procedure, with parameters as appropriate.

A further design alternative concerns the point within the main-line program at which execution can resume following completion of the exception handling. Considerations of clarity, implementation, and formal definition precluded resumption within an expression evaluation. Even resumption within a statement sequence (by resuming with the "next" statement, or repeating the "present" statement) has obscure semantics for some statements (for instance, the "next" statement for a RETURN statement, or the "present" statement for the declaration and initialization of a variable), and also constrains the compilation to retain the statement structure at run time with certain loss of efficiency. We decided that the block structure of the program is semantically well defined, does not conflict with efficient compilation, can be easily understood, and accords well with the paradigm of error recovery.

Consideration was given to creating variables in which to record the current exception. This was rejected because of the dynamic scoping required for exception handling, and the desirability of local exceptions. To provide such variables would have required a new kind of type whose values would have been context dependent, a horrible concept. We found, however, that all the useful constructs involving discrimination on exception names could be provided by the case selection within the exception-handler definition, and by the version of propagate-exception that cites no exception name and propagates the exception that caused entry to the exception handler.

The function returning the most recently generated exception was rejected for the same reason, and also because the most recently generated exception could have been completely handled while a previous exception remained outstanding. The most useful concept is the exception that caused entry to the exception handler.

Consideration was given to the scope to which an exception handler should apply. An approach in which the exception handling is defined for each statement individually leads to a verbose program in which the main flow of control is obscured, and also restricts the compilation process. The definition of one exception handler for all the statements of a scope is in keeping with the block structuring of the language. Where necessary, local scopes can be

constructed by use of the BEGIN...END statement.

It is important that the exception-handling mechanism react correctly if a second exception occurs during the handling of the first, for the circumstances necessitating error recovery make further exceptions more probable. If the second exception can be handled locally, it is appropriate to continue with the handling of the still-unresolved first exception. If the second exception cannot be handled locally, there can be no rule as to which should subsume the other. The design of the exception-handling facility allows the body of an exception handler to be a block within which is defined a further exception handler. This inner exception handler, on the occurrence of the second exception, can generate either the first exception or the second, or even an entirely different exception.

We might have chosen a design in which, if an exception is not handled within a process so that the process terminates with the exception still outstanding, the the exception is propagated into the context enclosing the declaration of the process. Complications would have arisen when two processes terminated in this way. Also, this enclosing context can not resume processing until all its subordinate processes have terminated and since, in the embedded systems context, processes seldom terminate, this would have had the effect of postponing most exceptions indefinitely. We therefore believe that such exceptions should be dealt with by an exception handler in the process that communicates with the rest of the system by means of a monitor, and this is easy to do in our language. In the absence of such a handler it seems simplest to let the process terminate, leaving the exception code in its activation record and accessible to diagnostic programs.

XI STANDARD PROCEDURES

A. *LOW and HIGH*

These procedures are intended primarily to allow a general programming style in dealing with length-unresolved array parameters. We have complied with both Ironman requirements—length-unresolved array types (7H) and ranges of the integers as actual array subscript ranges (3-3D). This combination of requirements makes it awkward to program the body of a routine that takes a length-unresolved array parameter unless it is possible to determine the subscript range of the corresponding actual parameter. These procedures serve that purpose; they can also be applied to length-resolved arrays.

B. *MOVE*

This procedure is provided in compliance with Ironman requirement 3-3E for a facility for catenation of contiguous sections of one-dimensional arrays of the same component type. Since requirement 3-3E forbids the use of catenated sections as general values, it seems that the simplest way of complying is to provide all the required facilities by a single procedure.

C. *AND_LONG and OR_LONG*

In compliance with Ironman requirement 6D for "short circuit" forms of the Boolean operations, we have made this the semantics of the infix Boolean operations. It follows that it is necessary to have "long circuit" forms as well; these are AND_LONG and OR_LONG. They are not infix because we expect that their use will be uncommon.

D. *Arithmetic Conversion Functions*

These are the functions specifically required by Ironman requirement 3-1B.

XII HIGH-LEVEL INPUT AND OUTPUT

The routines listed in Section XII of the Language Report are quite simple—essentially those of Pascal. Because they are simple they will not serve all needs but, based on the experience of Pascal, it is clear that they (or some refinement of them) will serve many needs. There are two methods of addressing those needs not fulfilled by these routines: elaboration of the routines given or construction of specialized, application-oriented input-output routines. The first method is appropriate for providing facilities that will be of common use in all implementations of the language. Because these facilities are part of the standard definition of the language, they must be provided in all implementations. They can be used to write portable programs that do input-output of limited sophistication. The large body of software that will be developed to support the language—e.g., editors, debuggers, and formatters—is a good example of a domain to which the input-output routines of Section XII are well suited. In such domains the portability of programs is critical; the sophistication with which they do input-output is not. (We expect, in Phase 2, to slightly elaborate the high-level facilities—e.g., by providing RESET and REWIND operations as in Pascal.)

The low-level input and output facilities and the encapsulation facility we have provided are sufficient to synthesize more elaborate facilities suitable to the requirements of specific applications. Indeed, this synthesis will be part of the design and implementation of most embedded system applications. The precise details of these more specific applications must be determined as part of the specification of the many standard library routines that will be similarly implemented; some will be maintained and controlled on a DoD-wide basis, others on a more limited basis. It is neither reasonable nor feasible to implement all application-specific routines as part of all implementations of the language. Thus more sophisticated routines to fulfill specific needs are not appropriate to Section XII.

XIII SEPARATE COMPILATION

A. *Design Objectives*

The design of the separate compilation facility has been guided by these objectives:

- Simplicity. The rules for segmentation and the meaning of a segmented program should be easy to understand. No special semantic constructs should be included just to support separate compilation.

- Compilation efficiency. To compile one segment, the compiler should not need to scan the text of related segments.

- Segment independence. The design should help minimize the need to recompile one segment when a related segment changes. There should be no dependence on the order in which related segments are compiled.

- Type integrity. The compiler should be able to check the interface between separately compiled segments as if they were compiled jointly. The burden placed on the linkage editor (called the "linker") to validate interfaces should be minimal.

- Execution efficiency. No degradation in speed and storage efficiency should be caused by segmentation.

B. *The Role of the Synopsis*

The designation of the skeleton of a segment that is its synopsis helps achieve the design objectives for separate compilation in several respects. It involves no new semantic constructs. The synopsis corresponding to an existing full segment can be mechanically extracted. Alternatively, if the synopsis is written first—a reasonable project management technique—then the implementer of the corresponding segment proceeds by expanding the skeleton of a segment into a full segment. The need to recompile one segment when another segment changes is minimized; only when the synopsis changes will recompilation of related segments be forced. Segments need not be compiled in a particular order, as is the case in some systems (e.g., when attribute files or symbol tables produced by the compiler must be transmitted between compilations—the synopsis takes the place of these attribute files). The synopsis permits interface checking to be performed completely at compilation time; the linker's only responsibility is to ensure that all object modules being loaded together have been compiled using the same synopsis for each segment and to announce which ones need to be recompiled if any are

invalid.

Our design makes use of two entities: the full segment and the segment synopsis. To allow consistency checking, our design requires the existence of a utility that can automatically extract the synopsis of a full segment. An alternative scheme would be to describe an entity called a *realization*, containing that information about a segment not included in the segment synopsis. No automatic extraction utility would then be required but a utility would be needed to merge a synopsis and realization into a full segment. We believe that the realization would be a difficult entity to read or write because it would only make sense in the presence of the corresponding synopsis.

C. *Exclusion of Nested Segments*

The ability to nest segments—for example, through use of the declaration INCLUDE M to take the place of the declaration of a module M that is separately compiled—was excluded from the language primarily to retain simplicity. Segment nesting would almost certainly complicate the linkage editor for the language, since it would involve linking to nonglobal external variables and would require checks for circular inclusion patterns that could not always be detected at compile time.

It would be feasible to add nesting of segments (with a resulting increase in linkage editor complexity), provided that segment imports were restricted to the exports of other segments (i.e., not permitted to be arbitrary items local to the enclosing environment). Such a restriction is mandated by Ironman 12B, and would be necessary to avoid greatly complicating the compiler—e.g., by adding mechanisms to produce symbol tables for internal environments of a segment to be transmitted between compilations.

D. *Linkage Editor Checking of Segment Consistency*

The linkage editor will be responsible for ensuring that all segments and synopses comprising a program are consistent. The compiler must include in each compiled segment S a record of the versions (e.g., times and dates of creation or last modification) of synopses used in the compilation of S. In our scheme, the linkage editor can do this efficiently by verifying that a segment S and all other segments depending on S have been compiled using the same version of the synopsis of S. If versions do not match, the linkage editor will require appropriate recompilations. Even then, if it is determined that S is consistent with the synopsis of S used in previous compilations, no further computation is required. The user will not need to maintain or refer to this information. The user simply requests that segments S1, ..., Sn be linked, and minimal consistency checking and minimal recompilation can occur automatically.

In place of the source synopsis, one might also allow use of a *compiled synopsis*. This is a compact encoding of the declarations in a synopsis, having the form of a partial symbol table, so that it can be used with little further processing in subsequent compilations. In the presence of such a compiled synopsis, appropriate additional consisting checking would be done. When given no compiled synopsis for S, the compiler would produce one before proceeding with the primary compilation.

Simpler schemes could be implemented. For example, the manipulation of compiled synopsis files could be avoided if source synopses were required to reside in separate files whose creation times could be used in composing synopsis identifiers. The method we have sketched has the advantages that trivial changes in the synopsis source (e.g., documentation changes or text formatting) need not force recompilation of related segments, that synopses can be included in files containing other source text, and that redundant processing of source synopses is minimized in typical cases.

XIV LOW-LEVEL AND IMPLEMENTATION-DEPENDENT FACILITIES

A. *Representation of Data*

If the attributes AT, PACKED, and ALIGNED are not used in a program, then the selection of these aspects of the representation of values is left entirely to program translators and the program is entirely implementation-independent and portable. By using these qualifiers, the programmer gains control over the representation of values, at the expense of implementation-dependency and loss of portability.

The attribute AT is used to qualify the field names of a record to specify their relative position within the record in implementation-specific units (usually in bits). The reader will find illustrations of this use in the Programming Examples. Note that this use of AT is semantically related, and syntactically similar, to the use of AT with variables and parameters—to specify the exact location in memory, in an implementation-meaningful way, of a variable or parameter.

The attribute PACKED is used to control the size in storage of the qualified entity. The simplest use is with the optional parameter omitted—some control over representation is obtained without loss of portability. The storage required for representing such entities will typically be smaller (and never greater) than that required for the unqualified entity. The translator is directed to reduce, so far as is reasonable in the particular implementation, the storage used to represent these values (typically at the cost of an increase in time to access the components of the packed value). Each implementation supports the PACKED qualifier in a manner meaningful for that implementation. The programmer is assured of a more economical use of storage where this is reasonable, and does not have to dictate a representation that is appropriate only for a single implementation. This qualification is often sufficient to accomplish the programmers intent without sacrificing portability.

The use of PACKED with a parameter allows more implementation-specific control of storage utilization. The parameter is a manifest constant, expressed in implementation-specific units, that specifies an upper-bound on the storage that may be used for the qualified entity. If a translator has the capability of representing the entity to meet this constraint, then it will do so; if not, it will indicate that the program is in error.

Many machines provide instructions that can only be applied to data aligned on specified boundaries in memory—e.g., double precision instructions on data at even addresses. Each implementation will provide an interpretation of the manifest constant in the ALIGNED attribute that allows the programmer to take advantage of such machine instructions.

B. *Pointers and Addresses*

POINTER data (as described in Section VI of the Language Report) and the operations applicable to pointers, ↑ and NEW (as described in Sections VIII and XI of the Language Report), are safe. That is, they obey strong-typing rules and cannot be used to create "dangling" pointers. In embedded system applications, however, it is often necessary for the programmer to engage in direct storage management. We provide four features to allow this direct control—FREE_POINTER, ADDR, MAIN_STORE_ARRAY, and MAKE_POINTER. These facilities are not safe: improperly used they can create dangling pointers and ill-formed objects. This means (1) that they must be used with great care and should be avoided where not needed and (2) that they should be used only within an encapsulation that shields its user from their dangerous properties.

Note the distinction between pointers and addresses. Pointers are typed entities with a language-specified semantics. Addresses are integers with an implementation-specified semantics (e.g., an address in main store) and can be sensibly mapped into pointers only by the programmer who is aware of the details of addressing and storage allocation in a particular implementation.

In Pascal, all anonymous variables are allocated in one region of storage, using a single implementation-defined allocation algorithm. Programming embedded system applications sometimes requires several different regions for anonymous variables, each region using a different allocation algorithm. Such regions are provided in Euclid as *zones*. The effect of Euclid zones can be defined in our language as an encapsulation using the facilities already described.

C. *Semaphores*

Considerations of efficiency and flexibility persuaded us to provide semaphores. Semaphores can be declared as components of data structures and provide more explicit control than signals and monitors over the interactions between processes. The use of semaphores, however, may increase the difficulty of understanding a program. In principle, signals and monitors can be fabricated from semaphores and semaphores can be fabricated from signals and monitors. But considerations of efficiency and program structuring, in certain low-level applications, indicate that both facilities should be available.

We selected binary semaphores as more suitable for inclusion than counting semaphores.

- Counting semaphores encourage the use of more complex exclusion structures and direct resource allocation. We consider these to be best expressed by explicit programming using the simple exclusion provided by binary semaphores.

- Binary semaphores allow optimized scheduling in which a high priority process, blocked by a low priority process, can have its priority transferred to the low priority process for just as long as is necessary to permit the low priority process to complete its critical section and release the semaphore. Without such a facility, high priority processes can not safely use semaphores or monitors, and complex exclusion mechanisms prevent rapid analysis of the cause of blocking.

The conditional claim operation on semaphores is present to permit the construction of highly time-critical programs.

D. *Low-Level Manipulation of Typed Data*

Implementation-independent language features such as overloaded routines and variant records make it possible to group and treat uniformly some families of objects of different types or structures. The utility of these groupings is limited by the fact that the common characteristic of a family must be a language-defined, machine-independent characteristic. In embedded system applications, this does not always suffice. It may be necessary to group objects that have only a machine-dependent common characteristic, namely, their physical realization on a specific machine. We allow such groupings only at the routine-invocation boundary, and hence provide it through a special formal parameter representation, SPACE. When a formal parameter of a routine is declared to have this representation, any actual parameter is permitted in an invocation of the routine. This feature is especially useful in low-level input and output where it is necessary to treat values as streams of bits in dealing with a device.

Because the common denominator of the actual parameters corresponding to a SPACE formal parameter is their physical realization, we permit them to be manipulated only in terms of that realization. This is done using the routines ADDR and SIZE (see Section XIV.XI of the Language Report).

An alternative to SPACE is a set of constructs with which a variable with one representation can be manipulated as though it had another representation. We believe that such constructs are confusing: they refer to a variable in two different ways in the same program fragment. Moreover, these constructs would only solve part of the problem: they still would not allow a formal parameter that matched any actual parameter. Therefore, we employ the SPACE formal parameter representation.

E. *Process Attributes*

We regard the PRIORITY attribute as advisory to the implementation, and not binding. Each implementation will aim to provide the best combination of scheduling discrimination and scheduling overhead. In some circumstances this best combination may be a form of deadline scheduling, and this is not excluded.

The presence of procedures, by which a process can change not only its own priority but also that of other processes, is a consequence of the requirement that a process be able to raise a termination exception in another process. A way of unambiguously referring to another process must, therefore, be available, and can be used in the changing priorities too. This is the application of the PROCESS_ID routine.

To facilitate space allocation, particularly in resource-critical applications, the INSTANCES and SIZE attributes are provided (as specified in Ironman Requirement 9A). An application that is not resource-critical may not need this optional facility. An alternative to the SIZE attribute would be the requirement that compilers calculate the space required, possibly in terms of an additional attribute specifying maximum depth of recursive procedure calls. This might or might not yield more precise space allocations, but would certainly yield more complex compilers and was therefore rejected.

F. *Traps and Exceptions*

The need for a distinction between exception handlers and trap handlers has been discussed above (see Section X). Trap handlers provide an interface between the potentially implementation-dependent error detection of a processor and the implementation-independent error recovery of the exception handlers.

It would have been possible to omit the trap handlers and to require all errors to be reported through the standard exceptions listed in Appendix C of the Language Report. Had we done so, every embedded system using exception handling would have been obliged to contain a full exception-reporting mechanism capable of distinguishing between exceptions not distinguished by the hardware (e.g., between floating-point overflow and divide by zero). Also, the absence of the trap handlers would have required a more complex exception handling mechanism, to provide diagnostic information during program test and integration, thus imposing an unnecessary overhead on operational systems.

We believe that exception handling in an operational embedded system should be oriented towards recovery of the embedded system and be largely independent of the apparent cause of the error. Thus the embedded system will use a very short list of exceptions, not distinguishing similar types of error but, instead, indicating what activity which was in progress when the error occurred.

In contrast, the trap handlers used for program testing may be much more elaborate than the standard set of trap handlers. Typical facilities required here will include logging of the error and interactive diagnosis of the error. If the problem can be corrected, or at least the damage repaired, the trap handler may then continue with the program, reporting no error. Such diagnostic trap handlers need access to the run time environment of the program, including the stack, process descriptors, etc. The language design permits such access, accepting that it is necessarily implementation-dependent.

Trap handlers can be used to extend the range of certain hardware operations, such as the return of zero after floating point underflow. These trap handlers must, with typical hardware, examine the program code to discover how to proceed.

Error indications reported by the hardware may concern hardware conditions rather than program exceptions—e.g., power failure, parity error, processing errors, or page-not-in-store indicators. Many such errors require immediate action and can be corrected, but reporting them to a user program is inappropriate. Trap handlers can be provided to perform the recovery action with minimal organizational overhead.

G. *Numeric Inquiries*

The implementation-specific numeric inquiries, provided in Section XIV of the Language Report, are in response in Ironman Requirement 3-1E.

H. *Code Inserts*

Machine code inserts are required (Ironman Requirement 11-E). We would have provided them in any case, as we believe that they are a necessary common denominator for the expression of machine-dependent input and output. The procedure call notation was adopted as suggestive, straightforward, and, most important, as making possible easy compiler checking of whether a machine-code insert is meaningful. For example, the procedure DECSYSTEM10 might indicate a Decsystem10 code insert and the procedure AN_UYK_7 might indicate an AN/UYK-7 code insert. Such procedures will have to be explicitly imported into any module in which they are used, as required by Ironman Requirements 11D and 11E.

I. *Low Level Input and Output*

Careful consideration was given to the possibility of defining a machine-independent set of low-level input and output functions. However, this was precluded by the extremely diverse input and output facilities of existing computers. Further, this area of the language is one in

which high efficiency, precise control over the hardware, and excellent error recovery are required. Thus we have concluded that only machine code inserts provide the extreme flexibility that is necessary.

We do not consider this use of machine-code inserts a flaw. In each implementation, skilled systems programmers will assemble machine-code inserts, device signals, and other language facilities into encapsulations appropriate for use by other programmers. These encapsulations will be programmed with detailed knowledge of the machine architecture and the local compiler to take full advantage of available resources.

We illustrate this point by showing two examples of interrupt-handling relevant to typical small computers. We explain, for each example, how a compiler can efficiently implement the semantics of mutual exclusion of the language primitives that are used.

```
PROCESS interrupt_handler ()

  MONITOR interrupt_scope
    IMPORT other_monitor_scope;
    VAR interrupt AT interrupt_location: SIGNAL;

    (* initialize *)
    WHILE active
    DO  (* activate device *)
        WAIT (interrupt)
        (* handle interrupt, calling          *)
        (* other_monitor_procedure            *)
        (* to communicate with rest of system *)
    ENDWHILE;
    TERMINATE_PROCESS;
  ENDMONITOR interrupt_scope

ENDPROCESS interrupt_handler
```

Here we create a process that declares a monitor and device signal to handle an interrupt. Since the device signal is local to the process (and cannot be exported), no other process can wait for this signal. Since the monitor exports no procedures, no other process can make a call on it requiring exclusion. Thus the only form of exclusion required of this monitor is that the device may signal the interrupt only while the process is in the WAIT, a simple structure readily compiled in terms of interrupt inhibition. The process communicates with the rest of the system through calling the procedure "other_monitor_procedure" which it imports; such calls may delay the process due to the exclusion implemented by the other monitor.

```
MONITOR input_module
  EXPORTS get
  VAR data_ready, buffer_ready: SIGNAL;
      interrupt AT interrupt_location: SIGNAL;
      buffer_full, buffer_empty: BOOLEAN;
  PROCEDURE get ((* args go here *))
    IF buffer_empty THEN WAIT (data_ready) ENDIF;
    (* get data *)
    SEND (buffer_ready);
    RETURN;
  ENDPROCEDURE get;

  PROCESS device_driver ()
    (* initialize *)
    WHILE active
    DO IF buffer_full
         THEN SEND (data_ready); WAIT (buffer_ready);
       ENDIF;
       (* activate device *)
       IF NOT buffer_empty THEN SEND (data_ready);
       ENDIF;
       WAIT (interrupt);
       (*handle interrupt*)
    ENDWHILE;
    TERMINATE_PROCESS;
  ENDPROCESS device_driver

  device-driver();

ENDMONITOR input_module
```

Here we follow closely the rules established in the Modula language, designed for, and implemented on, a single-processor PDP11 with a high degree of efficiency. The monitor module exports the procedure "get" for use by the rest of the system, but imports nothing. Local signals, "data_ready" and "buffer_ready", are used to coordinate the procedure "get" and the process "device_driver". The process "device_driver", declared within the monitor, may not call any procedure outside the monitor and may SEND only immediately before it WAITs.

The exclusion required in this example are that interrupts be inhibited for the duration of execution of procedure "get", thus excluding process "device driver". The rules on process "device driver" are designed to avoid the need to exclude procedure "get" and to ensure that housekeeping is only required should the process be obliged to wait on a signal other than its device signal. Related examples may be found in the Programming Examples.

J. *Program Translation and Configuration Discrimination*

We have provided compiler and representational directives as requested in Ironman Requirement 11. The precise list of such directives will be assembled during the design of the first production implementations of the language. Our module facility is a convenient and appropriate way of assembling the required constants, routines, representations, etc. into one or more units that can be explicitly imported into program segments using them.

Conditional compilation is a facility used to write a program that is "self-adapting" to a variety of implementations or configurations. It should be used in conjunction with some form

of configuration discrimination to assure that the "self-adapting" program is not used in a context to which it cannot adapt (e.g., a radically different machine). We have provided in our language for some aspects of conditional compilation by means of the notions of manifest constant, dead-code elimination, module template, and explicit importation into modules. To be widely applicable, these language features will be complemented by appropriate facilities in a "language support environment," such as a facility for using different versions of a module in different instances of a full system.

PART FIVE

ANALYSIS OF FEASIBILITY

CONTENTS

I THE FEASIBILITY OF FORMAL SEMANTIC DEFINITION

A. *Introduction*

In this section, we consider the feasibility of formal semantic definition of our language, and the reasons for carrying out such a definition. Our conclusions are:

- A complete formal definition of the language would be highly desirable. It would facilitate the analysis of interactions among different language features, help to detect ambiguities, and in general help to achieve a clean design. An ancillary benefit would be the future application of program verification, as verification technology becomes suitable for production use.

- The method of semantic definition by means of proof rules (the Hoare axiomatic approach) is presently applicable only to a restricted ('abstract') subset of the language. The features of such a subset are described in Section B. It would certainly be necessary to exclude from this subset all the implementation-dependent features described in Section XIV of the Language Report. Section B covers those language features for which we feel confident of the success of an axiomatic definition. Section C lists additional aspects of the language for which the success of axiomatic definition is possible, but somewhat in doubt in view of the state of the art. Section D lists features that cannot foreseeably be covered by a formal axiomatization and features for which axiomatization is simply inappropriate.

- It is probable that a complete semantic definition of the language could be given in operational or denotational terms. However, we feel that actually carrying out this task would consume an inordinate amount of effort, and might well require several years. It is doubtful that the benefits of such an exercise would justify the cost. Moreover, an axiomatic definition would still be superior to an operational or denotational definition for reasoning about program behavior and gaining insight into what is good or bad about a programming language (except for many issues of implementation and efficiency).

- Therefore, we argue that much of the potential benefit of formal semantic definition can be obtained by employing an axiomatic approach for a part of the language. As research progresses in this active area, it is highly likely that the part of the language amenable to axiomatic definition could be enlarged. We also recommend the development of a *standard interpreter* for the language. This would serve many needs outside the realm of formal definition and would also be a kind

of operational formal definition.

B. *Axiomatic Definition of an Abstract Subset*

1. *Introduction*

The axiomatic method (Hoare-type axioms and proof rules) has been used to good effect for defining the semantics of the languages Pascal [33] and Euclid [34]. Moreover, those responsible for the axiomatization of Euclid have stated that that exercise was valuable in revealing flaws in the language. Much the same approach could be employed to define the simpler aspects of our language.

In the remainder of this subsection, we list those features whose axiomatic definition would, in our opinion, pose no significant problem. The next subsection covers somewhat more difficult aspects of the language, whose inclusion in the axiomatic subset will depend on the results of a deeper investigation of the definitional task.

2. *Data Types of the Axiomatic Subset*

The axiomatic subset would include the following data types: UNORDERED, ORDERED, ALPHABET, ARRAY, RECORD, FILE, and SET. They are all close to types that have been successfully axiomatized in other languages. Note that we have deliberately omitted the type POINTER because it leads to aliasing problems that complicate the axiomatization of procedure call.

Types and representations interact with assignment and with binding of CONST and OUT formal parameters. However most of this interaction consists of type checking at compile-time and thus requires no axiomatization. In addition to this primarily syntactic type-checking, our language uses types semantically with overloaded routines. The version of the routine to be used in a particular invocation is selected according to the actual parameter types. This use of types requires axiomatization; however, the types of actual parameters are determinable at compile time and thus the handling of overloaded routine calls poses no problem.

The representation of a variable, like its type, can be used for the detection of certain program errors during compilation. As with type error checking, this feature does not need to be captured by the axiomatic semantics. However, representation is also used for detecting certain run-time errors—e.g., a value out of the intended range of a variable. This use is relevant to exceptions (see below). Finally, scale and precision qualifiers determine the actual effects of assignment for numeric variables. This last feature of representations is not difficult to incorporate in the assignment axiom since these qualifiers are manifest constants.

3. *Modules*

Some of the forms of encapsulation cited in the literature (e.g., [34] and [35] have been accompanied by rather complex proposals for axiomatization. This source of complexity is not an issue for our module because it is a compile-time resolvable scoping rule. The indirect method of axiomatization by translation to an equivalent block is thus available. Even if we arrive at no more direct method, we are assured of an axiomatization that is only marginally more complex than that of an Algol-60-style scoping rule. (The same argument assures that our module templates will not complicate the formal semantic definition.)

4. *Blocks*

The axiomatization of the block is essentially a matter of describing a systematic renaming of variables declared at block entry to avoid name clashes; the only additional feature needed is a "sequence axiom" to define the effect of executing the so-transformed block as the composition of the effects of sequentially executing its constituent statements.

5. *Assignment*

Assignment is readily axiomatized (in the standard manner using syntactic substitution) for the simple data types we have allowed. ARRAYs, RECORDs and SETs require some special treatment. However, adequate techniques are known for these types [33], [34]. As already mentioned, special provisions will also be necessary to deal with representation of numeric variables in assignments.

6. *Control Jumps*

The GOTO, EXIT, and RETURN statements are amenable to clean axiomatization where jumps out of lexical scopes are forbidden. (Note that jumps *into* lexical scopes are prohibited by the language itself.) For jumps out of lexical scopes, see below.

7. *Function Declarations and Calls*

The pure function in this language can be axiomatized in much the same way as has been done in Euclid [36]. Both here and in Euclid, such functions have no side effects, but they may access global variables (and therefore are not single-valued mathematical functions of their parameters). This dependence on implicit parameters results merely in a minor notational inconvenience for the axiomatization.

8. *Procedure Declarations and Calls*

A variety of axiomatic rules has been developed for procedures (with various types of parameter mechanisms, with and without side effects, recursion, etc.) in other languages [33], [34], and [37]. The main difficulty here is produced by accounting properly for aliasing of VAR parameters. The exclusion of pointer variables from the axiomatized subset, together with the restrictions made by the language, will serve to eliminate this problem. We also exclude value-returning procedures from the subset to eliminate the possibility of side effects in expressions. (However, see also C-3.) We are confident that minor modifications in existing types of procedure call rules, [34], [37] will be applicable to our procedures under these restrictions.

9. *Assertion*

We propose to use the Euclid assertion axiom (11.3 in [34]). The axiom will cover only the case where there is no run-time check of the assertion. In the case where the compiler produces checks and these are enabled during run time, the axiomatization would have to interact with the exception handler (which we have excluded from this subset).

10. *Conditional and Iterative Statements*

The axiomatization of conditional and iteration statements of the kinds present in our language is well understood and straightforward. See, e.g., [33] or [34].

11. *Expressions*

The only problems in an axiomatic treatment of expressions are those that arise from the possibility of side effects, and the interaction of such side effects with aliasing and reordering of expressions by an optimizing compiler. We have effectively eliminated aliasing as noted above. Our restrictions on procedures (see above) serve to rule out side effects. Relaxation of these restrictions is discussed in Section C.

C. *Possibly Axiomatizable Features*

1. *Parallelism*

The language provides a variety of features for parallel processing—process definition, invocation and termination, monitors, signals, and low-level synchronization (semaphores). A full axiomatization of parallel processes is not feasible for this project. It may be possible to treat a subset that includes processes and monitors, but with restrictions on the use of shared variables in parallel processes. See, e.g., [40] or [39].

2. *Pointers*

Pointers were excluded above in order to avoid explicit attention to aliasing in an axiomatic definition. They can be included in an enlarged axiomatic subset but substantial research may be required to arrive at a satisfactory treatment. For example, it may be possible to treat pointers as references to a global array [41]. (Note that the language rules out aliasing of array elements but not aliasing of pointed-to anonymous variables.)

3. *Side Effects in Expressions*

Value-returning procedures (ruled out above) provide a mechanism whereby the evaluation of expressions can produce side-effects. Such effects can be deeply hidden—e.g., through procedure call hierarchies—so that transitive closure is required to determine the set of all variables that may be subject to side-effects in a procedure call. This requirement leads to extremely awkward axiomatization, but does not rule it out entirely.

The interaction of side effects with subexpression reordering produced by an optimizing compiler is another pitfall to axiomatization. Our solution envisions a prohibition of such optimizations except where side effects can be guaranteed not to occur. For expressions that are evaluated in a fixed (left-to-right) order, the axiomatic treatment of side effects can be facilitated by applying the axiomatization to a virtual (rewritten) form. This form is derived from the original expression by systematically rewriting it into a sequence of "flattened" assignment statements to new temporary variables.

4. *Jumps Out of Lexical Scopes*

There are technical difficulties in applying the usual axioms for jumps to GOTOs, EXITs, and RETURNS that leave a lexical scope. While these problems may, in fact, turn out to be minor ones, we have not investigated the difficulty of modifying the rules to cover such cases.

5. *Exception Handling*

Exception-handling facilities are a relatively recent development in programming languages. Our exception handlers have a single simple form and semantics, and this may make them amenable to formal axiomatization. However, the interaction of the exception

mechanism with other parts of a language is very extensive. A determination of the feasibility of the inclusion of exception handling in the axiomatizable subset is a research issue. (Inclusion of exception handling in the axiomatization would, of course, permit us to cover the "checked" assertion statement [see above] and other run-time errors [e.g., variable out of range].)

D. *Facilities that Will Remain Unaxiomatized*

We feel that axiomatization is infeasible or inappropriate for the following aspects of the language:

- Implementation-dependent features (Section XIV of the Language Report)
- Compile-time resolvable features (e.g., type-checking and those representation checks that can be carried out during compilation.)
- *Simulated time* facilities of the language

The *simulated time* feature is included here in accord with our policy of being conservative in our estimates. It may turn out that this language feature can be provided with an axiomatization adapted from that used for parallelism and differing in its dependence on a simulated time clock.

II THE FEASIBILITY OF THE USER MANUAL

Our inspection of a range of user manuals indicates that the quality of a User Manual is strongly influenced by the quality of the underlying language definition report. It is our assessment that the best user manuals, particularly the Pascal user manual [22], appear to be those developed more or less directly from a high quality language definition report.

We envisage that the language documentation available to a user will contain three components:

- A Language Primer
- A language Definition Report
- A User Manual.

A. *The Primer*

The Primer will consist of a graded series of examples with annotations and explanations of language features. It will assume that its reader can program in some high-level language (e.g., Fortran, Algol, Pascal, or Jovial), and will include examples of programs that can be readily understood by any such reader. By gradually introducing language features and demonstrating their use, the Primer will bring the programmer to the point at which he will be able to write programs using much of the language. He will also have sufficient understanding of the language to be able to consult and understand the User Manual for further definition of language features he is already using, and also for language features not described in Primer.

Having read and understood the Primer, the programmer will not need to refer to it again; he should be encouraged to consult the User Manual. The Primer as here envisaged will not be appropriate for teaching a nonprogrammer how to program, nor will it teach the appropriate use of concepts such as abstraction or asynchronism.

B. *The Language Definition Report*

The Language Definition Report can be used as a reference by programmers, but most will find the User's Manual more helpful. The intent of this report is to provide a complete, precise, and succinct definition of the language. Those who are more interested in the language than in its use—for instance, programmers responsible for compilers and support software—will use the Report.

C. *The User Manual*

The User Manual will be based on the structure and content of the language definition report, in places even using the text of the report. But the User Manual will have a less formal, less terse style, with explanations of the intentions behind the language design and of appropriate methods for using the language features. The User Manual will contain many examples, both to indicate typical program structures, which might guide programmers in their own designs, and also to demonstrate fine points of the language.

The User Manual is intended to be a reference document, and is extensively cross referenced.

D. *Feasibility*

We see no problem in preparing an initial User Manual for the language during Phase II of the project. We have already prepared and submitted a tutorial introduction to part of the language, with a graduated series of examples that could form the basis of a Primer for the language, and these we will extend and improve. We believe that the material in our Language Report can be straightforwardly rearranged and supplemented by tutorial material, examples, and motivational material (such as is found in our Tutorial and Programming Examples), to yield a structure appropriate for a User Manual. Much of the material in the Design Discussion relating to the intended use of language features will similarly be appropriate for integration into the User Manual. We have been much aided, in the preparation of the Language Report, by the comments of our consultants on that document. We intend to pursue the same approach for the User Manual.

III THE FEASIBILITY OF THE TEST TRANSLATOR

Our test translator will be a compiler from the language to a simulated idealized machine. This approach is a little more complex than a pure interpreter would be, but it will better serve the overall needs of language development and evaluation, and it is entirely feasible within the level of effort and scheduling of this project. We choose to build a compiler for two reasons. First, the slow execution speeds of an entirely interpretive system might interfere with the successful evaluation of the language. Second, the treatment of problems of compilation will provide feedback on language design decisions and on the feasibility and cost of production compilation.

A. *Translator Structure*

Our translator will have three parts. The first part will be a syntax-directed transduction parser that produces an internal representation of programs, together with some further transformation and error detection, not conveniently expressible in a transduction parser, such as name-scope checking.

The intermediate language that results from this part will be compiled to an idealized machine modeled after that designed by Wirth for the Pascal-P System [23]. Finally, we will simulate this idealized machine on our DEC TOPS20 KL-10 Computer. Such a compiler/simulator is quite feasible; the areas in which a production compiler is most complex—optimization and code generation—are substantially simplified in the proposed prototype by the use of an idealized object machine.

B. *Software Tools*

We have a number of sophisticated software tools available to assist in the development of a prototype implementation of the language. We intend to use the Interlisp Programming System to host our test translator. In addition to an augmented Lisp language, Interlisp provides a complete user environment with debugging packages, file system handlers, measurement routines, and a program analysis subsystem (Masterscope) that will support the implementation of the test translator. We have extensive experience with Interlisp, which leads us to believe that its use will result not only in the timely development of a test translator, but also in the inclusion of debugging facilities that will aid the government's evaluation of the language.

C. *Parsing*

We will use a system that runs in Interlisp to produce Interlisp parsers for LALR(1) grammars. The system includes facilities for producing a very flexible finite-state lexical analyzer. By associating a transduction with each rule of a grammar, this system can be used to generate a parser that translates programs from external syntax to an internal form suitable for driving a compiler.

The parsers produced by this system have the benefits common to LR parsers of detecting an error at the point in the input text where it occurs. Thus, if instead of the legal syntactic fragment

(i + j) * (j + k),

our parser is given

(i +) * (j + k),

it will issue an error message such as

ILLEGAL LEXEME ")" IN THE CONTEXT

(i + !!!) !!! * (j + k)

WHERE "(", number, or identifier WOULD BE PERMITTED

thereby making the exact nature of the error clear to the programmer. This diagnostic facility will be a part of our test translator.

The grammar that is given as Appendix B of the Language Report is structured for clarity of presentation and is not LALR(1). However, we have developed an LALR(1) grammar that will be the basis of the LALR(1) parser of the test translator. Therefore, we are confident of being able to include the good syntactic error recovery mechanisms that are possible with LALR parsing in our test translator.

D. *Feasibility*

Given the availability of these tools, our experience in the implementation of language processors, and the estimates of our consultants based on their experience with similar efforts, we are confident that the test translator can be implemented according to the time schedule for Phase II.

Since all computation will be done on a DEC KL-10 under TOPS20, there will be no difficulty in installing the test translator on any such machine accessible to the government's language evaluators.

IV THE FEASIBILITY OF PRODUCTION COMPILATION

In designing a language for which efficient, high-quality production compilers are feasible, we have kept the design close to Pascal, an efficiently implementable language. Where we have added features in response to Ironman requirements, we have kept these additional features simple in design, few in number, and close to the style of Pascal. Languages become difficult to compile and inefficient to use primarily through the introduction of many complex, special-purpose mechanisms. A feature that must be supported by adding one or two fields to each procedure frame or to each process activation record will be very detrimental to the efficiency of compiled programs. Language extensions that are individually well understood and seemingly independent have a way of interacting to present great difficulties for the compilation of efficient code. For these reasons, we have kept the language lean, choosing simple extensions to the base language whose impact on compilation efficiency will be minimal and whose feasibility has been demonstrated through experience with existing compilers.

Undoubtedly there will be many short sections of program for which hand coding in assembly language can demonstrate an advantage over compiled code. The objective of the language design is to permit complete embedded systems to be developed in the language with a size and performance equivalent to that of the complete system programmed in assembly language. We regard the size of the compiled code as being the primary indicator of efficiency, not only because terse compiled code usually runs faster than verbose compiled code, but also because of the history of systems that are limited primarily by the available storage. In the handling of machine interrupts, performance is clearly more critical than code size, and there are many areas where code size and performance are equally important.

A. *Efficient Compilation of Basic Language Features*

This subsection considers the compilation of the basic features of the language, features which are present in Pascal and many other languages. The compilation of these features must be considered both to show that efficient compilation of them is possible, as demonstrated by prior implementations, and also to show that the design of this language does not contain features which preclude such techniques.

Consideration of previous language implementations shows that the storage space efficiency of new languages is most likely to be determined by the size of the stack frames and the activation records, and by the code size. The code size, which also affects performance, is likely to be affected particularly by accesses to variables and by the procedure call, entry, return, and parameter transfer sequences. All of these depend critically on the layout of information in storage and the mechanisms used to allocate space and to locate items. Two possible allocation

strategies are considered here:

- A stack based allocation strategy, derived from the implementations of Pascal,
- A substantially static allocation strategy, possibly more suitable for embedded systems.

We also consider the implementation of variable length arrays, both those whose length is determined only at scope entry, and also the length unresolved formal arrays.

1. *Variable Length Arrays*

Arrays whose index ranges are determined only on entry to the scope of their declaration cannot be allocated storage by the compiler as it allocates storage for the other local variables of the scope. Such arrays must be allocated storage on the stack at scope entry and accessed through a pointer, which is located amongst the local variables. The allocation and deallocation of a scope entry array must also update the top of stack. It would be possible to allocate all arrays in this manner, but minimal overheads require preallocation of known sized arrays at fixed positions within the local storage allocation, even when the limited addressing of the computer requires indirect access to them.

The initial design for Pascal provided no formal array parameters with the index ranges unresolved until procedure call, though this restriction has now been removed. Implementation of length unresolved formal arrays is quite straightforward. Either the index ranges can be passed as hidden parameters, or the array reference passed to the procedure can be an array descriptor with the index ranges within it. Another possibility is that space be allocated as a part of the array space allocation to accommodate the index ranges. The first of these incurs additional overhead on each call of a procedure with a length unresolved formal array parameter, but only for calls on such procedures. The last of these involves no call sequence overhead, but requires additional space to be allocated for every array, since only global optimization of the program can establish that an array will not be passed as an actual parameter to a length unresolved formal array parameter.

B. *A Stack Based Allocation Strategy*

In this strategy, space is allocated on a stack as it is required. Typically space is allocated on the stack at each procedure or function entry, and also at the declaration of arrays whose dimensions are determined only at scope entry. A separate stack is required for each process as is discussed below. On each procedure exit the space allocated on entry can be recovered, the nested nature of procedure calls ensuring that space is deallocated in the reverse order to that in which it was allocated, permitting the use of a stack. There is no need in general to allocate space on entry to inner blocks of the procedure because the space requirements can be fully anticipated by the compiler and combined with the allocation made on procedure entry.

The allocation of space only as it is required is of course the essence of the space savings obtained with this method, and is particularly convenient for recursive procedures. But it also results in different storage being allocated for different calls of a procedure, and thus difficulties in accessing variables. Variables that are global to the whole program cause no access difficulties, while variables local to the procedure are at the top of the stack and can be accessed relative to the register holding the pointer to the current stack frame. Given the limited addressing ranges of many computers, the need for accesses to the local variables to be relative to a register is inevitable and causes little overhead on modern computers with adequate addressing modes and register sets. The primary problem involves access to variables declared in some enclosing scope, not local to the procedure but not global to the whole program. The

original solution to this problem, a display of registers, one for each enclosing scope to record the addresses of their current allocations, is expensive to maintain during procedure entry and exit. The cost of the display is particularly inappropriate because it is observed in well structured programs there are very few accesses to variables requiring use of the display.

For the implementation of Pascal, Wirth and Hoare invented a new method of access to non-local variables. On procedure entry, a stack frame for that procedure is allocated on the stack. This contains:

- The space required for local variables and temporaries,
- The pointers to scope entry sized arrays,
- The parameter references or values,
- The link for the return to the caller,
- The address of the caller's stack frame,
- The address of the stack frame of the lexically enclosing scope,
- The address of the stack frame to be used for any procedures to be called (top of stack).

The link and the address of the caller's stack frame are needed to return control to the caller at the end of the procedure. The address of the stack frame of the lexically enclosing scope is used to access non-local variables, by following the chain of these addresses until the address of the stack frame belonging to the required scope is reached and the variable can be accessed. This access mechanism might appear to be inefficient, but can actually be highly optimized by a production compiler. The compiler must determine what non-local variables are accessed within the procedure, and thus what enclosing scopes must be addressable. Code is generated, within the procedure entry or elsewhere, to scan the chain of addresses and to record the addresses of the needed stack frames, in registers or local storage as determined by the register allocation optimizer. All reference to such variables can now be made relative to these stack frame addresses. Experience with the implementations of Pascal indicates that this scheme is usually very efficient.

1. *Procedure Call and Entry Instruction Sequences*

The actual instruction sequences required for procedure call and entry are highly dependent on the particular computer in question. The number of registers, the addressing modes available, and the size of the address field of an instruction all influence the code. Therefore we do not give here the code sequences for any particular machine, but rather describe the instructions required in terms of an idealized machine presenting no complications in any of these characteristics. The code sequence for procedure call is:

- Place parameter values, or references to them, where they will be accessible to the called procedure, on the stack or in registers. Note that representation conversion may be required. (Unknown number of instructions.)
- Load into a register the address of the stack frame of the scope lexically enclosing the called procedure. (One instruction, sometimes unnecessary.)
- Subroutine call. (One or two instructions.)

- Distribute the result and OUT parameters, possibly performing representation conversion. (Unknown number of instructions.)

where we assume that at least the address of the current stack frame is already present in a register.

The code sequence for procedure entry is:

- Load current stack frame address from stack frame of caller. (One instruction.)
- Save link, stack frame address of caller, stack frame address of lexically enclosing scope, other registers to be saved. (One or more instructions.)
- Move current stack frame address. (One instruction.)
- Check for stack overflow, and if necessary calculate stack frame address to be used for procedures to be called, storing in stack frame. (Three to five, plus one, instructions.)
- Scan lexical chain, if done within entry sequence, storing in stack frame any non-local stack frame addresses needed for variable access. (One instruction per level plus one per address stored.)

A highly optimizing compiler may radically modify this sequence according to the needs of the actual procedure. It is important to note that the call sequence is very short, even at the expense of a longer entry sequence and an additional entry in the stack frame. Procedure calls are very frequent in high level programs, and the overall code size is critically dependent on a brief call sequence, for there will be more, probably many more, procedure calls than procedures.

2. *Interactions with Other Language Features*

The stack-based allocation stategy described here interacts with the provision of recursive procedures, the provision of parallel processes, the presence of exception handlers, and storage allocation for dynamic variables.

Recursive procedures are present in Pascal and are fully accommodated by this mechanism. Parallel processes require a separate stack for each process, and space must be allocated for this stack on entry to the scope within which the process is declared. The allocation of the stacks of parallel processes, on scope entry, within the stack of their parent scope, is known as a *cactus stack allocation*. Strategies which allocate stack space for processes on activation, rather than on declaration, can be significantly more economical on space, but they must be much more complex and can not simply allocate stack space for processes within the stack of their parent scope.

The mechanism described above for accessing non-local variables is entirely applicable to the parallel process situation with cactus stacks. The chains of addresses of stack frames for lexically enclosing scopes enables each of several processes to see distinct variables for variables defined within the processes, and yet see a common variable declared within a scope that encloses both of them.

The interactions between the allocation strategy and the exception handler are discussed below. Dynamic variables can not be allocated space on the stack like other variables, since they may survive the scope within which they are created. The feasibility of a Zone or Heap

implementation, at least for non-time-critical applications has been demonstrated by the various implementations of Pascal.

C. *A Static Allocation Strategy*

A static allocation strategy may result in a larger storage requirement than a stack based allocation strategy, for there may be lost opportunities for the sharing of storage. Nevertheless a static allocation strategy may be more suitable for the compilation of complete embedded systems, which may benefit from the smaller code size and performance overheads on procedure call and entry, and which must provide storage for the maximum possible requirement to eliminate the risk of stack overflow. However much of the advantage of a static allocation is lost on small computers whose instructions have a short displacement field, requiring the use of relative addressing despite the static allocation.

In a fully static allocation strategy, the storage location for every variable is determined during the compilation or linkage editing of the program. In practice the allocations for arrays whose dimensions are not determined until scope entry, and for the local variables of recursive procedures, must still be made on the stack, while allocations for dynamic variables must still be made in a zone or heap. A static allocation requires of course that space be allocated in advance for the maximum number of instances of a process, and for the maximum size for each process. Procedures which are called from more than one process must be allocated distinct local storage for each such process. These allocations can be static, but the procedure must access its local variables relative to an address passed to it in the calling sequence, destroying some of the advantage of a static allocation. Careful (global) optimization can arrange for this base address to be common to many procedures that may be called by the processes, the allocations for each procedure allowing for the allocations to the other procedures, thus minimizing the amount of base address manipulation required. But such global optimization, and even the need to know that a procedure is called by multiple processes, essentially precludes separate compilation, a severe disadvantage during program development but less important for the compilation of the highly optimized operational version of the program.

The static allocation of variables does not however imply that every variable must be allocated distinct storage. The scope rules of the language and examination of the program can ensure that some variables can never coexist, and thus can safely be allocated to the same storage. Indeed it is possible, though unlikely, that the static allocations made by a compiler could be more compact than the dynamic allocations of the stack based strategy, which is forced into a slightly suboptimal allocation algorithm by the need to minimize the procedure call sequence, and which must allocate space for administrative overheads It is certain that the object code size will be smaller for a static allocation.

For a static allocation, each procedure is allocated space for:

- The local variables and temporaries,
- The link for the return to the caller,
- The pointers to variable length arrays allocated on the stack.

Recursive procedures are provided with this allocation on the stack, while a single location, statically allocated to the procedure, locates the stack allocation of the current instance of the recursion. In general, on entry to a recursive procedure, this pointer location must be saved in the allocated space on the stack, and it must be restored on exit.

1. *Procedure Call and Entry Instruction Sequences*

The code sequence for a simple procedure call, under static allocation and assuming a computer which does not need a base register to access its locals, is:

- Place parameter values, or references to them, into the local storage of the called procedure. Representation conversion may be required. (Unknown number of instructions.)
- Subroutine call. (One instruction.)
- Collect from the local storage of the called procedure the result and out parameters, possibly performing representation conversions. (Unknown number of instructions.)

The code sequence for procedure entry is even simpler, comprising:

- Save link and any other registers to be preserved. (One or more instructions.)

In practice many small computers need base registers to access their local variables and must use more complex call and entry sequences.

2. *Interactions with Other Language Features*

The interactions of static allocation strategies with other language features have been discussed above.

D. *Language Features Fully Resolved During Compilation*

Novel language features which can be fully resolved during compilation are unlikely to have a detrimental effect on the efficiency of the code produced by production compilers.

1. *Strong Typing*

The concept of strong typing, though new to production compilers, causes no problems to the compiler. The concept serves only to restrict the operations which must be compiled an thus assists in the generation of efficient code, while the checking required during compilation corresponds only to the checking that would have been required to detect type conversions in a weakly typed language. On balance, strong typing probably eases the implementation of production compilers.

2. *User-Defined Types*

Like all typing, the user-defined types are fully resolved during compilation. They add slightly, but not significantly, to the complexity of the compiler's symbol tables and of course require the processing of type declarations. Again since user-defined types may be regarded as a shorthand, and since the type checking serves only to restrict the operations permitted, no degradation in code quality is expected. Note particularly that the optimization phases of a production compiler will follow the checking and expansion of user types, and thus these typing constraints on the programmer will not necessarily preclude appropriate optimizations.

3. *Alias Checking*

As discussed in the Design Discussion, alias checking requires the construction of a complete transitive closure of the program segment under compilation. For all normal programs this should cause no problems, but pathological programs may impose a significant computational load on the compiler, possibly necessitating complex, efficient closure algorithms.

4. *Overloading*

The mechanisms required to resolve overloading are already present in all compilers, although in a specialized form, to resolve infix operators. The feasibility of the mechanism is entirely dependent on the strong and explicit typing of the language; for example, without the explicit typing required of literal constants, overloading could not be resolved. Since overloading merely provides the user with the ability to use the same name for procedures and functions, which otherwise would have needed different names, no degradation in code quality is expected.

5. *User-Defined Infix Operators*

With the typing and overloading facilities already present, even in simpler compilers, very little extra is required in the compiler to allow the user to define infix operators. Indeed the more systematic approach imposed by offering the facility to the user may, by eliminating arbitrary special features or shortcut compilation algorithms, improve the reliability of compilers. Extensions have been required in the syntax to permit the use of infix symbols at some places where routine names are permitted.

6. *Modules*

In choosing an encapsulation mechanism, we held uppermost the need to apply a proven technology, such as that of the Modula module [17], particularly for the design of a language for embedded systems. while interesting new discoveries and capabilities may result from the exploratory work on parameterized, multiple instance class generators, along the lines of the Euclid module [20] and the Alphard form [24], the need for these newer developments in the design of efficient embedded systems is not yet clear. We chose the Modula module [17], a mechanism that is comparable in power but simpler to implement than the newer schemes, since Modula modules neither generate multiple instances nor accept parameters. They require no heap for space allocation, and no special stack manipulation during variable instantiation. Modules simply define accessible scopes for user defined identifiers; they disappear completely during compilation. The feasibility and efficiency of this feature is demonstrated by several successful implementations of Modula. While it may, in the future, develop that the more advanced technology is applicable and effective, we chose to adhere to a proven technology, with which we can have full confidence in the feasibility of production compilation at the level of efficiency required for embedded systems.

7. *Module Templates and Instances*

To retain many of the benefits of parameterized modules, the language includes the module template, a limited semantic macro. Inclusion of templates can be guaranteed not to degrade run-time performance of programs since all uses of templates can be eliminated by a compiler phase that expands template instances by substitution. On the other hand, compiler writers need not include a special-instance expansion phase, and may even choose to optimize by sharing items defined by separate instantiations of a module template.

8. *Synopses, Segments, and Separate Compilation*

The feasibility of the proposed separate compilation facility is demonstrated by experience with LIS [26] and Mesa [27], which also use the notion of an interface specification, or synopsis. The design of the facility for this language is however simpler than those in LIS and Mesa. The unit of compilation is the module, the natural segmentation unit of the language. No special semantic constructs or scope rules have been introduced to support separate compilation. The facility has been designed to place minimal burden on the linkage editor; there are

- No interface checks that can not be performed at compilation time,
- No dependence on the order of segment compilation,
- No requirement for transmission of compiler-produced tables between compilations,
- No special rules or complications to add complexity to the compiler and the environment.

Note that our language requires that entities may only be imported into separate compilation units using the FROM option to specify the name of the separate compilation unit from which they come. In consequence, it will be straightforward in each implementation of the language to adopt simple naming conventions, appropriate to the local file system, for determining where the most recent version of the synopsis for a named module is to be found.

E. *Language Features Requiring Object Code Generation*

Only two features of the language require code generation beyond that already required for Pascal and other similar languages.

1. *Processes, Monitors, and Parallel Processing*

The feasibility of features included to support parallel processing has been demonstrated by experience with other multiprocessing languages that are based on Pascal. We use Modula's mechanisms, the monitor and signal (of which device signals are a special case), for mutual exclusion and synchronization. Implementations of the Modula language generate code for parallel processing, low-level input-output, and interrupt handling which is essentially as efficient as that for assembly language coding of such operations, and is very much more efficient than the use of most operating systems. Our generalization of Modula processes is in the style of Parallel Processing Pascal [25] and Concurrent Pascal [1], both of which have been used to implement effective operating systems. Our simulation facilities are adapted from SIMONE [21]. Of course, experience with these other languages is applicable to our design only because we have kept it as free of special rules and redundant mechanisms as possible.

2. *Exception Handling*

An intricate exception-handling mechanism in a language is especially likely to add overhead to compilers and to run-time systems. Our design reflects experience with Mesa, which has one of the more advanced exception-handling facilities that has been implemented successfully in a compiler. Our scheme is simpler than Mesa's, however, more closely resembling that of BLISS-11, in that the possibility of an exception does not interfere with the compiler's treatment of expression or statement structure. Only trap handlers can return into the middle of an expression, and this is an explicitly implementation-dependent facility that does not impact code generation. Exception handlers are not parameterized, so special scope rules for exception

parameters and special parameter binding facilities at run-time are not required. Even the introduction of a special conditional statement for exception handling, with sparse case selection and a special default case, need not add much burden to the compiler since interpretive processing of exceptions is a perfectly acceptable implementation option.

Simple implementations of the exception handling facility will record in the stack frame record, required on the stack for each procedure entry, an additional field to indicate the program scope and thus the exception handler to be used in the event of an exception. This field can also be used for other purposes, but strictly would not otherwise be needed. The setting of this field adds perhaps two instructions to the procedure entry sequence, to allow for exceptions which will very seldom occur. If this overhead is unacceptable, an alternative implementation is available that does not need this additional field. A table is constructed by the linkage editor that records the code address ranges for each exception handling scope and associates with each range the relevant handler. When an exception occurs, the table must be searched to match the current value of the program counter, and thus determine the scope and the required handler. Clearly the reduced overhead during normal operation is won at the price of greatly increased overhead for exception handling, a trade which will be advantageous for almost all systems.

F. *Feasibility of the Run-Time Nucleus*

The implementation of Modula on the PDP11 [17] requires a run-time nucleus of about 100 instructions, providing the scheduling, monitor exclusion, signal handling, interrupt handling, and process activation. The performance of this nucleus for the Modula language is as good as can be obtained on the PDP11 by assembler coding. If appropriate conventions in the use of the low-level input-output facilities are observed, as described in Section XIV of the Design Discussion, there is no reason to believe that the proposed language will be any less efficient. The run-time nucleus required to support the language will however be rather larger to provide these additional facilities:

- Semaphores
- Delay
- Simulated time
- Exception handling
- Stack manipulation for GOTO and EXIT
- Stack manipulation for recursive procedures
- Heap management.

Semaphores can share code with that already required to provide signals. The delay facility is additional, and the cost of its provision will depend on the expected number of delayed processes and the efficiency required. On some machines it may be appropriate to provide a very simple delay mechanism, efficient for small numbers of delayed processes, using a nonprimitive delay procedure to provide delays for all except the highest priority processes. The simulated time facility is also additional, and will add significantly to the size of the run-time nucleus. Few operational embedded systems will use simulated time, and therefore it may be desirable to procure alternative versions of the nucleus, with and without simulated time facilities, for the smaller computers.

Exception handling is not provided in Modula, and the existing implementations of exception handling in PL/1 [38] and Mesa [27] are extremely expensive. The drastically simplified exception handling proposed here is more similar to that of Bliss11 and will have a more reasonable cost. We anticipate that it will be more important to ensure that the exception handling does not increase the procedure call, entry, and return sequences than to minimize the nucleus size. Consequently, we expect that the exception handling code of the nucleus will be required to determine in which exception-handling scope an exception occurs, by searching for the current value of the program counter in a table of exception scopes rather than by maintaining an entry in the stack frame, at all times, to indicate the current exception handling scope. We also anticipate that the case selection within an exception handler will be interpretive. Thus the exception-handling facilities may add significantly to the size of the run-time nucleus.

Both the GOTO statement and recursive procedure entry and exit require rather careful manipulation of the stack, probably best implemented in the nucleus. Much of this manipulation can be in common with the exception handling.

The zone required for dynamic variables is also additional to Modula, and here too the size of this facility will depend on the anticipated usage and the efficiency required. Again, it may be desirable to procure alternative versions of the nucleus, with and without the zone, for small computers intended for operational embedded systems.

There are some language features that are best implemented by procedures rather than by in-line code—for instance, the fixed-point division on many computers. We anticipate that these procedures will be provided like other library procedures, being linked into the program only if required rather than being built into the run-time nucleus. Only those operations that are so intimately involved in the scheduling or storage management that they cannot be separated, and those operations that will be required by essentially all programs, should be included in the run-time nucleus.

By comparison with the run-time nucleus for Modula, the minimal run-time nucleus for this language, excluding the simulated time and heap facilities, is estimated to be about 200 instructions for the PDP11. A full nucleus with efficient mechanisms for delay and the heap could easily run to 500 instructions, though some degree of compromise will be possible between size and speed for these facilities. A full nucleus hosted on an effective operating system would probably be rather smaller, depending on the complexity of the operating-system interfaces. We believe that the size of the minimal nucleus is reasonable for operational embedded systems, even for microprocessor systems. The larger nucleus provides services appropriate only for rather larger computer systems, for which its size is acceptable. There are obviously many other ways to configure a nucleus appropriate for a particular embedded application, including the omission of exception-handling facilities and limitations on fixed-point arithmetic.

G. *Conclusion*

Throughout the language design, we have been at pains to consider the detrimental interactions that can arise when powerful extensions are made to a programming language. We are confident that high-quality compilers can be written for the language because we have systematically rejected speculative features and have consistently avoided over-design.

V THE FEASIBILITY OF COMPILER VALIDATION TOOLS

In principle, the validation of a compiler for a language with a formal definition ought to rest on a proof of its compliance with the definition. Such mathematical demonstrations of compiler correctness for a language satisfying the Ironman requirements cannot reasonably be expected in the near future. Therefore, traditional validation methods will continue to be useful.

A. *Language Test Sets*

Experience with COBOL [28], Algol 60 [29], and Algol 68 [30] has shown the value of a well-designed set of test programs in certifying the consistency and correctness of a set of compilers for the same language. Once a prototype compiler is available, construction of the test set will be aided by the ability to check that all significant cases in the compiler have been exercised.

B. *Simplicity of the Language*

This approach to compiler validation is feasible only because the language is simple and its features are simply designed. Multiplicity and prolixity of features generate such a large number of potential interactions that it becomes impractical to certify their correct compilation by testing.

C. *The Language Control Authority*

Responsibility for compiler validation should be vested in a common language control authority, responsible for collecting and distributing test cases. Compiler developers and maintainers should be encouraged to document subtle design bugs they have encountered and to invent test cases likely to detect similar problems in other compilers. Test cases may be either programs expected to produce a verifiable output when executed, or examples intended to produce object code that can be inspected for correctness and quality. Test cases will also be required to confirm that invalid programs provoke appropriate error reports.

The language control authority should provide public test cases for use by compiler developers, but it should also reserve secret examples that can be used for acceptance testing of allegedly correct compilers.

D. *Standardized Compilers*

Of course, to the extent that the number of distinct compilers for the common language can be minimized, the problem of compiler validation is also reduced. It is feasible and advisable to standardize large portions of compilers for related machines, and to maintain such compiler families by sharing standard modules. This approach reduces the likelihood of errors and inconsistencies, and it simplifies the task of inventing test cases for a particular family since the common structure of the members can be taken into account.

VI THE FEASIBILITY OF LINKAGE EDITING

The separate compilation facility has been designed to avoid placing unusual burdens on the language environment in general and the linkage editor in particular. Communication of interface information between segments is performed entirely by the compiler. The only validation that must be deferred until linking takes place is the check that ensures all segments have been compiled using the same set of synopses. Section XIII of the Design Discussion describes one scheme by which this checking can be accomplished.

Because each segment has global scope, there is no need for the linker to manage references to dynamically created variables. All external bindings will be to static locations. Since the compiler has the full declarations of imported items, no adjustments need be performed at link time to cater to particular representations. The linker simply uses the names of imported routines and variables and the names of their defining modules to look up their addresses, and then positions these addresses in the object code.

Since every object file will contain item and segment identifiers for each external entity it requires, a library look-up scheme can easily be implemented, to spare the user's having to designate each segment in his program individually to the linker.

VII INTERACTIVE SOURCE-LANGUAGE DEBUGGING AIDS

The provision of debugging aids is largely the responsibility of the environment, aided by symbol tables generated by the compiler and linkage editor. Examples exist of very elaborate symbolic debugging environments, notably the Interlisp system [31]. Such systems require years of development and are feasible only on a rather large computer. A more modest system, which could be made available quickly and is suitable for use on small machines, is described in [32].

The features that might be expected of any symbolic debugging environment include:

- Postmortem dump
- Flow of control trace
- Full trace with values assigned
- Interactive insertion of trace enables and disables
- Interactive insertion of breakpoints
- Interactive interrogation and modification of variables.

Facilities such as interactive modification of the program and incremental program entry and compilation would require major reconsideration of the language and have not been evaluated.

A. *Postmortem Dump*

The postmortem dump provides:

- The current location of control in the program
- The current nested-procedure call chain, including the location of each of the calls and the actual parameters
- The values of each variable existing at the time of the dump.

All this information can be presented in the form in which it is expressed in the source program, using symbolic names and data structures matching those of the source program. Ideally

the postmortem dump should seldom be needed, errors in the program being located using the interactive aids without the need to print long dumps.

B. *Flow of Control Trace*

The flow of control trace records the flow of control within the program, but does not record assignments or the values of expressions. This trace must record all procedure and function calls, the iterations of iterative control structures, and the cases selected in conditional control structures. Execution of a labeled statement causes recording of the label. It is envisaged that procedure calls will be provided by each implementation to enable and disable this trace; if such calls are included within conditional statements, the tracing can be made to depend on the actions of the program. It is also envisaged that such enabling and disabling procedure calls would be inserted into the program by the interactive aids. The scope and module structure of programs will allow precise control over the areas of program text within which tracing is enabled. The flow of control trace is valuable because it permits the programmer to follow the progress of his program but does not present to him the excessive quantity of information produced by a full trace.

C. *Full Trace*

The full trace reports the information reported by the flow of control trace and also reports the symbolic names of all locations to which assignments are made, together with the value assigned. The values of expressions controlling conditional and iterative structures are reported as are the values of the input and output parameters and the names of reference parameters. The full trace can be enabled and disabled in the same manner as the flow of control trace, and use of it without such selective controls is poor practice.

D. *Interactive Aids*

The interactive aids will include facilities for the insertion of breakpoints and of trace enables and disables into a program using the symbolic statement structure of that program (i.e., before or after any program unit that is syntactically a statement). The conditional statements, used to condition these enables and disables within a program, would not be available for interactive insertion, for, in a highly structured system, evaluation of such conditions would probably involve user-defined operations on user-defined types, operations that might change the meaning of the program. Interactive commands would also be provided to display the values of symbolically named variables, and also to change such values. The values of all the variables of a local scope might also be displayed. The interactive system should also contain a scrolling system to allow the user to reexamine past output of traces or values no longer visible on his display terminal.

E. *Asynchronous Parallel Processes*

The debugging of asynchronous parallel processes is still an unsolved problem. It is believed that the SIMULATED_TIME blocks of the language, coupled to procedures that reproduce the behavior of real-time devices, will go a long way towards providing the required facilities. Further, since interactions between processes should be through monitors, a tracing of all monitor procedure calls, reporting the calling process and the actual parameters, will provide the basis for the analysis of the interactions between processes.

F. *Feasibility*

We believe that any language designed to meet the requirements of Ironman will be suitable for interactive source level debugging. There will be no variables for which no symbolic name is defined, and no variables whose type or structure cannot be determined. Considerable support for symbolic debugging will be required from the compiler and linkage editor. The compiler, when compiling for debugging, must refrain from optimizations that destroy the statement structure of the program, and must generate an elaborate symbol table relating the source and object codes of the program. The linkage editor must link this symbol table together with the program object code as it builds the executable program. This technology is well understood and will present little difficulty on an appropriate computer.

PART SIX

ANALYSIS OF COMPLIANCE

CONTENTS

I SUMMARY

We have designed a language meeting the requirements of the Department of Defense for the development of embedded systems, as presented in Ironman [10]. We find that Ironman presents a successful analysis of the needs of DoD, and of the language features appropriate for meeting such needs in the current state of the art of language design.

In our design, we have found it possible to satisfy completely the Ironman requirements. Of course, our efforts have revealed instances of trade-off between specific requirements of Ironman, and we report such trade-offs here and in our Design Discussion. But, in all cases, we believe that it has been possible to design a language that complies with the intention of Ironman and the needs of DoD. In some places, Ironman imposes requirements on the implementation of the language, rather than on the language itself. We have complied with these requirements by designing a language that is thoroughly compatible with such an implementation. The language design itself does not impose these requirements on an implementation; clearly, such requirements will be imposed in the procurement of production implementations of the language.

The Ironman requirements comprise a set of general design criteria (Section 1) and a more detailed set of specific criteria (Sections 2 to 12). Throughout our analysis of the requirements and our design of the language, we have granted precedence to the general design criteria. In a very few cases, noted below, a mechanism specified in Ironman would interact with other language features to produce an overly complex, inefficient, or error-prone design. We have, in such cases, always chosen a simpler mechanism to meet the substance of the requirement expressed in Ironman. Our analysis of the language has satisfied us that it can readily be used to implement—with simplicity, reliability, maintainability, efficiency, and machine-independence—all the embedded-system aspects that we have been able to consider. If we have erred in such choices, we have erred on the side of safety; experience shows that it is very much easier to add features to a language with a clean and simple design than to add quality to a language with many intricately related or unrelated features.

We have recently received a further list of general criteria, to be used for the evaluation of the submitted designs. We are satisfied that these general criteria correspond closely with those upon which our design is based, and we are happy to have our design evaluated against these criteria.

In the sections below, we consider each Ironman requirement individually, to demonstrate the extent and method of our compliance with the requirements.

On the following pages, we present a tabular summary of the nature of our compliance with the Ironman requirements. The table lists the requirements of Ironman and the extent of our compliance with the requirements. Our compliance is reported in one of four categories:

A. Exact Compliance with the requirement, by means of the mechanism specified.

B. Effective Compliance with the requirement, by means of an alternative mechanism.

C. Substantial Compliance with the requirement, including compliance with all aspects of the requirement necessary for embedded systems.

D. Non-Compliance with the requirement.

	A	B	C	D
1. GENERAL DESIGN CRITERIA				
1A. Generality	•			
1B. Reliability	•			
1C. Maintainability	•			
1D. Efficiency	•			
1E. Simplicity	•			
1F. Implementability	•			
1G. Machine Independence	•			
1H. Formal Definition	•			
2. GENERAL SYNTAX				
2A. Character Set	•			
2B. Grammar	•			
2C. Syntactic Extensions	•			
2D. Other Syntactic Issues	•			
2E. Mnemonic Identifiers	•			
2F. Reserved Words	•			
2G. Numeric Literals	•			
2H. String Literals	•			
2I. Comments	•			

	A	B	C	D
3. TYPES				
3A. Strong Typing	•			
3B. Implicit Type Conversions	•			
3C. Type Definitions	•			
3.1. NUMERIC TYPES				
3-1A. Numeric Values	•			
3-1B. Numeric Operations	•			
3-1C. Numeric Variables	•			
3-1D. Floating-Point Precision	•			
3-1E. Floating-Point Implementation	•			
3-1F. Integer and Fixed-Point Numbers	•			
3-1G. Fixed-Point Scale	•			
3-1H. Integer and Fixed-Point Operations	•			
3.2. ENUMERATION TYPES				
3-2A. Enumeration Type Definitions	•			
3-2B. Ordered Enumeration Types	•			
3-2C. Boolean Type	•			
3-2D. Character Types	•			
3.3. COMPOSITE TYPES				
3-3A. Composite-Type Definitions	•			
3-3B. Component Specifications	•			
3-3C. Operations on Composite Types	•			
3-3D. Array Specifications	•			
3-3E. Operations on Subarrays		•		
3-3F. Nonassignable Record Components	•			
3-3G. Variant Types	•			
3-3H. Tag Fields	•			
3-3I. Definitions of Dynamic Types	•			
3-3J. Constructor Operations	•			
3.4. SET TYPES				
3-4A. Bit Strings	•			
3-4B. Bit-String Operations	•			
3.5. ENCAPSULATED DEFINITIONS				
3-5A. Encapsulated Definitions	•			
3-5B. Effect of Encapsulation	•			
3-5C. Own Variables	•			
3-5D. Operations Between Types	•			

	A	B	C	D
4. EXPRESSIONS				
4A. Form of Expressions	•			
4B. Type of Expressions	•			
4C. Side Effects			•	
4D. Allowed Usage	•			
4E. Constant Valued Expressions	•			
4F. Operator Precedence Levels	•			
4G. Effect of Parentheses			•	
5. CONSTANTS, VARIABLES, and DECLARATIONS				
5A. Declarations of Constants	•			
5B. Declarations of Variables	•			
5C. Scope of Declarations	•			
5D. Restrictions on Values	•			
5E. Initial Values	•			
5F. Operations on Variables	•			
6. CONTROL STRUCTURES				
6A. Basic Control Facility	•			
6B. Sequential Control	•			
6C. Conditional Control	•			
6D. Short Circuit Evaluation	•			
6E. Iterative Control	•			
6F. Control Variables			•	
6G. Explicit Control Transfer	•			
7. FUNCTIONS and PROCEDURES				
7A. Function and Procedure Definitions	•			
7B. Recursion and Nesting		•		
7C. Scope Rules	•			
7D. Function Declaration		•		
7E. Restrictions on Functions			•	
7F. Formal Parameter Classes	•			
7G. Parameter Specifications	•			
7H. Formal Array Parameters	•			
7I. Restrictions to Prevent Aliasing	•			

	A	B	C	D
8. INPUT-OUTPUT FACILITIES				
8A. Low-Level Input-Output Operations	•			
8B. Application-Level Input-Output Operations	•			
8C. Input Restrictions	•			
8D. Operating System Independence	•			
8E. Configuration Control		•		
9. PARALLEL PROCESSING				
9A. Parallel Control Structures	•			
9B. Parallel-Path Implementation	•			
9C. Mutual Exclusion and Synchronization	•			
9D. Scheduling	•			
9E. Real-Time Clock			•	
9F. Simulated-Time Clock	•			
9G. Catastrophic Failures	•			
10. EXCEPTION HANDLING				
10A. Exception-Handling Facility	•			
10B. Error Situations	•			
10C. Enabling Exceptions	•			
10D. Processing Exceptions			•	
10E. Order of Exceptions	•			
10F. Assertions	•			
10G. Suppressing Exceptions	•			
11. SPECIFICATIONS OF OBJECT REPRESENTATIONS				
11A. Data Representations	•			
11B. Multiple Representations	•			
11C. Translation Time Constants and Functions	•			
11D. Configuration-Dependent Specifications	•			
11E. Code Insertions	•			
11F. Optimization Specifications	•			
12. LIBRARY, SEPARATE COMPILATION, ETC.				
12A. Library Entries	•			
12B. Seperately Compiled Segments	•			
12C. Restrictions on Seperate Compilation	•			
12D. Generic Definitions	•			
13. SUPPORT FOR THE LANGUAGE				
13A. Defining Documents	•			
13B. Standards	•			
13C. Subset and Implementations Superset	•			
13D. Translator Diagnostics	•			
13E. Translator Characteristics	•			
13F. Translation and Execution Restrictions	•			
13G. Software Tools and Application Packages	•			

II GENERAL DESIGN CRITERIA

A. *Generality*

The language provides generality only to the extent necessary to satisfy the needs of embedded computer applications. In a few cases, it has been found that some of the more detailed requirements of Ironman cannot be satisfied without employing a more general facility than has previously been used successfully in a high-level language for embedded applications. In all such cases, we have emphasized in our design the ease with which programmers can avoid the use of novel features and any consequent, unforeseen loss of efficiency or reliability.

The language contains facilities for real-time control, exception handling, input-output to nonstandard peripheral devices, parallel processing, numeric computation, and file processing. In some cases, the facilities are machine-dependent, and their detailed specification must be given in an implementation manual. The language contains precisely those facilities required by Ironman.

B. *Reliability*

The language satisfies the requirement of reliability, in full accord with Ironman.

In some cases, the more detailed requirements specify lower-level features whose unreliable use cannot be detected automatically. We have classified such features in a separate section of the language description, to assure that they are not invoked without due consideration. Many programs can avoid such features altogether without significant loss of generality, efficiency, or convenience. When an application requires low-level features, we have assured that their use can be confined to isolated encapsulations, whose reliability follows from small size and the care expended in their construction. Furthermore, the correct use of these encapsulations by the rest of the program is enforced by the same stringent checks that are applied to the language as a whole.

C. *Maintainability*

The language subscribes fully to the objective of maintainability. Its rules are designed to assure that the most likely changes to program text can be made locally within a single context, and the consistency of such changes with the rest of the program can be fully checked by a translator. The effects of local changes are visible and are confined as far as possible to the local context in which they are made. Such changes can often be made with minimum recompilation of unchanged text.

II

D. *Efficiency*

The language is designed to enforce production of efficient and compact object programs and compact data representations. Nearly all its constructs, including procedure call and exit, parameter passing, encapsulation, process activation, mutual exclusion and synchronization, and response to interrupts have consistently efficient implementations.

In a few cases, the Ironman requires features that could be unexpectedly expensive, and their use in embedded programs might in some cases be unacceptable. We have clearly identified such features in our language design. We have tried to ensure that embedded programs can be written in a manner that avoids such features entirely without loss of generality, reliability, maintainability, or convenience. Embedded programs can be executed on a "naked" machine with small execution-time support packages. Furthermore, they can be translated simply and efficiently into high-quality code, without requiring any optimization beyond that already conventional, provided that the expensive features are omitted.

E. *Simplicity*

The language has been designed on the principle that simplicity is a necessary condition for meeting all the other design criteria. Simplicity is essential to ensure the success of a common, high-order, embedded-systems language in DoD, to achieve early implementation of such a language, and to gain wide acceptance for the merits of the language in embedded-system and language-support programming projects.

We have identified many cases in which almost all the requirements of Ironman can be met by uniform application of the same simple feature. However, it must be admitted that our language still contains several features that are not present in any previous practical language. Such features are included only to respond to the most explicit requirement in Ironman. We will welcome guidance and assistance, during Phase 2 of this project, to assure that such features meet real needs and will hope to simplify them further wherever possible.

F. *Implementability*

The language has been firmly based on the languages Pascal and Modula, which are well understood and can be implemented by simple, efficient techniques.

The semantics of each feature (except low-level and machine-dependent features) have been fully specified and understood, and interaction effects with other features have been predicted. In a few cases, we regarded these interaction effects as unfortunate, and we have simplified our design to avoid them. Such simplifications do not impact the usefulness of our language for embedded applications.

However, we must admit that our language still combines a number of features that have never been combined successfully in any previous language. We will organize our prototype implementation in Phase 2 to assure that any latent problems come to light early enough to be effectively addressed.

G. *Machine Independence*

The language has achieved a high degree of machine independence. Those features of the language whose detailed specification is necessarily dependent on implementation characteristics have been clearly separated from the other features, and many programmers can avoid entirely the use of such features. Where machine-dependent features are required by an embedded application, they will be explicitly enclosed in an isolated encapsulation. This encapsulation will

provide a machine-independent interface; when the program is moved to another machine, only this encapsulation, and not the programs referring to it, will need to be changed.

H. *Formal Definition*

There are essentially three methods of formal definition: axiomatic, operational, and denotational. We view the axiomatic approach as best suited to guiding language design. Moreover, because our language is based upon Pascal, it has a large subset for which an axiomatic definition is readily applicable. The subset is described in detail in the analysis of feasibility. However, the axiomatic approach is not, at least in the present state of that art, capable of providing formal semantics for the whole language. We have also considered the application of the other two approaches to formal definition, both to portions of the language and to the language as a whole. It is our considered opinion, amply reinforced by our consultants, that while an operational definition for the whole language is feasible in principle, the research effort involved in such an undertaking would not be justified by its possible benefits to the language design. The same conclusion is applicable, a fortiori, to a denotational definition of the language.

The foregoing considerations suggest that one might apply distinct definitional approaches to different parts of the language. Indeed, this possibility is suggested in Ironman requirement 1H. Unfortunately, there is no known method whereby one can achieve a consistent overall definition of a language (certainly not for one as complex as this one) by combining several such distinct techniques for different portions of the language. Thus, such a piece-meal approach would in the end inevitably force one to employ one of the several techniques on the whole language.

Our approach toward compliance with requirement 1H has, therefore, been to propose that a formal axiomatic definition be provided for as much of the language as can be handled by that technique at present. The vigor of current research activities in this area indicates that the subset to be so accommodated will, in all likelihood, be substantially extended as the work progresses. The analysis of feasibility contains a discussion of language features that could conceivably be axiomatized, although further research would be required for a definitive conclusion. The attempt to extend the axiomatic approach to deal with such additional features of our language has guided the detailed design of those features, and will assist in achieving verifiability of larger classes of programs. It is important to recognize that it is precisely those features in programming languages (admittedly important for flexibility and efficiency) that are impossible to axiomatize cleanly, that also make it impossible to reason convincingly about the actual effects of a program.

Considerations of formal definability have influenced the selection and design of the basic statements and control structures of the language, and the identification and classification of its low-level and machine-dependent features. We have not been asked to provide documentary evidence of this, but we hope that an analysis of the design will reveal our concern with this issue.

I. *Reliable, Provable Programs*

The great majority of the most useful features of our language are amenable to known proof techniques. Those features for which there is no known proof technique have been included only in response to an explicit requirement of Ironman. We believe that in many cases, the relevant proof technique will be discovered by future research, while in other cases the unprovable feature can be encapsulated and segregated within the larger program.

J. *Ease of Teaching and Learning*

This is, indeed, an extremely important criterion. We believe that our language meets it in high degree. We have developed, and present in our Tutorial and Programming Examples, a graded series of example programs. We believe they show the good prospects of achieving good skills in the use of the language within DoD, as well as the feasibility of an introductory primer and other instructional materials of high quality. We believe that those features that are more difficult to teach and to learn will be just those features most programmers will neither need nor wish to use.

K. *Practical, Implementable*

We believe that the most important and useful part of our language is easy and inexpensive to implement and that it can therefore become widely available within a short period.

If any inessential feature of the language should in Phase 2 present any unexpected problems for inexpensive implementation, we will be pleased to discuss resolutions of such conflicts.

L. *Addresses Difficult Issues*

Our preliminary design makes every attempt to avoid rather than to solve difficult problems in language design. Instead, we explore the underlying issues, and propose solutions that meet the objectives but avoid most of the difficulties. All the issues listed in this requirement have been squarely faced. Our recommendations are clearly described, and justified in the Design Discussion.

M. *Feasibility, Time Scales, and Confidence of Success*

We would like our language design to be evaluated in accordance with this additional criterion.

Past experience shows that the major risk attending a language design project is the extension of required time and the consequent escalation of cost. Indeed, there are several languages for which the progress from design requirements to eventual standardization has extended up to ten years. Such a delay might be tolerable if the eventual product at last meets its design requirements to the full. Unfortunately, past experience shows that this cannot be guaranteed.

Our language has been designed to eliminate this risk from the start. It has been constructed as a merger between two known languages, Modula and Pascal. Both are implementable by known efficient techniques; they are known to be adequate for systems programming and embedded applications. Both have proved to meet the general design criteria to a high degree. This merger language has then been extended by including additional facilities required by Ironman. Our implementation project, Phase 2, is similarly structured. First, using known techniques, we shall construct a simple efficient implementation of the merger language, producing code for implementation of embedded applications on a machine of the size and complexity of the DEC PDP 11. We shall then implement each additional feature of our language individually, checking that its inclusion does not impact the quality of the language or its implementation. In the event of any unforeseen problem or interaction effect, we will be able to present the technical alternatives clearly.

We are confident that our project plan minimizes the risk of perturbation, and that at the end of Phase 2 we shall deliver an implementable language of high quality, that can within a short period be widely implemented and safely accepted by programming projects in the armed

services, and can yield calculable and incalculable increases in the programming capabilities of the Department of Defense.

III SPECIFIC DESIGN CRITERIA

A. *General Syntax*

2A. Character Set

The characters of our language are the specified 64-character subset of ASCII. (This document is typeset using the character ≠ to denote the inequality operation; ≠ stands for the character # in the 64-character subset.)

2B. Grammar

Our grammar is quite short, comparable in length to Pascal's grammar. Care has been taken to avoid verbosity in programming—e.g., it is rarely necessary to use the Algol 60 Begin-End construction since a statement sequence separated by semicolons is syntactically allowed wherever a single statement is allowed. Analogous facilities are invoked with analogous syntax—e.g., the declaration of nomenclature whether as block locals, procedure, process, or function formal parameters, or record components. For facilities substantially identical to widely used Pascal facilities, we have either retained the Pascal notation or used a notation consistent with the style of our language.

2C. Syntactic Extensions

There is no facility of the kind prohibited.

2D. Other Syntactic Issues

Our compliance with the requirements for multiple occurrences of language-designed symbols, unmatched parentheses, and bracketed key word forms is manifest in our grammar. The requirement that source program line boundaries be equivalent to spaces must be interpreted in the light of the prohibition against comments and strings that extend over source line boundaries, but we provide a fully adequate resolution.

2E. Mnemonic Identifiers

An identifier may be spelled with any combination of letters, digits, and underscore characters ("_") beginning with a letter and not identical to a reserved word of the language. (The reserved words of the language are listed in Appendix A of our Language Report.) The underscore character is the intra-identifier break character. As required, we forbid the abbreviation

of identifiers or reserved words. These rules are presented in Section III of our Language Report.

2F. Reserved Words

We comply with this requirement. (A number of identifiers—INTEGER, BOOLEAN, etc.—have predefined meanings that are fundamental in the language semantics but that are syntactically merely identifiers.) The applicable rules of the language are described in Section III and Appendices A and B of the Language Report.

2G. Numeric Literals

We provide floating-point literals, fixed-point literals, and integer literals. The definitions of these categories of numeric literals—FloatingLiteral*, FixedLiteral*, and IntegerLiteral*—are presented in Section IV.B of the Language Report. Numeric literals are required to have the same values in programs as in data.

2H. String Literals

We provide string literals as quoted sequences of characters of a user-specified alphabet. These string literals are one-dimensional arrays of characters, of appropriate length. String literals must be terminated before the end of the line, but the implicit catenation of string literals on successive lines effectively permits arbitrary lengths. The ALPHABET notation allows the explicit naming of formatting, control, and other nonprinting characters, and thus allows their inclusion in string literals.

2I. Comments

Both required forms of comments are provided and described in Section III of the Language Report.

B. *Types*

3A. Strong Typing

The language is strongly typed. Variables, arrays, records, enumerations, and sets are strongly typed, as are files and pointers.

We make two explicit relaxations in this strong typing for procedure formal parameters, relaxations that we believe to be necessary to meet the requirements:

- We permit, as formal parameters, arrays for which the range of the indices is not determined but is derived from the range of the indices of the actual parameter (as required by Ironman 7H).

- We permit, as formal parameters, the type SPACE, which matches any type. This is needed to allow service routines, such as device drivers or space managers, to provide their service for different types of data. The SPACE formal parameter type, described in Language Report Section XIV.X, is implementation-dependent.

By confining these "loopholes" to the area of formal parameter to actual parameter matching, we ensure that any one program unit uses only a single representation for an object.

3B. Implicit Type Conversions

We comply with this requirement. The two relaxations described above, under 3A, are explicit and do not cause any change in the representation of the data object. Implicit representation conversion is provided between different representations of the same type. This may result in unexpected inefficiencies, but the language has been designed so that such inefficient situations cannot arise by oversight but only through the explicit declaration of a named type and of two or more representations for that named type.

3C. Type Definitions

Section VII.E of the Language Report describes our facility for defining new data types in programs and associating identifiers with types. As required, the definitions can be processed entirely during translation and their scope is the lexical scope of the block in which they are declared. No special restrictions are imposed on defined types, except that certain built-in types are defined to be pervasive throughout the program and do not need to be explicitly imported into modules. Defined types cannot be pervasive and must be imported into every module within which they are referenced.

3-1A. Numeric Values

We comply with this requirement. Section VI.A of the Language Report describes the numeric type facilities and Sections VIII.A, VIII.C, and IX.A describe the semantics of the numeric operations and numeric assignments.

3-1B. Numeric Operations

The required operations, complying with the requirement, are described in Sections VIII.A, VIII.C, and XI.A of the Language Report.

3-1C. Numeric Variables

Every numeric variable declaration requires a representation that specifies a range (Section VII.F of the Language Report), thus complying with this requirement. If the range determined at allocation time exceeds the range of the available physical representation, an exception (suppressible) will be generated.

3-1D. Floating-Point Precision

The relevant rules of our language, stated in Sections VI.A, VIII.A, and VIII.C of the Language Report, satisfy these requirements.

3-1E. Floating-Point Implementation

The relevant rules of our language, stated in Sections VIII.A, VIII.C, and IX.A of the Language Report, satisfy these requirements.

3-1F. Integer and Fixed-Point Numbers

The relevant rules of our language, stated in Sections VIII.A, VIII.C, and IX.A of the Language Report, satisfy these requirements. Any implicit representation conversion involving truncation of the less significant part of a value is an error detected during translation. Any implicit representation conversion involving truncation of the more significant part of a value may cause a range exception to be generated at run time.

3-1G. Fixed-Point Scale

The declaration of a fixed-point variable requires a representation that specifies a scale. An implementation may restrict the values of the scale to those it can implement efficiently. Other scales can be provided to those who need them by sets of library procedures (including overloaded infix operators), thus the space and performance penalties of inappropriate choices of scale will not be surprising.

3-1H. Integer and Fixed-Point Operations

Section VIII.C of the Language Report describes the semantics of our built-in operations on integers and fixed-point numbers, including division with remainder and scale conversion. We exclude implicit scale conversions that change the abstract value represented.

3-2A. Enumeration Type Definitions

We satisfy this requirement with our scalar types, described in Section VI.A of the Language Report. (Note that we use the phrase *scalar types* to designate what are called *enumeration types* in this requirement.) Our scalar types—ordered enumerations, unordered enumerations, and alphabets—are definable by enumeration of their elements, provide equality and inequality operations between their elements, and may be restricted to any contiguous subsequence. The ordered and unordered enumerations contain values that are identifiers. The alphabets contain values that are character literals, and are distinct from ordered enumerations so that printing nonalphabetic characters can be used as character literals but not as identifiers.

3-2B. Ordered Enumeration Types

The phrase "ordered enumeration types," as used in this requirement, is satisfied by our ordered enumeration types and ALPHABET types. Our ordered enumerations must contain the reserved words ORDERED or ALPHABET in their definitions, satisfying this requirement (Section VI.A of the Language Report).

3-2C. Boolean Type

Section VI.A of the Language Report states that Boolean is an unordered enumeration type, as required. Section VIII.A describes the built-in operations for this type, in compliance with the requirement.

3-2D. Character Types

Section VI.A of the Language Report describes our ALPHABET types, which satisfy this requirement.

3-3A. Composite-Type Definitions

Section VI.B of the Language Report describes our array and record types, which satisfy this requirement.

3-3B. Component Specifications

Section VI.B of the Language Report describes the specifications of record and array components, which satisfy this requirement.

3-3C. Operations on Composite Types

We satisfy this requirement except where it is modified by Requirement 7I—"Restrictions to Prevent Aliasing." The assignment and value-accessing operations for components of composite types are described in Section VIII.B of the Language Report. The constructor operation is described in Section VI.B. The aliasing restrictions on assignable components are described in Section X.A of the Language Report; otherwise, an assignable component is indeed permitted wherever a simple variable of the same type is permitted.

3-3D. Array Specifications

The array data type of the language is presented in Section VI.B of the Language Report. There is also a formal array type, permitted only for formal parameters, that is described in Section X.A of the Language Report. Both versions require that the number of dimensions be specified at translation time. The indices are specified as a representation (Section VI.B of the Language Report) that provides the required options—ranges determinable at the time of array allocation and restricted to contiguous sequences of the integers or of a scalar type.

3-3E. Operations on Subarrays

Section XI of the Language Report describes the MOVE procedure, which provides the required operations on subarrays (and only those operations). Where an array is moved onto itself, and where representation conversion is implicit, use of the MOVE procedure may involve hidden inefficiency.

3-3F. Nonassignable Record Components

We satisfy the two requirements with two different facilities. Nonassignable record components are obtained by using the CONST option in the declaration of the nonassignable component (Section VI.B of the Language Report). A component whose value is the dynamic value of an expression is obtained by enclosing the representation definition of the record in a module. This module exports that representation and a function, applicable only to that representation, that computes the desired expression. An example of this facility is given in Section X.B of the Design Discussion. It is even possible to obtain variant records whose variant case selection (tag) field is the dynamic value of an expression, though to do so the type definition of the record must be exported opaquely and all assignments to, and access to, components of the record must be made through exported functions and procedures. A transparent record, whose variant-case-selection field was the dynamic value of an expression, would lead to a violation of Ironman 3-3H and immediate disaster.

3-3G. Variant Types

Section VI.B of the Language Report describes our variant records, which satisfy this requirement.

3-3H. Tag Fields

Section VI.B of the Language Report describes the rules for the variant case selection (tag) field of a variant record, satisfying this requirement.

3-3I. Definitions of Dynamic Types

Pointer types are defined in Section VI.E of the Language Report. Such pointer types are dynamic types in that objects of the type can be dynamically allocated and deleted. The

referenced objects may be, or may contain, objects of their own type and may be modified during execution. Objects of the pointer type are distinct from other composite objects and from objects of their base type—i.e., an object of type pointer to integer is not the same type as an object of type integer. The user may use encapsulations to provide explicit allocation of storage for pointed-to objects.

3-3J. Constructor Operations

The primitive for constructing entities of a dynamic type creates a distinct element of the type as required. We also provide an unsafe explicit FREE_POINTER procedure (Section XIV.VI of the Language Report) that makes it possible to release storage even though there is a path to it. However, it is possible to construct encapsulations out of the primitives that satisfy the safety requirement as demonstrated in Section VI of the Design Discussion. (The unsafe primitives are desirable for providing the flexibility of programming needed for real time or other applications in which resource utilization is critical.)

3-4A. Bit Strings

This implementation requirement does not affect the language design; however, we agree that bit strings can readily be the default representation of Boolean vectors (if the hardware is suitable).

3-4B. Bit-String Operations

Equality and inequality are automatically defined for Boolean vectors as for other data types (see Section VI.A of the Language Report). Other operations on Boolean vectors may easily be defined by users or provided in a standard library.

3-5A. Encapsulated Definitions

This requirement is satisfied by our modules, described in Section X.B of the Language Report.

3-5B. Effect of Encapsulation

Modules comply with this rule, since internal declarations are hidden unless explicitly "exported." Similarly, the structure of exported types is hidden unless the exportation is explicitly "transparent."

3-5C. Own Variables

The nonexported local variables of a module satisfy these requirements. As required, they may be initialized at "the time of their apparent allocation"—that is, on entry to the smallest enclosing block.

3-5D. Operations Between Types

The module and exportation facilities satisfy this requirement of Ironman, as is explained in Section X.B of the Design Discussion.

C. *Expressions*

4A. Form of Expressions

The syntax of expressions, in Section VIII of the Language Report, meets this requirement.

4B. Type of Expressions

We meet this requirement. The details are in several categories: Section VI of the Language Report on types, Section VII on declarations, Section VIII on expressions, and Section X on functions and procedures, modules, and processes. Throughout the language, the type of every expression is completely and uniquely determined, with literals requiring a type qualifier, where necessary, to ensure this strong typing. It is possible, but never necessary, to create identity functions imposing no run-time overhead; these provide a lexical confirmation of the type of an expression.

4C. Side Effects

We provide pure functions (FUNCTIONs) and impure functions (PROCEDUREs). The former satisfy a more severe rule than 4C—no nonlocal assignment is permitted. The latter are unrestricted. We believe the intermediate requirement is not sufficient to ensure that reordering of expressions will not change their values. The rule, necessary to ensure that functions containing nonlocal assignment are still pure functions, is complex and involves partial program verification. We believe that the simplest, most easily understood, and most readily enforced rule will add most to program reliability.

4D. Allowed Usage

It is clear from the syntax, Appendix B of the Language Report, that this requirement is satisfied—manifest constants and scope entry constants are both expressions.

4E. Constant Valued Expressions

Similarly, it is clear that this requirement is met by the syntax. The language requires that some expressions (manifest constants, Section VI.D of of the Language Report) be evaluated during translation, and restricts such expressions to assure the feasibility of such evaluation. The language does not require translation evaluation of any other expression; rather, this is a requirement to be placed on production translators.

4F. Operator Precedence Levels

Section VIII.A of the Language Report describes the precedence levels of built-in operators—relational operators, multiplication operators, addition operators, and prefix operators. The user cannot alter their binding strengths or declare new operators, but he can give meaning to a number of (otherwise undefined) "free" operators at each level.

4G. Effect of Parentheses

Section VIII.A of the Language Report, as required, states that explicit parentheses dictate association if they are present. A mathematically sound treatment of operator association involves not just individual operators but all of the operators defined at a precedence level, including user-defined operators (described in Section VIII.A of the Design Discussion). The rules involved are sufficiently complex that we despair of even explaining their intent to typical users, let alone enforcing their observance. Instead we have established a simplified rule to

accommodate our user's conception of the operators +, *, AND, and OR as self associative, with all other operators being nonassociative. In expressions involving associating operators, the parse priority is from left to right, as in the base language Pascal, though semantically equivalent implementations are of course permitted. We have been unable to enforce the requirement that expressions involving different operand types should be parenthesized, because of difficulties involving requirement 7A for the extension of infix operators to new types and requirements 4A, 2B, and 2C for a simple, free-form, type-independent syntax without syntax extension. Such expressions are of course permitted to contain appropriate parentheses, and we encourage this practice.

D. *Constants, Variables, and Declarations*

5A. Declarations of Constants

The rules for CONST declarations in Section VII.A of the Language Report satisfy these requirements.

5B. Declarations of Variables

The rules for VAR declarations in Section VII.B of the Language Report satisfy these requirements. The additional characteristics of variables local to modules are described in Section X.B of the Language Report; they also comply with the requirement. Further such variables are deallocated on exit from their scope and no dangling references to them can remain after their deallocation.

5C. Scope of Declarations

The scope rules for declarations are presented in Section VII of the Language Report, except for declarations local to modules, which are described in Section X.B of the Language Report. These rules satisfy this requirement; local scopes are obtained with the BEGIN-END statement, and may be lexically embedded in an enclosing block. Scopes requiring restricted access to nonlocal variables are declared as MODULEs (Section X.B of the Language Report).

5D. Restrictions on Values

No values, of the kind enumerated in this requirement, may be used in any of the prohibited ways.

5E. Initial Values

Section VII.B of the Language Report describes the declaration and initialization of variables, satisfying this requirement. The symbol ":=" is used, optionally, to initialize variables; the symbol "=" is required to give the value of a constant.

5F. Operations on Variables

The implicit value access operation for variables is described in Section VIII.B of the Language Report. The assignment operation is described in Section IX.A of the Language Report. Assignment, equality, and inequality operations are automatically defined for every type, but this automatic definition can be overridden when necessary, as, for instance, for signals, files, and semaphores.

E. *Control Structures*

6A. Basic Control Facility

We satisfy these requirements. The (nonparallel) control mechanisms of our language are described in Section IX.B of the Language Report. They are "single capability" mechanisms and, since each is a statement, are nestable as required. There is no control definition facility. The BEGIN-END structure may be nested within any of the control statements, and the control statements may be nested within BEGIN-END statements, to provide local scopes as needed. The control statements are single-entry, single-exit constructs, with the exception of the GOTO statement and the GOTO-like EXIT statement.

6B. Sequential Control

Statement sequences do require explicit semicolons between statements, and an optional semicolon is allowed at the end of a statement sequence for those who prefer to regard the semicolon as a statement terminator.

6C. Conditional Control

The IF statement allows a path to be selected according to the value of a BOOLEAN expression. The SELECT statement allows one path to be selected as a computed choice among alternatives. In both cases, if none of the discriminating conditions apply, the semantics are that the statement has only the effect of the needed evaluation of the conditions. In both cases, there is an ELSE option to specify a single control path to be used when no other path is selected. Where the translator can determine the path that will be selected, the language permits but does not require compilation of only that path.

6D. Short Circuit Evaluation

The semantics of the standard infix operators AND and OR for BOOLEAN expressions, as presented in Section VIII.A of the Language Report, are "short circuit"—that is, the second argument is evaluated only when it might affect the result of the operation.

6E. Iterative Control

We provide three basic iterative constructs: a WHILE construct, which terminates when a continuation condition (including the disjunction of a set of conditions) is not met; a FOR construct, which iterates over subranges of integers and scalar types (with a loop counter accessible as a constant within the loop body); and a more general LOOP construct, which iterates until the EXIT statement is executed within its (lexical) body.

6F. Control Variables

We comply with the substance, but not the letter, of this requirement. Apart from the FOR statement—that does include the declaration of a local variable—we do not provide a special mechanism for associating variables with control statements that are "defined ... nowhere outside the control statement." (We can embed any control statement in a BEGIN...END scope in which local nomenclature is introduced; however, this nomenclature exists throughout the BEGIN...END scope and not just within the control statement.)

6G. Explicit Control Transfer

We comply with this requirement. We allow an explicit GOTO statement, which may transfer control to any labeled statement. We have attempted to control the interaction of this facility with the conditional and iterative control structure by allowing labels only for the statements of the statement sequence of a block. This serves to preclude transfers into iteration structures and into, or between alternative paths of, conditional structures, facilitating their use as the basis of the conditional compilation.

F. *Functions and Procedures*

7A. Function and Procedure Definitions

The definition of functions and procedures is described in Section X.A of the Language Report, including extension and replacement of prior definitions.

7B. Recursion and Nesting

We provide recursive function and procedure calls and nested definitions. Our experience indicates that available methods of implementing variable access are sufficiently efficient that no additional restriction to the Pascal base language,on the interaction between nested definitions and calls, is warranted. The language, as proposed, is such that efficient implementation is possible by a fully static allocation of all variables and temporaries, except dynamic types and variable length arrays. Alternatively, efficient implementation is possible using a stack allocation structure and lexicographic chaining, as exemplified by current Pascal implementations. Such a method is in any case required for processes.

7C. Scope Rules

We employ the static scoping rules of Algol 60, as augmented by our *modules*, to satisfy this requirement. The scope rules are described in Section VII of the Language Report, the rules associated with modules in Section X of the Language Report.

7D. Function Declaration

We do require the specification of result types for functions and value-returning procedures (i.e., impure functions). The type of the result is known during translation, but representational qualifiers such as ranges and also array dimension ranges can be determined on entry to the scope within which the function is declared. The association of infix operators is predefined rather than user-specifiable (see discussion of requirement 4G and Section VIII.A of the Design Discussion).

7E. Restrictions on Functions

As with Ironman 4C, discussed above in Section C, our pure functions satisfy a greater restriction, our impure functions are unrestricted, and we do not believe that the intermediate restriction stated in 7E is adequate to achieve the desired effect. (See the Design Discussion for an analysis of this issue.)

7F. Formal Parameter Classes

Section X.A of the Language Report describes these required classes of formal parameters.

7G. Parameter Specifications

Section X.A of the Language Report states the requirements on declaration of formal parameters, which satisfy this requirement.

7H. Formal Array Parameters

We provide a formal array type (see Section X.A of the Language Report) that satisfies this requirement. Section XI of the Language Report—Standard Functions and Procedures—describes the built-in functions whereby the subscript ranges of a formal array are determined during execution.

7I. Restrictions to Prevent Aliasing

Section X.A of the Language Report describes the restrictions we place on aliasing, to meet this requirement.

G. *Input-Output Facilities*

8A. Low-Level Input-Output Operations

Our general approach to input and output is to specify the highest level (most abstract) and the lowest level (most concrete). There will also be one or more intermediate levels constructed as modules of the language. Both the lowest level and the highest level, as here proposed, can be designed with confidence in the completeness and appropriateness of facilities of the language, because of the simplicity and generality of the low-level features, and of the simplicity and abstraction of the high-level features. Intermediate-level input and output facilities are difficult to design so that they are simple, general, machine-independent, and satisfying to a wide range of users. We will be happy to discuss the design of appropriate intermediate-level input and output facilities; in no case should such a design be finalized before a production implementation is completed.

To assure the generality of the low-level input and output facilities and their applicability to all embedded system applications and machines, we satisfied Ironman requirement 8A by means of machine code inserts and device signals, described in Section XIV.XII of the Language Report. It will be possible, using this facility for any particular device or machine, to provide by encapsulation the intermediate and higher-level facilities, or the specialized facilities needed by particular embedded systems.

8B. Application-Level Input-Output Operations

We provide the facilities required, slightly generalized from those of the Pascal base language, as described in Section XII of the Language Report. In the absence of specific directions as to the kind and extent of the random access facilities required, we have not designed that facility but would have no difficulty in doing so. We would continue to view the Pascal "file" as sequential, and would introduce an analogous "random access file" with appropriate operations.

8C. Input Restrictions

The files used for application-level input-output must be declared with type "FILE(Representation)", where Representation specifies the data representation of the elements of the file.

8D. Operating System Independence

The language has been designed so that all facilities are ultimately provided either by standard procedures or by code in the "nucleus." This nucleus will have to be present to support program execution but will, we believe, require only a few hundred to a thousand (36-bit) instructions, depending on the machine or operating system in question and the range of language facilities used in the particular embedded-system program. Our language and its run-time support can readily be hosted by conventional operating systems. The presence of an operating system typically changes the virtual machine perceived by a user. Thus a program written using machine-dependent facilities will typically have to be modified to be moved from a bare machine environment to an operating system of that same machine. All such implementation-dependencies should be encapsulated in library modules so that machine- and operating-system-dependence is localized.

8E. Configuration Control

The standard functions to interrogate the status of the built-in and predefined types and operations are given in Section XIV.XI of the Language Report. A similar set of procedures and functions may be provided by an implementation to interrogate and control the allocation of physical resources. Such procedures and functions are strictly implementation-dependent, and we have not attempted a definition of them here. The association of implementation-dependent variables, signals, etc. with machine locations through the AT qualifier depends on scope entry constants. Thus reassociation is possible through redeclaration, but not, of course, during use. Configuration control can also be a feature of the medium-level input and output facilities and can be provided through standard library routines constructed out of lower-level facilities, or can be adapted to the particular needs of a specialized embedded application.

H. *Parallel Processing*

9A. Parallel Control Structures

The parallel-processing facilities of the language are a slight generalization of those of the successful embedded-system language Modula, using monitors (see [7] and [2]) for synchronization. The process is the unit of definition of a parallel path and the process activation is the way of starting a parallel path. The form of definition of a process (see Section X.C of the Language Report) states that the maximum number of paths to execute concurrently must be determinable at scope entry time, as required, and that all local concurrency must cease before exit from a scope. This permits static allocation of storage, but does not (and should not) enforce it.

9B. Parallel-Path Implementation

Our parallel-processing facility is a slight extension of that utilized in Modula; it has been designed to retain the efficiency of execution time and space, and the convenience and flexibility of use displayed by the existing implementations of Modula. The language semantics are independent of the number of processors used during program execution.

9C. Mutual Exclusion and Synchronization

The language provides MONITORs, equivalent to Modula INTERFACE MODULEs and to the monitors of Hoare and Brinch Hansen; within these modules, the "signal" is available for synchronization. Second, we include binary semaphores—a stylistically lower-level facility—which permit the construction of more selective and even overlapping exclusions; these can be more efficient but also more prone to error and deadlock.

9D. Scheduling

Much of this requirement concerns the implementation rather than the design of the language, and will require careful reconciliation with requirement 9B.

We provide a priority for each process template in its declaration. This priority may be changed by the running process. The attribute has no formal semantic significance; it is implementation-dependent advice. Similarly, a first-in-first-out queueing discipline is permitted but not mandated. (It is mandated for reactivation after waiting for a signal.)

9E. Real-Time Clock

The language provides a procedure DELAY and a function CLOCK_TIME that provide a delay in terms of an implementation-dependent clock interval and a (cyclic) count of clock ticks. More elegant facilities involving acyclic time, time of day, and clock setting and correction can be provided by encapsulation. A clock giving the cumulative processing time on each path is not feasible using most existing hardware; this facility will be provided as an implementation-dependent standard procedure where supported by hardware.

9F. Simulated-Time Clock

The procedure DELAY and the function CLOCK_TIME, when used within a SIMULATED_TIME BEGIN-END statement, refer to the simulated-time clock of that statement. (See Section X.E of the Language Report.)

9G. Catastrophic Failures

The procedure TERMINATE can be used to provoke a termination exception within another process. The immediacy of the exception is an implementation consideration and is not here defined. The effect of the termination exception within that process will depend on the exception handlers defined within it, and cannot be guaranteed to cause the process to terminate.

I. *Exception Handling*

10A. Exception-Handling Facility

The exception-handling facility has been designed to be simple and free of interaction with other aspects of the language, while still meeting the requirement that recovery from unplanned situations be possible. The facility provided distinguishes between TRAPs, detected by the hardware, the nucleus, or the compiled code, and EXCEPTIONS, detected and propagated by program. The compiled code necessary to detect errors and to cause the TRAP will add to program size and execution time unless suppressed. Otherwise the exception-handling facilities need add nothing to execution time until an exception occurs (Section X.F of the Design Discussion).

10B. Error Situations

The list of standard exceptions to be generated by the standard set of trap handlers is given in Appendix C of the Language Report.

10C. Enabling Exceptions

The PROPAGATE_EXCEPTION statement is described in Section IX.A of the Language Report. We believe that it complies completely with this requirement.

10D. Processing Exceptions

The exception handlers are described in Section X.F of the Language Report. More complex exception-handling facilities have been described in the literature (and even implemented), but we were unable to reconcile them with the other requirements of Ironman (Section X of the Design Discussion). We believe that the proposed facility substantially satisfies the requirements expressed here. The "last enabled exception" is not a meaningful concept in a system that can recover from a second exception while handling the first, but a facility is provided whereby an exception handler can propagate the exception that invoked it. We rejected the concept of reinvoking exceptions at the join point of parallel paths because, in the embedded system context, many processes are designed never to terminate. (See Section X of the Design Discussion.)

10E. Order of Exceptions

Aside from the use of implementation-dependent facilities within a trap handler, only one exception will be detected in any expression. If more than one exceptional condition is detectable in an expression, declaration, or statement, and if the language definition does not determine which is to be detected, an implementation will detect any one of the exceptions and the other exceptions will not be detected.

10F. Assertions

Section IX.A of the Language Report describes the ASSERT statement, which complies fully with the language-related aspect of this requirement.

10G. Suppressing Exceptions

Section XIV.XIII of the Language Report describes the advisory compilation directives that can be used to suppress and permit exceptions by allowing the compiler to omit, or requiring it to include, the compiled code necessary to detect the corresponding errors.

J. *Specifications of Object Representations*

11A. Data Representations

Section VI of the Language Report describes the concept of a representation of a type. The qualifiers provided yield the required precision and flexibility while the concept of a representation separates most users from these considerations.

11B. Multiple Representations

The facilities provided by the Type and Representation facilities of the language comply precisely with this requirement.

11C. Translation Time Constants and Functions

Much of this requirement is more relevant to implementations of the language than to its design. The functions and constants required are described in Section XIV.XI of the Language

Report.

11D. Configuration-Dependent Specifications

The facilities for code insertion (Section XIV.XI of the Language Report) make it possible to use object machine features directly in programs. Frequently used machine features will be associated with specially compiled pseudo-routines as a convenience to programmers and to ensure efficient use. Other machine features can be accessed by use of the general machine code insert procedure.

In order to use machine-dependent features, it is necessary to import or to include (i.e., embed) the appropriate object machine encapsulation. Thus portions of the program that depend on the object machine or configuration are clearly designated as being machine-dependent. Within such regions, the parameters of the configuration, including designators for the machine model, hardware options, device configuration, operating system, and the like, can be used for conditional compilation and for module template instantiation, to tailor the program to the configuration.

11E. Code Insertions

We include machine (or other language) code through calls of standard procedures (described in Section XIV.XI of the Language Report). These procedures must be explicitly imported to satisfy Ironman requirement 11D. The sequence of expressions that are the arguments must be computable during translation and provide the instructions to be executed. The procedure name specifies the agent of execution—e.g., the machine or language processor.

11F. Optimization Specifications

Optimization specifications are of two forms, those applying to individual constructs, and those governing entire scopes of the program. The INLINE and PACKED attributes are of the first sort. The INLINE attribute of a routine indicates a preference for speed over space; the PACKED attribute of a structured type indicates the opposite preference.

Two compile-time evaluable procedures will be exported by the *compiler status module* (see Section XIV.XIII of the Language Report): the procedure FULL_OPTIMIZATION and the procedure COMPACTNESS_PREFERRED. Both act on a single BOOLEAN argument and are called in a user's program to guide the compiler's selection of code for entire scopes rather than single constructs. When full optimization is turned on (by calling the procedure on TRUE), the production of an efficient object program takes precedence over compilation efficiency. When full optimization is not turned on, the reverse is true. Similarly, when COMPACTNESS_PREFERRED is on, code compactness is given precedence over execution speed and otherwise the opposite is true.

In any implementation, there may be further optimization specifications to permit more detailed control of the code generation process. See Section XIV.XIII of the Language Report.

K. *Library, Separate Compilation, and Generic Definitions*

12A. Library Entries

The separate compilation facility is fully capable of supporting a library containing entries of the kinds mentioned in the requirement. The structure of the library is not constrained by the language. Library entries can be associated with particular applications, projects and users if the filing system of the environment is so structured.

12B. Separately Compiled Segments

The separate compilation facility of the language meets the requirement in every particular except one: satisfaction of the restriction on variable aliasing will not be fully checked at compile time. To do so for each segment would require a scan of the source texts of related segments and a transitive closure computation. This would make compilation intolerably expensive. Instead, the programming environment will provide a separate alias-checking facility. (See the Design Discussion, Section X.)

12C. Restrictions on Separate Compilation

Separate compilation in no way changes the meaning of a program. The design of the segmentation facility ensures that the consistency of shared definitions can be efficiently checked. Use of a segment synopsis to provide external definitions means that it is not necessary to have fully realized a module before compiling related segments.

12D. Generic Definitions

We have complied with this requirement by our TEMPLATE and INSTANCE facilities (Sections X.C and X.D of the Language Report). A module template is defined which, when instantiated, may contain and export a function or procedure. The parameters of the template may be the identifiers for variables, constants, types, representations, procedures, functions, or modules. (An expression or statement to be a parameter is enclosed within a function or procedure, thus ensuring the correct context for its evaluation without requiring any special scope rule.)

L. *Support for the Language*

These requirements apply more directly to production implementations of the language. We have taken care to make them satisfiable by such implementations, and we discuss some of the issues in our Analysis of Feasibility.

PART SEVEN

GLOSSARY

GLOSSARY

This is a glossary of some technical terms used frequently in the Language Report. The meanings of terms given here are intended to provide only an initial informal understanding of the term, sufficient to permit the reading of the text; the definitive meanings of the terms are given in the cited sections of the Language Report.

abstraction—the provision of a description for an entity that hides the internal details of the entity and retains only those properties that must be known to a user of the structure or object.

abstract type—a user-defined type (q.v.) whose internal structure is known only within its defining module.

activation—the operation of causing a parallel process to start running (Language Report, Section X.D).

actual parameters—the variables, named in the procedure call, upon which the procedure is to operate (Language Report, Section IX.A)

alias—one of the names for a variable that has two different names in some program scope (Language Report, Section X.A).

anonymous—a variable whose only name involves dereferencing of a pointer (Language Report, Section VI.C).

argument—parameter.

assertion—a BOOLEAN expression, optionally checkable at run time, and alleged by the programmer to evaluate to TRUE whenever encountered during program execution (Language Report, Section IX.A).

assignment—the operation of giving a new value to a variable.

associative—the property of a pair of operators, Op1 and Op2, such that, for all operands x, y, and z, ((x Op1 y) Op2 z) = (x Op1 (y Op2 z)) (Language Report, Section VIII.A).

asynchronous—happening at arbitrary times relative to other activities; usually applied to interactions between parallel processes.

attribute—a characteristic (such as the size or priority) of a declared entity.

block—a program unit consisting of local declarations, an exception-handler, and executable statements; the program unit that is the body of a procedure, function, module, process, or BEGIN...END statement (Language Report, Section X.B).

blocked—adjective applied to a process that cannot be run because it is waiting for the occurrence of some event—e.g., the release of a semaphore, the sending of a signal, or the availability of a monitor available (Language Report, Section X.E).

BNF—Backus-Naur form; a notation used to define the syntax of a language.

body—that part of a procedure, function, module, process, etc. that contains its local declarations, exception-handler, and executable statements (Language Report, Section X.B).

BOOLEAN—a type comprised of the values TRUE and FALSE (Language Report, Section VI.A).

call—the operation of invoking a procedure or function (Language Report, Section IX.A).

compiler directive—an instruction included in the program to advise the compiler how to compile the program, for instance INLINE (Language Report, Section XIV.XIII).

composition—the object formed by the physical, textual, or logical adjacency of a number of components.

conditional—a structured statement that consists of several subordinate statements and a means of choosing the subordinate statement, if any, to be executed.

conditional compilation—a compilation that depends on the values of constants or functions determinable at compile time—e.g., the values of configuration constants (Language Report, Section XIV.XIII).

configuration constant—a constant or function defined by an implementation to make available to the program some characteristic of the target machine of the compilation, such as word length or store size (Language Report, Section XIV.XIII).

context—the scope (q.v.) or environment—i.e., the names of the entities that can be referred to—at some point within a program (Language Report, Sections VII.A and X.B).

context-free syntax—a certain class of grammars, in which the grammar of this language is included.

critical region—a section of a program, involving parallel processes, in which exclusive access is required to some object (Language Report, Section X.E).

declaration—program text in which an object is defined that will be accessible within the scope (q.v.) of the declaration—e.g., simple declarations of constants and variables, or composite declarations of procedures and functions (Language Report, Sections VII and X).

declare—see declaration.

denotation—the description of a value.

designator—the name of a variable or constant involving zero or more of the operations subscripting, selection, and dereferencing.

dimension—the number of subscripts of an array.

disambiguation—the prefixing of a literal by a type or representation name where it would otherwise be unclear what type was intended (Language Report, Section VI.A); also, the discrimination between two identically named routines based on the number and type of their formal parameters (Language Report, Section X.A).

dynamic—allocated at run time.

dynamic variable—a variable accessed through a pointer rather than by name (Language Report, Section VI.C).

embedded system—a computer system, together with its software, which is part of some larger system—e.g., a real-time system controlling arbitrary hardware or communications equipment, or part of a weapons system.

encapsulation—the concept of enclosing the internal structure of objects, and of the procedures that manipulate them, within a boundary so as to permit explicit control of a user's access to this internal structure (Language Report, Section X.B).

entity—informal term used to refer to program components such as variables, procedures, etc.

enumeration—a type or representation whose possible values are defined by a list of permitted values, subject to certain other constraints (Language Report, Section VI.A).

error—the property of a program construction that is forbidden by the language definition; see also exception handling, trap.

exception handling—a facility provided in the language to allow recovery of a program from an unanticipated situation detected at run time (Language Report, Section X.F).

exclusion—that action of a monitor whereby two processes are prevented from being simultaneously active within a monitor (Language Report, Section X.E).

export—an entity accessible to the programs using a module or monitor (Language Report, Section X.B).

expression—a mathematical formula in the language that calculates a value (Language Report, Section VIII).

formal parameters—the names used within a routine or process body to denote the actual parameters (q.v.) with which the routine or process was invoked (Language Report, Section X.A).

free operator—infix operator that has no language-defined meaning and whose meaning therefore is defined by user definition (Language Report, Section VIII.A).

generic—see overloading.

global—a program entity that is accessible within a program scope but not declared by that scope.

identifier—name.

implementation—the combination of a language translator and the environment that will execute the object code produced by the translator.

implementation-dependent—a language feature whose meaning differs among implementations (Language Report, Section XIV).

implementation-independent—a language feature whose meaning is given by the language definition.

import—an entity that is not declared by a module but may be accessed within the module (Language Report, Section X.B).

index—subscript for an array.

infix—an operator on two arguments that is written between its arguments—e.g., + in (a*b)+(c-d) (Language Report, Section VIII).

initialization—the assignment of a value to a newly declared variable.

inner—of a scope, one that has is nested within an outer scope.

instance—the module resulting from the expansion of a module template (Language Report, Section X).

interrupt—an asynchronous signal from outside the computer to advise the computer program of some event outside the computer.

iteration—repetition, usually applied to program loops such as the FOR, WHILE,and LOOP statements (Language Report, Section XI.B).

label—a name that can be prefixed to a statement, permitting a GOTO statement to jump to that statement (Language Report, Section VII.C).

lexical—according to the structure of the program text.

lexical scope—the block within which a program name is declared.

lexical unit—a programmer defined or predefined name, a reserved word such as "BEGIN" or "DO", an operator such as "+" or ":=", or a punctuation mark such as "," or ";".

life-scope—the largest scope within which a name has a single meaning (Language Report, Section X.E).

linker—linkage editor or binder that combines separately compiled program segments into one executable program.

literal—a directly expressed constant value.

local variable—a variable declared within the current scope.

manifest constant—an expression satisfying rules ensuring that its value can be determined at compile time (Language Report, Section VI.D).

metacharacter—a symbol, used in the formal syntax of the language, that is not a symbol of the language but is used only to facilitate the definition of the grammar of the language (Language Report, Section III).

mismatch—failure of formal and actual parameters of a routine or process to agree in number or type (Language Report, Section X.A).

module—the primary mechanism for abstraction in the language; a collection of declarations, encapsulated so that their internal structure is not accessible outside the module (Language Report, Section X.B).

monitor—a module designed for use by several parallel processes, that allows only one process to execute within it at any time (Language Report, Section X.E).

monitor procedure—a procedure declared by a monitor.

nesting—the declaring of one scope within another scope.

nomenclature—names of program entities.

opaque—that property of an exported program entity whose internal structure is not known outside it defining module (Language Report, Section X.B).

outer—used of a scope within which the scope of interest has been declared.

overloading—the mechanism by which several procedures or functions may have the same name, provided that they operate on different numbers or types of parameters (Language Report, Section X.A).

packing—the allocation of storage to variables so as to minimize the use of storage, even at the cost of increase in execution time or program length (Language Report, Section XIV.VI).

parallel process—see *process.*

parameter—see *formal parameter* and *actual parameter.*

parameter binding—the process of associating formal parameters with actual parameters.

precedence—that property of an operator which determines how it shall be parsed with respect to adjacent operators (Language Report, Section VIII).

primitive—adjective applying to an entity whose meaning is given by the language definition—e.g., BOOLEAN or fixed-point addition.

priority—of operators, see *precedence*; of processes, a directive to the implementation advising which processes should be given preference in competition for the use of the processor.

process—one of several simultaneously executable program texts (Language Report, Section X.E).

pseudo-parallel—see *quasi-parallel.*

qualification—a modifier which, appropriately applied to a type, yields a representation of that type (Language Report, Section VI).

quasi-parallel—a section of program containing parallel processes that are not to be executed asynchronously in real time, but are to be executed sequentially while maintaining an appearance of parallelism with respect to a simulated time.

range—a form of qualification, consisting of a pair of bounds and applicable to integer and scalar types, specifying that values of the resulting representation must be within the specified bounds (Language Report, Section VI).

rational—the mathematical numbers exactly representable as the ratio of two integers (Language Report, Section VI.A).

record—a data structure containing elements, possibly of several different types, each of which is individually named (Language Report, Section VI.B).

representation—a machinable realization of a type including qualifications such as the range of values to be represented or the physical representation in storage (Language Report, Section VI).

reserved—lexical units whose meaning is defined by the language and can therefore not be used as identifiers or enumeration constants—e.g., ";", "BEGIN", etc. but not including identifiers with a predefined meaning such as "INTEGER".

routine—a procedure or a function.

row-major—that arrangement of the elements of an array in which the first dimension is varied most rapidly.

run time—the time of program execution.

scalar—a type or representation defined by a listing of its elements—i.e., an enumeration or alphabet (Language Report, Section VI.A).

scope—the section of program text within which a name has a single meaning (Language Report, Sections VII and X).

scope entry constant—an expression, used in a declaration, which must have a constant value defined no later than entry to the block within which the declaration occurs—e.g., the index range of an array (Language Report, Section VI.D).

segment—a portion of a program to be compiled separately (Language Report, Section XIII); also, a fragment of a syntax rule.

selector—the name of a record component (Language Report, Section VI.B).

semantic—concerning the meaning of the language rather than its grammar.

semaphore—a low-level mechanism used to control the interactions between two asynchronous parallel processes (Language Report, Sections XIV.VI and XIV.XI).

separate compilation—the process by which different parts of a program can be compiled at different times and subsequently linked together to form an executable program (Language Report, Section XIII).

sequential—in a single chronological sequence.

set—a type describing values which are mathematical subsets of the set of values of a base type, and upon which operations such as union and intersection are provided (Language Report, Section VI.C).

side effect—an assignment to a global variable made by a procedure.

signal—a high-level mechanism used to control the interaction of asynchronous parallel processes within a monitor (Language Report, Section X.E).

standard routine—a procedure or function that has been predefined in the language.

statement—a section of program text describing an action to be performed by the computer; also has a specialized syntactic meaning.

strongly typed—that property of a language whereby each value and variable has a specific type, known during compilation (Language Report, Section VI).

structured—of a statement or declaration, indicates that the statement or declaration is defined in terms of constituent statements and declarations; of a type or representation, indicates that the the type or representation is defined in terms of other types or representations; of a variable or value, indicates that the variable or value has a structured type; synonymous with composite.

subscript—index into an array.

suppress—the advice to the compiler that it need not generate object code to detect specified exceptional conditions at run time (Language Report, Section XIV.XIII).

synchronization—the operation of causing two asynchronous parallel processes to come into a particular time relationship, often causing one of them to be blocked until the other has completed some operation; see monitor, signal, and semaphore.

syntactic—relating to the grammar of the language (Language Report, Section III and Appendix B).

template—a schema for a program module, which may involve formal parameters instantiable using an INSTANCE declaration (Language Report, Sections X.C and X.D).

terminal—a syntactic primitive (Language Report, Section III).

transitive closure—the operation of computing, from a binary relation *R*, the relation R^* such that *a1* R^* *an* if and only if there exist *a2*, ..., *an* such that *a1 R a2 R ... R an*; computation required to determine whether a procedure or process activation is forbidden because of violation of the aliasing rules (Language Report, Section X.A).

translation time—at the time of compilation of the program.

transparent—that property of types and representations exported from modules, indicating that all program text within the scope of the defining module is permitted to access the internal structure of the type or representation (Language Report, Section X.B).

trap—an error indicated at run time by the hardware, nucleus, or compiled code (Language Report, Section XIV.X).

tuple—a sequence of values.

type—informally, a "kind of data" that may be manipulated by the program; more formally, the set of values that a variable can attain; also connotes the set of values together with a repertoire of applicable operations (Language Report, Section VI).

variant—one of the alternative values of a (record) type having alternative structures (Language Report, Section VI.B).

variant case selection field—a field of a variant record value used to determine which variant (q.v.) is represented by the value (Language Report, Section VI.B).

visible—that property of an entity that indicates that it can be referred to at a particular point in a program.

PART EIGHT

REFERENCES

REFERENCES

1. P. Brinch Hansen, "The Programming Language Concurrent Pascal," *IEEE Trans. on Software Engineering*, Vol. SE-1, No. 2, pp. 199-207 (June 1975).

2. P. Brinch Hansen, "Processes, Monitors, and Classes," *Software—Practice and Experience*, Vol. 6, pp. 165-200 (1976).

3. J. B. Goodenough, "Exception Handling: Issues and a Proposed Notation," *Comm. ACM*, Vol. 18, No. 12, pp. 683-696 (December 1975).

4. A. N. Habermann, "Critical Comments on the Programming Language Pascal," *Acta Informatica*, Vol. 3, pp. 47-57 (1973).

5. C. A. R. Hoare, "Notes on Data Structuring," in *Structured Programming*, O-J.Dahl et al., pp.83-174 (Academic Press, New York, 1972).

6. C. A. R. Hoare, private communication.

7. C. A. R. Hoare, "Monitors: An Operating System Structuring Concept," *Comm. ACM*, Vol. 17, No. 10, pp. 549-557 (1977).

8. J. J. Horning, private communication.

9. G. Holloway et al., "ECL Programmer's Manual," Center for Research in Computing Technology, Harvard University, Cambridge, Massachusetts (December 1974).

10. "Ironman," DoD High-Order Language Working Group, Department of Defense Requirements for High-Order Computer Programming Languages, Department of Defense (July 1977).

References

11. "Decsystem10 Assembly Language Handbook," DEC-10-NRCZ-D, Digital Equipment Corp., Maynard, Mass (1973).

12. C. Geschke and J. Mitchell, "On the Problem of Uniform References to Data Structures", *IEEE Trans. on Software Engineering*, Vol. SE-1, No. 2, pp. 207-219 (1975).

13. J. C. Reynolds, "Syntactic Control of Interference," Fifth ACM Symposium on Principles of Programming Languages, Tucson, January, 1978.

14. J. Welsh and C. Quinn, "A Pascal Compiler for ICL 1900 Series Computers," Dept. of Computer Science, Queen's University, Belfast (Sept. 1971).

15. N. Wirth, "Design and Implementation of Modula," *Software—Practice and Experience*, Vol. 7, pp. 67-84 (1977).

16. N. Wirth, "The Design of a Pascal Compiler," *Software—Practice and Experience*, Vol. 1, pp. 309-333 (1971).

17. N. Wirth, "Modula: A Language for Modular Multiprogramming," *Software—Practice and Experience*, Vol. 7, pp. 3-35 (1977).

18. N. Wirth, "The Use of Modula," *Software—Practice and Experience*, Vol. 7, pp. 37-65 (1977).

19. N. Wirth, private communication.

20. B. W. Lampson, et al., "Report on the Programming Language Euclid," *Sigplan Notices*, Vol. 12, No. 3 (1977).

21. W. H. Kaubisch, "Quasiparallel Programming," *Software—Practice and Experience*, Vol. 6, pp. 341-356 (1976).

22. K. Jensen and N. Wirth, "Pascal—User Manual and Report," in *Lecture Notes in Computer Science*, No. 18 (Springer-Verlag, New York, 1974).

23. K. V. Nori, et al., "The PASCAL(P) Compiler Implementation Notes," rev. ed. (Institut fur Informatik, Zurich, July 1976).

24. W. A. Wulf, R. L. London, and M. Shaw, "Abstraction and Verification in Alphard: Introduction to Language and Methodology," Carnegie-Mellon University and USc Information Sciences Institute Technical Reports (1976).

25. C. A. R. Hoare, private communication.

26. J. D. Ichbiah, J. P. Rissen, J. C. Heliard, and P. Cousot, "The System Implementation Language LIS Reference Manual," Technical Report 4549 E/EN, Compagnie Internationale pour l'Informatique (December 1974).

27. C. M. Geschke et al., "Early Experience with Mesa," *Comm. ACM*, Vol. 20, No. 8, pp. 540-552 (1977).

28. G. N. Beard, "The DoD COBOL Compiler Validation System," *AFIPS FJCC Conf. Proc.*, Vol. 41, Part II (1972).

29. B. A. Wichmann and B. Jones, "Testing Algol 60 Compilers," *Software—Practice and Experience*, Vol. 6 (1976).

30. D. Grune, "The MC Algol 68 Test Set," *Mathematisch Centrum*, IW 53/75, Amsterdam (December 1975).

31. W. Teitelman, "Interlisp Reference Manual," Xerox Palo Alto Research Center, Palo Alto, California (1975).

32. E. Satterthwaite, "Debugging Tools for High Level Languages." *Software—Practice and Experience*, Vol. 2, pp. 197-217 (1972).

33. C. A. R. Hoare and N. Wirth, "An Axiomatic Definition of the Programming Language PASCAL," *Acta Informatica*, Vol. 2, pp. 335-355 (1973).

34. R. L. London, et al., "Proof Rules for the Programming Language EUCLID," unpublished report (May 26, 1977).

35. B. Wegbreit and J. M. Spitzen, "Proving Properties of Complex Data Structures," *J. ACM*, Vol. 23, No. 2, pp. 389-396 (April 1976).

36. D. R. Musser, "A Proof Rule for Functions," Report ISI/RR-77-62, Univ. of Southern California, Information Sciences Institute, 4676 Marina del Rey, California (October 1977).

37. J. V. Guttag, J. J. Horning, and R. L. London, "A Proof Rule for EUCLID Procedures," Report ISI/RR-77-60, Univ. of Southern California, Information Sciences Institute, 4676 Marina del Rey, California (May 1977).

38. "PL/1 Checkout and Optimizing Compilers: Language Reference Manual," International Business Machines Corp., No. SC33-009-1 (1971).

39. S. Owicki and D. Gries, "An Axiomatic Technique for Parallel Programs I," *Acta Informatica*, Vol. 6, pp. 319-340 (1976).

40. S. Owicki, "Specifications and Proofs for Abstract Data Types in Concurrent Programs," TR 133, Digital Systems Laboratory, Stanford Univ. (April 1977).

41. R. Cartwright and D. Oppen, "Unrestricted Procedure Calls in Hoare's Logic," *Proc. Fifth ACM Sympos. on Principles of Programming Languages*, Tucson, Arizona (January 23-25, 1978).