UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

TATION PAGE

AD-A219 453	READ INSTRUCTIONS BEFORE COMPLETEING FORM
2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. Red Language Reference Manual	5. TYPE OF REPORT & PERIOD COVERED
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) John Nestor, Mary Van Deusen	8. CONTRACT OR GRANT NUMBER(s) MDA903-77-C-0330,-MOD P0003
9. PERFORMING ORGANIZATION AND ADDRESS Intermetrics, Inc. 701 Concord Avenue Cambridge, Massachusettes 02138	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Ada Joint Program Office United States Department of Defense Washington, DC 20301-3081	12. REPORT DATE March 8, 1979
	13. NUMBER OF PAGES 285
14. MONITORING AGENCY NAME & ADDRESS(If different from Controlling Office)	15. SECURITY CLASS (of this report) UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20 If different from Report)

UNCLASSIFIED

DTIC ELECTE MAR 0 2 1990 S B D

18. SUPPLEMENTARY NOTES

19. KEYWORDS (Continue on reverse side if necessary and identify by block number)

Ada Programming language, ANSI/MIL-STD-1815A, Ada Joint Program Office, AJPO

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

RED is a programming language designed, in accordance with the Dod "Steelman" requirements, for DoD embedded computer applications. The language combines features common to most existing high level languages with new capabilities for abstract data types, exception handling, multitasking, generic definitions and access to machine-depdendent facilities. This manual covers lexical structure, program structure, types, expressions, statements, procedures, functions and prameters, capsules, exception handling, multitasking, overloading and generics, machine dependent facilities, advanced definitions, and low level facilities.

DD FORM 1473 1 JAN 73 EDITION OF 1 NOV 65 IS OBSOLETE S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

RED LANGUAGE REFERENCE MANUAL

John Nestor and Mary Van Deusen
Intermetrics, Inc.
8 March 1979
IR-310-2

INTERMETRICS INCORPORATED • 701 CONCORD AVENUE • CAMBRIDGE MASSACHUSETTS 02138 • (617) 661-1840

This document is CDRL Item 0002AF, prepared under contract MDA903-77-C-0330, MOD P0003. The contract Technical Monitor for the government is Mr. William Carlson. The Project Manager for Intermetrics is Dr. Benjamin M. Brosgol.

ACKNOWLEDGEMENTS

The project manager is Benjamin Brosgol. The principal language designer is John Nestor. The authors of this document are John Nestor and Mary Van Deusen. The following individuals have participated in the design and review of the RED language: Peter Belmont, Toby Boyd, Robert Cassels, Dr. Mark Davis, Dr. Bruce Knobe, Dr. David Levine, Dr. Fred Martin, Eliot Moss, Craig Schaffert, Michael Tighe, and Mary Van Deusen. The project librarian is Judith Haigh. The diagrams were prepared by Robin Camardo and John Heymann

Substantial contributions have also been made by our consultants: Prof. Barbara Liskov (M.I.T.), Prof. William Wulf (Carnegie-Mellon), Prof. Paul Abrahams (N.Y.U), and Prof. Robert Dewar (N.Y.U).

Special thanks are made to Brian Reid (Carnegie-Mellon) whose SCRIBE document formatting program has greatly enhanced the appearance of this document.

We gratefully acknowledge the contributions of the many reviewers who have provided us with helpful comments and suggestions.

Table of Contents

1. INTRODUCTION

RED is a programming language designed, in accordance with the DoD "Steelman" requirements, for DoD embedded computer applications. The language combines features common to most existing high level languages with new capabilities for abstract data types, exception handling, multitasking, generic definitions, and access to machine-dependent facilities.

1.1 DESIGN GOALS

The RED language has been designed to reduce the total life cycle cost of designing, implementing, testing, and maintaining programs. For small and medium size programs, most modern high level languages are similar. However, for large programs, such as those commonly developed for embedded applications, the facilities provided by a language become crucial. What distinguishes the RED language is that it makes it easy to express solutions to the problems encountered when developing large programs.

Although many factors are involved in judging program quality, four key properties that will enhance the production of high quality programs have been specifically addressed in the design:

1) Modularity -- The program structure must be clearly modularized so as to facilitate design and maintenance.

2) Abstraction -- It must be possible to write programs in terms of a variety of abstractions that are appropriate to the application area, and in a notation that is in keeping with the style of the notation used for language-provided abstractions.

3) Reliability -- Coding and integration errors must be minimized either by elimination of whole classes of errors or by early detection.

4) Effectiveness -- The program must address the real problem and provide an effective solution. Use of assembly language should not be necessary.

Modularity

Cost effective program development and maintenance requires a modular design. The RED language provides a rich set of features for creating modules. Some of the kinds of modules supported are:

procedures
functions
tasks
data structures
abstract data types
schedulers
multitasking synchronization schemes
common data pools

These modules may be nested within a translation unit or may be separately translated in support of a large cooperative programming effort. Separate translation is provided as an integrated feature of the language.

The RED language also permits modules to be generalized by the use of its generic facility. For example, a sort procedure that sorts arrays with integer components can easily be generalized to sort arrays with components of any type.

Abstraction

The RED language allows existing language notations (e.g., operators) and features (e.g., the case statement) to be extended to encompass application specific abstractions. Once an abstraction has been written, it may be used as if it were a built-in feature of the language. Although application programmers may define their own abstractions (e.g. procedures and abstract data types), many centrally defined facilities (e.g. libraries, common routines and data, real time schedulers) will be written by systems programmers. For these kinds of abstractions, the language provides an extensive set of advanced features intended mainly for use by systems programmers. Once an abstraction is defined by a systems programmer, application programmers can use the abstraction without having to understand the advanced features used in its implementation.

Although the presence of advanced facilities in RED increases the apparent complexity of the language, it actually decreases the complexity of the overall programming process. Application programmers will be able to use more appropriate abstractions rather than having to work out less effective and less maintainable solutions to their problems.

Reliability

The RED language is designed to aid the programmer in the production of reliable programs. Particularly dangerous language features have been avoided. The language is fully type checked including the interfaces between separately translated modules. Extensive checking for error conditions is included; whenever possible, the checking is done during translation rather than at runtime. Facilities are also provided for detecting and handling runtime errors. Assertions may be specified at any point in a program, as an aid to program verification and as a way of detecting runtime errors. cases where efficiency is an overriding consideration, users can suppress the generation of code fo ecting runtime errors.

The scope rule ther with the capsule and expose declarations, allow users to completely ontrol the regions c program over which names are known.

Effectiveness

The RED language provides direct and convenient ways of dealing with real problems that have traditionally been either difficult or impossible to handle within a high level language. This means that users will not have to resort to assembly language to solve these problems. The RED language provides:

1) Access to machine-dependent features -- Facilities include the ability to specify physical representations, to access special memory addresses, to do hardware level I/O, and to handle hardware interrupts.

2) Control over all aspects of multitasking -- Users can define their own schedulers and synchronization schemes. Both multiprocessor systems with shared memory, as well as distributed systems, can be supported.

3) Control over storage management -- Users can select a dynamic storage management strategy that is appropriate to their application. In particular, applications which require dynamic storage management are not forced to pay the price of garbage collection, but can choose alternative methods.

1.2 SEMANTIC FRAMEWORK

This section discusses some of the key concepts which form the semantic framework for the RED language.

Scope Rules

A program consists of a nested set of scopes. When a name is used, a local definition is referenced if one exists; otherwise, a definition is sought for in enclosing scopes. There are two basic kinds of scopes: open scopes and closed scopes. Closed scopes differ from open scopes in that names of variable definitions in enclosing scopes may not be used unless they are explicitly imported.

A capsule is a scope with special properties. Definitions within a capsule are explicitly exported to those scopes which expose the capsule. Capsules are also the unit of separate translation. Definitions exported from one translation unit can be exposed in other translation units.

Immediate and Deferred Declarations

Declarations are divided into two groups: immediate declarations (e.g., a variable declaration) and deferred declarations (e.g., a procedure). Immediate declarations are elaborated when they are encountered, while deferred declarations are elaborated only when they are explicitly invoked.

Deferred declarations may have parameters, may be overloaded, and may be generic; immediate declarations may not.

Deferred declarations are closed scopes; immediate declarations are not scopes at all.

A body can contain declarations as well as statements. In a body, any declaration can appear before the statements; deferred declarations are also permitted to appear after the statements. All compound declarations (i.e., those containing bodies) are deferred declarations.

Types and Subtypes

Data items (e.g., variables) have two kinds of properties: those which must be known during translation (type properties) and those which must be known when a data item is created (constraint properties). A type consists of a type name and the type properties. A subtype consists of a type plus the constraint properties. The following are types:

```
INT
STRING[ASCII]
RECORD[c : INT, b : STRING[ASCII]]
```

The following are subtypes:

```
INT(1..10)
STRING[ASCII] (5)
RECORD[a : INT(0..i), b:STRING[ASCII] (j)]
```

Note that type properties are always enclosed in [ ], while constraint properties are always enclosed in ( ).

Subtypes are always specified for declared variables. For formal parameters and function results, either a type or a subtype is specified. A formal parameter which specifies a type can have actual

parameters with any subtype of that type.

Since types consist only of information known at translation time, all type checking (e.g., checking that the type of an actual parameter is the same as the type of the corresponding formal parameter) is done during translation. Types are compared by comparing their "extended name" (i.e., their identifier plus their type properties). Type checking does not involve structural equivalence (i.e., comparison of types which are recursive).

In addition to the rich set of built-in types, users can also define their own types in one of two ways: as "data structures" or as "abstract types". A data structure is an abbreviation for a composition of language types. For example,

```
ABBREV r : RECORD[a : INT(1..10),
                  b : STRING[ASCII] (5)];
VAR x : r;
```

is equivalent to

```
VAR x : RECORD[a : INT(1..10),
               b : STRING[ASCII] (5)];
```

An abstract type is a user-defined type (which is different from all other types) together with a set of procedures and functions which operate upon data items of the type. For example, a user can define a STACK type together with the PUSH and POP operations.

Multitasking

Concurrent elaborations are achieved via the multitasking facilities. Tasks are like procedures except they are invoked by a task invocation statement to produce a task activation. Activations of the task are elaborated concurrently with the invoker. Each activation of a task is named by a unique activation variable.

The elaboration of multiple tasks is under the control of a scheduler. In addition to a language defined priority scheduler, users can also define their own schedulers. The particular scheduler that is used for a task activation is selected based on the type of its activation variable.

Task activations can communicate in two basic ways: via shared memory or via message passing. Mutual exclusion over shared memory is achieved by datalocks (which are basically boolean semaphores) and a region statement. Message passing is supported by mailboxes (which are basically a queue of messages). A multiway wait statement is available which permits users to receive a message from any one of several mailboxes. Multiway waits on sending of messages are also provided. If there are several activations waiting to enter a region with some data lock, or to send a message to a mailbox, or to receive a message from a mailbox, they are queued in first-in first-out order.

Users can define their own synchronization schemes. This can be achieved either by defining these schemes based on datalocks or mailboxes or by way of the low-level multitasking facilities. Low-level multitasking facilities include latches (which are basically spin-locks) together with a standard set of low-level operations which are used to describe scheduling, region statements, and multiway waits. One important property of user-defined synchronization is that a particular scheme can be defined independent of any particular scheduler (i.e., it can be used without modification with any scheduler). Timing facilities are also available for measuring times or delaying based upon either

real time or activation times (i.e., the total time some activation has actually been running).

Generics

Any deferred declaration can be generic. A generic declaration is a generalized form from which a collection of different instances of the declaration can be derived. These instances are produced during translation. Instances differ in the values of a set of generic parameters. Generic parameters may be types, functions, procedures, tasks, or manifest values.

Generics are used to generalize a particular deferred declaration. For example, a generic sort procedure can sort arrays with components of any type. A generic stack type includes stack types which can have a component type (e.g., STACK [INT], STACK [STRING [ASCII]]). A generic integrate procedure can integrate any function, for example, from floats to floats.

Generic declarations allow the user to write a single definition which is specialized (i.e., instantiated) during translation for several specific uses, rather than having to write a separate definition for each of the separate uses.

1.3 PROGRAMMING CONSIDERATIONS

The characteristics of the RED language discussed above represent a solution to the problem of providing a standard language for military software production, one that can serve all applications without ignoring the special requirements of each. The solution presented here is based heavily on the data abstraction capabilities that permit the same language to be specialized as needed, but in a form that is invisible to the applications programmer and, perhaps more importantly, to the maintenance programmer. Changes required can be implemented in terms of underlying definitions so that most often programs need not be changed at all in order to operate differently. Such underlying modifications can, further, affect many applications programs, so that the maintenance effort is substantially reduced along with maintenance costs.

In order to provide comprehensive support within the context of one high-level language, the RED language necessarily includes complex features that will neither be needed or necessarily understood by all programmers. By separating out these complex features, it has been possible to retain a core of basic programming facilities that are similar to most other languages and, thus, easy to learn and use, yet flexible enough so that sophisticated applications can be expressed using only these basic facilities.

1.4 OVERVIEW OF THE LANGUAGE REFERENCE MANUAL

This LRM is divided into four major parts:

GREEN	(Chapters 1-7) - Basic Language Features. The features described here will be needed by all users. This part of the language is roughly equivalent to the PASCAL language. Simple programs can be written using only these features.
YELLOW	(Chapters 8-11) - Intermediate Language Features. The features described here will be needed by most users.
RED	(Chapters 12-14) - Advanced Language Features. The features described here are provided mainly for use by systems programmers, rather than by application programmers.
BLUE	Appendices, Index.

The division into basic, intermediate, and advanced parts is only approximate. Although most features described in a chapter belong in the part in which the chapter is placed, some features discussed may conceptually belong in some other part. For example, the type declaration is intermediate rather than basic, some aspects of the use of generics are advanced rather than intermediate, and definition of operators is intermediate rather than advanced.

1.5 MANUAL LAYOUT

This document is a language reference manual, designed to provide the user with a complete description of the format, usage and effects of all language features. Basic lexical elements of the language are described first (Chapter 2). Subsequent chapters present the various language constructs.

Each section of the manual follows the same basic five-part form given below, although any of the five parts may be omitted when it is not applicable.

1) Diagrams - A flow diagram format (described in Section 1.6) is used to specify the form for lexical elements and the syntax for language constructs.

2) Informal Description - The text immediately following the diagrams informally describes the purpose, use and meaning of the lexical element or language construct.

3) Rules - The heading RULES indicates that the following text gives rules completely defining the meaning of the lexical element or language construct, that are not already given by the syntax.

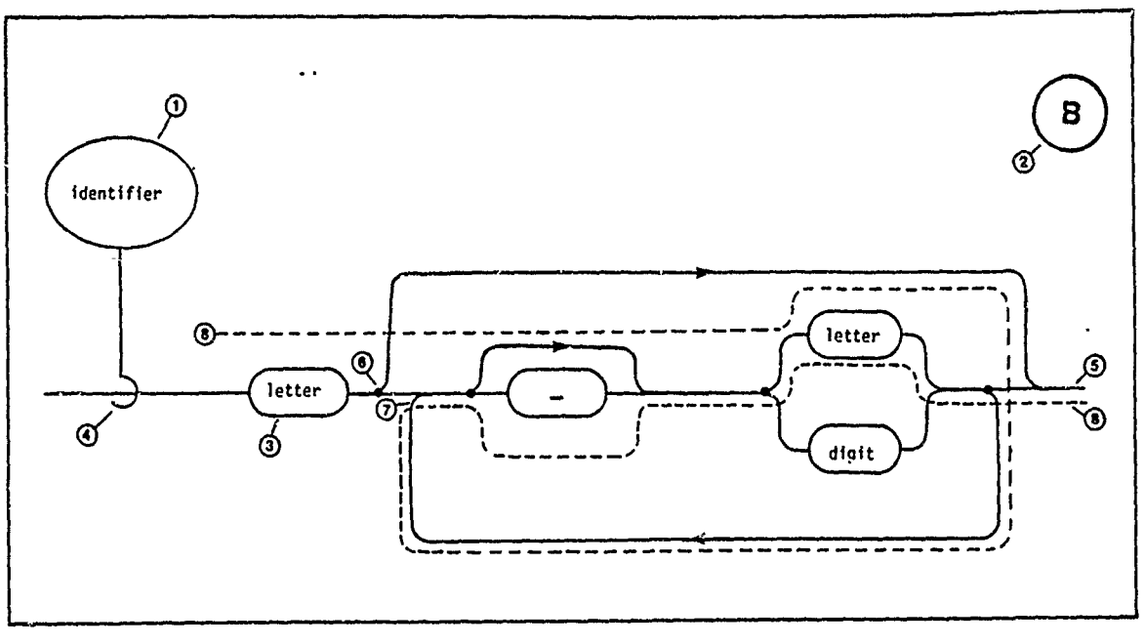
4) Notes - The heading NOTES indicates that the following text describes how the lexical element or construct interacts with other parts of the language. Rules from other sections, which are relevant to this lexical element or construct, may be summarized.

5) Examples - The heading EXAMPLES precedes sample coding sequances that illustrate the various valid forms of lexical elements or the use of language constructs.

1.6 FLOW DIAGRAMS

Flow diagrams are used in this manual to specify all the forms of a single lexical element or language construct. By tracing a path through a diagram, an instance of the element or language construct represented by that diagram may be produced. There is a path through a diagram for every valid instance. These diagrams, together with the rules, provide a complete description of the language. Rules for interpreting these diagrams are given below.

1.6.1 FLOW DIAGRAMS FOR LEXICAL ELEMENTS



RULES

1) Each diagram defines the forms for a particular lexical element (see Chapter 2). The name of the element being defined appears in the oval, (1), at the upper left of the diagram.

2) The diagram identifier, the letter in the upper right hand corner, (2), is associated with this specific diagram. An index of diagram identifiers may be found in Appendix F . In the example illustrated, the syntax diagram defines all possible forms of an identifier token.

3) Boxes with circular ends, (3), represent lexical elements or characters from which they are constructed.

4) If the rounded box represents an element defined in another syntax diagram, a letter above the rounded box is the diagram identifier associated with that element.

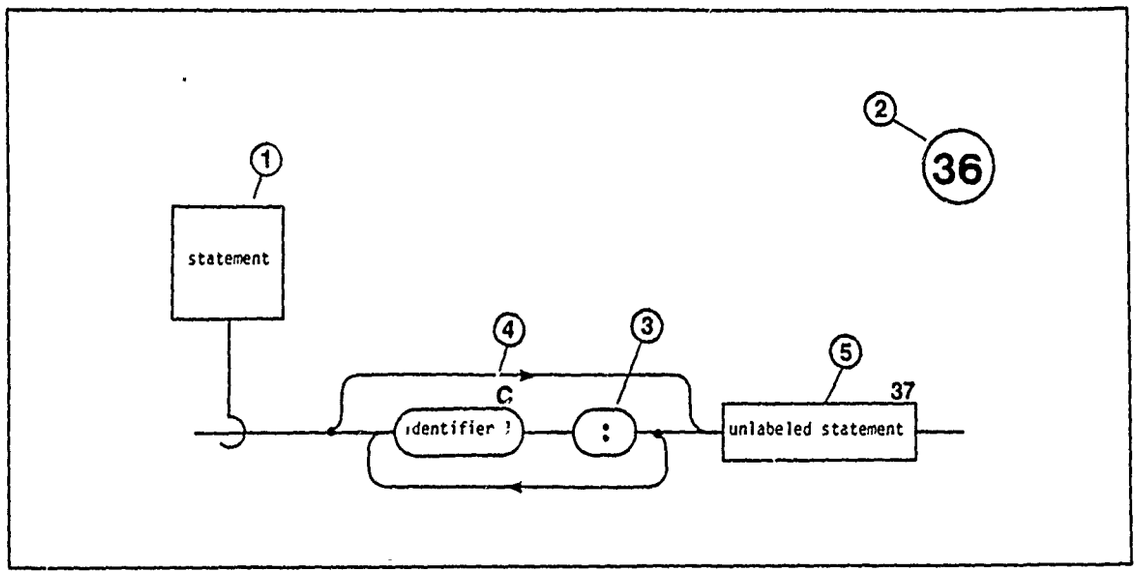
5) To generate forms of the element, the diagram is followed from left to right, from box to box, starting at the point of the junction of the definition box, (4), and ending when the end of the path, (5), is reached.

6) When, in following a diagram, a black dot, (6), is reached, any of the paths leaving the dot may be followed.

7) It is not legal to "back up" along a convergent path, (7).

8) When a box is encountered, the element it contains is added to the right of the preceding element. For example, the path shown by the dotted line, (8), generates the sequence "letter letter underscore letter" (e.g., AB_C).

1.6.2 SYNTAX DIAGRAMS FOR LANGUAGE CONSTRUCTS



The diagrams defining language constructs are similar to those for lexical elements with the following differences:

1) The name of the language construct being defined appears in a rectangular box, (1), in the upper left-hand corner. The illustrated example defines the syntax of a statement.

2) The diagram identifier, (2), for language construct diagrams is an integer. An index of diagram identifiers may be found in Appendix F .

3) Boxes with circular ends, such as, (3), represent lexical elements; reserved words appear in capital letters. A letter above the box, (4), identifies the diagram in which the element is defined.

4) Rectangular boxes within the diagram, such as (5), represent language constructs defined elsewhere. If a number appears above the box, the construct is defined in the diagram identified by that number.

5) Following a path through a grammar syntax diagram produces an instance of the construct.

2. LEXICAL STRUCTURE

2.1 CHARACTER SET AND TRANSLATOR INPUT

Programs are composed of any sequence of characters from the 95-character ASCII or basic 55-character set. Any program can be written using only the 55-character set given below. Rules for converting from the 95-character ASCII set to the 55-character set are given in the description for specific tokens and token separators.

RULES

No distinction is made between upper and lower case letters except within a *string literal*.

Basic 55-Character Set

```
 %&'()*+,-./:;<=>?
Ø123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ_
```

95-Character ASCII Set

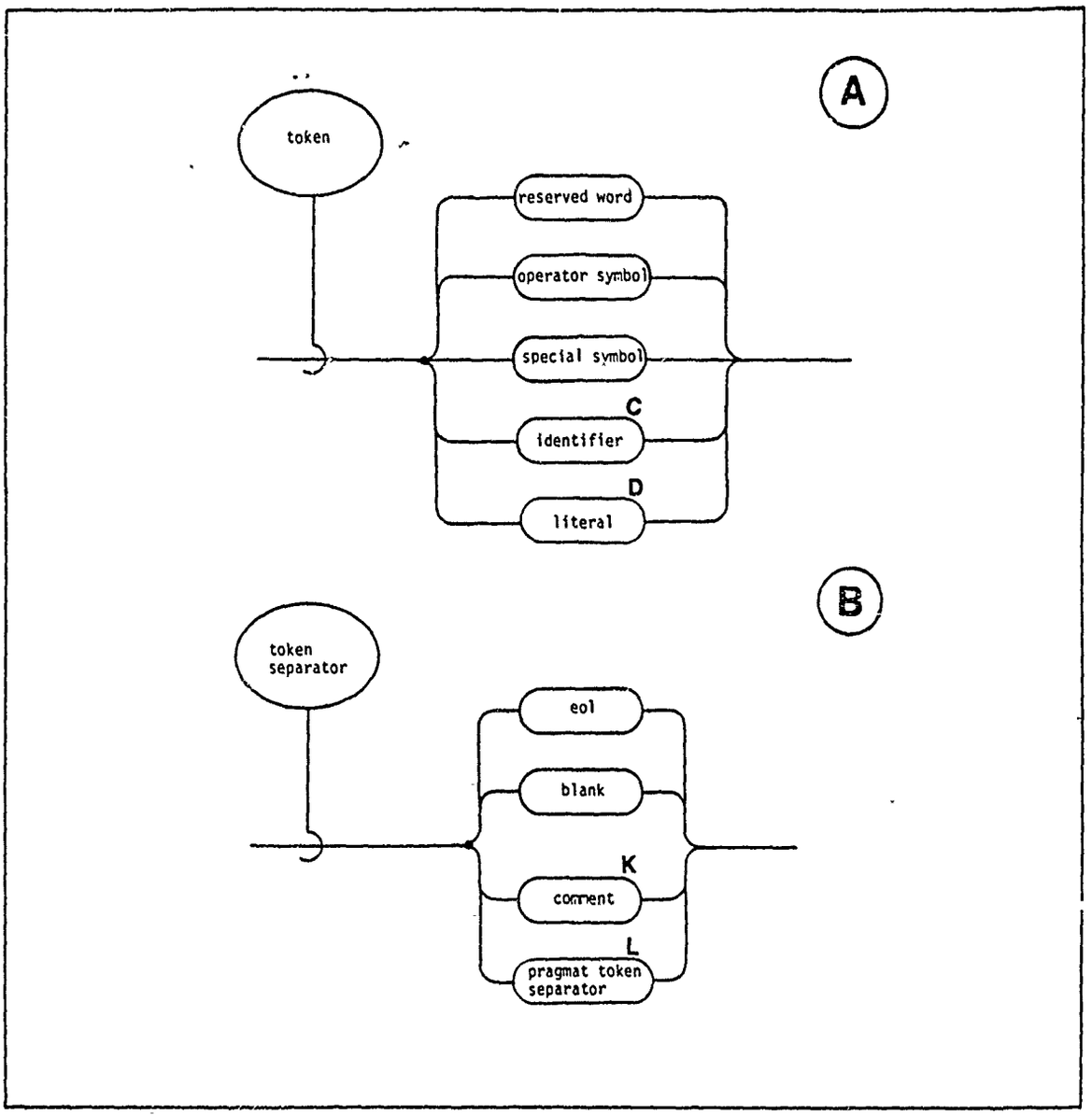
All characters in the basic 55-character set plus

```
!"#$@[\]^
`{|}~
abcdefghijklmnopqrstuvwxyz
```

NOTES

This document uses the 95-character ASCII set to describe the language.

2.2 TOKENS AND TOKEN SEPARATORS



A *token* is the basic component used to build all constructs of the language. It is an indivisible lexical unit that is interpreted as a complete 'word' by the translator. A *token separator* is required in some cases between *tokens* and can be used otherwise to improve readability. A *token* or *token* separator is composed of a contiguous sequence of characters.

RULES

Input text is organized into lines, each of which is composed of *tokens* and *token separators*. No *token* or *token separator* can extend over more than one line of text. An end of line, eol, is a token separator.

A *token separator* must appear between any two adjacent *tokens*, unless one of the *tokens* is a

special symbol or an operator symbol (such as <>) which does not have the form of an identifier (e.g., AND). One or more token separators may appear between any two tokens.

EXAMPLES

```
PERSON.WOMAN.ADA
NOT A                   % token separator required
F ( A : INT ) ;
F(A:INT);
END CASE                % token separator required
WHEN A=>                % token separator required
```

2.3 TOKENS

2.3.1 RESERVED WORDS

Reserved words have a fixed meaning within the syntax of the language.

RULES

The following are reserved words:

ABBREV	END	LOCATION	RENAMING
ABNORMAL	EXCEPTION	NAMED	REP
ALL	EXIT	NEEDS	REPEAT
ALLOC	EXPORTS	NEW	RERAISE
ASSERT	EXPOSE	NONE	RETURN
BEGIN	EXTERNAL	OUT	SUBTYPE
BY	FOR	OF	TASK
CAPSULE	FROM	PRAG	THEN
CASE	FUNC	PROC	TO
CONST	GENERIC	PTR	TYPE
CREATE	GOTO	RAISE	VAR
DO	GUARD	READONLY	WAIT
ELSE	IF	REVERSE	WHEN
ELSEIF	IMPORTS	REGION	WHILE

NOTES

Reserved words may not be redefined.

No distinction is made in the use of upper or lower case characters in a reserved word; thus, end, End, END, and enD are all equivalent.

2.3.2 OPERATOR SYMBOLS

Operator symbols are names of functions which are invoked with a special prefix or infix syntax (see Section 5.2).

RULES

The operator symbols are:

**	exponentiation
*, /	multiplication, division
DIV, MOD	integer division, modulo
+, -	addition, subtraction, prefix plus and minus
&	concatenation
=, /=, >, >=, <, <=	relations
IN	set membership
OR	or (set union),
XOR	exclusive or (set symmetric difference)
AND	and (set intersection)
NOT	logical negation (set complement)

NOTES

The definition of operator symbol names is discussed in Section 13.2.

2.3.3 SPECIAL SYMBOLS

Special symbols are *tokens* which have special meaning in the syntax.

RULES

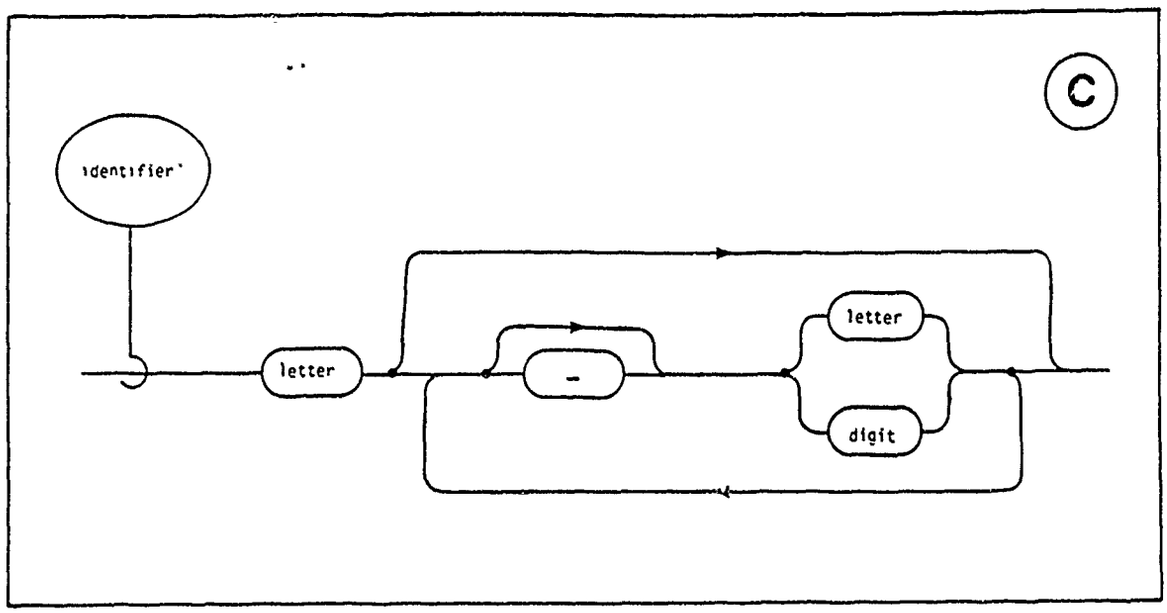
The table below lists the special symbols and their uses.

.	component selection, attribute inquiry
( )	parenthesization, subscripting, parameter lists
[ ]	type properties, translation time properties, constructors
,	list separation
:	name separation, goto labels
;	statement terminator, end of compound declaration headers
=>	alternative indication, function result
..	ranges
#	constant resolution
:=	assignment

All of the special symbols, except [,], and #, consist of characters exclusively from the 55-character set. The following 55-character alternates are provided.

<<	55-character form of [
>>	55-character form of]
::	55-character form of #

2.3.4 IDENTIFIERS



An *identifier* is a name which is associated with a language construct by a definition (see Section 3.5).

RULES

Reserved words and operator symbols (e.g., AND) may not be used as *identifiers*.

All characters in an *identifier*, including underscore, are significant.

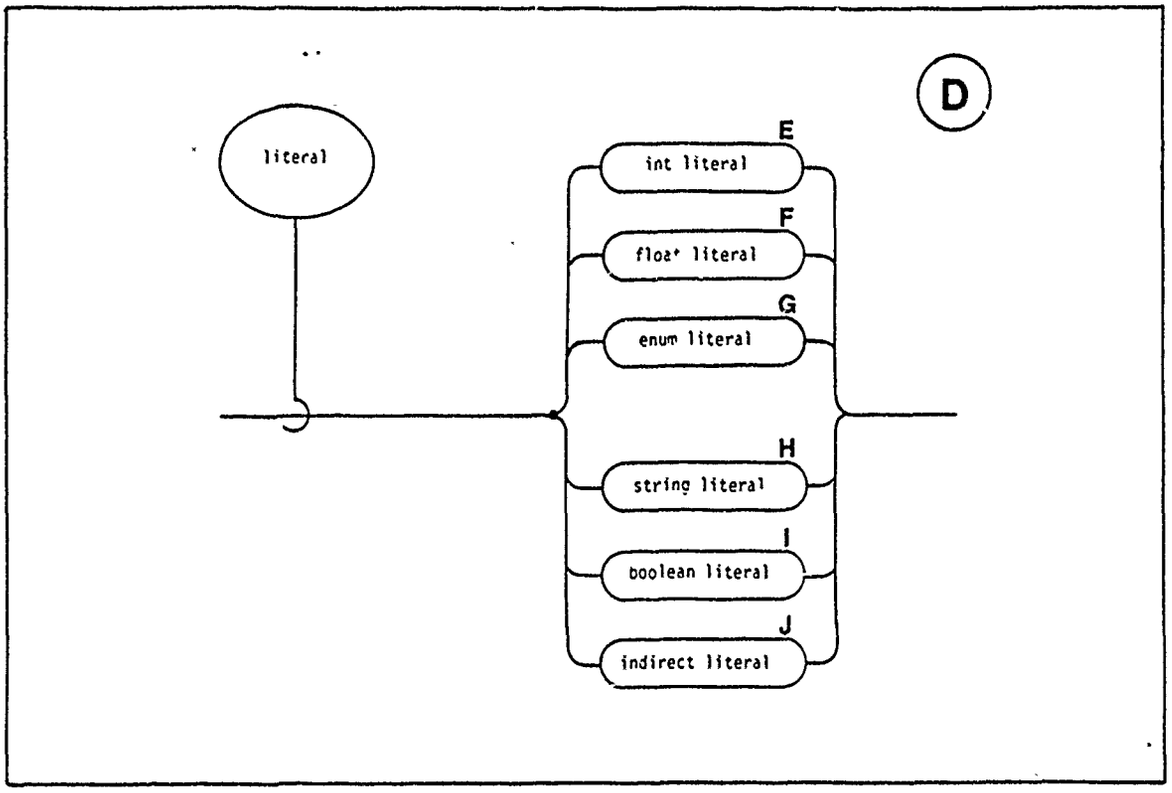
NOTES

No distinction is made in the use of upper or lower case characters in an *identifier* (e.g., Abc, abc, and ABC are all equivalent).

EXAMPLES

```
THIS_IS_A_VERY_LONG_NAME
This_is_a_very_long_name     % same as above
VELOCITY
UNIT_01
REAR_UNIT_01
THIS_IS__ILLEGAL             % two underscores together
AS_IS_THIS_                  % illegal
```

2.3.5 LITERALS



Literals are used to specify values for some built-in types and for user-defined indirect types (see Section 4.4.3).

RULES

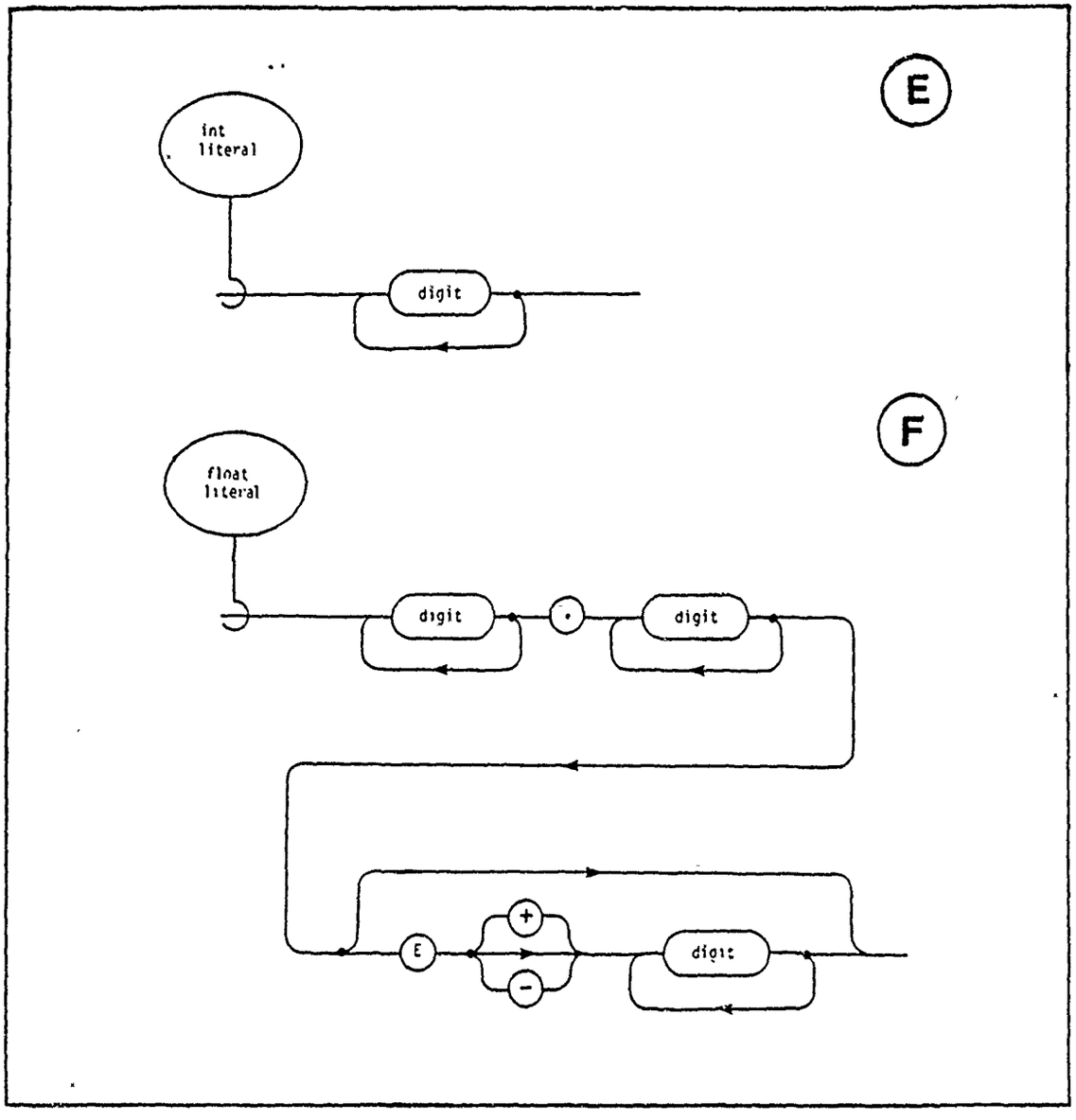
The values of all *literals* are known at translation time.

The rules for resolution of the *type* and *subtype* of a *literal* are described in Section 5.7.

NOTES

The following sections describe specific *literals*. User-defined literals, which are language constructs rather than *tokens*, are described in Sections 5.7 and 13.5.

2.3.6 NUMERIC LITERALS



Numeric literals specify integer values for the INT type and floating point values for the FLOAT type.

RULES

A floating point literal in E form is interpreted as the decimal number times ten to the integer value following E. The default precision of a floating point number is the number of digits preceding E, minus any leading zeros to the left of the decimal point.

NOTES

Numeric Literals are always positive values. A negative literal is obtained by preceding the *literal* with the prefix minus operator. This operation is performed at translation time.

Because a *float literal* is a *token*, blanks may not appear within the *float literal*.

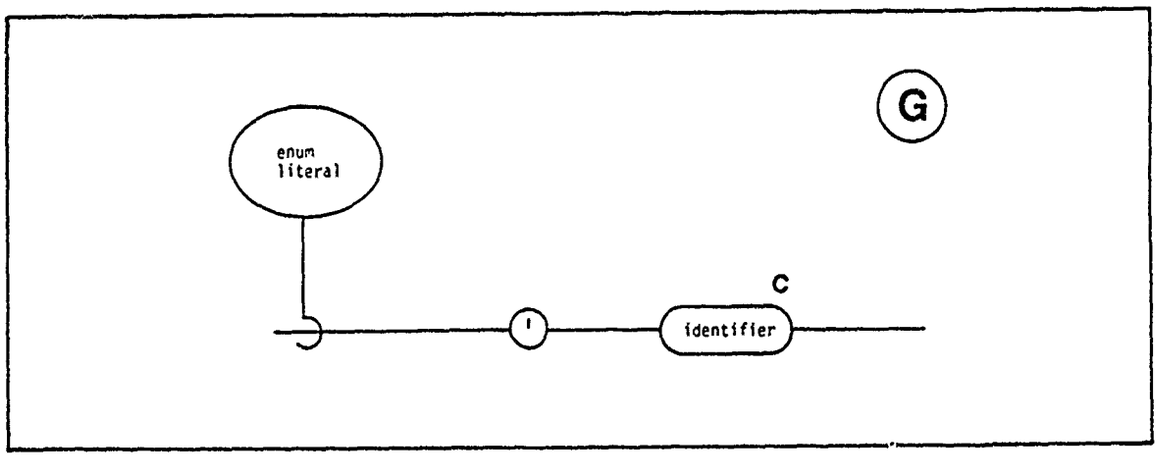
The precision of a *float literal* is first determined by context and, if that is not sufficient, the default precision is used (see Section 5.7).

No distinction is made in the use of upper or lower case e in a *float literal*.

EXAMPLES

```
% integer literals
0
2354

% floating point literals
3.6E7            % default precision 2
1.0E5            % default precision 2
7.25             % default precision 3
6.324e-6         % default precision 4
0.0              % default precision 1
0.012            % default precision 3
```

2.3.7 ENUM LITERALS



An *enum literal* specifies a named value of an enumeration type.

NOTES

The same enumeration literal may appear in several enumeration types. For example, the enumeration literal 'ORANGE may appear simultaneously in the ENUM types FRUIT and COLOR.

Because an *enum literal* is a *token*, no blank may appear between the apostrophe and the *identifier*.

Because the *enum literal* is distinguished by an apostrophe, the *identifier* following the apostrophe may be defined in the same scope (i.e., RED may be the name of a variable in the same scope in which 'RED is an *enum literal*).

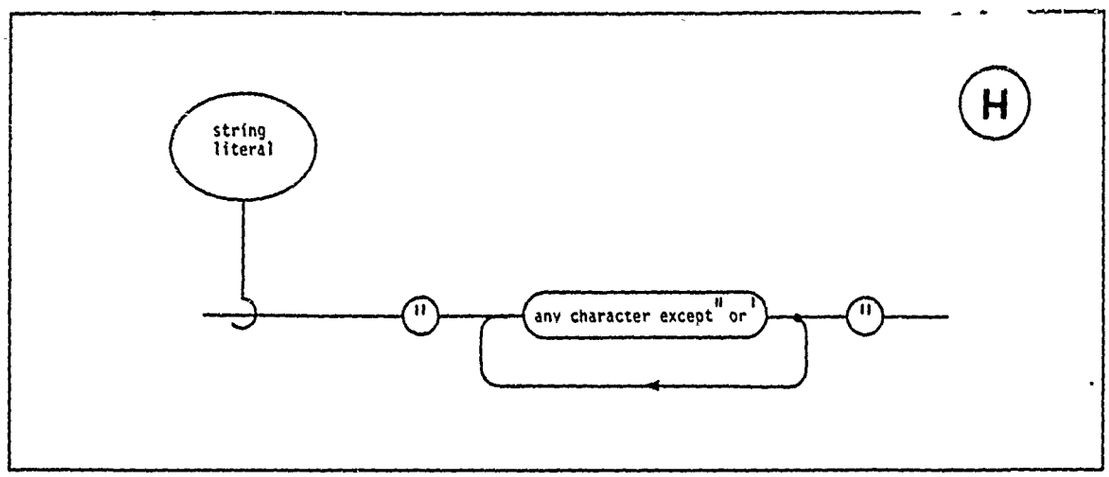
The language views a character set as an enumeration type, where each character corresponds to an *enum literal*.

No distinction is made in use of upper or lower case (e.g., 'RED, 'Red, 'red, and 'reD are all equivalent).

EXAMPLES

```
'RED
'POSITIVE
'ACTIVE
'LEFT_ALIGNED
'Right_Aligned
'RED'          % illegal closing apostrophe
'4TH           % illegal character beginning identifier
```

2.3.8 STRING LITERALS



A string literal specifies a value of a STRING type. A string is a sequence of characters. Each character is an enumeration literal. If a string includes only those characters in the 95-character set, the special literal form, *string literal*, can be used. A *string literal* is considered to be a shorthand form for the concatenation of characters defined by *enum literals*; e.g., "ABC" is considered a shorthand form of

```
'A & 'B & 'C
```

RULES

'' is the 55-character form of ".

NOTES

A *string literal* may not extend over one line. If an end of line is found before the terminating quote, the string literal token is terminated and an error message is issued. The & infix operator can be used to obtain a long string by concatenation.

Upper or lower case characters are distinguished in strings; e.g., "ABC" is not equivalent to "abc". All characters in the 95-character ASCII set have a corresponding enumeration literal (defined in Appendix C.15). For example, since 'lbracket is the enumeration literal for [and 'number the enumeration literal for #, the string

```
"[A#B]"
```

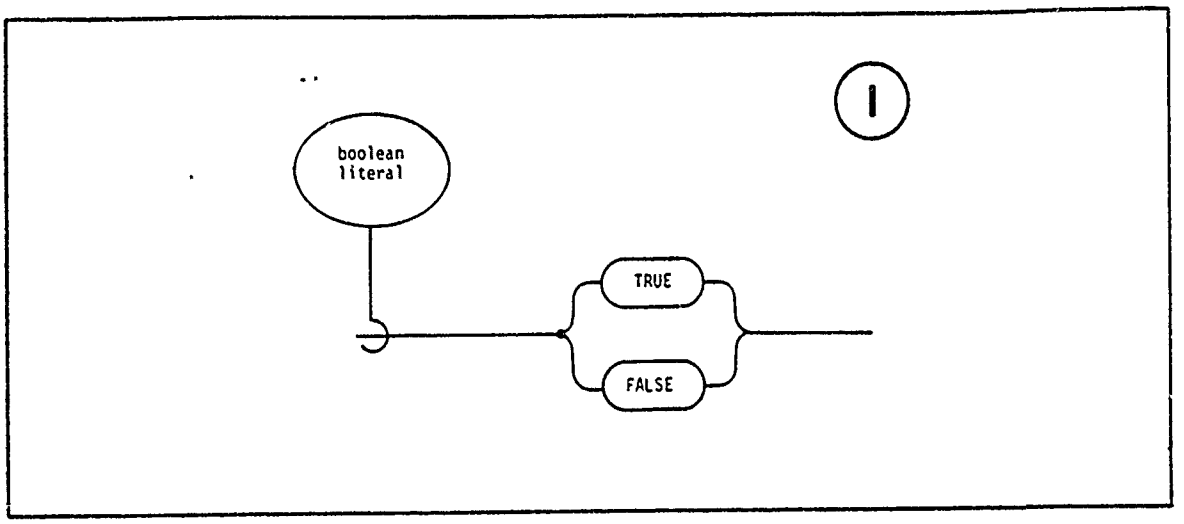
is equivalent to

```
'lbracket & "A" & 'number & "B" & 'rbracket
```

EXAMPLES

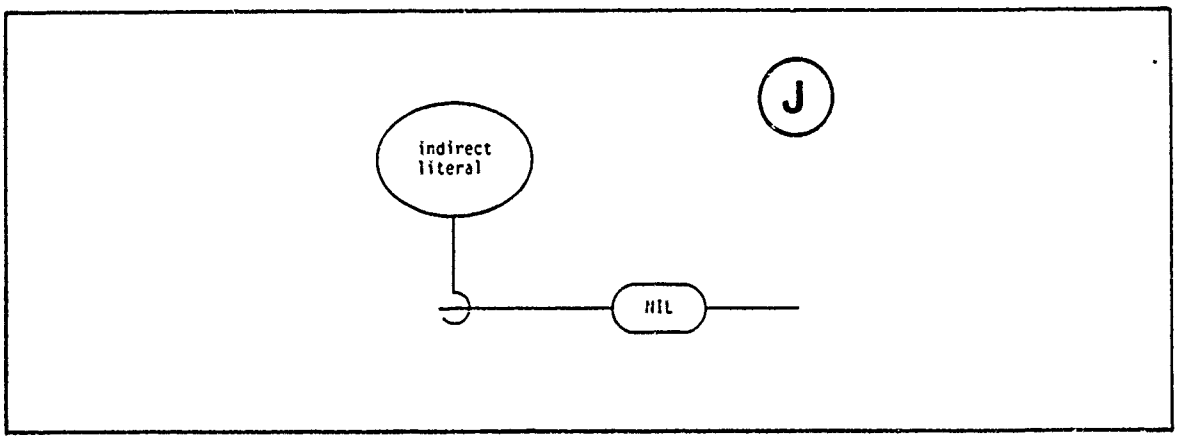
```
% creating a string too long for a single line

"VERY...LONG...STRING"
& "REST OF VERY LONG STRING"

% placing carriage return and line feed
%  at the end of a string
   "ONE LINE" & 'CR & 'LF
```

2.3.9 BOOLEAN LITERALS



A boolean literal specifies a value of type BOOL.

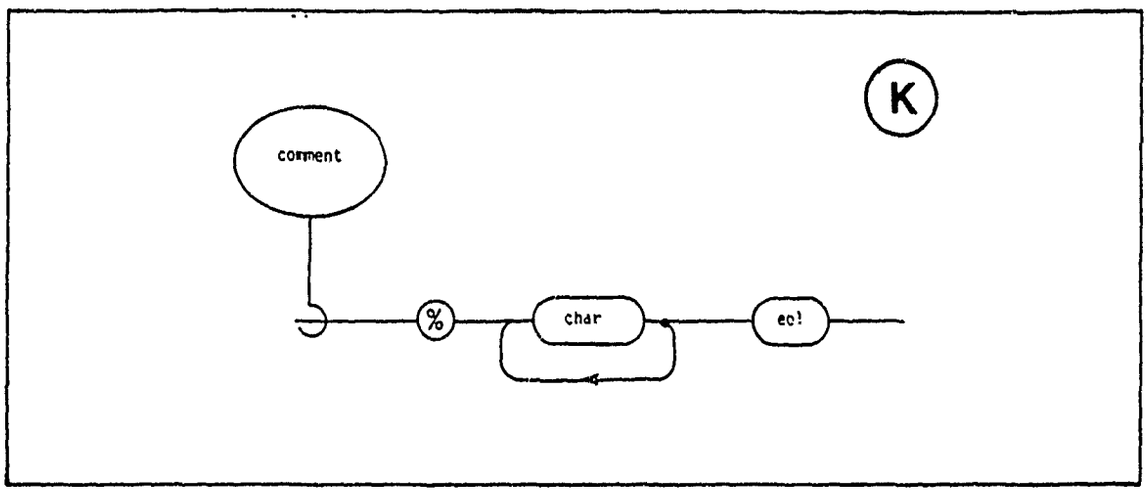
2.3.10 INDIRECT LITERAL



The indirect literal NIL specifies that value of an indirect type (see Section 4.4.3) which points to no dynamic variable.

2.4 TOKEN SEPARATORS

2.4.1 COMMENT

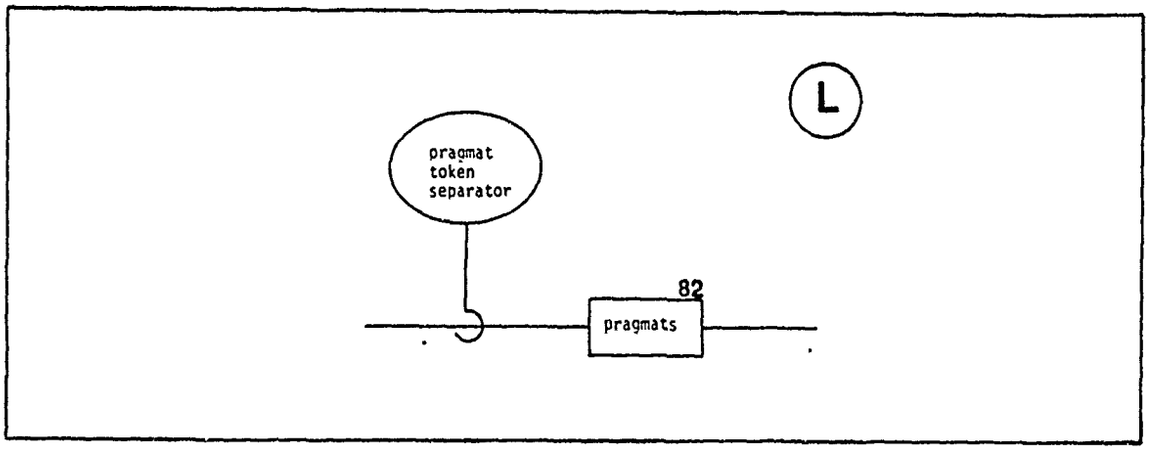


A *comment* provides program documentation.

RULES

A *comment* is terminated by the end of the line on which it appears. *Comments* are ignored by the translator.

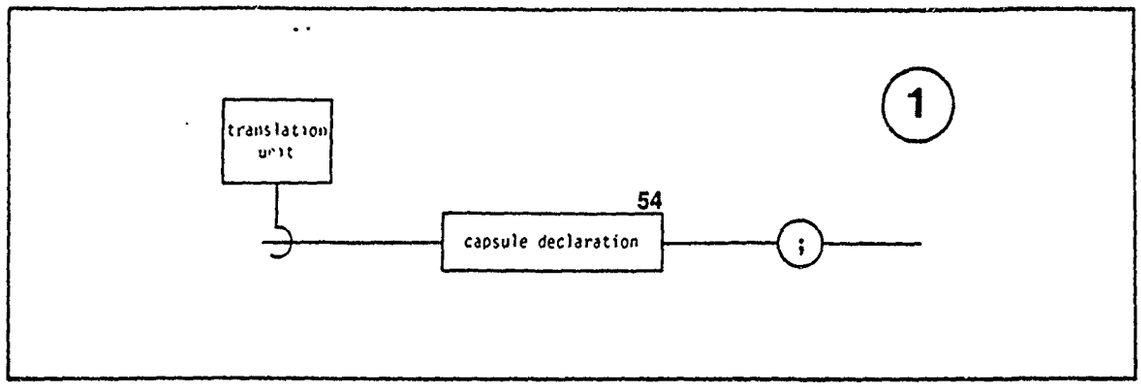
2.4.2 PRAGMAT



pragmats supply information to the translator which does not affect language semantics. *Pragmats* are described in Appendix B.

3. PROGRAM STRUCTURE

3.1 PROGRAM



A program consists of one or more *translation units* which may communicate with each other. Each *translation unit* is a *capsule declaration*. *Capsule declarations* may also be nested within *translation units* and are described more generally in Chapter 8. A *capsule declaration* consists of a header, a body, and an ending. The header names the capsule and provides an exports list that makes definitions within the capsule available to other translation units. The body consists of a sequence of *declarations*, *statements*, and *assertions*. *Declarations* provide definitions for names, *statements* specify actions to be performed, and *assertions* specify conditions that are to be true at the points where the *assertions* appear. The ending terminates the text of the *capsule declaration*.

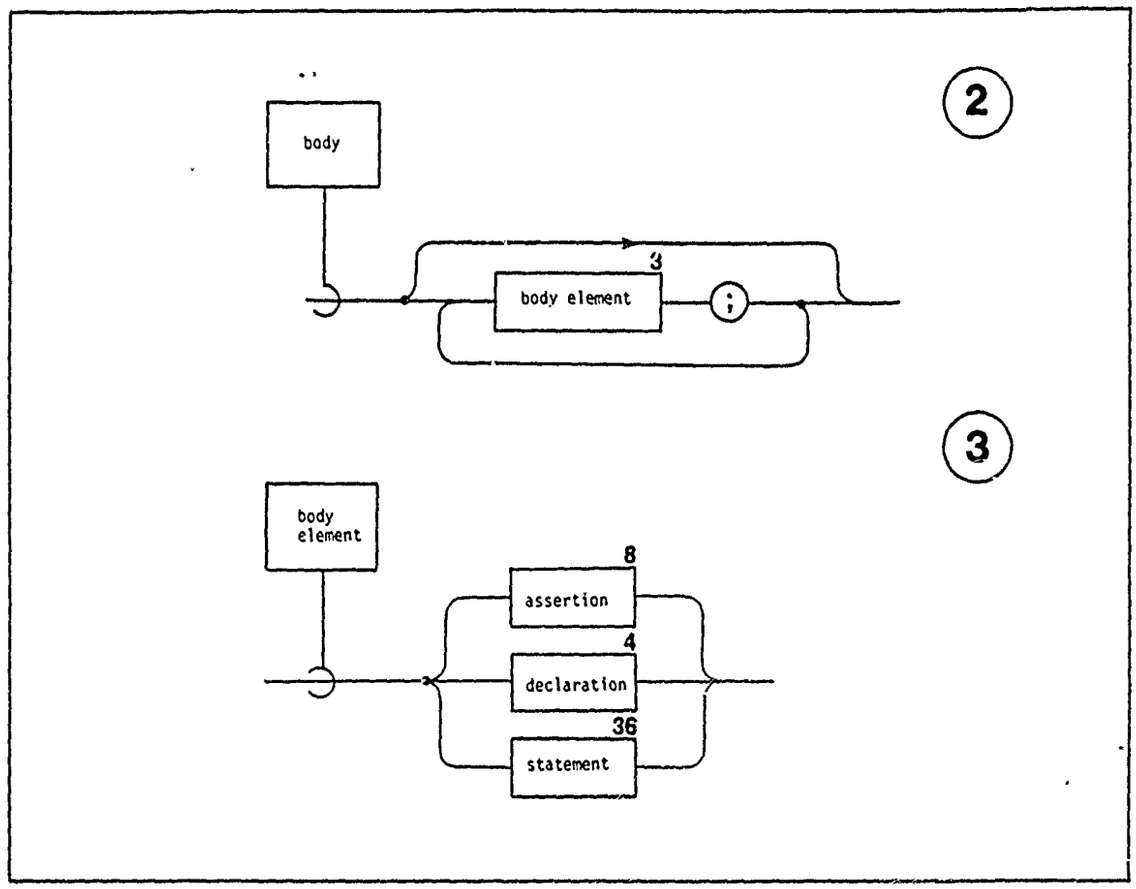
The intent of a program is realized by elaborating the program. The notion of elaboration is meant to provide a general way of describing closely related translator functions, such as execution for *statements* and evaluation for *expressions*. Elaboration can apply to *statements*, *assertions*, *declarations*, and *expressions*, and includes translation-time as well as execution-time activities. The elaboration of a compound syntactic unit is defined in terms of the elaboration of its constituent units.

In the simplest case, a program consists of just one *translation unit*. The invocation, initiated by the programming system, consists of elaboration of the *body* of that unit. When there is more than one *translation unit*, the user must select (via the programming system) a particular one to be invoked as the main capsule.

EXAMPLES

```
CAPSULE prime EXPORTS NONE;
   CONST max := 100;
    % this program prints all prime numbers that are
    %  less than or equal to max
   VAR numbers : ARRAY INT(2..max) OF BOOL
                 := [2..max : TRUE];

   FOR i : INT(2..max) REPEAT
      IF numbers(i) THEN
         VAR j : INT(2*i .. max+i) := 2*i;
         WHILE j <= max REPEAT
            numbers(i) := FALSE;
            j := j+i;
         END REPEAT;
      END IF;
   END REPEAT;

   OPEN (SYS_OUT, "TTY", 'NEW)
   WRITELN ("PRIME NUMBERS");
   FOR i : INT(2..max) REPEAT
      IF numbers(i) THEN
         WRITELN  (i);
      END IF;
   END REPEAT;
   CLOSE (SYS_OUT, 'SAVE);

END CAPSULE prime;
```

3.2 BODY



A *body* consists of a sequence of *body elements*, each of which is either a *declaration*, a *statement*, or an *assertion*. A *body* is used where a related sequence of *body elements* must be treated as a unit. *Statements* and *declarations* that can contain *bodies* are known as *compound statements* and *compound declarations*. For example, one possible form of an *if statement* is:

```
IF expression THEN body1 ELSE body2 END IF
```

In this example, body1 and body2 specify the actions to be taken after elaboration of the expression yields true or false, respectively. Depending on the *bodies* given, the actions may range from elaboration of a single element to a complex sequence of actions.

Empty bodies are permissible; this is useful when no action needs to be taken.

RULES

An *immediate declaration* (see Section 3.3) must precede all *statements*. A *deferred declaration* (see Section 3.3) or *generic declaration* (see Section 11.3) may either precede or follow all *statements*.

Elaboration of a *body* consists of the elaboration of all *body elements* other than *deferred* and *generic declarations*. The elements are elaborated in the sequence in which they are written, unless control is transferred by an *exit*, *return*, or *goto statement*.

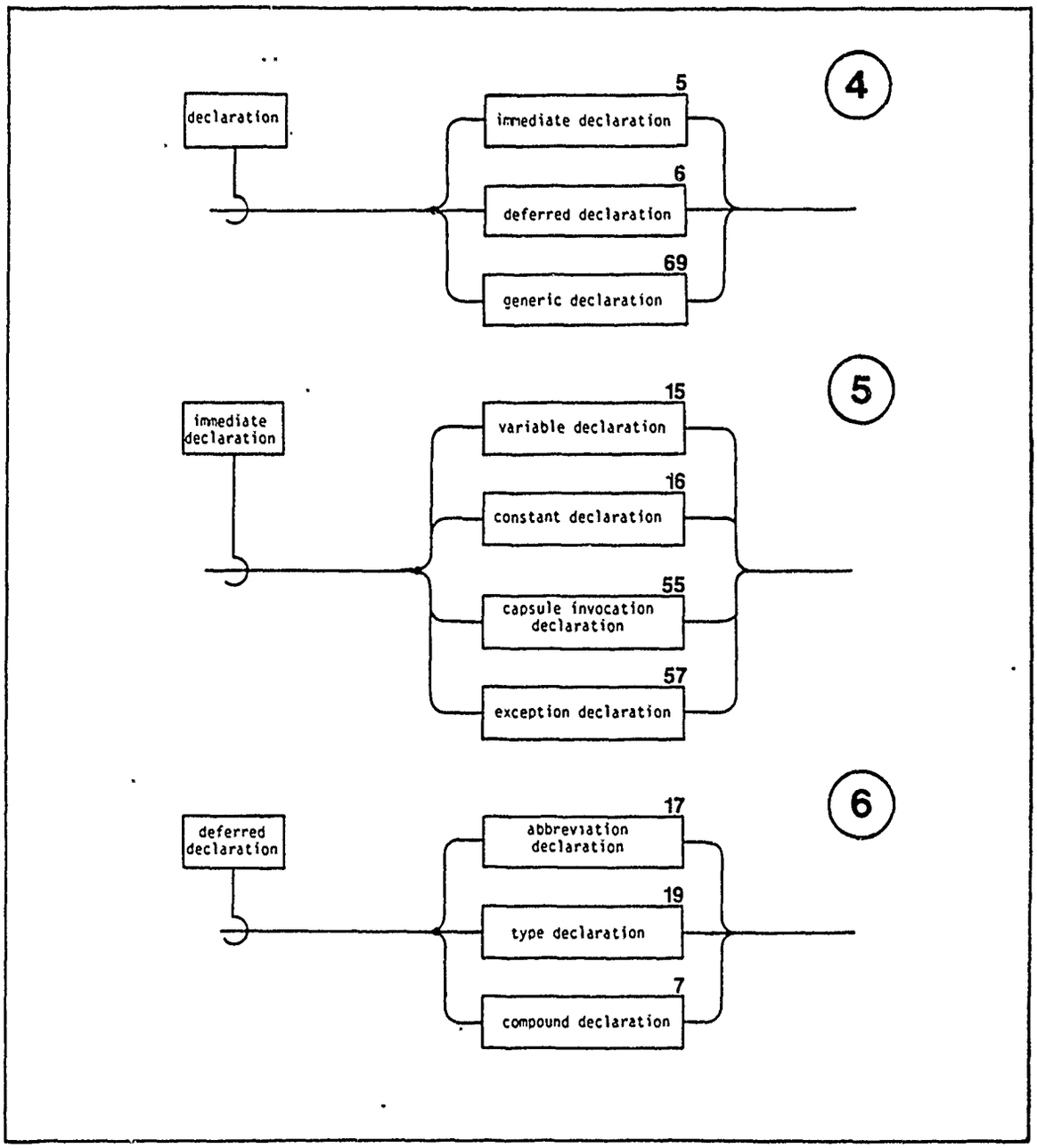
NOTES

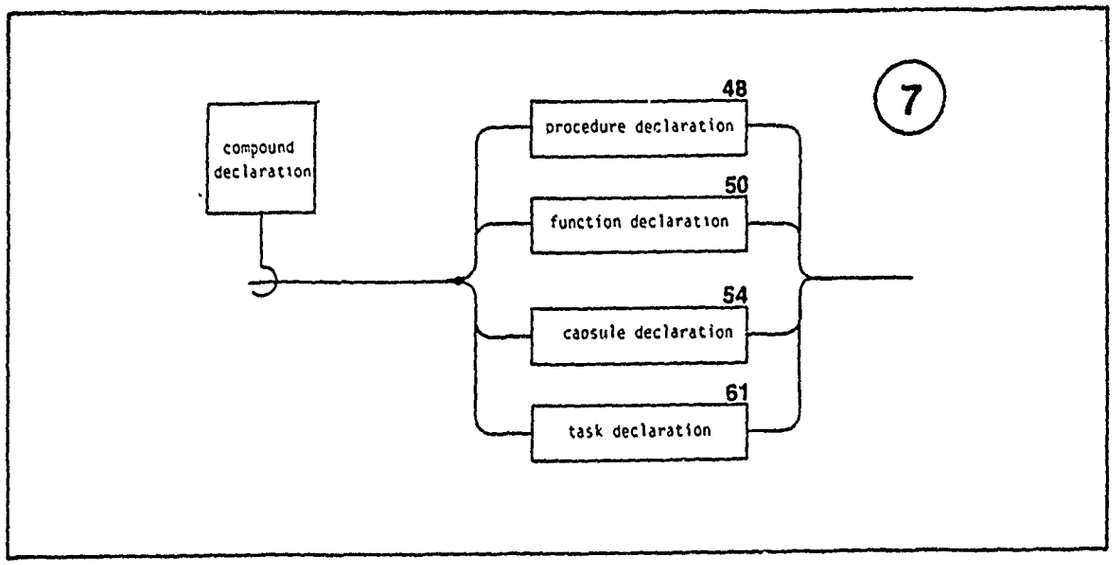
A *body* is an open scope (see Section 3.5). *Declarations* and goto labels define names within a *body*.

An *assertion* may appear at any point in the sequence of *body elements* since it may be useful before or after either a *declaration* or a *statement*.

Since *deferred declarations* and *generic declarations* are sometimes quite long, placing them after the *statements* often makes a program more readable. Placement of *deferred declarations* can be determined by programming standards and style considerations.

3.3 DECLARATIONS





All *declarations* define names. There are two basic kinds of *declarations*: immediate and deferred. *Immediate declarations* are elaborated when encountered during the elaboration of a *body*. *Deferred* declarations are not elaborated when first encountered; instead, they define deferred units which are elaborated only when invoked from elsewhere.

A *generic declaration* stands for a collection of *deferred declarations* and, therefore, is not elaborated when encountered (see Section 11). for a discussion of generics.

NOTES

Differences between *immediate* and *deferred declarations* are listed below.

Immediate declaration	Deferred declaration
elaborated when encountered	elaborated when invoked
can not have parameters	can have parameters (7.3)
must appear before statements	can appear either before or after statements (3.2)
can not be overloaded	explicit overloading is permitted for all except types (11.2)
can not be generic	can be generic (11.3)
can not have a translation time property list	can have a translation time property list (11.1.2)
are not compound	includes all compound declarations
are not scopes	are closed scopes(3.5)

Deferred declarations and their characteristics are summarized below.

Declaration	Invocation	Effect of Invocation
procedure (7.1)	procedure invocation statement	performs some action
function (7.2)	function invocation primary	produces a value
task (10.1)	task invocation statement	an activation of the task is elaborated concurrently
capsule (8.1)	capsule invocation declaration	makes exported definitions visible
abbreviation (4.4.1)	abbreviation invocation	produces an abbreviated type or subtype
type (4.4.2)	user-defined subtype	produces a subtype of a user-defined type

EXAMPLES

1) Immediate declarations.

```
VAR flag : BOOL := TRUE;
CONST pi := 3.14159;
EXCEPTION stack_underflow, stack_overflow;
EXPOSE ALL FROM compool;
```

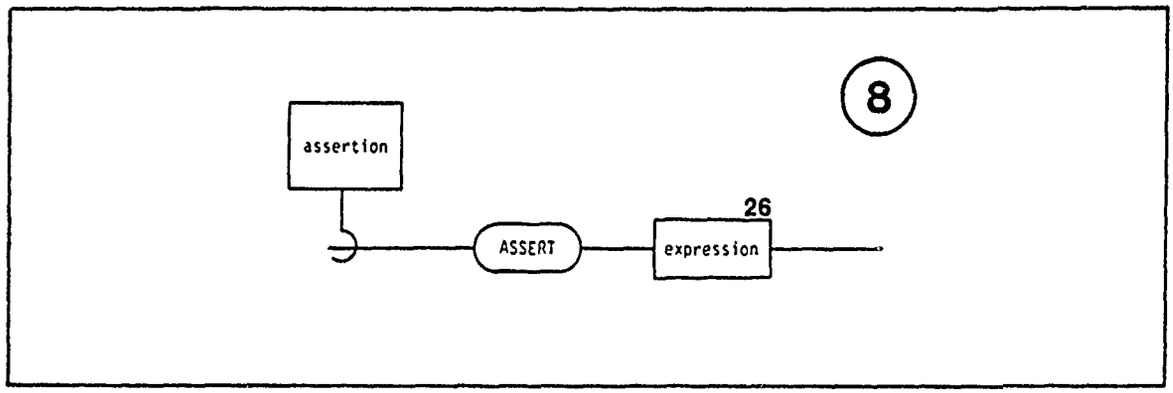
2) Deferred declarations.

```
ABBREV max_int : INT(min..max);

TYPE complex (n,m : FLOAT) : RECORD[r,i : FLOAT(10, n..m)];

PROC complex_complement (VAR x : complex);
   x.i := -x.i;
END PROC complex_complement;

FUNC even (i : INT) => BOOL;
   RETURN (i MOD 2) = 0;
END FUNC even;

TASK reader;
   ...
END TASK reader;

CAPSULE compool EXPORTS ALL;
   % variable declarations
   ...
END CAPSULE compool;
```

3) Generic declaration.

```
GENERIC t : TYPE NEEDS := (t,t);        % needs assignment
                                        %  operator
   % swap for any type
   PROC swap(VAR a,b : t);
      CONST c := a;
      a := b;
      b := c;
   END PROC swap;
```

3.4 ASSERTIONS



An *assertion* specifies a condition which will be true when the *assertion* is elaborated. *Assertions* are used to make programs easier to read and maintain, to provide information useful to an optimizing compiler, and to provide checkpoints for formal and informal verification of correctness.

RULES

The *expression* must have type BOOL. Elaboration of an *assertion* consists of testing the value of the *expression* and, if the *expression* is false, raising the X_ASSERT exception.

NOTES

An *assertion* does not necessarily imply runtime checking. If the condition can be checked during translation, then object code need not be generated for it. If an *assertion* is known to be false at translation time, a warning is issued. A *pragmat* is available for suppressing the X_ASSERT exception (see Appendix B).

EXAMPLES

```
FUNC sqrt (a : FLOAT) => FLOAT;
   ASSERT a > 0.0;
   ...
END FUNC sqrt;

PROC full_divide (a,b : INT, OUT c,d : INT);

   ASSERT a>=0 AND b>0;

   VAR x : INT(0..a) := 0;
   VAR y : INT(0..a) := a;

   ASSERT b*x+y = a;

   WHILE y >= b REPEAT
      ASSERT b*x+y = a;
      x := x+1;
      y := y-b;
   END REPEAT;

   ASSERT (b*x+y = a) AND  y<b;

   c := x ;
   d := y ;

   ASSERT (b*c+d = a) AND 0<=d AND  d<b;

END PROC full_divide;
```

3.5 NAMES AND SCOPES

A name is either an *identifier* or a *definable symbol* (see Section 13.2). *Definable symbols* are used only to refer to built-in operations or to functions and procedures that provide additional definitions for built-in operations.

Every use of a name must have a corresponding definition; definitions of names are never created by default. There are several forms of definition, including *declarations*, *formal parameters*, and goto labels. A name may have more than one definition; when this occurs, it must be possible to associate each use with the appropriate definition. The association is governed by the scoping rules.

A scope is a syntactic form in which names may be defined. An open scope is a scope in which all definitions in the enclosing scope are known, provided that those definitions do not conflict with a local definition of the open scope. A closed scope differs from an open scope in that matching identifiers (the target of an *exit statement*) and goto labels (see Section 6) from the enclosing scope are never available and *variables* from the enclosing scope are available only when explicitly listed in an imports list.

RULES

All *deferred declarations* (see Section 3.3) are closed scopes. *Bodies* (see Section 3.2), *compound* statements (see Section 6), and *generic declarations* (see Section 11.3) are open scopes.

Everything in one scope that is outside any scope contained within it, is called local to that scope. For all non-deferred definitions, two definitions are considered to conflict if the same name is associated with both. Conflicting definitions local to the same scope are not permitted. *Deferred* declarations with the same name do not necessarily conflict (see Section 11.1).

The definitions which are known in a scope are:

a) all local definitions; and

b) each definition which is known in the enclosing scope, is available, and does not conflict with a local definition.

For open scopes, all definitions known in the enclosing scope are available. For closed scopes, all definitions known in the enclosing scope are also available, with the following two exceptions:

a) goto labels and matching identifiers; and

b) variable definitions, that are not explicitly imported (see Section 3.7).

Any occurrence of a name other than a defining occurrence is a use of that name. A use of a name which is local to some scope must correspond to a definition which is known in that scope. Each use must correspond to exactly one definition. For names other than names of deferred units, at most one definition of a name will be known in each scope.

NOTES

Definitions which are exposed in a scope (see Section 8.2) are considered to be local definitions of that scope.

Uses of names local to some scope are uniformly associated with definitions; i.e., the definition associated with a particular use is the same, no matter where within the scope that use occurs.

Conflicting definitions of a single name can exist in different scopes without restriction. Conflicting definitions of a single name within a single scope are excluded since this would make it impossible to associate each use uniquely to a single definition.

A definition may be any of the following: a *declaration* (see 3.3), a *formal parameter* (see 7.3), a goto label (see 6), the index of a *repeat statement* (see 6.5), a matching identifier (see 6), a *formal parameter* (see 7.3), a *generic parameter* (see 11.3), or a needed name (see 11.4).

EXAMPLES

```
sample BEGIN                        % definition 1
         VAR a, b : INT(1..10);     % definition 2 and 3
         CONST c := 4;              % definition 4
         ...
         BEGIN
           % begin statement is an open scope
           VAR b : BOOL;            % definition 5

           ...sample...             % refers to definition 1
           ...a...                  % refers to definition 2
           ...b...                  % refers to definition 5
           ...c...                  % refers to definition 4
           ...p...                  % refers to definition 6
           ...x...                  % illegal

         END;
         ...
         PROC p (VAR x : INT);      % definition 6 and 7
           % procedure is a closed scope

           ...sample...             % illegal
           ...a...                  % illegal
           ...b...                  % illegal
           ...c...                  % refers to definition 4
           ...p...                  % refers to definition 6
           ...x...                  % refers to definition 7

         END PROC p;

END sample;
```

3.6 FORWARD REFERENCES TO NAMES

Use of a variable or constant name before it has been created is not permitted.

RULES

No use of a name defined as a *variable* or *constant* can appear before its definition.

A deferred unit is said to require a *variable* or *constant* if it either contains a use of the name of the *variable* or *constant* or contains an invocation of some other deferred unit that requires that *variable* or *constant*. No invocation of a deferred unit that requires a *variable* or *constant* may appear before the definition of that *variable* or *constant*.

NOTES

Forward references to deferred definitions, goto labels, and exceptions are always allowed.

EXAMPLES

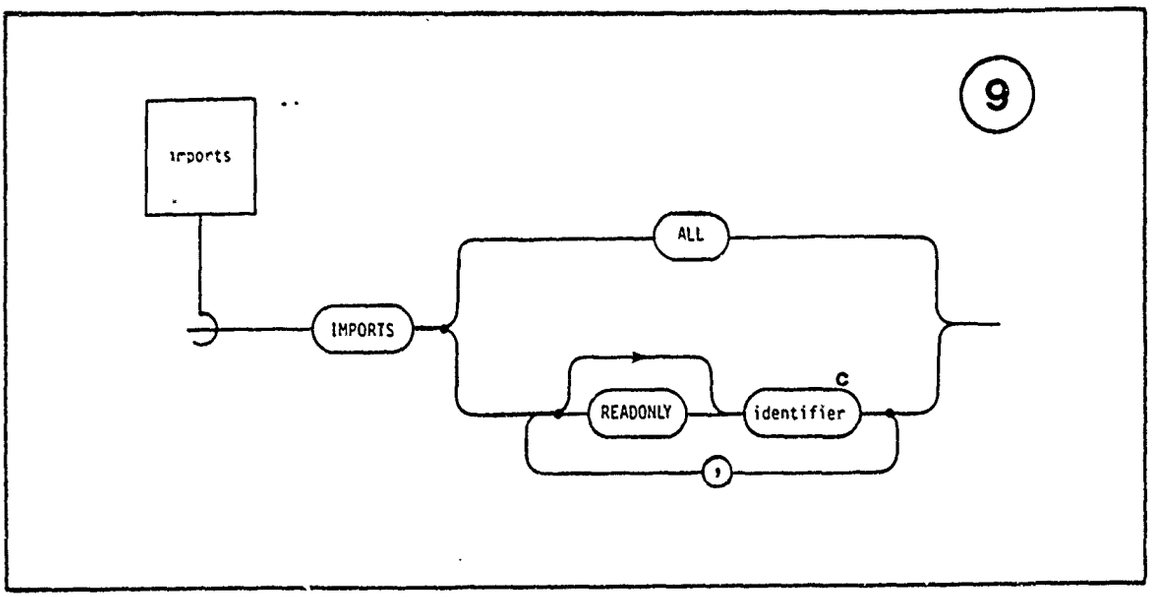
1) Legal forward reference to a deferred declaration.

```
BEGIN
   p;
   ...
   PROC p;
      ...
   END PROC p;
END;
```

2) Correct uses and incorrect forward references to immediate declarations.

```
VAR a : INT(1..10) := b + 3;  % incorrect, b has not been
                              % elaborated
VAR b : INT(1..5)  := 2;      % correct
VAR c : INT(1..20) := b;      % correct, b has been elaborated
```

3.7 IMPORTS LIST



A *compound declaration* (procedure, function, task, or capsule) may include an imports list. The imports list specifies that certain *variables*, known in the enclosing scope, are to be made available in the compound declaration. A *variable* imported into a compound declaration can be restricted so that it cannot be modified inside the compound declaration.

RULES

Every name in the imports list of a compound declaration must be associated with a variable definition (see Section 4.2) known in the immediately enclosing scope.

If ALL is specified, all variable definitions which are known in the enclosing scope are available to the compound declaration (see Section 3.5).

If a list of names is specified, the definitions associated with those names are available to the compound declaration.

If READONLY precedes a name, that variable is treated as a readonly data item within the compound declaration (see Section 4.2).

NOTES

Abbreviation declarations and *type declarations* are *deferred declarations* and, thus, are closed scopes. Neither *declaration* can include an import list so *variables* from the enclosing scope may not be used within either of these declarations.

A closed scope which imports ALL is, essentially, an open scope except that goto labels and matching identifiers are not available.

EXAMPLES

```
sample BEGIN

        VAR a,b : INT(0..10);
        CONST c := 4;
        ...
        PROC p1;
          % known = c, p1; p2, p3
          ...
        END PROC p1;

        PROC p2 IMPORTS a, READONLY b;
          % known = a, b(as readonly), c, p1, p2, p3
          ...
        END PROC p2;

        PROC p3 IMPORTS ALL;
          % known = a, b, c, p1, p2, p3
          ...
        END PROC p3;

END sample;
```

4. TYPES

4.1 TYPES AND SUBTYPES

Types and subtypes are used to specify the properties of data items. These properties control both the values that a data item may have, and the operators, functions, and procedures that may be applied to it.

Data properties may be divided into two groups: type properties, which must be known during translation; and constraint properties, which need not be known during translation, but which must be known when a data item is created. The type of a data item is the collection of all type properties. The subtype of a data item is the collection of all properties, both type and constraint properties. The *subtype* of a data item is said to belong to its *type*. Typically, many *subtypes*, each with a different set of constraint values, will belong to the same type. The two most common constraints on *types* are range constraints (which limit the range of values of data items having the *subtype*) and size constraints (which specify the size of data items having the *subtype*).

EXAMPLES

1) Types

```
INT
FLOAT
STRING[ASCII]
```

2) Subtypes

```
INT(1 .. 10)
FLOAT(5, -100.0 .. 100.0)
STRING[ASCII] (5)
```

4.1.1 USE OF TYPES AND SUBTYPES

A *subtype* must be specified wherever a data item is to be created, such as a *variable declaration*. A *type* or a *subtype* must be specified for each *formal parameter*. Invocation of a deferred unit with *formal parameters* is permitted only if the *type* of each *actual parameter* is the same as the *type* specified (or the *type* to which the specified *subtype* belongs) for the corresponding *formal parameter*. Verifying that *types* are the same is called type checking; since *types* consist only of properties that are known during translation, all type checking is done during translation (see Section 4.1.5).

EXAMPLES

```
VAR i : INT(1..10);
VAR s : STRING[ASCII] (5);
   ...
p(i);
q(i);
p(s); % illegal - type of s is not INT
   ...
PROC p (x : INT);
   ...
END PROC p;

PROC q (x : INT(1..10));
   ...
END PROC q;
```

4.1.2 SPECIFYING TYPES AND SUBTYPES

For some *types*, the only type property is their name. Examples are:

```
BOOL
INT
FLOAT
```

Other *types* require additional properties. These properties, for all *types* but arrays, are given in a comma-separated list enclosed in square brackets, called the type property list. For example,

```
ENUM['a, 'b, 'c]   % values of an enumeration type are always
                   %  known at translation time
```

is an enumeration type for the values 'a, 'b, and 'c .

When no additional constraints are needed on a type at the time of data creation, the subtype specification of the data looks the same as the type specification of the data. For example,

```
BOOL
ENUM['a, 'b, 'c]   % a variable of this subtype may take all listed
                   %  values
```

When additional constraints are needed, they are specified in a comma-separated list enclosed in parentheses, called the constraint property list. For example,

```
INT(0..10)                      % a single range constraint
FLOAT(10, 0.0 .. 50.0)          % 2 constraints, precision and range
ENUM['a, 'b, 'c] ('a .. 'b)     % a variable of this subtype may only
                                %  take on the constrained range of
                                %  values
```

In some cases, the properties of a *type* may themselves include *types*. For example,

```
RECORD[a : INT, b : BOOL]
STRING[ENUM['a, 'b, 'c]]    % the characters of which a STRING is
                            %  composed must be known at
                            %  translation time
```

This nesting allows *types* to be constructed based upon other *types*. For example,

```
RECORD[a : INT,
       b : UNION[w : BOOL,
                 x,y : RECORD[m,n : FLOAT],
                 z : ENUM['red, 'green]],
       c : ENUM['yes, 'no, 'maybe]]
```

Subtype constraints are specified by placing constraint property lists after the *type* and any *types* contained within the *type*. For example,

```
RECORD[a : INT(-5..5), b : BOOL]
STRING[ENUM['a, 'b, 'c] ('a .. 'b)] (10)   % this is a string
                                           %  of length 10 made up
                                           %  only of A's and B's
```

Array types and array subtypes are written in a special form. For example,

```
ARRAY INT OF FLOAT                  % a one-dimensional array type.
ARRAY INT, ENUM['a, 'b, 'c]         % a two-dimensional array type
                                    %  where the first dimension
      OF FLOAT                      %  is subscripted by integers
                                    %  and the second by enum
                                    %  literals.
ARRAY INT(1..10) OF                 % a one-dimensional array
      FLOAT(10, -5.0 ..5.0)         %  subtype.
ARRAY INT(1..10), ENUM['a, 'b, 'c]  % a two-dimensional array
      OF FLOAT(10, -5.0 .. 5.0)     %  subtype where the first
                                    %  dimension has size 10 and
                                    %  is indexed by integers
                                    %  and the second has size 3
                                    %  and is indexed by enum
                                    %  literals (e.g, x(5, 'b))
```

RULES

When a specification which could be either a type or a subtype (e.g., BOOL or ENUM['a, 'b]) is used in a context where either a type or a subtype is permitted (e.g., for a formal parameter), the specification is interpreted as a *type*.

EXAMPLES

```
VAR x : ENUM['a, 'b];                    % ENUM['a, 'b] is a subtype

PROC p(x : ENUM['a, 'b]);                % ENUM['a, 'b] is a type
  ...
END PROC p;

PROC q(y : STRING[ENUM['a, 'b]] (5));% ENUM['a, 'b] is a subtype
  ...
END PROC q;
```

4.1.3 RELATIONSHIP BETWEEN TYPES AND SUBTYPES

Given a *subtype*, the *type* to which it belongs can be found by deleting all constraint property lists. For example,

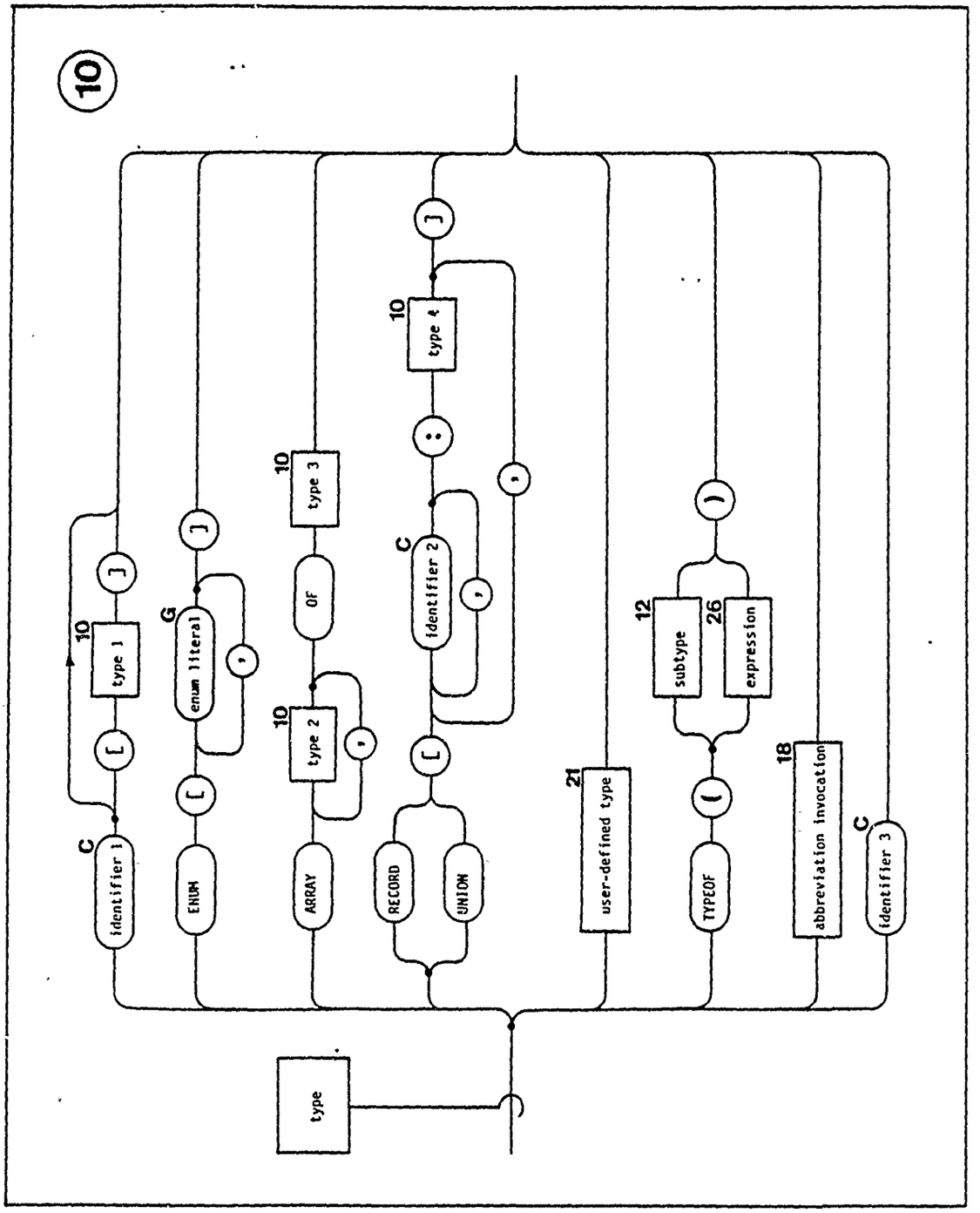
Subtype	Type
BOOL	BOOL
INT(0..10)	INT
ENUM['a, 'b, 'c] ('a .. 'b)	ENUM['a, 'b, 'c]
RECORD[a : INT(-5 .. 5), b : BOOL]	RECORD[a : INT, b : BOOL]
STRING[ENUM['a, 'b, 'c] ('a .. 'b)] (10)	STRING[ENUM['a, 'b, 'c]]
ARRAY INT(1..10), ENUM['a, 'b, 'c] OF FLOAT(10,0.0..100.0)	ARRAY INT, ENUM['a, 'b, 'c] OF FLOAT

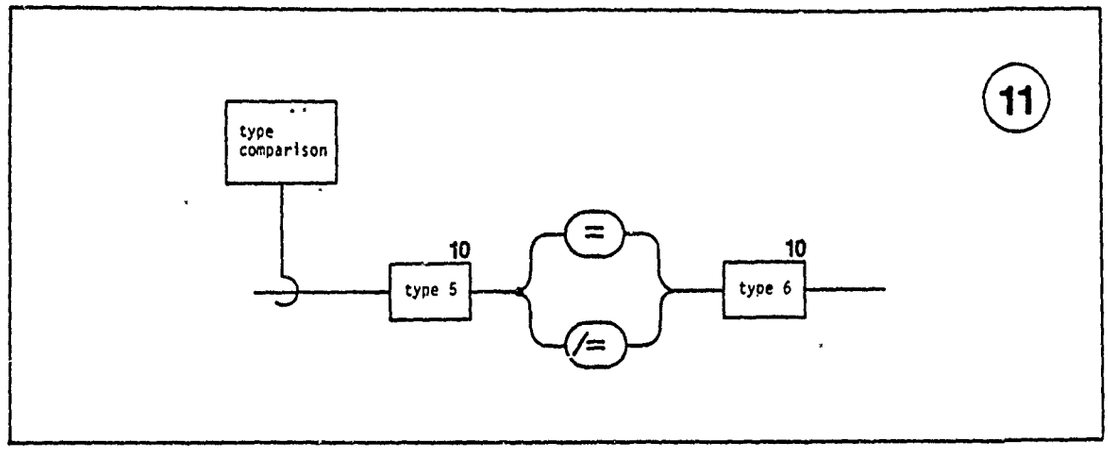
4.1.4 LANGUAGE-DEFINED AND USER-DEFINED TYPES

The language defines a flexible and useful set of *types*. These *types* are summarized in Section 4.3. Detailed rules are given in Appendix C.

In some cases, type and subtype specifications will be quite long. For this reason, a convenient abbreviation facility is provided (see Section 4.4.1). Users can also define the abstract types which are specifically needed for their applications. This capability is provided by the *type declaration* (see Section 4.4.2) together with the *capsule declaration* (see Chapter 8).

4.1.5 TYPES, TYPE EQUIVALENCE AND TYPE COMPARISON





Type equivalence rules are used to determine if two *types* are the same. Type checking is the comparison of two types using the type equivalence rules. *Types* are checked implicitly for assignment (see Section 6.1) and for invocation of deferred units (see Section 3.3). *Types* are checked explicitly for *type comparisons*. All type checking occurs at translation time.

RULES

Types

Identifier 1 must be the name of a built-in type. The *abbreviation invocation* must produce a *type*. *Identifier 3* must be associated with a *generic parameter* that has a *type generic constraint* (see Section 11.3.1).

Type Comparison

The result of elaborating a *type comparison* is a boolean which is true if *type 5* is the same as *type 6*.

To determine if two *types* are the same, the *types* are first expanded and then compared.

A *type* is expanded in the following cases:

a) If the *type* contains any *abbreviation invocations* (see Section 4.4.1), each *abbreviation* invocation is replaced by the type which is the result of the invocation.

b) For record or union types, any components of the form

```
comp1, comp2, ..., compi : type
```

are replaced by

```
comp1 : type,
comp2 : type,
...
compi : type
```

c) Any TYPEOF forms are replaced by their result type (see Section 4.5.2).

d) Any references to type generic parameters are replaced by their replacement elements (see

Section 11.3.1).

Two expanded types are the same if they meet the following requirements when compared:

a) corresponding type identifiers within the expanded types refer to the same definition;

b) the values of corresponding properties in type property lists are the same;

c) for ARRAY types, the corresponding index and component types are the same; and

d) for RECORD and UNION types, the component names are the same and occur in the same order.

NOTES

Built-in types are described in Section 4.3 TYPEOF is described in Section 4.5.2.

Type checking is simplified by the fact that all indirect types are new types and, thus, not expanded into their underlying types. This means that type expansion does not result in cycles.

EXAMPLES

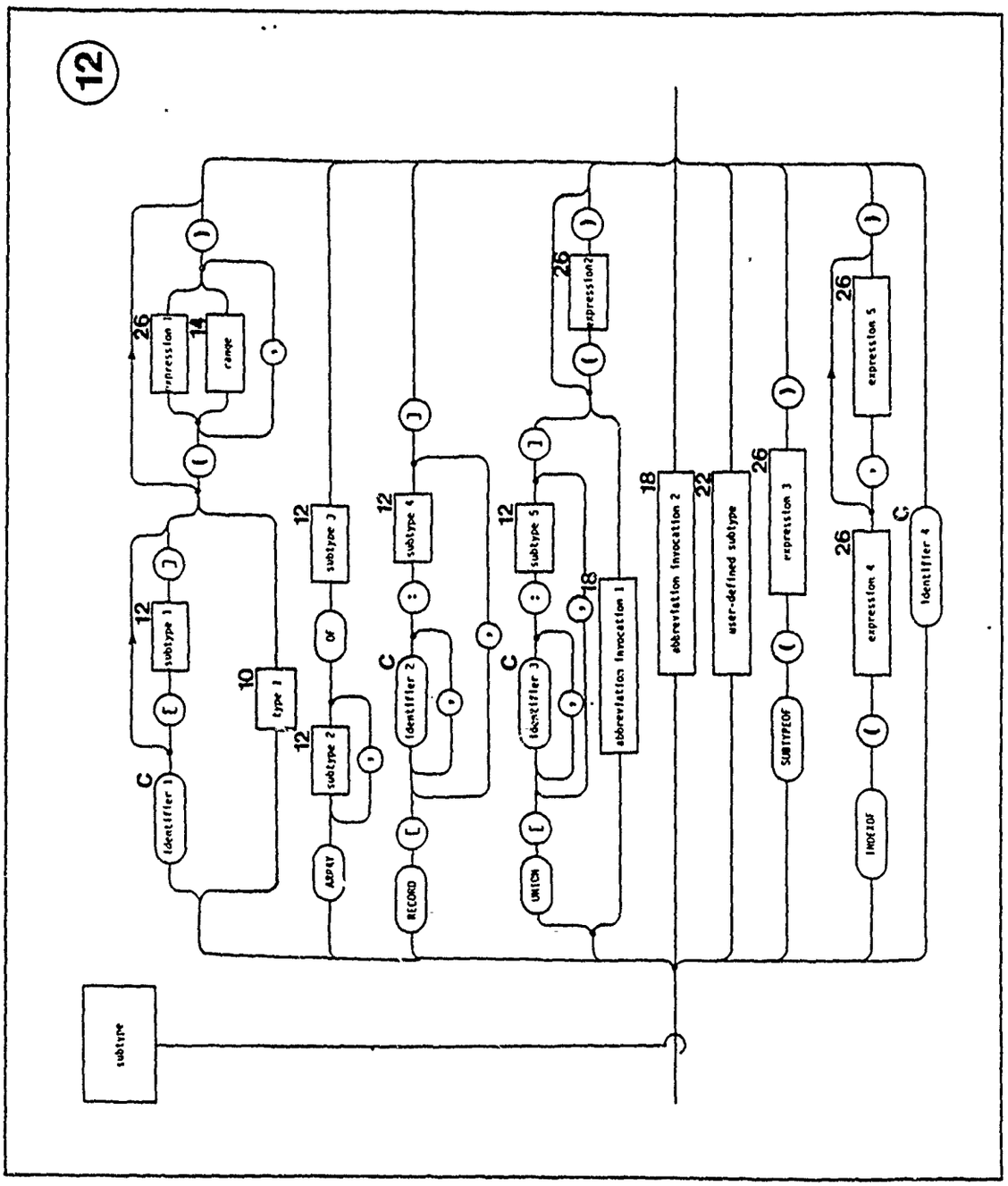
1) Expanding abbreviations

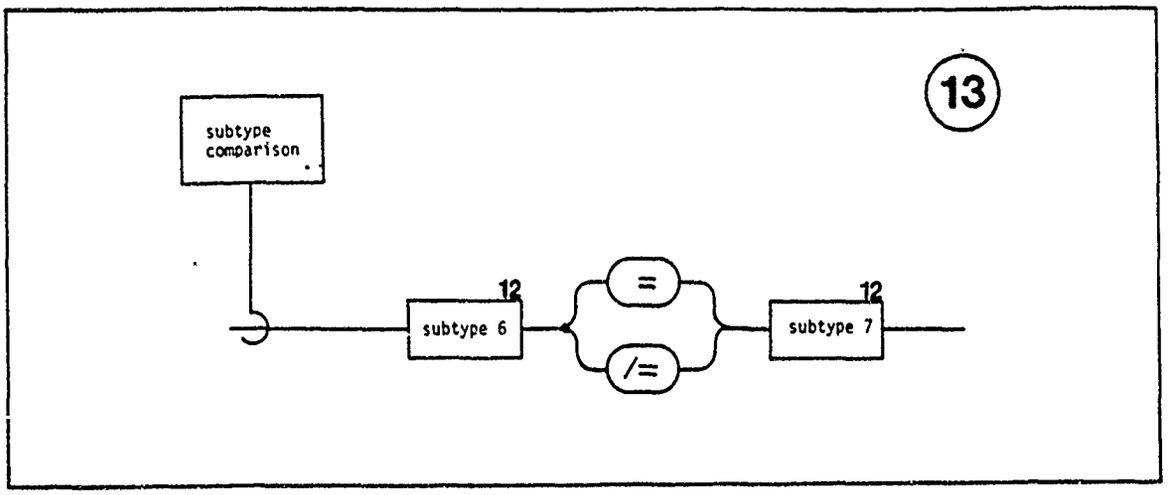
```
ABBREV a : BOOL;
TYPE   b : BOOL;

VAR m : a;
VAR n : b;
VAR o : BOOL;     % the types of m and o are the same but both
                  %  are different from the type of n
```

2) Expanding components

```
VAR a : RECORD[x,y : INT(1..10)];
VAR b : RECORD[x : INT(5..8),
               y : INT(12..15)];    % the types of a and b are
                                    %  the same since round
                                    %  bracketed information
                                    %  is removed to obtain
                                    %  the type.
VAR c : RECORD[y,x : INT(1..10)];   % the type of c is not the
                                    %  same as the type of a,
                                    %  since component names
                                    %  are in a different
                                    %  order.
```

4.1.6 SUBTYPES, SUBTYPE EQUIVALENCE AND SUBTYPE COMPARISON





Subtype equivalence rules are used to determine if two *subtypes* are the same. Two *subtypes* can be explicitly compared for equality using a *subtype comparison*.

RULES

Subtypes

Identifier 1 must be the name of a built-in type. Abbreviation invocation 1 must produce a UNION subtype with no subtype constraint. Abbreviation invocation 2 must produce a subtype. Identifier 4 must be associated with a generic parameter that has a subtype generic constraint (see 11.3.2).

Subtype Comparison

The result of elaborating a *subtype comparison* is a boolean which is true if *subtype 6* is the same as *subtype 7*.

Two *subtypes* are the same if their *types* are the same and if the values of the corresponding constraints in the constraint property lists are equal. *Subtype comparison* is only permitted for *subtypes* whose constraints have *types* for which = is defined.

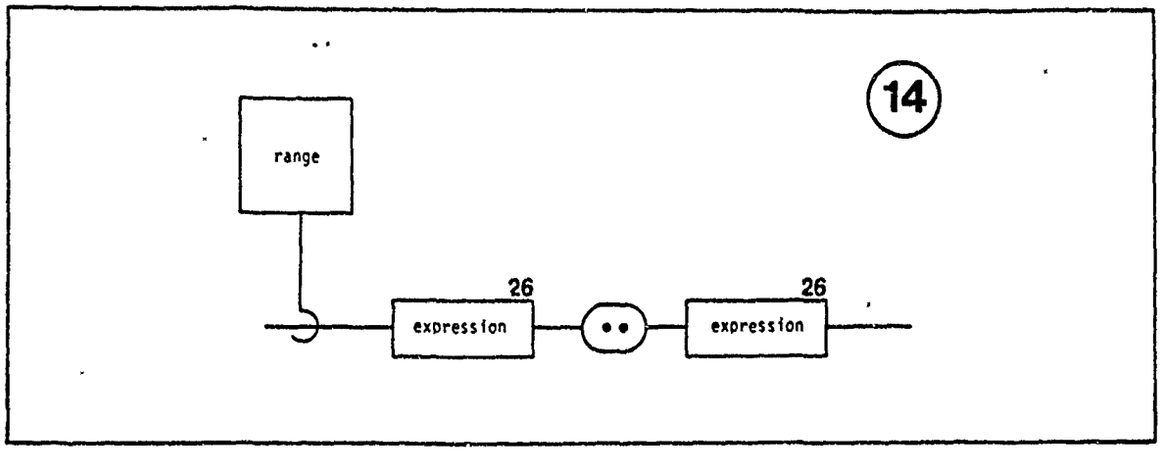
NOTES

Built-in types are described in Section 4.3. SUBTYPEOF and INDEXOF are described in Section 4.5.1.

Subtypes are compared for equality at translation time whenever possible, otherwise, comparison will be performed at runtime.

Two *subtypes* are implicitly compared for equality when an *actual parameter* is compared to a *formal parameter*, specified by *subtype*, and bound by VAR or READONLY (see Section 7.3). If the comparison produces the value false during implicit comparisons, the X_SUBTYPE exception is raised.

4.1.7 RANGE



A *range* represents a contiguous sequence of values of some *type*.

RULES

Expression 1 and *expression 2* must have the same *type*.

If *expression 1* is less than *expression 2*, the *range* represents all successive values beginning with the lowest value (*expression 1*) and ending with the highest value (*expression 2*). If *expression 1* and *expression 2* have the same value, the *range* represents only that value. If *expression 2* is less than *expression 1*, the *range* represents no values.

NOTES

Ranges with no values are useful when defining index variables (see Section 6.5), since they allow zero elaborations of a repeat statement body, and when defining arrays (see Section 4.3, Appendix C.7), since they allow empty arrays. *Ranges* are used for constraint properties of integer, enumeration, and floating point subtypes, for *case statement* value labels (see Section 6.4), and for slicing (see Section 5.3).

EXAMPLES

```
VAR w : ENUM[ 'one, 'two, 'three, 'four, 'five,
              'six, 'seven, 'eight, 'nine, 'ten] ('one .. 'five);
VAR x,y : INT(1..10) := 10;
VAR z : FLOAT(10,1.0 .. 100.0);
VAR a : ARRAY INT(1..x) OF BOOL;
...
FOR i : INT(x..y) REPEAT
   ...
END REPEAT;
CASE x
   WHEN 1.. 5 => ...
   WHEN 6..10 => ...
END CASE;

...a(x..y)...
```

4.2 VARIABLES AND CONSTANTS

Each *variable* or *constant* has a *subtype* which is known when it is created and does not change during its lifetime. There are two kinds of variables, defined variables and dynamic variables. The value of a *variable* may be both accessed and modified. The value of a *constant* may be accessed but not directly modified. At creation, all *constants* must be initialized to some value and each *variable* is either initialized to some value or is uninitialized. *Variables* and *constants* are data items are further described in Section 5.1.

NOTES

Defined *variables* are specified in the following ways:

a) a *variable declaration* (declared variable) (see Section 4.2.1)

b) an OUT *formal parameter* (parameter variable) (see Section 7.3)

c) a VAR *formal parameter* (parameter variable) (see Section 7.3)

d) a *repeat statement* with a for phrase (index variable) (see Section 6.5)

Definitions of declared variables, OUT parameter variables, and index variables create new variables. Definitions of VAR parameter variables associate new names with existing *variables*.

There are also dynamic variables which are not defined but rather created by elaboration of the ALLOC statement (see Section 4.4.3)

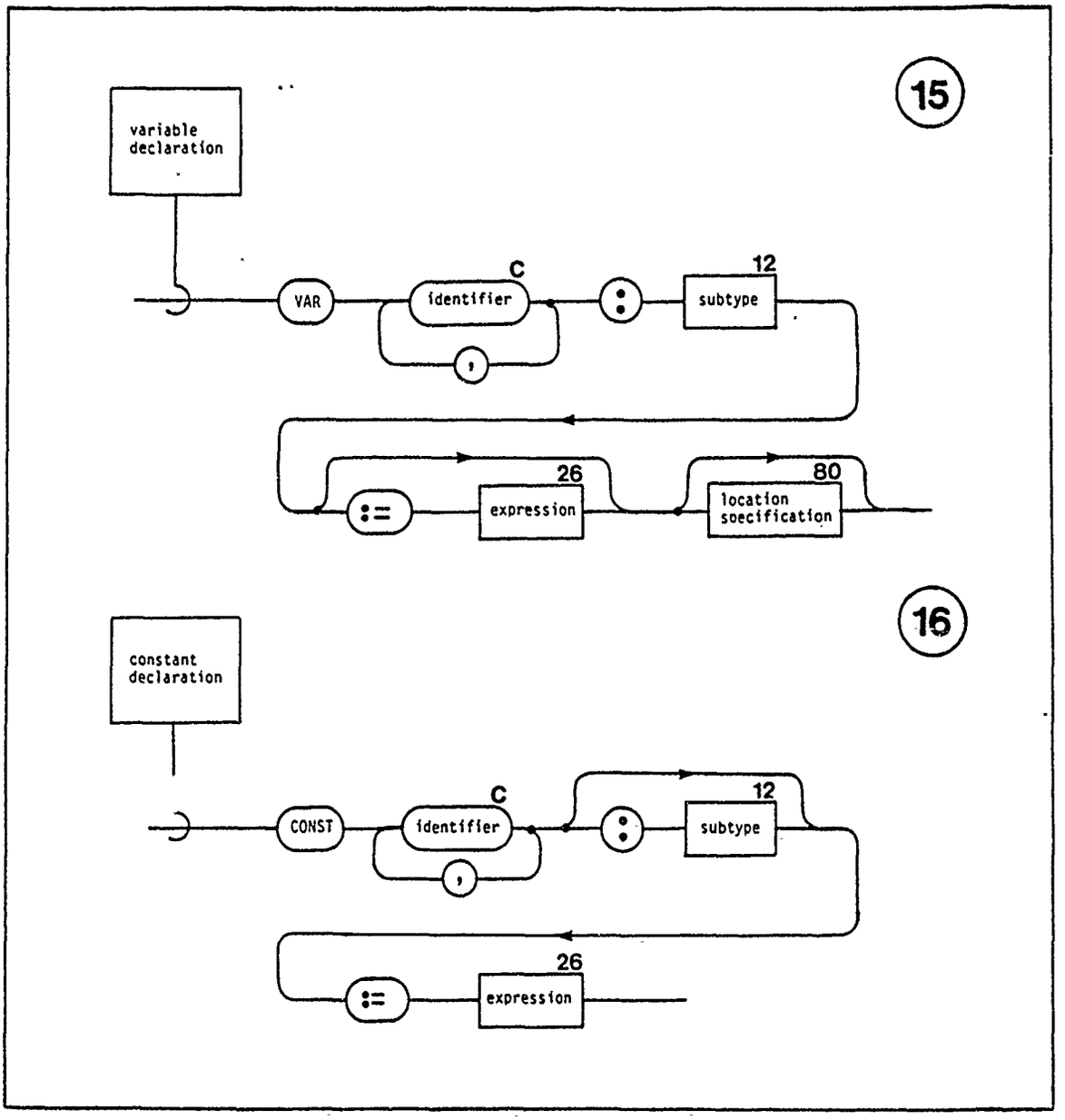
Constants are defined in the following ways:

a) a *constant declaration* (declared constant) (see Section 4.2.1)

b) a CONST *formal parameter* (parameter constant) (see Section 7.2)

Definitions of declared constants and parameter constants create new constants.

4.2.1 VARIABLE OR CONSTANT DECLARATION



These *declarations* define *variables* and *constants.*

RULES

Each *identifier* is defined as a *variable* (VAR declaration) or a *constant* (CONST declaration) in the scope in which the *declaration* is local.

If no *subtype* is specified for a *constant*, the *subtype* of the *constant* becomes the *subtype* of the initialization *expression.* Initialization is performed by assignment (:=).

Elaboration of a *variable declaration* results in the creation of a *variable.* If an initialization phrase is present, the *variable* is initialized to the value given by elaborating the *expression* in the initialization phrase. Elaboration of a *constant declaration* results in the creation and initialization of a *constant.*

NOTES

When initialization is specified, the *type* of the *variable* or *constant* must be assignable (see Section 4.3). The X_INIT exception is raised when there is an attempt to access the value of an uninitialized variable.

Some *variables* are automatically initialized, even when initialization is not explicitly specified. These are *variables* of indirect types and ACT, MAILBOX, DATA_LOCK, and FILE types. Automatic initialization can also be established for user-defined types (see Section 13.3).

Location specifications are used for machine-dependent representations and are described in Section 12.3.

EXAMPLES

```
VAR x : INT(1..100) := 5;
CONST y := TRUE;

CONST z : INT(0..10) := 10;
VAR a : FLOAT(10,1.0 .. 100.0);
```

4.3 OVERVIEW OF BUILT-IN TYPES

This section gives a brief overview of the *types* which are built into the language, along with their *subtypes*, procedures, and functions. A more detailed presentation of each of the *types* is given in Appendix C.

If a value can be assigned to a *variable* (i.e., if the := procedure is defined), the *type* of the *variable* is said to be assignable. Note that assignment is used in the following cases:

a) *assignment* statements (see Section 6.1);
b) initialization (see Section 4.2.1);
c) CONST formal parameters (see Section 7.3); and
d) OUT formal parameters (see Section 7.3).

The relational operators are =, /=, <, >, <=, >= (see Section 5.2).

Boolean

Type: BOOL
Subtype: BOOL
Unary: NOT
Binary: AND, OR, XOR, =, /=
Assignable: yes
Literals: see Section 2.3.9

Integer

Type: INT
Subtype: INT(min..max)
Unary: +, -
Binary: +, -, *, DIV, **, MOD, relational operators
Function: ABS, SUCC, PRED
Assignable: yes
Literals: see Section 2.3.6

Floating Point

Type: FLOAT
Subtype: FLOAT(precision, min..max)
Unary: +, -
Binary: +, -, *, /, **, relational operators
Function: ABS, FLOOR
Assignable: yes
Literals: see Section 2.3.6

Precision is the minimum number of decimal digits to be represented.

Enumeration

Type: ENUM[enum-literal1,enum-literal2,...,enum-literaln]
Subtype: ENUM[enum-literal1,enum-literal2,...,enum-literaln]
or
ENUM[enum-literal1,enum-literal2,...,enum-literaln](min..max)
Binary: relational operators, &
Function: SUCC, PRED, POS
Assignable: yes
Literals: see Section 2.3.7

Enumeration values are ordered as they appear in the type property list, with the leftmost being lowest. A range constraint in an enumeration subtype restricts values from the set of all possible values (in the *type*) to the set of legal values for this *subtype*.

Record

Type: RECORD[comp1:type1,comp2:type2,...,compn:typen]
Subtype: RECORD[comp1:subtype1, comp2:subtype2,...,compn:subtypen]
Binary: =, /=
Component Selection: record_var.comp
Assignable: yes, if all components are assignable
Constructor: see Section 5.6

Successive components having the same *type* can also be written as

comp1, comp2,...,compn:typej

Union

Type: UNION[comp1:type1, comp2:type2,...,compn:typen]
Subtype: UNION[comp1:subtype1, comp2:subtype2,...,compn:subtypen]
or
UNION[comp1:subtype1, comp2:subtype2,...,compn:subtypen]
(exp)
Binary: =, /=
Component Selection: union_var.comp
Tag Inquiry: union_var.TAG
Assignable: yes, if all components are assignable
Constructor: see Section 5.6

A union type consists of multiple components, only one of which may be accessed at any point in the lifetime of a union *variable* of this type. If a subtype constraint is present, *variables* with that *subtype* can have only the component whose name is specified in the subtype constraint as an enumeration value. For example,

```
UNION[a : INT(0..10), b : BOOL] ('b);
```

The tag inquiry returns the name (as an enumeration value) of the component currently accessible. The component which is present in a union may change over the lifetime of the union *variable*. Successive union components having the same *type* can also be written as

```
comp1,comp2,...compn:typej.
```

Array

Type: ARRAY dim-type1, dim-type2,...,dim-typen OF comp-type
Subtype: ARRAY dim-subtype1, dim-subtype2,...dim-subtypen OF comp-subtype
Binary: & (concatenation for one-dimensional array), =, /=
Component Selection: array_var(position1, position2,...,positionn)
array_var(min..max) (slicing for one-dimensional array)
Assignable: yes, if component type is assignable
Constructor: see Section 5.6

The dimensions must be integer or enumeration types or subtypes.

Set

Type: SET[type]
Subtype: SET[subtype]
Unary: NOT (complement)
Binary: AND (intersection), OR (union),
XOR (symmetric difference), IN (membership),
relational operations (subset relations)
Assignable: yes
Constructor: see Section 5.6

The type contained in the type property list can only be INT or an enumeration type.

String

Type: STRING[type]
Subtype: STRING[subtype] (length)
Binary: & (concatenation), relational operations
Component Selection: string_var(position)
string_var(min..max)
Assignable: yes
Literals: see Section 2.3.8

The *type* contained in the type property list must be an enumeration type.

Fixed Point

The first quarterly review of the DoD common language effort has emphasized the lack of a consensus on the military's requirement for a fixed point facility. The fixed point described in Steelman is actually a scaled integer facility. It seems that more discussion is necessary before a decision is reached by the military on the fixed point facility necessary for most applications. When this determination is reached, the design can be built into this language. Some of the possible design alternatives for fixed point are discussed in the companion Justification document.

NOTES

There are, besides the basic *types* discussed above, some other built-in types designed for special purposes. The MAILBOX, DATA_LOCK, and ACT types are described in Chapter 10. The LATCH type is described in Appendix C.13. FILE types are described in Appendix C.14. Pointers are not a language type but are instead provided via the indirect form of the *type declaration* (see Section 4.4.3).

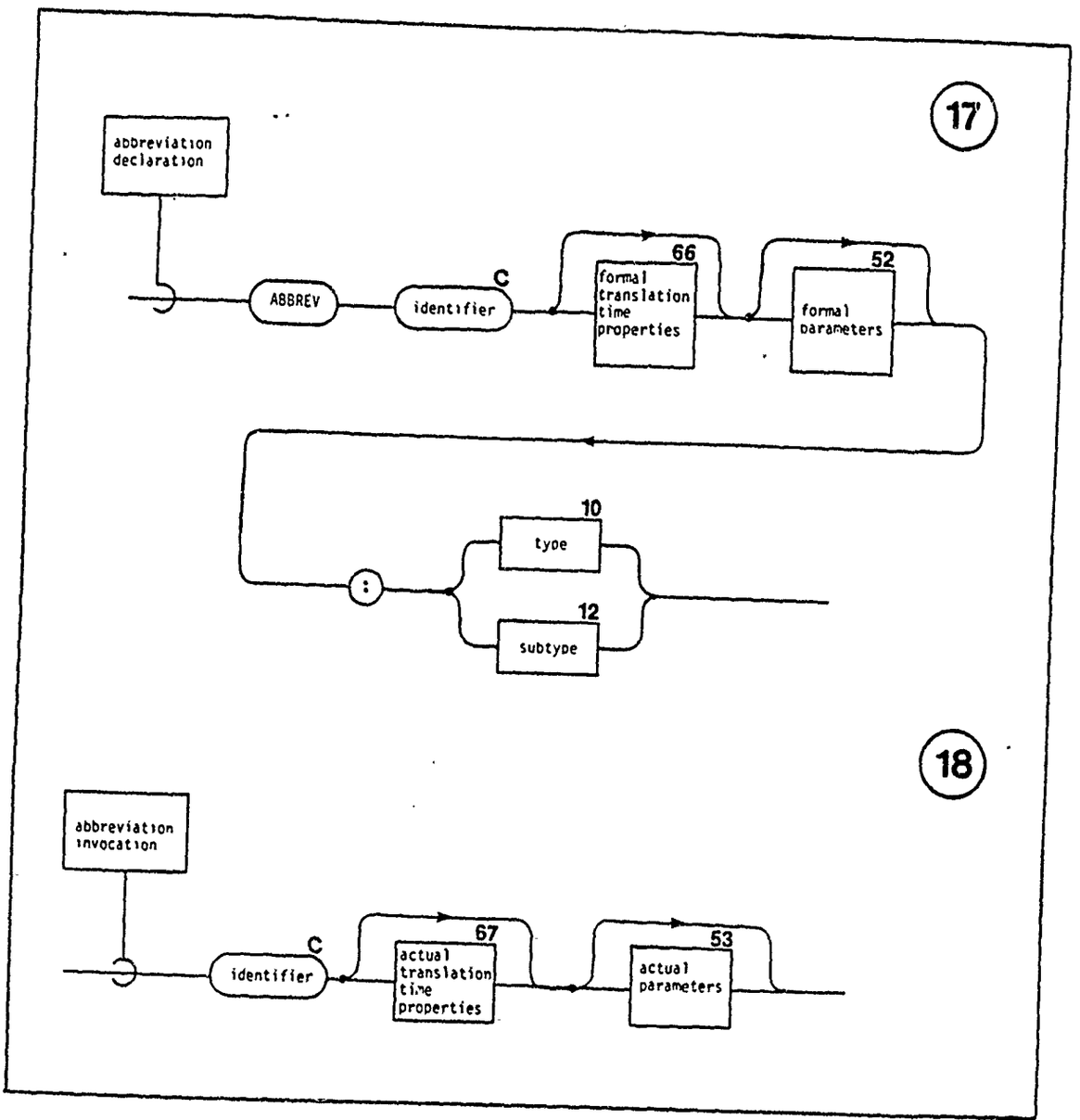
4.4 DECLARATION OF SUBTYPES AND TYPES

Two kinds of *declarations* can be used for *subtypes* and *types*, the *abbreviation declaration* and the *type declaration*. Both are *deferred declarations*.

An *abbreviation declaration* defines an abbreviation. Invocation of an abbreviation produces the *type* or *subtype* specified in the declaration of the abbreviation. An abbreviation is particularly useful when a *type* or *subtype* with a long specification is needed in several places in a program. As with all deferred units, an abbreviation can be parameterized. This permits a single abbreviation to be used to abbreviate a set of related *subtypes*.

A *type declaration* defines a new type distinct from all other *types*. The user can create an abstract data type by placing the *type declaration* within a *capsule declaration*, together with a set of procedures and functions which operate on *actual parameters* of the defined type. Since a *type* is a deferred unit, it may be parameterized (parameters are used to specify the constraint property list of *subtypes* of the new type) and may be generic (the translation time property list serves as the type property list). There are two basic forms of the *type declaration*: a *direct type declaration* and an *indirect type declaration*. *Variables* and constants having an indirect type can be used to reference dynamically allocated variables.

4.4.1 ABBREVIATION



When a long type or subtype specification is needed in several places in a program, an abbreviation for that *type* or *subtype* can be defined using the *abbreviation declaration.* Use of abbreviations can contribute to program reliability by ensuring that all uses of the abbreviation produce the same *type* or *subtype.* If the specification is changed, then only the abbreviation need be modified.

RULES

The identifier in an *abbreviation declaration* is defined to be an abbreviation in the scope in which its *declaration* appears. ..

Only CONST *formal parameters* may be used in an *abbreviation declaration.* An *abbreviation* declaration which abbreviates a *type* may not have any *formal parameters.*

Elaboration of an *abbreviation invocation* consists of elaborating the *actual parameters*, binding the *actual parameters* to the *formal parameters* of the named subtype abbreviation (see Section 7.3), and elaborating the *type* or *subtype* in the *abbreviation declaration.* The result of the invocation is the elaborated *type* or *subtype.* If the specification following the : could be a *type* or a *subtype*, then the result of an invocation can be used as either a *type* or a *subtype.* Abbreviations are assumed to be normal (see Section 7.2.1).

NOTES

An *abbreviation declaration* is a closed scope The *formal parameters* are defined in this scope. Since an *abbreviation declaration* cannot have an imports list, no variable names may be used within the *subtype.*

The only legal use of the abbreviation is in an *abbreviation invocation.* If the abbreviation is parameterized, the *identifier* may not be used without parameters as a *type.*

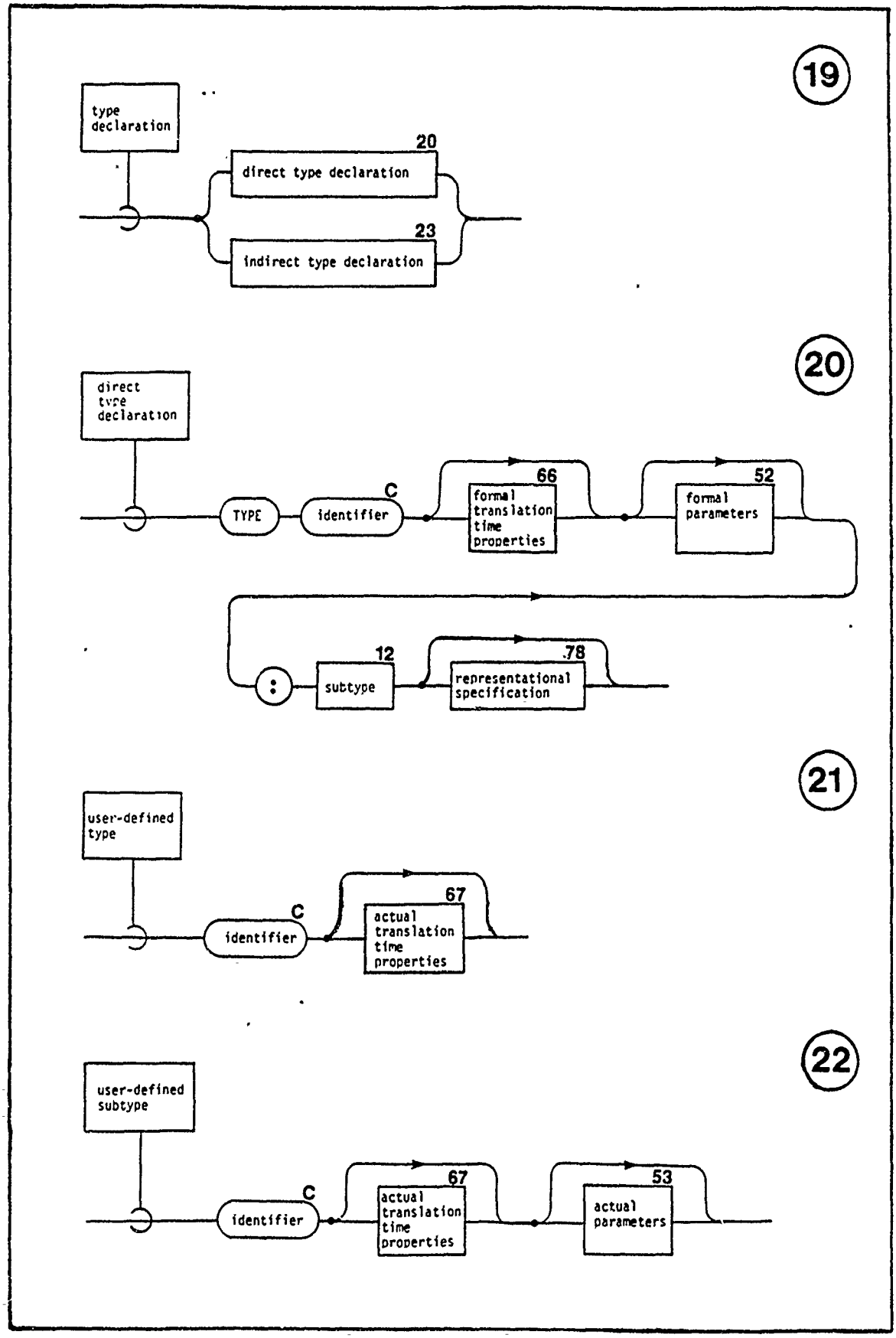
Recursive cycles involving only abbreviations are illegal since the resulting specification would be infinite.

Abbreviations can be overloaded (see Section 11.2) and can be generic (see Section 11.3).

EXAMPLES

```
ABBREV I10 : INT(1..10); % This is equivalent to
VAR y : I10;             % VAR y : INT(1..10);

ABBREV flags(n : INT) :
       ARRAY INT(1..n) OF BOOL; % This is equivalent to
VAR b : flags(10);              % VAR b : ARRAY INT(1..10) OF BOOL;

ABBREV ASCII : ENUM[...];     % see Appendix C for a complete
                              %  definition
VAR x : ASCII;                % ASCII is a subtype here
PROC p(x : STRING[ASCII]);    % ASCII is a type here
   ...
END PROC p;
```

4.4.2 DECLARING AND USING A NEW TYPE



A *type declaration* defines a new type, distinct from all other *types*. For example, the *declaration*

```
TYPE bits(n : INT) : ARRAY INT(1..n) OF BOOL;
```

defines a new type

```
bits
```

Since the *type declaration* is a *deferred declaration*, it defines a deferred unit (a *type*) which can be invoked. Invocation of a *type* produces a *subtype* of that *type*. For example,

```
bits(5)
```

is a *subtype* of the *type* bits. The *actual parameter* list forms the constraint property list of the *subtype*. For example, for the *subtype* bits(5), the constraint property list is (5). The *formal* parameters of a *type* are also used to define the attributes of *subtypes* of that *type* (see Section 4.5.3).

Each *subtype* of a *type* is defined in terms of the *subtype* specified in the *type declaration*, which is called the underlying subtype. For example, the underlying subtype of bits(5) is

```
ARRAY INT(1..5) OF BOOL
```

Each underlying subtype of a new type will belong to a *type* called the underlying type. For example, the underlying type of bits is ARRAY INT OF BOOL. Each *variable* (or *constant*) of some *user-defined subtype* has a component variable (or constant) called the underlying variable (or underlying constant) of the underlying subtype. The standard component selector .ALL is used to access the underlying variable or constant explicitly.

Several operations are automatically defined for each new type: access to the underlying variable or constant; access to components (if any), equality, and assignment. No other operations are automatically defined for a new type. The essential operation on which all other operations are based is .ALL qualification. Given a variable (or constant) with some new type, .ALL qualification produces the underlying variable (or constant). For example, given

```
VAR a : bits(5);
```

then

```
a.ALL
```

is the underlying variable of a and has subtype ARRAY INT(1..5) OF BOOL.

If the underlying type has components (e.g., is an array, record or union), then a component selector operation is automatically defined for the new type in terms of the component selector operation of the underlying type. For example,

```
a(i)
```

can be written instead of

```
a.ALL(i)
```

If the underlying type is assignable, then assignment is also defined for the new type in terms of

the assignment for the underlying type. For example, given

```
VAR b,c : bits(5);
```

then

```
b := c;
```

can be written instead of

```
b.ALL := c.ALL;
```

If equality is defined for the underlying type, then equality is also defined for the new type in terms of equality for the underlying type. For example,

```
b = c
```

can be written instead of

```
b.ALL = c.ALL
```

A new abstract type can be created by placing a *type declaration* in the *body* of a capsule. Operations for the new abstract type are user-defined by procedures and functions defined in the same capsule. The operations take parameters and/or produce results of the abstract type. These operations are implemented using .ALL or component selection to access the underlying variables and constants. One important property of an abstract data type is that it is possible to change the underlying type without affecting any users of the abstract type. To achieve this, users of the abstract type must be denied access to the underlying variables of the abstract type. This is accomplished by not exporting either .ALL qualification or any of the selector operations that are automatically defined.

RULES

The *identifier* in a *direct type declaration* is defined to be a type name in the scope in which the *type declaration* appears. If there is no formal translation time property list, then this *identifier* is the *type*. If there is a formal translation time property list, then the *types* consist of the type identifier together with an actual translation time property list.

Only CONST formal parameters may be used in a type declaration.

Elaboration of a user-defined subtype consists of elaborating the actual parameters, binding the actual parameters to the formal parameters of the named type (see Section 7.3), and elaborating the subtype. The result of a user-defined subtype is a subtype of the invoked type, whose constraint property list is the actual parameter list of the invocation. The underlying subtype of this result subtype is the elaborated *subtype*.

Each newly defined type has the following operations automatically defined:

a) Assignment (:=) is defined in terms of assignment for the underlying type. If the underlying type is not assignable, the defined type is not assignable.

b) Equality (=) is defined in terms of equality for the underlying type. If equality is not available for the underlying type then it is not available for the defined type.

c) Component selection, if the underlying type has components.

d) .ALL qualification, which allows access to underlying variables and constants.

Types are assumed to be normal (see Section 7.2.1).

NOTES

A new type is invoked to produce a *subtype* of that new type. Unlike an abbreviation, if the *type* is parameterized (and has no type property list), the *identifier* may be used without parameters as a *type*. If no parameter list is present in the *type declaration*, the defined type has only a single *subtype*. If a formal parameter list is present, the defined type has one or more *subtypes*.

A *type declaration* is a closed scope. The formal parameter names are defined in this scope. Since a *type declaration* cannot have an imports list, no variable names may be used within the *subtype*.

Users can define their own assignment (:=) procedure, equality (=) function, and selection functions for new types. A user definition of assignment, equality, or of selection will override the automatically provided assignment (see Section 13), equality (see Section 4.1.5), or selection (see Section 13.4). It is also possible to define initialization and finalization operations which are automatically invoked at the beginning and end (respectively) of the lifetime of a data item having a *user-defined type* (see Section 13.3).

The *type declaration*, when used in a *generic declaration* (see Section 11.3), can be used to create a family of *types*. Any actual translation time property list serves as the type property list.

Representation specifications are used for machine-dependent programs and are described in Section 12.2.

EXAMPLES

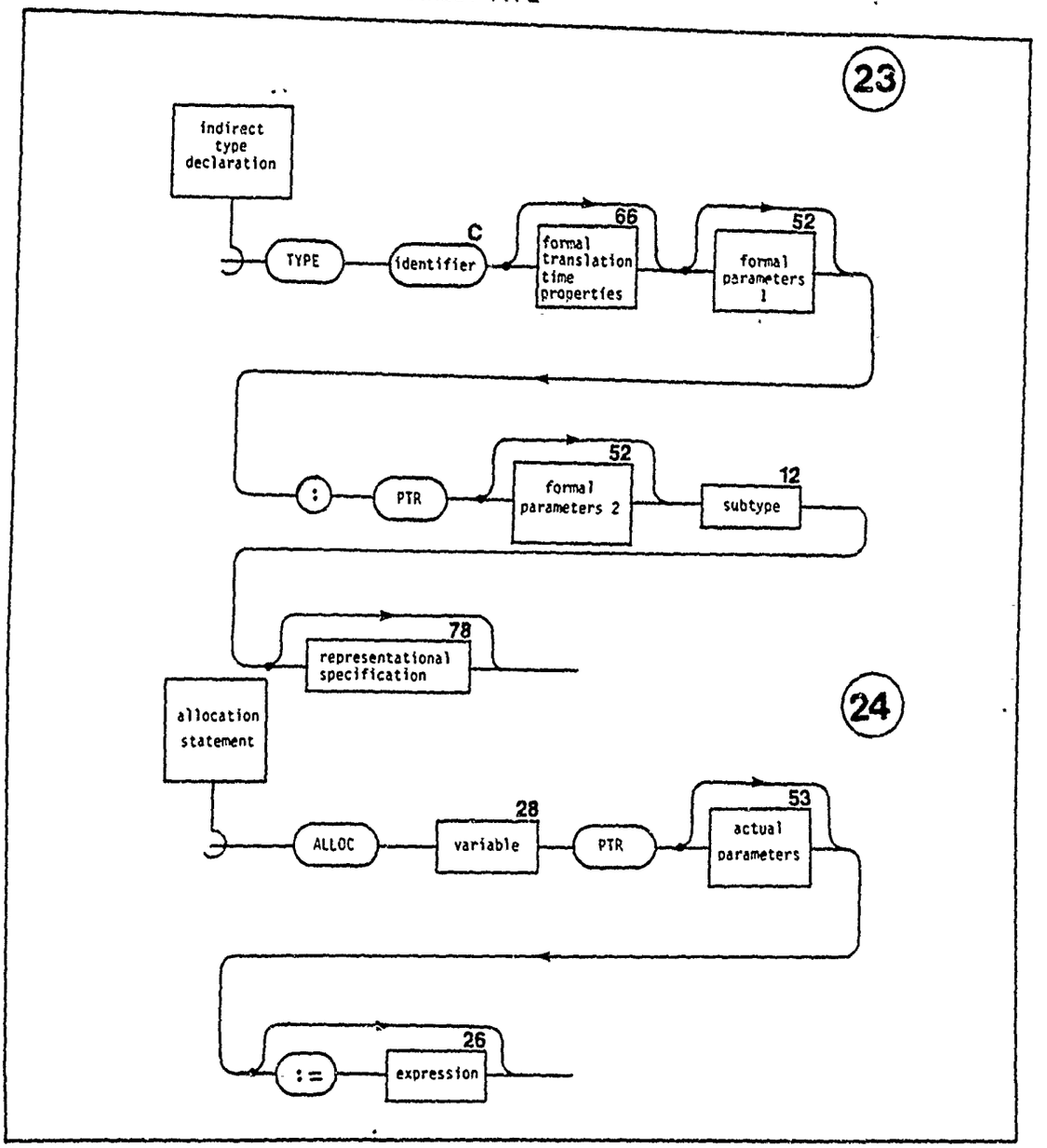
1) New string types

```
TYPE strng10 : STRING[ASCII] (10);

GENERIC i : INT
   TYPE mstrng[i] : STRING[ASCII] (i);

TYPE strng (j : INT) : STRING[ASCII] (j);

VAR x : strng10;      % underlying variable is a string
                      %  with length 10
VAR y : mstrng [k];   % underlying variable is a string with
                      %  length k where the value of k must
                      %  be known at translation time
VAR z : strng (m);    % underlying variable is a string with
                      %  length m where the value of m need
                      %  not be known until run time.
```

2) An abstract data type -- stacks

```
CAPSULE stackcap EXPORTS stack, init, push, pop;
   CONST size := 100;
   ABBREV elemtype : FLOAT(10, -1000.0 .. 1000.0);
   TYPE stack : RECORD[ top : INT(0..size),
                        elem : ARRAY INT(1..size) OF
                                 elemtype];

   PROC init (VAR s : stack);
      s.top := 0;
   END PROC init;

   PROC push (VAR s : stack, e :elemtype);
      s.top := s.top + 1;
      s.elem(s.top) := e;
   END PROC push;

   PROC pop (VAR s : stack, OUT e : elemtype);
      e := s.elem(s.top);
      s.top := s.top - 1;
   END PROC pop;

END CAPSULE stackcap;
```

4.4.3 DECLARING AND USING A NEW INDIRECT TYPE



A *variable* having an indirect type (called an indirect variable) is a pointer. The value of an indirect variable is either nil or a reference to some dynamic variable (i.e., it 'points' to the dynamic variable). For example, in

```
TYPE t : PTR STRING[ASCII] (5);
VAR  x,y : t;
```

the variables x and y are indirect variables that can either have the value nil or can point to some dynamic variable with subtype STRING[ASCII] (5). All indirect variables are automatically initialized to have value nil. There is also a literal for the value nil. For example,

```
x := NIL;
```

sets the value of x to be nil.

A dynamic variable is created by elaboration of an *allocation statement*. For example, elaboration of

```
ALLOC y PTR := "ABCDE";
```

creates a new dynamic variable with subtype STRING[ASCII] (5), initializes it to have the value "ABCDE", and sets y to point to this dynamic variable. Note that dynamic variables, unlike other variables, are not defined or named.

Dynamic variables are referenced via indirect variables. As with direct types, .ALL qualification and component selection operations (if the underlying dynamic variable has components) are automatically defined. These operations are permitted only if the value of the indirect variable is not nil. The operations provide access to the referenced dynamic variable or its components (i.e., they are 'dereferencing' operations). For example,

```
VAR s1,s2 : t;
ALLOC s2 PTR := "value";    % create dynamic variable
...

...s1.ALL...      % illegal since s1 is nil
...s1(i)...       % illegal since s1 is nil
...s2.ALL...      % the dynamic variable pointed to
                  %  by s2, having the value "ABCDE"
...s2(3)...       % a component of the dynamic
                  %  variable pointed to by s2,
                  %  having the value 'C

...
ALLOC s1 PTR;     % create dynamic variable 2
s1.ALL := s2.ALL; % sets the value of dynamic variable 2
                  %  to be equal to the value of dynamic
                  %  variable 1 ("value")
```

The lifetime of a dynamic variable is different than that of other *variables*. A dynamic variable exists as long as there is some way of accessing it. This means that the lifetime of a dynamic variable is not coupled to the elaboration of a scope.

As is the case for direct types, assignment (:=) is also automatically defined for indirect types. The assignment operation for indirects, however, is a "sharing" assignment. For example,

```
VAR a1,a2,b1,b2,b3 : t;
ALLOC b1 PTR := "VWXYZ" ;    % creates dynamic variable 1
ALLOC b3 PTR := "abcde";     % creates dynamic variable 2
a1 := NIL;                   % sets a1 to nil
a2 := a1;                    % sets a2 to nil
b2 := b1;                    % b2 now points to dynamic
                             %  variable 1
b1.ALL := b3.ALL;            % changes value of dynamic
                             %  variable 1
... b1.ALL ...               % has value "abcde"
... b2.ALL ...               % has value "abcde"
```

```
... b3.ALL ...                % has value "abcde"
```

The equality operators (=, /=) are also automatically defined for indirect types. For example,

```
...a1 = a2...          % true, both are nil
...b1 = b2...          % true, both point to same dynamic variable
...b2 = b3...          % false, each points to different
                       %  dynamic variable
...b2.ALL = b3.ALL..   % true, both dynamic variables have the
                       %  value "abcde"
```

As is the case for all data items, the *subtype* of a dynamic variable need not be known until the dynamic variable is created. Constraints on a dynamic variable which are to be resolved at creation time are specified via an allocation property list in an *allocation statement*. For example,

```
TYPE vstring : PTR(len : INT) STRING[ASCII] (len);

VAR v1,v2 : vstring;

ALLOC v1 PTR(3) := "abc";     % (3) is the allocation
                              %  property list
ALLOC v2 PTR(4) := "abcd";
```

Dynamic variables can contain components having indirect types which reference other dynamic variables. This means that recursive data structures and data structures having cycles can be created. For example,

```
TYPE list : PTR RECORD[ val : INT(0..100),
                        next : list];
VAR lst : list;

% create a singly linked list with 3 elements
ALLOC lst PTR := [val : 3, next : NIL];
ALLOC lst PTR := [val : 2, next : lst];
ALLOC lst PTR := [val : 1, next : lst];

% now make the list circular
lst.next.next.next := lst;
```

In addition to indirect variables, it is also possible to define indirect constants. Like all *constants*, an indirect constant must be initialized and its value may not be changed. The value of the dynamic variable which it references may, however, be changed.

When creating an abstract data type and its *subtype*, the programmer must ensure that the implementation of the abstract type is invisible to the user. This permits the implementation to be changed without affecting those parts of a program which use the abstract type. The programmer who implements an abstract type should be able to change the underlying type from a direct type to an indirect type (and vice versa), without affecting the users of the abstract type. For example, an abstract stack data type could be implemented using either an array (a direct type) or a linked list (achieved via an indirect type). For this reason, it is important that when a *type* is exported from a capsule used to realize an abstract data type, it should not be possible to detect outside the capsule whether the exported type was a direct or an indirect type.

As mentioned above, a dynamic variable exists as long as there is some way to access it. Detecting when there is no longer any way to access a dynamic variable and reclaiming the storage that was used for it usually involves a process called garbage collection. In some cases, the overhead of full garbage collection can be avoided and a simpler and less costly strategy used. For cases where this is impossible, the user can avoid garbage collection costs by the use of the FREE procedure. If v is an indirect variable that points to some dynamic variable and there are no other pointers to that dynamic variable, then

```
FREE(v);
```

reclaims the storage for that dynamic variable and sets the value of v to nil. If there were other pointers to the dynamic variable, the X_FREE exception is raised (this prevents the problem of dangling pointers). Although this avoids the cost of garbage collection, it introduces some cost in checking that there are no other pointers. For those cases where even this cost is unacceptable, it is possible to inhibit the generation of code for doing this checking by suppressing the X_FREE exception (see Appendix B).

RULES

Indirect Type Declaration

The *identifier* in an *indirect type declaration* is defined to be a type name in the scope in which the *indirect type declaration* appears. Indirect types are referenced using the same rules (both syntax and semantics) as for direct types (see Section 4.4.2).

Only CONST formal parameters may be used.

Indirect types are invoked using the same syntax as direct types (see Section 4.4.2). Elaboration of a *user-defined subtype* consists of elaborating the *actual parameters* and binding the *actual* parameters to the *formal parameters* of the named type (see Section 7.3).

The result of a *user-defined subtype* is a *subtype* of the invoked type, whose constraint property list is the actual parameter list of the invocation.

The value of an indirect variable or constant is either nil or a reference to some underlying dynamic variable. All indirect variables are automatically initialized to have the value nil.

The following operations are automatically defined for each indirect type:

a) Assignment (:=) is a sharing assignment. If the indirect subtypes of left hand side and right hand side are not equal the X_SUBTYPE exception is raised.

b) Equality (=) is defined to produce true if both of its actual parameters are nil or if both reference the same dynamic variable.

c) Component selection, if the underlying type has components. If the value of the *variable* or constant is nil when component selection is applied to the *variable* or constant, the X_NIL exception is raised.

d) .ALL qualification which gives access to the underlying dynamic variable. If the value of the variable or constant is nil when .ALL qualification is applied to the variable or constant, the X_NIL exception is raised.

Types are assumed to be normal (see Section 7.2.1).

Allocation Statement

The variable must have an indirect type on which .ALL qualification is available.

Elaboration of the allocation statement consists of:

a) elaborating the variable;

b) elaborating the actual parameters;

c) binding the actual parameters to the formal parameters 2 in the indirect type declaration associated with the type of the variable;

d) elaborating the subtype in the indirect type declaration;

e) allocating a dynamic variable having that subtype;

f) if initialization is specified, initializing the newly created dynamic variable (using :=); and

g) setting the variable named in the allocation statement to be a reference to the newly created dynamic variable.

The dynamic variable must have an assignable type if initialization is specified in an *allocation* statement.

NOTES

Because indirect types are always new types, and therefore named, the type equivalence rules are simplified (see Section 4.1.5).

An *indirect type declaration* is a closed scope. The formal parameter names (of both formal parameter lists) are defined in this scope. Since a *type declaration* cannot have an imports list, no variable names may be used within the *subtype*.

Representation specifications are used for machine-dependent programs and are described in Section 12.2.

EXAMPLES

1) Indirect string types

```
TYPE str10 : PTR STRING[ASCII] (10);

GENERIC i : INT
   TYPE mstr [i] : PTR STRING[ASCII] (i);

TYPE str(j : INT) : PTR STRING[ASCII] (j);

TYPE vstr : PTR(u : INT) STRING[ASCII] (u);

VAR w : str10;          % w can point only to strings
                        %  of length 10
VAR x : mstr [m];       % x can point only to strings
                        %  of length m where the value of
                        %  m must be known at translation
                        %  time
VAR y : str(n);         % y can point only to strings
                        %  of length n where the value of
                        %  n need not be determined until
                        %  run-time
VAR z : vstr;           % z can point to strings of any
                        %  length.  The length of the
```

```
                                % string is determined when the
                                % string is allocated.
```

2) Defines symbol tables which can hold symbols of different lengths

```
CAPSULE sym_tab_cap EXPORTS sym_tab, init, insert, look_up;

   TYPE sym : PTR (len : INT) STRING[ASCII](len);

   TYPE sym_tab (size : INT) : RECORD[top : INT(0..size),
                                      syms : ARRAY
                                             INT(1..size)
                                             OF sym];

   PROC init (VAR s : sym_tab);
      s.top := 0;
   END PROC init;

   FUNC look_up (READONLY s   : sym_tab,
                          val : STRING[ASCII])
                 => INT(0..s.top);
      FOR i : INT(1 .. s.top) REPEAT
         IF s.syms(i).ALL = val THEN
            RETURN i;
         END IF;
      END REPEAT;
      RETURN 0;
   END FUNC look_up;

   PROC insert (VAR s     : sym_tab,
                    val   : STRING[ASCII],
                OUT index : INT);
      index := look_up (s, val);
      IF index=0 THEN
         s.top := s.top + 1;
         ALLOC s.syms(s.top) PTR (val.LEN) := val;
         index := s.top;
      END IF;
   END PROC insert;

END CAPSULE sym_tab_cap;
```

4.5 TYPE AND SUBTYPE INQUIRY, PREDICATES AND ASSERTIONS

Since a *formal parameter* can specify a *type*, rather than a *subtype*, a deferred unit with a *formal* parameter can be invoked with *actual parameters* having any *subtype* belonging to that *type*. Although this flexibility is often quite useful, there are cases where it is desirable to further limit either the values or *subtypes* that *actual parameters* are permitted to have (in order to exclude values which are not meaningful for the deferred unit). In many cases, these limitations also allow the translator to produce more efficient code.

Some limitations can be achieved by specifying a *subtype* (rather than a *type*) for a *formal* parameter. A finer degree of control can be achieved by including an *assertion* at the beginning of the body of the deferred unit. *Assertions* concerning *subtypes* are supported by language facilities for inquiring about the *type*, *subtype* and subtype properties of a data item. These features are discussed in the following subsections.

Inquiry is also useful for several other purposes, including specifying the *subtype* of local data items of a procedure or function and accessing array index bounds.

4.5.1 SUBTYPE INQUIRY

RULES

If exp is any *expression*, then the result of elaborating

```
SUBTYPEOF(exp)
```

is the *subtype* of that *expression*.

If exp is an *expression* for an n-dimensional array and i is a manifest integer expression whose value is between one and n, then the result of elaborating

```
INDEXOF(exp, i)
```

is the i'th index subtype of the array. The form

```
INDEXOF(exp)
```

is equivalent to

```
INDEXOF(exp, 1)
```

EXAMPLES

```
EXCEPTION not_found;
...
FUNC f (x : INT) =>  SUBTYPEOF(x);
   ...
END FUNC f;

PROC q (VAR a,b : ARRAY INT OF FLOAT);
   ASSERT INDEXOF(a) = INDEXOF(b);
   ...
END PROC q;

FUNC search (a : ARRAY INT OF FLOAT, v : FLOAT) => INDEXOF(a);
   FOR i : INDEXOF(a) REPEAT
      IF a(i) = v THEN
         RETURN i;
      END IF;
   END REPEAT;
   RAISE not_found;
END FUNC search;

PROC r (VAR x,y : FLOAT);
   ASSERT SUBTYPEOF(x) = SUBTYPEOF(y);
   ...
END PROC r;
```

4.5.2 TYPE INQUIRY

RULES

If exp is an *expression*, then the result of elaborating

```
TYPEOF(exp)
```

is the *type* of that *expression*. If st is a *subtype*, then the result of elaborating

```
TYPEOF(st)
```

is the *type* to which that *subtype* belongs (see Section 4.1.3).

NOTES

Elaborations of TYPEOF takes place during translation.

4.5.3 ATTRIBUTES

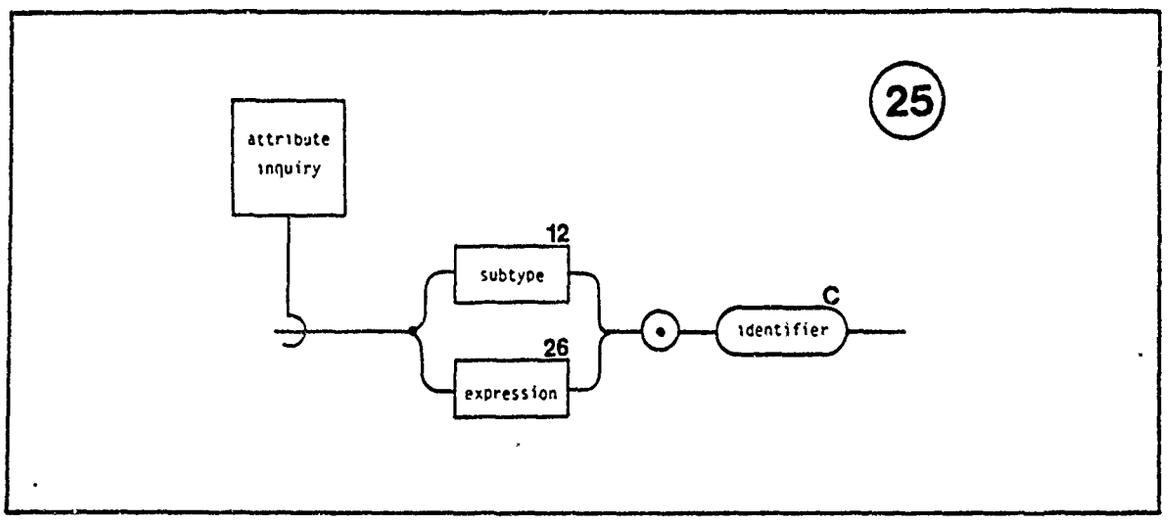
In addition to inquiry of an entire *subtype*, it is also possible to inquire about specific subtype constraints, called attributes.

RULES

The attributes of built-in types are listed in Appendix C.

Each *formal parameter* of a *user-defined type* is an attribute of that *type*. The *identifier* which is the name of the *formal parameter* is used as the attribute name.

ATTRIBUTE INQUIRY



Attribute inquiry allows attribute values to be accessed.

RULES

The *identifier* must be the name of an attribute of the specified *subtype* or of the *subtype* of the specified *expression*.

Elaboration of *attribute inquiry* produces the value of that attribute.

EXAMPLES

```
INT(1..10).MAX              % 10
FLOAT(5, 0.0 .. 10.0).PREC  %  5
STRING[ASCII](8).LEN        %  8


TYPE matrix (first, second : INT) : ARRAY INT(1..first),
                                        INT(1..second)
                                    OF FLOAT(10, -100.0 .. 100.0);
matrix(5,8).first       % 5
matrix(5,8).second      % 8


PROC s (VAR x : FLOAT);
   ASSERT x.PREC <= 10;
   ...
END PROC s;


FUNC search1 (a : ARRAY INT OF FLOAT, v : FLOAT)
              => INT(-1 .. INDEXOF(a).MAX);
   ASSERT INDEXOF(a).MIN = 0;
   FOR i : INDEXOF(a) REPEAT
      IF a(i) = v THEN
         RETURN i;
      END IF;
   END REPEAT;
   RETURN -1;
END FUNC search1;


PROC sort (VAR a : ARRAY INT OF STRING[ASCII]);
   ABBREV x : INDEXOF(a);
   FOR i : INT(x.MIN .. PRED(x.MAX)) REPEAT
      FOR j : REVERSE INT(i .. PRED(x.MAX)) REPEAT
         CONST k := j + 1;
         IF a(j) < a(k) THEN
            CONST t := a(j);
            a(j) := a(k);
            a(k) := t;
         END IF;
      END REPEAT;
   END REPEAT;
END PROC sort;
```

5. EXPRESSIONS

5.1 DATA ITEMS

There are five kinds of data items: defined variables, dynamic variables, defined constants, readonly data items, and temporary data items. Defined variables and constants are described in Section 4.2. Dynamic variables are described in Section 4.4.3. Readonly and temporary data items are discussed below.

The term *variable* is used to refer to defined and dynamic variables. The term *constant* is used to refer to defined constants, readonly data items, and temporary data items.

RULES

A data item can hold a single value. The value of any data item may be accessed (i.e., read). The value of *variables* (both defined and dynamic) and their components may also be modified. The value of other data items may not be modified.

A readonly data items is a *variable* whose use is restricted in certain contexts. Within those contexts, the value of the readonly data item cannot be modified directly. In some cases, however, it is possible for the value of the readonly data item to be modified indirectly, outside the context.

a) When an actual parameter variable is bound to a READONLY formal parameter, the formal parameter is treated as a readonly data item within a deferred declaration. Changes to the actual parameter variable will change the formal parameter (see Section 7.3). This is the only way that dynamic variables may be made readonly.

b) *Variables* imported READONLY into a *compound declaration* are treated as readonly data items within the *compound declaration*. The *variable* may be changed outside the *compound* declaration (see Section 3.7).

c) *Variables* exported READONLY from a capsule into a scope where the capsule is invoked are treated as readonly data items within that scope. The *variable* may be changed within the capsule (see Section 8.2)

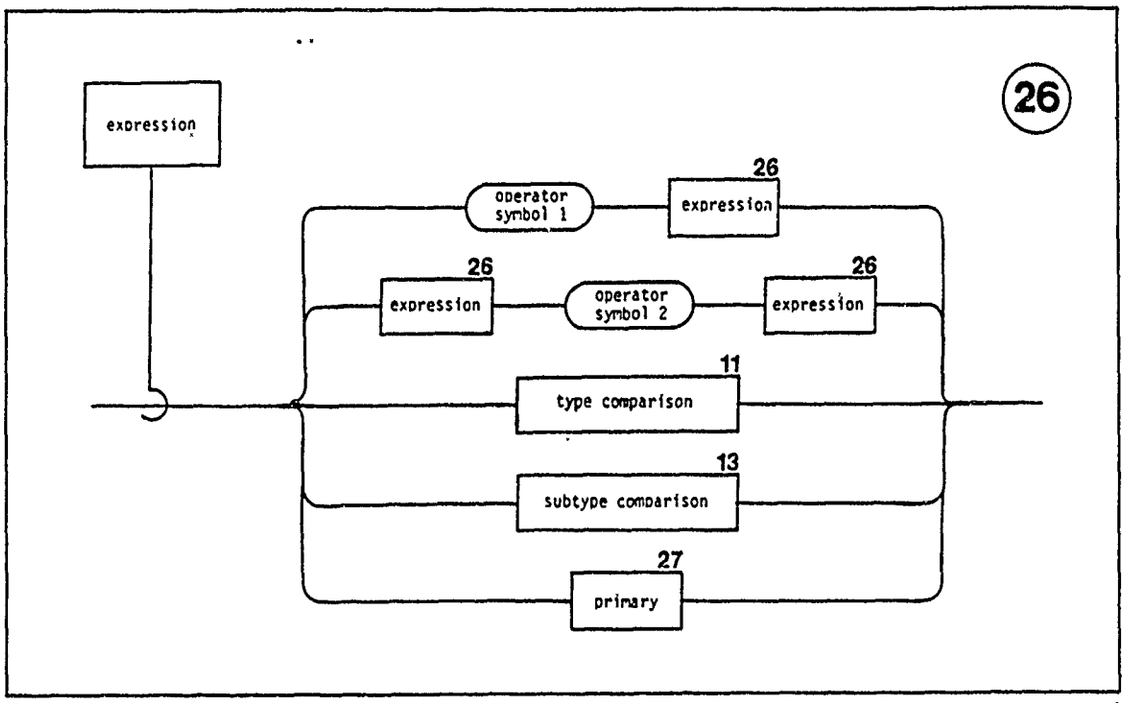
d) *Variables* exported from a capsule and exposed as READONLY are treated as readonly data items within the scope in which the capsule is invoked. The *variable* may be changed within the capsule (see Section 8.1).

A temporary data item is the result of a built-in or user-defined literal or constructor (see Sections 2.3.5, 5.6, and 5.7), the result of a function or operator (see Section 7.2), or the result of *attribute inquiry* (see Section 4.5.3). For convenience, these results will be referred to as values.

The lifetime of all defined variables and constants and *formal parameters* (and their components) is the lifetime of the scope immediately containing their definition. The lifetime of dynamic variables extends from their creation by the *allocation statement* until the time when they can no longer be accessed. The lifetime of temporary data items extends from the time they are produced until the end of the elaboration of the construct in which the temporary is used.

Initialization can optionally be specified whenever a *variable* is created. For some types, if no initialization is specified, there is a default initial value. Otherwise, the *variable* is uninitialized until a value has been assigned to it. An attempt to access the value of an uninitialized variable will raise the X_INIT exception.

5.2 OPERATORS AND OPERANDS



An *expression* is a computational rule for producing a data item. An *expression* can be a single operand or a combination of operators and operands. Operators are either prefix or infix operators. Prefix operators immediately precede an operand. Infix operators operate upon a left and right operand to produce a value. The association of operands to an operator is determined by the precedence of operators. Operands are associated with the operator of higher precedence.

Parentheses can be used to modify the association (see Section 5.3).

RULES

Operator symbol 1 is a prefix operator. Valid prefix operator symbols are +, -, and NOT. Operator symbol 2 is an infix operator. Valid infix operator symbols are **, *, /, MOD, DIV, &, +, -, =, /=, <, <=, >, >=, IN, AND, OR, and XOR .

The operands of prefix and infix operators are determined based on the built-in precedence of operators given below:

```
highest precedence
  +, -                  (prefix)
  **
  *, /, MOD, DIV, &
  +, -
  =, /=, <, <=, >, >=, IN
  NOT                   (prefix)
  AND
  OR, XOR
lowest precedence
```

Within a precedence level, associativity is left to right.

NOTES

Arithmetic Operators

Arithmetic operators (infix +, -, *, /, **, modulo (MOD), integer division (DIV), and prefix +, -) take operands of arithmetic types (INT, FLOAT, and FIXED) and return arithmetic values.

Concatenation Operator

A concatenation operator (infix &) takes string and enumeration operands and produces a string, or takes one-dimensional array operands and produces a one-dimensional array.

Relational Operators

Relational operators (infix =, not equal (/=), <, <=, >, >=, set membership (IN)) take arithmetic, boolean, enumeration, string, and set operands and return a boolean. For boolean operands, only = and /= are defined. For arithmetic operands, the relational operators define a numerical ordering. For enumeration and string operands, the relational operators define a collating sequence. For set operands, the relational operators define a subset relationship.

Logical Operators

Logical operators (infix AND, OR, XOR, and prefix NOT) take boolean operands and produce a boolean or take set operands and produce a set. For boolean operands, the logical operators define and, or, exclusive or, and complement. For set operands, the logical operators define set intersection, union, symmetric difference, and complement.

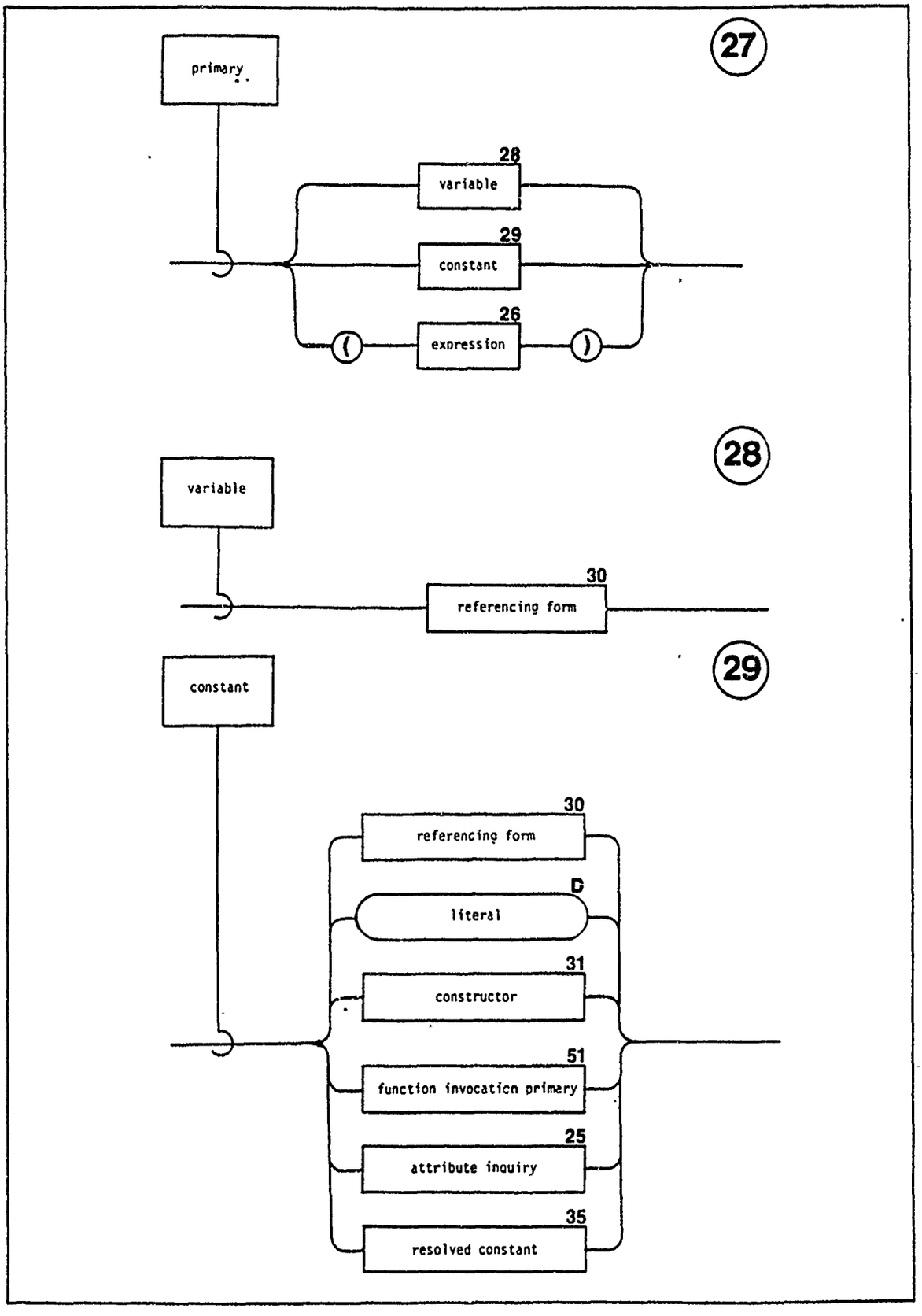
A *type* or *subtype comparison* is one which is used to compare *types* or *subtypes* and returns a boolean value. This form of *expression* is described in Section 4.5.1 and 4.5.2.

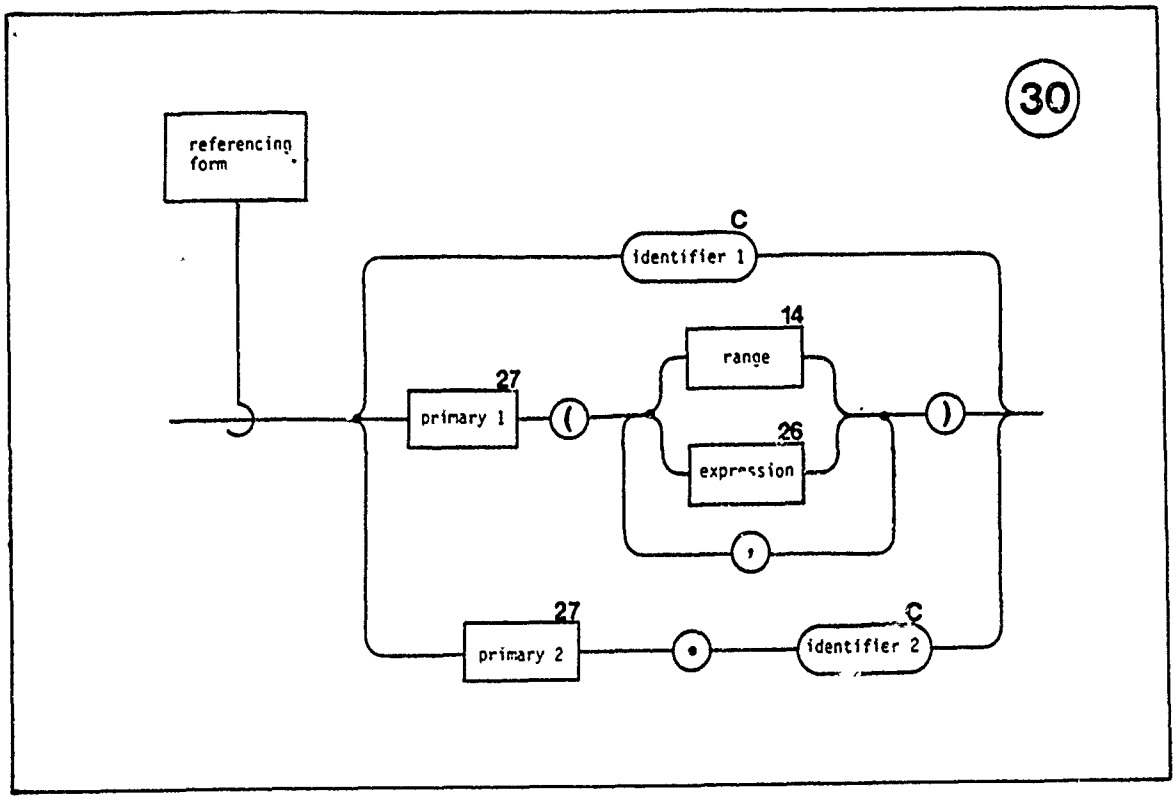
It is possible to overload the built-in definitions of operators to allow user-defined operators (see Section 13.2).

EXAMPLES

```
int_var                          % produces integer
3+4 = (7 DIV 3)+2                % produces boolean
'red > enum_var                  % produces boolean
array1(2..4) & array2(3..7)      % produces array
set1 AND set2                    % produces set
(3+i) DIV (7-j)                  % produces integer
```

5.3 PRIMARY


27
primary
28
variable
29
constant
26
(
expression
)
28
variable
30
referencing form
29
constant
30
referencing form
D
literal
31
constructor
51
function invocation primary
25
attribute inquiry
35
resolved constant



A *primary* is the basic form of an *expression*.

A *variable* can be an entire *variable* or one or more components of a *variable*. A *constant* can be an entire *constant* or one or more components of a *constant*. A *constant* can be either a reference to a defined constant, written as a *literal*, created by a *constructor*, the result of a function, or an attribute value.

A *variable* or *constant* which is a component of a record or union type is referenced by dot selection. An underlying dynamic *variable* of an indirect variable or constant (see Section 4.4.3) is referenced by the dot selector .ALL. A *variable* or *constant* which is one or more components of an array or string is referenced by subscripting to produce a single component or slice.

RULES

Identifier 1 must be associated with a *variable* or *constant* in the scope immediately containing the *expression* containing *identifier 1*.

The result of elaboration of a parenthesized expression is the result of elaboration of the *expression*.

NOTES

Rules for subscripting are described in Appendix C under ARRAY and STRING types. Rules for dot selection are described in Appendix C under RECORD and UNION types and in Section 13.4. User-defined subscripting and dot selection is described in Section 13.4 *Literals* and *constructors* are described in Section 5.4. User-defined literals and constructors are described in Section 13.5 *Function invocation primaries* are described in Section 7.2. *Attribute inquiry* is described in Section 4.5.3. *Resolved constants* are described in Section 5.7

5.4 BUILT-IN AND USER-DEFINED LITERALS AND CONSTRUCTORS

Literals and *constructors* are used to specify values for *variables* and *constants*. *Literals* are provided for specifying values of each basic language type. For example,

```
TRUE                        % a BOOL literal
'red                        % an ENUM literal
32                          % an INT literal
4.53E6                      % a FLOAT literal
"This is a string."         % a STRING literal
```

The literal, NIL, is also provided to specify a value for all indirect types. Chapter 2 gives forms for built-in literals and Section 5.5 gives rules for their types and subtypes.

Values for variables consisting of multiple components are written using constructors. For example,

```
VAR a : ARRAY  INT(1..10) OF BOOL;
VAR r : RECORD[ re,im : FLOAT(10, -100.0 .. 100.0)];

a := [1 : FALSE, 2..10 : TRUE];

r := [re : 3.6, im : -5.7];
```

Section 5.6 gives rules for built-in constructors.

Users can also define literal and constructor forms for new types (see Section 13.5). For example,

```
% suppose MILES, COMPLEX, and VSTRING are user-defined types
%   for which user-defined constructors are available

CONST d := 10.3#MILES;
VAR c : COMPLEX;
VAR v : VSTRING;

c := [re : 0.0, im : 2.3]#COMPLEX;
v := "This is the value"#VSTRING;
```

5.5 MANIFEST EXPRESSIONS AND CONDITIONAL TRANSLATION

A manifest expression is an *expression* whose value is known during translation. The simplest manifest expression is a *literal*. Manifest expressions have two important uses: first, they are used to achieve conditional translation and second, they are used as replacement elements for *generic* parameters with *value generic constraints* (see Section 11.3.4).

RULES

Manifest Expressions

The following are manifest expressions:

a) any *literal*;

b) the result of any of the following built-in operators when their operands (actual parameters) are manifest

```
BOOL, INT, FLOAT operations  --  all, except :=
ENUM operations              --  =, /=, &
STRING operations            --  &, =, /=, component selection
```

c) a parenthesized expression, where the *expression* is a manifest expression;

d) references to *constants* declared with the form

```
CONST id := exp;
```

where exp is a manifest expression; and

e) references to generic parameters with a value generic constraint (see Section 11.3.4).

No other *expressions* are manifest expressions.

All built-in arithmetic operations in manifest expressions are performed using the maximum precision and range of the target system.

Conditional Translation

If the condition of an *if* or *case statement* is a manifest expression, code is generated only for the selected alternative body. Any translation time errors that occur in those *bodies* not selected will be treated as warnings and will not prevent program execution.

If the subtype constraint (i.e., the tag value) of a union subtype (see Appendix C.6) is manifest, then only space for that component is reserved.

NOTES

Manifest expressions are guaranteed to be elaborated at translation time; however, this does not prohibit the translator from also elaborating any other *expression* whose value it can determine.

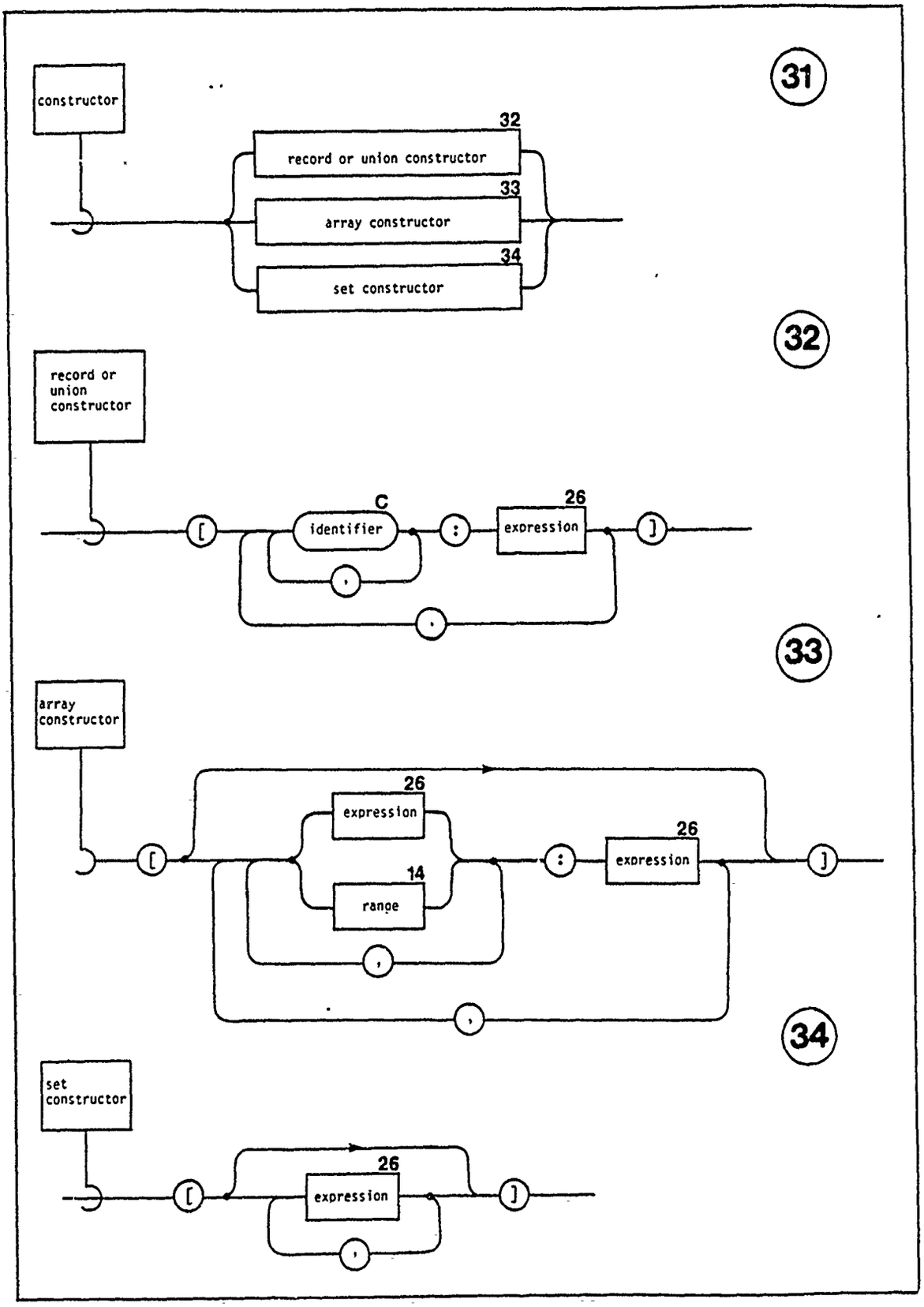
Section 5.7 gives rules for type and subtype resolution of manifest expressions.

EXAMPLES

1) Conditional translation

```
CONST machine := 'PDP11;
...
ABBREV word : UNION[ S360, S370 : bits(32),
                     PDP11      : bits(16),
                     CDC6600    : bits(60)] (machine);
...
IF machine = 'PDP11 THEN
   ...
END IF;
...
CASE machine
   WHEN 'S360, 'S370 => ...
   WHEN 'PDP11       => ...
   WHEN 'CDC6600     => ...
END CASE;
```

5.6 BUILT-IN CONSTRUCTORS



Constructors are used to construct values for records, unions, arrays, and sets.

RULES

Values can only be constructed for assignable types.

For a *record or union constructor* which is resolved to a record type, there must be a value specified for each component of the record type, and the components must be specified in the same order as in the record type.

For a *record or union constructor* which is resolved to a union type, there must be a value specified for only one component of the union type.

For an *array constructor* which is resolved to some array subtype, there must be a single value specified for each component. The range form is used to specify a single value for some contiguous range of components.

For a *set constructor* which is resolved to a set type, there can be zero or more values specified, where each value must have the component type of the set. An empty *set constructor* is the value of an empty set.

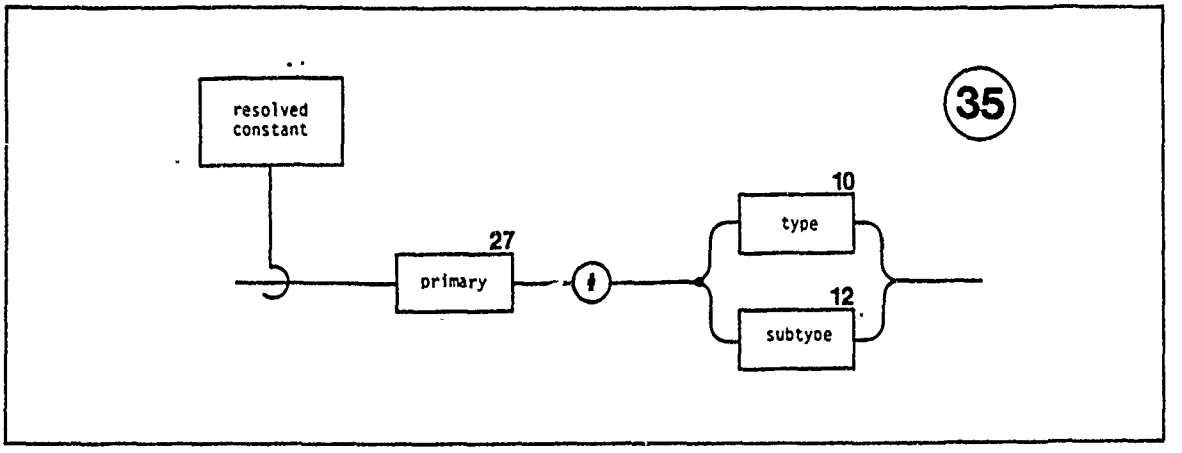
NOTES

Section 5.7 gives rules for type and subtype resolution of constructors.

EXAMPLES

```
VAR r1,r2 : RECORD[w,x : INT(1..10),
                   y : BOOL,
                   z : ENUM['a, 'b, 'c]];
VAR u1, u2 : UNION[w,x : INT(1..10),
                   y : BOOL,
                   z : ENUM['a, 'b, 'c]];
VAR s1, s2 : SET[ENUM['red, 'blue, 'yellow, 'green]];
VAR a1, a2 : ARRAY INT(1..3), ENUM['a, 'b, 'c]
                   OF INT(0..10);
VAR x : RECORD[a : BOOL,
               b : SET[INT(1..10)],
               c : UNION[d : INT(0..5),
                         e : BOOL]];

r1 := [w, x : 1, y : FALSE, z : 'a];
r2 := [w : 1, x : 2, y : TRUE, z : 'b];
u1 := [z : 'b];
u2 := [w : 3];
s1 := ['red, 'green];
s2 := [];
a1 := [1,'a : 0,  1,'b : 1,  1,'c : 2,
       2,'a : 3,  2,'b : 4,  2,'c : 5,
       3,'a : 6,  3,'b : 7,  3,'c : 8];
a2 := [1..3, 'a..'c : 10];
x := [a : FALSE, b : [2,3], c : [d : 3]];
```

5.7 VALUE RESOLUTION AND USER-DEFINED VALUES



The *resolved constant* has two purposes: it can be used to create a user-defined literal or constructor; and it can be used to specify the *type* or *subtype* of a manifest expression or *constructor*, when the *expression* would otherwise be ambiguous.

User-Defined Values

A literal or constructor can be defined for a user-defined type by overloading the definition of #. This is described in Section 13.5.

Type and Subtype Resolution

All *expressions* must have both a *type* and a *subtype*. For many kinds of *expressions*, the *type* and *subtype* of the *expression* depends only upon the *expression* itself. For manifest expressions and *constructors*, the *type* and *subtype* may also depend upon the context in which the manifest expression or *constructor* appears. If the context is the right hand side of an assignment, then the *type* and *subtype* are those of the left hand side. For example,

```
VAR color : ENUM[ 'red, 'orange, 'blue];
VAR fruit : ENUM[ 'apple, 'banana, 'orange];

color := 'orange;    % the subtype of 'orange is
                     %   ENUM[ 'red, 'orange, 'blue]
fruit := 'orange;    % the subtype of 'orange is
                     %   ENUM[ 'apple, 'banana, 'orange]
```

and

```
VAR r : RECORD[ b : INT(1..10)];
VAR u : UNION[ a,b,c : INT(1..5)];

r := [b : 3];        % the subtype of [b : 3] is
                     %   RECORD[ b : INT(1..10)]
u := [b : 3];        % the subtype of [b : 3] is
                     %   UNION[ a,b,c : INT(0..5)]
```

In some cases, the *type* and/or *subtype* of a *literal* or *constructor* can not be determined from the *literal* or *constructor* and its context. In these cases, the *type* or *subtype* must be explicitly specified. For example,

```
PROC p (c : color);                          % definition 1
   ...
END PROC p;

PROC p (f : fruit);                          % definition 2
   ...
END PROC p;

p ('orange);                                 % illegal, ambiguous
...
p ('orange#color);                           % legal, invokes def 1
```

and

```
ABBREV order1 : ENUM['a, 'b, 'c];
ABBREV order2 : ENUM['c, 'b, 'a];

VAR x : order1;
VAR y : order2;

...   'a < 'c    ...                  % illegal, ambiguous
...   'a#order1 < 'c#order1   ...     % legal, true
...   'a#order2 < 'c#order2   ...     % legal, false
...   x < 'b  ...                     % legal, context resolves 'b to
                                      %   order1
```

Manifest expressions can also depend upon context for their *type* and *subtype*. For example,

```
VAR s : STRING[ASCII] (5);
VAR x : FLOAT(3, 0.0 .. 100.0);
VAR y : FLOAT(5, 0.0 .. 100.0);
CONST pi := 3.14149;

s := "ABC" & 'CR & 'LF;      % legal, subtype of right-
                             %  hand side is resolved to subtype of
                             %  left hand side
x := pi;                     % subtype of pi is FLOAT(3, 0.0 .. 100.0)
y := pi;                     % subtype of pi is FLOAT(5, 0.0 .. 100.0)
```

The *type* or *subtype* of a manifest expression or of a *constructor* are resolved based on the context in which they appear. If context is not sufficient, a default is provided for integer, enumeration, floating point, or string values. In cases where resolution cannot be done from the context or default alone, the resolved constant form can be used to explicitly specify a *type* or a *subtype*.

RULES

Context Resolution

If a type or subtype unresolved expression appears:

a) on the right hand side of an assignment, it is resolved to the *subtype* of the left hand side.

b) as an *actual parameter*, it is resolved to the specified *type* or *subtype* of the corresponding *formal parameter*, if there is no ambiguity.

c) within a *constructor*, it is resolved to the component subtype of the constructed subtype.

d) in a resolved constant form, it is resolved to the specified *type* or *subtype*.

e) for unresolved float expressions, which appear as one operand of a built-in binary float operator (+, -, *, /, **, relationals), the precision is resolved to be equal to that of the other operand.

Default Resolution

When a *subtype* cannot be resolved from context, in the following cases, a default subtype is used. For an expression with value v, the default subtype for various types is shown below.

INT	the default subtype is INT(v..v)
ENUM[...]	the default subtype is ENUM[...]
FLOAT	the default subtype is FLOAT(p,v..v) where p is the maximum of the default precisions of all the float literals that are part of the manifest expression
STRING[ENUM[...]]	the default subtype is STRING[ENUM[...]] (len) where len is the number of components in v

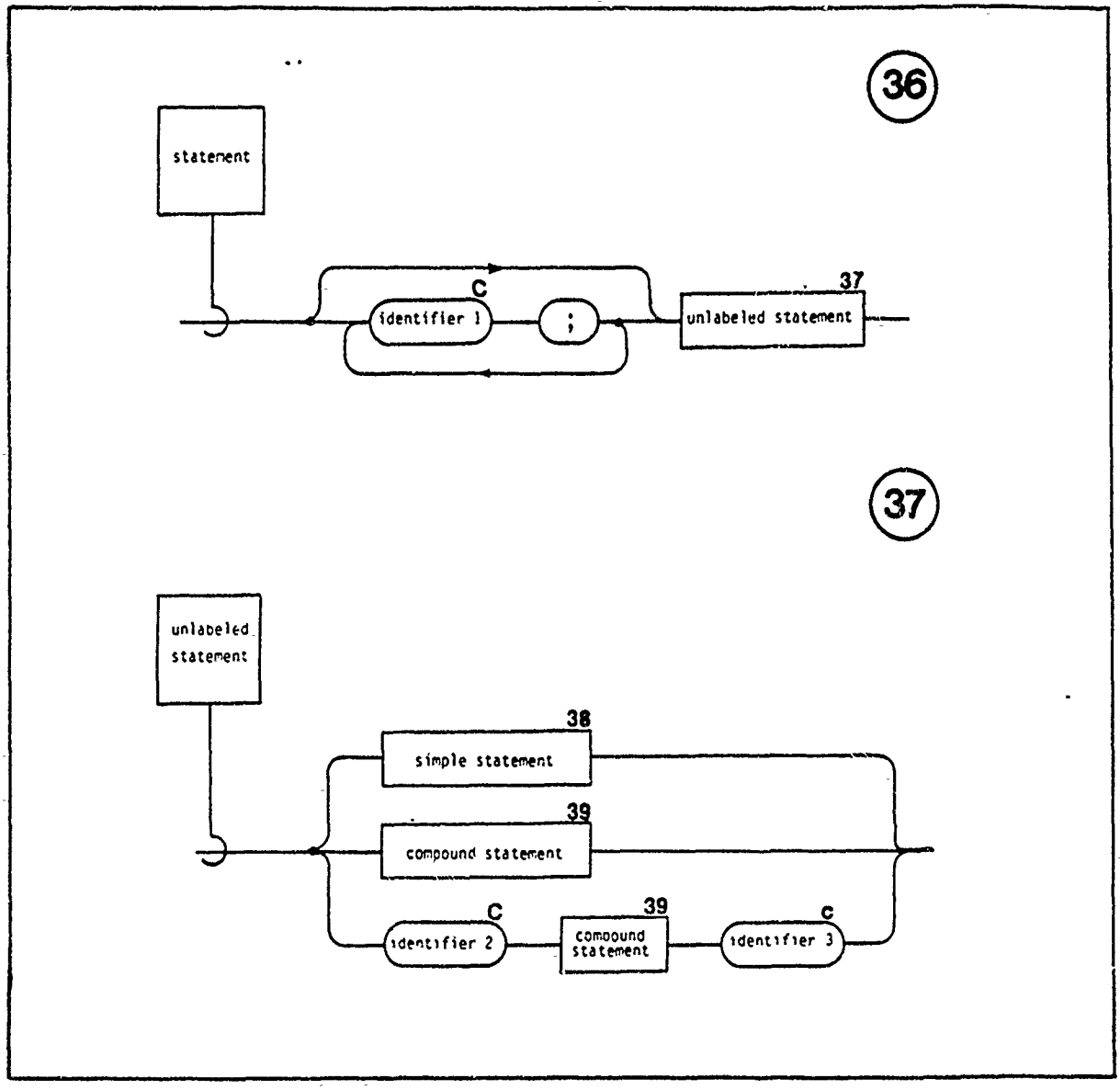
Explicit Resolution

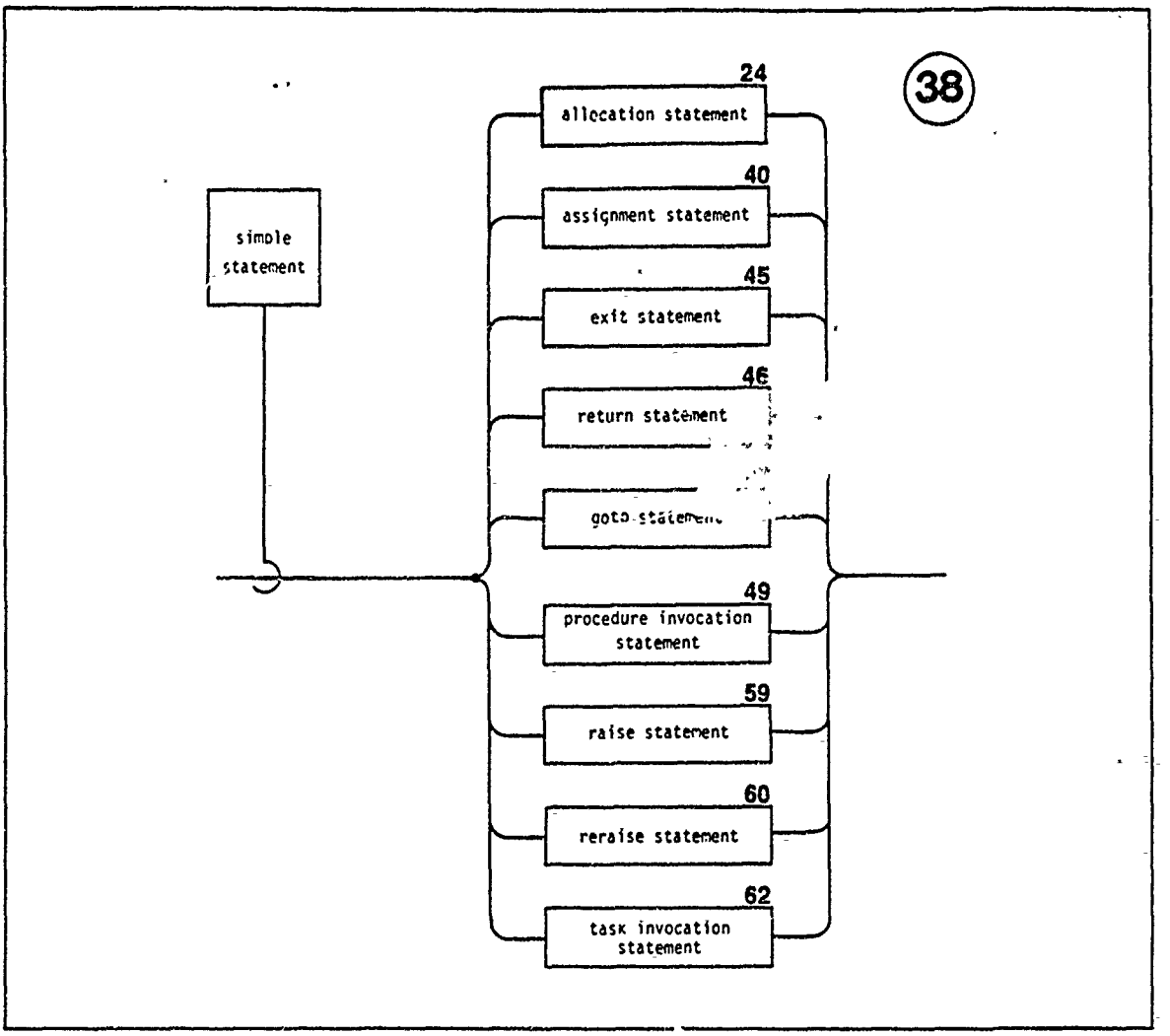
When a *type* or a *subtype* cannot be resolved from context or default, the manifest expression or *constructor* is written as a *resolved constant*. The unresolved expression preceding the # is resolved, if possible, to the *type* or *subtype* following #.

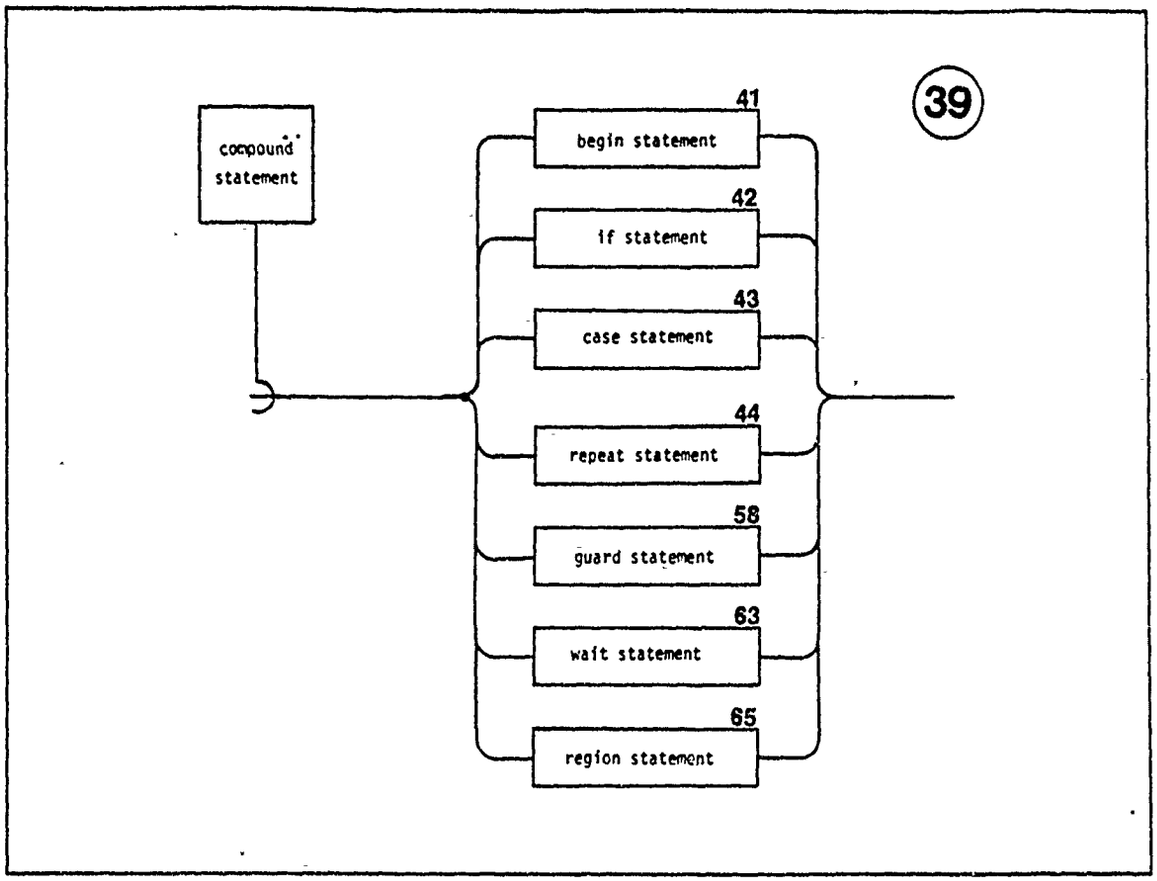
NOTES

The resolved constant form can always be used to resolve an *expression* with an ambiguous type or subtype.

6. STATEMENTS




38
simple statement
24
allocation statement
40
assignment statement
45
exit statement
46
return statement
goto statement
49
procedure invocation statement
59
raise statement
60
reraise statement
62
task invocation statement



Statements fall into two categories: compound and simple. A *compound statement* contains a header, one or more *bodies* (with delimiters between *bodies*), and an ending. Any *compound statement* can be prefixed and suffixed by matching identifiers. These *identifiers* can serve as the target of an *exit statement* and, in addition, enhance program readability. Each *compound statement* is an open scope. The only definitions which are local to a *compound statement* are the matching identifiers and, in the case of a *repeat statement*, the index variable. Notice, however, that each *body* contained in a *compound statement* is a scope that may contain local definitions. A *simple statement* does not contain a *body*, cannot be surrounded by matching identifiers, and is not a scope. Any *statement* may optionally be preceded by labels that can serve as the target for a *goto statement*.

RULES

Identifier 1 is called a goto label and is defined in the scope immediately containing the statement it prefixes.

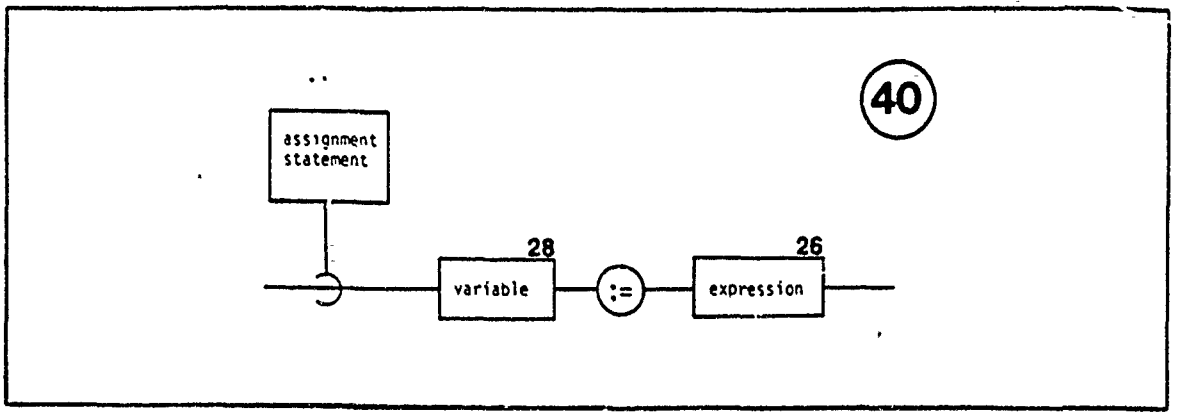
Identifiers 2 and *3*, called matching identifiers, must be identical. The matching identifier is defined in the scope of the *compound statement* which it brackets.

NOTES

Goto labels and matching identifiers are automatically inherited by open scopes (see Section 3.5) but are never inherited by closed scopes and may never be exported from a capsule (see Section 8.2). A matching label can never be the target of a *goto* statement, and a goto label can never be the target of an *exit statement*.

Even though there is no "empty statement", empty bodies are permitted (see Section 3.2).

6.1 ASSIGNMENT STATEMENT



The *assignment statement* is used to copy the value of an *expression* into a *variable*.

RULES

The *expression* must have the same *type* as the *variable*; that *type* must be assignable (see Section 4.3).

Elaboration of the *assignment statement* replaces the current value of the *variable* with the value of the *expression*.

The value of the *expression* must satisfy any subtype constraints of the *variable* (e.g., range, precision, array bounds); otherwise, the appropriate exception is raised.

NOTES

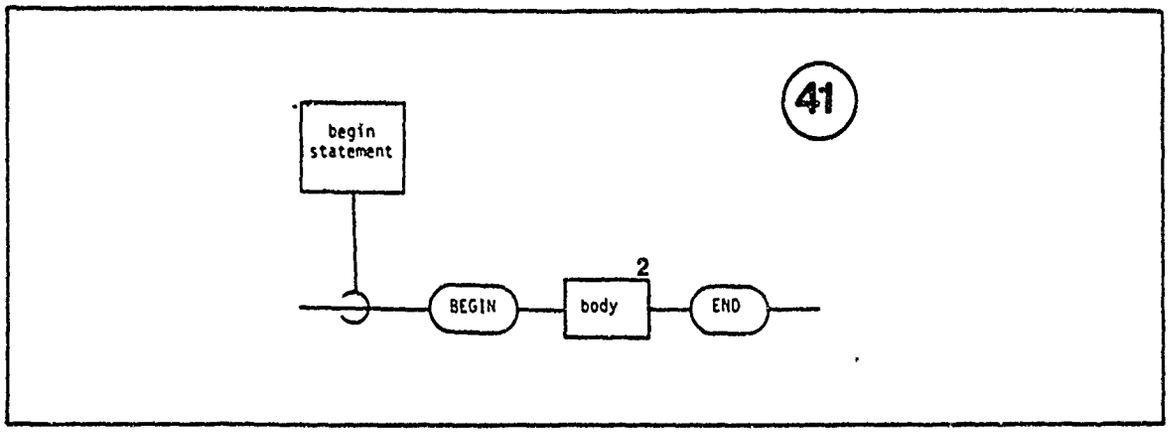
Entire arrays, array slices, and records may be assigned in a single assignment.

For rules concerning the assignment of built-in types, see Appendix C. For rules concerning assignment of user-defined types, see Section 13.2.

EXAMPLES

```
VAR x,y : INT(1..10);
VAR a1,a2 : ARRAY INT(1..10) OF BOOL;

a1 := f(a1);
a2 := a1;
a1(1..5) := a2(3..7);
x := 3;
y := x + 1;
```

6.2 BEGIN STATEMENT



The *begin statement* can be used to group together a sequence of related *statements*. Since the *begin statement* introduces a new open scope, it is also a convenient means for localizing *declarations* to the sequence of *statements* where they are needed.

RULES

The *begin statement* is elaborated by elaborating its *body*.

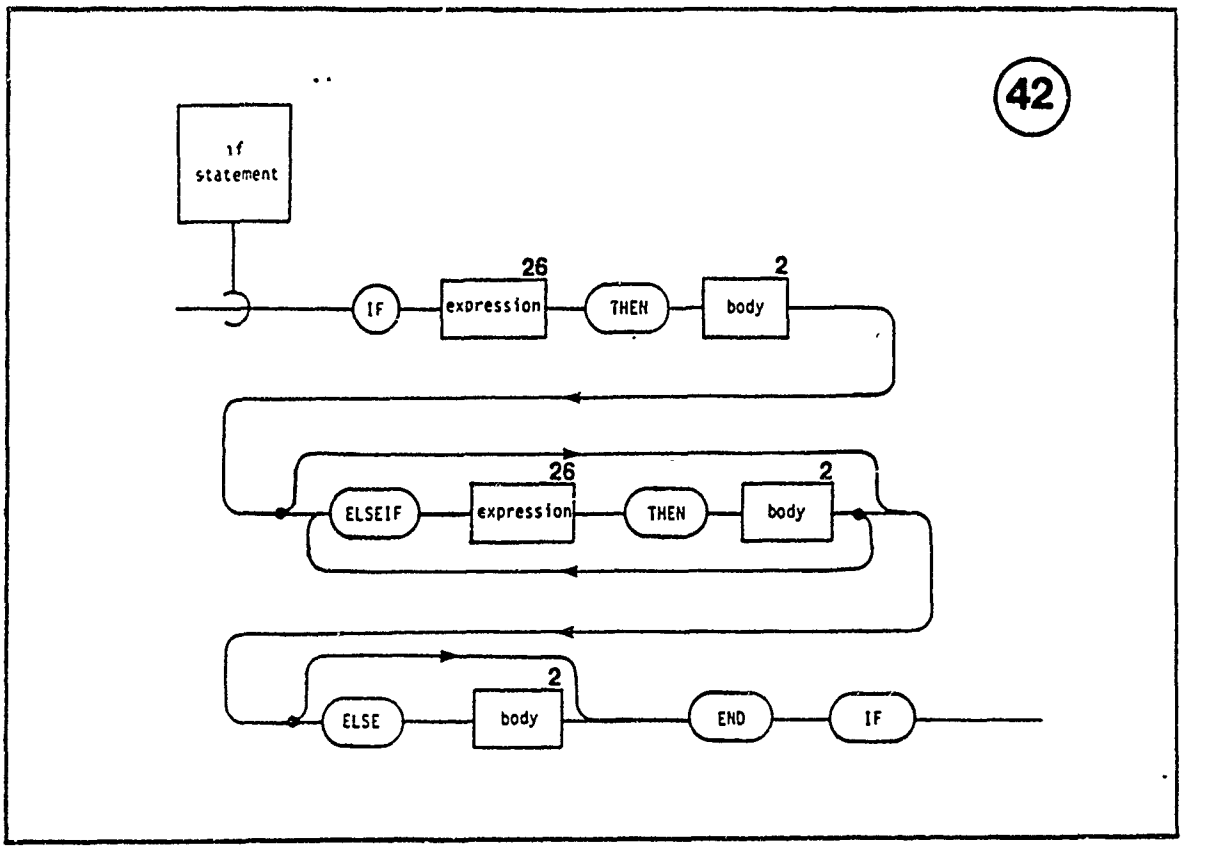
NOTES

All *compound declarations* and *compound statements* contain *bodies*. It is therefore unnecessary to write a *begin statement* in those contexts in order to achieve grouping of multiple *statements* into a single *statement* or to achieve a new scope for *declarations*.

EXAMPLES

```
BEGIN
   % swap record components
   VAR x : RECORD [a,b : INT(1..10)];
   READ (infile, x);
   BEGIN
      CONST t := x.a;
      x.a := x.b;
      x.b := t;
   END;
   WRITE (outfile, x);
END;
```

6.3 IF STATEMENT



The *if statement* selects one of several alternative bodies for elaboration, based on the value of one or more boolean expressions.

RULES

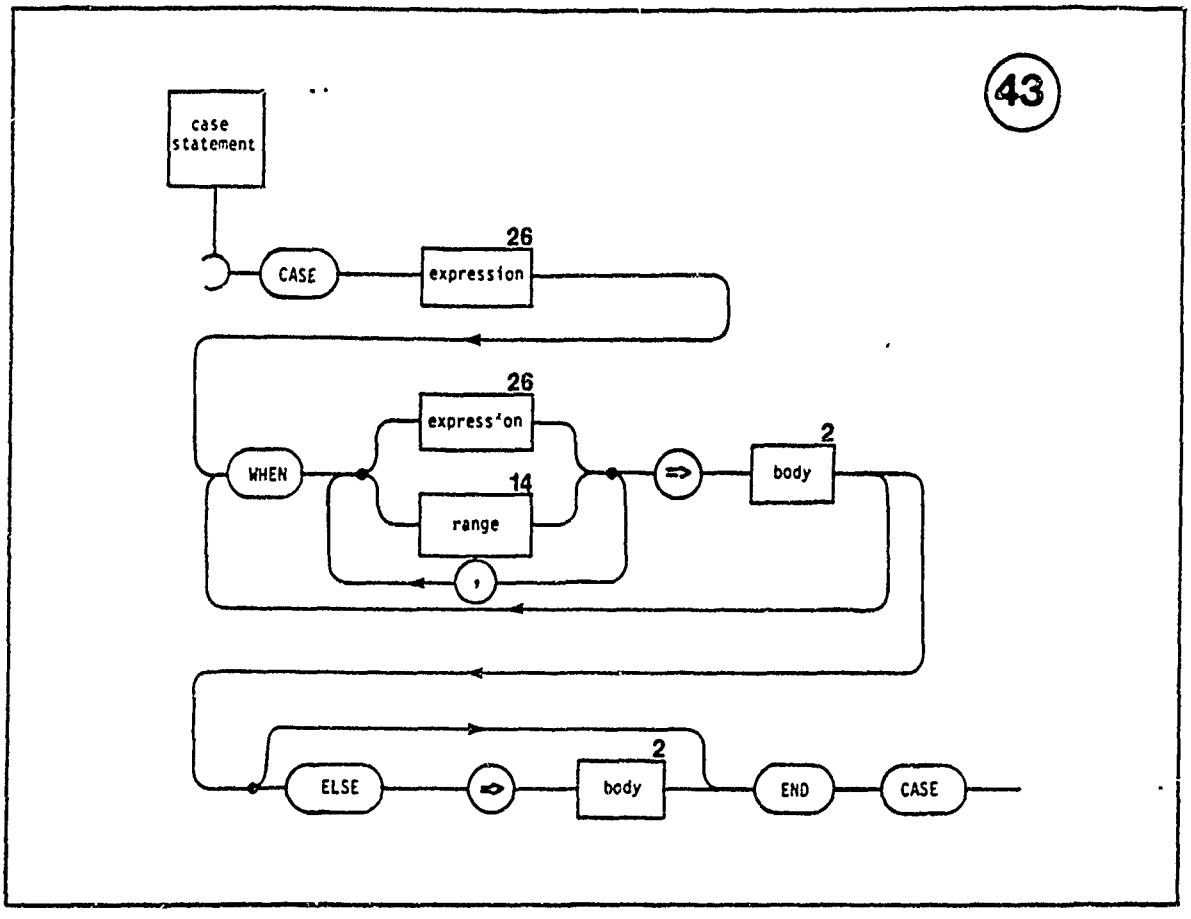
Each expression must have type BOOL.

Elaboration of an if statement proceeds by elaboration of each expression in order until either an expression with value true or a final ELSE is reached; then the corresponding body is elaborated. If no expression has value true and no final ELSE is present, none of the bodies is elaborated.

EXAMPLES

```
IF x < y THEN
   CONST t := x;
   x := y;
   y := t;
END IF;

IF i < j THEN
   k := i;
ELSE
   k := j;
END IF;

IF flag THEN
   i := i + 1;
ELSEIF j < 0 THEN
   i := 1;
ELSEIF k < 0 THEN
   i := 2;
ELSE
   i := i - 1;
END IF;
```

6.4 CASE STATEMENT



The *case statement* allows a *body* to be selected for elaboration from one or more alternate bodies, based on the value of a single *expression*.

RULES

Between each WHEN and => is a list of value labels. A value label can be either an *expression* (a single value label), or a *range* (a range value label). The *expressions* in all value labels and the *expression* following CASE must be of the same *type*. If only single value labels appear, the *type* must be one for which = is defined. If any range value labels appear, the *type* must be one for which = and < are defined.

Elaboration of a *case statement* consists of the following steps:

a) The value of the *expression* following CASE is compared to the values of the value labels. A match is found if the value of the *CASE* expression is either equal to the value of a single value label or is within the *range* of a range value label (see Section 4.1.7). The order in which value labels are examined is undefined.

b) If one or more matches are present, then the *body* associated with one of the matching value labels is elaborated.

c) If no match is found and the ELSE is present, the body following ELSE is elaborated.

d) If no match is found and the ELSE is not present, the X_CASE exception is raised.

NOTES

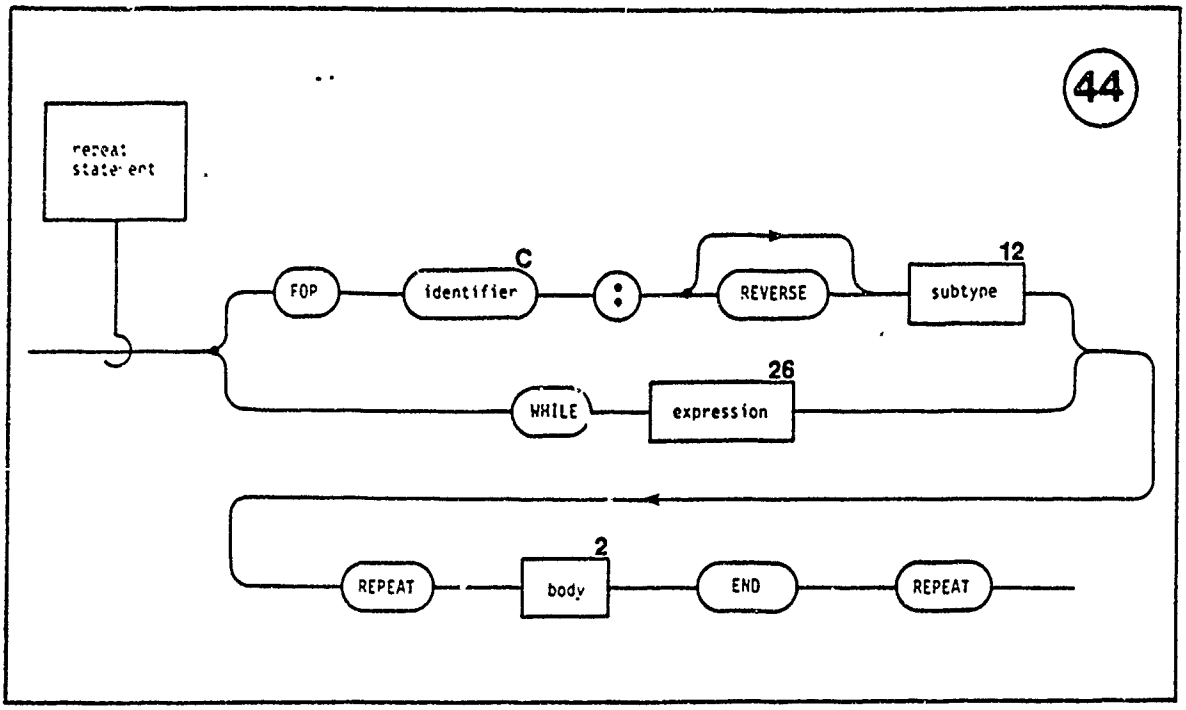
When more than one matching value label is found, it cannot be predicted which of the *bodies* associated with the matching labels will be elaborated.

Use of the *case statement* for *user defined types* is described in Section 13.6.

EXAMPLES

```
VAR i : INT(-10..10);
...
CASE i
   WHEN       0  => s := "ZERO";
   WHEN -10..-1  => s := "NEG";
   WHEN    1..5  => s := "SMALL";
   ELSE          => s := "LARGE";
END CASE;

VAR u : UNION [
           a : INT(1..10),
           b : BOOL,
           c : INT(-5..5) ]   ;

CASE u.TAG
   WHEN 'a => u.a := u.a + 1;
   WHEN 'b => u.b := NOT u.b;
   WHEN 'c => u.c := -u.c;
END CASE;

CASE TRUE
   WHEN i>=j => max := i;
   WHEN j>=i => max := j;
END CASE;
```

6.5 REPEAT STATEMENT



The *repeat statement* allows a *body* to be elaborated zero or more times. The for phrase has an index that takes on successive values of the specified subtype. The while phrase is used to achieve conditional repetition.

RULES

For Phrase

The *identifier*, called the index, is defined as a *variable* with the specified *subtype* in the scope of the *repeat statement*. The index is treated as a readonly data item within the *body* (see Section 4.2).

Elaboration of a *repeat statement* with for phrase proceeds as follows. The index is created before the first repetition. During the first repetition, the index has the lowest value of the subtype (i.e., .MIN). For each subsequent repetition, the index will have the successive value of the subtype (via SUCC) until the last value (.MAX) is reached. If REVERSE is specified, the index has the value .MAX for the first repetition, and has successive values (via PRED) until the last value (.MIN) is reached.

While Phrase

The *expression* must have type BOOL. Elaboration of the *repeat statement* with a while phrase repeatedly elaborates the *body* while this *expression* is true. The value of the *expression* is tested prior to each elaboration of the *body*.

NOTES

The index is local to the *repeat statement*, i.e., neither its definition nor its value is directly available outside the *repeat statement*. If the *subtype* of the index has no values the *body* is not elaborated.

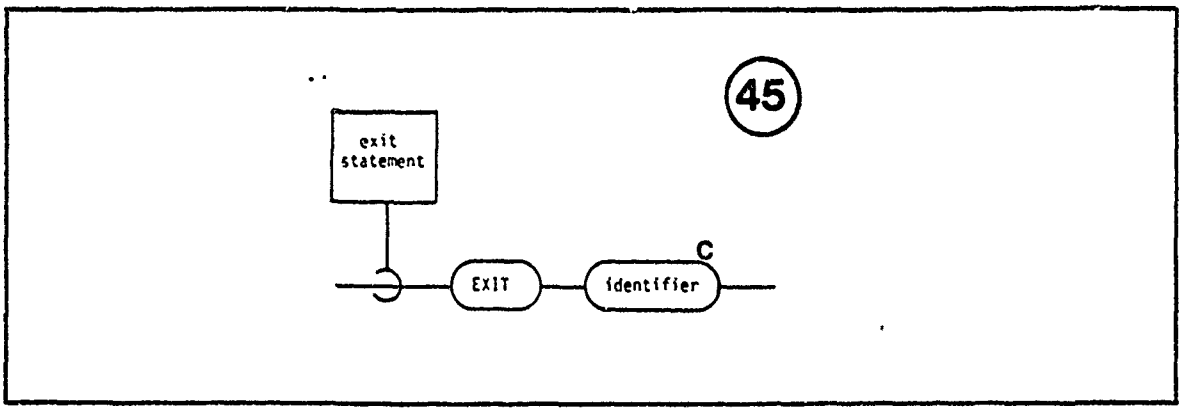
Additional termination conditions can be inserted anywhere within the *body* of a *repeat statement* by the conditional use of an *exit statement* (see Section 6.5).

Use of the *repeat statement* for *user-defined types* is described in Section 13.7.

EXAMPLES

```
r := 0;
q := 0;
WHILE y <= r REPEAT
   r := r - y;
   q := q + 1;
END REPEAT;

VAR a : ARRAY INT(1..n) OF INT(1..n);
FOR i : INT(1..n) REPEAT
   a(i) := i;
END REPEAT;
```

6.6 EXIT STATEMENT



The *exit statement* terminates the elaboration of an enclosing *compound statement*.

RULES

The *identifier* must be known as a matching identifier.

Elaboration of the *exit statement* causes the elaboration of the *compound statement* bracketed by the matching identifier to be terminated.

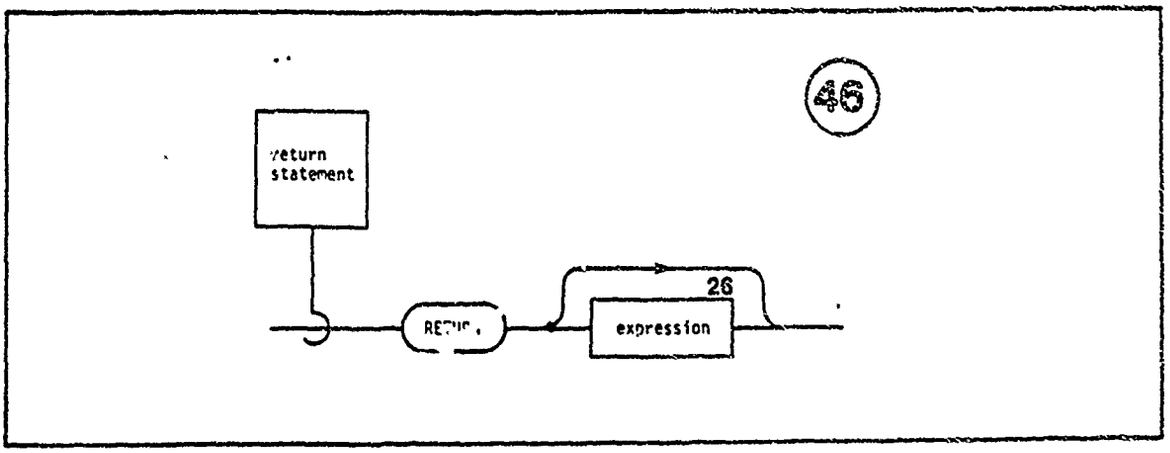
NOTES

An *exit statement* cannot be used to transfer control out of a *compound declaration* (procedure, function, etc), since matching identifiers are never inherited by closed scopes.

EXAMPLES

```
j := Ø;
search FOR i : INT(1..n) REPEAT
   IF a(i) = v THEN
      j := i;
      EXIT search;
   END IF;
END REPEAT search;
```

6.7 RETURN STATEMENT



The *return statement* terminates the invocation of a *compound declaration* (i.e., procedure, function, task, or capsule).

RULES

Elaboration of a *return statement* causes elaboration of the smallest enclosing procedure, function, task, or capsule to be terminated.

Expression must be present if the terminated construct is a function and may not be specified for any other construct. The *type* of the *expression* must be the same as that of the function result (see Section 7.2).

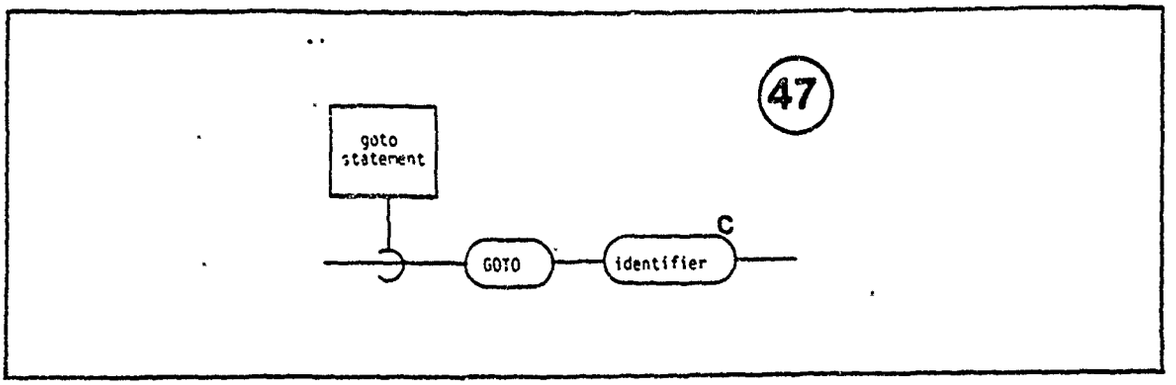
NOTES

A *return statement* is not needed for procedures, tasks, and capsules whose only "return point" is at the end of the *body*. The elaboration of a function must terminate through a *return statement* (or by raising an exception).

EXAMPLE

```
FUNC search (a : ARRAY INT OF INT, v : INT) => INDEXOF(a);
   FOR i : INDEXOF(a) REPEAT
      IF a(i) = v THEN
         RETURN i;
      END IF;
   END REPEAT;
   RAISE search_failure;
END FUNC search;
```

6.8 GOTO STATEMENT



The *goto statement* causes elaboration to continue at a specified *statement*.

RULES

The *identifier* must be known as a goto label. Elaboration of the *goto statement* causes elaboration to continue at the *statement* labeled by the goto label.

NOTES

Because the goto label of the target statement must be local or declared in an enclosing scope, no transfer is allowed into *bodies* or between alternatives of an *if* or *case statement*.

A *goto statement* cannot be used to transfer control out of a *compound declaration* (i.e., procedure, function, etc), since goto labels are never inherited by closed scopes.

EXAMPLE

```
sort :      i : INT(1..n-1) REPEAT
            a(i) > a(i+1) THEN
            CONST t := a(i);
            a(i) := a(i+1);
            a(i+1) := t;
            GOTO sort;
         END IF;
      END REPEAT;
```

7. PROCEDURES, FUNCTIONS, AND PARAMETERS

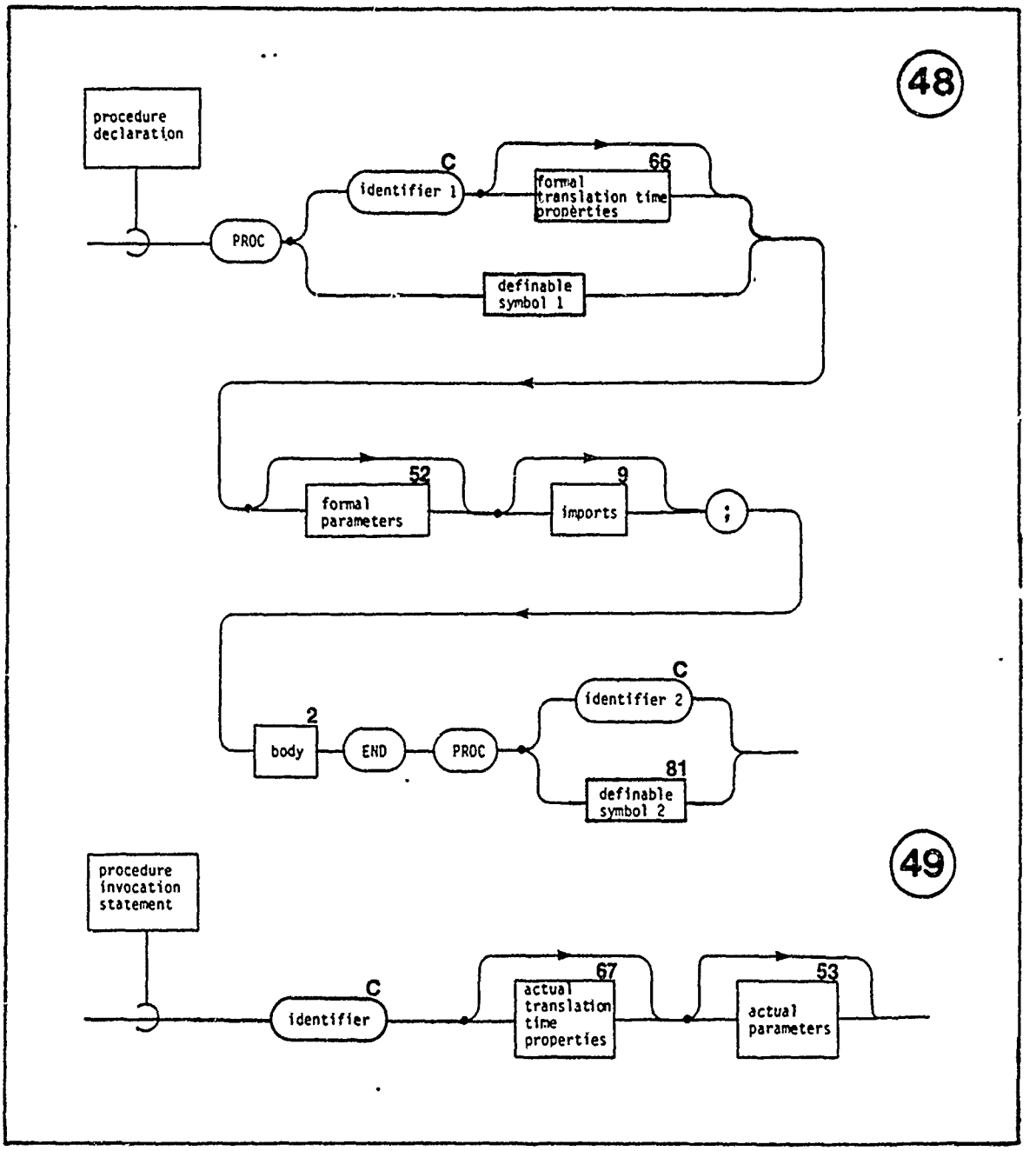
Procedures and functions are major language features for modularizing programs. Procedures and functions are defined by *deferred declarations*. Since procedures and functions are deferred units, they are elaborated when they are invoked, rather than when their *declaration* is encountered.

A procedure is elaborated when a *procedure invocation statement* occurs. A function is elaborated when a function invocation occurs in an expression.

Procedures and functions may be parameterized; when they are, *actual parameters* are bound to corresponding *formal parameters* of the *procedure* or *function declaration* at the time of invocation. This correspondence is based on the positions of the parameters in the *formal* and *actual parameter* lists.

Procedure and *function declarations*, like all *deferred declarations*, are closed scopes. Any *variables* from the enclosing scope used by a procedure or function must be explicitly imported.

7.1 PROCEDURE DECLARATION AND INVOCATION



A procedure is invoked by a procedure invocation statement to perform some action.

RULES

Identifier or definable symbol 1 is defined to be a procedure in the scope in which the *procedure declaration* appears. *Identifier* or *definable symbol 2* must be identical to *identifier* or *definable symbol 1*.

Elaboration of a *procedure invocation statement* proceeds in the following order:

a) *actual parameters* are elaborated;

b) *actual parameters* are bound to the *formal parameters* of the named procedure (see Section 7.3); and

c) the *body* of the named procedure is elaborated.

NOTES

A *procedure declaration* is a closed scope (see Section 3.5); the formal parameter names are defined in this scope.

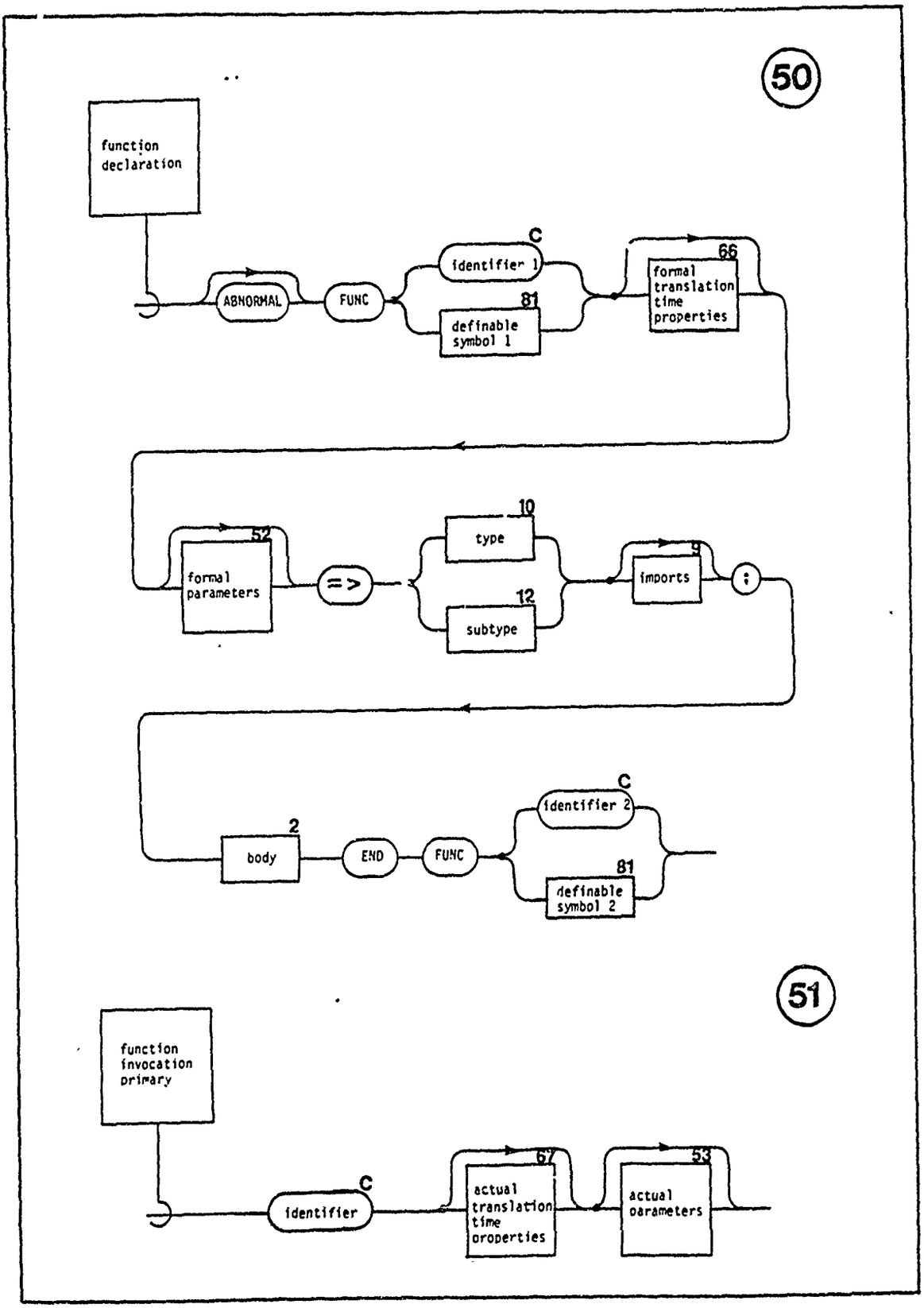
Procedures may be overloaded and may be generic. Overloading, generics, and use of the translation time property list are discussed in Chapter 11 Use of definable symbol names is discussed in Section 13.2.

The order in which *actual parameters* are elaborated and bound is unspecified.

EXAMPLE

```
PROC swap (VAR x,y : INT);
   CONST t := x;
   x := y;
   y := t;
END PROC swap;
```

7.2 FUNCTION DECLARATION AND INVOCATION



A function differs from a procedure in that a function produces a result, a temporary data item (see Section 5.1). This result is obtained during elaboration of the *body* by elaborating a *return* statement which specifies the value of the result to be produced. A function is elaborated when a function invocation occurs as an operand in an expression.

RULES

Identifier or *definable symbol 1* is defined to be a function in the scope in which the *function* declaration appears. *Identifier* or *definable symbol 2* must be the same as *identifier* or *definable symbol 1*.

The *type* or *subtype* following => is known as the result type or subtype. The result type or the type of the result subtype must be assignable.

Elaboration of a *function invocation primary* proceeds in the following order:

a) *actual parameters* are elaborated;

b) *actual parameters* are bound to the *formal parameters* of the named function (see Section 7.3);

c) the *body* of the function is elaborated until a *return statement* is encountered;

d) a result temporary data item is created, whose *subtype* is the result subtype (if specified) or the *subtype* of the *expression* in the *return statement* (if a result type is specified); and

e) the result of the function is produced by assigning (via :=) the value of the *expression* in the *return statement* to a result temporary data item. The := definition used is the one known in the scope where the function is defined.

There must be no way to reach the point immediately after the last *statement* in the *body* of the function, i.e., elaboration must always complete by the use of a *return statement*.

Additional rules for the *return statement* are given in Section 6.7, and (for side-effects and normality of functions) in Section 7.2.1.

NOTES

A *function declaration* is a closed scope (see Section 3.5), the formal parameter names and the result variable are defined in this scope.

The result subtype may depend on the *subtypes* and values of the *actual parameters*.

Functions may be overloaded and may be generic. Overloading, generics, and the use of the translation time property list is discussed in Chapter 11 Use of definable symbol names is discussed in 13.2.

The order in which *actual parameters* are elaborated and bound is unspecified.

EXAMPLE

```
FUNC hypot(side1, side2 : FLOAT) => FLOAT;
   RETURN sqrt (side1**2 + side2**2);
END FUNC hypot;
```

7.2.1 NORMALITY AND SIDE-EFFECTS

There are two kinds of functions: normal and abnormal functions. The result of the invocation of a normal function depends only upon the values of the actual parameters. Several invocations of a normal function with the same actual parameter values may be replaced by the translator with a single invocation (this is called common subexpression elimination). The result of the invocation of an abnormal function can depend upon the values of *variables* other than those in the *actual parameters*. All abnormal functions should be preceded by the reserved word ABNORMAL. Both abbreviations and *types* are assumed to be normal; the result of invoking an abbreviation or *type* should depend only upon the values of the *actual parameters*.

A function is said to have a side-effect if the function modifies any data whose lifetime is longer than the function invocation. Programs are more understandable, reliable, and verifiable when functions have no side-effects. However, there are cases where having side-effects is useful. The language allows normal functions to have side-effects providing these side-effects are restricted to modifications of data items that are local to the body of a capsule in which the function is also local. For normal functions with side-effects, the user must ensure that any common subexpression elimination will not have an undesired effect upon program behavior.

RULES

The order of elaboration within *expressions* is not defined. This means that the order in which side-effects occur within *expressions* is not guaranteed.

If more than one exception could be raised while elaborating an *expression*, which of these exceptions is actually raised is not defined.

If there are several invocations anywhere in a program of a normal function, a *type*, or an abbreviation whose corresponding *actual parameters* have the same value, these invocations may be replaced (by a translator) by a single invocation which occurs at the point of the first of these invocations.

A normal function may have only CONST and READONLY formal parameters.

If a normal function has an imports list, then:

a) the function must be local to a capsule *body*;

b) *variables* imported by the function must be defined locally in the capsule body in which the function is local;

c) no invocations of the function may appear anywhere within the capsule in which the function is defined; and

d) no *variables* imported by a normal function may be exported from the capsule.

NOTES

Failure to mark an abnormal function, or the presence of an abnormal abbreviation or *type*, means that common subexpression elimination may produce undesired results.

The only normal functions which have no parameters are functions which always produce the same constant value.

EXAMPLES

1) A normal function with no side-effects

```
FUNC fact (i : INT) => INT(1..10000);
   IF i = 0 THEN RETURN 1;
            ELSE RETURN i * fact(i-1);
   END IF;
END FUNC fact;
```

2) A normal function, fact, with a restricted side-effect which records the number of times it is called. This count is returned by an abnormal function, factcnt.

```
CAPSULE c EXPORTS fact, factcnt;
   VAR cnt : INT(0..10000) := 0;

   FUNC fact(i : INT) => INT(1..10000) IMPORTS cnt;
   % The result depends only on the value of
   % the parameter, i.
        VAR r : INT(1..10000) := 1;
        cnt := cnt + 1;
        FOR j : INT(1..i) REPEAT
           r := r * j;
        END REPEAT;
        RETURN r;
   END FUNC fact;

   ABNORMAL FUNC factcnt => INT(0..10000) IMPORTS cnt;
   % The result, r, is not computed based only upon the
   % parameters (of which there are none).
      RETURN cnt;
   END FUNC factcnt;
END CAPSULE c;
```

3) A normal function with a restricted side-effect -- a memo function

```
CAPSULE fmemo EXPORTS f;

   VAR oldx : INT(1..100) := 1;
   VAR oldy : INT(1..100) := realf (oldx);

   FUNC f (x : INT(1..100)) => INT(1..100) IMPORTS oldx, oldy;
      IF x /= oldx THEN
         oldx := x;
         oldy := realf (oldx);
      END IF;
      RETURN oldy;
   END FUNC f;

   FUNC realf (x : INT(1..100)) => INT(1..100);
      ...
   END FUNC realf;

END CAPSULE fmemo;
```

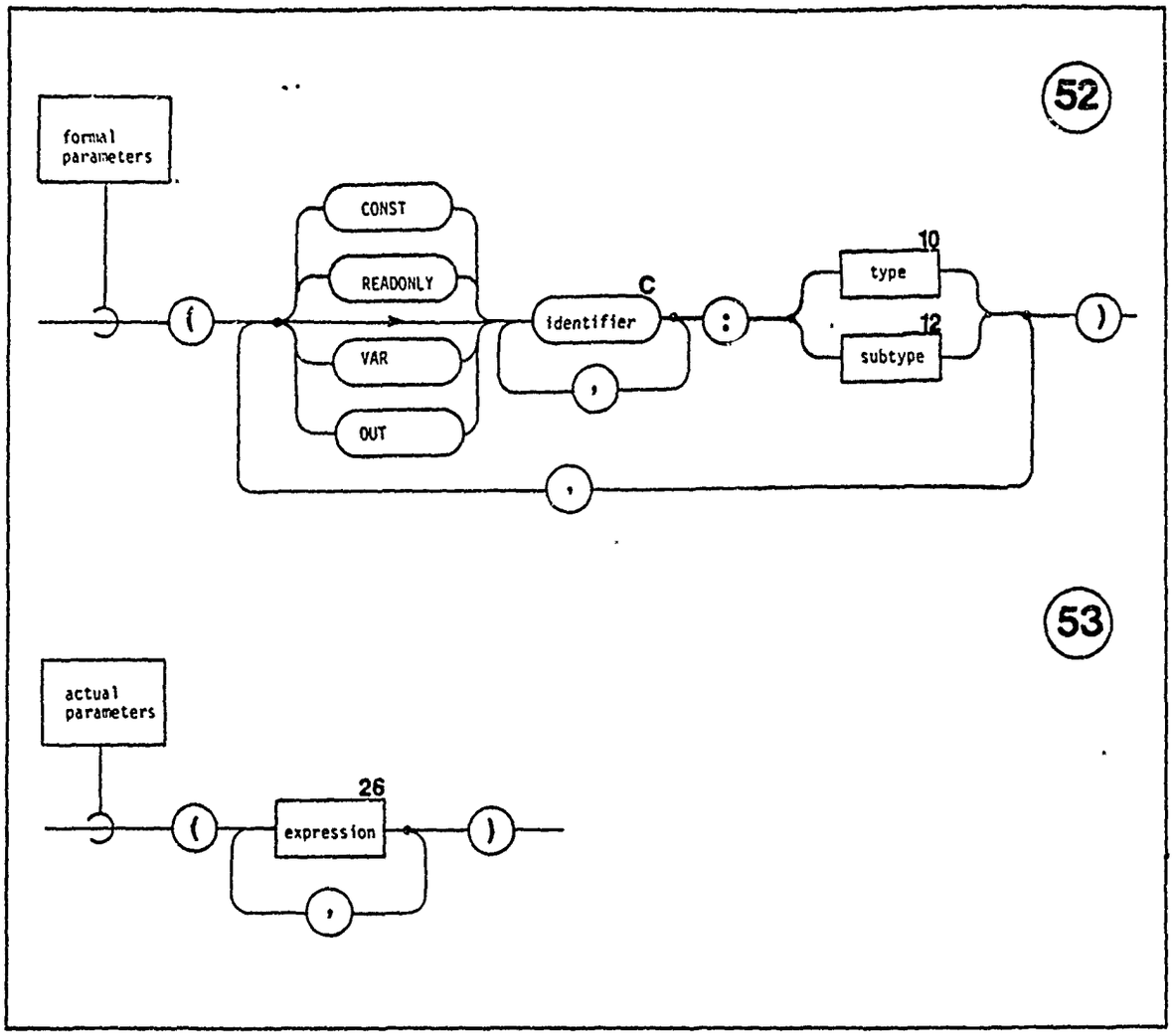
4) An abnormal random number generator function

```
CAPSULE randcap (initial_seed : INT) EXPORTS random;
   VAR seed : INT(-1000..1000) := initial_seed;
   ABNORMAL FUNC random => INT(-1000..1000) IMPORTS seed;
      seed := next(seed);
      RETURN seed;

      FUNC next (i : INT(-1000..1000)) => INT(-1000..1000);
         % computes next random number
      END FUNC next;
   END FUNC random;
END CAPSULE randcap;
```

5) A symbol table

```
CAPSULE symtab EXPORTS look_up;
   CONST size := 500;
   ABBREV sym : STRING[ASCII] (8);

   VAR limit : INT(0..size) := 0;

   VAR table : ARRAY INT(1..size) OF sym;

   ABNORMAL FUNC look_up (s : sym) => INT(0..size)
      IMPORTS READONLY limit, READONLY table;

      FOR i : INT(1..size) REPEAT
         IF table(i) = s THEN
            RETURN i;
         END IF;
      END REPEAT;
      RETURN 0;
   END FUNC look_up;

   FUNC insert (s : sym) => INT(1..size)
     IMPORTS limit, table;

      CONST i := lookup (s);
      IF i /= 0 THEN
         RETURN i;
      ELSE
         limit := limit + 1;
         table (limit) := s;
         RETURN limit;
      END IF;
   END FUNC insert;
END CAPSULE symtab;
```

7.3 FORMAL AND ACTUAL PARAMETERS



Parameters are used to pass information between a specific invocation of a deferred unit (i.e., a procedure, function, task, capsule, type, or abbreviation) and the deferred unit.

When a *deferred declaration* specifies a list of *formal parameters*, each invocation of the deferred unit defined by the declaration must then supply an *actual parameter* for each *formal parameter*. When the deferred unit is invoked, each *formal parameter* is bound to its associated *actual parameter*. The kind of binding is specified for each *formal parameter* by means of a binding class.

There are four binding classes: two for passing information into a deferred unit (CONST and READONLY); one for passing information out (OUT); and one for passing information both in and out (VAR).

RULES

Association of Actual and Formal Parameters

The number of *actual parameters* in an invocation must be equal to the number of *formal* parameters of the invoked deferred unit. *Actual parameters* are associated with *formal parameters* positionally.

Type Checking

Each *actual parameter* must have the same *type* as that specified for the corresponding *formal* parameter (or the *type* of the *subtype* specified).

Binding of Actual Parameters to Formal Parameters

If a *formal parameter* specifies a *type*, it acquires the *subtype* of the *actual parameter* with which it is bound.

If a *formal parameter* specifies a *subtype*, the *formal parameter* has that *subtype*; if the binding class is VAR or READONLY the *actual parameter* must have an equal *subtype* (otherwise the X_SUBTYPE exception is raised).

The binding class is CONST when no binding class is explicitly specified. The actual parameters associated with VAR and OUT formal parameters must be variables. For CONST and OUT, the parameter must have a type for which assignment is defined. Rules for each binding class are given here.

CONST	The formal parameter is a local constant to which the value of the actual parameter is assigned (via :=).
VAR	The formal parameter is a local name for the actual parameter variable.
OUT	The formal parameter is a local variable which is assigned (via :=) to the actual parameter variable upon normal completion (i.e., completion other than as the result of an unhandled exception) of the invocation.
READONLY	The formal parameter is a local name for the actual parameter. The formal parameter is treated as a readonly data item.

For CONST and OUT formal parameters, the definition of assignment used is one known in the scope where the formal parameter definition appears.

Order of Binding

The order in which *actual parameters* are bound to *formal parameters* is undefined. A *subtype* specified for a *formal parameter* may not depend upon the value or *subtype* of other *formal* parameters.

NOTES

Since there are two input binding classes, CONST and READONLY, there are some guidelines on when each should be used.

CONST — The *formal parameter* is a copy of the *actual parameter*. In most cases, this is the correct binding to use, since it prevents the occurrence of aliasing and unintended sharing.

READONLY — The *formal parameter* is a local name for the *actual parameter*. This binding class must be used when

a) the parameter type is a *type* for which assignment is not defined; or

b) sharing is desired (see Section 10.6).

When large objects are to be passed as input, the CONST binding may be less efficient (since copying is involved) than the READONLY binding. However, in many cases, a translator can optimize CONST bindings so that no copy is required.

8. CAPSULES

Capsules are the basic unit of separate translation (see Section 3.1) and can also be nested within other language constructs. Capsules can be used to create common data pools, libraries, and abstract data types, as well as independent, executable programs.

A capsule, like a procedure, is a deferred unit which is invoked. Capsules differ from procedures in that definitions local to a capsule body may be made known outside it. Selected definitions can be made locally known in each scope where the capsule is invoked. The *body* of a capsule can contain *statements* as well as definitions. These *statements* are elaborated to initialize *variables* and *constants* defined in the capsule.

There are two ways in which a capsule can be invoked.

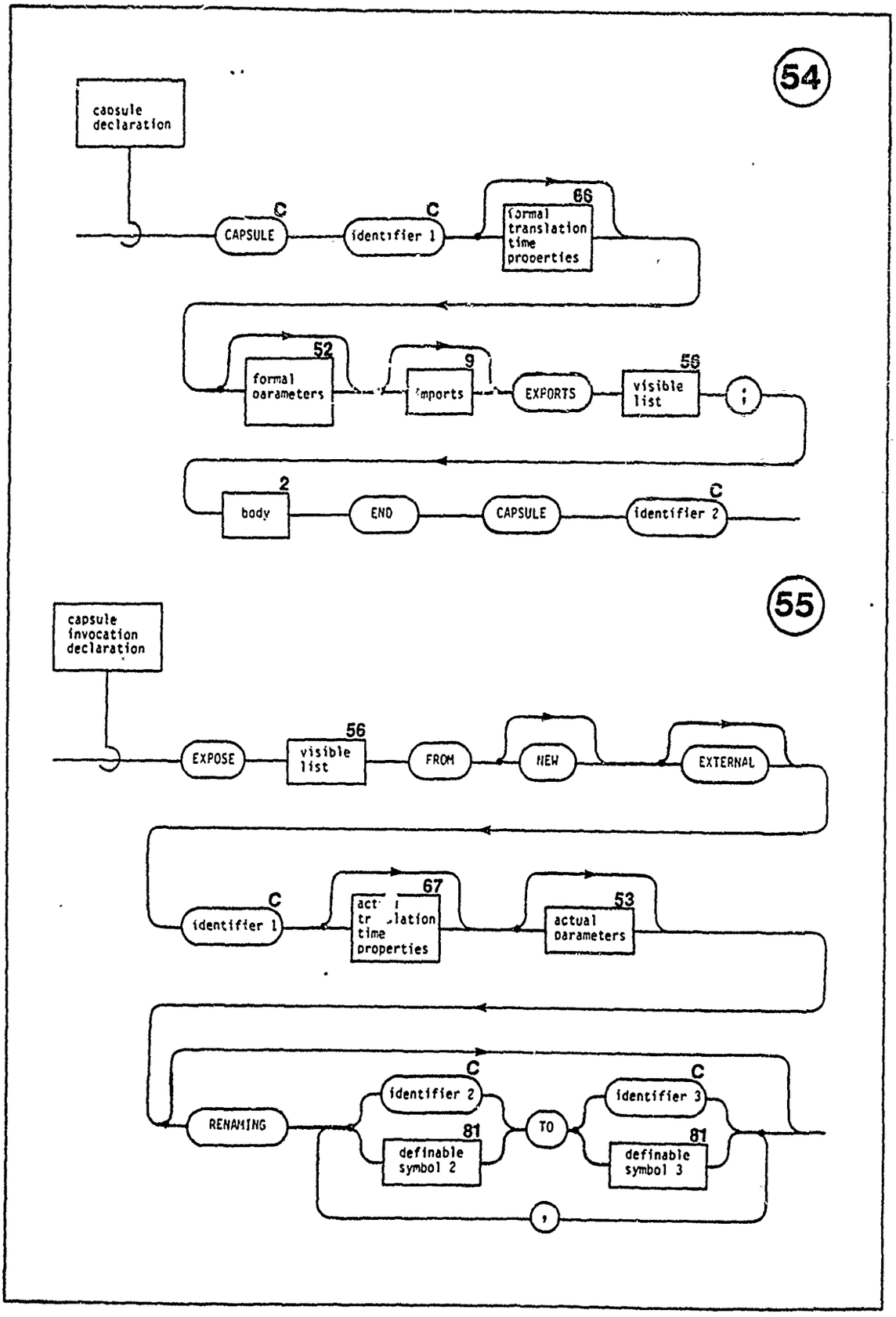
1) It can be invoked as new. Each such invocation will cause the capsule to be elaborated. Local *variable* and *constant declarations* in the capsule *body* create different *variables* and *constants* at each new invocation. Capsules invoked as new may be parameterized. *Actual parameters* are supplied each time a capsule is invoked and serve to specialize the capsule.

2) It can be invoked as old. In this case, all old invocations will reference a single version of the capsule which was elaborated upon entry to the scope where the capsule was defined. Each old invocation will reference the same set of *variables* and *constants* from the capsule. Capsules which are invoked as old may not have parameters.

NOTES

Definitions that are known in a scope come from three sources: local definitions written in the scope, definitions which become locally known by invoking a capsule in that scope, and definitions which are available (either implicitly or through an imports list) from the enclosing scope.

8.1 CAPSULE DECLARATION AND INVOCATION



A *capsule declaration* is invoked to make selected definitions in its *body* known in the scopes where invocations of the capsule appear.

RULES

Capsule Declaration

Identifier 1 in a *capsule declaration* is defined to be a capsule in the scope in which the *capsule* declaration appears. *Identifier 2* must be the same as *identifier 1*.

A *capsule declaration* which is invoked as old or which is a *translation unit* may not have any *formal parameters*. A capsule may not have *formal parameters* of the OUT binding class.

Capsule Invocation Declaration

The capsule with name *identifier 1* is invoked.

The lifetime of *actual parameters* passed to READONLY and VAR formal parameters must be greater than or equal to the lifetime of the *capsule invocation declaration* containing those *actual parameters*.

A *capsule invocation declaration* has two effects: it causes elaboration of a capsule declaration and it makes selected definitions visible. There are two ways in which a capsule may be invoked:

a) As a new invocation (if NEW is specified).
b) As an old invocation (if NEW is not specified).

Elaboration of a new *capsule invocation declaration* consists of

a) elaborating the *actual parameters*;

b) binding the *actual parameters* to the *formal parameters* of the invoked capsule (see Section 7.3); and

c) elaborating the *body* of the invoked capsule.

If there are any old invocations of a capsule, the *body* of the invoked capsule is elaborated once upon entry to the scope in which the *capsule declaration* appears.

A *capsule invocation declaration* makes selected definitions that are locally known in the *body* of the capsule also locally known in the scope where an invocation of the capsule appears (see Section 8.2). For new invocations, these definitions are the ones that have been created during elaboration of the *capsule invocation declaration*. For old invocations these definitions are the ones that were created upon entry to the scope where the *capsule declaration* appears.

The lifetime of any definitions made known by a new *capsule invocation declaration* is equal to the lifetime of the *capsule invocation declaration*. The lifetime of any definitions made known by an old capsule invocation is equal to the lifetime of the declaration of the invoked capsule.

If an invocation includes a RENAMING list, there must be a definition known (as a result of the visible list) for each identifier 2 and definable symbol 2 that appears. For each item in the list of the form

```
name 2 TO name 3
```

all definitions that would be known as name 2 are known instead as name 3 in the scope where the *capsule invocation declaration* appears. If name 3 is a *definable symbol*, all definitions of name 2 must satisfy all restrictions upon definitions of that *definable symbol*.

If EXTERNAL is specified, then *identifier 1* must be the name of some separate *translation unit*. If EXTERNAL is not specified, a definition for *identifier 1* must be known in the scope in which the *capsule invocation declaration* appears.

If there are several *capsule invocation declarations* local to the same scope, then none of these *declarations* may contain any uses of the definitions that it, or any later *capsule invocation* declaration, makes known.

NOTES

A *capsule declaration* is the only language construct to include an exports list.

A *capsule declaration* is a closed scope. The only definitions which are local to a *capsule declaration*, as opposed to the *body* of the capsule, are formal parameter definitions (see Section 3.5).

Variables declared in the capsule that are not exported act as "own" data of the capsule. Like all data in the capsule, such *variables* come into existence each time the *body* of the capsule is elaborated; the *statements* in the capsule body may be used to initialize them.

Translation time property lists are used to overload capsules and to create generic capsules.

EXAMPLES

1) Common group of declarations.

```
CAPSULE device_tables (ntty : INT, nprint : INT) EXPORTS ALL;

   CONST console := 1;

   VAR tty_tab     : ARRAY INT(1 .. ntty)   OF tty_info;
   VAR printer_tab : ARRAY INT(1 .. nprint) OF print_info;

   FOR i : INT(1 .. ntty) REPEAT
      init_tty (tty_tab(i), i);
   END REPEAT;

   FOR i : INT(1 .. nprint) REPEAT
      init_printer (printer_tab(i), i);
   END REPEAT;

END CAPSULE device_tables;

...
% Typical use
EXPOSE ALL FROM NEW device_tables (10,2);
```

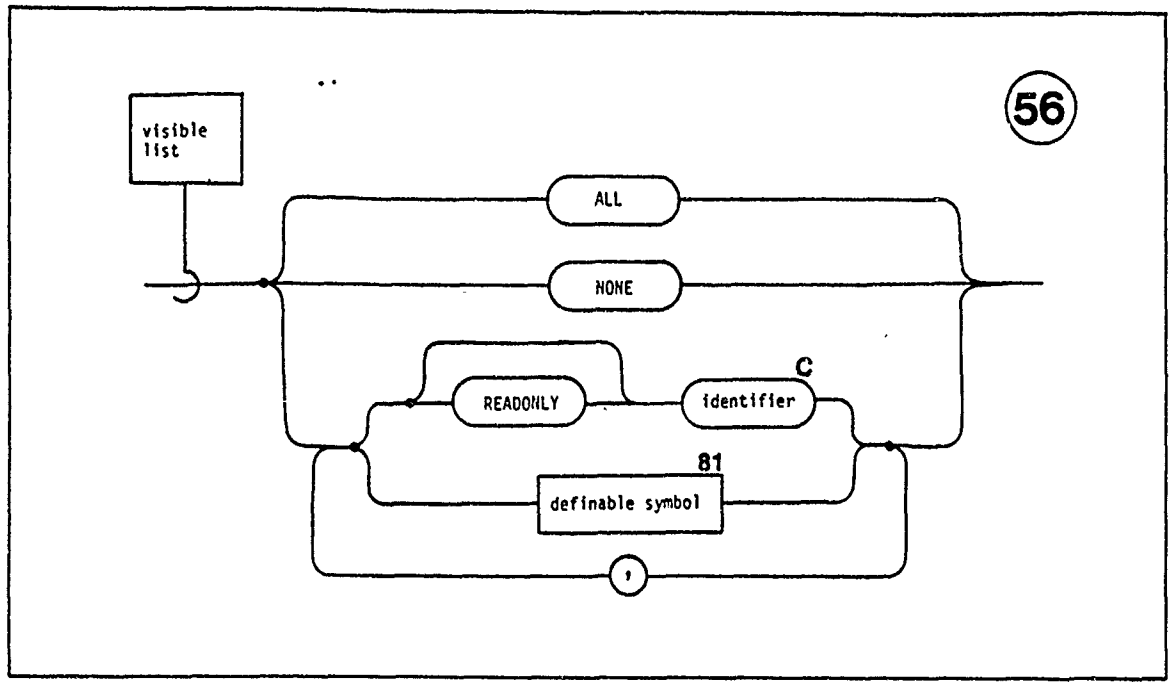
2) Abstract data type example.

```
CAPSULE queues EXPORTS queue, init_queue, enter, remove;

   ABBREV.elem : INT(0..255);

   TYPE queue (n : INT) : RECORD[ first,last :  INT(0 .. n-1),
                                  num        :  INT(0 .. n),
                                  items : ARRAY INT(0 .. n-1)
                                          OF elem];

   PROC init_queue (VAR q : queue);
      ASSERT q.n > 0;
      q.first := 0;
      q.last  := 0;
      q.num   := 0;
   END PROC init_queue;

   PROC enter (VAR q : queue, r : elem);
      ASSERT q.num < q.n;
      q.items(q.last) := r;
      q.last := (q.last + 1) MOD q.n;
      q.num := q.num + 1;
   END PROC enter;

   PROC remove (VAR q : queue, OUT r : elem);
      ASSERT q.num > 0;
      r := q.items(q.first);
      q.first := (q.first + 1) MOD q.n;
      q.num := q.num - 1;
   END PROC remove;

END CAPSULE queues;
```

3) Exposing a definition makes the definition locally known and, thus, able to be exported.

```
CAPSULE c EXPORTS x;
   ...
   VAR x ...
   ...
END CAPSULE c;

CAPSULE b EXPORTS x,y;        % only locally known definitions
                              % can be exported.
   VAR y ...
   EXPOSE x FROM c;           % makes x locally
                              % known in b
END CAPSULE b;
```

8.2 VISIBLE LIST



Visible lists control the availability of the definitions that are local to the *body* of a capsule, to those scopes where the capsule is invoked. *Visible lists* appear in two places:

a) After the word EXPORTS in the header of a *capsule declaration*. This *visible list* exports selected definitions, which are local to the *body* of the capsule, to invocations of the capsule.

b) After the word EXPOSE in a *capsule invocation declaration*. This *visible list* makes selected definitions which were exported from the invoked capsule, known in the scope where the *capsule invocation declaration* appears.

RULES

Visible List in a Capsule Declaration

If ALL is specified, all definitions which are local to the *body* of the capsule, except goto label definitions, are exported.

If **NONE** is specified, no definitions are exported.

If a list is specified, all definitions which are local to the body of the capsule, and whose names appear in the list, are exported. Names of goto labels may not appear. There must be a definition, local to the *body* of the capsule, of each name that appears in the list. The name of any variable definition may be preceded by *READONLY*; in this case the *variable* is treated as a readonly data item in those scopes where it is exposed.

Visible List in a Capsule Invocation Declaration

If ALL is specified, all definitions exported by the invoked capsule are made known.

If NONE is specified, no definitions are made known.

If a list is specified, all definitions which were exported from the invoked capsule and whose names appear in the list are made known. There must be an exported definition for each name that appears in the list. The name of any variable definition may be preceded by *READONLY*; in this case the *variable* is treated as a readonly data item in the scope where the capsule invocation appears.

NOTES

The capability of specifying ALL in a visible list makes it easy to create common data pools and libraries. Exporting no definitions is useful for main translation units. Exporting some definitions is useful for the creation of abstract data types.

When a *type* is made visible, assignment and selection operations are not automatically made visible. However, *attribute inquiry* for that *type* is automatically made visible when the *type* is made visible.

Capsule formal parameters and definitions which are available from the enclosing scope, may not be exported since they are not local to the capsule body.

The visible list of the *capsule invocation declaration* provides a convenient method of access control. For example, suppose the capsule math_library has been defined as a library of mathematical functions. In one scope only some of the functions may need to be known. The capsule might be invoked as

```
EXPOSE integrate, mean FROM math_library;
```

In some other scope a different set of functions might be needed. The invocation there might be

```
EXPOSE sin, cos FROM math_library;
```

9. EXCEPTION HANDLING

A major design criterion for this language is that it contribute toward the reliability of the systems that it is used to develop. Toward this end, a language facility, called exception handling, is provided so that a user can gain control and take appropriate action when a runtime error occurs. Both user-defined and language-defined runtime errors can be handled in this way.

There are three parts to exception handling: the definition of an exceptional condition, called an exception; the occurrence, or raising, of the exception; and the handling of the raised exception.

Exceptions are either language-defined (see Appendix D) or defined by the user in an *exception* declaration. Exception names follow normal scope rules. Exceptions are raised explicitly by the elaboration of a raise or reraise statement. Language-defined exceptions are also raised automatically when an exceptional condition occurs during elaboration. The handling of an exception is achieved by the *guard statement*, which allows the user to gain control when an exception is raised. The *guard* statement consists of two parts: a guarded body in which an exception might be raised; and a set of handlers which can handle exceptions raised in the guarded body. Separate handlers can be provided for specific exceptions, and a general handler can be provided for all exceptions not handled separately. When an exception having a handler is raised in the guarded body, the elaboration of the guarded body is terminated and the *body* of the handler is elaborated. Elaboration of the *guard* statement is completed when elaboration of the handler is completed.

Guard statements may be nested within one another. When an exception is raised, the *guard* statement containing the guarded body in which the exception is raised is examined first. If it does not contain a handler for that exception, then enclosing *guard statements* are examined for the appropriate handler, starting with the innermost. The *guard statement* selected must meet two criteria: it must contain a handler for this specific exception; and it must not contain a *deferred* declaration which contains the raised exception.

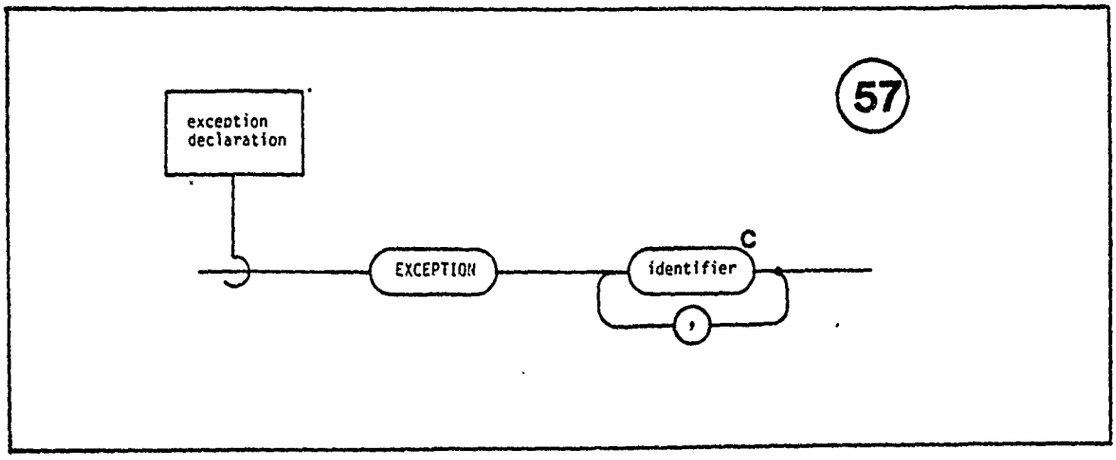
If no enclosing *guard statement* is found before an enclosing *deferred declaration* is found, for all *deferred declarations* except tasks, the search for a handler for the exception continues in the scope containing the invocation of the deferred unit. In the case of tasks, the task activation is terminated and no further searching occurs.

If the search for a handler causes completion of elaboration of the scope in which the exception name is defined, the exception name is changed to X_UNHANDLED and the search for the X_UNHANDLED exception begins.

It is possible, within a handler, to reraise the exception which caused the elaboration of the handler. This allows a local action to be taken before searching resumes for another handler for the same exception.

When efficiency of generated code is more important than the guarantee of reliability, the suppress pragmat can be used to suppress the raising of exceptions (see Appendix B).

9.1 EXCEPTION NAMES



Exception names are defined by the *exception declaration.*

RULES

Each *identifier* is defined as an exception in the scope immediately containing the *exception* declaration.

NOTES

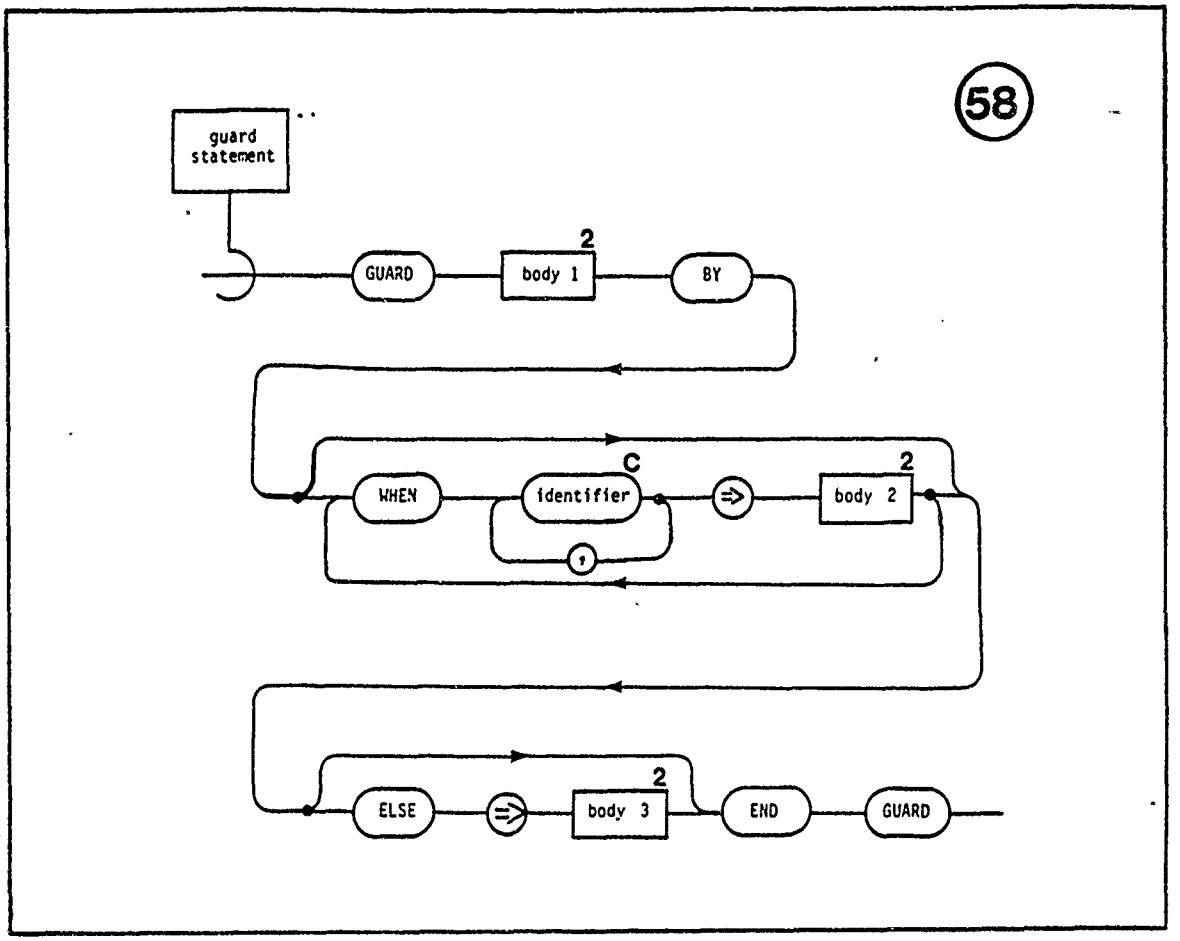
Exception names have the same scope rules as all other names (see Section 3.5).

Language-defined exceptions (see Appendix D) are predefined. No user-written exception declaration is needed.

EXAMPLES

```
EXCEPTION stack_overflow, stack_underflow;
```

9.2 GUARD STATEMENT



A *guard statement* allows the user to gain control and take appropriate action when an exception is raised.

RULES

Each *identifier* between WHEN and => must be known as an exception. All *identifiers* in a *guard* statement must be distinct.

Body 1 is known as the guarded body of the guard statement. Each *body 2* is a handler for the list of exceptions following the preceding WHEN. *Body 3* following ELSE => is a handler for all exceptions not otherwise handled.

Elaboration of a *guard statement* consists of elaboration of the guarded body.

NOTES

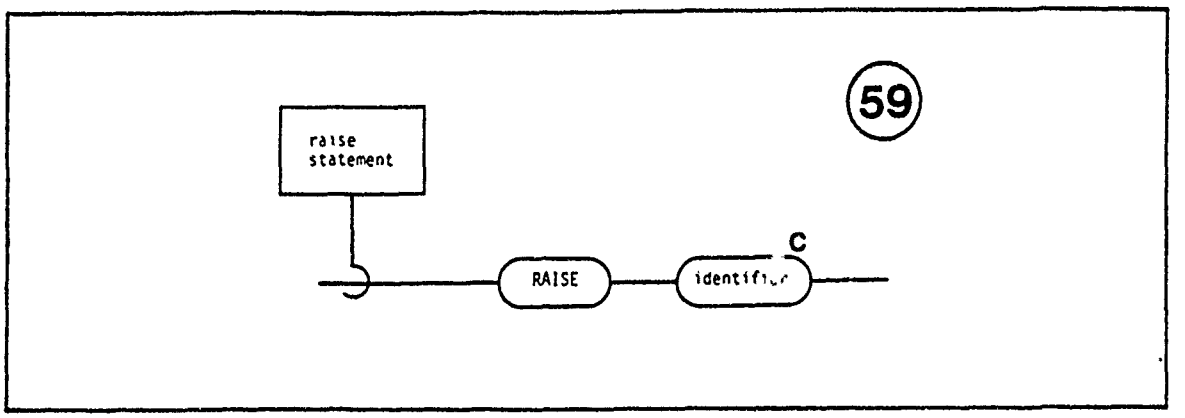
An exception may only be explicitly referenced in a *guard statement* if the *guard statement* is in a scope in which the exception name is known.

The semantics for finding a handler when an exception is raised is described in Section 9.3.

EXAMPLE

```
GUARD
   OPEN(infile, "file1", 'OLD);
BY
   WHEN X_FILENAME =>
            report( "Bad name-file1" );
   WHEN X_NOFILE =>
            report( "File1 does not exist" );
   WHEN X_FILE =>
            report( "Attempt to open infile twice" );
   ELSE =>
            report( "Unknown error when opening infile" );
END GUARD;
```

9.3 RAISING OF EXCEPTIONS



The *raise statement* can be used to raise an exception.

RULES

The *identifier* must be known as an exception.

When an exception is raised, a search is made for the smallest enclosing:

a) guarded body of a *guard statement*;
b) *deferred declaration*; or
c) *body* in which the exception is defined.

If the *body* of a *guard statement* is found and that *guard statement* has a handler for that exception (either specifically or via an ELSE clause), elaboration of the guarded body is terminated and the handler for that exception is elaborated. If the *guard statement* does not have a handler for that exception, the elaboration of the *guard statement* is terminated and that exception is reraised at the place where the *guard statement* appears.

If a *deferred declaration* which is not a *task declaration* is found, the invocation of the *deferred* declaration is terminated and the exception is reraised at the point of invocation of the deferred unit. If a *deferred declaration* is found which is a *task declaration*, the task activation is terminated.

If the *body* is found in which the exception is defined, the X_UNHANDLED exception is raised.

NOTES

An exception may only be raised where its name is known. Certain language-defined operations may raise language-defined exceptions.

The translator will issue a warning message if it discovers that an exception will always be raised at runtime. The translator will produce a list, for each *deferred declaration*, of exceptions which could be raised but not handled when that *deferred* declaration is invoked. It is possible to raise the X_TERMINATE exception in another activation (see Section 10.2).

EXAMPLES

1) Handling an exception raised in an invoked function.

```
FUNC sqrt (x : FLOAT) => FLOAT;
   ASSERT x >= 0.0;
   ...
END FUNC sqrt;
...
GUARD
   q := sqrt (r);
BY
   WHEN X_ASSERT => q := 0.0;
END GUARD;
...
```

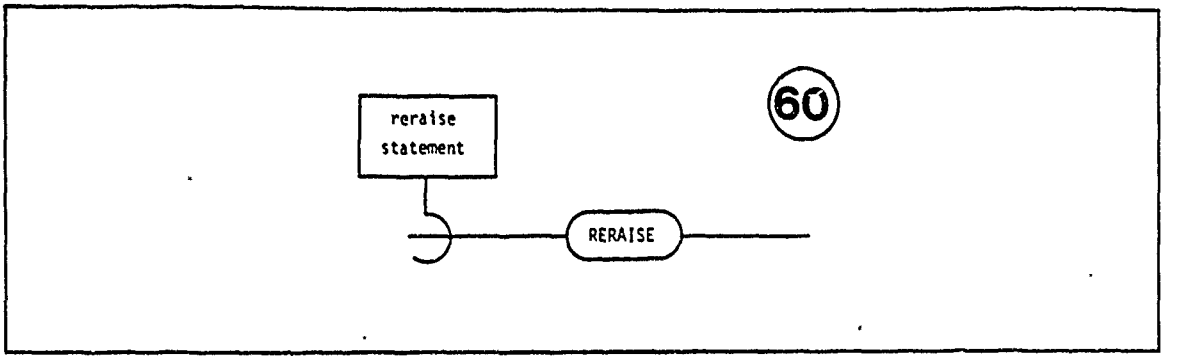
2) Given two procedures, action1 and action2, which both do the same thing; first try action1 and, if it fails, then try action2.

```
GUARD
   action1;
BY
   ELSE => action2;
END GUARD;
```

3) Changing to a more meaningful exception.

```
GUARD
   insert (table, new_entry);
BY
   WHEN X_ASSERT => RAISE table_error;
END GUARD;
```

9.4 RERAISING EXCEPTIONS



The *reraise statement* permits some action, such as clean-up or statistics gathering, to be taken when an exception is raised, before the exception propagates outside the *guard statement*.

RULES

The *reraise statement* must be contained in the *body* of a handler of a *guard statement*.

Elaboration of the *reraise statement* is equivalent to elaboration of

```
RAISE x;
```

where x is the exception being handled.

NOTES

Since an exception must be raised during elaboration of a guarded body of a *guard statement* in order to be handled by the handler of that *guard statement*, the reraising of an exception in a handler does not cause recursive elaboration of the same handler

EXAMPLES

1) How to do local cleanup when an exception is raised.

```
BEGIN
   OPEN (infile, "XYZ", 'NEW);
   GUARD
      % Process infile
      ...
   BY
      ELSE =>
         CLOSE (infile, 'DELETE);
         RERAISE;
   END GUARD;
   CLOSE (infile, 'SAVE);
END;
```

2) How to retry an action n times before failure occurs.

```
retry FOR i : INT(1 .. n) REPEAT
   GUARD
      action;
      EXIT retry; % successful completion
   BY
      ELSE =>
         IF i = n THEN
            RERAISE;
         END IF;
   END GUARD;
END REPEAT retry;
```

10. MULTITASKING

The multitasking facilities provide a means for scheduling and synchronizing multiple concurrent elaborations. The basic unit of multitasking is a task, which is defined by a *task declaration*. A task is invoked using the *task invocation statement* which produces an activation of the task. Each activation is "named" by a unique activation variable. Elaboration of task activations is under the control of schedulers which determine when elaboration of each activation can proceed.

Tasks can communicate in two basic ways: by message passing and by the use of shared variables. Message passing works well for distributed systems and contributes to program reliability. Several task activations can communicate via shared memory simply by importing the same variables, or by passing these variables as VAR or READONLY parameters. No automatic mutual exclusion is provided for shared variables; this must be accomplished by the user. The region statement is provided for this purpose.

Clocks and delays are also provided, both for real time and for activation times.

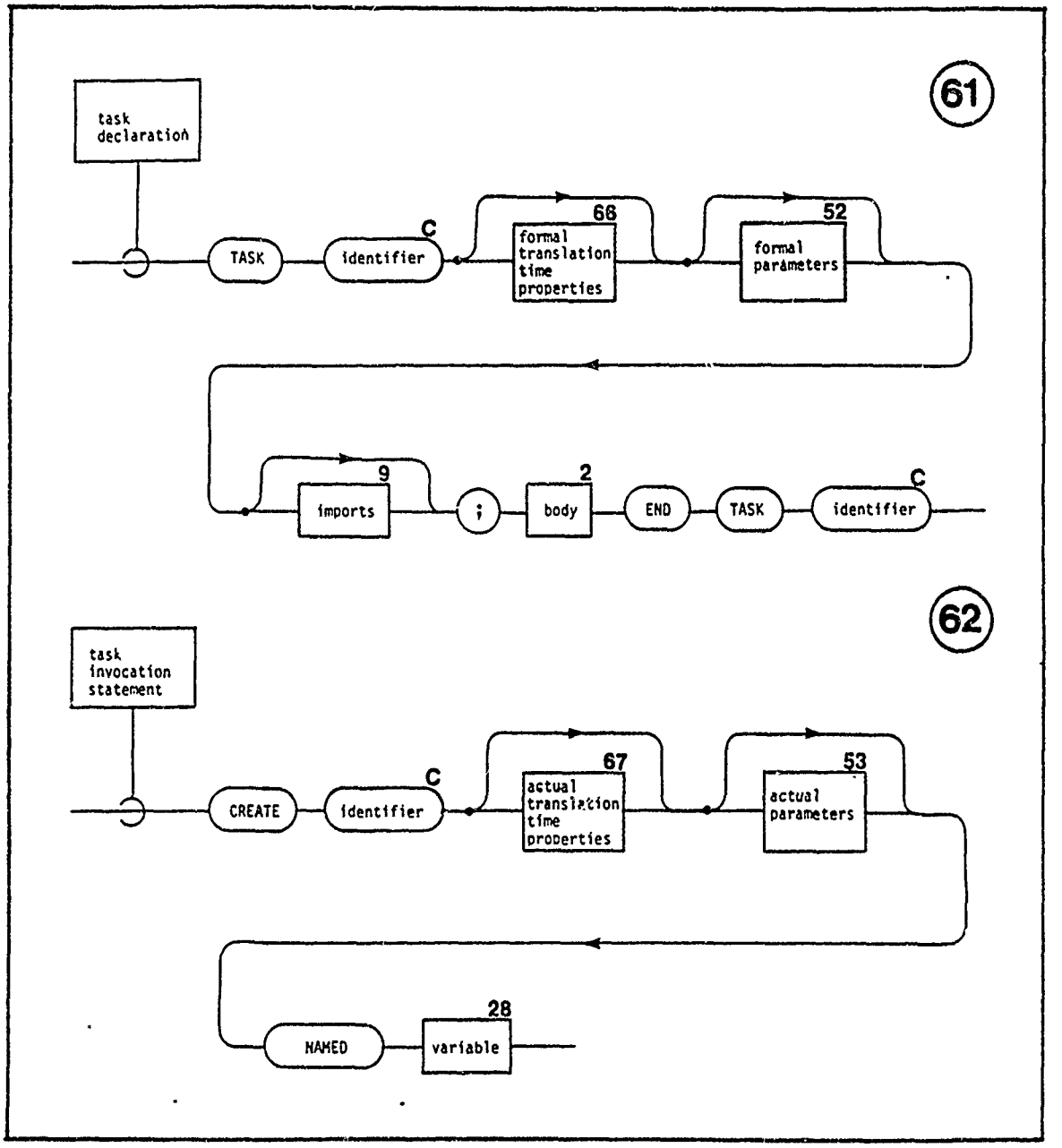
Non-busy multi-way waiting is available to wait for messages and for delays.

There are two levels of multitasking facilities

1) High-Level - These facilities will be used for most applications. Included here is a priority scheduler (via ACT variables), message passing (via MAILBOXes), and mutual exclusion (via DATA_LOCKs). These facilities are described in this chapter.

2) Low-Level - These facilities are provided to allow system programmers to define new schedulers and synchronization schemes for applications where the standard high-level facilities are not appropriate. Once defined, these new facilities can be used by application programmers in a manner that is similar to that used for the built-in high-level facilities. Low-level facilities are described in Chapter 14. Included there is a detailed description of the semantics of the create, wait, and region statements. Also included is a discussion of the LATCH data type, the low-level details of the ACT priority scheduler, and of techniques for handling hardware interrupts.

10.1 TASK DECLARATION, TASK CREATION, AND ACTIVATION VARIABLES



A *task declaration* is similar to a *procedure declaration.* Tasks, like procedures, are elaborated only when invoked. A task is invoked by a *task invocation statement* which creates an activation of the task which is elaborated concurrently with the elaboration of its invoker. When a task is invoked, an activation variable is specified as a way of "naming" that activation. All activations are named. For example,

```
TASK t (i : INT);
   ...
```

```
END TASK t;

VAR av : ACT;              % an activation variable
...
CREATE t(3) NAMED av;      % invokes t with actual parameter 3
                           %  and associates activation
                           %  variable av with this
                           %  activation of t
```

In addition to "naming" the activation, the activation variable determines which scheduler is to control the activation. The type of the activation variable is used in this determination. The ACT type selects the built-in priority scheduler discussed in the next section. Other activation variable types can also be defined for other kinds of user-defined schedulers (see Section 14.5). For example, a task may be scheduled with a user-defined round robin scheduler as follows.

```
TASK t;
   ...
END TASK t;

VAR arr : RR_ACT;

CREATE t NAMED arr;
```

An activation variable can be either active or inactive. All activation variables are initialized to be inactive. When a task activation is created, the activation variable which names the activation is changed from inactive to active. When the activation is complete, the *variable* is changed back to inactive.

An activation having an active activation variable can be either eligible to run or waiting. When an activation is created, it is eligible to run. Some operations (e.g., a DELAY) will cause an activation to wait.

Each step in the elaboration of a program is part of some activation. When a program is run, the system creates a single main activation which elaborates the *body* of the main capsule. Any activation can create other activations by elaborating a *task invocation statement.*

The language ensures that an activation of a task will not run longer than lifetime of the task's declaration. When the scope in which a task is declared is about to be left, the current activation waits until all activations of the task are complete.

RULES

Task Declaration

Identifier 1 is defined to be a task in the scope in which the *task declaration* appears. *Identifier 2* must be the same as *identifier 1*.

A task may not have OUT formal parameters.

The lifetime of a task begins at the beginning of elaboration of the scope in which it is declared and ends at the end of elaboration of the scope in which it is declared.

When an activation is about to leave the scope of a *task declaration*, it waits for all the activation variables associated with activations of that task to be inactive.

Task Invocation

Elaboration of a *task invocation statement* consists of

a) elaboration of the *actual parameters*;

b) binding of the *actual parameters* to the *formal parameters* of the named task (see Section 7.3);

c) preparing the activation variable to elaborate the *body* of the task; and

d) changing the *variable* from inactive to active. If the *variable* is not inactive, then the X_CREATE exception is raised.

The lifetime of the activation variable in a *task invocation statement* must be greater than or equal to the lifetime of the invoked task.

Any *actual parameters* passed either READONLY or VAR must have a lifetime greater than or equal to that of the invoked task.

NOTES

A more detailed description of the semantics of the *task invocation statement* can be found in Section 4.5.3

10.2 THE ACT PRIORITY SCHEDULER

There is a built-in priority scheduler. Elaboration of all task activations whose activation variables have type ACT is controlled by this scheduler. This section discusses the high level operations for the ACT scheduler. Low level ACT operations are discussed in Chapter 14. Techniques for defining other schedulers are discussed in Section 14.5.

ME

The result of the ME function is the activation variable of the activation that invokes ME.

Priorities

Scheduling of task activations whose activation variables have type ACT is determined based on priorities. Each activation has a priority which is an integer with subtype

```
INT(0..255)
```

Priority 0 is the lowest priority (the priority least likely to be scheduled) and priority 255 is the highest priority (the priority most likely to be scheduled). The priority of an activation, av, can be obtained by invoking the function

```
PRIORITY (av)
```

whose result is the priority of av.

The priority of any activation, av, can be set to value n, by invoking the procedure

```
SET_PRIORITY (av,n);
```

The initial priority of the main activation of a program is set by the user when the program is to be run. If no priority has been explicitly set, the initial priority of other activations is equal to the current priority of the creating activation.

Exterminate

Elaboration of the procedure invocation

```
EXTERMINATE (a);
```

will cause the X_TERMINATE exception to be raised in the activation currently associated with activation variable a. The invocation has no effect if a is inactive. If a is waiting, then it becomes eligible to run.

Scheduling Algorithm

At any time, there will be some activations which are eligible to be run. The ACT scheduling algorithm decides which of the activations are to be run. The language makes no assumptions about the number of activations which can be run concurrently. On some target systems, at most one activation will be running while on other systems, several activations can be running concurrently.

For the set of activations which are eligible to run, an activation with a higher priority will be

scheduled before an activation with a lower priority and, for activations having the same priority, those activations which have been eligible for the longest time will be scheduled over the other activations.

NOTES

ME, PRIORITY, SET_PRIORITY, and EXTERMINATE are described in detail under the ACT type in Appendix C.10. The scheduling algorithm is described in detail in Section 14.2.

EXAMPLES

1) Setting priorities

```
TASK t;
   ...
   IF important THEN
      SET_PRIORITY( ME, PRIORITY(ME) + 10 );
   ELSE
      SET_PRIORITY( ME, 10);
   END IF;
   ...
END TASK t;

VAR ta : ACT;
...
SET_PRIORITY( ta, 10 );
CREATE t NAMED ta;
```

10.3 MESSAGE PASSING USING MAILBOX VARIABLES

Message passing is done via mailboxes. Activations can send messages to a mailbox, and other activations can then receive these messages from the mailbox. A mailbox is a *variable* having a mailbox type. For example,

```
VAR m : MAILBOX[ STRING[ASCII](4) ] (3) ;
```

Here, m is a mailbox of size 3 capable of holding messages, each having the subtype STRING[ASCII](4). The size specifies that, at any time, up to 3 messages could have been sent but not yet received.

A mailbox is initially empty (i.e. holds no messages). The SEND procedure is used to send a message to a mailbox. For example,

```
SEND(m,"MES1");
```

sends the message "MES1" to mailbox m. Additional messages can be sent to m by additional sends. for example,

```
SEND(m,"MES2");
SEND(m,"MES3");
```

The RECEIVE procedure is used to receive a message from a mailbox. For example,

```
VAR v : STRING[ASCII] (4);
...
RECEIVE(m,v);
```

will place the next available message from mailbox m into variable v. Messages are stored in a mailbox in order of arrival so that the first message to be received from a mailbox will be the first message sent to the mailbox. In the above example the value of v after invoking RECEIVE would be "MES1".

When a mailbox becomes full, no more messages can be sent. If an attempt is made to send a message to a full mailbox, then the sender will wait until the mailbox is no longer full. If there is more than one activation waiting as a result of attempting to send a message to a full mailbox, these senders are queued in the order in which the sends were done. The first sender will therefore be the first to complete the send. A similar queueing occurs when receives are attempted on an empty mailbox.

Mailboxes with Size 0

When the size of a mailbox is 0, the sender can never get ahead of the receiver. For example,

```
VAR m1 : MAILBOX[ STRING[ASCII](4) ] (0);
```

In this case, the SEND(m1, "ABCD") will wait unless there is some receive request outstanding. In this latter case, SEND(m1, "ABCD") will send the message "ABCD" directly to the requesting receiver.

RULES

Messages for mailboxes must have an assignable type.

Messages sent to a mailbox and received from that mailbox are handled on a first in (i.e., first sent to the mailbox) first out (i.e., first received from the mailbox) basis. In particular, the i'th receive will get the message from the i'th send.

NOTES

The SEND and RECEIVE operations can also be used as waiting invocations in the *wait statement* (see Section 10.5).

In a typical program structure, each mailbox represents a service which can have several "server" activations that actually do the work. This structure allows the requester to be ignorant of the actual number of activations providing the service, and their identities.

More details about the MAILBOX type can be found in Section 14.1.

The MAILBOX type, along with the SEND and RECEIVE procedures are described in detail in Appendix C.11.

EXAMPLES

1) Simple producer-consumer.

```
VAR m : MAILBOX[s] (5);

TASK produce IMPORTS m, infile,  READONLY sdone;
   VAR data : s;
   WHILE NOT EOF(infile)  REPEAT
      READ(infile, data);
      SEND(m, data);
   END REPEAT;
   SEND(m, sdone);
END TASK produce;

TASK consume IMPORTS m, outfile,  READONLY sdone;
   VAR data : s;
   WHILE TRUE REPEAT
      RECEIVE(m,data);
      IF data = sdone THEN
         RETURN;
      END IF;
      WRITE(outfile, data);
   END REPEAT;
END TASK consume;

VAR pr, cs : ACT;

CREATE produce NAMED pr;
CREATE consume NAMED cs;
```

10.4 CLOCKS AND DELAYS

There are two basic kinds of clocks: a single real-time clock and a clock for each activation. The real-time clock measures the elapsed real-time since the program began to run. An activation clock measures the total real-time that a particular activation has been actually running since it was created. All times are positive integers and are measured in ticks. Ticks are an implementation-dependent unit. There are standard integer configuration constants

```
MILLISECONDS
SECONDS
MINUTES
HOURS
```

whose values are the (closest integer to the) number of ticks that occur in each millisecond, second, minute, and hour. The function

```
TIME
```

returns the value of the real time clock in ticks. The function

```
TIME(a)
```

returns the value of the activation clock for activation a in ticks.

An activation can be delayed for t ticks of real time by elaborating

```
DELAY(t);
```

An activation can be delayed until the value of the real time clock is t by elaborating

```
DELAY_UNTIL(t);
```

An activation can be delayed until the value of the activation clock for activation a has the value t by elaborating

```
DELAY_UNTIL(t,a);
```

An activation can be delayed until some activation variable a becomes inactive by elaborating

```
DELAY_UNTIL_INACTIVE(a);
```

NOTES

The TIME, DELAY, DELAY_UNTIL, and DELAY_UNTIL_INACTIVE procedures are described in Appendix C. The configuration constants MILLISECONDS, SECONDS, MINUTES, and HOURS are described in Section 12.1.

EXAMPLES

1) Delayminsec waits for i minutes plus j seconds of real time.

```
PROC delayminsec (i,j : INT);
   DELAY (i*MINUTES+j*SECONDS);
END PROC delayminsec;
```

2) Measuring the total time in seconds spent in running a procedure p.

```
CAPSULE c EXPORTS p,ptime;
   VAR total_time : INT(0..1000000) := 0;
   PROC p IMPORTS total_time;
      VAR enter : INT(0..1000000) := TIME(ME);
      ...
      total_time := total_time + (TIME(ME) - enter) DIV
                                          SECONDS;
   END PROC p;

   ABNORMAL FUNC ptime => INT(0..1000000)
        IMPORTS total_time;
      RETURN total_time;
   END FUNC ptime;
END CAPSULE c;
```

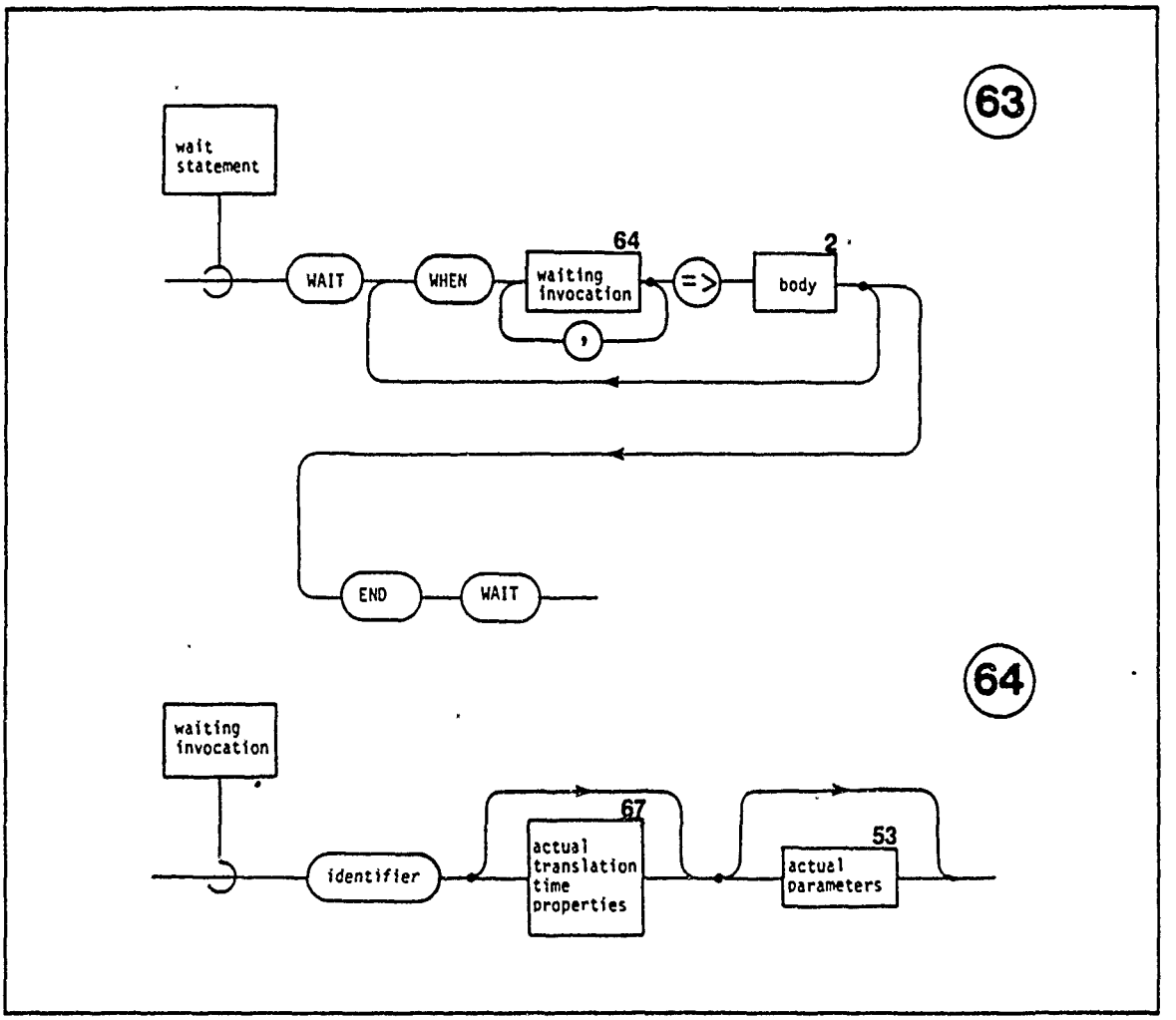
3) Performing an action every i seconds of real time

```
VAR t : INT(0..1000000) := TIME;
WHILE TRUE REPEAT
   action;
   t := t+i*SECONDS
   DELAY_UNTIL (t);
END REPEAT;
```

4) Reusing an activation variable.

```
BEGIN
   TASK t;
      ...
   END TASK t;

   VAR a : ACT;

   WHILE NOT done REPEAT
      CREATE t NAMED a;
      ...
      DELAY_UNTIL_INACTIVE(a);
   END REPEAT;
END;
```

10.5 WAITING



The *wait statement*, like a *case statement*, contains a sequence of when clauses. A *wait statement* differs in that *waiting invocations* are specified instead of *expressions*. A *wait statement* permits waiting until one of the set of *waiting invocations* completes and, when it completes, elaborating the *body* associated with that *waiting invocation*.

The following *waiting invocations* are built-in.

```
SEND(m,v)
RECEIVE(m,v)
DELAY(t)
DELAY_UNTIL(t)
DELAY_UNTIL(t,a)
DELAY_UNTIL_INACTIVE(a)
```

The first two are used to send and receive messages from some mailbox (see Section 10.3). The last four are used to cause delays (see Section 10.4). Additional *waiting invocations* can be defined using the low-level facilities discussed in Section 14.3. Note that arbitrary procedure and function invocations cannot be used as *waiting invocations*.

RULES

Elaboration of a wait statement consists of examining the waiting invocations following WHEN. If at least one can complete immediately, one of those that can complete is allowed to complete and the associated *body* is elaborated. If more than one could complete, only one will be allowed to complete; the one that actually completes is not defined. If no *waiting invocations* can complete immediately, the task activation which elaborated the *wait statement* waits until one can complete.

For SEND's and RECEIVE's used as *waiting invocations*, if none can complete immediately, the activation that elaborates the *wait statement* is placed on the FIFO waiting queue of each of the specified mailboxes. When one SEND or RECEIVE completes, the activation will be removed from the FIFO queues of the other mailboxes.

If a *waiting invocation* is SEND (m,v), the mailbox m must have a size which is greater than zero; otherwise, X_EMPTY_MAILBOX is raised.

NOTES

Low level details of the semantics of the *wait statement* are described in Chapter 14.

EXAMPLES

```
% wait on two mailboxes
TASK consume2 IMPORTS m, m1, outfile;
   VAR data : s;
   WHILE TRUE REPEAT
      WAIT
         WHEN RECEIVE(m,data),
              RECEIVE(m1,data) =>
                 WRITE(outfile, data);
      END WAIT;
   END REPEAT;
END TASK consume2;
```

10.6 SHARED VARIABLES

A *variable* is said to be shared if two or more activations can use it. Some cases of sharing are considered to be dangerous sharing. Dangerous sharing occurs if two activations simultaneously modify the same shared *variable* or if one activation modifies a shared *variable* while some other activation is accessing that shared *variable*. The translator will issue warning messages for those cases where dangerous sharing might occur.

When dangerous sharing of some shared *variable* is possible, the user must ensure that the activations that can use that shared *variable* elaborate these uses in an orderly way. If simultaneous use does occur, the effect (including the state of the *variable* and any value accessed) is not defined. For a shared *variable* where the user has ensured orderly access, there is a pragmat available to suppress the warning messages for dangerous sharing of that shared *variable* (see Appendix B).

NOTES

Let a1 and a2 be activations that share some variable v in a dangerous way. Orderly access can be ensured by either of the following means:

a) Using the *region statement* to surround any references to v in a1 and a2 so that only one of the references can happen at one time (see next section).

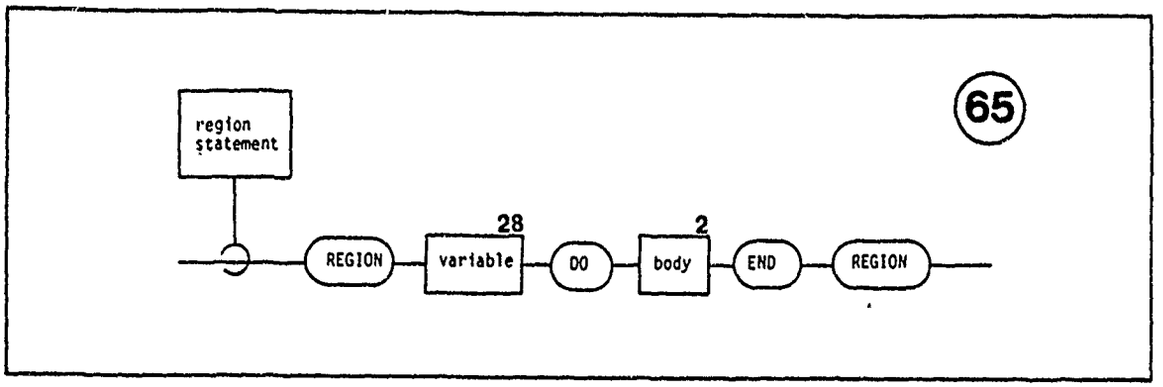
b) Synchronizing a1 and a2 by using messages so that references to v will not happen simultaneously (see example below).

EXAMPLES

1) Mutual exclusion with mailboxes.

```
VAR common : INT(0 .. 10);
VAR m : MAILBOX[ INT(0 .. 0) ](1);

TASK t1 IMPORTS m, common;
   VAR local : INT(0 .. 10);
   VAR right : INT(0..0);
   ...
   RECEIVE( m, right );
      common := local;
   SEND( m, right );
   ...
END TASK t1;

TASK t2 IMPORTS m, READONLY common;
   VAR local : INT(0 .. 10);
   VAR right : INT(0..0);
   ...
   RECEIVE( m, right );
      local := common;
   SEND( m, right );
   ...
END TASK t2;

VAR a1, a2 : ACT;
CREATE t1 NAMED a1;
CREATE t2 NAMED a2;
SEND( m, 0 );
```

10.7 REGION STATEMENT AND DATA LOCK VARIABLES



Use of the *region statement* is one way of ensuring the orderly access to a shared variable by several activations. The *region statement* is normally used in conjunction with a data lock variable. For example,

```
VAR d : DATA_LOCK;
```

A data lock variable has two possible states: locked and unlocked. Each data lock variable is automatically initialized to have the unlocked state.

Basically, the *region statement* is elaborated by elaborating its *body*. However, the *region* statement ensures that if several activations contain *region statements*, each specifying the same data lock variable, at most one of these activations will be elaborating the *body* of its *region statement*. If two or more activations attempt to elaborate *region statements* specifying the same data lock variable simultaneously, then all except one will wait (until that one completes elaboration of its *region* statement). If several activations are waiting for region access based on the same data lock variable, then access will be granted on a first-come first-served basis.

The *region statement* can also be defined to work for *variables* with *types* other than DATA_LOCK (see Section 14.4.1).

RULES

Elaboration of a *region statement* consists of the following steps:

a) lock the *variable*, wait if necessary until this can be done;

b) elaborate the *body*; and

c) unlock the *variable*.

Once the *variable* has been locked, it is guaranteed to be unlocked whenever the elaboration of the *body* completes. The unlocking will happen whether the *body* terminates normally, raises an exception, or does an exit, goto, or return.

NOTES

The DATA_LOCK type is described in Appendix C. Detailed semantics of the *region statement* and ways for using it with *variables* other than data lock variables are discussed in Chapter 14.

EXAMPLES

1) Simple mutual exclusion.

```
VAR common : INT(0 .. 10);
VAR d : DATA_LOCK;

TASK t1 IMPORTS d, common;
   VAR local : INT(0 .. 10);
   ...
   REGION d DO
      common := local;
   END REGION;
   ...
END TASK t1;

TASK t2 IMPORTS d, READONLY common;
   VAR local : INT(0 .. 10);
   ...
   REGION d DO
      local := common;
   END REGION;
   ...
END TASK t2;

VAR a1, a2 : ACT;

CREATE t1 NAMED a1;
CREATE t2 NAMED a2;
```

11. OVERLOADING AND GENERICS

Overloading is the association of a single name with multiple deferred units of the same kind. A name could be associated with several different procedures, for example, but not with a procedure and a function. All deferred units associated with a single overloaded name will normally perform logically related computations. For example, the overloaded name ABS is associated both with a built-in function for finding the absolute value of an integer and with another built-in function for finding the absolute value of a floating point number.

Interfaces are used to match each use of an overloaded name, during translation, to a particular deferred unit associated with that name.

In the simplest case, interfaces depend only upon a signature which includes the number, order, and types of a list of parameters. The use of an overloaded name in an invocation will be resolved to the deferred unit which has a matching signature; that is, the number, order, and types for the *actual* parameters are identical to the number, order and types for the *formal parameters*. For example,

```
BEGIN
   FUNC iszero (a : INT) => BOOL;      % iszero 1
      RETURN a=0;
   END FUNC iszero;

   FUNC iszero (a : FLOAT) => BOOL;    % iszero 2
      CONST delta := 1.0E-5;
      RETURN a < delta AND a > -delta;
   END FUNC iszero;

   VAR i : INT(-10..10);
   VAR f : FLOAT(10,-10.0 .. 10.0);

   ...iszero (i)...                     % invokes iszero 1
   ...
   ...iszero (f)...                     % invokes iszero 2
   ...
END;
```

In addition to the signature, interfaces can depend upon a translation time property list specified by a list enclosed in square brackets (i.e., [...]). The use of an overloaded name in an invocation will be resolved to a deferred unit which has a matching translation time property list. For example,

```
BEGIN
   FUNC zero [INT] => INT(0..0);                 % zero 1
      RETURN 0;
   END FUNC zero;

   FUNC zero [FLOAT] => FLOAT(10,0.0 .. 0.0);    % zero 2
      RETURN 0.0;
   END FUNC zero;

   ...zero [INT]...       % invokes zero 1
   ...
   ...zero [FLOAT]...     % invokes zero 2
   ...
END;
```

The information which is considered to be part of the interface depends upon the kind of deferred unit. For types, the translation time property list is the entire interface (signatures are not part of the interface for *types*). For other deferred units, the interface consists of both the translation time property list (if specified) and the signature.

There are two kinds of overloading: explicit overloading and generic overloading. Explicit overloading occurs when several distinct definitions of a name are written. Explicit overloading can be used for any kind of deferred unit except *types*. Generic overloading occurs when a single deferred declaration is replicated as a result of its appearance within a generic declaration. Generic overloading can be used for any kind of deferred unit.

NOTES

Names associated with variables, constants, exceptions, goto labels, or matching identifiers can not be overloaded.

11.1 INTERFACES

Interfaces are used to resolve each use of an overloaded name to one of the deferred units associated with that name. Interfaces include signatures (for all deferred units except *types*) and translation time property lists (if specified).

RULES

Each definition of a deferred unit has a formal interface. Each invocation of a deferred unit has an actual interface.

Each use of the (possibly overloaded) name of a deferred unit is resolved to the deferred unit associated with the name whose formal interface matches the actual interface of the use. If there is no such deferred unit, the use is in error.

A efinitions of a name conflict unless

a) They are both deferred units of the same kind, and
b) Their interfaces do not match.

The interface of a procedure, function, task, abbreviation, or capsule consists of a signature and, if specified, the translation time property list. The interface of a capsule or *type* consists of a translation time property list if any is specified.

Two interfaces match if:

a) Both have matching signatures or neither includes a signature, and
b) Both have matching translation time property lists or neither includes a translation time property list.

NOTES

Matching of interfaces is done during translation.

The scope rules (see Section 3.5) permit a local definition to override a conflicting definition in an enclosing scope; that is, one deferred declaration will override another deferred declaration in an enclosing scope if they have matching interfaces. A local declaration of one kind will override all declarations of another kind when all share the same name.

EXAMPLES

1) No signature and no translation time property list

```
PROC p; ... END PROC p;      % definition 1
...
p;                           % invocation 1
```

2) Signature and no translation time property list

```
PROC p (x : INT); ... END PROC p;    % definition 2
...
p (3);                               % invocation 2
```

3) Translation time property list but no signature

```
PROC p [INT]; ... END PROC p;        % definition 3
...
p [INT];                             % invocation 3
```

4) Both a signature and a translation time property list

```
PROC p [INT] (x : INT); ... END PROC p;   % definition 4
...
p [INT] (3);                              % invocation 4
```

11.1.1 SIGNATURES

A signature is part of the interface between any deferred unit except a *type* and its invocations. Signatures are derived from the formal and actual parameter lists and are not explicitly specified.

RULES

The formal signature of a procedure, function, task, abbreviation, or capsule is an ordered list of the *types* or *subtypes* specified for its *formal parameters*. The list is empty if it has no *formal* parameters.

The actual signature of a procedure, function, task, abbreviation, or capsule invocation is an ordered list of the *subtypes* of its *actual parameters*. If there are no *actual parameters*, the list is empty.

Two signatures match if their lists are the same length and each of their elements match. Two *types* match if they are equal (see Section 4.1.5). Two *subtypes* match if they belong to the same *type*. A *type* and a *subtype* match if the *subtype* belongs to the *type*.

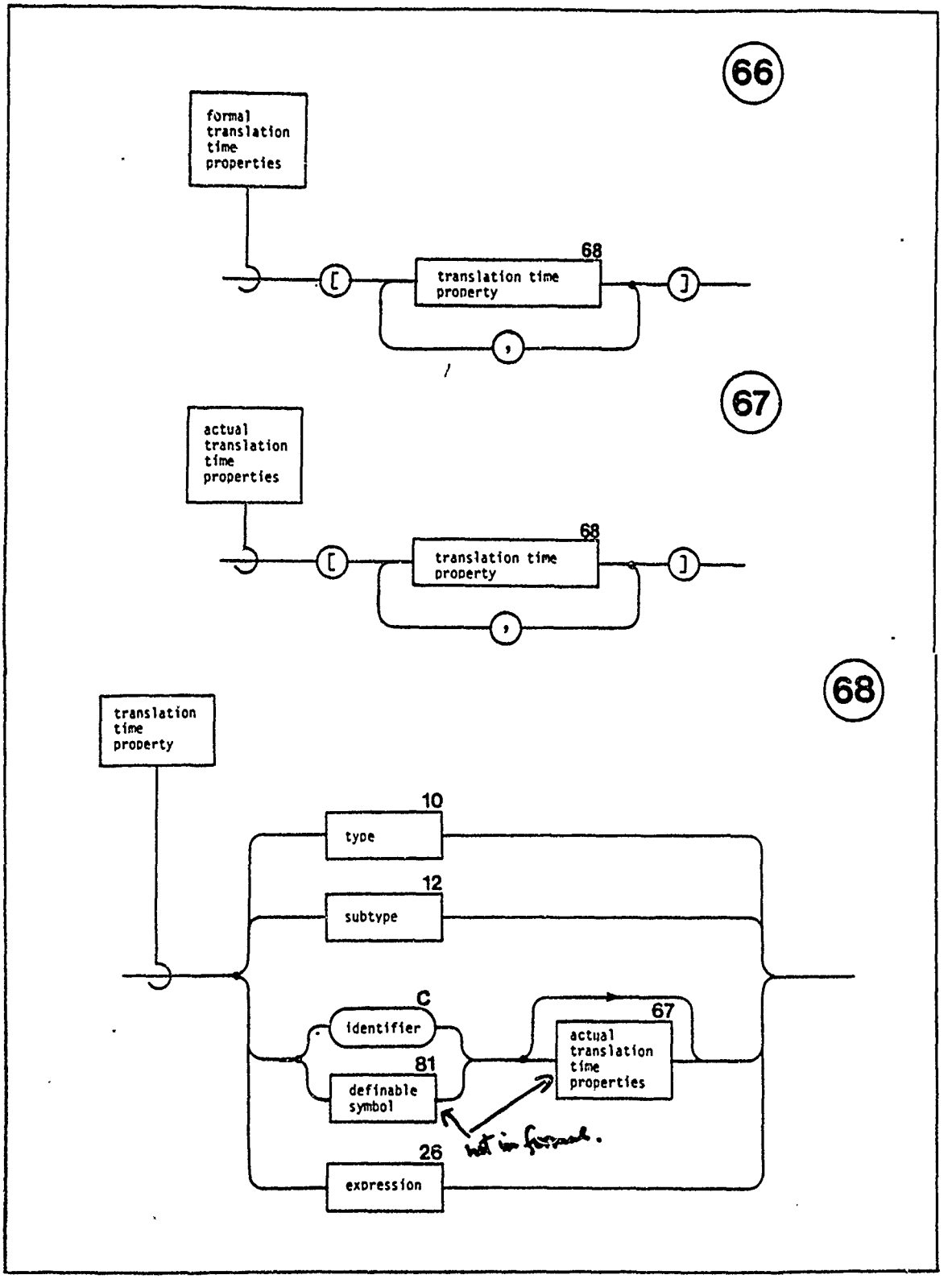
NOTES

Function result types or subtypes are not considered to be part of a signature.

EXAMPLES

```
FUNC sign => BOOL;                    % sign 1
   RETURN FALSE;
END FUNC sign;

FUNC sign (i : INT) => BOOL;          % sign 2
   RETURN i>=0;
END FUNC sign;

FUNC sign (i,j : INT) => BOOL;        % sign 3
   RETURN i+j >= 0;
END FUNC sign;

FUNC sign (x : FLOAT) => BOOL;        % sign 4
   RETURN x>=0.0;
END FUNC sign;

FUNC sign (x,y : FLOAT) => BOOL;      % sign 5
   RETURN x+y >= 0.0;
END FUNC sign;

VAR c1, c2, c3, c4, c5 : BOOL;
VAR k,l : INT(-50..50);
VAR q,r : FLOAT(10, -50.0 .. 50.0);

c1 := sign;          % invokes sign 1
c2 := sign (k);      % invokes sign 2
c3 := sign (k,l);    % invokes sign 3
c4 := sign (q);      % invokes sign 4
c5 := sign (q,r);    % invokes sign 5
```

11.1.2 TRANSLATION TIME PROPERTY LISTS



A translation time property list, if present, is part of the interface between a deferred unit and its invocations. If a formal translation time property list is included in a deferred unit, each of the

invocations must specify a matching actual translation time property list. Translation time property lists are always explicitly specified.

RULES

Each *expression* must be manifest and must be of a *type* for which equality (=) is defined. Each *identifier* must be the name of a function, procedure, or task. The *identifier* may be followed by an actual translation time property list only when it appears as a property in an actual generic property list. *Definable symbols* may only appear in an actual generic property list.

Two translation time property lists match if they have the same number of properties and if their corresponding properties match. Two *expressions* match if their values are equal. Two *types* match if they are equal (see Section 4.1.5). Two *subtypes* match if they belong to the same *type*. A *type* and a *subtype* match if the *subtype* belongs to the *type*. An *identifier* matches an actual property if they both refer to the same procedure, function, or task.

EXAMPLES

```
BEGIN
   FUNC zero [INT] => INT(0..0);                    % zero 1
      RETURN 0;
   END FUNC zero;

   FUNC zero [FLOAT, 5] => FLOAT(5,0.0 .. 0.0);    % zero 2
      RETURN 0.0;
   END FUNC zero;

   FUNC zero [FLOAT, 10] => FLOAT(10,0.0 .. 0.0); % zero 3
      RETURN 0.0;
   END FUNC zero;

   ...zero [INT]...          %invokes zero 1
   ...
   ...zero [FLOAT, 5]...     %invokes zero 2
   ...
   ...zero [FLOAT, 10]...    %invokes zero 3
   ...
END;
```

11.2 EXPLICIT OVERLOADING

Explicit overloading occurs when there are several distinct definitions of deferred units of the same kind, each of which has the same name, but a different interface.

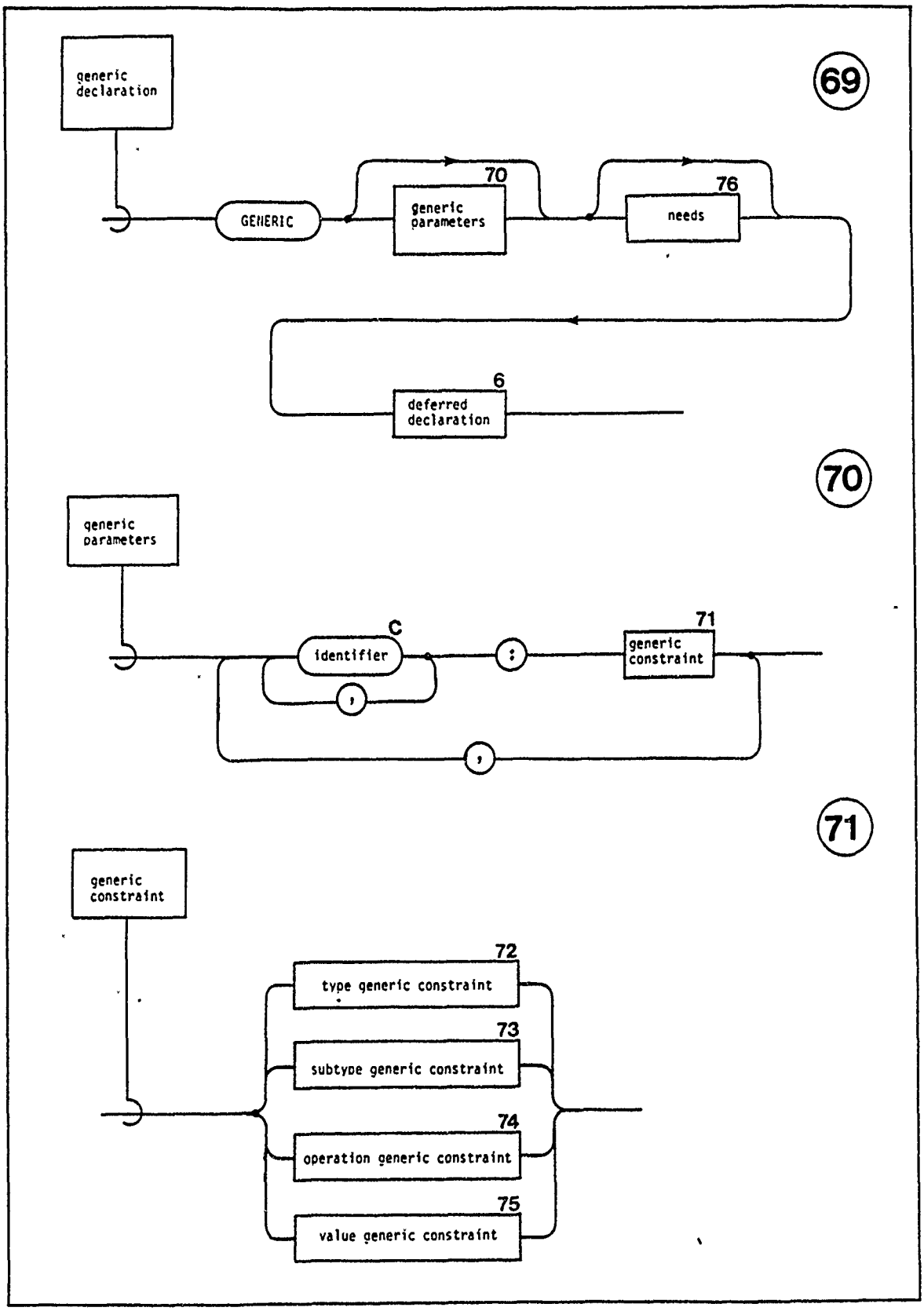
RULES

Explicit overloading can be used for any kind of deferred unit except *types*.

EXAMPLES

```
BEGIN
   CAPSULE stackcap EXPORTS stack, insert, remove;
      TYPE stack : ... ;

      PROC insert (VAR s : stack, i : INT);          % insert 1
         ...
      END PROC insert;

      PROC remove (VAR s : stack, OUT r : INT);     % remove 1
         ...
      END PROC remove;
   END CAPSULE stackcap;

   CAPSULE queuecap EXPORTS queue, insert, remove;
      TYPE queue : ... ;

      PROC insert (VAR s : queue, i : INT);          % insert 2
         ...
      END PROC insert;

      PROC remove (VAR s : queue, OUT r : INT);     % remove 2
         ...
      END PROC remove;
   END CAPSULE queuecap;

   EXPOSE stackcap;
   EXPOSE queuecap;

   VAR s : stack;
   VAR q : queue;
   VAR j : INT(0..10);

   insert (s, 3);      % invokes insert 1
   ...
   insert (q, 4);      % invokes insert 2
   ...
   remove (s, j);      % invokes remove 1
   ...
   remove (q, j);      % invokes remove 2
   ...
END;
```

11.3 GENERIC DECLARATION



A *deferred declaration* can be "generalized" by placing it in a *generic declaration* and by replacing specific *types*, *subtypes*, or procedure or function names by references to *generic parameters*. For example, a sort procedure which sorts arrays with integer components can be easily generalized to a generic sort procedure which will sort arrays with any component type.

RULES

The deferred declaration is called the pattern declaration and defines the overloaded name.

The interface of the pattern declaration must contain a use of each *generic parameter*. The use can be

a) within the formal translation time property list, or

b) within in the formal signature, or

c) in the *generic constraint* of some other *generic parameter* that has a use in the interface. The use cannot appear as a result type or subtype of a FUNC operation constraint.

If a *generic parameter* appears more than once in the formal interface, each of the corresponding places in an actual interface must resolve the *generic parameter* to the same replacement element.

The *generic declaration* is replaced during translation by a set of *deferred declarations* called the generated set.

Each *deferred declaration* in the generated set is obtained first by copying the pattern declaration and then substituting a specific replacement element for each *generic parameter* and for each needs list definition.

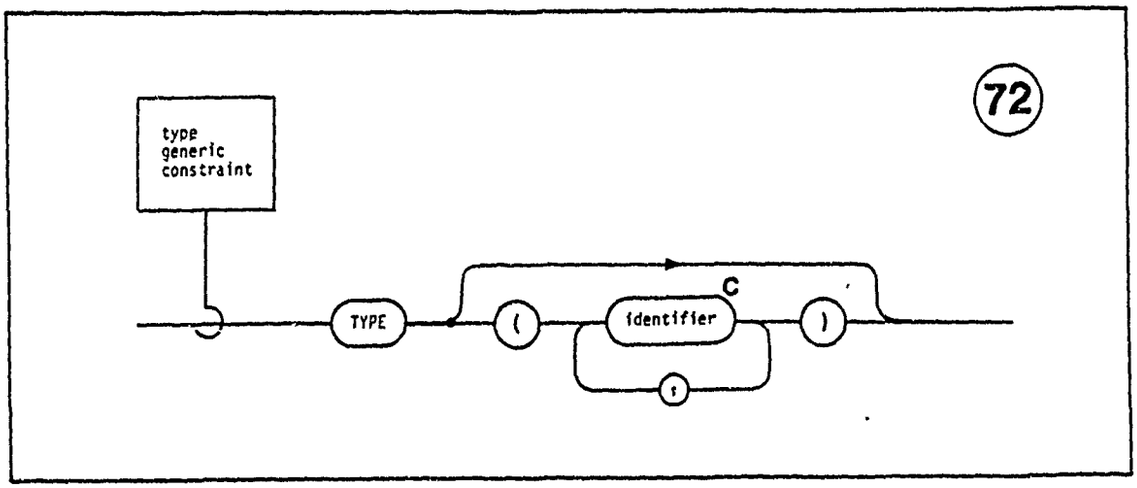
If the overloaded name is never invoked, then the generated set is empty. Otherwise, each invocation of the overloaded name is examined. Each invocation of the overloaded name will have an actual interface which will be used to set a replacement element for each generic parameter. Needed definitions are set to corresponding definitions known in the scope of invocation (see Section 11.4). A new copy of the pattern *declaration*, with replacement elements for its *generic parameters* and for each needs list definition, is added to the generated set if no equivalent copy has been added previously, as a result of examining some other invocation.

If a *generic declaration* is capable of generating some *deferred declaration* that conflicts with a particular definition, then the *generic declaration* itself is considered to conflict with that definition. The conflict exists even though the *deferred declaration* is not actually generated.

NOTES

A *generic declaration* is an open scope; the *generic parameters* and the needs list items are defined in this scope.

11.3.1 TYPE GENERIC CONSTRAINTS



A *generic parameter* with a *type generic constraint* has replacement elements which are *types*.

RULES

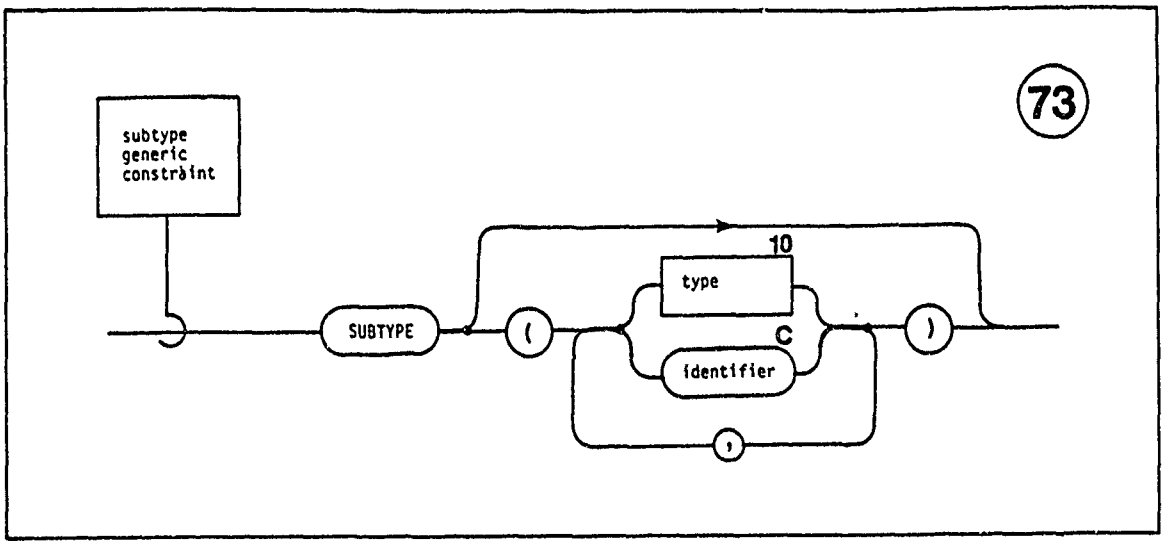
Any type (including those not known in the scope of the generic declaration) may be a replacement element for a generic parameter whose generic constraint is TYPE.

Any type whose name is id and is known in the scope of the generic declaration can be a replacement element for a generic parameter whose generic constraint is TYPE (..., id, ...).

EXAMPLES

```
GENERIC t : TYPE
    PROC p (VAR x : t) ; ... END PROC p;

GENERIC t : TYPE
    CAPSULE c[t]; ... END CAPSULE c;

GENERIC i : TYPE (INT, ENUM), t : TYPE(INT,FLOAT)
    PROC q (x : ARRAY i OF t); ... END PROC q;
```

11.3.2 SUBTYPE GENERIC CONSTRAINTS



A *generic parameter* with a *subtype generic constraint* has replacement elements which are *subtypes.*

RULES

Any subtype (including those whose types are not known in the scope of a generic declaration can be used as replacement elements for a generic parameter whose generic constraint is SUBTYPE.

Any subtype of type t may be used as a replacement element for a generic parameter whose generic constraint is SUBTYPE (...,t,...).

Any subtype whose type has the name id and is known in the scope of the generic declaration can be used as a replacement element of a generic parameter whose generic constraint is SUBTYPE(...,id,...).

If a subtype is used in several places within the formal interface, then the actual interface must specify the same subtype in each of the corresponding places. Otherwise, the X_SUBTYPE exception is raised when the invocation is elaborated.

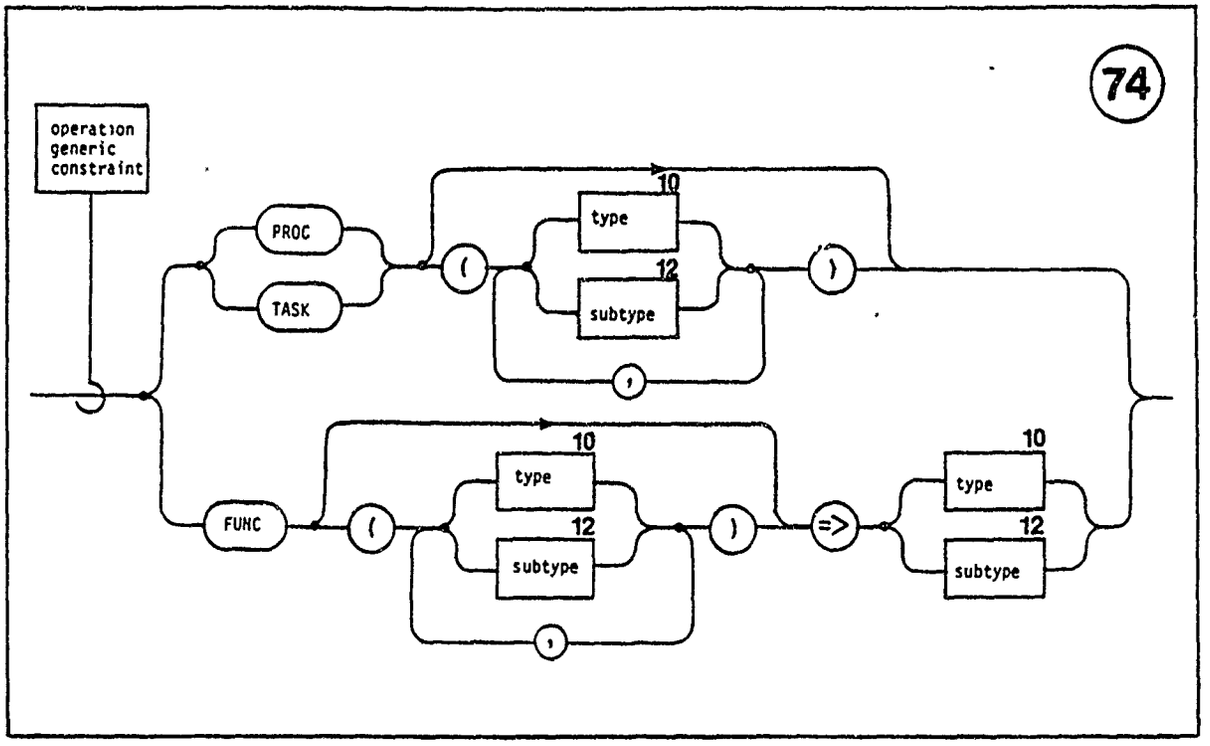
NOTES

For interface matching, only the *type* of *subtypes* is used. Subtype information is used to set *generic parameters* with a *subtype generic constraint*, as well as to check for consistency of *subtypes* (X_SUBTYPE checking).

EXAMPLES

```
% x and y have the same type, but not necessarily the same subtype
GENERIC t : TYPE
    PROC p (VAR x : t, VAR y : t); ... END PROC p;

% x and y have the same subtype, which can be referred to as s
%    within the body
GENERIC s : SUBTYPE
    PROC p (VAR x : s, VAR y : s); ... END PROC p;

% x and y have the same type but not necessarily the same subtype,
%    the subtype of x can be referenced as s1 while the subtype
%    of y can be referenced as s2
GENERIC t : TYPE, s1 : SUBTYPE(t), s2 : SUBTYPE(t)
   PROC p (VAR x : s1, VAR y : s2); ... END PROC p;

GENERIC s1 : SUBTYPE (ENUM), s2 : SUBTYPE (ENUM)
   FUNC f (a : ARRAY s1 OF s2, READONLY b : s2) => s1; ... END FUNC f;

GENERIC  s : SUBTYPE (ENUM)
   TYPE vstring [s] : PTR (n :INT) STRING [s] (n);
```

11.3.3 OPERATION GENERIC CONSTRAINTS



An *operation generic constraint* has replacement elements which are either procedures, functions, or tasks. In each case, the *types* or *subtypes* of the *formal parameters* are specified and, for functions, the result type or subtype is also specified.

RULES

The replacement elements for the PROC generic constraint are all procedures having the specified number of parameters of the specified types.

The replacement elements for the FUNC generic constraint are all functions having the specified number of parameters of the specified types and the specified result type.

The replacement elements for the TASK generic constraint are all tasks having the specified number of parameters of the specified types.

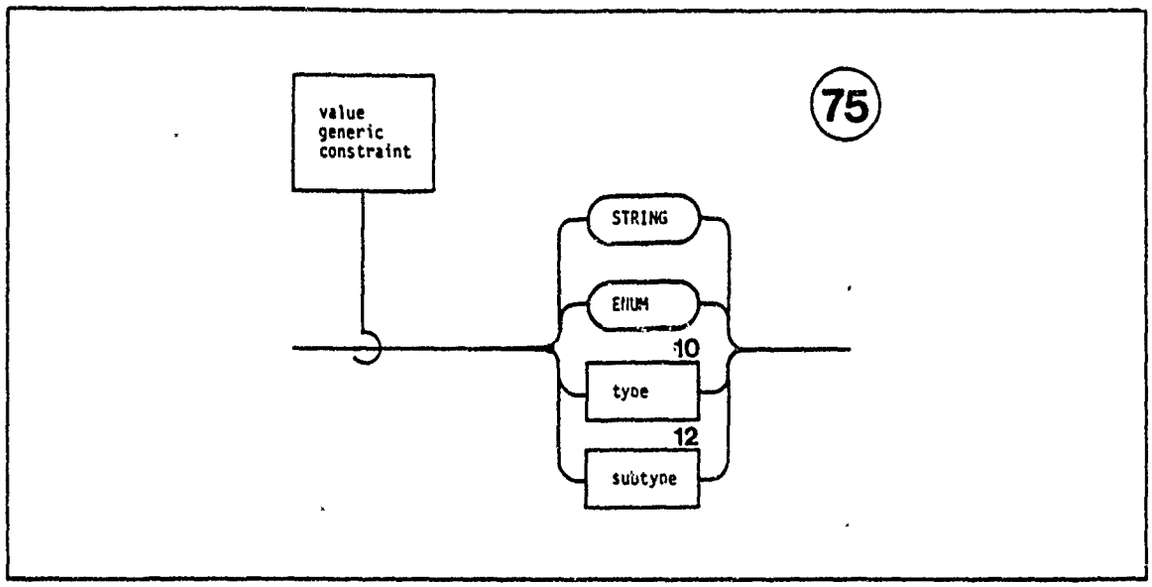
If a parameter subtype is specified, matching depends only upon the type to which it belongs. If the subtype parameter or result specified in the generic constraint does not match the actual subtype of the replacement element, the X_SUBTYPE exception is raised.

If the replacement element is itself overloaded or generic, then it is resolved in the calling scope and its *needed items* are bound there.

EXAMPLES

```
GENERIC f : FUNC(FLOAT) => FLOAT
   FUNC integrate [f] (low,high : FLOAT)
                        => FLOAT(20, -10E10 ..1.0E10);
      CONST delta := 1.0E-5;
      VAR r : FLOAT(20, -10E10 .. 1.0E10) := 0.0;
      VAR index : FLOAT(10, -1.0E10 .. 1.0E10) := low;
      WHILE index < high REPEAT
         r := r + delta * f(index);
         index := index + delta;
      END REPEAT;
      RETURN r;
END FUNC integrate;


GENERIC t : TYPE, less_than : FUNC(t,t)=>BOOL
   NEEDS :=(t,t);
     PROC sort [less_than] (VAR a : ARRAY INT OF t);
        CONST min := INDEXOF(a).MIN;
        CONST max := INDEXOF(a).MAX;
        FOR i : INT(1..max-min) REPEAT
           FOR j : REVERSE INT(max-i..max-1) REPEAT
              IF less_than(a(j), a(j+1)) THEN
                 CONST temp := a(i);
                 a(i) := a(i+1);
                 a(i+1) := temp;
              END IF;
           END REPEAT;
        END REPEAT;
     END PROC sort;


ABBREV r : RECORD[p,q,s : INT(0..100)];
VAR a : ARRAY INT(1..100) OF INT(0..100);
VAR b : ARRAY INT(0..500) OF r;

SORT[<] (a);      % sort in ascending order
SORT[>] (a);      % sort in descending order
...
SORT[p] (b);      % sort ascending based on field p
SORT[pq] (b);     % sort ascending on primary key p
                  %  and descending on secondary key q

FUNC p (x,y : r) => BOOL;
   RETURN x.p < y.p;
END FUNC p;

FUNC pq (x,y : r) => BOOL;
   CASE TRUE
      WHEN x.p < y.p => RETURN TRUE;
      WHEN x.p = y.p => RETURN x.q > y.q;
      WHEN x.p > y.p => RETURN FALSE;
   END CASE;
END FUNC pq;
```

11.3.4 VALUE GENERIC CONSTRAINTS



A *generic parameter* with a *value generic constraint* has replacement elements which are manifest values.

RULES

A *generic parameter* with a *value generic constraint* must appear as an item in a formal translation time property list within the interface.

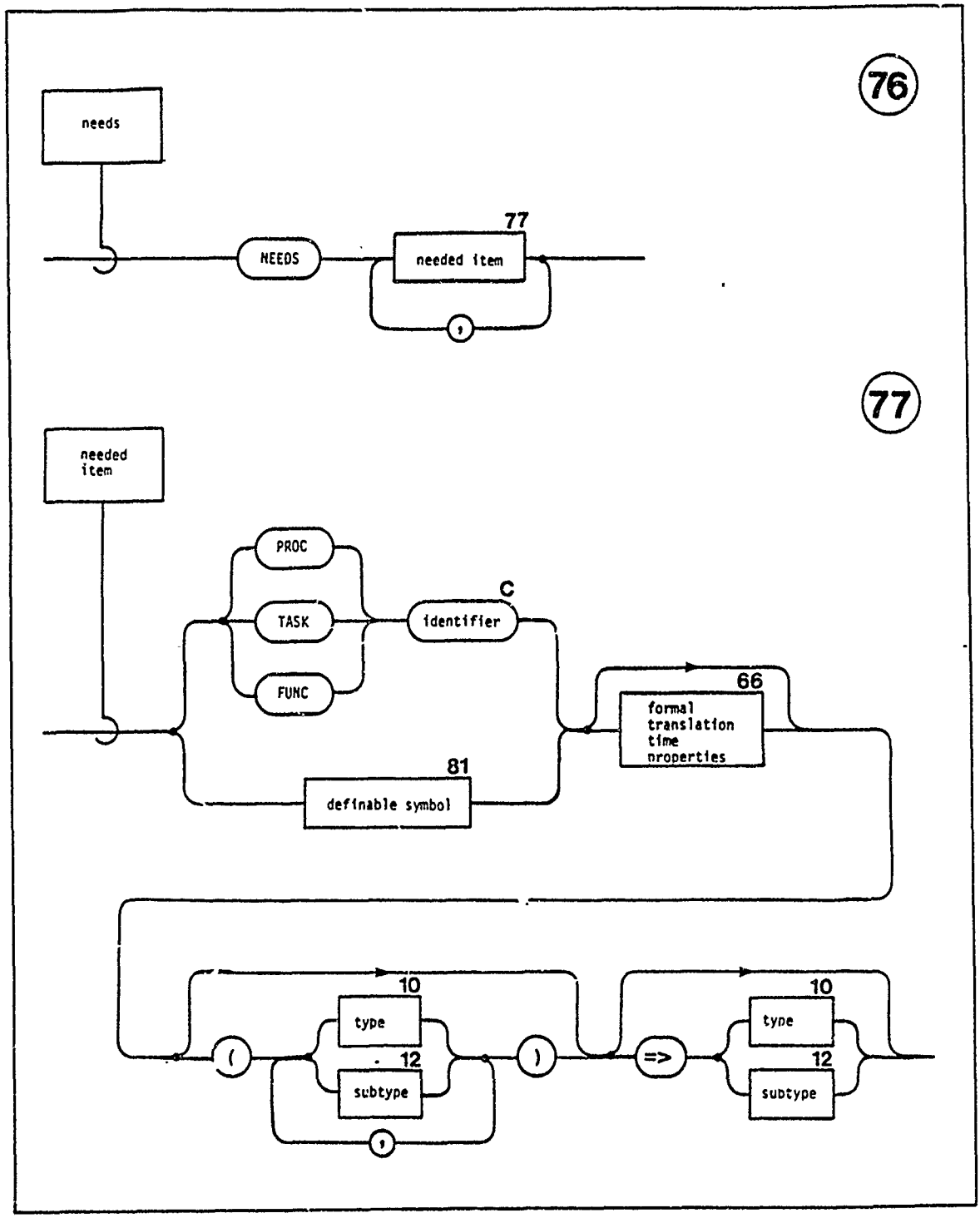
The only *types* and *subtypes* permitted are BOOL, INT, FLOAT, ENUM, and STRING types and subtypes.

For the STRING and ENUM constraints, uses of the formal generic parameter with the pattern declaration are type unresolved, if the replacement element is type unresolved. For the INT, FLOAT, ENUM[...], and STRING[t] constraints, uses of the formal generic parameter within the pattern declaration are subtype unresolved, if the replacement element is subtype unresolved.

EXAMPLES

```
CAPSULE fcap EXPORTS float, :=, +, ...;

   GENERIC i : INT(1..30)
      TYPE float [i] (a,b : FLOAT) : FLOAT (i, a..b);

   GENERIC i : INT(1..30)
      FUNC + (x,y : float [i]) float [i];
         CONST r := x.ALL + y.ALL;
         VAR z := float[i] (r..r);
         z.ALL := r;
      RETURN z;
      END FUNC +;
      ...
END CAPSULE fcap;
```

11.4 NEEDS LIST



Whenever a *generic parameter* is a *type*, it is possible that the pattern declaration inside the *generic declaration* requires some procedures, functions or tasks that operate on *variables* or *constants* of that *type*. Such procedures, functions and tasks must be obtained from the scope in which the overloaded name is invoked, since nothing is known about the *type* in the scope where the generic declaration appears. Such procedures, functions and tasks could be obtained by having

additional *generic parameters*, but this would lead to a proliferation of such parameters. Instead, the needs list provides this mechanism. The definitions that appear in the needs list contain sufficient information to identify the needed procedure, function, or task uniquely, as well as completely specify its interface.

RULES

When a generically overloaded name is invoked, and the *generic declaration* includes a needs list, each definition in the needs list is resolved to the definition of the same name and matching interface known in the scope where the invocation appears.

If the needed definition is, itself, generically overloaded, then it is resolved and its needs list is also resolved in that scope.

The needs list may not appear in any *generic declaration* whose pattern declaration is a *type*.

EXAMPLES

```
BEGIN
   GENERIC t : TYPE NEEDS :=(t,t)
      PROC swap (VAR a,b : t);
         CONST c := a;
         a := b;
         b := c;
      END PROC swap;

   VAR i,j : INT(1..10);
   VAR f,g : FLOAT(10, 1.0 .. 10.0);
   ...
   swap (i,j);
   ...
   swap (f,g);
   ...
END;
```

12. MACHINE DEPENDENT FACILITIES

This chapter discusses the facilities of the language for specifying machine dependent representations for *types*, specific locations for *variables*, techniques for using code written in other languages (including assembly language), and configuration capsules which encapsulate machine-dependent information. Other machine-dependent facilities include interrupts (see Section 14.1.4) and low-level I/O (see Section 14.6).

12.1 CONFIGURATION CAPSULES

A capsule with the name "configuration" is called a configuration capsule. A configuration capsule contains information that is dependent upon the particular target system on which a program is to be run. Some of the kinds of data that can be specified in a configuration capsule include:

a) An identification of the target computer and operating system if any. For example,

```
CONST machine := "IBM370";
CONST op_sys  := "VS1";
```

b) Information about memory. For example,

```
CONST mem_size := 2**18*bytes;    % 256K bytes
CONST bytes := 8;                 % 8 bits per byte
CONST words := 4*bytes;           % 4 bytes per word
```

c) Information about timing (see Section 10.4). For example,

```
CONST milliseconds := 2;      % 2 ticks per millisecond
CONST seconds := 1000*milliseconds;
CONST minutes := 60*seconds;
CONST hours := 60*minutes;
```

d) Information about I/O devices. For example,

```
CONST console := 15;      % device 15 is the console
CONST tapes := 4;         % there are 4 tape drives
```

e) Other machine-dependent data of interest.

RULES

When a translation unit contains a capsule invocation declaration which specifies the capsule name `configuration`, the translator may check the values of various definitions exported (even if they are not made visible) by that capsule, to determine the characteristics of the target system. For example, a compiler that is targeted for PDP-11 may check that the *constant* machine is defined to have the value "PDP11". The information may also be used, when several *translation units* are linked together to form an executable program, to verify that all *translation units* have specified consistent configurations.

EXAMPLES

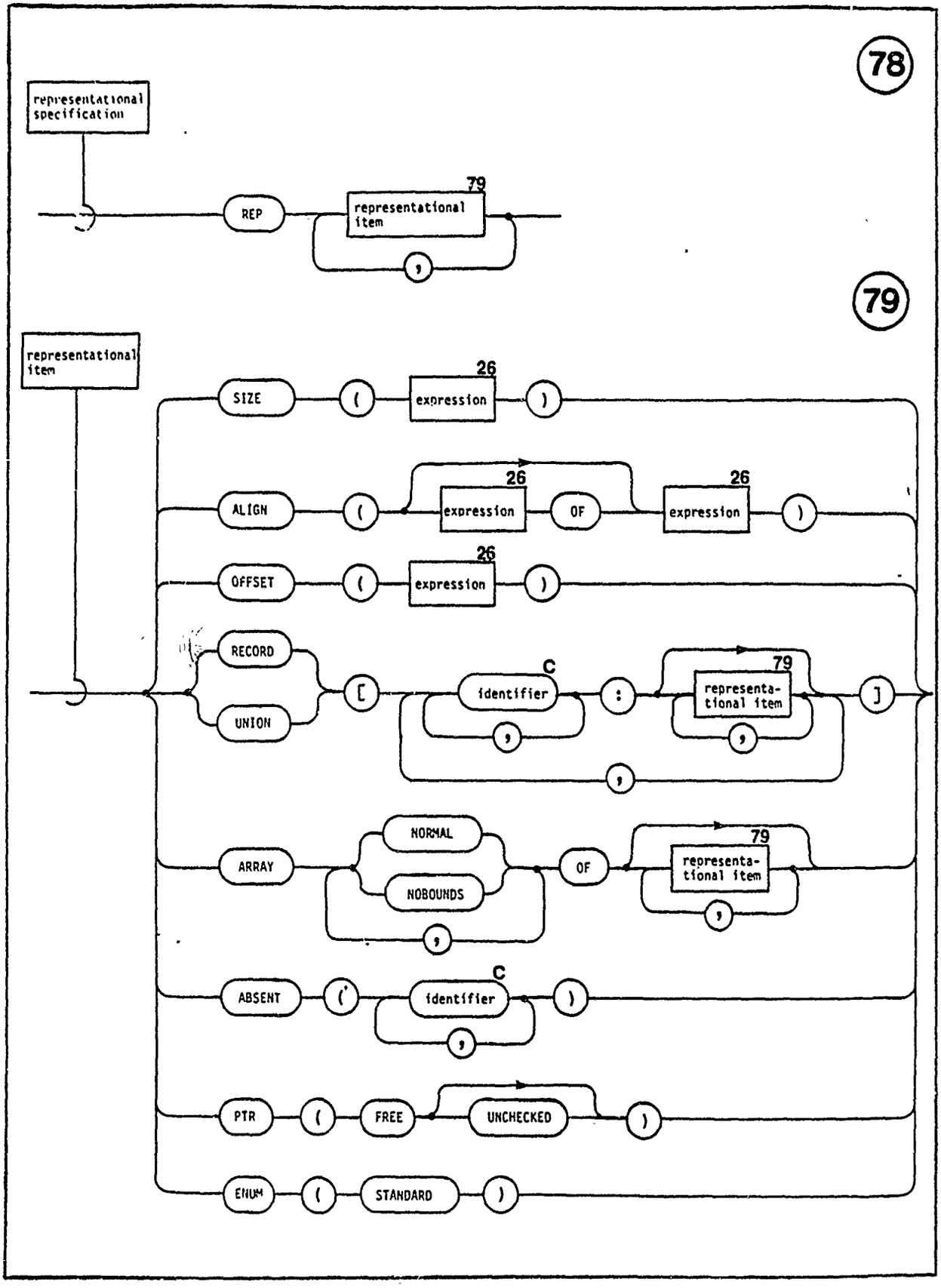
1) Definition of configuration capsules;

```
CAPSULE configuration ["IBM360"] EXPORTS ALL;
   CONST machine := "IBM360";
   CONST words := 32;   % 32 bits per word
END CAPSULE configuration;

CAPSULE configuration ["PDP11"] EXPORTS ALL;
   CONST machine := "PDP11";
   CONST words := 16;   % 16 bits per word
         ...
END CAPSULE configuration;
```

2) Use of configuration capsules:

```
CAPSULE trans_unit1 EXPORTS ... ;
   EXPOSE NONE FROM EXTERNAL configuration ["PDP11"];
   % this expose is for checking purposes only,
   %  no machine-dependent data is visible
         ...
END CAPSULE trans_unit1;

CAPSULE trans_unit2 EXPORTS ... ;
   EXPOSE words FROM EXTERNAL configuration ["PDP11"];
   % this translation unit uses the machine-dependent
   %  constant, words
         ...
END CAPSULE trans_unit2;

CAPSULE main_trans_unit EXPORTS NONE;
   EXPOSE NONE FROM EXTERNAL configuration ["PDP11"];
   EXPOSE ALL FROM EXTERNAL trans_unit1;
   EXPOSE ALL FROM EXTERNAL trans_unit2;
         ...
END CAPSULE main_trans_unit;
```

12.2 SPECIFICATION OF REPRESENTATIONS



A representation can be specified when a new type is declared (via the *type declaration*). A representation specifies the physical layout to be used for data items having *subtypes* of the new type. The layout is specified in terms of the memory of the target computer system.

RULES

When a representation is specified, the underlying type must consist only of built-in types. Each of the *representational items* are described below. Note that several different *representational items* may be specified for a given underlying subtype or one of its components.

Bit Numbering

The memory of the target computer is considered to be a sequence of bits. Suppose there are N bits per word. The high order bit of the first word is bit Ø. The low order bit of the first word is bit N-1. The high order bit of the m'th word is bit m*N. The low order bit of the m'th word is bit m*N+N-1.

SIZE

SIZE can be specified for any underlying subtype or component. The expression must have type INT and be greater than or equal to zero and specifies the maximum number of bits to be used to represent the underlying subtype or its components. If no size is specified, a default implementation-dependent size is used.

ALIGN

ALIGN can only be specified for the entire underlying subtype (i.e., it can not be specified for components). The form

```
ALIGN  (exp)
```

specifies that the representation for the underlying subtype is to start at a bit that is at any position p such that p = 0 MOD exp. The form

```
ALIGN(exp1 OF exp2)
```

specifies that the representation for the underlying subtype is to start at a bit that is at any position p such that p = exp1 MOD exp2. The *expressions* must all have type INT. The value of *expression* exp2 must be greater than zero. The value of exp1 must be in the range Ø .. exp2-1.

OFFSET

OFFSET can only be specified for components of records and unions. The value of the expression is the number of bits from the start of the record or union that the component representation is to start. The expression must have type INT and be greater than or equal to zero. If an offset is not specified for a record component, then that component starts immediately after the previous component. If an offset is not specified for a union component, then that component starts immediately after the tag. If no offset is specified for the first component of a record, that component starts at the start of the record. If no offset is specified for a union tag, the tag starts at the start of the union.

RECORD

This representational item can only be used for underlying subtypes or for components which have RECORD types. A representation can be specified for each component of the record.

UNION

This representational item can only be used for underlying subtypes or for components which have UNION types. A representation can be specified for each component of the union and a representation can also be specified for the special component named TAG.

ARRAY

This representational item can only be used for underlying subtypes or for components which have ARRAY types. For each index of the array, either NORMAL or NOBOUNDS must be specified. NOBOUNDS indicates that information used for array subscript bounds checking should not be stored.

ENUM

This item specifies that enumeration values are to be represented as the integers 0, 1, . . ., max. Since this representation is contiguous, checking for illegal values (e.g., on input) is greatly simplified.

ABSENT

The *identifiers* must each be names of attributes (i.e., formal parameters) of the type being defined. Absent attributes are not represented. Inquiry is not permitted for absent attributes.

PTR

This item specifies how the storage for dynamic variables is to be recovered. This representational item can only be specified for indirect types. FREE indicates that storage will be recovered only when the FREE procedure is explicitly invoked (i.e., there will not be any garbage collection). If UNCHECKED is specified, no representation will be provided for information used to check for the X_FREE exception when FREE is invoked for these dynamic variables.

Implementation-Dependent Representation

The *representational items* described here are only a basic set which will be supported by all implementations. Each implementation may provide additional implementation-dependent *representational items.*

Restrictions on Representation

A data item is said to have an explicit representation if

a) it is a component (including the .ALL component) of a data item of a user-defined type that has a representational specification, or

b) it is a component of a data item that has an explicit representation.

A data item that has an explicit representation may not be bound to a formal parameter that has a VAR or READONLY binding class. However, such data items can be used in expressions, in CONST or

OUT parameter positions, and as targets of the built-in assignment operators.

NOTES

When a representation is needed for a union where the tag is not to appear explicitly, this can be achieved by using a record with overlapping fields.

EXAMPLES

1) Record layout

```
TYPE packet : RECORD[a : INT(0..15),
                     b : BOOL,
                     c : ENUM['a, 'b, 'c],
                     d : ASCII,
                     e : FLOAT(5, -100.0 .. 100.0)]

     REP SIZE(2*WORDS), ALIGN(WORDS),
         RECORD[a : SIZE(4),
                b : SIZE(1),
                c : SIZE(2),
                d : SIZE(8), OFFSET(8),
                e : SIZE(1*WORDS), OFFSET(1*WORDS)];
```

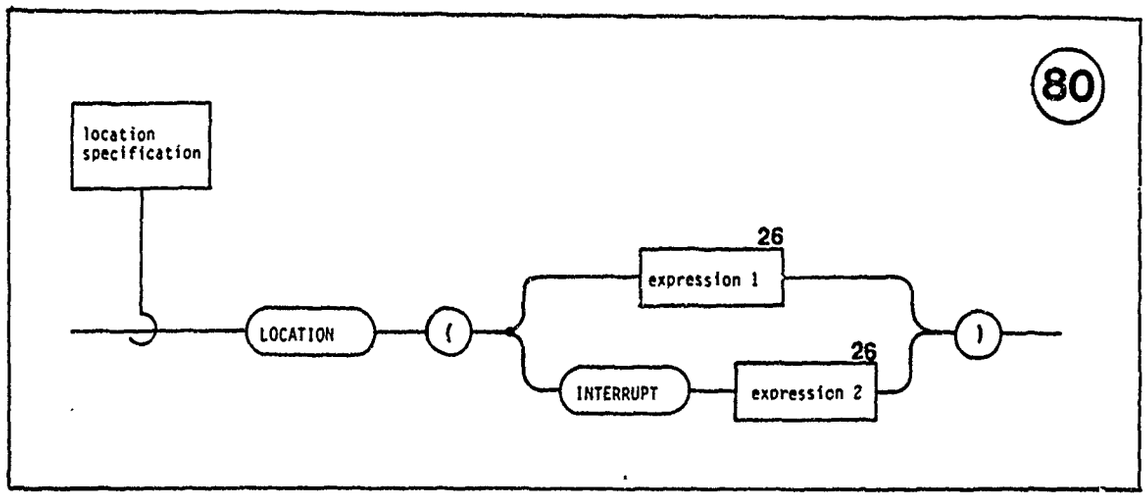
2) Pointers and array layout

```
TYPE myarray : PTR(n : INT(0..7))
       RECORD[i : INT(0..7),
              a : ARRAY(1..n) OF BOOL
   REP PTR (FREE UNCHECKED), SIZE(2*n+4),
       RECORD[i : SIZE(3),
              a : SIZE(2*n+1),
                  ARRAY UNCHECKED OF SIZE(1)];

PROC init_myarray (VAR m: myarray, j: INT(0..7));
     % m can be pass VAR, but m.i cannot
     ALLOC m PTR(j);
     m.i := j;
END PROC init_myarray;
```

Record with allocation time determined size and with overlaid fields

```
% If d is true, fields a and f are present
% If d is false, fields b and c are present
% Fields d and e are always present

TYPE myunion(n: INT):
          RECORD [a : ASCII,
                  b : BOOL,
                  c : INT(0..7),
                  d : BOOL,
                  e : FLOAT(5, -100.0 .. 100.0),
                  f : STRING[ASCII](6)]

   REP ABSENT(n), SIZE(n*BYTES),
          RECORD [a : SIZE(7), OFFSET(0),
                  b : SIZE(1), OFFSET(0),
                  c : SIZE(3), OFFSET(1),
                  d : SIZE(1), OFFSET(7),
                  e : SIZE(4*BYTES), OFFSET(1*BYTES),
                  f : SIZE(6*BYTES), OFFSET(5*BYTES)];

VAR x : myunion(5);
VAR y : myunion(11);

x.b := FALSE;
x.c := 3;
x.d := FALSE;   % b,c present; a,f absent
x.e := 0.0;

y.a := 'A;
y.d := TRUE;    % b,c absent; a,f present
y.e := 0.0;
y.f := "ABCDEF";
```

12.3 LOCATIONS



RULES

A *location specification* can appear on any *variable declaration*. The specification

```
LOCATION (e1)
```

will cause the variable to be stored at the memory location which is the value of e1. The expression e1 must have type INT and a value greater than or equal to zero. The specification

```
LOCATION (INTERRUPT e2)
```

can only appear for variables which have a MAILBOX type and causes that variable to be associated with hardware interrupt e2 (see Section 14.1.4).

EXAMPLE

```
% 360 program status words

TYPE psw: RECORD[sysmask: ARRAY INT(0..7) OF BOOL,
                     key : INT(0..15),
                 a,m,w,p : BOOL,
                   icode : INT(0 .. 2**18-1),
                 ilc, cc : INT(0..3),
                progmask : ARRAY INT(0..3) OF BOOL,
                    addr : INT(0 .. 2**24-1)]

REP SIZE(1*WORDS), ALIGN(2*WORDS),
    RECORD[sysmask : SIZE(8), ARRAY NOBOUNDS OF SIZE(1),
               key : SIZE(4),
           a,m,w,p : SIZE(1),
             icode : SIZE(2*BYTES),
            ilc,cc : SIZE(2),
          progmask : SIZE(4), ARRAY NOBOUNDS OF SIZE(1),
              addr : SIZE(3*BYTES)];

VAR ext_old_psw  : psw LOCATION (24*words);
VAR svc_old_psw  : psw LOCATION (32*words);
VAR prog_old_psw : psw LOCATION (40*words);
    ...
```

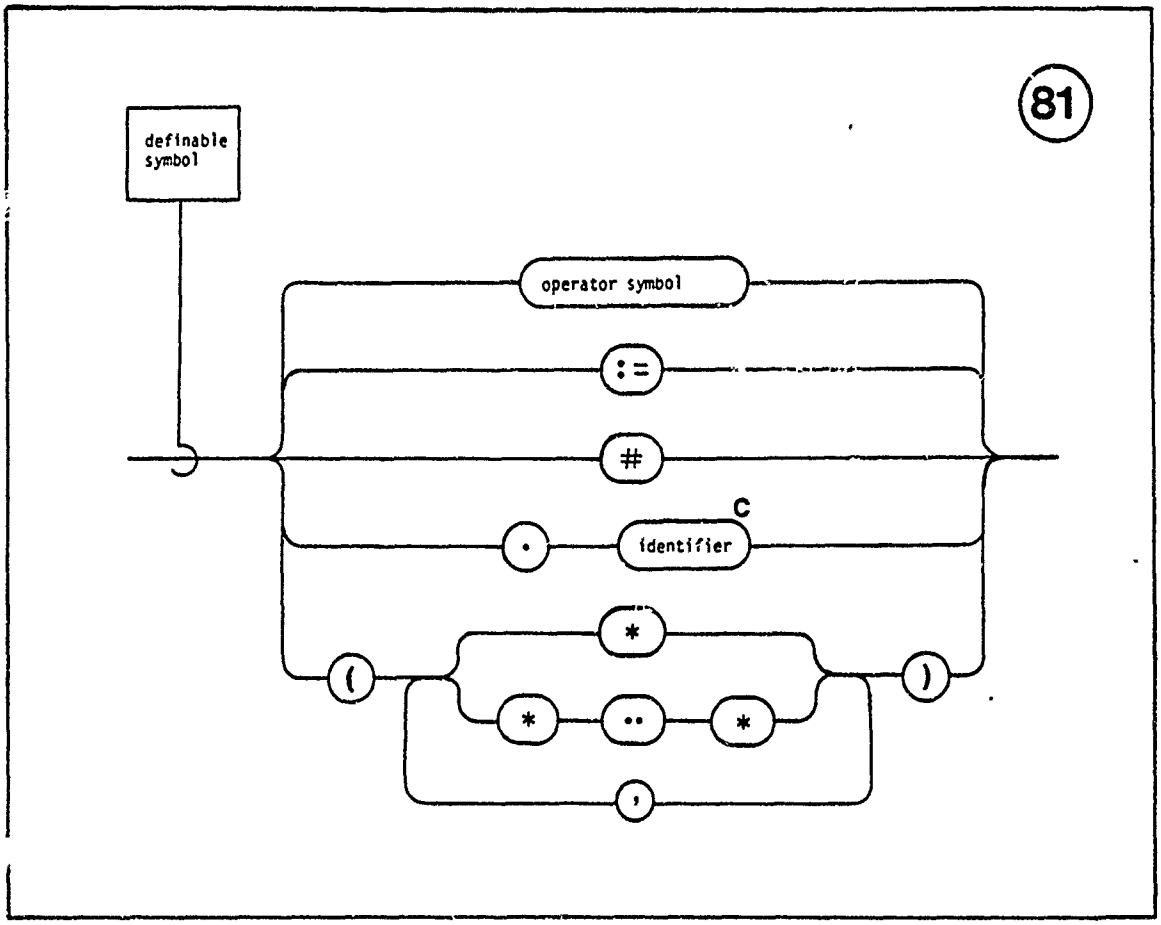
12.4 FOREIGN CODE

Foreign code (i.e., code written in other languages, including assembly language) can be used as part of a RED program if this feature is supported by the translator. This is achieved by converting the code via an appropriate utility into a RED *translation unit*. Information needed about linkage conventions, *types*, representation, etc., are supplied as directives to the utility program. Once this has been done, this new *translation unit* can be exposed by other *translation units* written in RED as though they also had been written in RED. Note, however, that the translator may have to handle these *translation units* in a somewhat different fashion than it would handle RED *translation units*. Note also that some translators may support open (i.e., inline) expansions of foreign code.

13. ADVANCED DEFINITIONS

This chapter discusses ways in which special language forms can be extended to work for data items having user-defined types. These forms include infix and prefix operators, assignment, selectors, and the case and repeat statements. Also included is a discussion of the automatically invoked initialization and finalization operations.

13.1 DEFINABLE SYMBOLS



Definable symbols are used to define functions for special expression forms and assignment procedures.

RULES

The operator symbols >, <=, >= are not definable symbols.

NOTES

Although definable symbols are defined as functions and procedures, they can not be invoked using standard invocation forms. They are involved only when the corresponding special syntactic form is used.

13.2 DEFINITION OF OPERATORS AND ASSIGNMENT

Users can supply definitions for the built-in prefix and infix operator symbols and for assignment. These definitions allow built-in operators to be extended to apply to data items with user-defined types.

RULES

Prefix Operators

A function with a single formal parameter and the name +, -, or NOT defines that prefix operator for a right operand having the type specified for the formal parameter.

Infix Operators

A function with two formal parameters and the name **, *, /, MOD, DIV, &, +, -, =, <, IN, AND, OR, or XOR defines that infix operator for a left operand having the type specified for the first formal parameter and a right operand having the type specified for the second formal parameter.

For the =, <, and IN infix operators, the function result must have subtype BOOL. For = and <, both formal parameters must have the same type.

Associativity

The infix operators +, *, AND, OR, XOR, and & are assumed to be associative. The expression

```
a op b op c
```

can be evaluated as either

```
(a op b) op c
```

or as

```
a op (b op c)
```

where op is one of these infix operators.

Equality and Ordering

The equals (=) infix operator is assumed to define an equivalence relation. In particular, = is assumed to be reflexive (i.e., a=a is true), symmetric (i.e., a=b is equivalent to b=a) and transitive (i.e., if a=b and b=c, then a=c). The operators = and <, taken together, are assumed to define a partial order (with a total order as a special case). In particular, < is assumed to be transitive and it is assumed that, at most, one of

```
a=b or
a<b or
b<a
```

will be true.

The other ordering infix operators (/=, >, <=, >=) cannot be explicitly user-defined. If = is defined, then /= is also automatically defined. If < is defined, then > is also automatically defined. If both = and < are defined, then <= and >= are also automatically defined. In particular,

Purpose	Operator	Definition
not equal	a/=b	NOT(a=b)
greater than	a>b	b<a
less than or equal	a<=b	a=b OR a<b
greater than or equal	a>=b	a=b OR b<a

Assignment

A procedure with two formal parameters and the name := defines assignment. Both formal parameters must have the same type. The first formal parameter corresponds to the left hand side (i.e., the target) and the second formal parameter corresponds to the right hand side (i.e., the source).

The assignment procedure is invoked for

a) the assignment statement

b) initialization

c) CONST and OUT formal parameters

d) constructors

e) message passing via mailboxes

For assignment, the following assumptions are made:

a) Any previous value of the first parameter is lost

b) The only value changed by assignment is the value of the first parameter

c) The value of the first parameter, after assignment, will be equal (i.e., their values can be used interchangeably) to the value of the second parameter before assignment. In particular, for types for which both := and = are defined, after elaboration of

```
a := b;
```

the expression

```
a = b
```

will have value true if a and b are non-overlapping. Side effect?

Overriding Default Definitions

For new types, := and = are automatically defined (see Sections 4.4.2 and 4.4.3). These automatic definitions will not occur if the user provides explicit definitions of := or = local to the same scope in which the type declaration is local.

NOTES

Since CONST and OUT binding depend upon :=, the definition of := should not use these binding classes (if it did, infinite recursion would result).

EXAMPLES

1) Integer sets via linked lists and sharing(default) assignment

```
CAPSULE intsets EXPORTS intset, :=, =, <, emptyintset,
   and, or, xor, in, insert;

   TYPE intset : PTR(i : INT) RECORD[val : INT(i..i),
                                     next : intset];

   % Intsets are immutable sets of integers.
   % Since the list cells are not modified,
   %  assignment can share these cells.
   % Default := is used

   FUNC insert (i : INT, set : intset)  => intset;
      VAR newset : intset;
      IF set = NIL OR set.val > i THEN
         ALLOC newset PTR(i) :=
                     [val : i, next : set];
      ELSE IF set.val = i THEN
         newset := set;
      ELSE
         ALLOC newset PTR(set.val) :=
                      [val : set.val,
                       next : insert(i, set.next)];
      END IF;
      RETURN newset;
   END FUNC insert;

   FUNC in (i : int, set : intset) => BOOL;
      IF set = NIL OR set.val > i THEN
         RETURN FALSE;
      ELSE IF set.val = i THEN
         RETURN TRUE;
      ELSE
         RETURN in (i, set.next);
      END IF;
   END FUNC in;
   ...

   % --- other funcs are similar ---

END CAPSULE intsets;
```

2) Integer sets via linked lists and copying (user-defined) assignment

```
CAPSULE mutable_intsets EXPORTS intset, :=, =, <,
   emptyintset, and, or, xor, in, insert;

   TYPE intset : intlist;

   TYPE intlist : PTR(i : INT) RECORD[val : INT(i..i),
                                      next : intset];

   % Intsets are mutable sets of integers.
   % Assignment must copy the list of cells.

   PROC := (VAR target : intset, READONLY source : intset);
      IF source.ALL = NIL THEN
         target.ALL := NIL;
      ELSE
         VAR newcell : intlist;
         ALLOC newcell PTR(source.val) := source.ALL;
         % Note recursion in assigning the
         %  .next component of source.ALL
         target.ALL := newcell;
      END IF;
   END PROC := ;


   PROC insert (i : INT, VAR set : intset);
      IF set.ALL = NIL OR set.val > i THEN
         VAR old : intlist := set.ALL;
         ALLOC set.ALL ptr(i);
         set.val := i;
         set.next.ALL := old;
      ELSE IF set.val = i THEN
      ELSE
         insert (i, set.next);
      END IF;
   END PROC insert;
         ...
END CAPSULE mutable_intsets;
```

13.3 INITIALIZATION AND FINALIZATION

When a new type is defined, it is possible to specify actions to be taken at the beginning and end of the lifetime of each new data item of that type.

RULES

When a new type is defined, the user may also define the procedures **INITIALIZE** and **FINALIZE** local to the same scope as the type definition. These procedures must each have a single formal parameter of the new type.

When a data item of the new type is created, **INITIALIZE** is automatically invoked. This invocation occurs prior to any explicitly specified initialization, but after the creation (and initialization) of the underlying variable or constant.

The created data item (which is considered to be variable during this invocation) is passed as the actual parameter.

Just before the end of the lifetime of a data item of the new type, **FINALIZE** is automatically invoked. The data item is passed as the actual parameter.

If the new type is defined within a capsule, then **INITIALIZE** and **FINALIZE** need not be exported (they must be exported only if they are to be explicitly invoked outside the capsule).

INITIALIZE and **FINALIZE** are not automatically called for **VAR** and **READONLY** formal parameters since these are not new variables or constants, but rather references to existing variables or constants.

NOTES

Note that INITIALIZE and FINALIZE should not have CONST or OUT parameters (if they do, an infinite recursion will occur).

EXAMPLES

```
CAPSULE stack_cap EXPORTS stack, push, pop;

   ABBREV elem : ... ;
   TYPE list = PTR RECORD[val  : elem,
                          next : list];
   TYPE stack(size : INT) : RECORD[cnt   : INT(0..size),
                                   first : list];
   PROC initialize (VAR s : stack);
      s.cnt := 0;
   END PROC initialize;

   PROC push (VAR s : stack, e : elem);
      ASSERT s.cnt < s.size;
      s.cnt := s.cnt+1;
      ALLOC s.first PTR := [val : e, next : s.first];
   END PROC push;

   ABNORMAL FUNC pop (VAR s : stack) => elem;
      ASSERT s.cnt > 0;
      CONST e := s.first.val;
      CONST n := s.first.next;
      FREE (s.first);
      s.first := n;
      RETURN e;
   END FUNC pop;

   PROC finalize (READONLY s : stack);
      WHILE s.first /= NIL REPEAT
         CONST t := s.first;
         s.first := s.first.next;
         FREE (t);
      END REPEAT;
   END PROC finalize;

END CAPSULE stack_cap;
```

13.4 DEFINITION OF SELECTOR OPERATIONS

Users can define selector operations (subscripting and dot selection) for new types.

RULES

Dot Selection

The dot selection form

```
primary.id
```

will invoke the function with name .id (which may have only a single formal parameter) and will pass primary as the only actual parameter.

Subscripting

The subscripting form

```
primary(exp)
```

will invoke the function with name (*) and will pass primary as the first actual parameter and exp as the second actual parameter. The subscripting form

```
primary(exp1 .. exp2)
```

will invoke the function with name (*..*) and will pass primary as the first actual parameter and exp1 and exp2 as the second and third actual parameters.

For multiple subscripts, the name of the function invoked will correspond to the invocation form. For example

```
primary(exp1, exp2 .. exp3, exp4 .. exp5)
```

will involve the function with name (*, *..*, *..*). The number of formal parameters of a subscripting function must be equal to one (for the primary) plus the number of stars in its name.

EXAMPLES

1) Dot selection

```
CAPSULE cmplx_cap EXPORTS complex, .re, .im, .mag, .ang, ...;

   TYPE complex : RECORD[re, im : FLOAT(5, -100.0 .. 100.0)];

   FUNC .mag (c : complex) => FLOAT(5, -100.0 .. 100.0);
      % definition 1
      RETURN SQRT (c.re**2 + c.im**2);
   END FUNC .mag;

   FUNC .ang (c : complex) => FLOAT(5, -100.0 .. 100.0);
      % definition 2
      RETURN ATAN2 (c.im, c.re);
   END FUNC .ang;

      ...

END CAPSULE cmplx_cap;

EXPOSE ALL FROM cmplx_cap;
VAR c : complex;
...
... c.RE ...     % uses automatic definition
... c.IM ...     % uses automatic definition
... c.MAG ...    % uses definition 1
... c.ANG ...    % uses definition 2
```

2) Subscripting

```
CAPSULE vec_mat EXPORTS scalar, vector,
         matrix, (*), (*,*), (*..*), (*..*, *..*),
         READONLY entire;

   TYPE scalar : FLOAT(10, -1000.0 .. 1000.0);
   TYPE vector (i : INT) : ARRAY INT(1..i) OF scalar;
   TYPE matrix (i,j : INT) : ARRAY INT(1..i), INT(1..j)
                             OF scalar;
   TYPE entirety : INT(0..0);
   VAR entire : entirety;

   FUNC (*..*) (READONLY v1 : vector, a : INT, b : INT)
            => vector(b-a+1);
      % definition 1
      % this overrides the default definition
      %  of vector slicing
      VAR v2 : vector(b-a+1);
      FOR i : INT(a..b) REPEAT
         CONST j := i-a+1
         v2(j) := v1(i);
      END REPEAT;
      RETURN v2;
   END FUNC(*..*);
```

```
    FUNC (*) (READONLY m : matrix, a : entirety, b : INT)
                => vector (INDEXOF(m.ALL));
       % definition 2
       VAR v : vector (INDEXOF(m.ALL));
       FOR i : INDEXOF(m.ALL) REPEAT
          v(i) := m(i, b);
       END REPEAT;
       RETURN v;
    END FUNC (*);

    FUNC (*,*) (READONLY m : matrix, a : INT,
                b : entirety) => vector (INDEXOF(m.ALL, 2));
       % definition 3
       VAR v : vector (INDEXOF(m.ALL, 2));
       FOR i : INDEXOF(m.ALL, 2) REPEAT
          v(i) := m(a, i);
       END REPEAT;
       RETURN v;
    END FUNC (*,*);

    FUNC (*..*, *..*) (READONLY m1 : matrix,
                       a,b : INT, c,d : INT)
                      => matrix (b-a+1, d-c+1);
       % definition 4
       VAR m2 : matrix (b-a+1, d-c+1);
       FOR i : INT(a..b) REPEAT
          CONST j := i-a+1;
          FOR k : INT(c..d) REPEAT
             CONST n := k-c+1;
             m2 (j,n) := m1 (i,k);
          END REPEAT;
       END REPEAT;
       RETURN m2;
    END FUNC (*..*, *..*);

       ...

END CAPSULE vec_mat;

EXPOSE ALL FROM vec_mat;

VAR r = vector(10);
VAR m = matrix(10, 10);
VAR i,j,l,k : INT(1..10);
...

... v(i) ...          % scalar - default definition for vector
... v(i..j) ...       % vector - definition 1
... m(i,j) ...        % scalar - default def for matrix
... m(entire,i) ...   % vector - definition 2
... m(i,entire) ...   % vector - definition 3
... m(i..j, k..n) ... % matrix - definition 4
```

13.5 DEFINITION OF RESOLUTION

Users can define the resolution form of expressions as a way of getting "literals" and "constructors" for new types.

RULES

. The resolution form

exp # t

where t is a type or a subtype is treated as the invocation of the function with name # and with translation time property list [t]. The expression exp is passed as the only actual parameter.

EXAMPLES

```
CAPSULE modint_cap EXPORTS modint, #, ... ;

   TYPE modint (modulo : INT) : INT(0..modulo-1);

   GENERIC s : SUBTYPE(modint)
      FUNC # [s] (i : INT) => s;
         RETURN i MOD s.modulo;
      END FUNC #;
      ...
END CAPSULE modint_cap;

CAPSULE cmpx_cap EXPORTS complex, #, ... ;

   TYPE complex = RECORD[re, im : FLOAT(5, -100.0 .. 100.0)];

   FUNC # [complex] (i : RECORD[re,im : FLOAT]) => complex;
      VAR j : complex;

      j.re := i.re;
      j.im := i.im;
      RETURN j;
   END FUNC #;

   FUNC # [complex] (i : RECORD[mag, ang : FLOAT]) => complex;
      VAR j = complex;
      j.re := i.mag * SIN(i.ANG);
      j.im := i.mag * COS(i.ANG);
      RETURN j;
   END FUNC #;

   ...

END CAPSULE cmpx_cap;
```

13.6 EXTENDING THE CASE STATEMENT TO USER-DEFINED TYPES

RULES

A case statement of the form

```
CASE e1
   WHEN e2, e3..e4 => body1
   WHEN e5         => body 2
END CASE;
```

is expanded to

```
BEGIN
   CONST c := e1;
   CASE TRUE
      WHEN c=e1, e3<=c AND c<=e4 => body1
      WHEN c=e5                  => body2
   END CASE;
END;
```

NOTES

A case statement with only single value labels can be used for any type for which equality (=) is defined. A case statement with range value labels can be used for any type for which equality (=) and ordering (<) are defined.

13.7 EXTENDING THE REPEAT STATEMENT TO USER-DEFINED TYPES

RULES

A repeat statement of the form

```
FOR i : s REPEAT
     body
END REPEAT;
```

is expanded to

```
rep BEGIN
   VAR i : s := s.MIN;
   IF i <= s.MAX THEN
      WHILE TRUE REPEAT
         body     %i is treated as read-only within
                  %  the body
         IF i = s.MAX THEN
            EXIT rep;
         END IF;
         i := SUCC(i);
      END REPEAT;
   END IF;
END rep;
```

A repeat statement of the form

```
FOR i : REVERSE s REPEAT
     body
END REPEAT;
```

is expanded to

```
rep BEGIN
   VAR i : s := s.MAX
   IF i >= s.MIN THEN
      WHILE TRUE REPEAT
         body     % i is treated as read-only
                  %  within the body
         IF i = s.MIN THEN
            EXIT rep;
         END IF;
         i := PRED(i);
      END REPEAT;
   END IF;
END rep;
```

NOTES

Any user defined subtype for which .MIN, .MAX, <, =, and SUCC (or PRED for the reverse form) can be used in the for phase of a repeat statement.

14. LOW LEVEL FACILITIES

This chapter discusses low-level facilities for multitasking and for I/O. It is anticipated that most programmers will use the high-level multitasking facilities (described in chapter 10) and the high-level I/O facilities (described in Appendix A). The low-level facilities are provided for system programmers as tools for building new high-level facilities.

14.1 MORE ABOUT MAILBOXES

Some of the low-level operations on mailboxes that were not described in Section 10.3 are described below. The use of SEND and RECEIVE as waiting invocations is described in Section 14.3.

14.1.1 AVAILABLE COUNTS

If m is a mailbox, then the result of

```
EMPTY_SLOTS (m)
```

is the number of messages that can be sent to m without waiting. This value is the number of empty buffers in m plus the number of unsatisfied receive requests on m. Note that the result of invoking EMPTY_SLOTS can change if messages are sent to m or if any of the receive requests are revoked.

If m is a mailbox, then the result of

```
FULL_SLOTS (m)
```

is the number of messages that can be received from m without waiting. This value is the number of full buffers in m plus the number of unsatisfied send requests on m. Note that the result can change if messages are received from m or if any of the send requests are revoked.

EXAMPLES

```
CAPSULE event_cap EXPORTS pevent,await,signal;
% This capsule defines pulsed events.  When a pulsed
% event is signaled, all awaiting activations continue.

  TYPE pevent : RECORD[ lock : DATA_LOCK,
                        queue : MAILBOX [INT(0..0)] (0) ];

  PROC signal (VAR e : pevent);
      REGION e.lock DO
         WHILE EMPTY_SLOTS (e.queue) > 0 REPEAT
            SEND (e.queue,0);
         END REPEAT;
      END REGION;
   END PROC signal;

   PROC await (VAR e : pevent);
      VAR t : INT(0..0);
      REGION e.lock DO    % this ensures that awaits arriving
      END REGION;         %  after a signal will not continue
      RECEIVE (e.queue,t);
   END PROC await;

END CAPSULE event_cap;
```

14.1.2 CONDITIONAL MESSAGE PASSING

If m is a mailbox, v is a message, and b is a boolean variable, then elaboration of

```
COND_SEND (m,v,b);
```

is similar to elaboration of

```
SEND (m,v)
```

except that when a wait would have been required, no send occurs and b is set to false. If the send can complete without waiting, then b is set to true. There is also a conditional receive procedure of the form

```
COND_RECEIVE (m,v,b);
```

with similar rules.

EXAMPLES

```
% Producer-consumer with busy waits.

VAR m : MAILBOX [s] (5);

TASK produce IMPORTS m;
   VAR d : s;
   ...
   BEGIN    % send with busy wait
      VAR b : BOOL := FALSE;
      WHILE NOT b REPEAT
         COND_SEND (m,d,b);
      END REPEAT;
   END;
   ...
END TASK produce;

TASK consume IMPORTS m;
   VAR d : s;
   ...
   BEGIN     % receive with busy wait
      VAR b : BOOL := FALSE;
      WHILE NOT b REPEAT
         COND_RECEIVE (m,d,b);
      END REPEAT;
   END;
   ...
END TASK consume;

VAR pr,cs : ACT;

SET_PRIORITY (pr,10);
SET_PRIORITY (cs,20);
CREATE consumer NAMED cs;
CREATE producer NAMED pr;
```

14.1.3 ASSIGNMENT AND EQUALITY OF MAILBOXES

Mailboxes are implemented as indirect types. Assignment for mailboxes is a pointer assignment. For the assignment

```
m1 := m2;
```

where m1 and m2 are both mailboxes, the message subtypes of both must be the same; however m1 and m2 need not have the same length. Equality returns true if both mailboxes have equal pointers.

EXAMPLES

```
% message passing with replys.

ABBREV sm : ... ; % message subtype
ABBREV sr : ... ; % reply subtype
ABBREV packet : RECORD[ data  : sm,
                        reply : MAILBOX [sr] (0) ];

VAR m : MAILBOX [packet] (10);

TASK sender (id : INT) IMPORTS m;
   VAR vm : sm; % message variable
   VAR vr : sr; % reply variable
   VAR r : MAILBOX [sr] (1);

   WHILE TRUE REPEAT
      ... % compute vm
      SEND (m, [data:vm, reply:r]); % send message
      RECEIVE (r,vr);               % get reply
      ... % use vr
   END REPEAT;

END TASK sender;

TASK receiver IMPORTS m;
   VAR p : packet;
   VAR vr : sr;   % reply variable
   WHILE TRUE REPEAT
      RECEIVE (m,p); % get message
      ... % process the message(p.data) and compute the reply(vr)
      SEND (p.reply,vr);   % send the reply
   END REPEAT;

END task receiver;

VAR sa : ARRAY INT(1..10) OF ACT;
VAR ra : ACT;

CREATE receiver NAMED ra;
FOR i : INT(1..10) REPEAT
   CREATE sender(i) NAMED sa(i);
END REPEAT;
```

14.1.4 INTERRUPTS

Hardware interrupts are accessed via mailboxes with an INTERRUPT location. The *subtype* of the message associated with a given interrupt are device and implementation dependent. When an interrupt occurs, associated data is collected and sent as a message. The interrupt is then cleared.

EXAMPLES

Suppose a keyboard causes interrupt 16 every time a character is typed. The interrupt is handled as follows.

```
VAR m : MAILBOX [ASCII] (5) LOCATION (INTERRUPT 16);

TASK keyboard_handler IMPORTS m;
   VAR char : ASCII;
   ...
   RECEIVE (m,char); % Get next interrupt
   ...
END TASK keyboard_handler;
```

14.2 MORE ABOUT THE ACT SCHEDULER

14.2.1 ACT VARIABLE STATES

Each ACT variable has a state that consists of three parts:

active - a boolean. Active is true if the ACT variable is associated with an activation that has been been created but is not yet completed, and false otherwise.

waiting - a boolean. Waiting is true if the activation associated with the ACT variable is currently waiting.

suspended - a boolean. An activation whose ACT variable has suspended set to true is not eligible to be run.

When an ACT variable is created, active, waiting, and suspended are automatically set to false.

If a is an act variable, then the following functions produce the current value of each of the three parts of the state.

```
ACTIVE(a)
WAITING(a)
SUSPENDED(a)
```

When an activation is associated with the act variable, the value of ACTIVE is true.

The suspended boolean is explicitly set by the user using the following procedures.

```
SUSPEND (a);      % sets suspended for a to true
UNSUSPEND (a);    % sets suspended for a to false
```

14.2.2 CRITICAL AREAS

When an activation is executing certain particularly critical areas of code it is desirable to

a) prevent the activation from being preempted by some other activation (even if that other activation has a higher priority), and

b) temporarily ignore any EXTERMINATE invocations for the activation.

This can be achieved by elaborating

```
CRITICAL;
```

at the beginning of the critical area and elaborating

```
NONCRITICAL;
```

at its end. When an activation is created it is noncritical. Invocation of CRITICAL makes it critical. Invocation of NONCRITICAL make it again noncritical. When a running activation is critical it is never preempted. When EXTERMINATE is invoked for some critical activation, no action is taken until the activation becomes noncritical; at which time X_TERMINATE is raised.

NOTES

An activation should be critical only for very short sections of code.

14.2.3 SCHEDULING ALGORITHM

This section gives the scheduling rules used by the ACT scheduler.

RULES

An activation is eligible to be run if

a) it is active;
b) it is not waiting;
c) it is not suspended.

If a and b are activations which are both eligible, the priority of a is higher than the priority of b, and b is not critical, then a will be running if b is running.

If a and b are activations which are both eligible, both have the same priority, a became eligible before b, and b is not critical, then a will be running if b is running.

If an activation is both running and critical, then it will continue to run until it becomes noncritical or until it waits.

If there are any activations that are eligible, then at least one of these will be running.

If two or more activations are running, no assumptions can be made about their relative speeds.

14.3 WAITING

There are two kinds of operations used for the wait statement: synchronizing operations associated with waiting invocations and scheduler operations that cause the activation to wait. Such operations can be defined to provide an alternative definition of the *wait statement*.

14.3.1 SYNCHRONIZING OPERATIONS FOR WAITING INVOCATIONS

RULES

For each kind of *waiting invocation* of the form

```
name(e1,e2,...,en)
```

the following five operations must be defined.

a) name_ST - a subtype. A variable with this subtype is created for every waiting invocation that is examined. For example,

```
VAR v : name_ST;
```

b) name_REQUEST (v,e1,e2,...,en) - a procedure. This operation is invoked to request that the waiting invocation be enqueued on the appropriate wait queue. This operation is invoked at most once for each waiting invocation in a wait statement.

c) name_TEST(v,e1,e2,...,en) - a boolean function. The result of this function is true if the waiting invocation can be successfully completed. The name_REQUEST procedure will have been invoked prior to invoking name_TEST.

d) name_COMPLETE(v,e1,e2,...,en) - a procedure. Invocation of this operation completes the waiting invocation. It will be invoked only after name_TEST has produced true.

e) name_REVOKE(v,e1,e2,...,en) - a procedure. This operation is invoked for every waiting invocation for which name_REQUEST has been invoked. No other operations for a particular waiting invocation will be invoked after name_REVOKE.

EXAMPLES

This example shows how the MAILBOX type could have been defined if it were not built-in. Note that this definition has been simplified from the built-in MAILBOX type in three ways.

a) It does not do all error checking.
b) It is not particularly efficient.
c) Not all operations are specified.
d) It does not handle zero-length mailboxes.

This example assumes that a QUEUE[s] data type has been previously defined. Queues are initially empty and have the following operations: INSERT, REMOVE, FIRST (gets first element without removing it), SIZE, and DEQUEUE (removes a specified value from the queue).

```
CAPSULE mailboxes
   EXPORTS mailbox, send, receive, full_slots, empty_slots, cond_send,
           cond_receive, send_ST, send_REQUEST, send_TEST,
           send_COMPLETE,
           send_REVOKE, receive_ST, receive_REQUEST, receive_TEST,
           receive_COMPLETE, receive_REVOKE          ;

   TYPE send_ST : INT(0..0);          % Note that these are dummys
   TYPE receive_ST : INT(0..0);       % In a more efficient
                                      %  implementation, these
                                      %  would actually be
                                      %  elements linked into
                                      %  the mailbox squeue
                                      %  and rqueue.

GENERIC s : SUBTYPE
   TYPE MAILBOX [s] (len:INT) :
           RECORD[ lock : DATA_LOCK,
                   msg  : QUEUE [s],
                   squeue : QUEUE [send_ST],
                   rqueue : QUEUE [receive_ST]  ];

   % Invariants:
   % (1) If SIZE(m.msg)>0 and SIZE(m.rqueue)>0 then
   %     FIRST(m.rqueue) has been sent a SYNC_SIGNAL
   %     or will call receive_TEST before waiting.
   % (2) If SIZE(m.msg)<m.len and SIZE(m.squeue)>0
   %     then FIRST(m.squeue) has been sent a SYNC_SIGNAL
   %     or will call send_TEST before waiting.

GENERIC t : TYPE
   ABNORMAL FUNC empty_slots (READONLY m : MAILBOX[t] ) => INT;
      RETURN (m.len-SIZE(m.msg)) + SIZE(m.rqueue);
   END FUNC empty_slots;
```

```
GENERIC t : TYPE
   ABNORMAL FUNC full_slots (READONLY m : MAILBOX[t] ) => INT;
      RETURN SIZE(m.msg) + SIZE(m.squeue);
   END FUNC full_slots;

GENERIC t : TYPE
   PROC send_REQUEST ( READONLY q : send_ST,
                       VAR m : MAILBOX [t],
                       READONLY v : t          );
      REGION m.lock DO
         INSERT (m.squeue,ME);   % needn't wake up self
      END REGION;
   END PROC send_REQUEST;

GENERIC t : TYPE
   ABNORMAL FUNC send_TEST ( READONLY q : send_ST,
                             READONLY m : MAILBOX [t],
                             READONLY v : t           )
                         => BOOL;
      RETURN FIRST (m.squeue)=ME  AND
                SIZE (m.msg) < m.len       ;
   END FUNC send_TEST;

GENERIC t : TYPE  NEEDS :=(t,t)
   PROC send_COMPLETE ( READONLY q ; send_ST,
                        VAR m : MAILBOX [t],
                        READONLY v : t         );
      REGION m.lock DO
         INSERT (m.msg,v);     % may invalidate invariant 1
         IF SIZE(m.msg)=1 AND SIZE(m.rqueue)>0 THEN
            SYNC_SIGNAL (FIRST(m.rqueue));
         END IF;
      END REGION;
   END PROC send_COMPLETE;

GENERIC t : TYPE
   PROC send_REVOKE ( READONLY q : send_ST,
                      VAR m : MAILBOX [t],
                      READONLY v : t          );
      REGION m.lock DO
         DEQUEUE (m.squeue,q);   % may disrupt invariant 2
         IF SIZE(m.squeue)>0 AND SIZE(m.msg)<m.len THEN
            SYNCH_SIGNAL (FIRST(m.squeue));
         END IF;
      END REGION;
   END PROC send_REVOKE;

GENERIC t : TYPE
   PROC receive_REQUEST ( READONLY q : receive_ST,
                          VAR m : MAILBOX [t],
                          READONLY v : t          );
      REGION m.lock DO
         INSERT (m.rqueue,ME);
      END REGION;
   END PROC receive_REQUEST;
```

```
GENERIC t : TYPE  NEEDS :=(t,t)
   ABNORMAL FUNC receive_TEST ( READONLY q : receive_ST,
                                READONLY m : MAILBOX [t],
                                READONLY v : t           ) => BOOL;
      RETURN FIRST(m.rqueue)=ME AND 0<SIZE(m.msg);
   END FUNC receive_TEST;

GENERIC t : TYPE  NEEDS :=(t,t)
   PROC receive_COMPLETE ( READONLY q : receive_ST,
                           VAR m : MAILBOX [t],
                           VAR v : t            );
      REGION m.lock DO
         REMOVE (m.msg,v);     % may disrupt invariant 2
         IF SIZE(m.msg)=m.len-1 AND 0<SIZE(m.squeue) THEN
            SYNC_SIGNAL (FIRST(m.squeue));
         END IF;
      END REGION;
   END PROC receive_COMPLETE;

GENERIC t : TYPE  NEEDS :=(t,t)
   PROC receive_REVOKE ( READONLY q : receive_ST,
                         VAR m : MAILBOX [t],
                         READONLY v : t           );
      REGION m.lock DO
         DEQUEUE (m.rqueue,q);     % may disrupt invariant 1
         IF SIZE(m.rqueue)>0 AND SIZE(m.msg)<m.len THEN
            SYNC_SIGNAL (FIRST(m.rqueue));
         END IF;
      END REGION;
   END PROC receive_REVOKE;

GENERIC t : TYPE  NEEDS :=(t,t)
   PROC s   : ( VAR m : MAILBOX [t],
                READONLY v : t          );
      WA
         EN send (m,v) =>   % this send is not an invocation
      END WAIT;
   END PROC send;

GENERIC t : TYPE  NEEDS :=(t,t)
   PROC receive ( VAR m : MAILBOX [t],
                  VAR v : t          );
      WAIT
         WHEN receive (m,v) => % this receive is not an invocation
      END WAIT;
   END PROC receive;
```

```
GENERIC t : TYPE NEEDS := (t,t)
   PROC cond_send ( VAR m : MAILBOX [t],
                    READONLY v : t,
                    VAR b : BOOL        );
      VAR q : send_ST;
      send_REQUEST (q,m,v);
      IF send_TEST (q,m.v) THEN
         send_COMPLETE (q,m,v);
         b := TRUE;
      ELSE
         b := FALSE;
      END IF;
      send_REVOKE (q,m,v);
   END PROC cond_send;

GENERIC t : TYPE NEEDS ::= (t,t)
   PROC cond_receive ( VAR m : MAILBOX [t],
                       VAR v : t,
                       VAR b : BOOL         );
      VAR q : receive_ST;
      receive_REQUEST (q,m,v);
      IF receive_TEST (q,m.v) THEN
         receive_COMPLETE (q,m,v);
         b := TRUE;
      ELSE
         b := FALSE;
      END IF;
      receive_REVOKE (q,m,v);
   END PROC cond_receive;
END CAPSULE mailboxes;
```

14.3.2 ACT SCHEDULER OPERATIONS FOR WAITING

There are three ACT scheduler operations for waiting.

a) SYNC_RESET - a procedure. This procedure is invoked immediately before the name_TEST functions are checked. Its effect is to override all previous SYNC_SIGNALs.

b) SYNC_WAIT - a procedure. This procedure is invoked after all name_TEST functions have produced false. Waiting occurs if SYNC_SIGNAL has not been invoked for this activation since the last time that SYNC_RESET was invoked.

c) SYNC_SIGNAL(a) - a procedure (where a is a variable with type ACT). This procedure is invoked to indicate that the result of one of the name_TEST operations of the specified activations will be different when next invoked. If the activation is waiting, as a result of performing SYNC_WAIT, it will be awakened.

NOTES

Extra invocations of SYNC_SIGNAL will not cause erroneous behavior since the name_TEST functions are again invoked before completion of any *waiting invocation*.

14.3.3 EXPANSION OF THE WAIT STATEMENT

The wait statement

```
WAIT
   WHEN send(m1,v), send(m2,v)  => body1
   WHEN DELAY (10*SECONDS)      => body2
END WAIT;
```

can be expanded to

```
BEGIN
   VAR q1 : SEND_ST;
   VAR q2 : SEND_ST;
   VAR q3 : DELAY_ST;
   VAR i : INT(1..2);

   SEND_REQUEST (q1,m1,v);
   SEND_REQUEST (q2,m2,v);
   DELAY_REQUEST (q3,10*SECONDS);

   loop WHILE TRUE REPEAT
      SYNC_RESET;
      CASE TRUE
         WHEN SEND_TEST (q1,m1,v) =>
            SEND_COMPLETE (q1,m1,v);
            i:=1;
            EXIT loop;
         WHEN SEND_TEST (q2,m2,v) =>
            SEND_COMPLETE (q1,m2,v);
            i:=1;
            EXIT loop;
         WHEN DELAY_TEST (q3,10*SECONDS) =>
            DELAY_COMPLETE (q3,10*SECONDS);
            i:=2;
            EXIT loop;
         ELSE =>
            SYNC_WAIT;
      END CASE;
   END REPEAT loop;

   SEND_REVOKE (q1,m1,v);
   SEND_REVOKE (q2,m2,v);
   DELAY_REVOKE (q3,10*SECONDS);

   CASE i
      WHEN 1 => body1
      WHEN 2 => body2
   END CASE;

END;
```

Note that this is only one of the possible expansions. Translators are free to choose any expansion that is consistent with the semantics of the language.

14.4 MORE ABOUT MUTUAL EXCLUSION

14.4.1 SEMANTICS OF THE REGION STATEMENT

RULES

A region statement has the form

```
REGION exp DO
   body
END REGION;
```

and is roughly equivalent to

```
LOCK (exp);
GUARD
   body
BY ELSE =>
   UNLOCK (exp);
   RERAISE;
END GUARD;
UNLOCK (exp);
```

Note that LOCK is invoked at entry to the region and that UNLOCK is invoked at exit from the region (even when the exit occurs as the result of an unhandled exception). Recall however, that a region statement guarantees that the region will be unlocked no matter how it is left. There are two cases where the rough expansion must be further refined

a) Exits and gotos out of the body of a region must also cause UNLOCK to be invoked.

b) If an EXTERMINATE occurs at certain critical points (such as after the LOCK but before the GUARD), unlocking will not occur in the rough expansion. This problem is avoided by making certain parts of the expansion be run as critical, so any exterminates will be deferred. In particular, invocations of LOCK and UNLOCK will be run as critical, while the elaboration of the body will not be critical.

NOTES

A *variable* with any *type* for which LOCK and UNLOCK procedures are defined can be used in the *region statement*.

EXAMPLES

This example shows how monitors can be defined in the language. First an example is given of how monitors can be used.

```
CAPSULE line_buffer EXPORTS send, receive;
   VAR lock : MONITOR_LOCK;
   VAR contents : line;
   VAR full : BOOL;
   VAR sender, receiver : MONITOR_QUEUE (lock);

   PROC receive (OUT text : line) IMPORTS ALL;
      REGION lock DO
         IF NOT full THEN
            DELAY (receiver);
         END IF;
         text := contents;
         full := FALSE;
         CONTINUE (sender);
      END REGION;
   END PROC receive;

   PROC send (text : line) IMPORTS ALL;
      REGION lock DO
         IF full THEN
            DELAY (sender);
         END IF;
         contents := text;
         full := TRUE;
         CONTINUE (receiver);
      END REGION;
   END PROC send;

END CAPSULE line_buffer;
```

The two data types needed for monitors, MONITOR_LOCK and MONITOR_QUEUE, together with their operations are defined as follows.

```
CAPSULE monitor_defs EXPORTS monitor_lock, monitor_queue,
                             lock, unlock,
                             delay, continue, :=          ;

   TYPE monitor_lock : PTR RECORD[external, urgent : DATA_LOCK];

   TYPE monitor_queue (lock : monitor_lock) : DATA_LOCK;

   PROC initialize (VAR m : monitor_lock);
      ALLOC m PTR;
      LOCK (m.urgent);  % lock the urgent queue so that the next
                        %  activation to lock will wait

   END PROC initialize;

   PROC initialize (VAR mq : monitor_queue);
      LOCK (mq.ALL);      % lock the monitor queue so that the next
                          %  activation to lock will wait

   END PROC initialize;

   PROC lock (VAR m : monitor_lock);
      LOCK (m.external);  % lock the main lock
   END PROC lock;

   PROC unlock (VAR m : monitor_lock);
      IF EXCESS_LOCKS (m.urgent) > 1 THEN
         UNLOCK (m.urgent);    % release next activation from
                               %  the urgent queue
      ELSE
         UNLOCK (m.external);   % unlock the main lock
      END IF;
   END PROC unlock;

   PROC continue (VAR mq : monitor_queue);
      IF EXCESS_LOCKS (mq.ALL) > 1 THEN
         UNLOCK (mq.ALL);          % release the next activation
                                   %  waiting on mq
         LOCK (mq.lock.urgent);    % wait on the urgent queue
      END IF;
   END PROC continue;

   PROC delay (VAR mq : monitor_queue);
      unlock (mq.lock);   % leave the region
      LOCK (mq.ALL);      % wait on the mq queue
   END PROC delay;

END CAPSULE monitor_defs;
```

14.4.2 MORE ABOUT DATALOCKS

In addition to lock and unlock, data lock variables also have operations EXCESS_LOCKS and OWNER. If d is a data lock variable, then the result of

```
EXCESS_LOCKS (d)
```

is the number of times that LOCK(d) has been invoked (but not necessarily completed) minus the number of times UNLOCK(d) has been called. The values that result can be interpreted as follows:

a) 0 - The lock is unlocked.
b) 1 - The lock is locked and there are no activations waiting.
c) >1 - There are activations waiting, attempting to lock the data lock.

If d is a data lock variable, then the result of

```
OWNER (d)
```

is the activation that locked the data lock or NIL_ACT if the data lock is unlocked.

14.4.3 LATCH

Latches are the basic low-level synchronization type. Since LOCK and UNLOCK procedures are available for latches, they can be used in the region statement.

RULES

Latches are either locked or unlocked. Elaboration of

```
LOCK (t);
```

(where t is a variable with type LATCH) will lock t if t is unlocked; otherwise, it will wait until t becomes unlocked. If there are several activations waiting to lock a latch, which one will actually succeed when the lock becomes unlocked is undefined. Elaboration of

```
UNLOCK (t);
```

will unlock latch t. Elaboration of

```
COND_LOCK (t,b);
```

where b is a boolean variable, will lock t if t is unlocked and set b to true; otherwise, t is not changed and b is set to false.

NOTES

The only virtue of latches is that they have an inexpensive implementation. In many cases, the waiting can be done as busy waiting.

14.5 USER-DEFINED SCHEDULERS

In addition to the built-in priority ACT schedulers, users can define their own schedulers. The scheduler to be used for an activation is determined, when the activation is created, by the type of the activation variable specified (after NAMED) in the task invocation statement. User-defined schedulers are capsules that define the type for their activation variables and a set of basic scheduling operations. The operations are implemented in terms of invocation of the fundamental operations of the built-in ACT scheduler. Synchronization schemes (including DATA_LOCK and MAILBOXes) are designed in such a way that they will work correctly with any scheduler (i.e., they are independent of the particular scheduler used).

14.5.1 ACT ASSIGNMENT AND EQUALITY

ACT is implemented as an indirect type. Assignment for variables with an ACT type is a pointer assignment. The assignment

```
a1 := a2;
```

where a1 and a2 are both ACT variables, sets a1 to point to the same information as a2. Equality returns true if both variables point to the same information. There is also a constant, NIL_ACT, which has type ACT and is a nil pointer.

NOTES

ACT assignment and equality are useful for creating queues of act variables.

14.5.2 LOW AND HIGH OPERATIONS

RULES

When an activation exists that is controlled by a user-defined scheduler, there are really two schedulers involved: the user-defined scheduler and the underlying ACT scheduler. There are also two activation variables involved, one for each scheduler. For example,

```
VAR ua : USER_ACT;
CREATE t NAMED ua;
```

creates an activation of task t, which is controlled by the USER_ACT scheduler. The two activation variables here are

a) ua -- The activation variable for the user-defined USER_ACT scheduler.

b) Another variable of type ACT (which is not explicitly named, but for reference purposes call it aa). The variable aa is the one by which the ACT scheduler controls scheduling of the activation.

The result of elaborating

```
ME
```

during elaboration of the activation of t will be aa and the result of elaborating

```
ME [USER_ACT]
```

during elaboration of the activation of t will be ua. The X_SCHED exception is raised if the activation variable of the invoking activation does not have type USER_ACT. The function ME is called a low-level operation (accessing the underlying ACT scheduler information) and the function ME [USER_ACT] is called a high-level operation (accessing the specified activation variable ua).

Variable ua normally contains sufficient information for the USER_ACT scheduler to find aa (actually the information that aa points to). This can be done by including aa as a component of ua or by making ua an index into some scheduling queue that contains aa.

There are also two sets of scheduling operations available: one high-level set for the USER_ACT scheduler and another low-level set for the ACT scheduler. The high-level USER_ACT operations are:

```
SYNC_RESET(ua);        %.used for waiting
SYNC_WAIT(ua);         % used for waiting
SYNC_SIGNAL(ua);       % used to end waiting
TASK_END(ua);          % invoked at the completion of the
                       %  activation
```

These all invoke procedures defined for the USER_ACT scheduler. The low-level operations are:

```
LOW_SYNC_RESET;
LOW_SYNC_WAIT;
LOW_SYNC_SIGNAL(a);
LOW_TASK_END(a);
```

These all invoke procedures defined for the ACT scheduler. The high-level operations will normally

invoke these low-level operations to actually achieve the scheduling.

To allow synchronization schemes to be independent of any particular scheduler, there is a way of getting from ACT variable aa to the high-level operations upon the USER_ACT variable ua. This is achieved by the following ACT scheduler operations:

```
SYNC_RESET;        % invokes SYNC_RESET(ua);
SYNC_WAIT;         % invokes SYNC_WAIT(ua);
SYNC_SIGNAL(aa);   % invokes SYNC_SIGNAL(ua);
TASK_END(aa);      % invokes TASK_END(ua);
```

The way in which these operations are achieved is described in the next subsection. These operations allow synchronization operations to be ignorant of the particular scheduler that is being used. For example, during the elaboration of the activation of t, invocation of

```
SYNC_WAIT;
```

will cause the invocation

```
SYNC_WAIT(ua);
```

to occur. This means that the synchronization operations need not directly invoke

```
SYNC_WAIT(ME[USER_ACT]);
```

If this had been necessary, the synchronization operation would have needed knowledge of the type USER_ACT.

Note that when activations are scheduled by the ACT scheduler (i.e., no user-defined scheduler is involved), then the following operations are equivalent:

```
SYNC_RESET;        % is equivalent to LOW_SYNC_RESET
SYNC_WAIT;         % is equivalent to LOW-SYNC_WAIT
SYNC_SIGNAL(a);    % is equivalent to LOW_SYNC_SIGNAL(a)
TASK_END(a);       % is equivalent to LOW_TASK_END(a)
```

14.5.3 TASK ACTIVATION CREATION AND COMPLETION

RULES

TASK INVOCATION STATEMENT

The *task invocation statement* has the form

```
CREATE t(e1,e2,...,en) NAMED av;
```

The effect of elaborating this statement is

a) create a new variable of type ACT (for reference call it aa).

b) create a new activation of t, place information about it (e.g., starting address, stack locations, save area, etc.) in aa.

c) bind the actual parameters (e1,e2,...,en) to this activation.

d) set up the following procedures

```
PROC SYNC_RESET;
   SYNC_RESET(av);
END PROC SYNC_RESET;

PROC SYNC_WAIT;
   SYNC_WAIT(av);
END PROC SYNC_WAIT;

PROC SYNC_SIGNAL;
   SYNC_SIGNAL(av);
END PROC SYNC_SIGNAL;

PROC TASK_END;
   TASK_END(av);
END PROC TASK_END;
```

and place the address of each in aa. This allows access to the high-level scheduler, given only the low-level activation variable aa.

e) elaborate the following procedure invocation:

```
TASK_START(av,aa);
```

For activations scheduled by the ACT scheduler (i.e., when av has type ACT), this invokes the ACT scheduler operation that starts up the activation. For activations scheduled by a user-defined scheduler, this invokes the operation required for that scheduler.

TASK COMPLETION

When a task activation has completed the elaboration of the task body, the invocation

```
TASK_END(ME);
```

is elaborated. This allows the scheduler to remove the activation from its queues.

14.5.4 EXAMPLE - A ROUND ROBIN SCHEDULER

```
CAPSULE round_robin (time_slice : INT) EXPORTS rr_act,
            task_start, task_end, sync_reset,
            sync_wait, sync_signal;

   CONST max := 256;      % maximum number of activations to
                          %  be scheduled
   TYPE rr_act : INT(1..max);

   VAR n : INT(0..max) := 0;   % index of last q entry in use
   VAR curr : INT(1..max) := 1;   % index of activations
                                  %  currently scheduled
   VAR dlock : DATA_LOCK;
   VAR asched : ACT;           % scheduler activation
   VAR q : ARRAY INT(1..max) OF ACT;  % activations to be
                                      %  be scheduled

   PROC task_start (VAR r : rr_act, VAR a : ACT) IMPORTS ALL;
      REGION dlock;
         ASSERT n < max;
         n := n + 1;
         q(n) := a;
         r.ALL := n;
         SUSPEND (a);
         TASK_START (a,a);
         SYNC_SIGNAL (asched);
      END REGION;
   END PROC task_start;

   PROC task_end (VAR r : rr_act) IMPORTS ALL;
      CONST i := r.ALL;
      VAR victim : act;
      REGION dlock DO;
         victim := q(i);
         q(i) := NIL_ACT;
      END REGION;
      LOW_TASK_END (victim);
   END PROC task_end;
```

```
        PROC sync_wait (VAR r : rr_act) IMPORTS asched;
           LOW_SYNC_WAIT;
           SYNC_SIGNAL (asched);  % signal that a non-waiting
                                  %  activation is available
        END PROC sync_wait;

        PROC sync_reset (VAR r : rr_act);
           LOW_SYNC_RESET;
        END PROC sync_reset;

        PROC sync_signal (VAR r : rr_act)  IMPORTS q;
           LOW_SYNC_SIGNAL (q(r.ALL));
        END PROC sync_signal;

        TASK scheduler IMPORTS ALL;
           WHILE TRUE REPEAT
              REGION dlock DO
                 VAR new : INT(1..max+1) :=  curr;
                 % search for next available activation
                 loop WHILE TRUE REPEAT
                     new := new + 1;
                     IF new > n THEN
                        new := new - n;
                     END IF;
                     IF q(new) /= NIL_ACT AND NOT WAITING (q(new)) THEN
                       EXIT loop;   % found one
                     END IF;
                     IF new = curr THEN  % all waiting or dead
                       SYNC_RESET;
                       UNLOCK (dlock);       % leave region
                          SYNC_WAIT;         % wait for a new task_start
                                             %  the end of a sync_wait
                       LOCK (dlock);         % reenter region
                    END IF;
                 END REPEAT loop;
                 IF new /= curr THEN
                    % schedule next activation
                    IF q(curr) /= NIL_ACT THEN
                       SUSPEND (q(curr));
                    END IF;
                    UNSUSPEND (q(new));
                    curr := new;
                 END IF;
              END REGION;
              DELAY (time_slice);
           END REPEAT;
        END TASK scheduler;

        q(1) := NIL_ACT;
        SET_PRIORITY (asched, 255);
        CREATE scheduler NAMED asched;

     END CAPSULE round_robin;
```

14.6 LOW-LEVEL I/O

Low-level I/O is provided to allow low-level access to the most primitive level of I/O provided in the target computer. The high-level I/O described in Appendix A can be defined using low-level I/O, together with the interrupt handling facilities described in Section 14.1.4.

RULES

Invocation of

```
LOW_IO (device, command, data);
```

will cause the specified command to be issued for the specified device, using the specified data. If there is no data needed, the invocation

```
LOW_IO (device, command);
```

may also be available. The types of each of the parameters is implementation-dependent. The effect of LOW_IO is implementation-dependent.

EXAMPLE

1) 360 example

```
TYPE CCWPTR : PTR CCW;
VAR CAW : CCWPTR LOCATION (72);
VAR STATUS : IO_STATUS;

CAW := ...                          % compute address of CCW
LOW_IO (device, 'SIO);              % executes an SIO for the
                                    %  specified device
...
LOW_IO (device, 'TIO, STATUS);      % executes a TIO for the
                                    %  specified device and
                                    %  returns result
                                    %  in STATUS
```

A. HIGH-LEVEL I/O

This appendix discusses a recommended set of definitions for user-level input-output programming. The definitions in this set include file types and the procedures and functions which operate on files. Files may be defined and associated with external physical files or devices. Physical files may be created, deleted, and renamed. The files may be organized for either sequential or random access and may be defined as either input, output, or update files. Files may be read or written and positioned to any component. Files may be interrogated to determine current size or position or to ascertain if the current position is at the end of the file. Additional file processing facilities are provided for files whose components are characters (i.e., have an ENUM type). The facilities allow strings of multiple characters to be read or written by a single invocation.

A.1 FILE VARIABLES, OPEN, AND CLOSE

A file is a variable having a FILE type. A file is associated with an ordered collection of components, all having the same type. The type property list specifies the type of the components and the constraint property list specifies the access method (sequential or random) and the file use (input, output, or update). For example,

```
VAR int_file : FILE [INT(min..max)] ('SEQ, 'INPUT);
```

defines int_file as a sequential file whose components are integers in the range min through max. The file is usable for input processing only. Note that the type of int_file is FILE [INT].

Before any file processing can occur, the file must be associated with a physical file. A physical file can be a file on a storage medium, a physical device, or a file on a storage medium representing a physical device (for example, a spooled card reader file). The association is made by the OPEN procedure, which also serves to specify the initial state of the file (old or new). If the initial state of the file was new, a file is created and its name placed in the file directory. If processing is attempted before a file is opened, an exception is raised. Assignment is not defined for file variables.

After all file processing has been completed, a file may be closed by the CLOSE procedure, which severs the association with the physical file and specifies subsequent file disposition (save or delete).

Although old, new, save, and delete do not apply to all physical devices, they are required for files associated with such devices. If the initial state or final disposition do not make sense, they are ignored. Requiring this information makes the program more portable, since physical devices are commonly represented by files on a storage medium. A file associated with a card reader, for example, can later be associated with a spooled card reader. Typical examples of file use are shown below:

1) File associated with file on storage medium

```
VAR int_file : FILE [INT(min..max)] ('SEQ, 'INPUT);

OPEN (int_file, "ABC", 'OLD);
... % file processing
CLOSE (int_file, 'SAVE);
```

2) File associated with physical device

```
VAR printer : FILE [ASCII] ('SEQ, 'OUTPUT);
OPEN (printer, "00E", 'OLD);
... % text printing
```

```
CLOSE (printer, 'SAVE);
```

Sequential and Random Files

A sequential file is a file which is processed by accessing each component in succession. A random file is a file which permits direct acces to any component regardless of its position with respect to other components. Not all processing procedures and functions are valid on sequential files (i.e., a card reader cannot be backed up to the start of the file after some reading has occurred).

Text and Standard Files

Any file whose components have an enumeration type (i.e., ASCII or some other character type) is called a text file. All other files are called standard files. There are two predefined text files, SYS_IN and SYS_OUT, which are files of ASCII. A set of procedures and functions is available for reading and writing text files in a line-oriented manner. Formatting is achieved by combining text files with string-conversion functions.

```
VAR text_file : FILE [ASCII] ('SEQ, 'INPUT);
OPEN (text_file, "SYSTEM", 'OLD);
...  % text file processing
CLOSE (text_file, 'DELETE);
```

File Use

An input file is a file which is read-only. An output file is a file which is write-only. An update file is a file which may be read or written.

A.2 FILE PROCESSING

File processing is accomplished by the READ, WRITE, SET_POSITION, and FILE_RENAME procedures and the SIZE, POSITION, and EOF functions.

READ and WRITE

READ moves data from an input or update file to a *variable* defined to be of the component subtype. After the READ, the file position is increased by one. WRITE moves data from a *variable* of the component subtype to an output or update file. After the WRITE, the file position is increased by one. If the data was written at the end of the file, the size of the file is increased by one.

```
VAR int_file : FILE [INT(0..100)] ('SEQ, 'INPUT);
OPEN (int_file, "ABC", 'OLD);
FOR i : INT(1..25) REPEAT   % process first 25 components
   VAR temp : INT(0..100);
   READ (int_file, temp);
   process (temp);
END REPEAT;
CLOSE (int_file, 'DELETE);
```

POSITION

POSITION returns the current file position. If the file is positioned at the start of the file (before the first component), POSITION will return 1. When a file is opened, the position will be 1. When a file is closed, the position is after the last record. If the file is positioned at the end of the file (after the last component), POSITION will return the number of components in the file plus one.

SET_POSITION

SET_POSITION moves a file to a specified position. If an attempt is made to position the file before 1 or past the end of the file, an exception is raised.

The procedure invocation

```
SET_POSITION (any_file, 1);
```

is equivalent to a rewind command.

The procedure invocation

```
SET_POSITION (any_file, POSITION(any_file) - 1);
```

is equivalent to a backspace command.

SIZE

SIZE returns the number of components in a file. If the file is empty, SIZE returns zero. The following example positions a file to its end and writes a new component.

```
SET_POSITION (any_file, SIZE(any_file) + 1);
WRITE (any_file, temp);
```

EOF

EOF returns a boolean value signifying whether the position is at the end of a file. If the file is positioned at its end, EOF returns TRUE. For example,

```
EOF (any_file)
```

FILE_RENAME

FILE_RENAME, in the presence of a file system, changes the name of an existing physical file. Any associations made between files and that physical file continue to exist. The old file name, however, can no longer be used.

```
OPEN (int_file, "ABC", 'OLD);
FILE_RENAME ("ABC", "XYZ");       % int_file is still associated
                                  %  with the same physical file,
                                  %  which is now known by
                                  %  another name
OPEN (int2_file, "ABC", 'OLD);    % illegal, file "ABC" is now
                                  %  unknown
```

EXAMPLES

1) Copies entire file

```
VAR in_file : FILE [INT(1..100)] ('SEQ, 'INPUT);
VAR out_file : FILE[INT(1..100)] ('SEQ, 'OUT);

OPEN (in_file, "ABC", 'OLD);
OPEN (out_file, "XYZ", 'NEW);

WHILE NOT EOF(in_file) REPEAT
   VAR temp : INT(1..100);
   READ (in_file, temp);
   WRITE (out_file, temp);
END REPEAT;

CLOSE (in_file, 'DELETE);
CLOSE (out_file, 'SAVE);
```

2) Sequential update processing

```
VAR update_file : FILE [INT(1..100)] ('SEQ, 'UPDATE);

OPEN (update_file, "ABC", 'OLD);

WHILE NOT EOF(update_file) REPEAT
   VAR temp : INT(1..100);
   READ (update_file, temp);
   temp := temp + 1;   % change data
   SET_POSITION (update_file,
                 POSITION(update_file) - 1);
   WRITE (update_file, temp);   % write changed data
                                % over old component
```

```
END REPEAT;

CLOSE (update_file, 'SAVE);
```

3) Copies entire file n times

```
% file OPENs and CLOSEs occur outside procedure to
%  avoid trying files to particular physical files
PROC copy (n : INT(1..10));
   FOR i : INT(1..n) REPEAT
      WHILE NOT EOF(in_file) REPEAT
         VAR temp : INT(1..100);
         READ (in_file, temp);
         WRITE (out_file, temp);
      END REPEAT;
      SET_POSITION (in_file, 1);   % rewind
   END REPEAT;
END PROC copy;
```

Implementation-Dependent Procedures and Functions

It is anticipated that a variety of implementation and/or device-dependent procedures and functions will be available for setting and interrogating file characteristics such as protection, physical blocking, buffering, and file space.

EXAMPLES

1) 370 VS example

```
VAR lib : FILE[ASCII] ('SEQ, 'INPUT);

SET_VOL (lib, "PRIVATE,RETAIN,SER=LIB003");
SET_UNIT (lib, "2314");
SET_DISP (lib, "SHR");

OPEN (lib, "XCOM.DTA3", 'OLD);
```

2) PDP-10 TOPS 10 example

```
VAR data : FILE[INT(1..50)] ('SEQ, 'OUTPUT);

SET_PRT (data, "<057>");     % set protection code
SET_VER (data, 4);           % version 4

OPEN (data, "DSKC:mydata.dat[10,50]", 'NEW);
```

A.3 TEXT FILES

A text file is a file whose components have an enumeration type. This enumeration type will specify a character set, usually ASCII. The following example defines a text file which might be a card reader.

```
VAR card_file : FILE [ASCII] ('SEQ, 'INPUT);
OPEN (card_file, "ABC", 'OLD);
...
CLOSE (card_file, 'DELETE);
```

Two text files are predefined, SYS_IN and SYS_OUT. These must be opened to associate them with an appropriate physical file for input and output.

File Processing

It is possible to perform character by character processing of a text file using the file processing procedures and functions already described. It is usually more convenient, however, to think of a text file as a sequence of lines rather than characters, where a line consists of the string of all characters up to, but not including, the carriage-return line-feed characters ('CR & 'LF). For example, if a text file contains

```
"ABC" & 'CR & 'LF & "DEFGH" & 'CR & 'LF
```

then that text file is considered to be composed of two lines, "ABC" and "DEFGH". The *type* of the characters making up the line are the same as the *type* of the file component. As shown in the above example, lines within the same file may be of different lengths.

The READLN procedure and the WRITELN procedure are provided to read lines from and write lines to text files. Also provided are: a SIZELN function to interrogate the size of the line about to be read; definitions of READLN, SIZELN, and EOF which assume that the file name is the predefined file SYS_IN; and a definition of WRITELN which assumes that the file name is the predefined file SYS_OUT.

READLN and WRITELN

READLN reads a line from an input or update text file into a variable. After the READLN, the file position is increased by the number of components contained in the line just read plus the carriage return and line feed. WRITELN writes a line from a variable of the component subtype into an output or update text file. After the WRITELN, the file position is increased by the number of components in the line just written plus the carriage return and line feed.

SIZELN

SIZELN returns the number of characters between the current position and the first end of line ('CR & 'LF).

EXAMPLES

1) Echo input to output, one line at a time, using SYS_IN and SYS_OUT.

```
OPEN (SYS_IN, "ABC", 'OLD);
OPEN (SYS_OUT, "XYZ", 'OLD);

WHILE NOT EOF REPEAT
   VAR line : STRING[ASCII] (SIZELN);
   READLN (line);
   WRITELN (line);
END REPEAT;

CLOSE (SYS_IN, 'SAVE);
CLOSE (SYS_OUT, 'SAVE);
```

2) Echo input to output, one line at a time, using user-defined text files.

```
VAR in_file : FILE[ASCII] ('SEQ, 'INPUT);
VAR out_file : FILE [ASCII] ('SEQ, 'OUT);

OPEN (in_file, "ABC", 'OLD);
OPEN (out_file, "XYZ", 'NEW);

WHILE NOT EOF(in_file) REPEAT
   VAR line : STRING[ASCII] (SIZELN(in_file));
   READLN (in_file, line);
   WRITELN (out_file, line);
END REPEAT;

CLOSE (in_file, 'DELETE);
CLOSE (out_file, 'SAVE);
```

3) Reverse lines from SYS_IN and output onto SYS_OUT.

```
OPEN (SYS_IN, "ABC", 'OLD);
OPEN (SYS_OUT, "XYZ", 'OLD);

WHILE NOT  EOF REPEAT
   VAR line : STRING[ASCII] (SIZELN);
   READLN (line);
   FOR i : INT(1 .. line.LEN DIV 2) REPEAT
      CONST j := line.LEN + 1 - i;
      CONST t := line (i);
      line (i) := line (j);
      line (j) := t;
   END REPEAT;
   WRITELN (line);
END REPEAT;

CLOSE (SYS_IN, 'SAVE);
CLOSE (SYS_OUT, 'SAVE);
```

Conversions Upon Input

Conversions are defined from strings to booleans, integers, floating point numbers, and enumeration values. Any leading or trailing blanks in the input strings are ignored. The form of the remaining characters must be a legal *literal* of the *type* to which the form will be converted, with an optional plus or minus sign preceding an integer or floating point literal. If the minus sign is present,

the *literal* will be converted to a negative value.

If an integer or floating point subtype is specified as the target of the conversion, the *string* literal must be within the target range or an exception is raised. If the *string literal* has more precision than the target floating point subtype, truncation is performed.

Conversions Upon Output

Conversions are defined from booleans, integers, floating point numbers, and enumeration values to strings. Each value is converted to the literal form and, if the value is a negative integer or floating point number, a minus sign is inserted before the integer or floating point literal. The format of a *string literal* of a floating point number is an optional minus sign followed by

```
n.nnnnE+mmm
```

or

```
n.nnnnE-mmm
```

where the number of n's is the precision of the floating point number.

If the target is a string subtype (that is, if a length is specified) and the length is longer than the *string literal*, the string is padded on the left with blanks. If the length is shorter than the *string* literal, an exception is raised.

EXAMPLES

1) Reads in a line which contains only an integer.

```
VAR line : STRING[ASCII] (SIZELN);
READLN (line);
s := CNVT [INT] (line);
```

2) Writes out a line which contains only an integer.

```
WRITELN (CNVT [STRING [ASCII]] (i));
```

3) Reads in a line which contains two integers, one in the first five columns and the next in the second five columns.

```
VAR i,j : INT(1..50);
ASSERT SIZELN >= 10;
VAR s : STRING[ASCII] (SIZELN);
READLN (s);
i := CNVT [INT] (s(1..5));
j := CNVT [INT] (s(6..10));
```

Output Formatting

To simplify the conversion of floating point numbers to ASCII strings for output, two format functions are provided in addition to the standard conversion function.

A floating point number can be output without an exponent by invoking the FORMAT function and

passing the floating point value, the number of characters to be output to the left of the decimal point, and the number of characters to be output to the right of the decimal point. Zeros on the left are not suppressed. If the value is negative, a minus sign is output to the left of the string. The total number of characters in

```
FORMAT (float_number, n, m)
```

is n + m + 2 (for the sign and decimal point). If the first significant digit would be lost, the X_FORMAT exception is raised.

A floating point number can be output with an exponent by invoking the FORMAT function with one additional parameter, the number of characters to be output to the right of E. The sign of the exponent will always be printed. The first significant digit will appear as the leftmost character. If the value is negative, a minus sign is printed to the left of the string. The total number of characters in

```
FORMAT (float_number, n, m, o)
```

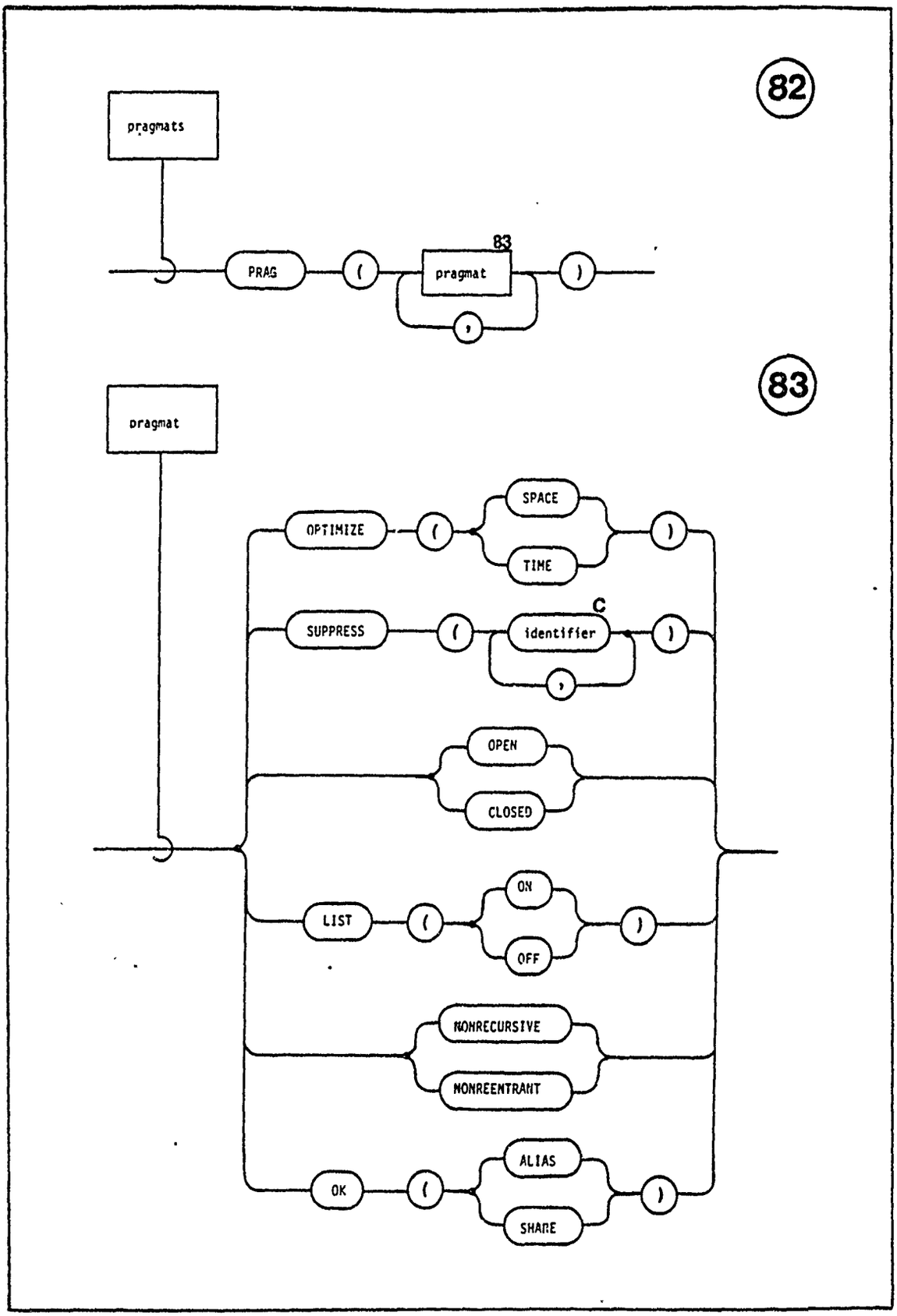
is n + m + o + 4 (for the sign, decimal point, E, and the exponent sign). If the exponent will not fit in o digits, the X_FORMAT exception is raised.

EXAMPLES

```
CNVT [STRING[ASCII]] (372.65)
CNVT [STRING[ASCII]] (-50.0)
CNVT [STRING[ASCII]] (10050.0)
CNVT [STRING[ASCII]] (572.0)
CNVT [STRING[ASCII]] (.0000000001)
          3.726E+2
          -5.00E+1
        1.00500E+4
          5.720E+2
  1.000000000E-10


FORMAT (372.65, 3, 3)
FORMAT (-50.0, 3, 3)
FORMAT (10050.0, 3, 3)
FORMAT (572.0, 3, 3)
FORMAT (.0000000001, 3,.3)
          372.650
         -050.000
         X_FORMAT
          572.000
          000.000


FORMAT (372.65, 3, 3, 3)
FORMAT (-50.0, 3, 3, 3)
FORMAT (10050.0, 3, 3, 3)
FORMAT (572.0, 3, 3, 3)
FORMAT (.0000000001, 3, 3, 3)
      372.650E+000
     -500.000E-001
      100.500E+002
```

```
572.000E+000
100.000E-012
```

B. PRAGMATS


82
pragmats
PRAG
(
83
pragmat
,
)
83
pragmat
OPTIMIZE
(
SPACE
TIME
)
SUPPRESS
(
C
identifier
,
)
OPEN
CLOSED
LIST
(
ON
OFF
)
NONRECURSIVE
NONREENTRANT
OK
(
ALIAS
SHARE
)

Pragmats supply information, that does not affect program semantics, to the translator. Some *pragmats* are language-defined; all translators are expected to recognize these pragmats (although the translator is not required to take any action). A translator can also permit additional pragmats to be specified.

RULES

A list of *pragmats* can appear between any two *tokens* in a program. Certain of the *pragmats* are further restricted in where they can appear. The restrictions for each *pragmat* is given below.

OPTIMIZE

This *pragmat* is used to inform the translator whether or not to optimize.

If the OPTIMIZE pragmat appears on a scope, it is applied to this scope, and to all scopes nested within this scope that do not, themselves, specify an OPTIMIZE pragmat.

If the OPTIMIZE pragmat appears on a *variable* or *constant declaration*, it controls the representation for the data item.

If the OPTIMIZE pragmat appears on a type declaration, it applies to the representation of all data items having that type.

SUPPRESS

Each of the *identifiers* must be exceptions. This *pragmat* indicates that no code need be generated to check for any of the listed exceptions during elaboration of the scope on which it is specified. Code will still be generated, however, for the guarded bodies of *guard statements* which explicitly handle the listed exceptions. If the exception actually occurs, the effect is undefined.

OPEN and CLOSED

These pragmats are specified on a deferred declaration or an an invocation of a deferred declaration. On a deferred declaration, they refer to all invocations. On an invocation, they refer to only that specific invocation. OPEN requests the translator to attempt to compile the invocation open (inline). CLOSED requests the translator to attempt to compile the invocation closed (out of line).

LIST

This pragmat can be specified between any two tokens. LIST(OFF) specifies that the source listing is not to be printed until the next LIST(ON) pragmat appears.

NONRECURSIVE and NONREENTRANT

These *pragmats* can be specified on a *deferred declaration*. They are used to inform the translator that the *deferred declaration* will not be invoked recursively (i.e., during its own elaboration) or reentrantly (simultaneously by two or more activations). If the translator or linker discovers that these *pragmats* were wrong, an error will be issued.

OK

This pragmat is used to turn off warning messages issued by the translator when it discovers that

dangerous aliasing or dangerous sharing will, or might, occur. The OK(ALIAS) pragmat can appear either on deferred declarations or an invocations. The OK(SHARE) pragmat can appear on variable declarations, constant declarations, and formal parameter definitions.

Placement of Pragmats

Pragmats can appear:

a) On a scope. The *pragmat* must appear immediately before the first *token* of the scope. For example,

```
PRAG (SUPPRESS (X_INIT)) BEGIN
   PRAG (OPTIMIZE (SPACE))
   ...
   PRAG (OPTIMIZE (TIME)) PROC p;
      ...
   END PROC p;
END;
```

b) On a *variable declaration*, *constant declaration*, formal parameter definition, or actual parameter. The *pragmat* must appear between the last *token* and the terminating ";" or "," or ")". For example,

```
VAR x : INT(0..10) OPTIMIZE (TIME);
PROC q (VAR y : INT(0..10)
   PRAG (OK (SHARE))  IMPORTS x;
   ...
END PROC q;
...
q (x PRAG(OK(ALIAS))  );
```

c) On a *deferred declaration*. For *compound declarations*, the *pragmat* must appear immediately before the ";" terminating the header. For *type* and *abbreviation declarations*, the *pragmat* must appear immediately before the terminating ";". For example,

```
TYPE t : INT(0..10) PRAG (OPTIMIZE (TIME));
PROC p (v : BOOL) IMPORTS w PRAG (OPEN);
   ...
END PROC p;
FUNC q (x : t) => t PRAG (NONRECURSIVE, NONREENTRANT);
   ...
END FUNC q;
```

C. BUILT-IN TYPES

C.1 BOOL

Subtype Form

BOOL

Values: false,true

Literals: FALSE, TRUE

Operations

Purpose	Operation	Notes
logical negation	NOT(BOOL)=>BOOL	
logical and	AND(BOOL,BOOL)=>BOOL	1
logical or	OR(BOOL,BOOL)=>BOOL	1
exclusive or	XOR(BOOL,BOOL)=>BOOL	
equality	=(BOOL,BOOL)=>BOOL	
assignment	:=(BOOL,BOOL)	

1) These are conditional operations. If the result is known after elaborating the left operand, the right operand is not elaborated.

C.2 ENUM

Subtype Forms

1) ENUM[e11,e12,...,e1n]
2) ENUM[e11,e12,...,e1n](min..max)

where e1's are enumeration literals and min and max are expressions with type ENUM[e11,e12,...,e1n]. A single literal may appear only once in the extension. Form 1 is equivalent to

ENUM[e11,e12,...,e1n](e11..e1n)

Values

The values are the enumeration literals that appear in the type property list restricted to the range min..max. The values are ordered so that if i<j then e1i<e1j.

Literals: all enumeration literals.

Operations

Purpose	Operations	Notes
successor	SUCC(ENUM)=>ENUM	1
predecessor	PRED(ENUM)=>ENUM	1
position	POS(ENUM)=>INT	2
equality	=(ENUM,ENUM)=>BOOL	3
ordering	<(ENUM,ENUM)=>BOOL	3
assignment	:=(ENUM,ENUM)	3,4

1) The result subtype is the subtype of the actual parameter. X_RANGE is raised for the successor of the last value and the predecessor of the first value.
2) When the actual parameter has type ENUM[e11,e12,...e1n] and value e1i, then the result has subtype INT(1..n) and value i.
3) Both operands must have the same type.
4) X_RANGE is raised if the source value is not valid for the target subtype.

Attribute Inquiry

Attribute	Result Subtype	Result Value
ENUM[...](min..max).MIN	ENUM[...](min..max)	min
ENUM[...](min..max).MAX	ENUM[...](min..max)	max

C.3 INT

Subtype Form

```
INT(min..max)
```

where min and max are expressions having type INT. If the values of min or max exceed the implementation defined largest range, then X_MAXRANGE is raised.

Values

Integers (whole numbers) in the range min..max.

Literals: All integer literals.

Operations

Purpose	Operation	Notes
plus	+(INT)=>INT	1
minus	-(INT)=>INT	1
addition	+(INT,INT)=>INT	1
subtraction	-(INT,INT)=>INT	1
multiplication	*(INT,INT)=>INT	1
exponentiation	**(INT,INT)=>INT	1,2
modulo	MOD(INT,INT)=>INT	1,3,4
division	DIV(INT,INT)=>INT	1,3,4
successor	SUCC(INT)=>INT	5
predecessor	PRED(INT)=>INT	5
equality	=(INT,INT)=>BOOL	
ordering	<(INT,INT)=>BOOL	
absolute value	ABS(INT)=>INT	1
assignment	:=(INT,INT)	6
conversion	IFLOAT(INT,INT)=>FLOAT	7

1) Result subtype is INT(imin,imax), where imin and imax values are selected by the implementation so that any result value will be included in the range. Note, however, that this range will never exceed an implementation defined largest range even if result values outside this range are possible. If this largest range is exceeded, X_OVERFLOW is raised.

2) Raises X_NEG_EXP if the value of the second actual parameter value is less than zero. 1**0=1

3) Raises X_ZERO_DIVIDE if the the value of the second actual parameter is zero.

4) The following identities hold:

a) X=Y*(X DIV Y) + (X MOD Y)

b) Either X MOD Y = 0 or X MOD Y and Y have the same algebraic sign.

c) ABS(X MOD Y) < ABS(Y)

5) The result subtype is the subtype of the actual parameter. X_RANGE is raised for the successor of the last value and the predecessor of the first value.

6) Raises X_RANGE if the source value is not valid for the target subtype.

7) This function is used to convert an integer to a floating point value. The first actual parameter is the precision of the result and must be manifest. The second actual parameter is the integer to be converted. See note 1 under FLOAT concerning the range of the result.

Attribute Inquiry

Attribute	Result Subtype	Result Value
INT(min..max).MIN	INT(min..max)	min
INT(min..max).MAX	INT(min..max)	max

C.4 FLOAT

Subtype Form

```
FLOAT(prec,min..max)
```

where prec has type INT and must be manifest and min and max are expressions having type FLOAT. If the value of prec exceeds an implementation defined largest precision, or if the values of min and max exceed an implementation defined largest range, then X_MAXRANGE is raised.

Values

Approximate floating point numbers having precision prec in the range min..max. The precision is the minimum number of decimal digits to be represented. For values around zero there is a smallest non-zero absolute value that is implementation dependent.

Literals: All FLOAT literals.

Operations

Purpose	Operation	Notes
plus	+(FLOAT)=>FLOAT	1
minus	-(FLOAT)=>FLOAT	1
addition	+(FLOAT,FLOAT)=>FLOAT	1
subtraction	-(FLOAT,FLOAT)=>FLOAT	1
multiplication	*(FLOAT,FLOAT)=>FLOAT	1
division	/(FLOAT,FLOAT)=>FLOAT	1,2
exponentiation	**(FLOAT,INT)=>FLOAT	1,3
equality	=(FLOAT,FLOAT)=>BOOL	
ordering	<(FLOAT,FLOAT)=>BOOL	
absolute value	ABS(FLOAT)=>FLOAT	1
assignment	:=(FLOAT,FLOAT)	4
conversion	FLOOR(FLOAT)=>INT	5,6

1) Result subtype is FLOAT(prec,imin..imax) where prec is the maximum of the precisions of the actual parameter(s). Imin and imax values are selected by the implementation so that any result value will be included in the range. Note however that this range will never exceed an implementation defined largest range even if result values outside this range are possible. If the largest range is exceeded, X_OVERFLOW is raised.

2) Raises X_ZERO_DIVIDE if the value of the second actual parameter is zero.

3) a**0 = 1. Raises X_NEG_EXP if the value of the second parameter is less than zero.

4) Raises X_RANGE if source value is not valid for target subtype. If the target has less precision than the source, truncation is performed.

5) Rounds any fractional part down. FLOOR(X) <= X

6) See note 1 under INT operations.

Attribute Inquiry

Attribute	Result Subtype	Result Value
FLOAT(prec,min..max).PREC	INT(prec..prec)	prec
FLOAT(prec,min..max).MIN	FLOAT(prec,min..max)	min
FLOAT(prec,min..max).MAX	FLOAT(prec,min..max)	max

Machine Dependent Inquiry Functions

Purpose	Operation
actual precision	ACTP(FLOAT)=>INT
radix	RADIX(FLOAT)=>INT
minimum exponent	EMIN(FLOAT)=>INT
maximum exponent	EMAX(FLOAT)=>INT

C.5 RECORD

Subtype Form

```
RECORD[id1:S1,id2:S2,...,idn:Sn]
```

If several adjacent components have the same subtype, then

```
id1,idi+1,...,idj:S
```

can be used as a shorthand for

```
id1:S,idi+1:S,...,idj:S
```

Values

An ordered set of name-value pairs, one for each identifier, where the name of the i'th value is `idi` and the subtype of the i'th value is `Si`.

Components

A record variable (or constant) is made up of one or more component variables (or constants). Each component is named by a distinct identifier. The components may have different subtypes.

Constructor: See Section 5.6.

Operations

Purpose	Routine	Notes
equality	=(RECORD,RECORD)=>BOOL	1,2
assignment	:=(RECORD,RECORD)	1,3

1) Actual parameters must have the same type.

2) Each of the component types must have = defined. Each of the components are compared using = for the component type.

3) Each of the component types must have := defined. Each of the components are assigned in a undefined order using the := for the component type.

Record Component Selection

The result of

```
R.C
```

where R has a RECORD type is component C of record R. The result is a variable if R is a variable.

C.6 UNION

Subtype Forms

1) UNION[id1:S1,id2:S2,...,idn:Sn]
2) UNION[id1:S1,id2;S2,...,idn:Sn](exp)

where exp has type ENUM['id1,'id2,...,'idn]. None of the id1 may be TAG. If several components have the same subtype then

id1,id1+1,...,idj:S

can be used as a shorthand for

id1:S,id1+1:S,...,idj:S

Values

The discriminated union of the values of each of the component subtypes. Equivalently, a tag-value pair in which the subtype of the value is the one named by the tag. Variables having subtypes of form 2 may only have values whose tag is the value of exp.

Components

At any time a union variable (or constant) has exactly one named component variable (or constant). Consider

UNION[id1:S1,id2:S2,...,idn:Sn]

The possible component names are id1, id2, ..., idn. The current component name is called the tag. When the tag is id1, the component variable (or constant) has subtype S1.

Constructor: See Section 5.6.

Operations

Purpose	Operation	Notes
equality	=(UNION,UNION)=>BOOL	1,2
assignment	:=(UNION,UNION)	1,3

1) Actual parameters must have the same type.
2) Each of the component types must have = defined. Result is true if types are equal and the current components are equal.
3) Each of the component types must have := defined. The component is assigned using := for its type. X_RANGE is raised if the source is not valid for the target.

Union Component Selection

The result of

```
U.C
```

where U has a UNION type is component C of union U. If the selected union component is not currently present, then the X_TAG exception is raised. The result is a variable if U is a variable. The result of

```
U.TAG
```

is a value with subtype ENUM['id1,'id2,...,'idn] where the id's are the component names of the UNION type of U in order. The result value is the name of the component currently held by the union.

C.7 ARRAY

Subtype Form

```
ARRAY index1,index2,...,indexn OF comp
```

This is the subtype for an n-dimensional array whose index subtypes are index1, index2,...,indexn. The component subtype is comp. Each index subtype must be either an INT or ENUM subtype.

Values

Arrays of values of the component subtype, where there is one value in the array for each combination of values of the index subtypes.

Components

An array variable (or constant) has zero or more components, variables (or constants), all with the same component subtype. A component is selected by supplying a value for each index. Note that if any index subtype has no values, the array will have no components.

Constructor: See Section 5.6.

Operations

Purpose	Operation	Notes
equality	=(ARRAY,ARRAY)=>BOOL	1,2
concatenate	&(ARRAY,ARRAY)=>ARRAY	1,3,4
assignment	:=(ARRAY,ARRAY)	1,3,5

1) Actual parameters must have the same type.

2) The component type must have = defined. The result is true if the index subtypes are equal and corresponding components are equal.

3) The component type must have := defined. Components are assigned using := for the component type. If the source and target overlap assignment is done in such a way that a target component will not be modified before its use as a source component.

4) Only 1-dimensional arrays whose index type is INT may be used as actual parameters. If the actual parameters have respectively m and n components, then the index subtype of the result is INT(1..m+n).

5) X_ARRAY is raised if the number of values in the corresponding index subtypes of the source and the target are not equal.

Index Subtype

INDEXOF is described in Section 4.5.1.

Simple Array Element Selection

The result of

```
A(exp1,exp2,...,expn)
```

where A is a n-dimensional array and the type of each expi is the i'th index type of the array is the component of A at position specified by the value of the exp's. If the value of any exp is not valid for the corresponding index subtype, the X_SUBSCRIPT exception is raised. The result is a variable if A is a variable.

Subarray Selection

The result of

```
A(min..max)
```

where A is a 1-dimensional array and the type of min and max is the index type of that array, is a subarray of A. If the range min .. max is non-empty and contains an index outside the array bounds, then X_SUBSCRIPT is raised. The result has the same component type as A. The index subtype of the result is the index subtype of A restricted to the range min..max. The result is a variable if A is a variable.

C.8 STRING

Subtype Form

STRING[S](len)

where S is an ENUM subtype and len is an expression with type INT.

Values

Character strings with length len. Each character will be a value of the component subtype S. If len is less than or equal to zero then the string is empty.

Literals: All string literals.

Operations

Purpose	Operation	Notes
concatenate	&(ENUM,ENUM)=>STRING	1,2
concatenate	&(ENUM,STRING)=>STRING	1,2
concatenate	&(STRING,ENUM)=>STRING	1,2
concatenate	&(STRING,STRING)=>STRING	1,2
equality	=(STRING,STRING)=>BOOL	1,3
ordering	<(STRING,STRING)=>BOOL	1,4
assignment	:=(STRING,STRING)	1,5

1) Actual parameters must have the same type or have an ENUM type which is the same as the component type of the string.

2) The length of the result is the sum of the lengths of the actual parameters.

3) If the actual parameters have a different length then the result is false.

4) Ordering is based on ordering of the type of the characters. Characters are compared left to right with characters on the left being most significant. If the first actual parameter is a prefix of a longer second parameter the result is true. If the second actual parameter is a prefix of a longer first actual parameter then the result is false.

5) Both actual parameters must have the same length.

Attribute Inquiry

Attribute	Result Subtype	Result Value
STRING[S](len).LEN	INT(len..len)	len

Component Selection

The result of

```
V(I)
```

where V is a STRING and I is an integer whose value is between 1 and the length of V, is the I'th character of V. If I is out of bounds, then X_SUBSCRIPT is raised. If V has subtype STRING[S](n) then the result has subtype S. The result is a variable if V is a variable. The result of

```
V(I..J)
```

where V is a string and I and J are integers whose value is between 1 and the length of V is a substring of V starting at position I and ending at position J. If I or J is out of bounds, then X_SUBSCRIPT is raised. If V has subtype STRING[S](n) then the result has subtype STRING[S](J-I+1).

C.9 SET

Subtype Form

```
SET[S]
```

where S is an INT or ENUM subtype.

Values: Subsets of the set of all values of subtype S.

Operations

Purpose	Operation	Notes
equality	=(SET,SET)=>BOOL	1
subset	<(SET,SET)=>BOOL	1
membership	IN(t,SET[t])=>BOOL	
complement	NOT(SET)=>SET	2
intersection	AND(SET,SET)=>SET	1,2
union	OR(SET,SET)=>SET	1,2
symmetric difference	XOR(SET,SET)=>SET	1,2
assignment	:=(SET,SET)	1

1) Both actual parameters must have the same subtype.

2) The result subtype is the same as the subtype of the actual parameter(s).

C.10 ACT

Subtype Form

ACT

Values and Components

An ACT variable is used to control the elaboration of some activation and contains all information needed to record all aspects of that elaboration. There are three components of the value of special interest to the user.

a) A state - which has three subcomponents as follows

   1) active - a boolean. All ACT variables are automatically initialized to be inactive (i.e. active=false).

   2) waiting - a boolean. Initially false.

   3) suspended - a boolean. Initially false.

b) A priority - A component whose subtype is INT(0..255).

c) A clock - This measures the total real time that the activation has been running.

Predefined Constant: NIL_ACT. Representing a unique, always inactive value.

Operations

Purpose	Operation	Notes
current activation	ME=>ACT	1
active	ACTIVE(ACT)=>BOOL	2
waiting	WAITING(ACT)=>BOOL	3
suspended	SUSPENDED(ACT)=>BOOL	4
priority	PRIORITY(ACT)=>INT	5
set priority	SET_PRIORITY(ACT,INT)	6
suspend	SUSPEND(ACT)	7
unsuspend	UNSUSPEND(ACT)	8
time	TIME(ACT)=>INT	9
delay	DELAY_UNTIL(ACT,INT)	10,11
delay	DELAY_UNTIL_INACTIVE(ACT)	11,12
wait	SYNC_WAIT ( )	13
reset	SYNC_RESET( )	14
signal	SYNC_SIGNAL(ACT)	15
create	TASK_START(ACT,ACT)	16
finish	TASK_END(ACT)	17
low level	LOW_SYNC_WAIT	18
	LOW_SYNC_RESET	18
	LOW_SYNC_SIGNAL(ACT)	18,19
	LOW_TASK_END(ACT)	20
terminate	EXTERMINATE(ACT)	21
terminate control	CRITICAL	22

	NONCRITICAL	23
assignment	:=(ACT,ACT)	24
equality	=(ACT,ACT)=>BOOL	25

1) Result is the activation variable associated with the invoking activation.

2) Result is true if the specified activation is active.

3) Result is true if the specified activation variable is active and waiting.

4) Result is true if the specified activation variable is suspended.

5) Result is the priority of the specified activation.

6) Sets the priority of the specified activation.

7) Sets suspended to true.

8) Sets suspended to false.

9) The result is the value of the activation clock of the specified activation in ticks.

10) Delays until the activation clock of the specified activation is greater than or equal to the time specified in ticks.

11) These operations can be used as waiting invocations in the wait statement.

12) Waits until the specified activation variable is inactive.

13) Equivalent to LOW_SYNC_WAIT if the current activation was created with an ACT-type activation variable, or to SYNC_WAIT(av) if the current activation was created with activation variable av of user-defined type.

14) Equivalent to LOW_SYNC_RESET or SYNC_RESET(av) depending on how activation a was created.

15) Equivalent to LOW_SYNC_SIGNAL(a) or SYNC_SIGNAL(av) depending on how activation a was created.

16) Used to start an activation. Called as part of the elaboration of a task invocation statement. Assigns second parameter to first.

17) Equivalent to LOW_TASK_END(a) or TASK_END(av) as in (15).

18) LOW_SYNC_WAIT sets waiting to true unless LOW_SYNC_SIGNAL has been called for this activation since LOW_SYNC_RESET was invoked.

19) Sets waiting to false.

20) Invoked when an activation is complete. Can also be used to terminate some other task.

21) For the given activation, sets waiting to false, and raises the X_TERMINATE exception.

22) Make activation critical.

23) Make activation noncritical.

24) This is a sharing assignment.

25) The result is true if both actual parameters are the same activation variable.

C.11 MAILBOX

Subtype Form

MAILBOX[S] (len)

where S is the subtype of messages and len is an integer which is greater than or equal to zero. Assignment must be defined for the type of S.

Values and Components

A mailbox can hold up to len messages having subtype S. All mailbox variables are automatically initialized to hold no messages. Messages are queued first-in first-out.

Operations

Purpose	Operation	Notes
send	SEND(MAILBOX[t],t)	1,2
receive	RECEIVE(MAILBOX[t],t)	2,3
test empty	EMPTY_SLOTS(MAILBOX)=>INT	4
test full	FULL_SLOTS(MAILBOX)=>INT	5
conditional send	COND_SEND (MAILBOX[t],t,BOOL)	6
conditonal receive	COND_RECEIVE (MAILBOX[t],t,BOOL)	7
low level ops	SEND_ST	8
	SEND_REQUEST(MAILBOX[t],t)	9
	SEND_TEST(MAILBOX[t],t)	9
	SEND_COMPLETE (MAILBOX[t],t)	9
	SEND_REVOKE(MAILBOX[t],t)	9
	RECEIVE_ST	10
	RECEIVE_REQUEST (MAILBOX[t],t)	11
	RECEIVE_TEST(MAILBOX[t],t)	11
	RECEIVE_COMPLETE (MAILBOX[t],t)	11
	RECEIVE_REVOKE (MAILBOX[t],t)	11
assignment	:=(MAILBOX,MAILBOX)	12
equality	=(MAILBOX,MAILBOX)	13

1) Sends a message to a mailbox. Sender waits if the mailbox is full until the mailbox is no longer full. If there are several waiting senders they are queued first-in first-out.

2) These operations can be used as waiting invocations in the wait statement. If SEND is called on a zero length mailbox, in a multi-way wait statement, then X_EMPTY_MAILBOX is called.

3) Receives a message from a mailbox. Receiver waits if the mailbox is empty until the mailbox is no longer empty. If there are several waiting receivers, they are queued first-in first-out.

4) Count of empty message slots plus waiting receivers.

5) Count of full message slots plus waiting senders.

6) Try to send; indicate success as a returned boolean (true=success. Never wait.

7) Try to receive; indicate success as a returned boolean. Never wait.

8) Subtype for low-level implementation of sends. See Section 14.3.1.

9) Operations for low-level implementation of sends.

10) Subtype for low-level implementation of receives. See Section 14.3.1.

11) Operations for low-level implementation of receives.

12) This is a sharing assignment.

13) The result is true if both actual parameters are the same mailbox.

Attribute Inquiry

Attribute	Result Subtype	Result Value
MAILBOX[S](len).LEN	INT(len..len)	len

C.12 DATA_LOCK

Subtype Form

DATA_LOCK

Values and Components

A data lock is either locked or unlocked. All data lock variables are automatically initialized to be unlocked. If a data lock is locked it has an owner, which is the activation that locked it.

Operations

Purpose	Operation	Notes
test locked	EXCESS_LOCKS(DATA_LOCK)=>INT	
owner	OWNER(DATA_LOCK)=>ACT	
lock	LOCK(DATA_LOCK)	2,3
unlock	UNLOCK(DATA_LOCK)	1,3
conditional lock	COND_LOCK(DATA_LOCK,BOOL)	

1) If not locked, raises X_LOCK.

2) If data lock is unlocked sets the data lock to be locked and the owner to be the invoking activation; otherwise the invoker waits until the data lock becomes unlocked. If there are several activations suspended these are processed on a first-come first-served basis.

3) These are low-level operations used to implement user-defined locking operations and in the implementation of the REGION statement.

C.13 LATCH

The LATCH type is the basic low level synchronization type.

Subtype Form: LATCH

Values: latched, unlatched

Operations

Purpose	Routine	Notes
lock	LOCK(LATCH)	1
unlock	UNLOCK(LATCH)	2
test and set	COND_LOCK(LATCH,BOOL)	3

1) Waits until the actual parameter has value unlatched; then changes the value to latched and continues. If several tasks are waiting, only one will continue.

2) Changes the value of the actual parameter to unlatched.

3) The second parameter is a status result. If latch has the value unlatched, then performs a LOCK and sets result to true, otherwise sets result to false.

C.14 FILE

Subtype Form

```
FILE [S] (access, use)
```

Each component will have subtype S. Assignment must be defined for the type of S. The type of access must be ENUM['SEQ, 'RANDOM]. The type of use must be ENUM['INPUT, 'OUTPUT, 'UPDATE].

Values and Components

A file variable (also called a file) is associated with an ordered sequence of components each having subtype S. The components will reside on some particular physical file of the target system. A file variable is associated with a particular physical file by invocation of the OPEN procedure (the file is then said to be open). The association is broken by invocation of the CLOSE procedure (the file is then said to be closed). Initially all files are closed.

The access of a file determines how the components are accessed.

'SEQ	sequential access
'RANDOM	random access

The use of a file determines how the file will be used.

'INPUT	components are read but not written.
'OUTPUT	components are written but not read.
'UPDATE	components are both written and read.

When a file is open it has a size which is the number of components in the file and a position which is an integer whose value is greater than or equal to 1 and which is less than or equal to the size of the file plus 1. Position 1 is before all components in the file. Position 2 is before the second component, etc.

Operations

Purpose	Operation	Notes
open	OPEN(FILE,STRING, ENUM['OLD), 'NEW]	1,2
close	CLOSEFILE, ENUM['SAVE, 'DELETE])	3,4
size	SIZE(FILE)=>INT	3,5,6,15
position	POSITION(FILE)=>INT	3,7,15
set position	SET_POSITION(FILE,INT)	3,5,8
end of file	EOF(FILE)=>BOOL	3,9,15
read	READ(FILE,t)	3,10,11,12
write	WRITE(FILE,t)	3,11,13,14
read line	READLN(FILE,STRING)	15,16,17
write line	WRITELN(FILE,STRING)	16,18
line size	SIZELN(FILE)=>INT	15,16,19
read line	READLN(STRING)	15,20
write line	WRITELN(STRING)	21
size line	SIZELN()=>INT	15,23
eof	EOF()	15,22
other	see note	24

1) Raises X_FILE if file is already open.

2) The string specifies the physical file with which the file variable is to be associated. The interpretation of the string is implementation-dependent. If there is no such device or physical file then the X_FILENAME exception is raised. The enumeration value specifies whether an existing file is to be used ('OLD) or a new file is to be created ('NEW). If for 'OLD there is no existing physical file then the X_NOFILE exception is raised. The position of the file is set to 1. A 'NEW file initially has size 0

3) Raises X_FILE if file is not open.

4) The enumeration value 'SAVE indicates that the physical file is to be saved for use by latter programs. The value 'DELETE indicates that the physical file is not be saved. The current position is taken as the new end of file.

5) These procedures and functions may not be available for all physical files. If not, the X_FILE exception is raised. They will, however, always be available for 'RANDOM access files.

6) Returns the size of the file.

7) Returns the current position.

8) This procedure repositions the file to the specified new position. If the integer parameter is less than 1 or greater than the size of the file +1 then the X_FILEPOS exception is raised.

9) Returns true if the position of the file is equal to the size of the file +1.

10) Legal only if use is 'INPUT or 'UPDATE. Otherwise, X_FILE is raised.

11) The second parameter must have the same type as the components of the file.

12) The component following the current position is assigned to the second parameter. The current position is incremented by one. If EOF is true, then the X_EOF exception is raised.

13) If EOF is true prior to the invocation, then the number of components in the file is increased by one. The component following the current position is assigned the value of the second parameter. The current position is incremented by one.

14) Legal only if use is 'OUTPUT or 'UPDATE. Otherwise X_FILE is raised.

15) These are abnormal functions (see).

16) The file type must be FILE[ENUM[...]]. The ENUM type must include 'CR and 'LF.

17) READLN reads components until a 'CR followed by a 'LF is found. All components prior to the 'CR 'LF are returned as characters in the string. Raises X_LN if no 'CR followed by 'LF is found or if the string is of the wrong length.

18) Writes each character in the string as a component in the file and then writes a 'CR followed by a 'LF.

19) Returns the number of characters between the current position and the position immediately before the next 'CR followed by 'LF.

20) READLN(s) is equivalent to READLN(SYS_IN,s).

21) WRITELN(s) is equivalent to WRITELN(SYS_OUT,s).

22) EOF is equivalent to EOF(SYS_IN).

23) SIZELN is equivalent to SIZELN(SYS_IN).

24) It is anticipated that a variety of implementation-dependent and/or device-dependent routines will be available for setting and interrogating file characteristics such as protection, physical blocking, and file space.

Attribute Inquiry

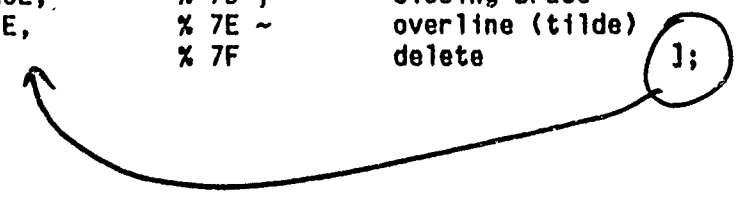
Attribute	Result Subtype	Result Value
FILE[S](access, use).ACCESS	ENUM['SEQ, 'RANDOM]	access
FILE[S](access, use).USE	ENUM['INPUT, 'OUTPUT, 'UPDATE]	use

C.15 ASCII

The 128 character ASCII character set is a language defined abbreviation. Its definition appears below.

```
ABBREV ASCII : ENUM[
'NUL,          % 00          null
'SOH,          % 01          start of heading
'STX,          % 02          start of text
'ETX,          % 03          end of text
'EOT,          % 04          end of transmission
'ENQ,          % 05          enquiry
'ACK,          % 06          acknowledge
'BEL,          % 07          bell
'BS,           % 08          backspace
'HT,           % 09          horizontal tabulation
'LF,           % 0A          line feed
'VT,           % 0B          vertical tabulation
'FF,           % 0C          form feed
'CR,           % 0D          carriage return
'SO,           % 0E          shift out
'SI,           % 0F          shift in
'DLE,          % 10          data link escape
'DC1,          % 11          device control 1
'DC2,          % 12          device control 2
'DC3,          % 13          device control 3
'DC4,          % 14          device control 4
'NAK,          % 15          negative acknowledge
'SYN,          % 16          synchronous idle
'ETB,          % 17          end of transmission block
'CAN,          % 18          cancel
'EM,           % 19          end of medium
'SUB,          % 'A          substitute
'ESC,          % 1B          escape
'FS,           % 1C          file separator
'GS,           % 1D          group separator
'RS,           % 1E          record separator
'US,           % 1F          unit separator
'SP            % 20 blank    space (normally non-printing)
'EXCLAIM,      % 21 !         exclamation point
'QUOTE,        % 22 "         quotation marks
'NUMBER,       % 23 #         number sign
'DOLLAR,       % 24 $         dollar sign
'PERCENT,      % 25 %         percent
'AMPERSAND,    % 26 &         ampersand
'APOS,         % 27 '         apostrophe
'LPAREN,       % 28 (         opening parenthesis
'RPAREN,       % 29 )         closing parenthesis
'STAR,         % 2A *         asterisk
'PLUS,         % 2B +         plus
'COMMA,        % 2C ,         comma
'MINUS,        % 2D -         hyphen (minus)
'PERIOD,       % 2E .         period
'SLASH,        % 2F /         slant
'N_0,          % 30 0
'N_1,          % 31 1
'N_2,          % 32 2
'N_3,          % 33 3
```

```
'N_4,           % 34 4
'N_5,           % 35 5
'N_6,           % 36 6
'N_7,           % 37 7
'N_8,           % 38 8
'N_9,           % 39 9
'COLON,         % 3A :       colon
'SEMI,          % 3B ;       semicolon
'LESS,          % 3C <       less than
'EQUAL,         % 3D =       equals
'GREATER,       % 3E >       greater than
'QUESTION,      % 3F ?       question mark
'AT,            % 40 @       commercial at
'A,             % 41 A
'B,             % 42 B
...
'Z,             % 5A Z
'LBRACKET,      % 5B [       opening bracket
'BSLASH,        % 5C \       reverse slant
'RBRACKET,      % 5D ]       closing bracket
'CIRCUMFLEX,    % 5E ^       circumflex
'UNDERSCORE,    % 5F _       underline
'GRAVE,         % 60 `       grave accent
'L_A,           % 61 a
'L_B,           % 62 b
...
'L_Z,           % 7A z
'LBRACE,        % 7B {       opening brace
'BAR,           % 7C |       vertical line
'RBRACE,        % 7D }       closing brace
'TILDE,         % 7E ~       overline (tilde)
'DEL            % 7F         delete          ];
```



D. EXCEPTIONS

Exception	Raised When
X_ASSERT	Assertion is false
X_CASE	No case value label or else clause matching case expression value
X_EMPTY_MAILBOX	Attempt to define zero length mailbox
X_EOF	Attempt to read past eof
X_FILE	File processing attempted on unopen file, file processing routine not available for file, routine not available for file, or attempt to wait on input file or read output file
X_FILENAME	Physical file name unknown
X_FILEPOS	Attempt to position outside of file
X_FREE	Attempt to free space for dynamic variable still referenced by indirect variable
X_INIT	Attempt to access uninitialized variable
X_LN	'CR not followed by 'LF
X_LOCK	Attempt to unlock data-lock which already is unlocked or attempt to find owner of unlocked data-lock
X_MAXRANGE	Value is outside of implementation-defined largest range
X_NEG_EXP	Attempt to raise a number to a negative power
X_NOFILE	Designation of nonexistent file as 'OLD
X_OVERFLOW	Result of operation is outside of implementation-selected range for INT or FLOAT
X_RANGE	Source value is not valid for the target (ENUM, INT, FLOAT)
X_SUBSCRIPT	Attempt to access beyond ARRAY or STRING bounds
X_SUBTYPE	Attempt to pass actual of different subtype to VAR or READONLY formal parameter
X_TAG	Selection of UNION component not currently present
X_UNHANDLED	Exception has not been handled when completing its scope
X_ZERO_DIVIDE	Attempt to divide INT or FLOAT by zero

E. GLOSSARY

abbreviation
: shorthand notation for type or subtype specification.

abnormal function
: a function upon which common subexpression elimination may not be performed.

access method
: sequential or random file access.

active activation variable
: an activation variable associated with a specific activation.

activation
: an elaboration of a task which can occur concurrent with other activations.

activation clock
: measures total real time that a particular activation has run.

activation variable
: means for 'naming' an activation.

actual parameter
: expression in an invocation, bound to formal parameter in declaration of the invoked deferred unit.

assignable types
: those types for which assignment (:=) is defined.

allocation statement
: creates a dynamic variable.

attributes
: subtype constraints.

available definitions
: for an open scope, all definitions known in the immediately enclosing scope. Fc a closed scope, all definitions, except variable, goto label, and matching identifier definitions, from the immediately enclosing scope, together with all imported definitions.

binding
: association of actual with formal parameters.

body
: possibly empty sequence of declarations, assertions, and statements.

closed scope
: scope which is similar to open scope, except that variable definitions only become available by being imported.

compound statement or declaration
: statement or declaration containing a body.

conditional translation
: selection, during translation, between alternative bodies or selection of a single union component to be present

conflicting definitions
: definitions of the same name among which a use of the name cannot discriminate.

constant
: data item, having a subtype, whose value cannot be modified.

constructor
: way of building a value of a record, union, array, or user-defined type.

dangerous sharing
: non-orderly use of a shared variable.

declaration
: one form of language construct used for defining a name.

deferred declaration
: a declaration which is elaborated when invoked, not when encountered.

deferred unit
: the entity defined by a deferred declaration.

definition
: any way of associating a name with a language construct.

delaying
: non-busy waiting of an activation upon a clock value.

data item
: defined variables and constants, dynamic variables, readonly data items and temporary data items.

dot selection
: qualification to obtain component of record or union. Also used to dereference pointers.

dynamic variable
: the variable pointed to be an indirect variable or constant.

elaboration
: the action required at translation time and runtime to cause statement execution, expression evaluation, and declaration processing.

exception
: an exceptional condition which can be raised and handled.

explicit overloading
: overloading which occurs by writing two definitions of the same name with different interfaces.

exported definitions
: definitions local to a capsule which may be made known outside the capsule.

exposing
: making exported names of a capsule known.

extermination
: raising X_TERMINATE in another activation.

external capsule
: a capsule which was separately translated.

file
: a variable of a FILE type which is associated with an ordered collection of components, all having the same type.

file use
: input, output, or update files.

flow diagrams
: syntactical description technique.

formal parameter
: placeholder within a deferred declaration.

generic constraint
: restricts the set of replacement elements which may be associated with a particular generic

parameter.

generic overloading
: overloading which occurs when a deferred declaration is replicated generically.

generic parameter
: placeholder within a generic declaration.

generated set
: all definitions created as a result of a generic declaration.

goto label
: used only as target of goto statement.

guarded body
: body of a guard statement in which an exception might be raised and, so, is protected.

handler
: part of a guard statement which specifies a body to be elaborated if a specific exception is raised during elaboration of the guarded body.

handling an exception
: terminating the guarded body in which the exception is raised and attempting to find a handler for the exception.

immediate declaration
: a declaration which is elaborated when encountered.

imported definitions
: variable definitions which become available to closed scopes through an imports list.

inactive activation variable
: an activation variable associated with no activation.

index
: counter for repeat statement.

indirect variable or constant
: typed pointer.

infix operator
: operator (**, *, /, MOD, DIV, &, +, -, =, /=, <, <=, >, >=, IN, AND, OR, XOR) which takes two operands.

initialization
: assigning a first value to a data item.

inquiry
: operations which interrogate the type, subtype, or attributes of data items.

interface
: information, contained in both definitions and uses of overloaded names, which is used to associate a use with a single definition.

known in a scope
: all definitions which can be associated with a use of a name within a scope. Includes all local definitions plus all non-conflicting available definitions.

lifetime
: the span of execution time that a construct exists.

literal
: token used to specify a value of a built-in or user-defined type.

local definitions
: all definitions either defined in a scope or exposed into it by a capsule invocation.

mailbox
: queue of messages used for inter-activation communication.

manifest expression
: expression which is elaborated during translation.

matching identifier
: used for documentation and as target of exit statement.

message passing
: transmittal of messages between activations by means of mailboxes.

mutual exclusion
: orderly access to shared variables accomplished via the region statement.

name
: an identifier or definable symbol associated with defined language constructs.

needed name
: a name which appears in a generic declaration and is bound at point of invocation rather than point of generic declaration.

new capsule
: a capsule whose invocation results in the elaboration of the capsule (and therefore creation of separate copies of local data).

new type
: type, defined by type declaration, distinct from all other types.

normal function
: a function upon which common subexpression elimination may be done.

old capsule
: a capsule whose invocation does not result in elaboration of the capsule.

open scope
: scopes following normal block-structured rules.

overloading
: association of a single name with multiple deferred units of the same kind.

own data
: data local to a capsule.

pattern declaration
: the deferred declaration which is generically replicated.

physical file
: a file on a storage medium, a physical device, or a file on a storage medium representing a physical device.

precedence
: for operators, the relative strength of operand association.

prefix operator
: operator (+, -, NOT) which takes only a single operand.

primary
: expression operand.

priority
: absolute (rather than relative) importance of an activation.

raising an exception
: causing the occurrence of an exceptional condition which may be handled by a guard statement.

random file
: a file which permits direct access to any component, regardless of its position with respect to other components.

range
: a contiguous sequence of values of some type.

readonly data item
: variable whose use is restricted to access only in some contexts but which may still be modified in other contexts.

real-time clock
: measures elapsed real-time since program began to run.

renaming
: changing of a name through a capsule invocation.

replacement element
: the item which appears in an invocation and is substituted for the corresponding generic parameter in the definition.

result type or subtype
: type or subtype of function result.

routine
: procedure, function, or operator.

scheduler
: determines which activation will next run. The built-in priority scheduler may be replaced with a user-defined scheduler.

sequential file
: file which is processed by accessing each component in succession.

shared variable
: a variable which is accessible to multiple activations.

side-effect
: modification of data by a function when the lifetime of the data is longer than the lifetime of the function invocation.

signature
: number, order, and types of parameters. Part of interface.

slice
: contiguous components contained within one dimensional array.

standard file
: a file whose components do not have an enumeration type.

state of a file
: initial state (old or new) and final disposition (save or delete).

subscripting
: qualification to obtain component of array or array slice.

subtype
: translation-time and runtime properties of a data item.

subtype constraints
: round-bracketed information of subtype which can be left unresolved until runtime.

temporary data item
: data item whose lifetime is tied to its use rather than to a scope. Includes the result of literals, constructors, functions, and attribute inquiry.

text file
: a file whose components have an enumeration type.

translation time property list
: part of the interface of deferred declarations used for resolving name conflicts.

type
: translation time properties of a data item.

type and subtype resolution
: determination of type and subtype of expression.

type checking
: verifying, at translation time, that two types are the same.

underlying subtype
: subtype specification used to define a new type.

underlying type
: type of underlying subtype.

underlying variable or constant
: representation (obtained by .ALL) of variable or constant of a new type. An underlying variable may also be the dynamic variable pointed to by an indirect variable or constant.

value labels
: labels of case statement.

variable
: data item, having a subtype, whose value can be modified. Includes defined and dynamic variables.

visible definitions
: definitions which are exported from a capsule and exposed by a capsule invocation.

waiting
: non-busy suspension of an activation until some event occurs.

F. DIAGRAM CROSS REFERENCE

DIAGRAM CROSS-REFERENCE

(alphabetically)

LEXICAL SYNTAX DIAGRAMS

	Diagram Identifier	Section Number
boolean literal	I	2.3.9
comment	K	2.4.1
enum literal	G	2.3.7
float literal	F	2.3.6
identifier	C	2.3.4
indirect literal	J	2.3.10
int literal	E	2.3.6
literal	D	2.3.5
pragmat token separator	L	2.4.2
string literal	H	2.3.8
token	A	2.2
token separator	B	2.2

DIAGRAM CROSS-REFERENCE

(by diagram identifier)

LEXICAL SYNTAX DIAGRAMS

	Diagram Identifier	Section Number
token	A	2.2
token separator	B	2.2
identifier	C	2.3.4
literal	D	2.3.5
int literal	E	2.3.6
float literal	F	2.3.6
enum literal	G	2.3.7
string literal	H	2.3.8
boolean literal	I	2.3.9
indirect literal	J	2.3.10
comment	K	2.4.1
pragmat token separator	L	2.4.2

DIAGRAM CROSS-REFERENCE

(alphabetically)

GRAMMAR SYNTAX DIAGRAMS

DIAGRAM CROSS-REFERENCE

(by diagram identifier)

GRAMMAR SYNTAX DIAGRAMS

	Diagram Identifier	Section Number
translation unit	1	3.1
body	2	3.2
body element	3	3.2
declaration	4	3.3
immediate declaration	5	3.3
deferred declaration	6	3.3
compound declaration	7	3.3
assertion	8	3.4
imports	9	3.7
type	10	4.1.5
type comparison	11	4.1.5
subtype	12	4.1.6
subtype comparison	13	4.1.6
range	14	4.1.7
variable declaration	15	4.2.1
constant declaration	16	4.2.1
abbreviation declaration	17	4.4.1
abbreviation invocation	18	4.4.1
type declaration	19	4.4.2
direct type declaration	20	4.4.2
user-defined type	21	4.4.2
user-defined subtype	22	4.4.2
indirect type declaration	23	4.4.3
allocation statement	24	4.4.3
attribute inquiry	25	4.5.3
expression	26	5.2
primary	27	5.3
variable	28	5.3
constant	29	5.3
referencing form	30	5.3
constructor	31	5.6
record or union constructor	32	5.6
array constructor	33	5.6
set constructor	34	5.6
resolved constant	35	5.7
statement	36	6
unlabeled statement	37	6
simple statement	38	6
compound statement	39	6
assignment statement	40	6.1
begin statement	41	6.2
if statement	42	6.3
case statement	43	6.4
repeat statement	44	6.5
exit statement	45	6.6
return statement	46	6.7
goto statement	47	6.8
procedure declaration	48	7.1

procedure invocation statement	49	7.1
function declaration	50	7.2
function invocation primary	51	7.2
formal parameters	52	7.3
actual parameters	53	7.3
capsule declaration	54	8.1
capsule invocation declaration	55	8.1
visible list	56	8.2
exception declaration	57	9.1
guard statement	58	9.2
raise statement	59	9.3
reraise statement	60	9.4
task declaration	61	10.1
task invocation statement	62	10.1
wait statement	63	10.5
waiting invocation	64	10.5
region statement	65	10.7
formal translation time properties	66	11.1.2
actual translation time properties	67	11.1.2
translation time property	68	11.1.2
generic declaration	69	11.3
generic parameters	70	11.3
generic constraint	71	11.3
type generic constraint	72	11.3.1
subtype generic constraint	73	11.3.2
operation generic constraint	74	11.3.3
value generic constraint	75	11.3.4
needs	76	11.4
needed item	77	11.4
representational specification	78	12.2
representational item	79	12.2
location specification	80	12.3
definable symbol	81	13.1
pragmats	82	B
pragmat	83	B

DIAGRAM CROSS-REFERENCE

(alphabetically)

XREF OF LEXICAL NAMES

	Diagram Identifier	Section Number
boolean literal		
literal	D	2.3.5
comment		
token separator	B	2.2
enum literal		
literal	D	2.3.5
type	10	4.1.5
float literal		
literal	D	2.3.5
identifier		
abbreviation declaration	17	4.4.1
abbreviation invocation	18	4.4.1
attribute inquiry	25	4.5.3
capsule declaration	54	8.1
capsule invocation declaration	55	8.1
constant declaration	16	4.2.1
definable symbol	81	13.1
direct type declaration	20	4.4.2
enum literal	G	2.3.7
exception declaration	57	9.1
exit statement	45	6.6
formal parameters	52	7.3
function declaration	50	7.2
function invocation primary	51	7.2
generic parameters	70	11.3
goto statement	47	6.8
guard statement	58	9.2
imports	9	3.7
indirect type declaration	23	4.4.3
needed item	77	11.4
procedure declaration	48	7.1
procedure invocation statement	49	7.1
raise statement	59	9.3
record or union constructor	32	5.6
referencing form	30	5.3
repeat statement	44	6.5
representational item	79	12.2
statement	36	6

subtype	12	4.1.6
subtype generic constraint	73	11.3.2
task declaration	61	10.1
task invocation statement	62	10.1
token	A	2.2
translation time property	68	11.1.2
type	10	4.1.5
type generic constraint	72	11.3.1
unlabeled statement	37	6
user-defined subtype	22	4.4.2
user-defined type	21	4.4.2
variable declaration	15	4.2.1
visible list	56	8.2
waiting invocation	64	10.5

indirect literal

literal	D	2.3.5

int literal

literal	D	2.3.5

literal

constant	29	5.3
token	A	2.2

pragmat token separator

string literal

literal	D	2.3.5

token

token separator

DIAGRAM CROSS-REFERENCE

(alphabetically)

XREF OF SYNTACTIC NAMES

Index