

Interim Version of 4th Working Draft Modula-2 Standard

Section intro

0 Foreword

There will be some notes here about origins of Modula-2, including credit to Professor Wirth as inventor (also development from Pascal and Modula, history of the standardisation programme etc).

This section will also cover: features and application domain of Modula-2, significant changes/additions to the language/library, remarks about the syntactic metalanguage, notes (to be drafted by Keith Hopper) about the assumed uniprocessor model and credits to individuals involved, which will take the following form:

Kees Pronk, Delft University of Technology, Netherlands

In the foregoing example, both an individual and the organisation supporting his or her involvement in the work of ISO/IEC JTC1/SC22/WG13 are credited. It is the intention of WG13 to give credit to all who participated in the formulation of the standard through attendance of meetings - having first formally obtained the assent of both the individual (and, where appropriate, their supporting organisation) to be listed in the document.

This International Standard uses the Vienna Development Method - Specification Language (VDM-SL) to define the semantics of the language in a mathematically rigorous manner. VDM is based upon conventional mathematics (see Annex D). However, the semantics of the language is also specified in natural language, albeit in a form that is bound to be ambiguous. If the natural language requirements and those expressed in the VDM-SL conflict, then there is a mistake in the standard. In many cases, small details of the language semantics may only be resolved by reference to the VDM-SL. By this means, the natural language definition can be kept relatively simple.

The development of large parts of the formal specification of the semantics of Modula-2 in this International Standard was sponsored by the UK Alvey Research Programme. The Alvey sponsored work was carried out at Leicester University.

NOTE — the base document for formulation of this International Standard was Programming in Modula-2 by Niklaus Wirth, various editions, published by Springer-Verlag.

Temporary notes:

i) Currently, the style of the VDM-SL in this draft differs from that anticipated in the forthcoming international standard for VDM-SL (see Normative References). It is intended that the style be brought in line with the VDM-SL standard in due course.

ii) In order to satisfy the requirement that only editorial changes have been made in this third working draft of the Modula-2 standard, over and above those decisions taken at the last WG13 meeting, certain presentational methods have been adopted. In particular section 8.2 is simply a place holder and Annex G contains two alternative forms of the proposed exception handling library module. In addition, some temporary notes and indications of "To do" against specific points have been included for the benefit of the reader.

iii) WG13 has indicated that work on four further aspects of the standard library may be included in future working drafts of the standard, these deal with BCD (for binary coded decimal arithmetic), internationalisation, date and time manipulation and support for scientific applications.

iv) WG13 also plans to make provision for alternative national character set representations, but the exact form of this has not yet been developed.

v) This draft includes definition modules of proposed standard library modules that contain implementation defined types and constant values. The precise way of dealing with this has not yet been decided.

1 SCOPE

1.1 This International Standard specifies

- syntax of the language;
- required symbols for program representation, literal strings, comments and data;
- semantics of the language, including those violations of the standard that a conforming implementation is required to detect;
- required system modules that a conforming implementation must supply;
- required separate modules that a conforming implementation must supply;
- standard library;
- compliance requirements for implementations.

1.2 This International Standard does not specify

- the underlying representation of predefined data types;
- the method by which the implementation is invoked (including identification of the program module and associated definition and implementation modules);
- performance and certain capacity aspects;
- the correspondence between module names and system file names (where files are used);
- the effect of executing a program that contains any violation of the standard, unless the standard exception handling mechanism has been invoked.

2 NORMATIVE REFERENCES

The following standards contain provisions which, through reference in this text, constitute provisions of this International Standard. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based upon this International Standard are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below. Members of IEC and ISO maintain registers of currently valid International Standards.

ISO 646, *The 7-bit coded character set for information processing exchange*

BS 6154, *Method of defining syntactic metalanguage*

ISO xxxx, *The Vienna Development Method - Specification Language* (VDM-SL)

3 DEFINITIONS

For the purpose of this International Standard, the following definitions apply:

3.1 Processor: The combination of hardware, operating system, implementation, etc. which permits the translation and execution of Modula-2 programs. (A “processor” may include two machines, if cross- compilation is involved.)

3.2 Implementation: An implementation of Modula-2 comprises the following:

- a translator
- the required system and separate modules
- optionally, modules of the standard library
- proprietary system or separate modules
- any software needed to support or integrate the above

An implementation provides the means by which separate modules may be combined to produce an executable program.

3.3 Implementation defined: A feature of the language or standard library whose behaviour may be constrained by an implementation or whose attributes (e.g. the range of values of a type) may be chosen by an implementor. (For example the range of values of the **INTEGER** type is implementation defined.) An implementation defined language feature must be supplied in an implementation; an implementation defined library feature must be supplied if the library module which contains it is supplied. Implementation defined behaviour must be predictable, and implementation defined attributes must always be available in an implementation. This International Standard requires that such features be fully documented by an implementor (and may therefore be validated). See clause 4.8 for documentation requirements.

3.4 Implementation dependent: A feature of the language or standard library whose precise behaviour is at the discretion of an implementor. (For example the order of evaluation of operand in an expression is implementation dependent.) An implementation dependent language feature must be supplied in an implementation; an implementation dependent library feature must be supplied if the library module which contains it is supplied. Implementation dependent behaviour is not necessarily predictable.

3.5 Error: A construct in a program which is in violation of the syntax or static semantics of the language, thus preventing meaning being ascribed to any text containing the construct. (For example an “expression” which includes a plus operator between two arrays is an error.) An error prevents valid execution and must be detected by an implementation.

3.6 Exception: The violation, by a program, of the dynamic semantics of the language, or of a library module. It is a run-time occurrence which invalidates the normal semantics of an executing program. (For example, dereferencing a pointer variable whose value is **NIL**, is an exception.)

An exception in a program may be detected and reported by an implementation, in which case this International Standard defines the semantics of the program, thus:

- either exceptional termination shall occur, or
- the program shall provide a procedure to handle the exception (cf. 7.3 the standard module **EXCEPTIONS**).

If an implementation does not detect an exception, or if, for other reasons, an implementation permits execution to continue after an exception has occurred, no meaning is given to the program by the standard.

NOTE — This International Standard requires that certain exceptions be detected by an implementation in some mode of its use. The function *mandatory-exception* is used in the formal specification in these cases and *non-mandatory-exception* is used in other cases.

4 REQUIREMENTS

An implementation complying with the requirements of this International Standard shall meet the following:

4.1 Language and ordering of declarations

A conforming implementation shall:

- a) accept programs for translation in the form specified in clause 5;
- b) accept all the features of the language specified in clause 6 with the meanings defined in clause 6;
- c) with respect to the two alternative predicates for declarations given in clause 6.2.2.1, translate source code in one of two ways:
 - 1) if the implementation is of a type identified as ‘declare before use in declarations’ (see clause 4.8a), accept all features of the language - i.e. the first predicate in clause 6.2.2.1 shall apply;
 - 2) if the implementation is of a type identified as ‘declare before use’ (see clause 4.8a), then it shall reject any program not satisfying the second predicate in clause 6.2.2.1 and issue a unique identifying message during translation.

NOTES

- a) Clause 6.2.2.1 contains two alternative predicates for declarations - and implementors may choose which approach to support. It is expected that so-called ‘multi-pass’ implementations (i.e. those that require programs to observe a ‘declare before use in declaration’ rule) will support the predicate that returns true and that so-called ‘single-pass’ implementations (i.e. those that require programs to observe a more strict ‘declare before use’ rule) will support the alternative predicate.
- b) The requirement to provide a message when the first predicate for declarations (in clause 6.2.2.1) is not satisfied does not preclude a conforming implementation from emitting other (possibly erroneous) messages first. The exact form of the required message is not defined by this international standard, due to the varying translation styles and user interfaces.
- c) It should be noted that there is a requirement for all conforming implementations to accept, and process correctly, the **FORWARD** directive.
- d) In one respect, the VDM-SL behaves like an implementation in giving a rule which satisfies the requirements but does not give the requirements themselves. This issue concerns the evaluation order, mainly that of constituents within expressions. The natural language states that any evaluation order is permitted while the VDM-SL is written by giving only one such order. When this situation arises, a technical note is added to the VDM-SL to indicate the over-specification in this case.
- e) The definition overspecifies the *required* behaviour of the procedures in separate modules in the case where there is concurrent use of the procedures: *None* of the operations is required to be atomic. For example (from the I/O library section ??): If the input stream is

Hello Professor, we hope you are well.

and two concurrent calls of `ReadToken` were made in the program, it is *not* required that, after the calls are complete, one array contain “Hello” and the other “Professor,”. Such a program is incorrect (because it shares a common resource between concurrent activities without arranging for mutual exclusion between them) and the result is *undefined*.

4.2 Required Modules

A conforming implementation shall provide all the required system modules specified in Clause 7 and all the required separate modules specified in Clause 8. The required system module **SYSTEM** may be extended but not reduced by the implementor. All other required modules may not be extended or reduced by the implementor.

NOTE — The phrase “not reduced” is meant to prevent features specified by the standard from being omitted by the implementor.

A conforming implementation may not provide other system modules.

4.3 Errors

A conforming implementation shall determine whether or not a program contains errors and shall report the result of this determination to the user of the implementation. If the implementation has not attempted to discover all errors after one has been reported, this fact shall also be reported to the user.

4.4 Exceptions

A conforming implementation shall treat exceptions in the following manner:

- a) It shall have a mode of use whereby all language exceptions (specified in clause 6) designated mandatory shall be detected (either before or during execution) and a report propagated through to the environment from which the program was invoked. The method of reporting shall be in accordance with that specified in clause 6.12.
- b) It shall be free to detect or not to detect all other types of language exceptions (specified in clause 6). The documentation requirements of clause 4.8c shall apply to any modes of use which are utilised to provide such detection.
- c) It shall detect (either before or during execution) all exceptions specified in system modules (defined in clause 7) and propagate a report through to the environment from which the program was invoked. The method of reporting shall be in accordance with that specified in clause 6.12.
- d) It shall detect (either before or during execution) all exceptions specified in standard library modules (defined in clauses 8 and 9) and propagate a report through to the environment from which the program was invoked. The method of reporting shall be in accordance with that specified in clause 6.12.

Temporary note: The "mode of use" terminology is designed to allow the generation of code to detect and report exceptions to be user selectable. It is intended that all exceptions defined in the standard be classified and that detection of a core subset be mandated in this way. Details of the proposals for mandatory exception detection are given in Annex F. The method of reporting proposal is still under development.

NOTE —

- a) The requirement to propagate the report to the environment from which the program was invoked is designed to avoid reference to the program user (since there may not be one, in an embedded system application). It is accepted that reporting an exception from an embedded system application may have no meaning (and disastrous consequences!).
- b) There is no implied requirement that implementors provide a 'traceback' facility.
- c) Sub-clauses (c) and (d) of this requirement arise from the fact that all exceptions defined for system and standard library modules have been classified as mandatory, and therefore there is no need to distinguish between two types - as there is for language exceptions.

4.5 Extensions

A conforming implementation shall be able to detect the inclusion of extensions to the language included in any source text offered for translation.

4.6 Implementation-defined features

A conforming implementation shall process implementation-defined aspects of the language in the manner described in the implementation documentation (see clause 4.8b).

4.7 Implementation-dependent features

A conforming implementation is free to detect and report, or otherwise process, any use of implementation-dependent features.

NOTE — Reliance on an implementation dependency, such as order of evaluation of parameters, is in fact an error in a program. However, few production compilers will be able to reliably detect this. Hence, Clause 4.7 cannot be more specific about processing of implementation dependent features.

4.8 Documentation

A conforming implementation shall be accompanied by documentation which includes the following features:

- a) a statement as to whether the implementation supports the ‘declare before use in declarations’ or ‘declare before use’ semantics - see clauses 4.1c and 6.2.2.1.
- b) a definition of all implementation defined features;
- c) a statement defining the exception-detection capability of the implementation; this statement shall specify the influence of any appropriate implementation options (or modes) that are provided;
- d) a general statement about processing of implementation dependent features, especially where these are treated as undetected errors;
- e) a statement describing any features accepted by the implementation that are extensions;
- f) a statement describing any system or separate library modules that are provided (other than those described in Clauses 7, 8 and 9);
- g) information about the facilities that are available to enable users to produce the equivalent of all non-system library modules. Such information shall be provided in the case of all required (non- system) modules. In the case of non-required modules, such information shall be included for all modules that are provided with the implementation, and for all non-provided modules where the information is relevant to the type of implementation.

NOTE — this sub-clause is to be interpreted such that cross-compilers for embedded systems applications do not need to be accompanied by information on facilities to implement the standard input/output library, if the standard input/output library is not itself provided with the compiler. This clause is still under study and development, particularly in regard to facilities to incorporate modules written in other programming languages.

4.9 Statement of Compliance

A conforming implementation shall be accompanied by a compliance statement, conforming to the following requirements:

- a) If the implementation complies in all respects with the requirements of this standard, the compliance statement shall be:

“<This implementation> complies with the requirements of ISO xxxx.”

- b) If the implementation complies with some but not all of the requirements of this standard, the compliance statement shall be:

“<This implementation> partially complies with the requirements of ISO xxxx, details of the non-conformities are as follows:”

followed by a reference to or a complete list of the requirements of this standard with which the implementation does not comply.

- c) In either (a) or (b) (whichever is applicable), the text <This implementation> shall be replaced by an unambiguous name identifying the implementation.

4.10 Separate Modules

If any or all of the separate modules provided with the implementation have the same module names as the standard modules specified in Clause ??, the provided modules shall conform to the appropriate requirements of Clause ??, and shall not be extended or reduced in terms of the types, constants, variables or procedures that they export. If any of the objects of one of the separate modules provided with a conforming implementation has the same name as an object of one of the modules specified in Clause ??, the entire standard separate module shall be provided.

5 Lexis

The Modula-2 Lexis is the specification of the lexical elements of Modula-2 and the specification of the structure of source code which a conforming implementation shall accept. Furthermore the Lexis is the specification of how a conforming implementation shall accept white space, comments and source directives in a compilation unit.

This Lexis is formally expressed using the *Standard Syntactic Metalanguage* (BS 6154, 1981); all the symbols and pervasive identifiers of Modula-2 are defined below, in this clause.

NOTES

1 This International Standard does not specify how compilation units be stored nor whether a conforming implementation may accept a sequence of compilation units as input from a single document.

2 The lexis is formally expressed as a syntax which defines tokens in terms of characters. A conforming implementation may accept a tokenized compilation unit, which has been produced according to the syntax, rather than as a stream of characters.

A conforming implementation shall provide all keywords and required symbols specified below and shall recognize identifiers and literals as specified.

A conforming implementation shall provide preferred left and right brackets if its processor supports them. A conforming implementation shall provide the required left and right brackets.

A conforming implementation shall provide preferred left and right braces if its processor supports them. A conforming implementation shall provide the required left and right braces.

A conforming implementation shall provide the preferred case separator if its processor supports it. A conforming implementation shall provide the required case separator.

A conforming implementation shall provide preferred dereferencing operator if its processor supports it. A conforming implementation shall provide the required dereferencing operator.

A comment shall have no effect on the meaning of a standard program.

A source code directive shall have no effect on the meaning of a standard program. An implementation shall detect, and shall issue a warning concerning, a source code directive whose body is not recognised.

5.1 Source Code Structure

modula2 source code =
 { separator }, (delimited word, { delimited word | symbol | separator } | symbol, { delimited word | symbol | separator }) ;

delimited word = word token, (separator | symbol) ;

5.2 Words

word token = identifier | keyword | numeric literal | character number literal ;

5.3 Identifiers

identifier = full identifier | portable identifier | pervasive identifier ;

5.3.1 Portable Identifiers

portable identifier = (simple letter | low line), { [low line], simple alphanumeric } ;

low line = "_" ;

CHANGE — The Lowline symbol ('_') is permitted anywhere in an identifier and may be repeated.

5.3.2 Full Identifiers

full identifier = (letter | low line), { [low line], national alphanumeric } ;

5.3.3 Pervasive Identifiers

pervasive identifier =

"ABS"	"BOOLEAN"	"CARDINAL"	"CAP"
"CHR"	"CHAR"	"COMPLEX"	"CMPLX"
"DEC"	"DISPOSE"	"ENTER"	"EXCL"
"FALSE"	"FLOAT"	"HALT"	"HIGH"
"IM"	"INC"	"INCL"	"INT"
"INTERRUPTIBLE"	"INTEGER"	"LEAVE"	"LENGTH"
"LFLOAT"	"LONGCOMPLEX"	"LONGREAL"	"MAX"
"MIN"	"NEW"	"NIL"	"ODD"
"ORD"	"PROC"	"PROT"	"PROTECTION"
"RE"	"REAL"	"SIZE"	"TRUE"
"TRUNC"	"UNINTERRUPTIBLE"	"VAL"	

CHANGE — The following pervasive identifiers have been added to the language:

COMPLEX, CMPLX, ENTER, IM, INT, INTERRUPTIBLE, LEAVE, LENGTH, LFLOAT, PROT, PROTECTION, RE and UNINTERRUPTIBLE

NOTE — Pervasive identifiers are not reserved words; if redeclared in a program, they will no longer be pervasive in the scope of the redeclared identifier; see Section 6.2 and see also 6.10.4 page (??).

5.4 Keywords

keyword =

"AND"	"ARRAY"	"BEGIN"	"BY"
"CASE"	"CONST"	"DEFINITION"	"DIV"
"DO"	"ELSE"	"ELSIF"	"END"
"EXIT"	"EXPORT"	"FOR"	"FORWARD"
"FROM"	"IF"	"IMPLEMENTATION"	"IMPORT"
"IN"	"LOOP"	"MOD"	"MODULE"
"NOT"	"OF"	"OR"	"POINTER"
"PROCEDURE"	"QUALIFIED"	"RECORD"	"REM"
"REPEAT"	"RETURN"	"SET"	"THEN"
"TO"	"TYPE"	"UNTIL"	"VAR"
"WHILE"	"WITH"		

CHANGE — The keywords FORWARD and REM have been added.

5.5 Symbols

symbol = required symbols | symbols with alternatives | quoted string | quoted character ;

5.5.1 Required Symbols

required symbols =

comma	period	ellipsis
semicolon	single quote	double quote
assignment operator	plus operator	set union operator
string catenate symbol	minus operator	set difference operator
multiplication operator	set intersection operator	division operator
symmetric set difference	and synonym	equals operator
not equals operator	not equals synonym	less than operator
greater than operator	less than or equal operator	is subset of operator
greater than or equal operator	is superset of operator	not synonym
left parenthesis	right parenthesis ;	

```

comma = "," ;

plus operator = "+" ;

set union operator = "+" ;

string catenate symbol = "+" ;

CHANGE — The string catenate symbol has been added.
```

```

period = "." ;

ellipsis = "...";

semicolon = ";" ;

single quote = "'" ;

double quote = '"' ;

assignment operator = ":@" ;

minus operator = "-" ;

set difference operator = "-" ;

multiplication operator = "*" ;

set intersection operator = "*" ;

division operator = "/" ;

symmetric set difference = "/" ;

and synonym = "&" ;

equals operator = "=" ;

not equals operator = "<>" ;

not equals synonym = "#" ;

less than operator = "<" ;

greater than operator = ">" ;

less than or equal operator = "<=" ;
```

is subset of operator = "<=" ;
greater than or equal operator = ">=" ;
is superset of operator = ">=" ;
not synonym = "~" ;
left parenthesis = "(" ;
right parenthesis = ")" ;

5.5.2 Symbols with alternatives

symbols with alternatives =
left bracket | right bracket | left brace | right brace
case separator | dereferencing operator ;

5.5.2.1 Brackets

left bracket = preferred left bracket | required left bracket ;
preferred left bracket = "[" ;
required left bracket = "(" ;
right bracket = preferred right bracket | required right bracket ;
preferred right bracket = "]" ;
required right bracket = ")" ;

CHANGE — All implementations shall provide ‘(!’ and ‘!’)’ as alternatives to ‘[’ and ‘]’.

5.5.2.2 Braces

left brace = preferred left brace | required left brace ;
preferred left brace = "{" ;
required left brace = "(" ;
right brace = preferred right brace | required right brace ;
preferred right brace = "}" ;
required right brace = ":" ;

CHANGE — All implementations shall provide ‘(:’ and ‘:)’ as alternatives to ‘{’ and ‘}’.

5.5.2.3 Case Separator

case separator = preferred case separator | required case separator ;
preferred case separator = "|" ;
required case separator = "!" ;

CHANGE — All implementations shall provide ‘!’ as an alternative to ‘|’.

5.5.2.4 Dereferencing Operator

dereferencing operator = preferred dereferencing operator | required dereferencing operator ;

preferred dereferencing operator = "^" ;

required dereferencing operator = "@" ;

CHANGE — All implementations shall provide ‘@’ as an alternative to ‘^’.

5.6 Constant Literals

5.6.1 Numeric Literals

numeric literal = whole number literal | real literal ;

5.6.2 Whole Number Literals

whole number literal = decimal number | octal number | hexadecimal number ;

decimal number = digit, { digit } ;

digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;

octal number = octal digit, { octal digit }, "B" ;

octal digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" ;

hexadecimal number = digit, { hex digit }, "H" ;

hex digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" | "A" | "B" | "C" | "D" | "E" | "F" ;

5.6.3 Real Literals

real literal = digit, { digit }, ".", { digit }, [scale factor] ;

scale factor = "E", ["+" | "-"], digit, { digit } ;

5.6.4 String Literals

quoted string = "'", { national character - "'", "" }, "" | "'", { national character - "'", "" }, "" ;

5.6.5 Character Literals

quoted character = "'", national character - "'", "" | "'", national character - "'", "" ;

character number literal = octal digit, { octal digit }, "C" ;

5.7 Letters and National characters

CHANGE — The concept of what characters are permissible as letters (for use in identifiers) and what characters are permissible within quoted strings, quoted characters, comments and source code directives has been extended to facilitate character sets not based on ISO 646.

letter = ? implementation-defined alphabetic character ? ;

national character = national graphic character | format effector | space ;

national graphic character =
 simple alphanumeric | punctuation character | national alphanumeric |
 implementation defined graphic | national punctuation ;

national alphanumeric = letter | national numeric ;

national numeric = ? implementation-defined numeric character ? ;

simple alphanumeric = simple letter | digit ;

simple letter = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M"
 | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z"
 | "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m"
 | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" ;

implementation defined graphic = ? implementation-defined character ? ;

implementation defined format effector = ? implementation-defined format effector ? ;

punctuation character = ? ISO 646 punctuation character ? ;

national punctuation = ? implementation-defined punctuation character ? ;

format effector = ? implementation-defined format effector ? ;

5.8 Separators

separator = white space | comment | source code directive ;

white space = space | newline | white space control ;

space = " " ;

new line = ? possibly empty implementation-defined character sequence ? ;

white space control = ? implementation-defined character sequence ? ;

5.8.1 Comments

comment = "(*", comment body, "*)" ;

comment body = { national character | separator } ;

5.8.2 source code directives

source code directive = "<*", directive body, ">" ;

directive body = comment body ;

CHANGE — A syntax for source code directives has been introduced.

6 The Language

To explain parts of the VDM-SL by means of natural language comments, technical notes (annotations) are added as appropriate.

The VDM-SL notation is divided into three parts. Firstly, there is the abstract syntax which gives an abstract version of the underlying structure of the language. In a few cases in which the abstract syntax is not directly related to the conventional concrete syntax, technical notes are added to explain the divergence. Secondly, there is the static semantics which gives the rules to be applied by a processor to determine the legality of a Modula-2 program. Lastly, there is the dynamic semantics which gives the rules for the execution of a program which satisfies the static semantics.

6.1 Programs and Compilation Units

A text that is accepted by a Modula-2 language processor as a single component is called a compilation unit. There are three classes of compilation unit: program modules, definition modules, and implementation modules. The compilation units of a program other than the program module form pairs of modules. each pair consisting of a definition module and an implementation module having the same identifier.

6.1.1 Programs

Concrete Syntax

A program module together with the modules it uses (and the modules they subsequently use) and any system modules used shall constitute a Modula-2 program.

compilation unit = program module | definition module | implementation module ;

NOTE — As well as these three classes of modules, there are also local modules, which may be declared within implementation modules, program modules, and procedures, but which are not compilation units in their own right.

Abstract Syntax

types

Program :: *pmod* : Program-module
 dmods : Definition-modules
 imods : Implementation-modules

;

Definition-modules = *Definition-module-set*

;

Implementation-modules = *Implementation-module-set*

;

Compilation-unit = *Program-module* | *Definition-module* | *Implementation-module*

NOTE — If a definition module does not contain any procedure definitions or opaque type definitions there must still be an associated (possibly empty) implementation module. Likewise, if there are no exports from a module and the implementation

module is used solely for initialisation, there must still be an associated (and possibly empty) definition module.

Static Semantics

For a particular program there shall be a single program module, and each definition module shall have a single corresponding implementation module and each implementation module shall have a single corresponding definition module

There shall exist at least one valid compilation order for the compilation units that make up the program.

NOTE — The current treatment of a module is dependent upon definition modules (from which it imports) being processed first. The sequence in which modules are processed by the implementation is called the “compilation order”. If there are many possible orders which satisfy the constraints given in 6.1.9, this International Standard does not define which order is chosen.

functions

```

wf-program : Program → B
wf-program (mk-Program (pmod, dmods, imods))  $\triangleq$ 
18   let  $\rho_{ext} = \text{external-environment}(dmods)$  in
32   wf-modules(pmod, dmods, imods) $\rho_{ext} \wedge$ 
20   wf-program-module(pmod) $\rho_{ext} \wedge$ 
21    $\forall dmod \in dmods \cdot \text{wf-definition-module}(dmod)\rho_{ext} \wedge$ 
22    $\forall imod \in imods \cdot \text{wf-implementation-module}(imod)\rho_{ext} \wedge$ 
35   compilation-orders(pmod, dmods, imods)  $\neq \{\}$ 

```

annotations The correctness of each of the compilation units is checked with respect to the definitions of the definition modules. The import lists of the compilation units must be such that it is possible to construct at least one a valid compilation order.

Dynamic Semantics

The effect of the execution of a Modula-2 program is described by changes to the program state giving the current values of variables and the current condition of the input and output streams. The program state shall be initialised with the input streams required by the program; and no variables shall be defined in this initial state. The termination of the execution of the program module shall be the completion of the program; no variables shall exist on this completion.

A program shall have a surrounding dynamic environment that is constructed from the environments defined by the definition modules and the implementation modules that form part of the program; together with the system environment. The program shall be executed in the context of the surrounding dynamic environment.

The entry procedure of a protected program module shall be called after the initialisation of the separate modules on which the program module depends; the exit procedure shall be called on completion of the statement part of the program module.

NOTE — The meaning of the program is given by the output produced from the input to the program. During the computation of the program, information may be held within the program (called the program state), but this is not observable directly.

functions

```

m-program : Program → External-states → External-states
m-program (pmod, dmods, imods)(in-states)  $\triangleq$ 
77   let  $\rho_{ext} = \text{merge-modules}(pmod, dmods, imods)$  in
77   let initial-state = initialise-state(in-states) in
77   execute-program(pmod, dmods, imods) $\rho_{ext}$  stop cenv initial-state

```

annotations The input and output streams are modelled by a mapping from a set of external identifiers to a sequence of values. The Modula-2 I/O library, for example, will associate each channel

identifier with an external identifier in an implementation defined manner to gain access to the input and output streams. The modules that make up the program are “link-edited”, which models the allocation of storage, and the external environment for the program constructed. The program is then executed in the context of this environment. The result of the program execution is defined by any changes to the input streams.

Auxiliary Functions

functions

$external-environment : Definition-modules \rightarrow Environment$

$external-environment(defs) \triangleq$

```
226   let  $\rho = \text{let } \rho_{defs} = \text{merge-environments}(\{d\text{-definition-module}(def)\rho \mid def \in defs\})$  in
277   overwrite-environment( $\rho_{defs}$ ) $\rho_{system}$  in
   $\rho$ 
```

annotations Those definition modules that are directly imported are elaborated to build an environment corresponding to the imports of the definition module.

operations

$merge-modules : Program-module \times Definition-modules \times Implementation-modules \xrightarrow{o} Environment$

$merge-modules(pmod, dmods, imods) \triangleq$

```
35   let  $order \in \text{compilation-orders}(pmod, dmods, imods)$  in
18   merge-external-modules( $order$ )  $\rho_{system}$ 
```

annotations The set of possible valid orders of compilation is produced and one of these orders is chosen arbitrarily. The function is modelling the link-edit process.

By merging the environments in this order, the environment of a system module will be overwritten if one of the definition modules has the same identifier as that of the system module.

;

$merge-external-modules : Module-order \rightarrow Environment \xrightarrow{o} Environment$

$merge-external-modules(order)\rho \triangleq$

```
   if  $order = []$ 
   then return  $\rho$ 
??   else def  $\rho_{new} = m\text{-compilation-unit}(\text{hd } order) \rho$ ;
18   merge-external-modules( $\text{tl } order$ )  $\rho_{new}$ 
```

annotations The declarations of each module are elaborated in order. Elaboration of the declarations allocates storage and builds an environment, and the resulting external environment for the program is constructed.

;

$execute-program : Program-module \times Definition-modules \times Implementation-modules \rightarrow Environment \xrightarrow{o} ()$

$execute-program(pmod, dmods, imods)\rho \triangleq$

```
   let  $mk\text{-Program-module}(name, imports, block, -) = pmod$  in
25   let  $names = \text{directly-imports}(imports)$  in
274   let  $\rho_{pmod} = \text{associated-module}(name)\rho$  in
??   def  $\rho_{cont} = c\text{-tix}(\{\text{HALT} \mapsto \text{terminate-actions}\}) \rho$ ;
37   ( $\text{initialise-modules}(names, dmods, imods, \{\}) \rho_{cont}$ ;
40    $\text{optional-call-of-domain-entry-procedure}() \rho_{cont}$ ;
??   def  $\rho_{pcont} = c\text{-tix}(\{\text{HALT} \mapsto \text{terminate-actions}\}) \rho_{pmod}$ ;
38   ( $\text{execute-module-block}(block) \rho_{pcont}$ ;
41    $\text{optional-call-of-domain-exit-procedure}() \rho_{cont}$  ))
```

annotations The program is executed by first initialising the modules directly imported by the program module in the order defined by their occurrence in the import lists (this may cause the initialisation of other imported modules). The local environment for the program module is retrieved from the surrounding environment and the module block of the program module is executed in the context of this environment.

terminate-actions is part of the continuation ‘*stop*’ of the execution of the main program, see 6.11.4.5

6.1.2 Program Modules

A program module consists of a collection of import lists, a collection of declarations and a sequence of statements. The modules that the program module imports (and the modules that these modules consequently import, etc.) determine the collection of modules that constitute the program. After the initialisation of the implementation modules and the initialisation of the local modules of the program module, the statement sequence of the program module is executed.

Concrete Syntax

program module = "MODULE", module identifier, [protection], ";", import lists, module block, module identifier, "." ;

module identifier = identifier ;

protection = left bracket, constant expression, right bracket ;

The module identifier components of a program module shall be identical.

A module with a protection expression in the heading shall be a directly protected module.

Abstract Syntax

types

Program-module :: *name* : *Identifier*
 imports : *Import-lists*
 block : *Module-block*
 protection : [*Expression*]

Static Semantics

The program module shall not be imported into itself. The identifiers that are imported into the program module shall not be declared in that module. Any identifiers, either implicitly or explicitly imported into the program module shall be present in the external environment. The environment for the block of a program module shall be the external environment restricted to the identifiers imported into the program module.

A protection expression specified in the heading of a directly protected program module shall be a constant expression in the pervasive identifier environment, and shall be evaluated in that environment.

functions

$wf\text{-}program\text{-}module : Program\text{-}module \rightarrow Environment \rightarrow \mathbb{B}$
 $wf\text{-}program\text{-}module (mk\text{-}Program\text{-}module (name, imports, block, protection))\rho_{ext} \triangleq$
 $name \notin elems\ directly\text{-}imports (imports) \wedge$
 $wf\text{-}import\text{-}lists (imports)\rho_{ext} \wedge$
 $consistent\text{-}imports (imports, block)\rho_{ext} \wedge$
 $let\ \rho_{block} = d\text{-}import\text{-}lists (imports)\rho_{ext}\ in$
 $wf\text{-}module\text{-}block (block)\rho_{block} \wedge$
 $is\text{-}valid\text{-}access\text{-}control (protection)\rho_{std\text{-}ids}$

Dynamic Semantics

A program shall have a surrounding dynamic environment that is constructed from the environments defined by the definition modules that constitute the program, together with an environment defined by the system modules. The environment for the execution of the program module, the definition modules, and the implementation modules shall be constructed by elaborating the definitions of definition modules in an order related to the import lists of the definition modules. Possible orders are deduced from the import lists of definition modules.

NOTES

- 1 The environment defined by the system modules contains the identifiers associated with the constants, types, variables, and procedures available in those (required) system modules.
- 2 The program module and any implementation modules are ignored in the construction of the surrounding environment.

operations

$m\text{-}program\text{-}module : Program\text{-}module \rightarrow Environment \xrightarrow{o} Environment$
 $m\text{-}program\text{-}module (mk\text{-}Program\text{-}module (name, imports, block, protection))\rho \triangleq$
 $let\ \rho_{import} = d\text{-}import\text{-}lists (imports)\rho\ in$
 $def\ pval = evaluate\text{-}protection (protection)\ \rho_{std\text{-}ids};$
 $let\ \rho_{internal} = add\text{-}protection (pval)\rho_{import}\ in$
 $def\ \rho_{block} = m\text{-}module\text{-}block (block)\ \rho_{internal};$
 $return\ overwrite\text{-}mod\text{-}environment (\{name \mapsto \rho_{block}\})\rho$

annotations

The environment for the block component of a program module is derived from the environment defined by all the definition modules that make up the program. It is derived from that environment by only importing those modules listed in the import lists. The protection expression (if present) is evaluated in an environment consisting of the standard identifiers only and is added to the environment for use by the protection procedures.

6.1.3 Definition Modules

The compilation units of a program other than the program module form pairs of modules. Each pair of modules consists of a definition module and an implementation module. The definition module is used to define the objects that are exported, i.e. to define the objects belonging to the definition module/implementation module pair that may be used in other modules.

Concrete Syntax

definition module = "DEFINITION", "MODULE", module identifier, ";", import lists, definitions, "END", module identifier, "." ;

The module identifier components of a definition module shall be identical.

Abstract Syntax

types

Definition-module :: *name* : *Identifier*
 imports : *Import-lists*
 defs : *Definitions*

Declaration Semantics

functions

d-definition-module : *Definition-module* \rightarrow *Environment* \rightarrow *Environment*

d-definition-module (*mk-Definition-module* (*name*, *imports*, *defs*)) $\rho_{ext} \triangleq$

24 *let* $\rho_{block} = d\text{-import-lists}(\text{imports})\rho_{ext}$ *in*
232 *let* $\rho_{def} = \text{overwrite-environment}(d\text{-definitions}(\text{defs})\rho_{def})\rho_{block}$ *in*
274 *construct-module-environment* (*name*, ρ_{def})

annotations The surrounding environment is restricted by the import list, and then used in the elaboration of the definitions component of the definition module. The result of this function is the environment constructed from the elaboration of the definitions.

Static Semantics

A definition module shall not be imported into itself. The identifiers that are imported into the definition module shall not be redefined in that module. The environment for the elaboration of the definitions of the definition module shall be the external environment restricted to the identifiers imported into the definition module.

functions

wf-definition-module : *Definition-module* \rightarrow *Environment* $\rightarrow \mathbb{B}$

wf-definition-module *mk-Definition-module* (*name*, *imports*, *defs*) $\rho_{ext} \triangleq$

name $\notin \text{elems directly-imports}(\text{imports}) \wedge$
25 *wf-import-lists* (*imports*) $\rho_{ext} \wedge$
28 *consistent-imports* (*imports*, *block*) $\rho_{ext} \wedge$
24 *let* $\rho_{block} = d\text{-import-lists}(\text{imports})\rho_{ext}$ *in*
44 *wf-definitions* (*defs*) ρ_{block}

annotations The import lists are checked against the surrounding environment (as only objects associated with identifiers that occur in the surrounding environment may be imported) and then the definitions are checked using the surrounding environment restricted by the import list.

Dynamic Semantics

operations

m-definition-module : *Definition-module* \rightarrow *Environment* \xrightarrow{o} *Environment*

m-definition-module (*mk-Definition-module* (*name*, *imports*, *defs*)) $\rho_{ext} \triangleq$

24 *let* $\rho_{block} = d\text{-import-lists}(\text{imports})\rho_{ext}$ *in*
277 *let* $\rho_{def} = \text{overwrite-environment}(m\text{-definitions}(\text{defs})\rho_{def})\rho_{block}$ *in*
274 *return* *overwrite-mod-environment* ($\{\text{name} \mapsto \rho_{def}\}$) ρ_{ext}

annotations The elaboration of a definition module is defined by restricting the surrounding environment to the import lists, and then elaborating the definitions of the definition module in this restricted environment. The result of the elaboration is an environment.

6.1.4 Implementation Modules

An implementation module has three purposes. Firstly, it declares the details about objects exported from the associated definition module that are hidden from the modules that use these objects; secondly, it declares objects that are local to the implementation module, and thus hidden from modules that use the objects exported from the definition module; and thirdly, it defines the statements that are executed prior to any use of the exported objects.

Concrete Syntax

```
implementation module =  
  "IMPLEMENTATION", "MODULE", module identifier, [ protection ], ";", import lists, module block, module identifier, "." ;
```

The module identifier components of an implementation module shall be identical.

Abstract Syntax

types

$$\begin{aligned} \textit{Implementation-module} :: \textit{name} & : \textit{Identifier} \\ & \textit{imports} : \textit{Import-lists} \\ & \textit{block} : \textit{Module-block} \\ & \textit{protection} : [\textit{Expression}] \end{aligned}$$

Static Semantics

An implementation module shall not be imported into itself. The identifiers that are imported into the implementation module shall not be declared in that module. Any identifiers, either implicitly or explicitly imported into the implementation module shall be present in the external environment. The environment for the block of an implementation module shall be the external environment restricted to the identifiers imported into the implementation module.

A protection expression specified in the heading of a directly protected implementation module shall be a constant expression in the pervasive identifier environment, and shall be evaluated in that environment.

functions

$$\begin{aligned} & \textit{wf-implementation-module} : \textit{Implementation-module} \rightarrow \textit{Environment} \rightarrow \mathbb{B} \\ & \textit{wf-implementation-module} (\textit{mk-Implementation-module} (\textit{name}, \textit{imports}, \textit{block}, \textit{protection})) \rho_{ext} \triangleq \\ & \quad \textit{name} \notin \textit{elems directly-imports} (\textit{imports}) \wedge \\ 25 \quad & \textit{wf-import-lists} (\textit{imports}) \rho_{ext} \wedge \\ 28 \quad & \textit{consistent-imports} (\textit{imports}, \textit{block}) \rho_{ext} \wedge \\ 274 \quad & \textit{let } \rho_{def} = \textit{associated-module} (\textit{name}) \rho_{ext} \textit{ in} \\ 24 \quad & \textit{let } \rho_{block} = \textit{d-import-lists} (\textit{imports}) \rho_{ext} \textit{ in} \\ 276 \quad & \textit{let } \rho_{mod} = \textit{merge-environments} (\{\rho_{import}, \rho_{def}\}) \textit{ in} \\ 23 \quad & \textit{wf-module-block} (\textit{block}) \rho_{mod} \wedge \\ 41 \quad & \textit{is-valid-access-control} (\textit{protection}) \rho_{std-ids} \end{aligned}$$

Dynamic Semantics

NOTE — The storage associated with an implementation module is allocated before the initialisation of any of the modules.

operations

$m\text{-implementation-module} : \text{Implementation-module} \rightarrow \text{Environment} \xrightarrow{o} \text{Environment}$

$m\text{-implementation-module} (mk\text{-Implementation-module} (name, imports, block, protection)) \rho_{ext} \triangleq$

```

274 let  $\rho_{def} = associated\text{-module} (name) \rho_{ext}$  in
24 let  $\rho_{block} = d\text{-import-lists} (imports) \rho_{ext}$  in
276 let  $\rho_{mod} = merge\text{-environments} (\{\rho_{import}, \rho_{def}\})$  in
41 def  $pval = evaluate\text{-protection} (protection) \rho_{std\text{-ids}}$ ;
275 let  $\rho_{mod'} = add\text{-protection} (pval) \rho_{mod}$  in
24 def  $\rho_{block} = m\text{-module-block} (block) \rho_{mod'}$ ;
274 return  $overwrite\text{-mod-environment} (\{name \mapsto \rho_{block}\}) \rho_{ext}$ 

```

annotations The elaboration of an implementation module is defined by restricting the surrounding environment to the import lists and to the definitions contained in the associated definition module, and then elaborating the declarations of the implementation module in this restricted environment. The result of the elaboration is an environment.

6.1.5 Module Blocks

A module block consists of a sequence of declarations and an optional sequence of statements. It is a component of a program module, an implementation module, or a local module.

Concrete Syntax

module block = declarations, ["BEGIN", statement sequence], "END" ;

Abstract Syntax

types

$Module\text{-block} :: decls : \text{Declarations}$
 $actions : \text{Block-body} ;$

$Block\text{-body} = \text{Statement}^*$

Static Semantics

The control variable of any for statement contained in the statement sequence shall be declared as a simple variable in the declarations. Any return statements contained within the statement sequence shall be simple return statements. Any exit statements contained in the statement sequence shall be contained within a loop statement of the statement sequence.

functions

$wf\text{-module-block} : \text{Module-block} \rightarrow \text{Environment} \rightarrow \mathbb{B}$

$wf\text{-module-block} (mk\text{-Module-block} (decls, actions)) \rho \triangleq$

```

276 let  $\rho_{body} = overwrite\text{-environment} (d\text{-declarations} (decls) \rho_{block}) \rho$  in
47 wf-module-declarations (decls)  $\rho_{body} \wedge$ 
77 wf-block-body (actions)  $\rho_{body} \wedge$ 
132 wf-control-variables (decls, actions)  $\rho_{body} \wedge$ 
139 wf-simple-return-statements (actions)  $\wedge$ 
140 wf-exit-statements (actions)

```


annotations The environment for the statement sequence component of a module block is constructed by elaborating the declarations of the block to give an environment, and overwriting the surrounding environment with this environment.

CHANGE — Both “single-pass” (enforce declare before use) and “multi-pass” (enforce declare before use in declarations only) implementations are permitted by this International Standard (see 6.2.2.1). Forward use is allowed in the context of **POINTER T0** and in procedure types.

Dynamic Semantics

The dynamic semantics of a module block shall be the allocation of the storage associated with the block. The execution of the block body shall occur during module initialisation (see 6.1.10).

operations

$$m\text{-module-block} : \text{Module-block} \rightarrow \text{Environment} \xrightarrow{o} \text{Environment}$$

$$m\text{-module-block}(mk\text{-Module-block}(decls, -))\rho \triangleq$$

$$47 \quad m\text{-declarations}(decls)\rho$$

6.1.6 Import Lists

An import list defines the objects which are to be imported into a program, definition, implementation, or local module by listing the identifiers associated with each of the objects. Identifiers which are defined by the closure of the imported identifiers are implicitly imported.

Concrete Syntax

import lists = { import list } ;

Abstract Syntax

types

$$\text{Import-lists} = \text{Import-list}^*$$

Declaration Semantics

functions

$$d\text{-import-lists} : \text{Import-lists} \rightarrow \text{Environment} \rightarrow \text{Environment}$$

$$d\text{-import-lists}(imports)\rho \triangleq$$

$$226 \quad \text{merge-environments}(\{d\text{-import-list}(import)\rho \mid import \in \text{elems } imports\})$$

annotations An environment is constructed from the surrounding environment by restricting it to define only those identifiers occurring in an import list component, together with any implicitly imported identifiers which are defined by the closure of the imported identifiers; such an environment is constructed for each import list of the module. The result of this function is the environment formed by merging these environments.

Static Semantics

An identifier shall only appear once in all of the import lists of a module. Each import list shall contribute to the internal environment for the module that contains it.

NOTE — An identifier may be associated with only one object in a single scope.

functions

$wf\text{-}import\text{-}lists : Import\text{-}lists \times Environment \rightarrow \mathbb{B}$

$wf\text{-}import\text{-}lists(import)s\rho \triangleq$

26 $\forall import \in \text{elems } imports \cdot wf\text{-}import\text{-}list(import)\rho \wedge$

25 $\forall i, j \in \text{inds } imports \cdot i = j \vee (imported\text{-}names(import(i)) \cap imported\text{-}names(import(j)) = \{\})$

Auxiliary Functions

functions

$directly\text{-}imports : Import\text{-}lists \rightarrow Identifier^*$

$directly\text{-}imports(import)s \triangleq$

25 $[imported\text{-}module\text{-}name(import(i)) \mid i \in \text{inds } imports]$

annotations Construct a sequence of identifiers containing the identifiers of all the modules whose identifiers occur in the import lists of a module.

;

$imported\text{-}module\text{-}name : Import\text{-}list \rightarrow Identifier^*$

$imported\text{-}module\text{-}name(import) \triangleq$

cases $import$:

$mk\text{-}Simple\text{-}import(ids) \rightarrow ids,$

$mk\text{-}Unqualified\text{-}import(from, -) \rightarrow [from]$

end

annotations Construct a sequence of identifiers containing the identifiers of all the modules that occur in an import list of a module.

;

$imports\text{-}of : Import\text{-}lists \rightarrow Identifier\text{-}set$

$imports\text{-}of(import)s \triangleq$

25 $\bigcup \{imported\text{-}names(import) \mid import \in \text{elems } imports\}$

annotations Construct a set of identifiers from the import lists of a module.

;

$imported\text{-}names : Import\text{-}list \rightarrow Identifier\text{-}set$

$imported\text{-}names(import) \triangleq$

cases $imports$:

$mk\text{-}Simple\text{-}import(ids) \rightarrow \text{elems } ids,$

$mk\text{-}Unqualified\text{-}import(-, ids) \rightarrow ids$

end

annotations Construct a set of identifiers from the import list of a module. The identifiers in the set represent objects that are imported into a module. Note that the identifier in the **FROM** component (ie. the name of the module) of an unqualified import is not in the set.

6.1.6.1 Import List

Concrete Syntax

import list = simple import | unqualified import ;

Abstract Syntax

types

$$\textit{Import-list} = \textit{Simple-import} \mid \textit{Unqualified-import}$$

Declaration Semantics

functions

$$d\text{-import-list} : \textit{Import-list} \rightarrow \textit{Environment} \rightarrow \textit{Environment}$$

$$d\text{-import-list}(\textit{import})\rho \triangleq$$

$$\begin{array}{l} 26 \quad (is\text{-Simple-import}(\textit{import}) \rightarrow d\text{-simple-import}(\textit{import})\rho, \\ 27 \quad is\text{-Unqualified-import}(\textit{import}) \rightarrow d\text{-unqualified-import}(\textit{import})\rho) \end{array}$$

Static Semantics

functions

$$wf\text{-import-list} : \textit{Import-lists} \times \textit{Environment} \rightarrow \mathbb{B}$$

$$wf\text{-import-list}(\textit{imports})\rho \triangleq$$

$$\begin{array}{l} 27 \quad (is\text{-Simple-import}(\textit{import}) \rightarrow wf\text{-simple-import}(\textit{import})\rho, \\ 28 \quad is\text{-Unqualified-import}(\textit{import}) \rightarrow wf\text{-unqualified-import}(\textit{import})\rho) \end{array}$$

6.1.6.2 Simple Imports

A simple import specifies a list of identifiers. Each identifier appearing in the list may be used in the module containing the simple import; it refers to the object denoted by the identifier that belongs to the environment immediately surrounding the module.

Concrete Syntax

simple import = "IMPORT", module identifier list, ";" ;

module identifier list = identifier list ;

Abstract Syntax

types

$$\textit{Simple-import} :: \textit{ids} : \textit{Identifier}^*$$

Declaration Semantics

functions

$$d\text{-simple-import} : \textit{Simple-import} \rightarrow \textit{Environment} \rightarrow \textit{Environment}$$

$$d\text{-simple-import}(\textit{mk-Simple-import}(\textit{ids}))\rho \triangleq$$

$$\begin{array}{l} 29 \quad \textit{let closure} = \bigcup \{x\text{-closure}(id)\rho \mid id \in \textit{elems ids}\} \textit{ in} \\ 277 \quad \textit{restrict-environment}(\textit{closure})\rho \end{array}$$

annotations This operation constructs an environment from the surrounding environment by restricting it to the identifiers occurring in the import list, together with any implicitly imported identifiers which are defined by the closure of the imported identifiers.

Static Semantics

Each of the identifiers in the identifier list shall be distinct. Each of the identifiers in the identifier list shall denote an object which is known in the environment surrounding the module containing the simple import.

functions

$wf-simple-import : Simple-import \rightarrow Environment \rightarrow \mathbb{B}$

$wf-simple-import (mk-Simple-import (ids)) \rho \triangleq$
 $\forall i, j \in \text{inds } ids \cdot i = j \vee ids(i) \neq ids(j) \wedge$

276 $\text{elems } ids \subseteq \text{identifiers-in-scope}(\rho)$

6.1.6.3 Unqualified Import

An unqualified import specifies a module identifier and a list of identifiers. Each identifier in the list may be used in the module containing the unqualified import; it refers to the object denoted by this identifier that belongs to the environment of the specified module identifier.

Concrete Syntax

unqualified import = "FROM", module identifier, "IMPORT", identifier list, ";" ;

Abstract Syntax

types

$Unqualified-import :: from : Identifier$
 $ids : Identifier\text{-}set$

The number of elements in the set of identifiers of the abstract representation of an unqualified import shall be equal to the number of identifiers in the identifier list of the concrete syntax.

NOTE — Thus the identifier list of the unqualified import clause contains distinct identifiers.

Declaration Semantics

functions

$d-unqualified-import : Unqualified-import \rightarrow Environment \rightarrow Environment$

$d-unqualified-import (mk-Unqualified-import (from, ids)) \rho \triangleq$

274 $\text{let } \rho_{mod} = \text{associated-module}(from) \rho \text{ in}$
29 $\text{let } closure = \bigcup \{x\text{-closure}(id) \rho_{mod} \mid id \in ids\} \text{ in}$
277 $\text{restrict-environment}(closure) \rho_{mod}$

annotations This function constructs an environment by accessing, within the surrounding environment, the environment of the module associated with the identifier appearing in the **FROM** component of the unqualified import. This environment is restricted by the identifiers occurring in the identifier list, together with any implicitly imported identifiers which are defined by the closure of the imported identifiers.

Static Semantics

The module identifier shall be the identifier of a module which is known in the environment surrounding the module containing the unqualified import. Each of the identifiers in the identifier list shall be exported from this module.

functions

$wf\text{-}unqualified\text{-}import : Unqualified\text{-}import \rightarrow Environment \rightarrow \mathbb{B}$
 $wf\text{-}unqualified\text{-}import (mk\text{-}Unqualified\text{-}import (from, ids))\rho \triangleq$
276 $from \in identifiers\text{-}in\text{-}scope (\rho) \wedge$
273 $is\text{-}module (from)\rho \wedge$
28 $is\text{-}exported (from, ids)\rho$

Auxiliary Functions

functions

$is\text{-}exported : Identifier \times Identifier\text{-}set \rightarrow Environment \rightarrow \mathbb{B}$
 $is\text{-}exported (from, ids)\rho \triangleq$
274 $\text{let } \rho_{def} = associated\text{-}module (from)\rho \text{ in}$
276 $ids \subseteq identifiers\text{-}in\text{-}scope (\rho_{def})$

annotations The identifiers that are imported from a module must be exported from that module; thus they must be in the domain of the environment associated with the module from which they were exported.

6.1.6.4 Import Consistency

None of the identifiers in the identifier list shall be the same as the identifier of the module containing the simple import. Each identifier in the set of identifiers that is the closure of each of the identifiers in the identifier list shall not be otherwise introduced in the module containing the simple import.

functions

$consistent\text{-}imports : Import\text{-}lists \times Module\text{-}block \rightarrow Environment \rightarrow \mathbb{B}$
 $consistent\text{-}imports (imports, block)\rho \triangleq$
 $\text{let } mk\text{-}Module\text{-}block (decls, -) = block \text{ in}$
68 $\text{let } declcd = identifiers\text{-}declared\text{-}in (decls)\rho \text{ in}$
25 $\text{let } imported = imports\text{-}of (imports) \text{ in}$
29 $\text{let } closure = \bigcup \{x\text{-}closure (id)\rho \mid id \in imported\} \text{ in}$
 $closure \cap declcd = \{\}$

annotations Construct a set containing the identifiers which are declared in the declarations of a module block and use this set to check the consistency of each import list of the import lists.

The set of identifiers that are explicitly and implicitly imported must not be redeclared in the block that imports those identifiers. The implicitly imported identifiers are given by the closures of all the explicitly imported identifiers.

6.1.6.5 Closure

The closure of an identifier defines the set of identifiers that is implicitly imported or exported if that identifier is imported or exported.

Static Semantics

The closure of an identifier shall be a set consisting of that identifier, together with the closure of the type if the identifier is associated with a type.

The closure of a type shall be the empty set, except in the case that the type is an enumerated type, when the closure shall be the identifiers associated with the values of the enumerated type.

NOTES

1 Normally the closure of an identifier is a singleton set containing that identifier; for an identifier denoting an enumerated type the closure set includes the enumerated constants of the type as well as the identifier denoting the type itself.

2 Thus if an identifier denoting with an enumerated type is imported (exported), then the occurrence of that identifier in the import list (export list) is an abbreviation for the occurrence of that identifier, together with all of the identifiers associated with the constant values of the enumerated type.

Language Clarification

If an identifier associated with a module that contains an unqualified export list is imported (exported), then the identifiers occurring in the export list of the module shall not be automatically imported (exported).

functions

$x\text{-closure} : Identifier \rightarrow Environment \rightarrow Identifier\text{-set}$

```
276  $x\text{-closure}(id)\rho \triangleq$   
    let  $object = \text{access-environment}(id)\rho$  in  
    if  $is\text{-Typed}(object)$   
29    then  $\{id\} \cup x\text{-closure-typed}(object)\rho$   
    else  $\{id\}$ ;
```

$x\text{-closure-typed} : Typed \rightarrow Environment \rightarrow Identifier\text{-set}$

```
271  $x\text{-closure-typed}(typed)\rho \triangleq$   
    let  $struc = \text{structure-of}(typed)\rho$  in  
    if  $is\text{-Enumerated-structure}(struc)$   
    then let  $mk\text{-Enumerated-structure}(values) = struc$  in  
        elems  $values$   
    else  $\{ \}$ 
```

annotations The closure of a type is the empty set, except in the case that the type is an enumerated type, when it is the set of identifiers associated with the values of the enumerated type.

6.1.7 Export Lists

An export list defines the objects which are exported from a module by listing the identifiers denoting each of the objects. Identifiers which are defined by the closure of the exported identifiers are implicitly exported.

Concrete Syntax

export list = unqualified export | qualified export ;

Abstract Syntax

types

$$Export\text{-}list = Unqualified\text{-}export \mid Qualified\text{-}export$$

Declaration Semantics

functions

$$d\text{-}export\text{-}list : Export\text{-}list \rightarrow Environment \rightarrow Environment$$

$$d\text{-}export\text{-}list (export) \rho \triangleq$$

$$\begin{array}{ll} 30 & (is\text{-}Unqualified\text{-}export(export) \rightarrow d\text{-}unqualified\text{-}export(export) \rho, \\ 31 & is\text{-}Qualified\text{-}export(export) \rightarrow d\text{-}qualified\text{-}export(export) \rho) \end{array}$$

Static Semantics

functions

$$wf\text{-}export\text{-}list : Export\text{-}list \times Environment \rightarrow \mathbb{B}$$

$$wf\text{-}export\text{-}list (export) \rho \triangleq$$

$$\begin{array}{ll} 31 & (is\text{-}Unqualified\text{-}export(export) \rightarrow wf\text{-}unqualified\text{-}export(export) \rho, \\ 32 & is\text{-}Qualified\text{-}export(export) \rightarrow wf\text{-}qualified\text{-}export(export) \rho) \end{array}$$

6.1.7.1 Unqualified Exports

An unqualified export only appears within a local module. It defines the identifiers that can be used in the immediately surrounding module without needing to be qualified by the identifier of the module containing the unqualified export.

Concrete Syntax

unqualified export = "EXPORT", identifier list, ";" ;

Abstract Syntax

types

$$Unqualified\text{-}export :: ids : Identifier\text{-}set$$

The number of elements in the set of identifiers of the abstract representation of an unqualified export shall be equal to the number of identifiers in the identifier list of the concrete syntax.

NOTE — Thus the identifier list of the unqualified export clause contains distinct identifiers.

Declaration Semantics

functions

$$d\text{-}unqualified\text{-}export : Unqualified\text{-}export \rightarrow Environment \rightarrow Unqualified\text{-}environment$$

$$d\text{-}unqualified\text{-}export (mk\text{-}Unqualified\text{-}export (ids)) \rho \triangleq$$

$$\begin{array}{ll} 29 & let\ closure = \bigcup \{x\text{-}closure(id) \rho \mid id \in ids\} \text{ in} \\ 277 & restrict\text{-}environment(closure) \rho \end{array}$$

annotations Construct a new environment from the environment of the module containing the unqualified export by restricting it to the identifiers occurring in the identifier list together with any implicitly exported identifiers which are defined by the closure of the identifiers in the identifier list.

The meaning of an unqualified export is just the environment for the module containing the export. Note that visibility rules defined by import/export have been checked by the correctness conditions, so there is no need to restrict the environment by the export list.

Static Semantics

Any identifiers that are exported, together with any implicit exports that are defined by the closure of the exported identifiers, shall not be otherwise introduced in the surrounding environment of the module containing the unqualified export.

functions

$wf\text{-}unqualified\text{-}export : Unqualified\text{-}export \rightarrow Environment \rightarrow \mathbb{B}$

$wf\text{-}unqualified\text{-}export (mk\text{-}Unqualified\text{-}export (ids))\rho \triangleq$

$ids \cap identifiers\text{-}in\text{-}scope (\rho) = \{ \}$

Concrete Syntax

qualified export = "EXPORT", "QUALIFIED", identifier list, ";" ;

Abstract Syntax

types

$Qualified\text{-}export :: ids : Identifier\text{-}set$

The number of elements in the set of identifiers of the abstract representation of an qualified export shall be equal to the number of identifiers in the identifier list of the concrete syntax.

NOTE — Thus the identifier list of the qualified export clause contains distinct identifiers.

Declaration Semantics

functions

$d\text{-}qualified\text{-}export : Qualified\text{-}export \rightarrow Environment \rightarrow Module\text{-}environment$

$d\text{-}qualified\text{-}export (mk\text{-}Qualified\text{-}export (ids))\rho \triangleq$

$let\ closure = \bigcup \{x\text{-}closure (id)\rho \mid id \in ids\} in$

$restrict\text{-}environment (closure)\rho$

annotations This operation constructs an environment from the environment of the module containing the qualified export by restricting it to the identifiers occurring in the identifier list of the qualified export, together with any implicitly exported identifiers which are defined by the closure of the identifiers in the identifier list.

The meaning of an unqualified export is just the environment for the module containing the export. Note that visibility rules defined by import/export have been checked by the correctness conditions, so there is no need to restrict the environment by the export list.

Static Semantics

functions

$$\begin{aligned} wf\text{-qualified-export} &: \text{Qualified-export} \rightarrow \text{Environment} \rightarrow \mathbb{B} \\ wf\text{-qualified-export}(mk\text{-Qualified-export}(ids))\rho &\triangleq \\ \text{true} \end{aligned}$$

6.1.8 The Program Module and Module Pairs

For a particular program, each definition module shall have a single corresponding implementation module with the same identifier, and each implementation module shall have a single corresponding definition module with the same identifier. The identifier of the program module shall not be the same as the identifier of any definition module/implementation module pair which is part of the program.

For each definition module, implementation module pair the following shall be true.

- Except for opaque types and procedure declarations, no identifier defined in the definition module component of a module pair shall be re-declared in the associated implementation module component.
- Each opaque type defined in the definition module shall be declared in the associated implementation module as a pointer type or an opaque type.

Language Clarification

An opaque type shall not be redeclared as a scalar type.

NOTE — An opaque type may be redeclared as an address type, since an address type is also a pointer type.

Each procedure heading in the definition module shall have a matching procedure declaration in the associated implementation module. The formal parameter list of the procedure heading of the definition module shall match the formal parameter list of the procedure heading of the procedure declaration of the implementation module.

The formal parameter lists of procedure headings shall match if the formal types of each of the value parameter specification or variable parameter specification components of corresponding formal parameters are the same.

functions

$$\begin{aligned} wf\text{-modules} &: \text{Program-module} \times \text{Definition-modules} \times \text{Implementation-modules} \rightarrow \text{Environment} \rightarrow \mathbb{B} \\ wf\text{-modules}(pmod, dmods, imods)\rho &\triangleq \\ (\forall dmod \in dmods \cdot dmod.name \neq pmod.name \wedge \\ &\quad \exists! imod \in imods \cdot dmod.name = imod.name \wedge \\ &\quad wf\text{-module-pair}(dmod, imod)\rho) \wedge \\ 32 \quad &(\forall imod \in imods \cdot \exists! dmod \in dmods \cdot imod.name = dmod.name); \end{aligned}$$

$$wf\text{-module-pair} : \text{Definition-module} \times \text{Implementation-module} \rightarrow \text{Environment} \rightarrow \mathbb{B}$$

$$\begin{aligned} wf\text{-module-pair}(dmod, imod)\rho &\triangleq \\ 33 \quad &\text{let } \rho_{def} = \text{definition-environment}(dmod)\rho \text{ in} \\ 33 \quad &\text{let } \rho_{res} = \text{residual-environment}(dmod)\rho_{def} \text{ in} \\ 33 \quad &\text{let } \rho_{imp} = \text{imported-environment}(imod)\rho \text{ in} \\ 33 \quad &\text{let } \rho_{mod} = \text{module-environment}(imod)\rho \text{ in} \\ 33 \quad &no\text{-imports-from-definition-module}(\rho_{def}, \rho_{imp}) \wedge \\ 33 \quad &no\text{-illegal-redeclarations}(\rho_{res}, \rho_{mod}) \wedge \\ 34 \quad &opaque\text{-types-declared}(\rho_{def}, \rho_{mod}) \wedge \\ 34 \quad &procedures\text{-declared}(\rho_{def}, \rho_{mod}) \end{aligned}$$

```

annotations          Construct environments for the definitions and declarations components of an definition module
                        implementation module pair. Check that there are no redeclarations of any of the identifiers
                        defined in the definition module, except for opaque types and procedures. Opaque types and
                        procedures must be redeclared in the implementation module.
;

definition-environment : Definition-module  $\rightarrow$  Environment  $\rightarrow$  Environment
definition-environment (dmod) $\rho \triangleq$ 
  let mk-Definition-module (name, -, -) = dmod in
274   associated-module (name) $\rho$ 
annotations          Construct an environment for a definition module.
;

residual-environment : Definition-module  $\rightarrow$  Environment  $\rightarrow$  Environment
residual-environment (mk-Definition-module (-, -, -)) $\rho_{def} \triangleq$ 
  let mk-Definition-module (name, -, -) = dmod in
273   let dprocs = procedures-of ( $\rho_{def}$ ) in
288   let dopaques = opaque-types-of ( $\rho_{def}$ ) in
277   restrict-environment (dom dprocs  $\cup$  dopaques) $\rho_{def}$ 
annotations          Construct an environment from a definition module and remove any components which are
                        derived from declarations of procedure headings or opaque types.
;

imported-environment : Implementation-module  $\rightarrow$  Environment  $\rightarrow$  Environment
imported-environment (imod) $\rho \triangleq$ 
  let mk-Implementation-module (-, imports, -, -) = imod in
24   d-import-lists (imports) $\rho$ 
annotations          Construct an environment for any imported objects.
;

module-environment : Implementation-module  $\rightarrow$  Environment  $\rightarrow$  Environment
module-environment (imod) $\rho \triangleq$ 
  let mk-Implementation-module (name, imports, block, -) = imod in
274   let  $\rho_{def}$  = associated-module (name) $\rho$  in
24   let  $\rho_{import}$  = d-import-lists (imports) $\rho$  in
276   let  $\rho_{module}$  = merge-environments ( $\{\rho_{import}, \rho_{def}\}$ ) in
  let mk-Module-block (decls, -) = block in
46   d-declarations (decls) $\rho_{module}$ 
annotations          Construct an environment for an implementation module.
;

no-imports-from-definition-module : Environment  $\times$  Environment  $\rightarrow \mathbb{B}$ 
no-imports-from-definition-module ( $\rho_{def}, \rho_{imp}$ )  $\triangleq$ 
276   let defed = identifiers-in-scope ( $\rho_{def}$ ) in
276   let imported = identifiers-in-scope ( $\rho_{imp}$ ) in
  imported  $\cap$  defed =  $\{\}$ 
annotations          Construct a set of identifiers that contains the identifiers defined in the definitions component
                        of a definition module and construct a set containing the identifiers contained in the import
                        lists component of an implementation module. The result is true if and only if the intersection
                        of these two sets is empty.
;

no-illegal-redeclarations : Environment  $\times$  Environment  $\rightarrow \mathbb{B}$ 
no-illegal-redeclarations ( $\rho_{res}, \rho_{mod}$ )  $\triangleq$ 
276   identifiers-in-scope ( $\rho_{res}$ )  $\cap$  identifiers-in-scope ( $\rho_{mod}$ ) =  $\{\}$ 

```

annotations	Only those identifiers that are associated with opaque types and procedure headings defined in the definition module (i.e. those identifiers in the domain of the definition module environment associated with opaque types and procedures) may be redeclared in the declarations component of the associated implementation module.
;	
	$opaque\text{-}types\text{-}declared : Environment \times Environment \rightarrow \mathbb{B}$
	$opaque\text{-}types\text{-}declared (\rho_{def}, \rho_{mod}) \triangleq$
288	let $dopaques = opaque\text{-}types\text{-}of (\rho_{def})$ in
279	$\forall id \in dopaques \cdot is\text{-}pointer\text{-}type(id)\rho_{mod} \vee is\text{-}opaque\text{-}type(id)\rho_{mod}$
annotations	All of the opaque types defined in a definition module, and thus in the environment of the definition module, must be redeclared as a pointer type (which includes the address type) or an opaque type in the associated implementation module.
;	
	$procedures\text{-}declared : Environment \times Environment \rightarrow \mathbb{B}$
	$procedures\text{-}declared (\rho_{def}, \rho_{mod}) \triangleq$
273	let $dprocs = procedures\text{-}of (\rho_{def})$ in
273	let $iprocs = procedures\text{-}of (\rho_{mod})$ in
	$\forall id \in \text{dom } dprocs \cdot dprocs(id) = iprocs(id)$
annotations	Extract the procedure headings from the definition module environment and the implementation module environment and check that they are identical.

Language Clarification

The definition of a procedure in a definition module and the associated declaration of that procedure in the implementation module must match in the sense that corresponding parameters must have the same type. The identifiers used to denote the parameters need not be the same, and the identifiers used to denote the types need not be the same.

6.1.9 Module Compilation Order

The compilation order of the compilation units of a program defines the order in which they may be compiled. In particular, the objects defined in a definition module are available in subsequent compilations.

Static Semantics

The compilation order of the program module and each of the definition and implementation modules shall be such that each definition module shall be compiled before the corresponding implementation module, and before any program module or other definition or implementation module that imports any object it defines.

NOTES

- 1 A program, definition, or implementation module may only be elaborated if any objects it imports have been defined by the elaboration of the definition modules that defines them. This restriction is satisfied if no module indirectly imports itself.
- 2 A module S directly imports a module T if T occurs in the import list of S .
- 3 A module S indirectly imports another module U if module S directly imports U , or if module S directly imports a module T and T indirectly imports module U .

types

$$Module\text{-}order = Compilation\text{-}unit^*$$

functions

$compilation-orders : Program-module \times Definition-modules \times Implementation-modules \rightarrow Module-order-set$

$compilation-orders (pmod, dmods, imods) \triangleq$
 $\{order : Module-order \mid$

35 $is-complete-order (order, pmod, dmods, imods) \wedge$
 35 $is-correct-order (order, pmod, dmods, imods)\}$

annotations The result is a set of possible module compilation orders. For a compilation order to be valid, it must contain the identifiers of all of the compilation units that make up the program; and the order must be such that definition modules that associate identifiers with objects must be elaborated before the identifiers are imported into another module.

Auxiliary Functions

functions

$is-complete-order : Module-order \times Program-module \times Definition-modules \times Implementation-modules \rightarrow \mathbb{B}$

$is-complete-order (order, pmod, dmods, imods) \triangleq$
 $rng\ order = \{pmod\} \cup dmods \cup imods \wedge$
 $\forall i, j \in inds\ order \cdot i = j \vee order(i) \neq order(j)$

annotations The sequence of modules that make up a module compilation order contain the program module, and all of the definition modules and the associated implementation modules that constitute the program. The program module, each definition module and each implementation module may occur only once in the module order.

;

$is-correct-order : Module-order \times Program-module \times Definition-modules \times Implementation-modules \rightarrow \mathbb{B}$

$is-correct-order (order, pmod, dmods, imods) \triangleq$
 $\forall i \in inds\ order \cdot$

35 $let\ cdefs = compiled-definition-modules (order(1, \dots, i-1))\ in$
 $cases\ order(i) :$
 25 $mk-Program-module (-, imports, -, -) \rightarrow let\ names = directly-imports (imports)\ in$
 $elems\ names \subseteq cdefs,$
 25 $mk-Definition-module (-, imports, -) \rightarrow let\ names = directly-imports (imports)\ in$
 $elems\ names \subseteq cdefs,$
 $mk-Implementation-module (name, imports, -, -) \rightarrow name \in cdefs \wedge$
 25 $let\ names = directly-imports (imports)\ in$
 $elems\ names \subseteq cdefs$
 end

annotations Check that the sequence of modules in the module compilation order occur in a correct order, i.e. a definition module must be compiled before it, or any object defined by it, can be imported by another module and the definition module must be compiled before its associated implementation module is compiled.

;

$compiled-definition-modules : Module-order \rightarrow Identifier-set$

$compiled-definition-modules (order) \triangleq$
 $\{id \in Identifier \mid mk-Definition-module (id, -, -) \in elems\ order\}$

annotations Construct the set of the identifiers of the definition modules that appear in a sequence of program, definition, and implementation modules.

6.1.9.1 Module Dependencies

NOTE — The definition modules that must be compiled before a compilation unit are the definition modules the compilation unit depends; these are defined by the function below. If a definition module of a program is modified, then all the compilation units that depend on that definition module must also be recompiled after the definition module itself is recompiled.

functions

$dependent-modules : Compilation-unit \times Program-module \times Definition-modules \times Implementation-modules \rightarrow Module-order-set$

```
35  dependent-modules (module, pmod, dmods, imods)  $\triangleq$ 
    let orders = compilation-orders (pmod, dmods, imods) in
77  let depend = {x : Module-order |  $\exists order \in orders \cdot is-suffix ([module] \curvearrowright x, order)$ } in
    let size = mins {len x | x  $\in$  depend} in
    {dep  $\in$  depends | len dep = size}
```

annotations The result of this operation is a set of compilation orders for the compilation units that must be recompiled if the compilation unit that is the first parameter is recompiled.

6.1.10 Module Initialisation Order

6.1.10.1 Implementation Module Initialisation

The initialisation order of the implementation modules of a program defines in what order the statement sequence are to be executed.

Dynamic Semantics

The program shall be initialised by processing the import lists of the program module. The import lists of a module shall be processed by processing each import list in order of occurrence. If an import list is an unqualified import, the import list shall be processed by initialising the module pair (consisting of a definition module and implementation module, both with the same identifier) whose module identifier appears in the import list. If an import list is a simple import, the import list shall be processed by initialising in turn each module pair whose identifier appears in the identifier list of the import list. A module pair shall be initialised by executing the module pair, but this shall not be done if the module pair has already been initialised, or if the module pair is currently being executed.

A module pair shall be executed by first processing the import lists contained in the definition module, then by processing the import lists of the implementation module, and finally by executing the implementation module block.

An implementation module block shall be executed by initialising any local modules, followed by executing the module body. These activities shall take place in the context of the part of the surrounding dynamic environment that applies to the implementation module.

If the program module is protected, the entry procedure of the program module shall be called after the initialisation of the separate modules on which the program depends; the exit procedure shall be called on completion of the statement part of the program.

If any implementation module is protected, the entry procedure of the implementation module shall be called after the initialisation of separate modules whose initialisation is required to be completed before the initialisation of the module; the exit procedure shall be called on completion of the statement part of the module.

NOTE — Since the implementation modules of a program are initialised in a defined order; it is an exception if a variable is referenced before it has a value assigned to it.

CHANGE — The second edition of *Programming in Modula-2* does not deal with the effects of import lists on the order of the initialisation of modules. The third edition states how import lists appearing in the implementation modules are used in the

definition of the initialisation order. The rule for import lists in definition modules has been extended to cover the program module and definition modules.

This International Standard defines an order for the initialisation of modules when there is a circular reference in the import lists. The third edition of *Programming in Modula-2* states that the order of initialisation is undefined in such a case.

operations

initialise-modules :

$Identifier^* \times Definition-modules \times Implementation-modules \times Identifier-set \rightarrow Environment \xrightarrow{o} ()$

initialise-modules (*names*, *dmods*, *imods*, *done*) $\rho \triangleq$

if *names* = []

then skip

37 else def *initialised* = *initialise-module*(hd *names*, *dmods*, *imods*, *done*) ρ ;

37 *initialise-modules*(tl *names*, *dmods*, *imods*, *initialised*) ρ

annotations

Modules are initialised in the order defined by the value *names*, which is a sequence of identifiers. If the sequence of identifiers of modules to be initialised is empty, then there is no initialisation to do and thus there is no change to the state. If the sequence is not empty, the modules are initialised in the order described by the ordering of the identifiers of the modules in the sequence. The values *done* and *initialised* are sets of identifiers which denote those modules that have already been initialised. The function *initialise-module* returns a set of identifiers denoting those modules that have been initialised during its evaluation.

;

initialise-module :

$Identifier \times Definition-modules \times Implementation-modules \times Identifier-set \rightarrow Environment \xrightarrow{o} Identifier-set$

initialise-module (*name*, *dmods*, *imods*, *done*) $\rho \triangleq$

if *name* \notin *done*

38 then (let *imported* = *imported-by-definition-module* (*name*, *dmods*) \curvearrowright

38 *imported-by-implementation-module* (*name*, *imods*) in

37 def *initialised* = *initialise-imports*(*imported*, *dmods*, *imods*, *done* \cup {*name*}) ρ ;

40 *optional-call-of-domain-entry-procedure*() ρ ;

let *imod* \in *imods* be st *imod.name* = *name* in

38 *initialise-implementation-module*(*imod*) ρ ;

41 *optional-call-of-domain-exit-procedure*() ρ ;

return *initialised*)

else return *done*

annotations

If the module definition/implementation pair has not already been initialised, a sequence of module identifiers is constructed from the import lists of the definition and implementation module, then each module in this sequence is initialised and then the implementation module of the module pair is initialised. The result is the set of identifiers denoting those modules that have been initialised during its evaluation.

The entry procedure of a protected implementation module is called after the initialisation of separate modules whose initialisation is to be completed before the initialisation of the module; the exit procedure is called on completion of the statement part of the module.

;

initialise-imports :

$Identifier^* \times Definition-modules \times Implementation-modules \times Identifier-set \rightarrow Environment \xrightarrow{o} Identifier-set$

initialise-imports (*names*, *dmods*, *imods*, *done*) $\rho \triangleq$

if *names* = []

then return *done*

37 else def *initialised* = *initialise-module*(hd *names*, *dmods*, *imods*, *done*) ρ ;

37 *initialise-imports*(tl *names*, *dmods*, *imods*, *initialised*) ρ

annotations Modules are initialised in the order defined by the value *names*, which is a sequence of identifiers. The values *done* and *initialised* are sets of identifiers which denote those modules that have already been initialised. The function *initialise-module* returns a set of identifiers denoting those modules that have been initialised during its evaluation. The result is a set containing the identifiers of those modules that have either been initialised, or are in the process of being initialised.

;

initialise-implementation-module : *Implementation-module* \rightarrow *Environment* \xrightarrow{o} ()

initialise-implementation-module (*imod*) $\rho \triangleq$
 let *mk-Implementation-module* (*name*, -, *block*, -) = *imod* in
 274 let $\rho_{mod} = \text{associated-module}(\text{name})\rho$ in
 38 *execute-module-block*(*block*) ρ_{mod}

annotations An implementation module is initialised by executing the block component of that module in the appropriate environment for that module.

;

execute-module-block : *Module-block* \rightarrow *Environment* \xrightarrow{o} ()

execute-module-block (*block*) $\rho \triangleq$
 (let *mk-Module-block* (*decls*, *actions*) = *block* in
 39 *initialise-local-modules*(*decls*) ρ ;
 ?? def $\rho_{cont} = c\text{-fixe}(\{\text{RETURN} \mapsto \text{skip}\})\rho$;
 ?? *m-statement-list*(*actions*) ρ_{cont})

annotations A module block is executed by first initialising any local modules contained in the declarations of the block, and then executing the sequence of statements that compose the body of the block. The execution of the module body may be terminated by a return statement.

Auxiliary Functions

functions

imported-by-definition-module : *Identifier* \times *Definition-modules* \rightarrow *Identifier*^{*}

imported-by-definition-module (*nm*, *dmods*) \triangleq
 let *mk-Definition-module* (*name*, *imports*, -) \in *dmods* be st *name* = *nm* in
 25 *directly-imports* (*imports*)

annotations Construct a sequence of identifiers that contains the identifiers of those modules that are directly imported by the definition module.

;

imported-by-implementation-module : *Identifier* \times *Implementation-modules* \rightarrow *Identifier*^{*}

imported-by-implementation-module (*nm*, *imods*) \triangleq
 let *mk-Implementation-module* (*name*, *imports*, -, -) \in *imods* be st *name* = *nm* in
 25 *directly-imports* (*imports*)

annotations Construct a sequence of identifiers that contains the identifiers of those modules that are directly imported by the implementation module.

;

protection-expression-of : *Identifier* \times *Implementation-modules* \rightarrow *Expression*

protection-expression-of (*nm*, *imods*) \triangleq
 let *mk-Implementation-module* (*name*, -, -, *protection*) \in *imods* be st *name* = *nm* in
protection

6.1.10.2 Local Module Initialisation

The initialisation order of local modules of a block defines in what order they are to be executed.

Dynamic Semantics

Any local modules contained in the declarations of a module block shall be initialised in the textual order in which they occur. A local module shall be initialised by first initialising any modules declared in the declarations of the local module, and then by executing the statement sequence of the local module. The execution of a simple return statement contained in the statement sequence shall cause the execution of the statement sequence to terminate.

NOTE — If an implementation chooses the version of the well-ordering predicate (section 6.2.2.1) that allows use of an identifier in a statement before its declaration, then, in the body of a local module, it is possible to use a variable which is declared later on in the program text. Such a use may give rise to an exception because, since the initialisation code of the module in which the variable is declared has not yet been obeyed, the value of the variable may be undefined.

This situation does not usually arise with variables which are imported from separate modules because the rules for initialisation of separate modules guarantee that the initialisation code of the module which exported the variable is obeyed before that of the module which imports it.

operations

```

initialise-local-modules : Declarations → Environment  $\xrightarrow{o}$  ()
initialise-local-modules (decls) $\rho \triangleq$ 
  if decls = []
  then skip
  else let decl = hd decls in
    if is-Module-declaration(decl)
39   then initialise-a-local-module(decl)  $\rho$ 
    else skip;
39   initialise-local-modules(tl decls)  $\rho$ 

```

annotations The local modules contained in the declarations of a module block are initialised in the order in which they occur. This is done by examining the declarations in order, and if the declaration is for a module, that module body is executed.

Auxiliary Functions

operations

```

initialise-a-local-module : Module-declaration → Environment  $\xrightarrow{o}$  ()
initialise-a-local-module (mod) $\rho \triangleq$ 
  let mk-Module-declaration (name, -, -, block, -) = mod in
274 let  $\rho_{mod}$  = associated-module (name) $\rho$  in
40  optional-call-of-domain-entry-procedure()  $\rho$ ;
??  def  $\rho_{cont}$  = c-tixe({RETURN ↦ skip})  $\rho_{mod}$ ;
38  execute-module-block(block)  $\rho_{cont}$ ;
41  optional-call-of-domain-exit-procedure()  $\rho$ 

```

annotations The environment for the execution of the block of a local module is the environment defined by its declarations. The block of a local module is executed in this environment.

6.1.11 Protected Modules

A directly protected module is a module with a protection expression in the heading of that module.

A program module, implementation module, or local module may specify, by including a protection expression in its heading, that the execution of enclosed statement sequences is to be protected. Protection is provided by surrounding the externally accessible procedure and module bodies of that module by calls to access control procedures known in the environment surrounding the module, supplying the value of the protection expression as the actual parameter.

The domain of protection of a directly protected module covers the statement part of that module and the statement parts of all procedures and modules whose declarations are nested within the protected module, except for those which are in the domain of protection of a nested module which is itself directly protected.

A directly protected procedure is a proper procedure or a function procedure, declared within the domain of protection of a protected module, which is exported and thus may be called from outside the domain of protection of that module.

NOTE — Procedures may also be called from outside the domain of protection by being imported into a nested protected module.

Modules and procedures in the domain of protection which are not directly protected are indirectly protected since they may only be invoked by the directly protected module or by directly protected procedures.

The procedure identified by **ENTER** in the environment of the statement part of a directly protected module or procedure is the protection domain entry procedure of that module or procedure.

The procedure identified by **LEAVE** in the environment of the statement part of a directly protected module or procedure is the protection domain exit procedure of that module or procedure.

The entry and exit procedures are the access control procedures of a directly protected module or procedure.

Static Semantics

A protection expression specified in the heading of a directly protected module shall be a constant expression in the environment surrounding that module. The surrounding environment of a protected program module or protected implementation module is the required pervasive identifiers.

CHANGE — In *Programming in Modula-2*, identifiers that are in scope in a protection expression of a separate module are not explicitly discussed, and the only examples given of protection expressions are single literal values.

It is necessary to liberalise this, at least to the pervasive identifiers, in order to handle the definition of protection (for which, in general, literal values may vary between implementations).

In the surrounding environment of directly protected modules, the identifiers **ENTER** and **LEAVE** shall be procedure designators, or procedure constant designators, of a type with one value parameter which is assignment compatible with the type of the protection expression.

ENTER and **LEAVE** shall not be designators of procedure variables in directly protected modules.

It shall be an error for an access procedure of a directly protected procedure or module to be a directly protected procedure with itself as an access procedure.

Dynamic Semantics

Entering a domain of protection shall generate a call to the entry procedure of the directly protected module or procedure, supplying as an actual parameter the value of the protection expression in the environment of the module heading.

operations

```

optional-call-of-domain-entry-procedure : Environment  $\xrightarrow{o}$  ()
optional-call-of-domain-entry-procedure ()  $\rho \triangleq$ 
275   let  $pdom = protection\_domain(\rho)$  in
      if  $pdom \neq nil$ 
      then let  $mk\_Protection\_domain(enter, -, val) = pdom$  in
           return  $enter(val)$ 
      else skip

```

annotations If the protection expression is present, call the domain entry procedure for the module.

Leaving a domain of protection shall generate a call to the exit procedure of the directly protected module or procedure, supplying as actual parameter the value of the protection expression in the environment of the module heading.

optional-call-of-domain-exit-procedure : *Environment* \xrightarrow{o} ()

optional-call-of-domain-exit-procedure () $\rho \triangleq$

```

275  let pdom = protection-domain ( $\rho$ ) in
    if pdom  $\neq$  NIL
    then let mk-Protection-domain ( $-, leave, val$ ) = pdom in
        return leave(val)
    else skip

```

annotations If the protection expression is present, call the domain exit procedure for the module.

Auxiliary Functions

functions

is-valid-access-control : *Expression* \rightarrow *Environment* $\rightarrow \mathbb{B}$

is-valid-access-control (*expr*) $\rho \triangleq$

```

    if expr  $\neq$  nil
214   then is-constant-expression (expr)  $\rho \wedge$ 
152       let atype = t-expression (expr)  $\rho$  in
276       let enter = access-environment (ENTER)  $\rho$  in
41       is-valid-access-control-type (enter, atype)  $\rho \wedge$ 
276       let leave = access-environment (LEAVE)  $\rho$  in
41       is-valid-access-control-type (leave, atype)  $\rho$ 
    else true;

```

is-valid-access-control-type : *Expression-typed* \times *Expression-typed* \rightarrow *Environment* $\rightarrow \mathbb{B}$

is-valid-access-control-type (*control*, *atype*) $\rho \triangleq$

```

    cases control :
41      mk-Constant-value (type,  $-$ )  $\rightarrow$  is-access-parameter (type, atype)  $\rho$ ,
41      mk-Procedure-const (proc,  $-$ )  $\rightarrow$  is-access-parameter (proc, atype)  $\rho$ ,
      STANDARD-PROPER-PROCEDURE  $\rightarrow$  atype = PROTECTION-TYPE
    end

```

annotations Check that any procedure calls to access control procedures contained in a sequence of statements are valid.

;

is-access-parameter : *Expression-typed* \times *Expression-typed* \rightarrow *Environment* $\rightarrow \mathbb{B}$

is-access-parameter (*type*, *atype*) $\rho \triangleq$

```

279   is-proper-procedure-type (type)  $\rho \wedge$ 
285   let parms = parameter-types-of (type)  $\rho$  in
220   is-parameters-match (parms, atype)  $\rho$ 

```

operations

evaluate-protection : *Expression* \rightarrow *Environment* \xrightarrow{o} [*Protection-domain*]

evaluate-protection (*protection*) $\rho \triangleq$

```

    if protection = nil
    then nil
??   else let val = m-expression (protection)  $\rho$  in
184       def eden = m-entire-value (mk-Entire-value (ENTER))  $\rho$ ;
184       def lden = m-entire-value (mk-Entire-value (LEAVE))  $\rho$ ;

```

```
return mk-Protection-domain (eden, lden, val)
```

6.2 Definitions and Declarations

Definitions and declarations serve to introduce the identifiers of a module or procedure. Definitions appear in definition modules; declarations appear in program, implementation, and local modules and in procedures.

6.2.1 Definitions

Definitions serve to introduce the identifiers of a definition module, and to specify certain permanent properties of the object which each of these identifiers denotes. The properties include whether the object is a constant and its associated value, a type and possibly its structure, a variable and its type, or a procedure and its type. The identifier is used to refer to the associated object, but only in those parts of the program which are within the scope of the definition.

Concrete Syntax

definitions = { definition } ;

definition =
 "CONST", { constant declaration, ";" } |
 "TYPE", { type definition, ";" } |
 "VAR", { variable declaration, ";" } |
 procedure heading, ";" ;

Abstract Syntax

Definitions = *Definition**

Definition = *Constant-declaration* | *Type-definition* | *Variable-declaration* | *Procedure-heading*

Procedure-heading = *Proper-procedure-heading* | *Function-procedure-heading*

Declaration Semantics

functions

$d\text{-definitions} : \text{Definitions} \rightarrow \text{Environment} \rightarrow \text{Environment}$

$d\text{-definitions}(defs)\rho \triangleq$

if $defs = []$

then ρ_{empty}

43 else let $\rho_{def} = d\text{-definition}(\text{hd } defs)\rho$ in $d\text{-definitions}(\text{tl } defs)\rho_{def}$

annotations Each definition is elaborated in an environment that has been constructed from the surrounding environment and the other preceding definitions.

;

$d\text{-definition} : \text{Definition} \rightarrow \text{Environment} \rightarrow \text{Environment}$

$d\text{-definition}(def)\rho \triangleq$

49 $(is\text{-Constant-declaration}(def) \rightarrow d\text{-constant-declaration}(def)\rho,$

45 $is\text{-Type-definition}(def) \rightarrow d\text{-type-definition}(def)\rho,$

51 $is\text{-Variable-declaration}(def) \rightarrow d\text{-variable-declaration}(def)\rho,$

56 $is\text{-Proper-procedure-heading}(def) \rightarrow d\text{-proper-procedure-heading}(def)\rho,$

61 $is\text{-Function-procedure-heading}(def) \rightarrow d\text{-function-procedure-heading}(def)\rho)$

Static Semantics

An identifier defined in a definition module shall only have one defining occurrence within the scope of the definition module.

If an identifier defined by a definition is used in another definition then the defining definition shall textually precede the using definition; except that a type may be used in a definition of a pointer type or a procedure type which textually precedes the definition of that type.

functions

```

wf-definitions : Definitions → Environment →  $\mathbb{B}$ 
wf-definitions (defs)  $\rho \triangleq$ 
67   is-disjoint-definitions (defs)  $\wedge$ 
67   is-complete-definitions (defs)  $\rho \wedge$ 
     $\forall i \in \text{inds } \text{defs} \cdot$ 
81d   let predefs = front (defs, i - 1) in
43   let  $\rho' = d\text{-definitions } (\text{predefs}) \rho$  in
44   wf-definition (defs(i))  $\rho'$ 

annotations      The identifiers defined in a definition must all be distinct. Any identifiers used to define an
                   identifier in a sequence of definitions must already be defined.
;

wf-definition : Definition → Environment →  $\mathbb{B}$ 
wf-definition (def)  $\rho \triangleq$ 
49   (is-Constant-declaration(def) → wf-constant-declaration (def)  $\rho$ ,
45   is-Type-definition(def) → wf-type-definition (def)  $\rho$ ,
51   is-Variable-declaration(def) → wf-variable-declaration (def)  $\rho$ ,
56   is-Proper-procedure-heading(def)  $\rho \rightarrow$  wf-proper-procedure-heading (def)  $\rho$ ,
61   is-Function-procedure-heading(def)  $\rho \rightarrow$  wf-function-procedure-heading (def)  $\rho$ )

```

Dynamic Semantics

operations

```

m-definitions : Definitions → Environment  $\xrightarrow{o}$  Environment
m-definitions (defs)  $\rho \triangleq$ 
    if defs = []
    then return  $\rho_{\text{empty}}$ 
44   else def  $\rho_{\text{def}} = m\text{-definition}(\text{hd } \text{defs}) \rho$ ;
44   m-definitions(tl defs)  $\rho_{\text{def}}$ 

annotations      Each definition is elaborated in an environment that has been constructed from the surrounding
                   environment and the definitions that have been elaborated so far.
;

m-definition : Definition → Environment  $\xrightarrow{o}$  Environment
m-definition (def)  $\rho \triangleq$ 
49   ( is-Constant-declaration(def) → m-constant-declaration(def)  $\rho$ ,
46   is-Type-definition(def) → m-type-definition(def)  $\rho$ ,
51   is-Variable-declaration(def) → m-variable-declaration(def)  $\rho$ ,
77   is-Proper-procedure-heading(def)  $\rho \rightarrow$  m-proper-procedure-heading(def)  $\rho$ ,
77   is-Function-procedure-heading(def)  $\rho \rightarrow$  m-function-procedure-heading(def)  $\rho$ )

```

6.2.1.1 Type Definitions

Type definitions serve to introduce an identifier that denotes a type and to specify that type and possibly its structure. If the structure is omitted, the type definition defines an opaque type.

Concrete Syntax

type definition = type declaration | opaque type definition ;

procedure heading = proper procedure heading | function procedure heading ;

opaque type definition = identifier ;

Abstract Syntax

Type-definition = *Type-declaration* | *Opaque-type-definition*

Opaque-type-definition :: *id* : *Identifier*

Declaration Semantics

functions

$d\text{-type-definition} : \text{Type-definition} \rightarrow \text{Environment} \rightarrow \text{Environment}$

$d\text{-type-definition}(def)\rho \triangleq$

50 $(is\text{-Type-declaration}(def) \rightarrow d\text{-type-declaration}(def)\rho,$
45 $is\text{-Opaque-type-definition}(def) \rightarrow d\text{-opaque-type-definition}(def)\rho);$

$d\text{-opaque-type-definition} : \text{Opaque-type-definition} \rightarrow \text{Environment} \rightarrow \text{Environment}$

$d\text{-opaque-type-definition}(mk\text{-Opaque-type-definition}(id))\rho \triangleq$

277 $\text{let } typenm = generate\text{-type-name}(\rho) \text{ in}$
271 $\text{let } \rho_{tenv} = overwrite\text{-struc-environment}(\{typenm \mapsto opaque\})\rho \text{ in}$
271 $overwrite\text{-type-environment}(\{id \mapsto typenm\})\rho_{tenv}$

annotations

A unique type name is generated for the opaque type, and the type identifier is associated with that type name; the environment is updated to record the new type and the structure of the type as being opaque.

Static Semantics

functions

$wf\text{-type-definition} : \text{Type-definition} \rightarrow \text{Environment} \rightarrow \mathbb{B}$

$wf\text{-type-definition}(def)\rho \triangleq$

50 $(is\text{-Type-declaration}(def) \rightarrow wf\text{-type-declaration}(def)\rho,$
45 $is\text{-Opaque-type-definition}(def) \rightarrow wf\text{-opaque-type-definition}(def)\rho);$

$wf\text{-opaque-type-definition} : \text{Opaque-type-definition} \rightarrow \text{Environment} \rightarrow \mathbb{B}$

$wf\text{-opaque-type-definition}(mk\text{-Opaque-type-definition}(-))\rho \triangleq$
true

Dynamic Semantics

operations

$$m\text{-type-definition} : \text{Type-definition} \rightarrow \text{Environment} \xrightarrow{o} \text{Environment}$$

50
$$m\text{-type-definition}(def)\rho \triangleq$$
$$\begin{array}{ll} (is\text{-Type-declaration}(def) & \rightarrow m\text{-type-declaration}(def)\rho, \\ is\text{-Opaque-type-definition}(def) & \rightarrow \text{skip}) \end{array}$$

annotations There are no dynamic semantics for opaque type definitions.

6.2.2 Declarations

Each identifier (except pervasive identifiers) occurring in a module or procedure must be introduced by a declaration, or be imported from some other module, or be exported from some nested local module. A declaration serves to introduce an identifier and to specify certain permanent properties of the object which the identifier denotes. These properties include whether the object is a constant and its associated value, a type and its structure, a variable and its type, a procedure and its associated block, or a local module. The identifier is used to refer to the associated object, but only in those parts of the program which are within the scope of the declaration.

Concrete Syntax

declarations = { declaration } ;

declaration =
"CONST", { constant declaration, ";" } |
"TYPE", { type declaration, ";" } |
"VAR", { variable declaration, ";" } |
procedure declaration, ";" |
module declaration, ";" ;

Abstract Syntax

$$Declarations = Declaration^*$$
$$Declaration = Constant\text{-}declaration \mid Type\text{-}declaration \mid Variable\text{-}declaration \\ \mid Procedure\text{-}declaration \mid Module\text{-}declaration$$

Declaration Semantics

functions

$$d\text{-declarations} : Declarations \rightarrow Environment \rightarrow Environment$$

$$d\text{-declarations}(decls)\rho \triangleq$$
$$\begin{array}{l} \text{if } decls = [] \\ \text{then } \rho_{empty} \\ 46 \quad \text{else let } \rho_{decl} = d\text{-declaration}(\text{hd } decls)\rho \text{ in } d\text{-declarations}(\text{tl } decls)\rho_{decl} \end{array}$$

annotations Each declaration is elaborated in an environment that has been constructed from the surrounding environment and the other preceding declarations.

;

$d\text{-declaration} : \text{Declaration} \rightarrow \text{Environment} \rightarrow \text{Environment}$

$d\text{-declaration}(decl)\rho \triangleq$

50 $(is\text{-Type-declaration}(decl) \rightarrow d\text{-type-declaration}(decl)\rho,$
 51 $is\text{-Variable-declaration}(decl) \rightarrow d\text{-variable-declaration}(decl)\rho,$
 77 $is\text{-Procedure-declaration}(decl) \rightarrow d\text{-procedure-declaration}(decl)\rho,$
 65 $is\text{-Module-declaration}(decl) \rightarrow d\text{-module-declaration}(decl)\rho)$

Static Semantics

The identifiers declared in a declaration shall have only one defining occurrence within the scope of the declaration.

If an identifier introduced by a declaration is used in another declaration then the defining declaration shall textually precede the using declaration; except that:

- A type can be used in a declaration of a pointer type or a procedure type which textually precedes the declaration of that type if the declarations are in the same block.
- If an identifier defined in a local module is exported, the scope of that identifier shall be the block which contains the module declaration.

functions

$wf\text{-declarations} : \text{Declarations} \rightarrow \text{Environment} \rightarrow \mathbb{B}$

$wf\text{-declarations}(decls)\rho \triangleq$

68 $is\text{-disjoint-declarations}(decls) \wedge$
 68 $is\text{-complete-declarations}(decls)\rho \wedge$
 $\forall i \in \text{inds } decls \cdot$
 51d $\text{let } predecls = \text{front}(decls, i - 1) \text{ in}$
 46 $\text{let } \rho' = d\text{-declarations}(predecls)\rho \text{ in } wf\text{-declaration}(decl(i))\rho'$

annotations The identifiers defined in a declaration must all be distinct. Any identifiers used to declare an identifier in a sequence of declarations must already be declared.

;

$wf\text{-module-declarations} : \text{Declarations} \rightarrow \text{Environment} \rightarrow \mathbb{B}$

$wf\text{-module-declarations}(decls)\rho \triangleq$

48 $wf\text{-imports}(decls)\rho \wedge$
 48 $wf\text{-ordered-declarations}(decls)\rho \wedge$
 $\forall i \in \text{inds } decls \cdot$
 46 $\text{let } \rho_{decl} = d\text{-declarations}(decls)\rho \text{ in } wf\text{-declaration}(decl(i))\rho_{decl}$

Dynamic Semantics

operations

$m\text{-declarations} : \text{Declarations} \rightarrow \text{Environment} \xrightarrow{o} \text{Environment}$

$m\text{-declarations}(decls)\rho \triangleq$

$\text{if } defs = []$
 $\text{then return } \rho_{empty}$
 77 $\text{else def } \rho_{decl} = m\text{-declaration}(\text{hd } decls) \rho;$
 47 $m\text{-declarations}(\text{tl } decls) \rho_{decl}$

annotations Each declaration is elaborated in an environment that has been constructed from the surrounding environment and the declarations that have been elaborated so far.

Auxiliary Definitions

functions

```

wf-imports : Declarations → Environment →  $\mathbb{B}$ 
wf-imports (decls) $\rho \triangleq$ 
46   let  $\rho_{decl} = d\text{-declarations}(decls)\rho$  in
     $\forall mk\text{-Module-declaration}(-, imports, -, -, -) \in \text{elems } decls \cdot$ 
25     wf-import-lists (imports) $\rho_{decl}$ ;

wf-declaration : Declaration → Environment →  $\mathbb{B}$ 
wf-declaration (decl) $\rho \triangleq$ 
  cases decl :
54   mk-Procedure-declaration () → wf-proper-procedure (decl) $\rho$ ,
59   mk-Function-procedure-declaration () → wf-function-procedure (decl) $\rho$ ,
66   mk-Module-declaration () → wf-module (decl) $\rho$ 
  end

```

6.2.2.1 Predicates for Declarations

NOTE — Implementations may choose either of the versions of the function *wf-ordered-declarations* defined here.

functions

```

wf-ordered-declarations : Declarations → Environment →  $\mathbb{B}$ 
wf-ordered-declarations (decls) $\rho \triangleq$ 
  true;

wf-ordered-declarations : Declarations → Environment →  $\mathbb{B}$ 
wf-ordered-declarations (decls) $\rho \triangleq$ 
   $\forall i \in \text{inds } decls \cdot$ 
std   let predecls = front (decls, i - 1) in
46   let  $\rho_{decl} = d\text{-declarations}(\text{predecls})\rho$  in
48   wf-declaration (decl(i)) $\rho_{decl}$ 

```

6.2.3 Constant Declarations

A constant declaration defines an identifier as a synonym for a value. Each occurrence of the identifier is equivalent to an explicit occurrence of the value itself at that point.

Concrete Syntax

constant declaration = identifier, "=", constant expression ;

NOTE — A constant expression is one that can be completely evaluated by a mere textual scan without executing the program - see section 6.7.7.

Abstract Syntax

Constant-declaration :: *id* : *Identifier*
 expr : *Expression*

Declaration Semantics

functions

$d\text{-constant-declaration} : \text{Constant-declaration} \rightarrow \text{Environment} \rightarrow \text{Environment}$

$d\text{-constant-declaration} (mk\text{-Constant-declaration} (id, expr))\rho \triangleq$

152 $\text{let } type = t\text{-expression} (expr)\rho \text{ in}$

218 $\text{let } value = \text{evaluate-constant-expression} (expr)\rho \text{ in}$

270 $\text{overwrite-const-environment} (\{id \mapsto mk\text{-Constant-value} (type, value)\})\rho$

annotations The constant expression is evaluated and the result is associated with the identifier and added to the environment.

Static Semantics

The expression component of a constant declaration shall be constant, that is, such that its value can be determined statically. If the expression component of a constant declaration contains any other identifiers, then these identifiers either shall have been introduced in a constant declaration, or shall denote a standard function which may be used in a constant expression (see 6.7.7).

functions

$wf\text{-constant-declaration} : \text{Constant-declaration} \rightarrow \text{Environment} \rightarrow \mathbb{B}$

$wf\text{-constant-declaration} (mk\text{-Constant-declaration} (id, expr))\rho \triangleq$

70 $id \notin \text{identifiers-used-in-expression} (expr) \wedge$

214 $\text{is-constant-expression} (expr)\rho \wedge$

152 $wf\text{-expression} (expr)\rho$

Dynamic Semantics

The elaboration of a constant declaration shall associate the identifier component of the declaration with an object; the object is the value obtained from evaluating the constant expression.

operations

$m\text{-constant-declaration} : \text{Constant-declaration} \rightarrow \text{Environment} \xrightarrow{o} \text{Environment}$

$m\text{-constant-declaration} (mk\text{-Constant-declaration} (id, expr))\rho \triangleq$

152 $\text{let } type = t\text{-expression} (expr)\rho \text{ in}$

153 $\text{def } value = m\text{-expression}(expr)\rho;$

270 $\text{return } \text{overwrite-const-environment} (\{id \mapsto mk\text{-Constant-value} (type, value)\})\rho$

6.2.4 Type Declarations

Type is an attribute that is possessed by every value and every variable. A data type defines a set of values which variables declared with that type may assume. A distinction is made between elementary types and structured types. Elementary types include those defined by the pervasive identifiers **CARDINAL**, **COMPLEX**, **INTEGER**, **REAL**, **LONGREAL**, **CHAR**, and **BOOLEAN** and user defined types derived from these, as well as enumerated, set, pointer, and procedure types. There are two different structured types, namely arrays and records.

A type declaration is the mechanism used to give another name to an existing type, or to create new types. It introduces an identifier to denote the type. In the case of a structured type it also defines the structure of variables of that type. Each occurrence of new type defines a type that is different from any other type.

Concrete Syntax

The required types and the corresponding type-identifiers are as specified in 6.9.1

type declaration = identifier, "=", type ;

Abstract Syntax

Type-declaration :: *id* : *Identifier*
 type : *Type*

Declaration Semantics

functions

d-type-declaration : *Type-declaration* → *Environment* → *Environment*

d-type-declaration (*mk-Type-declaration* (*id*, *type*)) $\rho \triangleq$
 77 let (*typen*, ρ_{type}) = *d-type* (*type*) ρ in
 271 *overwrite-type-environment* ($\{id \mapsto typen\}$) ρ_{type}

annotations The type is elaborated and the environment updated with the identifier and the result of this elaboration.

Static Semantics

functions

wf-type-declaration : *Type-declaration* → *Environment* → \mathbb{B}

wf-type-declaration (*mk-Type-declaration* (*id*, *type*)) $\rho \triangleq$
 73 *id* \notin *identifiers-used-in-type* (*type*) \wedge
 77 *wf-type* (*type*) ρ

Dynamic Semantics

operations

m-type-declaration : *Type-declaration* → *Environment* \xrightarrow{o} *Environment*

m-type-declaration (*mk-Type-declaration* (*id*, *type*)) $\rho \triangleq$
 50 return *d-type-declaration* (*mk-Type-declaration* (*id*, *type*)) ρ

6.2.5 Variable Declarations

A variable declaration serves to introduce variables and associate them with a unique identifier and a fixed data type and structure.

Concrete Syntax

variable declaration = variable identifier list, ":", type ;

variable identifier list = identifier, [machine address], { ",", identifier, [machine address] } ;

machine address = left bracket, value of machine address or address type, right bracket ;

value of machine address or address type = constant expression ;

NOTE — The machine address component is a constant expression that requires an explicit import of MACHINEADDRESS or ADDRESSVALUE from SYSTEM.

The identifiers occurring in each variable declaration shall be distinct and each identifier shall occur in only one variable declaration.

Abstract Syntax

Variable-declaration :: *ids* : *Variable-identifier-set*
 type : *Type*

Variable-identifier :: *id* : *Identifier*
 addr : [*Expression*]

annotations Identifiers may only occur once in a variable identifier list, and thus the cardinality of the set of identifiers of a *Variable-declaration* is the same as the number of identifiers in the variable identifier list.

Declaration Semantics

Variables occurring in the same variable identifier list shall have the same type.

functions

d-variable-declaration : *Variable-declaration* \rightarrow *Environment* \rightarrow *Environment*

d-variable-declaration (*mk-Variable-declaration* (*vars*, *type*)) $\rho \triangleq$

77 *let* (*typen*, ρ_{type}) = *d-type* (*type*) ρ *in*

272 *overwrite-var-environment* ($\{id \mapsto typen \mid mk\text{-}Variable\text{-}identifier(id, -) \in vars\}$) ρ_{type}

annotations The type component is elaborated and the identifiers that are introduced are each associated with the result of the elaboration and added to the environment.

Static Semantics

functions

wf-variable-declaration : *Variable-declaration* \rightarrow *Environment* $\rightarrow \mathbb{B}$

wf-variable-declaration (*mk-Variable-declaration* (*vars*, *type*)) $\rho \triangleq$

$\text{let } ids = \{id \mid is\text{-}Identifier(id) \wedge mk\text{-}Variable\text{-}identifier(id, -) \in vars\} \text{ in}$

73 $ids \cap identifiers\text{-}used\text{-}in\text{-}type(type) = \{\}$ \wedge

77 *wf-type* (*type*) $\rho \wedge$

$\forall mk\text{-}Variable\text{-}identifier(-, expr) \in vars .$

$expr = \text{nil} \vee$

152 $(wf\text{-}expression(expr)\rho \wedge$

214 $is\text{-}constant\text{-}expression(expr)\rho \wedge$

152 $\text{let } etype = t\text{-}expression(expr)\rho \text{ in}$

282 $is\text{-}machine\text{-}address\text{-}type(etype)\rho \vee$

282 $is\text{-}address\text{-}type(etype)\rho)$

Dynamic Semantics

operations

m-variable-declaration : *Variable-declaration* \rightarrow *Environment* \xrightarrow{o} *Environment*

m-variable-declaration (*mk-Variable-declaration* (*ids*, *type*)) $\rho \triangleq$

51 $\text{let } \rho_{var} = d\text{-variable-declaration}(mk\text{-}Variable\text{-}declaration(ids, type))\rho \text{ in}$

52 $allocate\text{-}variables(ids) \rho_{var}$

annotations The elaboration of a variable declaration involves the elaboration of the type information to give the structure of the storage associated with the type, followed by the allocation of that storage.

Auxiliary Definitions

operations

$allocate_variables : Variable_identifier_set \rightarrow Environment \xrightarrow{o} Environment$

$allocate_variables(vars)\rho \triangleq$

if $vars = \{\}$

then return ρ

else let $var \in vars$ in

52 def $\rho_{var} = allocate_a_variable(var)\rho$;

52 $allocate_variables(vars - \{var\})\rho_{var}$

annotations Each of the identifiers in the variable declaration has storage allocated for it.

;

$allocate_a_variable : Variable_identifier \rightarrow Environment \xrightarrow{o} Environment$

$allocate_a_variable(var)\rho \triangleq$

let $mk_Variable_identifier(id, expr) = var$ in

276 let $type = access_environment(id)\rho$ in

?? def $var = m_type(type)\rho$;

(if $expr \neq nil$

153 then def $val = m_expression(expr)\rho$;

300 $alias_machine_address(var, val)$

else skip;

272 return $overwrite_var_environment(\{id \mapsto var\})\rho$)

annotations If the variable is of elementary type, storage is allocated for it and information of the storage added to the environment. If the variable is structured, then storage is allocated for each of its components, and the storage information is added to the environment. The result of this operation is the updated environment.

6.2.6 Procedure Declarations

A procedure declaration associates a sequence of actions with an identifier. There are two kinds of procedures, namely proper procedures and function procedures. Proper procedures are activated by a procedure call. Function procedures are activated by a function designator as a component of an expression, and yield a result that is used as an operand in the expression that contains the activation. A function procedure is distinguished from a proper procedure by the specification of a function result type following the formal parameter list.

Concrete Syntax

procedure declaration = proper procedure declaration | function procedure declaration ;

Abstract Syntax

$Procedure_declaration = Proper_procedure_declaration \mid Function_procedure_declaration$

6.2.6.1 Proper Procedure Declarations

A proper procedure declaration associates a collection of local declarations and a statement sequence with an identifier. The declaration consists of a procedure heading followed by procedure body. The heading serves to declare the procedure identifier and the formal parameters. The procedure body defines the local declarations and the sequence of actions that are associated with the procedure identifier. A proper procedure does not return a result.

Concrete Syntax

```
proper procedure declaration =  
    proper procedure heading, " ;",  
    ( proper procedure block, procedure identifier | "FORWARD" ) ;  
  
procedure identifier = identifier ;
```

The procedure identifier component of the proper procedure heading component of a proper procedure declaration shall be the same as the procedure identifier component following the proper procedure block of that proper procedure declaration.

CHANGE — The keyword **FORWARD** is not in early editions of *Programming in Modula-2*.

TO DO — The check that a procedure that appears in a forward declaration must be completely declared within the same scope as the forward declaration needs to be added to the definition.

TO DO — Add **FORWARD** to the abstract syntax.

Abstract Syntax

Proper-procedure-declaration :: *head* : *Proper-procedure-heading*
 block : *Proper-procedure-block*

Declaration Semantics

functions

```
d-proper-procedure-declaration : Proper-procedure-declaration → Environment → Environment  
d-proper-procedure-declaration (mk-Proper-procedure-declaration (head, -)) ρ  $\triangleq$   
55   let ptype = construct-proper-procedure-type (head) in  
86   let struc = d-proper-procedure-type (ptype) ρ in  
    let proc = mk-Proper-procedure-structure (struc) in  
274  let level = current-level (ρ) in  
273  overwrite-proc-environment ({name ↦ mk-Procedure-const (proc, level)}) ρ
```

annotations In the environment for the block containing the declaration of a proper procedure, the object associated with the procedure identifier is the type of the procedure.

Static Semantics

The identifiers declared in the formal parameter list shall be distinct from the identifiers declared in the declarations component of the proper procedure block.

The environment for the proper procedure block shall be that constructed from the declarations of the block containing the procedure declaration overwritten by the parameters declared in the procedure heading.

It shall be an error for an access procedure of a directly protected procedure or module to be a directly protected procedure with itself as an access procedure.

NOTE — The following is illegal (because an infinite recursion would occur as **ENTER** is used as access procedure for itself).

```
MODULE any [42];  
  EXPORT ENTER;  
  ...  
END any;
```

functions

$wf\text{-}proper\text{-}procedure\text{-}declaration : Proper\text{-}procedure\text{-}declaration \rightarrow Environment \rightarrow \mathbb{B}$
 $wf\text{-}proper\text{-}procedure\text{-}declaration (mk\text{-}Proper\text{-}procedure\text{-}declaration (head, block))\rho \triangleq$
56 $wf\text{-}proper\text{-}procedure\text{-}heading (head)\rho \wedge$
55 $is\text{-}procedure\text{-}declarations\text{-}distinct (head, block) \wedge$
53 $\text{let } \rho_1 = d\text{-}proper\text{-}procedure\text{-}declaration (head)\rho \text{ in}$
274 $\text{let } \rho_2 = new\text{-}level(\rho_1) \text{ in}$
57 $wf\text{-}proper\text{-}procedure\text{-}block (block)\rho_2 \wedge$
56 $is\text{-}valid\text{-}access\text{-}control\text{-}in\text{-}procedure (\rho_2)$
annotations The environment for the proper procedure block is constructed from the environment of the declarations that contain the proper procedure declaration overwritten by the parameters declared in the proper procedure heading. The proper procedure block introduces a new scope which includes the parameter declarations. If the proper procedure is in a protected domain, check that the types of the domain entry and exit procedures are correct.

;

$wf\text{-}proper\text{-}procedure : Proper\text{-}procedure\text{-}declaration \rightarrow Environment \rightarrow \mathbb{B}$
 $wf\text{-}proper\text{-}procedure (decl)\rho_{proc} \triangleq$
 $\text{let } mk\text{-}Proper\text{-}procedure\text{-}declaration (head, block) = decl \text{ in}$
56 $\text{let } \rho_{parm} = d\text{-}proper\text{-}procedure\text{-}heading (head)\rho_{proc} \text{ in}$
274 $\text{let } \rho_{block} = new\text{-}level(\rho_{parm}) \text{ in}$
57 $wf\text{-}proper\text{-}procedure\text{-}block (block)\rho_{block}$
annotations Check the statements of a proper procedure block.

Dynamic Semantics

A procedure call specifies the activation of the block associated with the procedure denotation. If the procedure has any formal parameters, the designator is followed by a list of actual parameters that shall be bound to their corresponding formal parameters defined in the proper procedure declaration. The correspondence shall be established by the positions of the parameters in the lists of actual and formal parameters respectively. If any parameter is a value parameter, the corresponding actual parameter shall be an expression which is evaluated prior to the procedure activation, and is assigned to the corresponding formal parameter.

If the procedure is directly protected, then on activation of the procedure the entry procedure shall be called prior to the initialisation of any local modules declared in the block of the procedure; the exit procedure shall be called on completion of the statement part of the procedure.

operations

$m\text{-}proper\text{-}procedure\text{-}declaration : Proper\text{-}procedure\text{-}declaration \rightarrow Environment \xrightarrow{o} Proper\text{-}procedure\text{-}value$
 $m\text{-}proper\text{-}procedure\text{-}declaration (mk\text{-}Proper\text{-}procedure\text{-}declaration (head, block))\rho \triangleq$
 $\text{let } mk\text{-}Proper\text{-}procedure\text{-}heading (name, parms) = head \text{ in}$
55 $\text{let } fids = identifiers\text{-}of\text{-}formal\text{-}parameter\text{-}list (parms) \text{ in}$
68 $\text{let } ids = identifiers\text{-}declared\text{-}in (block.decls) \text{ in}$
 $\text{let } f = \lambda args .$
221 $\text{def } \rho_1 = bind\text{-}arguments(parms, args) \rho \text{ in}$
40 $(optional\text{-}call\text{-}of\text{-}domain\text{-}entry\text{-}procedure()) \rho;$
58 $m\text{-}proper\text{-}procedure\text{-}block(block) \rho_1;$
41 $optional\text{-}call\text{-}of\text{-}domain\text{-}exit\text{-}procedure() \rho;$
299 $deallocate\text{-}parameters(fids) \rho_1;$
299 $deallocate\text{-}storage(ids) \rho_1) \text{ in}$
275 $\text{let } pid = allocate\text{-}procedure\text{-}id() \rho \text{ in}$
 $\text{let } den = mk\text{-}Proper\text{-}procedure\text{-}value (f) \text{ in}$
273 $\text{return } (overwrite\text{-}proc\text{-}environment (\{name \mapsto pid\})\rho,$
275 $overwrite\text{-}den\text{-}environment (\{pid \mapsto den\})\rho)$

annotations The elaboration of a proper procedure declaration is a proper procedure denotation that is the meaning of the parameters followed by the meaning of the block of the procedure. On activation, the actual parameters are bound to the formal parameters and the proper procedure block is executed. On the completion of the execution of the proper procedure block component, the storage associated with the parameters and local parameters is freed.

Auxiliary Functions

functions

construct-proper-procedure-type : *Proper-procedure-heading* \rightarrow *Proper-procedure-type*

construct-proper-procedure-type (*pph*) \triangleq
 let *mk-Proper-procedure-heading* (*-*, *parms*) = *pph* in
 let *fpl* = *construct-parameter-type-list* (*parms*) in
 mk-Proper-procedure-type (*fpl*)

annotations Convert a proper procedure heading into a proper procedure type.

;

construct-parameter-type-list : *Formal-parameter-list* \rightarrow *Formal-parameter-type-list*

construct-parameter-type-list (*fpl*) \triangleq
 [*construct-parameter-type* (*fpl*(*i*)) | *i* \in *inds fpl*]

annotations Convert a formal parameter list into a formal parameter type list.

;

construct-parameter-type : *Formal-parameter* \rightarrow *Formal-parameter-type*

construct-parameter-type (*fp*) \triangleq
 cases *fp* :
 mk-Value-parameter-specification (*-*, *ftype*) \rightarrow *mk-Value-formal-type* (*ftype*),
 mk-Variable-parameter-specification (*-*, *ftype*) \rightarrow *mk-Variable-formal-type* (*ftype*)
 end

annotations Convert a value or variable parameter specification into a value or variable formal type.

;

is-procedure-declarations-distinct : *Proper-procedure-heading* \times *Proper-procedure-block* \rightarrow \mathbb{B}

is-procedure-declarations-distinct (*head*, *block*) \triangleq
 let *fids* = *identifiers-of-formal-parameter-list* (*head.parms*) in
 let *vars* = *identifiers-declared-in* (*block.decls*) in
 fids \cap *vars* = { }

annotations Check that the identifiers declared in a proper procedure heading are not re-declared in the associated proper procedure block.

;

identifiers-of-formal-parameter-list : *Formal-parameter-list* \rightarrow *Identifier-set*

identifiers-of-formal-parameter-list (*parms*) \triangleq
 { *identifier-of-a-formal-parameter* (*parm*) | *parm* \in *elems parms* }

annotations The result is a set of identifiers that name the formal parameters of a formal parameter list.

;

identifier-of-a-formal-parameter : *Formal-parameter* \rightarrow *Identifier*

identifier-of-a-formal-parameter (*parm*) \triangleq
 cases *parm* :
 mk-Value-parameter-specification (*id*, *-*) \rightarrow *id*,
 mk-Variable-parameter-specification (*id*, *-*) \rightarrow *id*
 end

annotations The result is an identifier that is the name of a formal parameter.
 ;

$$is\text{-}valid\text{-}access\text{-}control\text{-}in\text{-}procedure : Proper\text{-}procedure\text{-}block \rightarrow Environment \rightarrow \mathbb{B}$$

$$is\text{-}valid\text{-}access\text{-}control\text{-}in\text{-}procedure (block)\rho \triangleq$$

$$\text{let } mk\text{-}Proper\text{-}procedure\text{-}block(-, actions) = block \text{ in}$$

$$is\text{-}valid\text{-}access\text{-}control\text{-}procedures(actions)\rho$$

annotations Check that any calls of access control procedures are valid.

6.2.6.2 Proper Procedure Headings

A proper procedure heading declares the name and the formal parameters of a proper procedure.

Concrete Syntax

proper procedure heading = "PROCEDURE", procedure identifier, [formal parameter list] ;
 formal parameter list = "(" , [formal parameter, { ";", formal parameter }], ")" ;

Abstract Syntax

$Proper\text{-}procedure\text{-}heading :: name : Identifier$
 $parms : Formal\text{-}parameter\text{-}list$

$Formal\text{-}parameter\text{-}list = Formal\text{-}parameter^*$

Declaration Semantics

functions

$d\text{-}proper\text{-}procedure\text{-}heading : Proper\text{-}procedure\text{-}heading \rightarrow Environment \rightarrow Environment$
 $d\text{-}proper\text{-}procedure\text{-}heading (mk\text{-}Proper\text{-}procedure\text{-}heading(-, parms))\rho \triangleq$

$$\text{let } lenv = \bigcup \{ d\text{-}formal\text{-}parameter(parm)\rho \mid parm \in \text{elems } parms \} \text{ in}$$

$$overwrite\text{-}var\text{-}environment(lenv)\rho$$

annotations Construct an environment that associates each parameter with its type. This environment is added to the current environment.

Static Semantics

The identifiers declared as formal parameters in the formal parameter list shall be distinct from one another, and distinct from any type identifier component.

functions

$wf\text{-}proper\text{-}procedure\text{-}heading : Proper\text{-}procedure\text{-}heading \rightarrow Environment \rightarrow \mathbb{B}$
 $wf\text{-}proper\text{-}procedure\text{-}heading (mk\text{-}Proper\text{-}procedure\text{-}heading(-, parms))\rho \triangleq$

$$is\text{-}formal\text{-}parameters\text{-}distinct(parms) \wedge$$

$$\forall parm \in \text{elems } parms .$$

$$wf\text{-}formal\text{-}parameter(parm)\rho$$

Dynamic Semantics

annotations There are no dynamic semantics for proper procedure headings.

Auxiliary Functions

functions

$$is_formal_parameters_distinct : Formal_parameter_list \rightarrow \mathbb{B}$$
$$is\text{-}formal\text{-}parameters\text{-}distinct(parms) \triangleq$$

```
55 let ids = identifiers-of-formal-parameter-list(parms) in
```

$$\text{card } ids = \text{len } parms \wedge$$
$$ids \cap identifiers-used-in-parameters(parms) = \{ \}$$

annotations	Check that the identifiers declared in a formal parameter list are all distinct.
-------------	--

6.2.6.3 Proper Procedure Blocks

A proper procedure block defines the sequence of actions associated with a proper procedure declaration.

Concrete Syntax

```
proper procedure block = declarations, [ "BEGIN", statement sequence ], "END" ;
```

Abstract Syntax

Proper-procedure-block :: *decls* : *Declarations*
actions : *Block-body*

Static Semantics

The return statements of a proper procedure block shall be simple return statements.

functions

$$wf_proper_procedure_block : Proper_procedure_block \rightarrow Environment \rightarrow \mathbb{B}$$
$$wf\text{-}proper\text{-}procedure\text{-}block\ (mk\text{-}Proper\text{-}procedure\text{-}block\ (decls, actions))\rho \triangleq$$

46 let $\rho_1 = d\text{-declarations}(decls)\rho$ in

47 $wf\text{-}declarations\ (decls)\rho_1 \wedge$

246 let $\rho_{body} = \text{overwrite-environment}(d\text{-declarations}(decls)\rho_{body})\rho_{proc}$ in

$$?? \quad wf\text{-}block\text{-}body(actions)\rho_{body} \wedge$$
139 *wf-simple-return-statements (actions)*

annotations	Check the declarations and body of a proper procedure block.
-------------	--

Dynamic Semantics

The activation of a proper procedure block shall be defined by elaborating the declarations of the block, initialising any local modules, and then executing the statement sequence component of the block. The execution of a return statement in the statement sequence shall cause termination of the proper procedure block.

operations

```

m-proper-procedure-block : Proper-procedure-block  $\rightarrow$  Environment  $\xrightarrow{o}$  ()
m-proper-procedure-block (mk-Proper-procedure-block (decls, actions)) $\rho \triangleq$ 
47   def  $\rho_{body} = m\text{-declarations}(decls) \rho$ ;
39   (initialise-local-modules(decls)  $\rho_{body}$ );
??   def  $\rho_{cont} = c\text{-tixe}(\{\text{RETURN} \mapsto \text{skip}\}) \rho_{body}$ ;
??   m-block-body(actions)  $\rho_{cont}$ 

```

6.2.6.4 Function Procedure Declarations

A function procedure declaration associates a collection of local declarations and a statement sequence with an identifier. The declaration consists of a procedure heading followed by procedure body. The heading serves to declare the procedure identifier, the formal parameters, and the type of the result of the function procedure. The procedure body defines the local declarations and the sequence of actions that are associated with the procedure identifier and that calculate the result of the procedure.

Concrete Syntax

```

function procedure declaration =
  function procedure heading, ";",
  ( function procedure block, procedure identifier | "FORWARD" ) ;

```

The procedure identifier component of the proper procedure heading component of a function procedure declaration shall be the same as the procedure identifier component following the function procedure block of that function procedure declaration.

Abstract Syntax

```

Function-procedure-declaration :: head : Function-procedure-heading
                                block : Function-procedure-block

```

Declaration Semantics

functions

```

d-function-procedure-declaration : Function-procedure-declaration  $\rightarrow$  Environment  $\rightarrow$  Environment
d-function-procedure-declaration (mk-Function-procedure-declaration (head, -)) $\rho \triangleq$ 
60   let pctype = construct-function-procedure-type (head) in
87   let struc = d-function-procedure-type (pctype) $\rho$  in
    let proc = mk-Function-procedure-structure (struc) in
274  let level = current-level( $\rho$ ) in
273  overwrite-proc-environment ( $\{ name \mapsto mk\text{-Procedure-const}(proc, level) \}$ ) $\rho$ 

```

annotations In the environment containing the declaration of a proper procedure, the object associated with the procedure identifier is the type of the procedure.

Static Semantics

The identifiers declared in the formal parameter list shall be distinct from the identifiers declared in the declarations component of the function procedure block.

The environment for the function procedure block shall be that constructed from the declarations of the block containing the procedure declaration overwritten by the parameters declared in the procedure heading.

It shall be an error for an access procedure of a directly protected procedure or module to be a directly protected procedure with itself as an access procedure.

functions

```

wf-function-procedure-declaration : Function-procedure-declaration → Environment → ℬ
wf-function-procedure-declaration (mk-Function-procedure-declaration (head, block)) ρ  $\triangleq$ 
61   wf-function-procedure-heading (head) ρ ∧
60   is-function-declarations-distinct (head.parms, block) ∧
61   let ρ1 = d-function-procedure-heading (head) ρ in
78   let (rtype, -) = d-type-identifier (head.return) ρ in
274  let ρ2 = new-level (ρ1) in
62   wf-function-procedure-block (block) (ρ2, rtype) ∧
60   is-valid-access-control-in-function (ρ)

```

annotations

The environment for the function procedure block is constructed from the declarations of the surrounding environment overwritten by the parameters declared in the proper procedure heading. The function procedure block introduces a new scope which includes the parameter declarations. If the proper procedure is in a protected domain, check that the types of the domain entry and exit procedures are correct.

;

```

wf-function-procedure : Function-procedure-declaration → Environment → ℬ
wf-function-procedure (decl) ρproc  $\triangleq$ 
   let mk-Function-procedure-declaration (head, block) = decl in
61   let ρparm = d-function-procedure-heading (head) ρproc in
274  let ρproc = new-level (ρparm) in
   let mk-Function-procedure-block (decls, -) = block in
46   let ρbody = d-declarations (decls) ρproc in
78   let (rtype, -) = d-type-identifier (head.return) ρ in
62   wf-function-procedure-block (block) (ρbody, rtype)

```

Dynamic Semantics

A function call specifies the activation of the block associated with the procedure denotation. If the procedure has any formal parameters, the designator is followed by a list of actual parameters that shall be bound to their corresponding formal parameters defined in the function procedure declaration. The correspondence shall be established by the positions of the parameters in the lists of actual and formal parameters respectively. If any parameter is a value parameter, the corresponding actual parameter shall be an expression which is evaluated prior to the procedure activation, and is assigned to the corresponding formal parameter.

If the procedure is directly protected, then on activation of the procedure the entry procedure shall be called prior to the initialisation of any local modules declared in the block of the procedure; the exit procedure shall be called on completion of the statement part of the procedure.

operations

$m\text{-function-procedure-declaration} : \text{Function-procedure-declaration} \rightarrow \text{Environment} \xrightarrow{\circ} \text{Function-procedure-value}$
 $m\text{-function-procedure-declaration}(\text{mk-Function-procedure-declaration}(\text{head}, \text{block}))\rho \triangleq$
 let $\text{mk-Function-procedure}(\text{name}, \text{parms}, -) = \text{head}$ in
 55 let $\text{fids} = \text{identifiers-of-formal-parameter-list}(\text{parms})$ in
 68 let $\text{ids} = \text{identifiers-declared-in}(\text{block.decls})$ in
 let $f = \lambda \text{args} .$
 221 def $\rho_1 = \text{bind-arguments}(\text{parms}, \text{args}) \rho$ in
 40 (optional-call-of-domain-entry-procedure()) ρ ;
 62 def $\text{rval} = m\text{-function-procedure-block}(\text{block}) \rho_1$ in
 41 (optional-call-of-domain-exit-procedure()) ρ ;
 299 deallocate-parameters(fids) ρ_1 ;
 299 deallocate-storage(ids) ρ_1 ;
 return rval) in
 275 let $\text{pid} = \text{allocate-procedure-id}() \rho$ in
 let $\text{den} = \text{mk-Function-procedure-value}(f)$ in
 273 return (overwrite-proc-environment($\{\text{name} \mapsto \text{pid}\}$)) ρ ,
 275 overwrite-den-environment($\{\text{pid} \mapsto \text{den}\}$)) ρ

annotations

The elaboration of a function procedure declaration is a function procedure denotation that is the meaning of the parameters followed by the meaning of the block of the procedure. On activation, the actual parameters are bound to the formal parameters and the function procedure block is executed. On the completion of the execution of the function procedure block component, the storage associated with the parameters and local parameters is freed.

Auxiliary Functions

functions

$\text{is-function-declarations-distinct} : \text{Function-procedure-heading} \times \text{Proper-procedure-block} \rightarrow \mathbb{B}$
 $\text{is-function-declarations-distinct}(\text{head}, \text{block}) \triangleq$
 55 let $\text{fids} = \text{identifiers-of-formal-parameter-list}(\text{head.parms})$ in
 68 let $\text{vars} = \text{identifiers-declared-in}(\text{block.decls})$ in
 $\text{fids} \cap \text{vars} = \{\}$;

 $\text{construct-function-procedure-type} : \text{Function-procedure-heading} \rightarrow \text{Function-procedure-type}$
 $\text{construct-function-procedure-type}(\text{pph}) \triangleq$
 let $\text{mk-Function-procedure-heading}(-, \text{parms}, \text{return}) = \text{pph}$ in
 55 let $\text{fpl} = \text{construct-parameter-type-list}(\text{parms})$ in
 $\text{mk-Function-procedure-type}(\text{fpl}, \text{return})$;

 $\text{is-valid-access-control-in-function} : \text{Proper-procedure-block} \rightarrow \text{Environment} \rightarrow \mathbb{B}$
 $\text{is-valid-access-control-in-function}(\text{block})\rho \triangleq$
 let $\text{mk-Function-procedure-block}(\text{decls}, \text{actions}) = \text{block}$ in
 ?? $\text{is-valid-access-control-procedures}(\text{actions})\rho$

annotations

Check that any calls of access control procedures are valid.

6.2.6.5 Function Procedure Headings

A function procedure heading declares the name and the formal parameters of a function procedure, and defines the type of the result to be associated with the procedure identifier.

Concrete Syntax

function procedure heading = "PROCEDURE", procedure identifier, formal parameter list, function result type ;

Abstract Syntax

Function-procedure-heading :: *name* : *Identifier*
 parms : *Formal-parameter-list*
 return : *Type-identifier*

Declaration Semantics

functions

$d\text{-function-procedure-heading} : \text{Function-procedure-heading} \rightarrow \text{Environment} \rightarrow \text{Environment}$
 $d\text{-function-procedure-heading}(\text{mk-Function-procedure-heading}(-, \text{parms}, \text{return}))\rho \triangleq$
63 $\text{let } \text{lenv} = \bigcup \{d\text{-formal-parameter}(\text{parms}(i))\rho \mid i \in \text{inds } \text{parms}\} \text{ in}$
272 $\text{overwrite-var-environment}(\text{lenv})\rho$

annotations Construct an environment that associates each parameter with its type. This environment is added to the current environment.

Static Semantics

The identifiers declared as formal parameters in the formal parameter list shall be distinct. The qualident that defines the return type of the function procedure shall be a type.

functions

$wf\text{-function-procedure-heading} : \text{Function-procedure-heading} \rightarrow \text{Environment} \rightarrow \mathbb{B}$
 $wf\text{-function-procedure-heading}(\text{mk-Function-procedure-heading}(-, \text{parms}, \text{return}))\rho \triangleq$
57 $\text{is-formal-parameters-distinct}(\text{parms}) \wedge$
78 $wf\text{-type-identifier}(\text{return})\rho \wedge$
 $\forall \text{parm} \in \text{elems } \text{parms} \cdot$
63 $wf\text{-formal-parameter}(\text{parm})\rho$

Dynamic Semantics

annotations There are no dynamic semantics for function procedure headings.

6.2.6.6 Function Procedure Blocks

A function procedure block defines the sequence of actions associated with a function procedure declaration.

Concrete Syntax

function procedure block = declarations, "BEGIN", statement sequence, "END" ;

CHANGE — *Programming in Modula-2* allows the body of a function procedure to be null, but the well-formed rules in this International Standard demand at least one return statement. The concrete syntax has been changed to reflect this.

Abstract Syntax

Function-procedure-block :: *decls* : *Declarations*
 actions : *Block-body*

Static Semantics

The block of a function procedure shall contain at least one return statement that defines the result of the procedure; the return statements shall be function procedure return statements. The type of the expression component of every return statement in the body of the procedure block shall be assignment compatible with the return type of the procedure.

functions

```
wf-function-procedure-block : Function-procedure-block → Environment × Typed → B
wf-function-procedure-block (mk-Function-procedure-block (decls, actions))(ρ, rtype) ≜
46   let ρ1 = d-declarations (decls) ρ in
47   wf-declarations (decls) ρ1 ∧
246  let ρblock = overwrite-environment (d-declarations (decls) ρblock) ρ in
??   wf-block-body (actions) ρblock ∧
139  wf-function-return-statements (rtype, actions) ρblock
```

Dynamic Semantics

The activation of a function procedure block shall be defined by elaborating the declarations of the block, initialising any local modules, and then executing the statement sequence component of the block. The execution of a return statement in the statement sequence shall cause termination of the function procedure block. It shall be an exception if termination of the statement sequence is not by the execution of a return statement.

operations

```
m-function-procedure-block : Function-procedure-block → Environment  $\xrightarrow{o}$  Value × Environment
m-function-procedure-block (mk-Function-procedure-block (block)) ρ ≜
47   def ρbody = m-declarations(decls) ρ;
39   ( initialise-local-modules(decl) ρbody;
??   def ρcont = c-tixe({RETURN ↦ return(x)}) ρbody;
??   m-block-body(actions) ρcont;
306  mandatory-exception(NORETURN) )
```

TO DO — What is x in the parameters of *c-tixe*? (the same question applies to D106)

6.2.6.7 Parameters

A parameter provides a disciplined means of communication between the invoking code and the procedure block. There are two kinds of parameters: value parameters and variable parameters.

Concrete Syntax

formal parameter = value parameter specification | variable parameter specification ;

Abstract Syntax

Formal-parameter = *Value-parameter-specification* | *Variable-parameter-specification*

Declaration Semantics

functions

```
d-formal-parameter : Formal-parameter → Environment → Environment  
d-formal-parameter (parm)  $\triangle$   
  cases parm :  
63   mk-Value-parameter-specification () → d-value-parameter-specification (parm)  $\rho$ ,  
64   mk-Variable-parameter-specification () → d-variable-parameter-specification (parm)  $\rho$   
  end
```

Static Semantics

functions

```
wf-formal-parameter : Formal-parameter → Environment →  $\mathbb{B}$   
wf-formal-parameter (parm)  $\rho \triangle$   
  cases parm :  
64   mk-Value-parameter-specification () → wf-value-parameter-specification (parm)  $\rho$ ,  
64   mk-Variable-parameter-specification () → wf-variable-parameter-specification (parm)  $\rho$   
  end
```

6.2.6.7.1 Value Parameters

Value parameters provide a one-way communication by passing a value into the procedure block. A value parameter acts like a local variable to which the result of the evaluation of the corresponding actual parameter is assigned as an initial value.

Concrete Syntax

value parameter specification = identifier list, ":", formal type ;

Abstract Syntax

Value-parameter-specification :: *id* : *Identifier*
 ftype : *Formal-type*

Translation Note

Each identifier in the identifier list is associated with the translation of the formal type to construct the corresponding abstract syntax.

Declaration Semantics

functions

```
d-value-parameter-specification : Value-parameter-specification → Environment → Value-formal-typed  
d-value-parameter-specification (mk-Value-parameter-specification (id, ftype))  $\rho \triangle$   
90  let mk-Value-formal-typed (type) = d-formal-type (ftype)  $\rho$  in  
    { id ↦ type }
```


Static Semantics

functions

$$\begin{aligned}
& wf_value_parameter_specification : Value_parameter_specification \rightarrow Environment \rightarrow \mathbb{B} \\
& wf_value_parameter_specification (mk_Value_parameter_specification (-, ftype)) \rho \triangleq \\
90 \quad & wf_formal_type (ftype) \rho
\end{aligned}$$

Dynamic Semantics

The dynamic semantics of value formal parameters are given by procedure activation (see 6.8.3).

6.2.6.7.2 Variable Parameters

Variable parameters provide a means of two-way communication by passing a variable into the procedure block. Variable parameters correspond to actual parameters that are variables, and they stand for these variables.

Concrete Syntax

variable parameter specification = "VAR", identifier list, ":", formal type ;

Abstract Syntax

$$\begin{aligned}
Variable_parameter_specification &:: id \quad : Identifier \\
&\quad ftype : Formal_type
\end{aligned}$$

Translation Note

Each identifier in the identifier list is associated with the translation of the formal type to construct the corresponding abstract syntax.

Declaration Semantics

functions

$$\begin{aligned}
& d_variable_parameter_specification : Variable_parameter_specification \rightarrow Environment \rightarrow Variable_formal_typed \\
& d_variable_parameter_specification (Variable_parameter_specification(id, ftype)) \rho \triangleq \\
90 \quad & \text{let } mk_Variable_formal_typed (type) = d_formal_type (ftype) \rho \text{ in} \\
& \{id \mapsto type\}
\end{aligned}$$

Static Semantics

functions

$$\begin{aligned}
& wf_variable_parameter_specification : Variable_parameter_specification \rightarrow Environment \rightarrow \mathbb{B} \\
& wf_variable_parameter_specification (mk_Variable_parameter_specification (-, ftype)) \rho \triangleq \\
90 \quad & wf_formal_type (ftype) \rho
\end{aligned}$$

Dynamic Semantics

The dynamic semantics of variable formal parameters are given by procedure activation (see 6.8.3).

6.2.7 Module Declarations

A (local) module consists of a collection of import lists, an optional export list, a collection of declarations, and a sequence of statements. The import lists, if present, control the visibility of identifiers declared outside the module but usable within it. The export list, if present, controls the visibility of identifiers declared inside the module and usable outside it. The module thus acts as a wall around its local objects whose transparency is controlled.

Concrete Syntax

```
module declaration =  
    "MODULE", module identifier, [ protection ], ";",  
    { import lists }, [ export list ], module block, module identifier, "." ;
```

The module identifier components of a module declaration shall be the same.

Abstract Syntax

```
Module-declaration :: name      : Identifier  
                   imports     : Import-lists  
                   export      : [Export-list]  
                   block       : Module-block  
                   protection  : [Expression]
```

Declaration Semantics

functions

$d\text{-module-declaration} : \text{Module-declaration} \rightarrow \text{Environment} \rightarrow \text{Environment}$

$d\text{-module-declaration} (mk\text{-Module-declaration} (name, imports, export, block, -))\rho \triangleq$

```
24  let  $\rho_{imported}$  =  $d\text{-import-lists} (imports)\rho$  in  
??  let  $\rho_{internal}$  =  $d\text{-module-block} (block)\rho_{imported}$  in  
30  let  $\rho_{exported}$  =  $d\text{-export-list} (export)\rho_{internal}$  in  
274 let  $\rho_{module}$  =  $overwrite\text{-mod-environment} (\{name \mapsto \rho_{exported}\})\rho$  in  
    if  $is\text{-Unqualified-export}(export)$   
276 then  $merge\text{-environments} (\{\rho_{exported}, \rho_{module}\})$   
    else  $\rho_{module}$ 
```

annotations The result of elaborating a module declaration is a new environment that is derived from the environment associated with the declarations of the surrounding scope that precede the module declaration. The standard identifiers and their associated objects are available in the scope of the module block, but may be overwritten by imported identifiers, or by identifiers introduced by the declarations of the module. The result of this operation is the environment for subsequent declarations following the module declaration.

Static Semantics

An internal module shall not be imported into itself. The identifiers that are imported into the internal module shall not be declared in that module. Any identifiers, either implicitly or explicitly imported into the internal module shall be present in the surrounding environment. The environment for the block of an internal module shall be the surrounding environment restricted to the identifiers imported into the internal module.

A protection expression specified in the heading of a directly protected internal module shall be a constant expression in the surrounding environment, and shall be evaluated in that environment.

functions

$wf\text{-}module\text{-}declaration : Module\text{-}declaration \rightarrow Environment \rightarrow \mathbb{B}$

$wf\text{-}module\text{-}declaration (mk\text{-}Module\text{-}declaration (name, imports, export, block, protection))\rho_{sur} \triangleq$
 $name \notin \text{elems directly-imports}(imports) \wedge$
 $consistent\text{-}imports(imports, block)\rho_{sur} \wedge$
 $wf\text{-}use\text{-}of\text{-}imports(imports)\rho_{sur} \wedge$
 $wf\text{-}export\text{-}list(export)\rho_{sur} \wedge$
 $\text{let } \rho_{block} = d\text{-import-lists}(imports)\rho_{sur} \text{ in}$
 $consistent\text{-}exports(export, block)\rho_{block} \wedge$
 $wf\text{-}module\text{-}block(block)\rho_{block} \wedge$
 $is\text{-}valid\text{-}access\text{-}control(protection)\rho_{sur}$

annotations The imports are well-defined with respect to the declarations of the module if each individual import is well-defined with respect to the declarations of the module. The environment to check the consistency of the block component of the module is constructed from the environment associated with the declarations of the current scope that precede the module declaration, restricted to the identifiers of the import lists and augmented by the standard identifiers that are not redefined by any imports.

;

$wf\text{-}module : Module\text{-}declaration \rightarrow Environment \rightarrow \mathbb{B}$

$wf\text{-}module(decl)\rho_{sur} \triangleq$
 $\text{let } mk\text{-}Module\text{-}declaration(-, imports, -, block, protection) = decl \text{ in}$
 $\text{let } \rho_{block} = d\text{-import-lists}(imports)\rho_{sur} \text{ in}$
 $wf\text{-}module\text{-}block(block)\rho_{block}$

Dynamic Semantics

The result of elaborating a module declaration shall be the creation of the storage associated with the module, together with a new environment that shall be derived from the environment associated with the declarations of the current scope that precede the module declaration. The standard identifiers and their associated objects are available in the scope of the module block, but may be overwritten by imported identifiers, or by identifiers introduced by the declarations of the module.

operations

$m\text{-}module\text{-}declaration : Module\text{-}declaration \rightarrow Environment \xrightarrow{o} Environment$

$m\text{-}module\text{-}declaration (mk\text{-}Module\text{-}declaration (name, imports, export, block, protection))\rho_{sur} \triangleq$
 $\text{let } \rho_{imported} = d\text{-import-lists}(imports)\rho_{sur} \text{ in}$
 $\text{def } \rho_{internal} = m\text{-}module\text{-}block(block) \rho_{imported};$
 $\text{def } pval = evaluate\text{-}protection(protection) \rho_{sur};$
 $\text{let } \rho_{block} = add\text{-}protection(pval)\rho_{internal} \text{ in}$
 $\text{let } \rho_{exported} = d\text{-export-list}(export)\rho_{block} \text{ in}$
 $\text{let } \rho_{module} = overwrite\text{-}mod\text{-}environment(\{name \mapsto \rho_{block}\})\rho \text{ in}$
 $\text{if } is\text{-}Unqualified\text{-}export(export)$
 $\text{then return } merge\text{-}environments(\rho_{exported}, \rho_{module})$
 $\text{else return } \rho_{module}$

Auxiliary Functions

functions

$wf\text{-}use\text{-}of\text{-}imports : Import\text{-}lists \rightarrow Environment \rightarrow \mathbb{B}$

$wf\text{-}use\text{-}of\text{-}imports(imports)\rho \triangleq$
 $\forall mk\text{-}Unqualifying\text{-}import(from, -) \in \text{elems } imports .$
 $from \in \text{identifiers-in-scope}(\rho);$

consistent-exports : *Export-list* \times *Declarations* \rightarrow *Environment* $\rightarrow \mathbb{B}$

consistent-exports (*export*, *decls*) $\rho \triangleq$
 68 let *dids* = *identifiers-declared-in* (*decls*) ρ in
 cases *export* :
 mk-Unqualified-export (*ids*) \rightarrow *ids* \subseteq *dids*,
 mk-Qualified-export (*ids*) \rightarrow *ids* \subseteq *dids*
 end

annotations The identifiers exported from a module must be declared or defined in that module.

6.2.8 Auxiliary Functions

6.2.8.1 Well-formed Definitions

functions

is-disjoint-definitions : *Definitions* $\rightarrow \mathbb{B}$

is-disjoint-definitions (*defs*) \triangleq
 $\forall i, j \in \text{inds } \text{defs} \cdot$
 $i = j \vee$
 67 *identifiers-defined* (*defs*(*i*)) \cap *identifiers-defined* (*defs*(*j*)) = { }

annotations Check that no identifier is defined twice in a sequence of definitions.

;

identifiers-defined : *Definition* \rightarrow *Identifier-set*

identifiers-defined (*def*) \triangleq
 67 *identifiers-explicitly-defined* (*def*) \cup *identifiers-implicitly-defined* (*def*);

identifiers-explicitly-defined : *Definition* \rightarrow *Identifier-set*

identifiers-explicitly-defined (*def*) \triangleq
 cases *def* :
 mk-Constant-declaration (*id*, -) \rightarrow { *id* },
 mk-Type-declaration (*id*, -) \rightarrow { *id* },
 mk-Opaque-type-definition (*id*) \rightarrow { *id* },
 mk-Variable-declaration (*ids*, -) \rightarrow *ids*,
 mk-Proper-procedure-heading (*name*, -) \rightarrow { *name* },
 mk-Function-procedure-heading (*name*, -, -) \rightarrow { *name* }
 end

annotations Return a set containing those identifiers that have been explicitly defined by a definition.

;

identifiers-implicitly-defined : *Definition* \rightarrow *Identifier-set*

identifiers-implicitly-defined (*def*) \triangleq
 cases *def* :
 mk-Type-declaration (-, *type*) \rightarrow *identifiers-defined-in-type* (*type*),
 mk-Variable-declaration (-, *type*) \rightarrow *identifiers-defined-in-type* (*type*),
 others \rightarrow { }
 end

annotations Return a set containing those identifiers that have been implicitly defined in a definition - namely enumerated constants.

;

is-complete-definitions : *Definitions* \rightarrow *Environment* $\rightarrow \mathbb{B}$

is-complete-definitions (*defs*) $\rho \triangleq$
 $\forall \text{def} \in \text{elems } \text{defs} \cdot$
 255 *shielded* (*def*) \subseteq *identifiers-defined-in* (*defs*) \cup *identifiers-in-scope* (ρ);

$identifiers-defined-in : Definitions \rightarrow Identifier-set$

$identifiers-defined-in (defs) \triangleq$

67 $\bigcup \{ identifiers-defined (def) \mid def \in \text{elems } defs \}$

annotations The result of this operation is a set containing those identifiers that have been declared in a sequence of declarations.

6.2.8.2 Well-formed Declarations

functions

$is-disjoint-declarations : Declarations \rightarrow \mathbb{B}$

$is-disjoint-declarations (decls) \triangleq$

$\forall i, j \in \text{inds } decls .$

$i = j \vee$

68 $identifiers-declared (decls(i)) \cap identifiers-declared (decls(j)) = \{ \}$

annotations Check that no identifier is declared twice in a sequence of declarations.

;

$identifiers-declared : Declaration \rightarrow Identifier-set$

$identifiers-declared (decl) \triangleq$

68 $identifiers-explicitly-declared (decl) \cup identifiers-implicitly-declared (decl);$

$identifiers-explicitly-declared : Declaration \rightarrow Identifier-set$

$identifiers-explicitly-declared (decl) \triangleq$

cases $decl$:

$mk-Constant-declaration (id, -) \rightarrow \{ id \},$

$mk-Type-declaration (id, -) \rightarrow \{ id \},$

$mk-Variable-declaration (ids, -) \rightarrow ids,$

$mk-Proper-procedure-declaration (head, -) \rightarrow \{ head.name \},$

$mk-Function-procedure-declaration (head, -) \rightarrow \{ head.name \},$

$mk-Module-declaration (name, -) \rightarrow \{ name \}$

end

annotations Return a set containing those identifiers that have been declared in a declaration.

;

$identifiers-implicitly-declared : Declaration \rightarrow Identifier-set$

$identifiers-implicitly-declared (decl) \triangleq$

cases $decl$:

69 $mk-Type-declaration (-, type) \rightarrow identifiers-defined-in-type (type),$

69 $mk-Variable-declaration (-, type) \rightarrow identifiers-defined-in-type (type),$

others $\rightarrow \{ \}$

end

annotations Return a set containing those identifiers that have been implicitly declared in a declaration - namely enumerated constants.

;

$is-complete-declarations : Declarations \rightarrow Environment \rightarrow \mathbb{B}$

$is-complete-declarations (decls) \rho \triangleq$

$\forall decl \in \text{elems } decls .$

276 $shielded (decl) \subseteq identifiers-declared-in (decls) \cup identifiers-in-scope (\rho);$

$identifiers-declared-in : Declarations \rightarrow Identifier-set$

$identifiers-declared-in (decls) \triangleq$

68 $\bigcup \{ identifiers-declared (decl) \mid decl \in \text{elems } decls \}$

annotations The result of this operation is a set containing those identifiers that have been declared in a sequence of declarations.

Qualified Identifiers used in Types

functions

identifiers-defined-in-type : *Type* \rightarrow *Identifier-set*

identifiers-defined-in-type (*type*) \triangleq

cases *type* :

mk-Type-identifier (*qid*) $\rightarrow \{ \}$,

mk-Enumerated-type (*values*) \rightarrow *elems values*,

mk-Subrange-type () $\rightarrow \{ \}$,

69 *mk-Set-type* (*btype*) \rightarrow *identifiers-defined-in-type* (*btype*),

69 *mk-Pointer-type* (*type*) \rightarrow *identifiers-defined-in-type* (*type*),

mk-Procedure-type () $\rightarrow \{ \}$,

mk-Function-procedure-type () $\rightarrow \{ \}$,

69 *mk-Array-type* (*itype*, *ctype*) \rightarrow *identifiers-defined-in-type* (*itype*) \cup *identifiers-defined-in-type* (*ctype*),

69 *mk-Record-type* (*fields*) \rightarrow *identifiers-defined-in-fields-list* (*fields*)

end

annotations Return a set of identifiers containing all the identifiers defined in a type component of a definition. These are the enumerated type constants (identifiers).

;

identifiers-defined-in-fields-list : *Fields-list* \rightarrow *Identifier-set*

identifiers-defined-in-fields-list (*fields*) \triangleq

if *fields* = nil

then { }

69 else $\bigcup \{ \textit{identifiers-defined-in-fields}(\textit{field}) \mid \textit{field} \in \textit{elems fields} \}$

annotations Return the identifiers defined in a fields list. These are the enumerated type constants (identifiers).

;

identifiers-defined-in-fields : *Fields* \rightarrow *Identifier-set*

identifiers-defined-in-fields (*field*) \triangleq

cases *field* :

69 *mk-Fixed-fields* (*-*, *type*) \rightarrow *identifiers-defined-in-type* (*type*),

69 *mk-Variant-fields* (*-*, *-*, *variants*, *elsep*) \rightarrow *identifiers-defined-in-variants* (*variants*) \cup

69 *identifiers-defined-in-fields-list* (*elsep*)

end

annotations Return the identifiers defined in fields. These are the enumerated type constants (identifiers) defined in any type components of the components of the fields.

;

identifiers-defined-in-variants : *Variant** \rightarrow *Identifier-set*

identifiers-defined-in-variants (*variants*) \triangleq

69 $\bigcup \{ \textit{identifiers-defined-in-a-variant}(\textit{variant}) \mid \textit{variant} \in \textit{elems variants} \}$

annotations Return the identifiers defined in variants. These are the enumerated type constants (identifiers) defined in the components of the variants.

;

identifiers-defined-in-a-variant : *Variant* \rightarrow *Identifier-set*

identifiers-defined-in-a-variant (*variant*) \triangleq

let *mk-Variant* (*-*, *fields*) = *variant* in

69 *identifiers-defined-in-fields-list* (*fields*)

annotations Return the identifiers defined in a variant. These are the enumerated type constants (identifiers) defined in the components of the variant.

Qualified Identifiers Used in Declarations

functions

identifiers-used-in : *Declarations* \rightarrow *Qualident-set*

identifiers-used-in (*decls*) \triangleq

70 $\bigcup \{ \text{identifiers-used}(\text{decl}) \mid \text{decl} \in \text{elems } \text{decls} \}$

annotations The result is a set containing those qualified identifiers that have been used in a sequence of declarations.

;

identifiers-used : *Declaration* \rightarrow *Qualident-set*

identifiers-used (*decl*) ρ \triangleq

cases *decl* :

70 *mk-Constant-declaration* (\cdot , *expr*) \rightarrow *identifiers-used-in-expression* (*expr*),

73 *mk-Type-declaration* (\cdot , *type*) \rightarrow *identifiers-used-in-type* (*type*),

73 *mk-Variable-declaration* (\cdot , *type*) \rightarrow *identifiers-used-in-type* (*type*),

mk-Procedure-declaration () $\rightarrow \{ \}$,

mk-Function-procedure-declaration () $\rightarrow \{ \}$,

mk-Module-declaration (\cdot , *decls*) $\rightarrow \{ \}$

end

annotations The result is a set containing those qualified identifiers that have been used in the declaration of some identifier.

;

identifiers-used-in-expression : *Expression* \rightarrow *Qualident-set*

identifiers-used-in-expression (*expr*) \triangleq

70 (*is-Infix-expression*(*expr*) \rightarrow *identifiers-used-in-infix-expression* (*expr*),

70 *is-Prefix-expression*(*expr*) \rightarrow *identifiers-used-in-prefix-expression* (*expr*),

71 *is-Value-designator*(*expr*) \rightarrow *identifiers-used-in-value-designator* (*expr*),

71 *is-Function-call*(*expr*) \rightarrow *identifiers-used-in-function-call* (*expr*),

72 *is-Value-constructor*(*expr*) \rightarrow *identifiers-used-in-value-constructor* (*expr*),

is-Constant-literal(*expr*) $\rightarrow \{ \}$)

annotations The result is a set of qualified identifiers used in an expression; this is the set containing all the qualified identifiers used in the components of the expression.

;

identifiers-used-in-infix-expression : *Infix-expression* \rightarrow *Qualident-set*

identifiers-used-in-infix-expression (*expr*) \triangleq

let *mk-Infix-expression* (*left*, \cdot , *right*) = *expr* in

70 *identifiers-used-in-expression* (*left*) \cup *identifiers-used-in-expression* (*right*)

annotations The result is the set of qualified identifiers used in an infix expression; this is the set that is the union of the sets of qualified identifiers used in both of the expression components of the infix expression.

;

identifiers-used-in-prefix-expression : *Prefix-expression* \rightarrow *Qualident-set*

identifiers-used-in-prefix-expression (*pexpr*) \triangleq

let *mk-Prefix-expression* (\cdot , *expr*) = *pexpr* in

70 *identifiers-used-in-expression* (*expr*)

annotations The result is the set of qualified identifiers used in a prefix expression; this is the set of qualified identifiers used in the expression component of the prefix expression.

;

identifiers-used-in-value-designator : *Value-designator* \rightarrow *Qualident-set*

identifiers-used-in-value-designator (*vdesig*) \triangleq

cases *vdesig* :

mk-Entire-value (*desig*) $\rightarrow \{desig\}$,

71 *mk-Indexed-value* (*desig*, *expr*) \rightarrow *identifiers-used-in-value-designator* (*desig*) \cup
70 *identifiers-used-in-expression* (*expr*),

71 *mk-Selected-value* (*desig*, -) \rightarrow *identifiers-used-in-value-designator* (*desig*),

71 *mk-Dereferenced-value* (*desig*) \rightarrow *identifiers-used-in-value-designator* (*desig*)

end

annotations

The result is the set of qualified identifiers used in a value designator; this is either the set of qualified identifiers used in an indexed value, a selected value, or a dereferenced value.

The set of qualified identifiers used in a indexed value is the qualified identifiers used in the value designator component together with the qualified identifiers used in the expression component.

The set of qualified identifiers used in a selected value are the qualified identifiers used in the value designator component.

The set of qualified identifiers used in a dereferenced value are the qualified identifiers used in the value designator component.

;

identifiers-used-in-function-call : *Function-call* \rightarrow *Qualident-set*

identifiers-used-in-function-call (*fcall*) \triangleq

let *mk-Function-call* (*desig*, *args*) = *fcall* in

71 *identifiers-used-in-value-designator* (*desig*) \cup *identifiers-used-in-arguments* (*args*)

annotations

The result is the set of qualified identifiers used in a function call; this is the set of qualified identifiers used in the function designator component, together with the qualified identifiers used in the arguments of the function call.

;

identifiers-used-in-arguments : *Actual-parameters* \rightarrow *Qualident-set*

identifiers-used-in-arguments (*args*) \triangleq

71 $\bigcup \{ \text{identifiers-used-in-an-argument} (arg) \mid arg \in \text{elems } args \}$

annotations

The result is the set of qualified identifiers used in the arguments of a function call; this set is the union of all the qualified identifiers used in each of the arguments.

;

identifiers-used-in-an-argument : *Actual-parameter* \rightarrow *Qualident-set*

identifiers-used-in-an-argument (*arg*) \triangleq

71 (*is-Variable-designator* (*arg*) \rightarrow *identifiers-used-in-variable-designator* (*arg*),

70 *is-Expression* (*arg*) \rightarrow *identifiers-used-in-expression* (*arg*),

is-Type-parameter (*arg*) \rightarrow let *mk-Type-parameter* (*type*) = *arg* in
 $\{type\}$)

annotations

The result is a set of qualified identifiers used in an argument to a procedure or function call; this set is the qualified identifiers used in the actual parameter.

;

identifiers-used-in-variable-designator : *Variable-designator* \rightarrow *Qualident-set*

identifiers-used-in-variable-designator (*vdesig*) \triangleq

cases *vdesig* :

mk-Entire-designator (*desig*) $\rightarrow \{desig\}$,

mk-Indexed-designator (*desig*, *expr*) $\rightarrow identifiers-used-in-variable-designator(desig) \cup$
 $identifiers-used-in-expression(expr)$,

mk-Selected-designator (*desig*, -) $\rightarrow identifiers-used-in-variable-designator(desig)$,

mk-Dereferenced-designator (*desig*) $\rightarrow identifiers-used-in-variable-designator(desig)$

end

annotations The result is the set of qualified identifiers used in a variable designator; this set is either the qualified identifiers used in an indexed variable, a selected variable, or a dereferenced variable.

The set of qualified identifiers used in an entire variable is the qualified identifier used to denote the entire variable.

The set of qualified identifiers used in a indexed variable is the qualified identifiers used in the variable designator component together with the qualified identifiers used in the expression component.

The set of qualified identifiers used in a selected variable are the qualified identifiers used in the variable designator component.

The set of qualified identifiers used in a dereferenced variable are the qualified identifiers used in the variable designator component.

;

identifiers-used-in-value-constructor : *Value-constructor* \rightarrow *Qualident-set*

identifiers-used-in-value-constructor (*expr*) \triangleq

is-Array-constructor (*expr*) $\rightarrow identifiers-used-in-array-constructor(expr)$,

is-Record-constructor (*expr*) $\rightarrow identifiers-used-in-record-constructor(expr)$,

is-Set-constructor (*expr*) $\rightarrow identifiers-used-in-set-constructor(expr)$

annotations The result is the set of qualified identifiers used in a value constructor; this set is either the qualified identifiers used in an array constructor, a record constructor, or a set constructor.

;

identifiers-used-in-array-constructor : *Array-constructor* \rightarrow *Qualident-set*

identifiers-used-in-array-constructor (*expr*) \triangleq

let *mk-Array-constructor* (*qid*, *sval*) = *expr* in

$\{qid\} \cup \bigcup \{identifiers-used-in-element(el) \mid el \in elems\ sval\}$

annotations The result is the qualified identifiers used in an array constructor; this set is the union of the qualified identifiers used in all the expression components of the array constructor together with the qualified identifier used to denote the array.

;

identifiers-used-in-record-constructor : *Record-constructor* \rightarrow *Qualident-set*

identifiers-used-in-record-constructor (*expr*) \triangleq

let *mk-Record-constructor* (*qid*, *sval*) = *expr* in

$\{qid\} \cup \bigcup \{identifiers-used-in-element(el) \mid el \in elems\ sval\}$

annotations The result is the set of qualified identifiers used in a record constructor; this set is the union of the qualified identifiers used in all the expression components of the record constructor.

;

identifiers-used-in-element : *Element* \rightarrow *Qualident-set*

identifiers-used-in-element (*el*) \triangleq

let *mk-Element* (*val*, *rep*) = *el* in

$identifiers-used-in-expression(val) \cup identifiers-used-in-expression(rep)$

70

```

annotations      The result is the set of qualified identifiers used in an element of a value constructor.
;

identifiers-used-in-set-constructor : Set-constructor  $\rightarrow$  Qualident-set
identifiers-used-in-set-constructor (expr)  $\triangleq$ 
  let mk-Set-constructor (qid, def) = expr in
73  {qid}  $\cup \bigcup \{ \textit{identifiers-used-in-member}(\textit{expr}) \mid \textit{expr} \in \textit{elems def} \}$ 

annotations      The result is the set of qualified identifiers used in a set constructor; this set is the union of the
                  qualified identifiers used in all the members of the set constructor together with the qualified
                  identifier used to denote the set.
;

identifiers-used-in-member : Member  $\rightarrow$  Qualident-set
identifiers-used-in-member (mem)  $\triangleq$ 
  cases mem :
70   mk-Singleton (expr)       $\rightarrow$  identifiers-used-in-expression (expr),
70   mk-Interval (min, max)  $\rightarrow$  identifiers-used-in-expression (min)  $\cup$  identifiers-used-in-expression (max)
  end

annotations      The result is the set of qualified identifiers used in a member; this set is the union of the
                  qualified identifiers used in the expression component(s) of the member.
;

identifiers-used-in-type : Type  $\rightarrow$  Qualident-set
identifiers-used-in-type (type)  $\triangleq$ 
  cases type :
    mk-Type-identifier (qid)       $\rightarrow$  {qid},
    mk-Enumerated-type ()            $\rightarrow$  {},
    mk-Subrange-type (rtype, range)  $\rightarrow$  let mk-Interval (min, max) = range in
                                         {rtype}  $\cup$ 
70                                         identifiers-used-in-expression (min)  $\cup$ 
70                                         identifiers-used-in-expression (max),
73   mk-Set-type (btype)            $\rightarrow$  identifiers-used-in-type (btype),
    mk-Pointer-type ()              $\rightarrow$  {},
    mk-Procedure-type ()            $\rightarrow$  {},
    mk-Function-procedure-type ()  $\rightarrow$  {},
73   mk-Array-type (itype, ctype)  $\rightarrow$  identifiers-used-in-type (itype)  $\cup$  identifiers-used-in-type (ctype),
73   mk-Record-type (fields)        $\rightarrow$  identifiers-used-in-fields-list (fields)
  end

annotations      The result is the set of qualified identifiers used in a type; the members of this set are the
                  qualified identifiers used to denote type names, any qualified identifiers that appear in any
                  constant expressions, and any qualified identifiers that occur to define parameters. The set
                  does not include those type qualified identifiers that are shielded either by POINTER TO or by
                  PROCEDURE.
;

identifiers-used-in-fields-list : Fields-list  $\rightarrow$  Qualident-set
identifiers-used-in-fields-list (fields)  $\triangleq$ 
  if fields = nil
  then { }
73  else  $\bigcup \{ \textit{identifiers-used-in-fields}(\textit{field}) \mid \textit{field} \in \textit{elems fields} \}$ 

annotations      The result is the set of qualified identifiers used in a fields list; these are the qualified identifiers
                  used in each of the components of the fields list.
;

```

```

identifiers-used-in-fields : Fields → Qualident-set
identifiers-used-in-fields (field)  $\triangleq$ 
  cases field :
73   mk-Fixed-fields (-, type)           → identifiers-used-in-type (type),
74   mk-Variant-fields (-, tagt, variants, elsep) → { tagt } ∪
75                                           identifiers-used-in-variants (variants) ∪
76                                           identifiers-used-in-fields-list (elsep)
  end
annotations      The result is the set of qualified identifiers used in fields; these are the qualified identifiers used
                     in each of the components of the fields.
;

identifiers-used-in-variants : Variant → Qualident-set
identifiers-used-in-variants (variants)  $\triangleq$ 
74   ∪ { identifiers-used-in-a-variant (variant) | variant ∈ elems variants }
annotations      The result is the set of qualified identifiers used in variants; these are the qualified identifiers
                     used in each of the components of the variants.
;

identifiers-used-in-a-variant : Variant → Qualident-set
identifiers-used-in-a-variant (variant)  $\triangleq$ 
  let mk-Variant (labels, fields) = variant in
70   identifiers-used-in-expression (labels) ∪ identifiers-used-in-fields-list (fields)
annotations      The result is the set of qualified identifiers used in a variant; these are the qualified identifiers
                     used in the labels component and fields component of the variant.
;

identifiers-used-in-parameters : Formal-parameter-type-list → Qualident-set
identifiers-used-in-parameters (parms)  $\triangleq$ 
74   ∪ { identifiers-used-in-a-parameter (parm) | parm ∈ elems parms }
annotations      The result is the set of qualified identifiers used in parameters; these are the qualified identifiers
                     used in each of the components of the parameters.
;

identifiers-used-in-a-parameter : Formal-parameter-type → Qualident-set
identifiers-used-in-a-parameter (parm)  $\triangleq$ 
  cases parm :
74   mk-Value-formal-type (fptype) → identifiers-used-in-formal-type (fptype),
74   mk-Variable-formal-type (fptype) → identifiers-used-in-formal-type (fptype)
  end
annotations      The result is the set of qualified identifiers used in a parameter; these are the qualified identifiers
                     used to denote type names in the formal type of the parameter.
;

identifiers-used-in-formal-type : Formal-type → Qualident-set
identifiers-used-in-formal-type (fptype)  $\triangleq$ 
  cases fptype :
73   mk-Parameter-formal-type (type) → identifiers-used-in-type (type),
75   mk-Array-formal-type (type) → identifiers-used-in-array-formal-type (type)
  end
annotations      The result is the set of qualified identifiers used in a formal type; these are the qualified
                     identifiers used to denote type names in either a parameter formal type or an array formal
                     type.

```

```

;

identifiers-used-in-array-formal-type : Array-formal-type  $\rightarrow$  Qualident-set
identifiers-used-in-array-formal-type (ftype)  $\triangleq$ 
  cases ftype :
    mk-Type-identifier (qid)  $\rightarrow$  { qid },
75    mk-Array-formal-type (type)  $\rightarrow$  identifiers-used-in-array-formal-type (type)
  end

annotations      The result is the set of qualified identifiers used in a formal type; these are the qualified
                   identifiers used to denote type names in an array formal type.

```

6.2.8.2.1 Shielded Identifiers

functions

```

shielded-in : Declarations  $\rightarrow$  Qualident-set
shielded-in (decls)  $\triangleq$ 
75     $\bigcup \{ \textit{shielded}(\textit{decl}) \mid \textit{decl} \in \textit{elems decls} \}$ 
annotations      The result of this operation is a set containing those (qualified) identifiers that are shielded in
                   a sequence of declarations. An identifier is shielded if it follows POINTER TO or PROCEDURE etc.
;

shielded : Declaration  $\rightarrow$  Qualident-set
shielded (decl)  $\triangleq$ 
  cases decl :
75    mk-Type-declaration (-, type)  $\rightarrow$  shielded-in-type (type),
75    mk-Variable-declaration (-, type)  $\rightarrow$  shielded-in-type (type),
    others  $\rightarrow$  { }
  end

annotations      The result of this operation is a set containing those (qualified) identifiers that are shielded in
                   a declaration.
;

shielded-in-type : Type  $\rightarrow$  Qualident-set
shielded-in-type (type)  $\triangleq$ 
  cases type :
    mk-Type-identifier ()  $\rightarrow$  { },
    mk-Enumerated-type ()  $\rightarrow$  { },
    mk-Subrange-type ()  $\rightarrow$  { },
    mk-Set-type (btype)  $\rightarrow$  { },
73    mk-Pointer-type (btype)  $\rightarrow$  identifiers-used-in-type (btype)  $\cup$  shielded-in-type (btype),
74    mk-Procedure-type (parms)  $\rightarrow$  identifiers-used-in-parameters (parms),
74    mk-Function-procedure-type (parms, return)  $\rightarrow$  identifiers-used-in-parameters (parms)  $\cup$  { return },
75    mk-Array-type (itype, ctype)  $\rightarrow$  shielded-in-type (ctype),
75    mk-Record-type (fields)  $\rightarrow$  shielded-in-fields-list (fields)
  end

annotations      Return a set of identifiers containing all the identifiers that are shielded in a type component
                   of a definition.
;

shielded-in-fields-list : Fields-list  $\rightarrow$  Qualident-set
shielded-in-fields-list (fields)  $\triangleq$ 
  if fields = nil
  then { }
76 else  $\bigcup \{ \textit{shielded-in-fields}(\textit{field}) \mid \textit{field} \in \textit{elems fields} \}$ 

```

```

annotations      Return the identifiers shielded in a fields list; these are the identifiers that are shielded in each
                  of the components of the field list.
;

shielded-in-fields : Fields → Qualident-set-set
shielded-in-fields field  $\triangleq$ 
  cases field :
75     mk-Fixed-fields (-, type)           → shielded-in-type (type),
76     mk-Variant-fields (-, -, variants, elsep) → shielded-in-variants (variants) ∪ shielded-in-fields-list (elsep)
  end

annotations      Return the identifiers shielded in fields.
;

shielded-in-variants : Variant* → Qualident-set
shielded-in-variants (variants)  $\triangleq$ 
76   ∪ {shielded-in-a-variant (variant) | variant ∈ elems variants}

annotations      Return the identifiers shielded in variants.
;

shielded-in-a-variant : Variant → Qualident-set
shielded-in-a-variant (variant)  $\triangleq$ 
  let mk-Variant (-, fields) = variant in
75   shielded-in-fields-list (fields)

annotations      Return the identifiers shielded in a variant.

```

6.3 Types

A type determines a set of values and, in the case of elementary types, a collection of fundamental operations that may be performed on these values.

Concrete Syntax

`type = type identifier | new type ;`

A type identifier is considered as an elementary type identifier or a structured type identifier, according to the type that it denotes.

Abstract Syntax

$Type = Type\text{-}identifier \mid New\text{-}type$

Declaration Semantics

functions

$d\text{-}type : Type \rightarrow Environment \rightarrow (Typed \times Environment)$
 $d\text{-}type (type) \rho \triangleq$
cases $type$:
78 $mk\text{-}Type\text{-}identifier (-) \rightarrow d\text{-}type\text{-}identifier (type) \rho,$
79 $mk\text{-}New\text{-}type (-) \rightarrow d\text{-}new\text{-}type (type) \rho$
end

Static Semantics

functions

$wf\text{-}type : Type \rightarrow Environment \rightarrow \mathbb{B}$
 $wf\text{-}type (type) \rho \triangleq$
cases $type$:
78 $mk\text{-}Type\text{-}identifier (-) \rightarrow wf\text{-}type\text{-}identifier (type) \rho,$
79 $mk\text{-}New\text{-}type (-) \rightarrow wf\text{-}new\text{-}type (type) \rho$
end

6.3.1 Type Identifiers

Concrete Syntax

`type identifier = qualified identifier ;`

Abstract Syntax

$Type\text{-}identifier :: qid : Qualident$

Declaration Semantics

functions

$$\begin{aligned}
& d\text{-type-identifier} : \text{Type-identifier} \rightarrow \text{Environment} \rightarrow (\text{Typed} \times \text{Environment}) \\
& d\text{-type-identifier}(\text{mk-Type-identifier}(qid))\rho \triangleq \\
271 \quad & \text{let } typed = \text{type-of}(qid)\rho \text{ in} \\
& \quad \text{mk-}(typed, \rho)
\end{aligned}$$

Static Semantics

The qualified identifier shall denote a type.

functions

$$\begin{aligned}
& wf\text{-type-identifier} : \text{Type-identifier} \rightarrow \text{Environment} \rightarrow \mathbb{B} \\
& wf\text{-type-identifier}(\text{mk-Type-identifier}(qid))\rho \triangleq \\
?? \quad & wf\text{-qualident}(qid)\rho \wedge \\
270 \quad & is\text{-type}(qid)\rho
\end{aligned}$$

Dynamic Semantics

operations

$$\begin{aligned}
& m\text{-typed} : \text{Typed} \rightarrow \text{Environment} \xrightarrow{o} \text{Variable} \\
& m\text{-typed}(type)\rho \triangleq \\
298 \quad & (is\text{-Basic-type}(type) \rightarrow \text{def } loc = \text{generate-location}(); \\
& \quad \quad \quad \text{mk-Elementary-variable}(loc, \text{nil}), \\
271 \quad & is\text{-Type-name}(type) \rightarrow \text{let } ts = \text{structure-of}(type)\rho \text{ in} \\
79 \quad & \quad \quad \quad m\text{-structure}(ts)\rho, \\
& is\text{-System-storage-type}(type) \rightarrow \text{is not yet defined})
\end{aligned}$$

annotations Note that type identifier have been replaced by type-names at this point.

6.3.2 New Types

A new type is the mechanism used for creating types; it defines the structure of values and variables of that type. Each occurrence of a new type denotes a type that is different from any other type.

Concrete Syntax

new type = enumeration type | subrange type | set type | pointer type | procedure type | array type | record type ;

Abstract Syntax

New-type = *Enumeration-type* | *Subrange-type* | *Set-type* | *Pointer-type* | *Procedure-type* | *Array-type* | *Record-type*

Declaration Semantics

functions

$$d\text{-new-type} : \text{New-type} \rightarrow \text{Environment} \rightarrow (\text{Typed} \times \text{Environment})$$

$$d\text{-new-type}(\text{type})\rho \triangleq$$

cases *type* :

82 $mk\text{-Subrange-type}(-) \rightarrow d\text{-subrange-type}(\text{type})\rho,$
 83 $mk\text{-Set-type}(-) \rightarrow d\text{-set-type}(\text{type})\rho,$
 84 $mk\text{-Pointer-type}(-) \rightarrow d\text{-pointer-type}(\text{type})\rho,$
 85 $mk\text{-Procedure-type}(-) \rightarrow d\text{-procedure-type}(\text{type})\rho,$
 92 $mk\text{-Array-type}(-) \rightarrow d\text{-array-type}(\text{type})\rho,$
 93 $mk\text{-Record-type}(-) \rightarrow d\text{-record-type}(\text{type})\rho$

end

Static Semantics

functions

$$wf\text{-new-type} : \text{New-type} \rightarrow \text{Environment} \rightarrow \mathbb{B}$$

$$wf\text{-new-type}(\text{type})\rho \triangleq$$

cases *type* :

82 $mk\text{-Subrange-type}(-) \rightarrow wf\text{-subrange-type}(\text{type})\rho,$
 83 $mk\text{-Set-type}(-) \rightarrow wf\text{-set-type}(\text{type})\rho,$
 84 $mk\text{-Pointer-type}(-) \rightarrow wf\text{-pointer-type}(\text{type})\rho,$
 85 $mk\text{-Procedure-type}(-) \rightarrow wf\text{-procedure-type}(\text{type})\rho,$
 92 $mk\text{-Array-type}(-) \rightarrow wf\text{-array-type}(\text{type})\rho,$
 93 $mk\text{-Record-type}(-) \rightarrow wf\text{-record-type}(\text{type})\rho$

end

Dynamic Semantics

operations

$$m\text{-structure} : \text{Structure} \rightarrow \text{Environment} \xrightarrow{o} \text{Variable}$$

$$m\text{-structure}(\text{struc})\rho \triangleq$$

81 $(\text{is-Enumerated-structure}(\text{struc}) \rightarrow m\text{-enumerated-structure}(\text{struc})\rho,$
 83 $\text{is-Subrange-structure}(\text{struc}) \rightarrow m\text{-subrange-type-structure}(\text{struc})\rho,$
 84 $\text{is-Set-structure}(\text{struc}) \rightarrow m\text{-set-structure}(\text{struc})\rho,$
 84 $\text{is-Pointer-structure}(\text{struc}) \rightarrow m\text{-pointer-structure}(\text{struc})\rho,$
 85 $\text{is-Procedure-structure}(\text{struc}) \rightarrow m\text{-procedure-structure}(\text{struc})\rho,$
 92 $\text{is-Array-structure}(\text{struc}) \rightarrow m\text{-array-structure}(\text{struc})\rho,$
 94 $\text{is-Record-type}(\text{struc}) \rightarrow m\text{-record-structure}(\text{type})\rho,$
 $\text{struc} = \text{OPAQUE} \rightarrow \text{is not yet defined})$

annotations

In the functions that follow, the declaration semantics translate a new type into a type structure in which any new type component of the new type has been replaced by a unique type name (a ‘hidden’ name which cannot be generated in a program text). Thus the ‘new types’ defined by some type declarations are each denoted by a name that is unique with respect to the complete program. Any type identifier occurring within a new type is replaced by its corresponding unique type name. The unique type name is associated with its structure by a mapping which is a component of the environment. Type identifiers are associated with their unique type names by a mapping that is another component of the environment. Thus an identifier denoting a type is mapped by a component of the environment into a unique type name, and that type name is mapped by a component of the environment into the structure of that type.

In the meaning functions, the meaning of a new type occurring in a type declaration is its type name and structure added to the environment. The meaning of a new type occurring in a variable declaration is its type name and structure added to the environment together with the necessary storage allocation occurring in the state. The meaning of a type identifier occurring in a type declaration is to just associate it with its type name.

During the processing of new types in the meaning function, the new types are translated to type structures and added to the environment and the necessary type name associated with that structure. The meaning of a type occurring in a variable declaration is to allocate storage for that type and associate that identifier with the storage by using a mapping in the environment. The structure of the type defined in the environment is used for this storage allocation, rather than the type information defined by the program text. All elementary variables are treated the same way, an elementary variable is allocated and associated with the identifier that is to denote that variable. Identifiers that have been declared as structured objects have their type name looked up in the environment to give their structure, and storage allocated for this structure. Thus there are no meaning functions for elementary types, (storage allocation is defined in the meaning function for a variable declaration), and the meaning functions for structured objects are defined on type structures.

6.3.2.1 Enumeration types

An enumeration type defines an ordered set of values by enumerating identifiers that denote the elements of the set. The ordering of these values is determined by the order in which the identifiers denoting the values are written, i.e. if x textually precedes y then x is less than y . The identifiers are used as constants in the scope of the declaration of the enumeration type. The ordinal number of a value of an enumeration type is determined by mapping the values of the type on to consecutive values of the unsigned type starting from zero.

Concrete Syntax

enumeration type = "(" identifier list, ")" ;

identifier list = identifier, { ",", identifier } ;

Abstract Syntax

Enumeration-type :: values : Identifier*

Declaration Semantics

An Enumeration type declaration includes the declaration of the identifier constants.

functions

$d\text{-enumeration-type} : \text{Enumeration-type} \rightarrow \text{Environment} \rightarrow (\text{Type-name} \times \text{Environment})$

$d\text{-enumeration-type} (mk\text{-Enumeration-type} (values))\rho \triangleq$

277 let $typenm = generate\text{-type-name} (\rho)$ in

81 let $\rho_{const} = add\text{-to-constants} (typenm, values)\rho$ in

let $enum = mk\text{-Enumerated-structure} (values)$ in

271 $mk\text{-}(typenm, overwrite\text{-struc-environment} (\{typenm \mapsto enum\})\rho_{const})$

annotations

A unique type name is generated to denote the new type that is being defined. The identifiers occurring in the identifier list are added to the environment as constants, each constant being associated with the value it denotes. The structure of the new type is also added to the environment.

Static Semantics

The identifiers occurring in the identifier list component of the enumeration type shall be distinct.

functions

$wf-enumeration-type : Enumeration-type \rightarrow Environment \rightarrow \mathbb{B}$
 $wf-enumeration-type (mk-Enumeration-type (values)) \rho \triangleq$
_{std} $is-unique (values)$

Dynamic Semantics

operations

$m-enumerated-structure : Enumerated-structure \rightarrow Environment \xrightarrow{o} Variable$
 $m-enumerated-structure (mk-Enumerated-structure (-)) \rho \triangleq$
₂₉₈ $def\ loc = generate-location();$
₂₇₀ $mk-Elementary-variable (loc, nil)$

Auxiliary Definitions

functions

$add-to-constants : Type-name \times Identifier^* \rightarrow Environment \rightarrow Environment$
 $add-to-constants (typen, values) \rho \triangleq$
 $let\ consts = \{id \mapsto mk-Constant-value (typen, mk-Enumerated-value (id, values)) \mid id \in elems\ values\}$ in
₂₇₀ $overwrite-const-environment (consts) \rho$

6.3.2.2 Subrange Types

A subrange type defines a type that is a subrange of an ordinal type. The declaration of such a type specifies the smallest and the largest values in the subrange. The associated ordinal type is called the range type of the subrange type, and the operations applicable to values of the range type are applicable to operands of the subrange type. A value to be assigned to a variable of a subrange type must lie within the interval specified by the smallest and largest value.

Concrete Syntax

subrange type = [range type], left bracket, constant expression, "...", constant expression, right bracket ;

range type = ordinal type identifier ;

CHANGE — A subrange of the host type may be specified.

The range type may be specified by a qualified identifier preceding the bounds. If it is omitted, the translator shall determine the type according to the following table, which gives the range type of the subrange type [M..N].

		N			
		unsigned type	signed type	ZZ-type	the type T
M	unsigned type	unsigned type	error	unsigned type	error
	signed type	error	signed type	signed type	error
	ZZ-type	unsigned type	signed type	M<0 signed type M≥0 unsigned type	error
	some other type T	error	error	error	the type T

NOTE — If the range type is not a subrange type, the host type of the subrange type is the range type; otherwise it is the host type of the subrange type.

Language Clarification The range type may itself be a subrange type.

Abstract Syntax

Subrange-type :: *rtype* : *Qualident*
 range : *Interval*

annotations If the range type is omitted, the range type of the subrange is supplied by the translator using the table given above.

Declaration Semantics

functions

d-subrange-type : *Subrange-type* \rightarrow *Environment* \rightarrow (*Type-name* \times *Environment*)
d-subrange-type (*mk-Subrange-type* (*rtype*, *range*)) $\rho \triangleq$
271 let *rtypen* = *type-of* (*rtype*) ρ in
218 let *values* = *evaluate-constant-expression* (*range*) ρ in
 let *sub* = *mk-Subrange-structure* (*rtypen*, *values*) in
277 let *typenm* = *generate-type-name* (ρ) in
271 *mk-*(*typenm*, *overwrite-struc-environment* ($\{ \textit{typenm} \mapsto \textit{sub} \}$)) ρ

annotations The range type of the subrange is determined, and the range is evaluated. A unique type name is generated to denote the new type that is being defined and the structure of the new type added to the environment.

Static Semantics

The range type of the subrange type shall be an ordinal type. Both constant expressions shall be of this same ordinal type, and the values defined by the subrange shall be a subset of the values defined by the range type.

The value of the first constant expression of a subrange type specifies the smallest value, and this shall be less than or equal to the value of the second constant expression which specifies the largest value of the subrange type.

functions

wf-subrange-type : *Subrange-type* \rightarrow *Environment* $\rightarrow \mathbb{B}$
wf-subrange-type (*mk-Subrange-type* (*rtype*, *range*)) $\rho \triangleq$
?? *wf-qualident* (*rtype*) $\rho \wedge$
280 *is-ordinal-type* (*rtype*) $\rho \wedge$
205 *wf-interval* (*range*) ρ
 \wedge
214 *is-constant-expression* (*range*) $\rho \wedge$
218 let *cvalues* = *evaluate-constant-expression* (*range*) ρ in
290 let *rvalues* = *values-of* (*type-of* (*rtype*) ρ) ρ in
 cvalues $\neq \{ \}$ \wedge
 cvalues \subseteq *rvalues*

Dynamic Semantics

operations

$$m\text{-subrange-type-structure} : \text{Subrange-type-structure} \rightarrow \text{Environment} \xrightarrow{o} \text{Variable}$$

$$m\text{-subrange-type-structure} (mk\text{-Subrange-type-structure} (-, range))\rho \triangleq$$

```

205   def values = m-interval(range) ρ;
298   def loc = generate-location();
   return mk-Elementary-variable (loc, values)

```

6.3.2.3 Set Types

A set type determines the set of values that is structured as the power set of a type, called the base type of the set type. Thus each value of a set type is a set whose members are unique values of its base type.

Concrete Syntax

set type = "SET", "OF", base type ;

base type = ordinal type ;

ordinal type = ordinal type identifier | enumeration type | subrange type ;

ordinal type identifier = type identifier ;

Abstract Syntax

Set-type :: *btype* : *Type*

Declaration Semantics

functions

$$d\text{-set-type} : \text{Set-type} \rightarrow \text{Environment} \rightarrow (\text{Type-name} \times \text{Environment})$$

$$d\text{-set-type} (mk\text{-Set-type} (btype))\rho \triangleq$$

```

77   let mk- (btypen, ρtenv) = d-type (btype) ρ in
   let set = mk-Set-structure (btypen) in
277   let typenm = generate-type-name (ρtenv) in
271   mk- (typenm, overwrite-struct-environment ({typenm ↦ set}) ρtenv)

```

annotations The base type of the set type is elaborated, a unique type name is generated to denote the new type that is being defined, and the structure of the new type added to the environment.

Static Semantics

The base type of the set type shall be an ordinal type.

functions

$$wf\text{-set-type} : \text{Set-type} \rightarrow \text{Environment} \rightarrow \mathbb{B}$$

$$wf\text{-set-type} (mk\text{-Set-type} (btype))\rho \triangleq$$

```

77   wf-type (btype) ρ ∧
280   is-ordinal-type (btype) ρ

```

Dynamic Semantics

operations

$$\begin{aligned} m\text{-set-structure} &: \text{Set-structure} \rightarrow \text{Environment} \xrightarrow{o} \text{Variable} \\ m\text{-set-structure} (mk\text{-Set-structure} (-))\rho &\triangleq \\ 298 \quad \text{def } loc &= \text{generate-location}() ; \\ \text{return } &mk\text{-Elementary-variable} (loc, nil) \end{aligned}$$

6.3.2.4 Pointer Types

A pointer type determines a set of values which may be used to reference variables of another type; a pointer to type **T** is said to be bound to **T**. A pointer value is obtained by a call to an allocation procedure in a storage management module. The pervasive identifier **NIL** denotes the nil-value for all pointer types.

Concrete Syntax

pointer type = "POINTER", "T0", bound type ;
bound type = type ;

Abstract Syntax

Pointer-type :: *btype* : *Type*

Declaration Semantics

functions

$$\begin{aligned} d\text{-pointer-type} &: \text{Pointer-type} \rightarrow \text{Environment} \rightarrow (\text{Type-name} \times \text{Environment}) \\ d\text{-pointer-type} (mk\text{-Pointer-type} (btype))\rho &\triangleq \\ 77 \quad \text{let } mk\text{-}(typen, \rho_{type}) &= d\text{-type} (btype)\rho \text{ in} \\ \text{let } point &= mk\text{-Pointer-structure} (typen) \text{ in} \\ 277 \quad \text{let } typenm &= \text{generate-type-name} (\rho_{type}) \text{ in} \\ 271 \quad mk\text{-}(typenm, &\text{overwrite-struc-environment} (\{ typenm \mapsto point \}))\rho_{type} \end{aligned}$$

annotations The bound type is elaborated, a unique type name is generated to denote the new type being defined, and the structure of the new type added to the environment.

Static Semantics

The identifier that denotes the type to which the pointer is bound shall denote a type.

functions

$$\begin{aligned} wf\text{-pointer-type} &: \text{Pointer-type} \rightarrow \text{Environment} \rightarrow \mathbb{B} \\ wf\text{-pointer-type} (mk\text{-Pointer-type} (btype))\rho &\triangleq \\ 77 \quad wf\text{-type} (btype)\rho \end{aligned}$$

Dynamic Semantics

operations

$$\begin{aligned} m\text{-pointer-structure} &: \text{Pointer-structure} \rightarrow \text{Environment} \xrightarrow{o} \text{Variable} \\ m\text{-pointer-structure} (mk\text{-Pointer-structure} (-))\rho &\triangleq \\ 298 \quad \text{def } loc &= \text{generate-location}() ; \end{aligned}$$

return *mk-Elementary-variable* (*loc*, *nil*)

6.3.2.5 Procedure Types

A procedure type defines the structural specification of the number and types of parameters of procedures of that type and, if it is a function procedure type, the type of the result.

A variable of a procedure type may be assigned a procedure value, denoted by a procedure identifier. In this case the types of the formal parameters of a procedure value shall be the same as those defined in the formal type list of the procedure type and the procedure shall be declared at level 0 (see 6.4.1).

Concrete Syntax

procedure type = proper procedure type | function procedure type ;

Abstract Syntax

Procedure-type = *Proper-procedure-type* | *Function-procedure-type*

Declaration Semantics

functions

$d\text{-procedure-type} : \text{Type} \rightarrow \text{Environment} \rightarrow (\text{Typed} \times \text{Environment})$

$d\text{-procedure-type}(\text{type})\rho \triangleq$

cases *type* :

86 $mk\text{-Proper-procedure-type}(-) \rightarrow d\text{-proper-procedure-type}(\text{type})\rho,$

87 $mk\text{-Function-procedure-type}(-) \rightarrow d\text{-function-procedure-type}(\text{type})\rho$

end

Static Semantics

functions

$wf\text{-procedure-type} : \text{Type} \rightarrow \text{Environment} \rightarrow \mathbb{B}$

$wf\text{-procedure-type}(\text{type})\rho \triangleq$

cases *type* :

86 $mk\text{-Proper-procedure-type}(-) \rightarrow wf\text{-proper-procedure-type}(\text{type})\rho,$

87 $mk\text{-Function-procedure-type}(-) \rightarrow wf\text{-function-procedure-type}(\text{type})\rho$

end

Dynamic Semantics

operations

$m\text{-procedure-structure} : \text{Procedure-structure} \rightarrow \text{Environment} \xrightarrow{o} \text{Variable}$

$m\text{-procedure-structure}(\text{struc})\rho \triangleq$

cases *struc* :

86 $mk\text{-Proper-procedure-structure}(-) \rightarrow m\text{-proper-procedure-structure}(\text{struc})\rho,$

87 $mk\text{-Function-procedure-structure}(-) \rightarrow m\text{-function-procedure-structure}(\text{struc})\rho$

end

Proper Procedure Type

A proper procedure type defines the structural specification of the number and types of parameters of a procedure.

Concrete Syntax

proper procedure type = "PROCEDURE", [formal parameter type list] ;

Abstract Syntax

Proper-procedure-type :: *parms* : *Formal-parameter-type-list*

Declaration Semantics

functions

```
d-proper-procedure-type : Proper-procedure-type → Environment → (Proper-procedure-typed × Environment)
d-proper-procedure-type (mk-Proper-procedure-type (parms)) ρ  $\triangleq$ 
88   let nparms = d-formal-parameter-type-list (parms) ρ in
      let proc = mk-Proper-procedure-structure (nparms) in
277   let typenm = generate-type-name (ρ) in
271   mk- (typenm, overwrite-struc-environment ({ typenm ↦ proc }) ρ)
```

annotations The type of each of the parameters is elaborated, a unique type name is generated to denote the new type that is being defined, and the structure of the new type added to the environment.

Static Semantics

functions

```
wf-proper-procedure-type : Proper-procedure-type → Environment →  $\mathbb{B}$ 
wf-proper-procedure-type (mk-Proper-procedure-type (parms)) ρ  $\triangleq$ 
88    $\forall \text{parm} \in \text{elems } \text{parms} \cdot \text{wf-formal-parameter-type } (\text{parm}) \rho$ 
```

Dynamic Semantics

operations

```
m-proper-procedure-structure : Proper-procedure-structure → Environment  $\xrightarrow{o}$  Variable
m-proper-procedure-structure (mk-Proper-procedure-structure (-)) ρ  $\triangleq$ 
298   def loc = generate-location() ;
      return mk-Elementary-variable (loc, nil )
```

Function Procedure Type

A function procedure type defines the structural specification of the number and types of parameters and the type of the result.

Concrete Syntax

function procedure type = "PROCEDURE", formal parameter type list, function result type ;

function result type = ":", type identifier ;

Abstract Syntax

Function-procedure-type :: *parms* : *Formal-parameter-type-list*
 return : *Type-identifier*

Declaration Semantics

functions

$$\begin{aligned}
 & d\text{-function-procedure-type} : \text{Function-procedure-type} \rightarrow \text{Environment} \rightarrow (\text{Function-procedure-typed} \times \text{Environment}) \\
 & d\text{-function-procedure-type} (mk\text{-Function-procedure-type} (parms, retrn))\rho \triangleq \\
 88 \quad & \text{let } nparms = d\text{-formal-parameter-type-list} (parms)\rho \text{ in} \\
 77 \quad & \text{let } mk\text{-}(nreturn, -) = d\text{-type} (retrn)\rho \text{ in} \\
 & \text{let } fun = mk\text{-Function-procedure-structure} (nparms, nreturn) \text{ in} \\
 277 \quad & \text{let } typenm = generate\text{-type-name} (\rho) \text{ in} \\
 271 \quad & mk\text{-}(typenm, overwrite\text{-struc-environment} (\{typenm \mapsto fun\})\rho)
 \end{aligned}$$

annotations The type of each of the parameters is elaborated, the type of the returned value is elaborated, a unique type name is generated to denote the new type that is being defined, and the structure of the new type added to the environment.

Static Semantics

functions

$$\begin{aligned}
 & wf\text{-function-procedure-type} : \text{Function-procedure-type} \rightarrow \text{Environment} \rightarrow \mathbb{B} \\
 & wf\text{-function-procedure-type} (mk\text{-Function-procedure-type} (parms, retrn))\rho \triangleq \\
 77 \quad & wf\text{-type} (retrn)\rho \wedge \\
 88 \quad & \forall parm \in \text{elems } parms \cdot wf\text{-formal-parameter-type} (parm)\rho
 \end{aligned}$$

Language Clarification There is no restriction on the type of the returned value.

Dynamic Semantics

operations

$$\begin{aligned}
 & m\text{-function-procedure-structure} : \text{Function-procedure-structure} \rightarrow \text{Environment} \xrightarrow{o} \text{Variable} \\
 & m\text{-function-procedure-structure} (mk\text{-Function-procedure-structure} (-))\rho \triangleq \\
 298 \quad & \text{def } loc = generate\text{-location}(); \\
 & mk\text{-Elementary-variable} (loc, nil)
 \end{aligned}$$

Formal Parameter Type Lists

A formal parameter type list defines the structural specification of the number and types of parameters of a procedure.

Concrete Syntax

formal parameter type list = "(" [formal parameter type { ", ", formal parameter type }], ")" ;

formal parameter type = variable formal type | value formal type ;

Abstract Syntax

Formal-parameter-type-list = *Formal-parameter-type**

Formal-parameter-type = *Variable-formal-type* | *Value-formal-type*

Declaration Semantics

functions

```
d-formal-parameter-type-list : Formal-parameter-type-list → Environment → Formal-parameters-typed  
d-formal-parameter-type-list (fps) $\rho \triangleq$   
  if fps = []  
  then []  
88   else [d-formal-parameter-type (hd fps) $\rho$ ]  $\curvearrowright$  d-formal-parameter-type-list (tl fps) $\rho$ ;  
  
d-formal-parameter-type : Formal-parameter-type → Environment → (Typed × Environment)  
d-formal-parameter-type (type) $\rho \triangleq$   
  cases type :  
88     mk-Variable-formal-type (-) → d-variable-formal-type (type) $\rho$ ,  
89     mk-Value-formal-type (-)    → d-value-formal-type (type) $\rho$   
  end
```

Static Semantics

functions

```
wf-formal-parameter-type : Formal-parameter-type → Environment →  $\mathbb{B}$   
wf-formal-parameter-type (type) $\rho \triangleq$   
  cases type :  
89     mk-Variable-formal-type (-) → wf-variable-formal-type (type) $\rho$ ,  
89     mk-Value-formal-type (-)    → wf-value-formal-type (type) $\rho$   
  end
```

Variable Formal Types

A variable formal parameter type defines the structural specification of the type of a variable parameter of a procedure.

Concrete Syntax

variable formal type = "VAR", formal type ;

Abstract Syntax

Variable-formal-type :: *ftype* : *Formal-type*

Declaration Semantics

functions

```
d-variable-formal-type : Variable-formal-type → Environment → Variable-formal-typed  
d-variable-formal-type (mk-Variable-formal-type (ftype)) $\rho \triangleq$   
90   mk-Variable-formal-typed (d-formal-type (ftype) $\rho$ )
```

annotations Translate the formal type to its type name.

Static Semantics

functions

$wf\text{-}variable\text{-}formal\text{-}type : Variable\text{-}formal\text{-}type \rightarrow Environment \rightarrow \mathbb{B}$

$wf\text{-}variable\text{-}formal\text{-}type (mk\text{-}Variable\text{-}formal\text{-}type (ftype))\rho \triangleq$

90 $wf\text{-}formal\text{-}type (ftype)\rho$

Dynamic Semantics

operations

$m\text{-}variable\text{-}formal\text{-}type : Variable\text{-}formal\text{-}type \rightarrow Environment \xrightarrow{o} Variable\text{-}formal\text{-}typed$

$m\text{-}variable\text{-}formal\text{-}type (mk\text{-}Variable\text{-}formal\text{-}type (ftype))\rho \triangleq$

90 $\text{return } mk\text{-}Variable\text{-}formal\text{-}typed (d\text{-}formal\text{-}type (ftype)\rho)$

annotations Translate the formal type to its type name.

Value Formal Types

A value formal parameter type defines the structural specification of the type of a value parameter of a procedure.

Concrete Syntax

value formal type = formal type ;

Abstract Syntax

$Value\text{-}formal\text{-}type :: ftype : Formal\text{-}type$

Declaration Semantics

functions

$d\text{-}value\text{-}formal\text{-}type : Value\text{-}formal\text{-}type \rightarrow Environment \rightarrow Value\text{-}formal\text{-}typed$

$d\text{-}value\text{-}formal\text{-}type (mk\text{-}Value\text{-}formal\text{-}type (ftype))\rho \triangleq$

90 $mk\text{-}Value\text{-}formal\text{-}typed (d\text{-}formal\text{-}type (ftype)\rho)$

annotations Translate the formal type to its type name.

Static Semantics

functions

$wf\text{-}value\text{-}formal\text{-}type : Value\text{-}formal\text{-}type \rightarrow Environment \rightarrow \mathbb{B}$

$wf\text{-}value\text{-}formal\text{-}type (mk\text{-}Value\text{-}formal\text{-}type (ftype))\rho \triangleq$

90 $wf\text{-}formal\text{-}type (ftype)\rho$

Dynamic Semantics

operations

$m\text{-}value\text{-}formal\text{-}type : Value\text{-}formal\text{-}type \rightarrow Environment \xrightarrow{o} Value\text{-}formal\text{-}typed$

$m\text{-}value\text{-}formal\text{-}type (mk\text{-}Value\text{-}formal\text{-}type (ftype))\rho \triangleq$

90 $\text{return } mk\text{-}Value\text{-}formal\text{-}typed (d\text{-}formal\text{-}type (ftype)\rho)$

annotations Translate the formal type to its type name.

Formal Types

A formal type defines the structural specification of the type of a parameter of a procedure.

Concrete Syntax

formal type = type identifier | open array formal type ;

open array formal type = "ARRAY", "OF", { "ARRAY", "OF" }, type identifier ;

CHANGE — This International Standard permits multidimensional open array parameters.

Abstract Syntax

Formal-type = *Parameter-formal-type* | *Array-formal-type*

Parameter-formal-type :: *type* : *Type-identifier*

Declaration Semantics

functions

d-formal-type : *Formal-type* → *Environment* → (*Typed* × *Environment*)

d-formal-type (*type*) $\rho \triangleq$

cases *type* :

90 *mk-Parameter-formal-type* (-) → *d-parameter-formal-type* (*type*) ρ ,

91 *mk-Array-formal-type* (-) → *d-array-formal-type* (*type*) ρ

end;

d-parameter-formal-type : *Parameter-formal-type* → *Environment* → *Variable-typed*

d-parameter-formal-type (*mk-Parameter-formal-type* (*type*)) $\rho \triangleq$

77 let *mk-(atype, -)* = *d-type* (*type*) ρ in

atype

annotations Translate the type to its type name.

Static Semantics

functions

wf-formal-type : *Formal-type* → *Environment* → \mathbb{B}

wf-formal-type (*type*) $\rho \triangleq$

cases *type* :

90 *mk-Parameter-formal-type* (-) → *wf-parameter-formal-type* (*type*) ρ ,

91 *mk-Array-formal-type* (-) → *wf-array-formal-type* (*type*) ρ

end;

wf-parameter-formal-type : *Parameter-formal-type* → *Environment* → \mathbb{B}

wf-parameter-formal-type (*mk-Parameter-formal-type* (*type*)) $\rho \triangleq$

77 *wf-type* (*type*) ρ

Dynamic Semantics

annotations The dynamic semantics of formal types is the same as (and is thus given by) the declaration semantics.

Abstract Syntax

Array-formal-type :: *type* : *Open-component-type*

Open-component-type = *Type-identifier* | *Array-formal-type*

Declaration Semantics

functions

d-array-formal-type : *Array-formal-type* → *Environment* → *Variable-typed*

d-array-formal-type (*mk-Array-formal-type* (*type*)) $\rho \triangleq$
91 let *mk*-(*typen*, -) = *d-open-component-type* (*type*) ρ in
 mk-Open-array-typed (*typen*);

d-open-component-type : *Open-component-type* → *Environment* → (*Typed* × *Environment*)

d-open-component-type (*type*) $\rho \triangleq$
 cases *type* :
78 *mk-Type-identifier* (-) → *d-type-identifier* (*type*) ρ ,
91 *mk-Array-formal-type* (-) → *d-array-formal-type* (*type*) ρ
 end

annotations Translate the type to its type name.

Static Semantics

functions

wf-array-formal-type : *Array-formal-type* → *Environment* → \mathbb{B}

wf-array-formal-type (*mk-Array-formal-type* (*type*)) $\rho \triangleq$
91 *wf-open-component-type* (*type*) ρ ;

wf-open-component-type : *Open-component-type* → *Environment* → (*Typed* × *Environment*)

wf-open-component-type (*type*) $\rho \triangleq$
 cases *type* :
78 *mk-Type-identifier* (-) → *wf-type-identifier* (*type*) ρ ,
91 *mk-Array-formal-type* (-) → *wf-array-formal-type* (*type*) ρ
 end

Dynamic Semantics

annotations The dynamic semantics of formal types is the same as (and is thus given by) the declaration semantics.

6.3.2.6 Array Types

An array type is structured as a mapping from a set of values defined by its index type onto distinct components. Each one of these components has the same type as the component type of the array type.

Concrete Syntax

array type = "ARRAY", index type, { ",", index type }, "OF", component type ;

index type = ordinal type ;

component type = type ;

The specification of an array type by using a sequence of two or more index types is an abbreviated notation for an array type specified to have as its index type the first index type in the sequence, and to have a component type that is an array type using the sequence of index types with the original first index type removed and with the same component type as the original specification. The abbreviated form and the full form shall be equivalent.

Abstract Syntax

Array-type :: *itype* : *Type*
 ctype : *Type*

Declaration Semantics

functions

$d\text{-array-type} : \text{Array-type} \rightarrow \text{Environment} \rightarrow (\text{Type-name} \times \text{Environment})$
 $d\text{-array-type}(\text{mk-Array-type}(\text{itype}, \text{ctype}))\rho \triangleq$
77 let $\text{mk-}(t_{ni}, \rho_1) = d\text{-type}(\text{itype})\rho$ in
77 let $\text{mk-}(t_{nc}, \rho_2) = d\text{-type}(\text{ctype})\rho_1$ in
 let $\text{array} = \text{mk-Array-structure}(t_{ni}, t_{nc})$ in
277 let $\text{typenm} = \text{generate-type-name}(\rho_2)$ in
271 $\text{mk-}(\text{typenm}, \text{overwrite-struct-environment}(\{\text{typenm} \mapsto \text{array}\})\rho_2)$

annotations The types of the index and components are elaborated, a unique type name is generated to denote the new type that is being defined, and the structure of the new type added to the environment.

Static Semantics

The index type shall be an ordinal type.

functions

$wf\text{-array-type} : \text{Array-type} \rightarrow \text{Environment} \rightarrow \mathbb{B}$
 $wf\text{-array-type}(\text{mk-Array-type}(\text{itype}, \text{ctype}))\rho \triangleq$
280 $\text{is-ordinal-type}(\text{itype})\rho \wedge$
77 $wf\text{-type}(\text{itype})\rho \wedge$
77 $wf\text{-type}(\text{ctype})\rho$

Dynamic Semantics

The elaboration of an array type shall result in an array variable, which associates with each value in the set specified by its index type, a variable whose type is specified by the component type of the array.

operations

$m\text{-array-structure} : \text{Array-structure} \rightarrow \text{Environment} \xrightarrow{o} \text{Array-variable}$
 $m\text{-array-structure}(\text{mk-Array-structure}(\text{itype}, \text{ctype}))\rho \triangleq$
290 let $\text{array} = \text{values-of}(\text{itype})\rho$ in
78 return $\text{mk-Array-variable}(\{d \mapsto m\text{-typed}(\text{ctype})\rho \mid d \in \text{array}\})$

annotations The result is an array variable, which is a mapping from each value in the set specified by its index type to a distinct variable whose type is specified by the component type of the array.

6.3.2.7 Record Types

A record type specifies a collection of components which may be of different types. Each component is called a field and is selected by a name. The structure and values of a record type are specified by the structure and values of the field lists of the record type. Some components called variants may optionally be present depending on the values of other components called tags.

TO DO — D92 did not allow empty records, these are now allowed except that operations imported from **SYSTEM** may not be used. The VDM will need to be updated to reflect this.

Concrete Syntax

record type = "RECORD", [field list] , "END" ;

field list = fields, { ";", fields } ;

Abstract Syntax

Record-type :: *fields* : *Field-list*

Field-list = *Fields**

Declaration Semantics

functions

$d\text{-record-type} : \text{Record-type} \rightarrow \text{Environment} \rightarrow (\text{Type-name} \times \text{Environment})$

$d\text{-record-type} (mk\text{-Record-type} (fields))\rho \triangleq$

$_{94} \quad \text{record-declaration-environment} (fields)\rho$

Static Semantics

All the fields identifiers of a record type shall be distinct.

functions

$wf\text{-record-type} : \text{Record-type} \rightarrow \text{Environment} \rightarrow \mathbb{B}$

$wf\text{-record-type} (mk\text{-Record-type} (fields))\rho \triangleq$

$_{98} \quad \text{is-distinct-identifiers} (fields) \wedge$

$_{93} \quad wf\text{-field-list} (fields)\rho;$

$wf\text{-field-list} : \text{Fields}^* \rightarrow \text{Environment} \rightarrow \mathbb{B}$

$wf\text{-field-list} (fieldss)\rho \triangleq$

$_{77} \quad \forall fields \in \text{elems } fieldss \cdot wf\text{-fieldss} (fields)\rho$

Dynamic Semantics

The elaboration of a record type shall result in a record variable that associates each field identifier to a variable whose type shall be the same as the type associated with the field identifier.

operations

$m\text{-record-structure} : \text{Record-structure} \rightarrow \text{Environment} \xrightarrow{o} \text{Record-variable}$

$m\text{-record-structure}(mk\text{-Record-structure}(fields))\rho \triangleq$
 $record\text{-declaration}(fields)\rho$

Auxiliary Functions

functions

$record\text{-declaration-environment} : \text{Field-list} \rightarrow \text{Environment} \rightarrow (\text{Type-name} \times \text{Environment})$

$record\text{-declaration-environment}(fields)\rho \triangleq$
 98 let $mk\text{-}(nfields, \rho_1) = build\text{-field-list-structure}(fields)\rho$ in
 let $struc = mk\text{-Record-structure}(nfields)$ in
 277 let $typenm = generate\text{-type-name}(\rho_1)$ in
 271 $mk\text{-}(typenm, overwrite\text{-struc-environment}(\{typenm \mapsto struc\})\rho_1)$

annotations The type of each of the components of the record is elaborated, a unique type name is generated to denote the new type that is being defined, and the structure of the new type added to the environment.

$record\text{-declaration} : \text{Field-list} \rightarrow \text{Environment} \xrightarrow{o} \text{Record-variable}$

$record\text{-declaration}(fieldss)\rho \triangleq$
 95 def $var = merge \{m\text{-fields-structure}(fieldss(i))\rho \mid i \in \text{inds } fieldss\};$
 return $mk\text{-Record-variable}(var)$

annotations The result is a record variable that consists of a mapping from the identifier field names to the associated storage.

Fixed Fields

Concrete Syntax

fields = fixed fields | variant fields ;

fixed fields = identifier list, ":", type ;

Abstract Syntax

$Fields = Fixed\text{-fields} \mid Variant\text{-fields}$

$Fixed\text{-fields} :: ids : Identifier^*$
 $type : Type$

Declaration Semantics

functions

$d\text{-fields} : Fields \rightarrow \text{Environment} \rightarrow (\text{Typed} \times \text{Environment})$

$d\text{-fields}(fields)\rho \triangleq$
 cases $fields$:
 94 $mk\text{-Fixed-fields}(-, -) \rightarrow d\text{-fixed-fields}(fields)\rho,$
 96 $mk\text{-Variant-fields}(-, -, -, -) \rightarrow d\text{-variant-fields}(fields)\rho$
 end;

$d\text{-fixed-fields} : \text{Fixed-fields} \rightarrow \text{Environment} \rightarrow (\text{Fixed-fields-structure} \times \text{Environment})$

$d\text{-fixed-fields} (mk\text{-Fixed-fields} (ids, type))\rho \triangleq$

77 $\text{let } mk\text{-}(tn, \rho_1) = d\text{-type}(type)\rho \text{ in}$
 $mk\text{-}(mk\text{-Fixed-fields-structure}(ids, tn), \rho_1)$

annotations The type of the fixed fields component of the record is elaborated, a

unique type name is generated to denote any new type that may be defined; the result is the structure of the component.

Static Semantics

The type shall be already defined, or shall be an (anonymous) new-type.

functions

$wf\text{-fields} : \text{Fields} \rightarrow \text{Environment} \rightarrow (\text{Typed} \times \text{Environment})$

$wf\text{-fields} (fields)\rho \triangleq$

cases $fields$:

95 $mk\text{-Fixed-fields}(-, -) \rightarrow wf\text{-fixed-fields}(fields)\rho,$

96 $mk\text{-Variant-fields}(-, -, -, -) \rightarrow wf\text{-variant-fields}(fields)\rho$

end;

$wf\text{-fixed-fields} : \text{Fixed-fields} \rightarrow \text{Environment} \rightarrow \mathbb{B}$

$wf\text{-fixed-fields} (mk\text{-Fixed-fields} (ids, type))\rho \triangleq$

77 $wf\text{-type}(type)\rho$

Dynamic Semantics

operations

$m\text{-fields-structure} : \text{Fields-structure} \rightarrow \text{Environment} \xrightarrow{o} \text{Variable}$

$m\text{-fields-structure} (struc)\rho \triangleq$

cases $struc$:

95 $mk\text{-Fixed-fields-structure}(-, -) \rightarrow m\text{-fixed-fields-structure}(struc)\rho,$

97 $mk\text{-Variant-fields-structure}(-, -, -, -) \rightarrow m\text{-variant-fields-structure}(struc)\rho$

end;

$m\text{-fixed-fields-structure} : \text{Fixed-fields-structure} \rightarrow \text{Environment} \xrightarrow{o} \text{Record-components}$

$m\text{-fixed-fields-structure} (mk\text{-Fixed-fields-structure} (ids, type))\rho \triangleq$

78 $\text{return } \{id \mapsto m\text{-typed}(type)\rho \mid id \in \text{elems } ids\}$

Variant Fields

Some components of a record type, called variants, may optionally be present, depending on the value of other components, called tags.

Concrete Syntax

variant fields = "CASE", [tag identifier], ":", tag type, "OF", variant list, "END" ;

tag identifier = ordinal type identifier ;

tag type = ordinal type ;

variant list = variant, { case separator, variant }, ["ELSE", field list], ;

Abstract Syntax

Variant-fields :: tag : [Identifier]
tag : Qualident
variants : Variant*
elsep : [Field-list]

Declaration Semantics

functions

$d\text{-variant-fields} : \text{Variant-fields} \rightarrow \text{Environment} \rightarrow (\text{Variant-fields-structure} \times \text{Environment})$

$d\text{-variant-fields}(\text{mk- Variant-fields}(tag, tagt, variants, elsep))\rho \triangleq$

271 let $tagt = \text{type-of}(tagt)\rho$ in
99 let $\text{mk-}(nvariants, \rho_2) = \text{build-variant-structure}(variants)\rho_1$ in
98 let $\text{mk-}(elseps, \rho_3) = \text{build-field-list-structure}(elsep)\rho$ in
mk- ($\text{mk- Variant-fields-structure}(tag, tagt, nvariants, elseps), \rho_3$)

annotations The type of the tag component of the variant field is elaborated. The structure of the variant components, and the ‘ELSE’ component are elaborated; the result is the structure of the component.

Static Semantics

The type of the tag field shall be an ordinal type.

The case labels of a case label list of a variant shall be constant expressions. The host type of the type of the tag field and the type of each of the case labels shall be identical, and the set of values defined by each of the case label lists shall be distinct from each other.

functions

$wf\text{-variant-fields} : \text{Variant-fields} \rightarrow \text{Environment} \rightarrow \mathbb{B}$

$wf\text{-variant-fields}(\text{mk- Variant-fields}(tag, tagt, variants, elsep))\rho \triangleq$

?? $wf\text{-qualident}(tagt)\rho \wedge$
271 let $type = \text{type-of}(tagt)\rho$ in
280 $is\text{-ordinal-type}(type)\rho \wedge$
 $\forall \text{variant} \in \text{elems } variants .$
97 $wf\text{-variant}(\text{variant})\rho \wedge$
 $(elsep = \text{nil} \vee$
93 $wf\text{-field-list}(elsep)\rho) \wedge$
let $labels = \{ \text{variant.labels}\rho \mid \text{variant} \in \text{elems } variants \}$ in
152 $\forall lab \in labels . t\text{-expression}(lab)\rho = type$

Dynamic Semantics

operations

$m\text{-variant-fields-structure} : \text{Variant-fields-structure} \rightarrow \text{Environment} \xrightarrow{o} \text{Record-components}$

$m\text{-variant-fields-structure } \text{Variant-fields-structure}(tag, tagt, variants, elsep) \rho \triangleq$
if $tag = \text{nil}$

99 then $allocate\text{-tagged-variables}(tagt, variants, elsep) \rho$
100 else $allocate\text{-variant-variables}(tag, tagt, variants, elsep) \rho$

Variants

Concrete Syntax

variant = [variant label list, ":", field list] ;

variant label list = variant label, { ":", variant label } ;

variant label = constant expression, [":", constant expression] ;

Abstract Syntax

$Variant :: labels : \text{Set-definition}$

$fields : \text{Field-list}$

Declaration Semantics

functions

$d\text{-variant} : \text{Variant} \rightarrow \text{Environment} \rightarrow (\text{Variant-structure} \times \text{Environment})$

$d\text{-variant } (mk\text{-Variant}(labels, fields)) \rho \triangleq$

218 let $nlabels = \bigcup \{ evaluate\text{-constant-expression } (label) \rho \mid label \in labels \}$ in

98 let $mk\text{-}(nfields, \rho_1) = build\text{-field-list-structure } (fields) \rho$ in
 $mk\text{-}(mk\text{-Variant-structure } (nlabels, nfields), \rho_1)$

annotations The value of the labels that are associated with the variant are evaluated. The structures of the fields are elaborated; the result is the structure of the component.

Static Semantics

The case labels of a case label list of a variant shall be constant expressions.

functions

$wf\text{-variant} : \text{Variant} \rightarrow \text{Environment} \rightarrow \mathbb{B}$

$wf\text{-variant } (mk\text{-Variant}(labels, fields)) \rho \triangleq$

203 $wf\text{-set-definition } (labels) \rho \wedge$

93 $wf\text{-field-list } (fields) \rho \wedge$

123 $wf\text{-label } (labels) \rho$

Dynamic Semantics

The elaboration of a variant shall be the same as the elaboration of the fields component.

Section 6.3, 90-11-08

operations

$m\text{-variant-structure} : \text{Variant-structure} \rightarrow \text{Environment} \xrightarrow{o} \text{Variant-component}$

$m\text{-variant-structure} (mk\text{-Variant-structure} (labels, fields)) \rho \triangleq$
 let $mk\text{-Set-value} (value) = labels$ in
 94 def $store = record\text{-declaration}(fields) \rho$;
 return $mk\text{-Variant-component} (value, store)$

Auxiliary Functions

functions

$build\text{-field-list-structure} : \text{Field-list} \rightarrow \text{Environment} \rightarrow (\text{Field-list-structure} \times \text{Environment})$

$build\text{-field-list-structure} (fieldss) \rho \triangleq$
 if $fieldss = [] \vee fieldss = \text{nil}$
 then $mk\text{-}([], \rho)$
 94 else let $mk\text{-}(firstfield, \rho_1) = d\text{-fields} (\text{hd } fieldss) \rho$ in
 98 let $mk\text{-}(remainder, \rho_2) = build\text{-field-list-structure} (\text{tl } fieldss) \rho_1$ in
 $mk\text{-}([firstfield] \curvearrowright remainder, \rho_2)$

functions

$is\text{-distinct-identifiers} : \text{Field-list} \rightarrow \mathbb{B}$

$is\text{-distinct-identifiers} (fields) \triangleq$
 98 let $ids = identifiers\text{-of-field-list} (fields)$ in
 std $is\text{-uniques} (ids)$

annotations Check that all the identifiers of a field list are distinct.

functions

$identifiers\text{-of-field-list} : \text{Field-list} \rightarrow \text{Identifier}^*$
 $identifiers\text{-of-field-list} (fields) \triangleq$
 if $fields = []$
 then $[]$
 98 else $identifiers\text{-of-fields} (\text{hd } fields) \curvearrowright identifiers\text{-of-field-list} (\text{tl } fields)$

annotations The result of this function is a sequence containing the field identifiers of a record. The order of the identifiers in the sequence is identical to the order in which they occur in the record.

functions

$identifiers\text{-of-fields} : \text{Fields} \rightarrow \text{Identifier}^*$
 $identifiers\text{-of-fields} (field) \triangleq$
 cases $field$:
 98 $mk\text{-Fixed-fields} () \rightarrow identifiers\text{-of-fixed-fields} (field),$
 99 $mk\text{-Variant-fields} () \rightarrow identifiers\text{-of-variant-fields} (field)$
 end

annotations The result of this function is a sequence containing the field identifiers of a field of a record, the order of the identifiers in the sequence is identical to the order in which they occur in the field.

functions

$identifiers\text{-of-fixed-fields} : \text{Fixed-fields} \rightarrow \text{Identifier}^*$
 $identifiers\text{-of-fixed-fields} (field) \triangleq$
 let $mk\text{-Fixed-fields} (ids, -) = field$ in
 ids

functions

```

identifiers-of-variant-fields : Variant-fields  $\rightarrow$  Identifier*
identifiers-of-variant-fields (field)  $\triangleq$ 
  let mk-Variant-fields (tag, -, variants, elsep) = field in
  let tagid = if tag = nil then [] else [tag] in
98   tagid  $\curvearrowright$  identifiers-of-variants (variants)  $\curvearrowright$  identifiers-of-field-list (elsep)

```

functions

```

identifiers-of-variants : Variant*  $\rightarrow$  Identifier*
identifiers-of-variants (variants)  $\triangleq$ 
  if variants = []
  then []
99   else identifiers-of-a-variant (hd variants)  $\curvearrowright$  identifiers-of-variants (tl variants)

```

annotations The result of this function is a sequence containing the field identifiers of a variant field component of a record. The order of the identifiers in the sequence is identical to the order in which they occur in the variant field component.

functions

```

identifiers-of-a-variant : Variant  $\rightarrow$  Identifier*
identifiers-of-a-variant (variant)  $\triangleq$ 
  let mk-Variant (-, fields) = variant in
98   identifiers-of-field-list (fields)

```

annotations The result of this function is a sequence containing the field identifiers of a variant component of a variant field a record. The order of the identifiers in the sequence is identical to the order in which they occur in the component.

functions

```

build-variant-structure : Variant*  $\rightarrow$  Environment  $\rightarrow$  Variant-structure*
build-variant-structure (variants) $\rho$   $\triangleq$ 
  if variants = []
  then []
99   else d-variant (hd variants) $\rho$   $\curvearrowright$  build-variant-structure (tl variants) $\rho$ 

```

annotations Construct a sequence of variant structures from a sequence of variants.

Storage Allocation for Variant Components of Records

```

allocate-tagged-variables : Typed  $\times$  Variant-structure*  $\times$  Field-list-structure  $\rightarrow$  Environment  $\xrightarrow{o}$  Variable
allocate-tagged-variables (tagt, variants, elsep) $\rho$   $\triangleq$ 
78   def tagloc = m-typed (tagt)  $\rho$ ;
98   def vars = { m-variant-structure (variants(i))  $\rho$  | i  $\in$  inds variants };
99   def elsepvars = allocate-elsep-variables (tagt, vars, elsep)  $\rho$ ;
  let tagvar = mk-Tag-variable (tagloc, vars  $\cup$  elsepvars) in
100  return merge { tag-tagged-variable (tagvar, var) | var  $\in$  (vars  $\cup$  elsepvars) }

```

annotations Allocate storage for a variant fields component of a variant record.

```

allocate-elsep-variables : Typed  $\times$  Variant-component-set  $\times$  Field-list-structure  $\xrightarrow{o}$  Variant-component-set
allocate-elsep-variables (tagt, vars, elsep) $\rho$   $\triangleq$ 
  if elsep = nil
  then return { }

```

```

290 else let elseplabels = values-of (tagt) – merge {var.labels | var ∈ elems vars} in
94   def elsepfields = record-declaration(elsep) ρ;
   return {mk-Variant-component (elseplabels, elsepfields)}

```

annotations Allocate storage space for the field list component following the **ELSE** keyword of a variant field.

functions

tag-tagged-variable : *Tag-variable* × *Variant-component* → *Record-Variable*

tag-tagged-variable (*varc*) \triangleq
 let *mk-Variant-component* (*labels*, *loc*) = *varc* in
 {*id* ↦ *mk-Tagged-variable* (*loc* (*id*), *tagloc*, *labels*) | *id* ∈ dom *loc*}

annotations Construct a tagged variable from the tag location and the range of values associated with that variant component.

allocate-variant-variables : *Identifier* × *Typed* × *Variant-structure** × *Field-list-structure* → *Environment* \xrightarrow{o} *Variable*

allocate-variant-variables (*tag*, *tagt*, *variant*, *elsep*) *ρ* \triangleq

```

78 def tagloc = m-typed(tagt) ρ;
98 def vars = {m-variant-structure(variant(i)) ρ | i ∈ dom variant};
99 def elsepvars = allocate-elsep-variables(tagt, vars, elsep) ρ;
  return {tag ↦ mk-Tag-variable (tagloc, vars ∪ elsepvars)}
         ∪ merge {tag-variant-variable(tagloc, var) | var ∈ (vars ∪ elsepvars)}

```

annotations Allocate a storage location for the tag. Allocate storage for each variant component and for the else component if it is present.

functions

tag-variant-variable : *Elementary-variable* × *Variant-component* → *Record-variable*

tag-variant-variable (*tagloc*, *varc*) \triangleq
 let *mk-Variant-component* (*labels*, *loc*) = *varc* in
 {*id* ↦ *mk-Variant-variable* (*loc* (*id*), *tagloc*, *labels*) | *id* ∈ dom *loc*}

annotations Construct a mapping that associates a variant variable with each of the label values of a variant component.

6.4 Compatibility

6.4.1 Expression Compatibility

TO DO — This section to be written.

At present the generic function `TE` returns the type of an expression or sub-expression. This type is the host type, therefore two expressions, `expra` and `exprb`, are expression compatible if:

$$TE(expra)\rho = TE(exprb)\rho$$

A function `is-expression-compatible` will be introduced to make this concept clearer.

An value of an expression of type Ta shall be expression-compatible with the value of an expression of type Tb if any of the following statements is true:

- a) ...
- b) Ta is a character type and the value of the other expression is a string constant of length 1; or
- c) Tv is a character type and the value of the other expression is the empty string constant; or
- d) ...

6.4.2 Assignment Compatibility

A variable of type Tv denoted by a variable designator shall be assignment-compatible with the value of an expression of type Te if any of the following statements is true:

- a) Tv and Te are identical types and that type shall not be an open array type, Tv is a subrange of the type Te , Te is a subrange of the type Tv , or Tv and Te are both subranges of the same type; or
- b) Tv is the unsigned type and Te is the signed type or a subrange of the signed type; or
- c) Tv is the signed type and Te is the unsigned type or a subrange of the unsigned type; or
- d) Tv is a pointer type and the expression has the value denoted by the pervasive identifier `NIL`; or
- e) Tv is a proper procedure type or function procedure type, and the value of the expression is a proper procedure or function procedure value with the same structure as the procedure type Tv ; or
- f) Tv is a character type and the value of the expression is a string constant of length 1; or
- g) Tv is a character type and the value of the expression is the empty string constant; or
- h) Tv is a type with a structure of the form `ARRAY [...] OF CHAR` and the expression is a string constant, whose length is less than or equal to the number of components of the array associated with the type Tv ; or
- i) Tv is the address type and Te is a pointer type or Te is the address type and Tv is a pointer type.

NOTES

1 A variable designator only occurs on the left-hand-side of an assignment statement, or as an actual parameter that corresponds to a variable formal parameter.

2 It should be noted that variable identifiers that occur in expressions are value designators.

CHANGE — The index type of an ARRAY OF CHAR that is the target of a string assignment can be any ordinal type.

Language Clarification

The change to Modula-2 string handling of always adding a string terminator to a string value has not been adopted. (This modification is described in the fourth edition of *Programming in Modula-2*.)

functions

$is_assignment_compatible : Typed \times Typed \rightarrow Environment \rightarrow \mathbb{B}$

$is_assignment_compatible(ltype, rtype)\rho \triangleq$
 $wf_simple_assignment(ltype, rtype)\rho \vee$
 $wf_unsigned_assignment(ltype, rtype)\rho \vee$
 $wf_signed_assignment(ltype, rtype)\rho \vee$
 $wf_pointer_assignment(ltype, rtype)\rho \vee$
 $wf_procedure_assignment(ltype, rtype)\rho \vee$
 $wf_character_assignment(ltype, rtype)\rho \vee$
 $wf_string_constant_assignment(ltype, rtype)\rho \vee$
 $wf_pointer_and_address_assignment(ltype, rtype)\rho$

functions

$wf_simple_assignment : Typed \times Expression_typed \rightarrow Environment \rightarrow \mathbb{B}$

$wf_simple_assignment(ltype, rtype)\rho \triangleq$
 $host_type_of(ltype)\rho = rtype \wedge$
 $\neg is_open_array(rtype)$

annotations

The type of the variable designator and the type of the expression are identical, or the type of the variable designator is a subrange of the type of the expression. The type of the variable designator is a subrange of the result type of the expression if the host type of the variable designator is identical to the result type of the expression. The result type of the expression is not an open array type.

functions

$wf_unsigned_assignment : Typed \times Expression_typed \rightarrow Environment \rightarrow \mathbb{B}$

$wf_unsigned_assignment(ltype, rtype)\rho \triangleq$
 $host_type_of(ltype)\rho = UNSIGNED_TYPE \wedge$
 $rtype = SIGNED_TYPE$

annotations

The type of the variable designator is the unsigned type or a subrange of the unsigned type and the type of the expression is the signed type.

functions

$wf\text{-signed-assignment} : Typed \times Expression\text{-typed} \rightarrow Environment \rightarrow \mathbb{B}$

$wf\text{-signed-assignment}(ltype, rtype)\rho \triangleq$
 $host\text{-type-of}(ltype)\rho = \text{SIGNED-TYPE} \wedge$
 $rtype = \text{UNSIGNED-TYPE}$

annotations The type of the variable designator is the signed type or a subrange of the signed type and the type of the expression is the unsigned type.

functions

$wf\text{-pointer-assignment} : Typed \times Expression\text{-typed} \rightarrow Environment \rightarrow \mathbb{B}$

$wf\text{-pointer-assignment}(ltype, rtype)\rho \triangleq$
 $is\text{-pointer-type}(ltype)\rho \wedge$
 $rtype = \text{NIL-TYPE}$

annotations The type of the variable designator is a pointer type and the expression has the value denoted by the pervasive identifier NIL.

functions

$wf\text{-procedure-assignment} : Typed \times Expression\text{-typed} \rightarrow Environment \rightarrow \mathbb{B}$

$wf\text{-procedure-assignment}(ltype, rtype)\rho \triangleq$
 $is\text{-procedure-type}(ltype)\rho \wedge$
 $structure\text{-of}(ltype)\rho = rtype \wedge$
 $level\text{-of}(rtype) = 0$

annotations The type of the variable designator is a proper procedure type or function procedure type and the type of the expression is a proper procedure or function procedure value with the same structure.

functions

$wf\text{-character-assignment} : Typed \times Expression\text{-typed} \rightarrow Environment \rightarrow \mathbb{B}$

$wf\text{-character-assignment}(ltype, rtype)\rho \triangleq$
 $host\text{-type-of}(ltype)\rho = \text{CHARACTER-TYPE} \wedge$
 $is\text{-string-type}(rtype) \wedge$
 $rtype.length = 1$

annotations The type of the variable designator is a character type and the type of the expression is a string constant of length 1.

$wf-string-constant-assignment : Typed \times Expression-typed \rightarrow Environment \rightarrow \mathbb{B}$

$wf-string-constant-assignment(ltype, rtype)\rho \triangleq$
 $component-type-of(ltype)\rho = \text{CHARACTER-TYPE} \wedge$
 $is-string-type(rtype) \wedge$
 $rtype.length \leq upper-bound-of(ltype) - lower-bound-of(ltype) + 1$

annotations The type of the variable for string constant assignment is an array of characters and the lower bound of the array is zero. The result of the expression is a string constant, and the length of the string constant is less than or equal to $n+1$, the number of elements in the array associated with the type T_1 .

$wf-pointer-and-address-assignment : Typed \times Expression-typed \rightarrow Environment \rightarrow \mathbb{B}$

$wf-pointer-and-address-assignment(ltype, rtype)\rho \triangleq$
 $(is-address-type(ltype)\rho \wedge$
 $is-pointer-type(rtype)\rho) \vee$
 $(is-address-type(rtype)\rho \wedge$
 $is-pointer-type(ltype)\rho)$

annotations The type of the variable is the address type and the result of the expression is a pointer type, or vice-versa.

6.4.3 Parameter Compatibility

A formal parameter shall be parameter compatible with an actual parameter if any of the following statements is true, except in the case when the formal parameter is of a type that is one of the storage types that is exported from the module SYSTEM:

- a) If the parameter is a value parameter, the corresponding actual parameter shall be an expression whose value is assignment-compatible with the type of the formal parameter; or
- b) If the formal parameter is an open array value parameter, and the actual parameter is an array whose component type is assignment compatible with the component type of the open array formal parameter; or
- c) If the parameter is a variable parameter, the actual parameter shall be a designator denoting a variable whose type shall be identical to the type of the formal parameter; or
- d) If the formal parameter is an open array variable parameter, and the actual parameter is an array whose component type is the same as the component type of the open array formal parameter.

Language Clarification

A component type is not parameter compatible with an open array of that component type; e.g. CHAR is not compatible with ARRAY OF CHAR.

functions

$is-parameter-compatible : Formal-parameter-typed \times Actual-parameter \rightarrow Environment \rightarrow \mathbb{B}$

$is-parameter-compatible(fp, ap)\rho \triangleq$
 cases fp :
 $mk-Value-formal-type(ft) \rightarrow ap \in Expression \wedge$
 $is-value-compatible(ft, TE(ap)\rho)\rho,$
 $mk-Variable-formal-type(ft) \rightarrow ap \in Variable-designator \wedge$
 $is-variable-compatible(ft, TV(ap)\rho)\rho$
 end

annotations If the formal parameter is a value parameter, then check that the actual parameter is an expression and that the formal and actual parameters are assignment compatible. If the formal parameter is a variable parameter, then check that the actual parameter is a variable and that the formal and actual parameters are compatible.

6.4.3.1 Value Parameter Compatibility

The type of a value parameter shall be assignment compatible to the type of the actual parameter, except in the case when the formal parameter is of a type that is one of the storage types that is exported from the module SYSTEM.

functions

$is-value-compatible : Variable-typed \times Expression-typed \rightarrow Environment \rightarrow \mathbb{B}$

$is-value-compatible(ft, et)\rho \triangleq$
 ($is-system-parameter(ft) \rightarrow is-system-compatible(ft, et)\rho,$
 $ft \in Open-array-typed \rightarrow is-array-type(et)\rho \wedge$
 $\text{let } mk-Open-array-typed(otype) = ft \text{ in let } ctype = component-type-of(et) \text{ in } is-value-compatible(otype, ctype)\rho$
 $ft \in Typed \rightarrow is-assignment-compatible(ft, et)\rho$)

annotations For formal and actual parameters to be compatible, the formal parameter is constructed from a storage type exported from the module SYSTEM, and the actual parameter is system compatible with it; the formal parameter is an open array, and the actual parameter is an array whose component type is compatible with the component type of the open array formal parameter; or the formal parameter is assignment-compatible with the actual parameter.

6.4.3.2 Variable Parameter Compatibility

A variable parameter shall be of an identical type to the actual parameter, except in the case when the formal parameter is of a type that is one of the storage types that is exported from the module SYSTEM.

functions

$is-variable-compatible : Variable-typed \times Variable-typed \rightarrow \mathbb{B}$

$is-variable-compatible(ft, vt)\rho \triangleq$
 ($is-system-parameter(ft) \rightarrow is-system-compatible(ft, vt)\rho,$
 $ft \in Open-array-typed \rightarrow is-array-type(vt)\rho \wedge$

$ft \in Typed \rightarrow ft = vt)$ $\text{let } mk\text{-}Open\text{-}array\text{-}typed(otype) = ft \text{ in } \text{let } ctype = \text{component-type-of}(vt) \text{ in } is\text{-}variable\text{-}compatible(otype)$

annotations For formal and actual parameters to be compatible, the formal parameter is constructed from a storage type exported from the module SYSTEM, and the actual parameter is system compatible with it; or the type of the formal parameter must be identical with the type of the actual parameter.

Auxiliary Definitions

functions

$is\text{-}system\text{-}parameter : Variable\text{-}typed \rightarrow Environment \rightarrow \mathbb{B}$

$is\text{-}system\text{-}parameter(vtype)\rho \triangleq$
 $(vtype \in Typed \rightarrow s\text{-}system\text{-}storage(vtype)\rho,$
 $vtype \in Open\text{-}array\text{-}typed \rightarrow \text{let } mk\text{-}Open\text{-}array\text{-}typed(type) = vtype \text{ in } is\text{-}system\text{-}storage(type)\rho)$

annotations If the formal parameter is a system storage type, or is an open array whose components are of a system storage type.

functions

$is\text{-}system\text{-}storage : Typed \rightarrow Environment \rightarrow \mathbb{B}$

$is\text{-}system\text{-}storage(type)\rho \triangleq$
 $type\text{-of}(type)\rho = address\text{-}type \vee$
 $type\text{-of}(type)\rho = loc\text{-}type \vee$
 $(component\text{-}type\text{-of}(type)\rho = loc\text{-}type \wedge$
 $is\text{-}zero\text{-}array\text{-}type(type)\rho)$

annotations A system storage type is either a storage type exported from the module SYSTEM, or an array whose index type is a whole number type with a lower bound of zero and whose component type is a storage type exported from system SYSTEM.

6.4.3.3 System Parameter Compatibility

A formal parameter shall be system parameter compatible with an actual parameter if any of the following statements is true:

- a) A formal parameters of type LOC is system parameter compatible with any data type whose SIZE is 1.
- b) A formal parameters of type ARRAY [0..n-1] OF LOC is system parameter compatible with any data type whose SIZE is equal to n.
- c) A formal parameters of type ARRAY OF LOC is system parameter compatible with any data type.
- d) A formal parameters of type ARRAY OF ARRAY [0..n-1] OF LOC is system parameter compatible with any data type whose SIZE is a multiple of n.

$is-system-compatible : Variable-typed \times Expression-typed \rightarrow Environment \rightarrow \mathbb{B}$

$$is-system-compatible(ftype, atype)\rho \triangleq$$

$$(ftype \in Typed \rightarrow (type-of(ftype)\rho = address-type \wedge$$

$$is-pointer-type(atype)\rho) \vee$$

$$tsize(ftype)\rho = tsize(atype)\rho,$$

$$ftype \in Open-array-typed \rightarrow \text{let } mk-Open-array-typed(type) = ftype \text{ in } \exists n \in \mathbb{N} \cdot$$

$$tsize(type) \times n = tsize(atype)\rho \wedge$$

$$atype \notin \{\mathbb{Z}\text{-type}, \mathbb{R}\text{-type}\})$$

annotations Parameters of type LOC are such that any data type whose SIZE is 1 can be passed when such a parameter is declared. Parameters of type ARRAY [0..n-1] OF LOC are such that any data type whose SIZE is equal to n can be passed.

Parameters of type ARRAY OF LOC are such that any data type can be passed when such a parameter is declared. Parameters of type ARRAY OF ARRAY [0..n-1] OF LOC are such that any data type whose SIZE is a multiple of n can be passed.

Language Clarification

Number constants cannot be passed to open arrays of LOC.

6.5 Statements

Statements denote actions. Statements are executed and their execution has an effect which is a transformation of the state of the computation.

Concrete Syntax

```
statement =
    empty statement      | assignment statement | procedure call      | return statement    |
    with statement       | if statement       | case statement      | while statement     |
    repeat statement     | loop statement     | exit statement      | for statement ;
```

NOTE — Certain statements are not composed of any parts that are themselves statements; these are the empty statement, the assignment statement, the procedure call, the return statement, and the exit statement. The remaining statements have components that are themselves statements; these are the with statement, the if statement, the case statement, the while statement, the repeat statement, the loop statement, and the for statement.

Abstract Syntax

types

```
Statement = Empty-statement
           | Assignment-statement
           | Procedure-call
           | Return-statement
           | With-statement
           | If-statement
           | Case-statement
           | While-statement
           | Repeat-statement
           | Loop-statement
           | Exit-statement
           | For-statement
```

Static Semantics

functions

$wf_statement : Statement \rightarrow Environment \rightarrow \mathbb{B}$

$wf_statement(st)\rho \triangleq$

```
110   (is-Empty-statement(st)       $\rightarrow$  wf-empty-statement(st) $\rho$ ,
111   is-Assignment-statement(st)  $\rightarrow$  wf-assignment-statement(st) $\rho$ ,
114   is-Procedure-call(st)         $\rightarrow$  wf-procedure-call(st) $\rho$ ,
??   is-Return-statement(st)       $\rightarrow$  wf-return-statement(st) $\rho$ ,
118   is-With-statement(st)         $\rightarrow$  wf-with-statement(st) $\rho$ ,
120   is-If-statement(st)           $\rightarrow$  wf-if-statement(st) $\rho$ ,
121   is-Case-statement(st)         $\rightarrow$  wf-case-statement(st) $\rho$ ,
124   is-While-statement(st)        $\rightarrow$  wf-while-statement(st) $\rho$ ,
125   is-Repeat-statement(st)       $\rightarrow$  wf-repeat-statement(st) $\rho$ ,
125   is-Loop-statement(st)         $\rightarrow$  wf-loop-statement(st) $\rho$ ,
126   is-Exit-statement(st)         $\rightarrow$  wf-exit-statement(st) $\rho$ ,
127   is-For-statement(st)          $\rightarrow$  wf-for-statement(st) $\rho$ )
```

Dynamic Semantics

operations

$$m\text{-statement} : \text{Statement} \rightarrow \text{Environment} \xrightarrow{o} \mathbb{B}$$

$$m\text{-statement}(st)\rho \triangleq$$

110	$(\text{is-Empty-statement}(st) \rightarrow m\text{-empty-statement}(st)\rho,$
111	$\text{is-Assignment-statement}(st) \rightarrow m\text{-assignment-statement}(st)\rho,$
114	$\text{is-Procedure-call}(st) \rightarrow m\text{-procedure-call}(st)\rho,$
??	$\text{is-Return-statement}(st) \rightarrow m\text{-return-statement}(st)\rho,$
119	$\text{is-With-statement}(st) \rightarrow m\text{-with-statement}(st)\rho,$
120	$\text{is-If-statement}(st) \rightarrow m\text{-if-statement}(st)\rho,$
122	$\text{is-Case-statement}(st) \rightarrow m\text{-case-statement}(st)\rho,$
124	$\text{is-While-statement}(st) \rightarrow m\text{-while-statement}(st)\rho,$
125	$\text{is-Repeat-statement}(st) \rightarrow m\text{-repeat-statement}(st)\rho,$
126	$\text{is-Loop-statement}(st) \rightarrow m\text{-loop-statement}(st)\rho,$
126	$\text{is-Exit-statement}(st) \rightarrow m\text{-exit-statement}(st)\rho,$
128	$\text{is-For-statement}(st) \rightarrow m\text{-for-statement}(st)\rho)$

6.5.1 Statement Sequences

A statement sequence specifies a sequence of actions in terms of the statements of the statement sequence.

Concrete Syntax

statement sequence = statement, { ";", statement } ;

Abstract Syntax

types

$$\text{Statement-sequence} :: \text{actions} : \text{Statement}^+$$

Static Semantics

functions

$$wf\text{-statement-sequence} : \text{Statement-sequence} \rightarrow \text{Environment} \rightarrow \mathbb{B}$$

$$wf\text{-statement-sequence}(mk\text{-Statement-sequence}(\text{actions}))\rho \triangleq$$

$$108 \quad \forall \text{action} \in \text{elems actions} \cdot wf\text{-statement}(\text{action})\rho$$

annotations Check each statement of the statement sequence.

Dynamic Semantics

The execution of a statement sequence shall cause the execution of the first statement in the sequence followed by the execution of any remaining statements of the sequence.

operations

$$m\text{-statement-sequence} : \text{Statement-sequence} \rightarrow \text{Environment} \xrightarrow{o} ()$$

$$m\text{-statement-sequence}(mk\text{-Statement-sequence}(\text{actions}))\rho \triangleq$$

(let $[first] \curvearrowright rest = \text{actions}$ in

$$109 \quad m\text{-statement}(first)\rho ;$$

if $rest \neq []$

$$109 \quad \text{then } m\text{-statement-sequence}(rest)\rho$$

else skip)

6.5.2 Empty Statements

An empty statement contains no symbols and denotes no action. Its use permits the relaxation of punctuation rules in statement sequences.

Concrete Syntax

empty statement = ;

Abstract Syntax

types

Empty-statement ::

Static Semantics

functions

$wf_empty_statement : Empty_statement \rightarrow Environment \rightarrow \mathbb{B}$
 $wf_empty_statement (mk_Empty_statement())\rho \triangleq$
true

Dynamic Semantics

An empty statement shall denote no action; its execution shall produce no changes to the state.

operations

$m_empty_statement : Empty_statement \rightarrow Environment \xrightarrow{o} ()$
 $m_empty_statement (mk_Empty_statement())\rho \triangleq$
skip

6.5.3 Assignment Statements

An assignment statement specifies that a variable is to be given a value.

Concrete Syntax

assignment statement = variable designator, ":", expression ;

Abstract Syntax

types

Assignment-statement :: *desig* : *Variable-designator*
expr : *Expression*

Static Semantics

The variable designator shall denote a variable. The type of the expression shall be assignment-compatible with the type of the variable designator.

functions

$wf\text{-}assignment\text{-}statement : Assignment\text{-}statement \rightarrow Environment \rightarrow \mathbb{B}$
 $wf\text{-}assignment\text{-}statement(mk\text{-}Assignment\text{-}statement(design, expr))\rho \triangleq$
145 $wf\text{-}variable\text{-}designator(design)\rho \wedge$
152 $wf\text{-}expression(expr)\rho \wedge$
163 $is\text{-}assignment\text{-}compatible(t\text{-}variable\text{-}designator(design)\rho, t\text{-}expression(expr)\rho)\rho$

Dynamic Semantics

The execution of an assignment statement shall cause the expression on the right-hand side of the assignment operator to be evaluated and the resulting value to replace the current value of the variable designated on the left-hand-side of the assignment operator. The order of evaluation of the variable designator and the expression shall be implementation dependent; the value which is the result of evaluating the expression shall be assigned to the designated variable.

If the assignment is to an array variable, component by component assignment of the array value to the array variable shall occur, except that assignment of an undefined value shall be permitted. If the assignment is a string constant to an array of CHAR, and the string constant does not fill the array, the string shall be terminated with an end of string character and the remainder of the array shall become undefined.

If the assignment is to a record variable, assignment shall be component by component. Assignment of an undefined component shall be permitted.

If the assignment is to a tag of a record and the new value of the tag causes a new variant to be selected, the values of the variables of the field list sequence associated with the old value of the tag shall become undefined. If the new value of the tag does not cause a new variant to be selected, the assignment to the tag shall produce no change to the associated variant.

It shall be an exception if the value to be assigned to a tag is such that no variant is associated with that value, and if there is no ELSE field list sequence.

NOTES

- 1 The evaluation of either the variable designator or the expression could produce a transformation of the state, i.e. have side-effects.
- 2 The order of assignment to the components of an array does not affect the meaning of a standard program.
- 3 The order of assignment to the components of a record does not affect the meaning of a standard program.

operations

$m\text{-}assignment\text{-}statement : Assignment\text{-}statement \rightarrow Environment \xrightarrow{o} ()$
 $m\text{-}assignment\text{-}statement(mk\text{-}Assignment\text{-}statement(design, expr))\rho \triangleq$
145 $def\ lhs = m\text{-}variable\text{-}designator(design)\rho;$
153 $def\ rhs = m\text{-}expression(expr)\rho;$
112 $assign(lhs, rhs)\rho$

annotations

Evaluate the designator and the expression; this evaluation can be done in any order, and assign the result of the expression evaluation to the target.

When the target of the assignment is a variant of a record, then in the case of explicit tags, the evaluation of the target designator will check that the tag values that select that component

are correctly set. In the case of implicit tag values selecting the variant, then these implicit tags will be set to the appropriate values.

Auxiliary Definitions

operations

```

assign : Variable × Value → Environment  $\xrightarrow{o}$  ()
assign (var, val) ρ  $\triangleq$ 
  cases var:
112   mk-Elementary-variable(-, -) → elementary-variable-assignment(var, val) ρ,
112   mk-Array-variable(-)         → array-variable-assignment(var, val) ρ,
113   mk-Record-variable(-)        → record-variable-assignment(var, val) ρ,
113   mk-Tag-variable(-, -)        → tag-variable-assignment(var, val) ρ,
113   mk-Variant-variable(-, -, -) → variant-variable-assignment(var, val) ρ,
114   mk-Tagged-variable(-, -, -)  → tagged-variable-assignment(var, val) ρ
end

```

annotations This function is used to assign to elementary variables, and is used recursively to assign to array, record, tag, variant, and tagged variables. An undefined value can only occur as part of an array or record value; it cannot occur as an elementary value.

operations

```

elementary-variable-assignment : Elementary-variable × Value → Environment  $\xrightarrow{o}$  ()
elementary-variable-assignment (var, val) ρ  $\triangleq$ 
  let mk-Elementary-variable(loc, range) = var in
  if range = nil
298   then change-value(loc, val)
  elseif val ∈ range ∨ val = UNDEFINED
298   then change-value(loc, val)
306   else mandatory-exception(ASSIGN-RANGE)

```

annotations Check that the value to be assigned is within range or is undefined; if so change the value of the target designator, if the value is not within the range there is an exception. An undefined value which is assigned here is part of an array or a record.

operations

```

array-variable-assignment : Array-variable × Value → Environment  $\xrightarrow{o}$  ()
array-variable-assignment (var, val) ρ  $\triangleq$ 
  for all i ∈ dom var
  do if i ∈ dom val
112   then assign(var(i), val(i)) ρ
150   elseif i = succ(maxs(dom val))
112   then assign(var(i), END-OF-STRING-CHAR) ρ
112   else assign(var(i), UNDEFINED) ρ

```

annotations Component by component assignment of the array value to the array variable occurs, except that assignment of an undefined value is permitted.

If the assignment is a string constant to an array of CHAR, and the string constant does not fill the array, the string is terminated with an end-of-string character and the remainder of the array is set to undefined; the second and third arms of the conditional are only pertinent to this. The well-formed conditions guarantee that the length of the string constant is less than or equal to the number of the components of the array.

operations

record-variable-assignment : *Record-variable* \times *Value* \rightarrow *Environment* \xrightarrow{o} ()

record-variable-assignment (*var*, *val*) $\rho \triangleq$

for all *id* \in dom *var*

112 do *assign*(*var*(*id*), *val*(*id*)) ρ

annotations

Assignment is component by component. Assignment of an undefined value is permitted.

operations

tag-variable-assignment : *Tag-variable* \times *Value* \rightarrow *Environment* \xrightarrow{o} ()

tag-variable-assignment (*var*, *val*) $\rho \triangleq$

let *mk-Tag-variable*(*tvar*, *variants*) = *var* in

let *tval* = *variable-value*(*tvar*) in

let *current* = *tagged-variant*(*tval*, *variants*) ρ in

if *current* \neq *tagged-variant*(*val*, *variants*) ρ

113 then (*set-to-undefined*(*current*) ρ ;

112 *assign*(*tvar*, *val*) ρ)

112 else *assign*(*tvar*, *val*) ρ

annotations

If the assignment is to a tag of a record and the new value of the tag causes a new variant to be selected, the variables of the field list sequence associated with the old value of the tag is set to undefined. If the new value of the tag does not cause a new variant to be selected, the assignment to the tag variable produces no change to the associated variant.

operations

set-to-undefined : *Record-components* \xrightarrow{o} ()

set-to-undefined (*field*) $\rho \triangleq$

300 let *locs* = *locs-of* (*field*) in

for all *loc* \in *locs*

298 do *change-value*(*loc*, UNDEFINED)

annotations

Set all of the variables in components of a variant record to undefined.

functions

tagged-variant : *Value* \times *Variant-component-set* \rightarrow *Record-components*

tagged-variant (*val*, *variants*) \triangleq

if *val* $\in \bigcup \{v.labels \mid v \in variants\}$

then let *v* \in *variants* be st *val* \in *v.labels* in *v.fields*

elseif *val* = UNDEFINED

then { }

306 else *mandatory-exception*(TAG-RANGE)

annotations

Return the (variant) component of a record which corresponds to the value of the associated tag. If the tag contains an undefined value, then the result is an empty record. If the value to be assigned to a tag variable is such that no variant is associated with that value; it is an exception if there is no ELSE field list sequence.

operations

variant-variable-assignment : *Variant-variable* \times *Value* \rightarrow *Environment* \xrightarrow{o} ()

variant-variable-assignment (*vvar*, *val*) $\rho \triangleq$

let *mk-Variant-variable*(*var*, -, -) = *vvar* in

112 *assign*(*var*, *val*) ρ


```

219  def argvals = m-actual-parameters(args) ρ;
??   ( is-Procedure-value(f)           → f(argvals) ,
226   is-Standard-procedure-designator(f) → m-standard-procedure-designator(f) (argvals) ρ,
??   is-Coroutine-procedure-designator(f) → m-coroutine-procedure-designator(f) (argvals) ρ)

```

annotations The evaluation order for the designator and the actual parameters given above is one of those permitted; the actual parameters may be evaluated before the designator or both may be evaluated in parallel.

6.5.4.1 Procedure Designators

Concrete Syntax

procedure designator = value designator ;

Abstract Syntax

types

Procedure-designator = *Value-designator*
 | *Standard-procedure*
 | *Coroutine-procedure*;

Standard-procedure :: *id* : *Identifier*
 desig : *Standard-procedure-designator*;

Coroutine-procedure :: *qid* : *Qualident*
 desig : *Coroutine-procedure-designator*

Static Semantics

functions

wf-procedure-designator : *Function-designator* × *Actual-parameters* → *Environment* → \mathbb{B}

wf-procedure-designator (*fdesig*, *args*) ρ \triangleq

```

183  ( is-Value-designator(fdesig)       → wf-value-designator (fdesig) ρ ∧
220   is-Standard-procedure(fdesig) → let mk-Standard-procedure(id, desig) = fdesig in
??   is-standard-procedure (id) ρ ∧
225   wf-standard-procedure-designator (desig) (args) ρ,
??   is-Coroutine-procedure(fdesig) → let mk-Coroutine-procedure(qid, desig) = fdesig in
??   is-coroutine-procedure (qid) ρ ∧
338   wf-coroutine-procedure-designator (desig) (args) ρ)

```

Dynamic Semantics

operations

m-procedure-designator : *Procedure-designator* → *Environment* \xrightarrow{o} *Value*

m-procedure-designator (*mk-Procedure-call*(*fdesig*, *args*)) ρ \triangleq

```

183  ( is-Value-designator(fdesig)       → m-value-designator(fdesig) ρ,
    is-Standard-procedure(fdesig) → let mk-Standard-procedure(-, desig) = fdesig in
    return desig,
    is-Coroutine-procedure(fdesig) → let mk-Standard-procedure(-, desig) = fdesig in
    return desig)

```

6.5.5 Return Statements

A return statement specifies the termination of the execution of the block of a proper procedure, a function procedure, or a module.

Concrete Syntax

return statement = simple return statement | function return statement ;

Abstract Syntax

types

$Return-statement = Simple-return-statement \mid Function-return-statement$

annotations The check that simple return statements are only contained in modules and proper procedures, and that function return statements must be contained in function procedures is given in the consistency check of a block.

6.5.5.1 Simple Return Statements

A simple return statement specifies the termination of the execution of a proper procedure block or a module block.

Concrete Syntax

simple return statement = "RETURN" ;

Abstract Syntax

types

$Simple-return-statement :: RETURN$

Static Semantics

functions

$wf-simple-return-statement : Simple-return-statement \rightarrow Environment \rightarrow \mathbb{B}$
 $wf-simple-return-statement(mk-Simple-return-statement(RETURN))\rho \triangleq$
true

Dynamic Semantics

The execution of a simple return statement shall terminate the execution of a proper procedure block or a module block.

operations

$m-simple-return-statement : Simple-return-statement \rightarrow Environment \xrightarrow{o} ()$
 $m-simple-return-statement(mk-Simple-return-statement(RETURN))\rho \triangleq$
?? $c-exit(RETURN) \rho$

6.5.5.2 Function Return Statements

A function return statement specifies the termination of the execution of a function procedure block and defines the value to be returned by that function procedure.

Concrete Syntax

function return statement = "RETURN", expression ;

Abstract Syntax

types

$$\begin{aligned} \text{Function-return-statement} &:: \text{type} : \text{Typed} \\ &\quad \text{expr} : \text{Expression} \end{aligned}$$

Static Semantics

The type of the expression shall be assignment compatible with the return type of the function procedure in which the function return statement appears.

functions

$$\begin{aligned} \text{wf-function-return-statement} &: \text{Function-return-statement} \rightarrow \text{Environment} \rightarrow \mathbb{B} \\ \text{wf-function-return-statement}(\text{mk-Function-return-statement}(\text{type}, \text{expr}))\rho &\triangleq \\ 102 \quad &\text{is-assignment-compatible}(\text{type}, \text{t-expression}(\text{expr})\rho)\rho \wedge \\ 152 \quad &\text{wf-expression}(\text{expr})\rho \end{aligned}$$

Dynamic Semantics

The execution of a function return statement shall terminate the execution of a function procedure and return a value. If the type of the function result is a subrange type, the value returned shall belong to the set of values defined by the subrange type, otherwise it shall be an exception.

operations

$$\begin{aligned} \text{m-function-return-statement} &: \text{Function-return-statement} \rightarrow \text{Environment} \xrightarrow{o} \text{Value} \\ \text{m-function-return-statement}(\text{mk-Function-return-statement}(\text{type}, \text{expr}))\rho &\triangleq \\ 153 \quad &\text{def result} = \text{m-expression}(\text{expr})\rho; \\ 281 \quad &\text{if is-subrange-type}(\text{type})\rho \\ 290 \quad &\text{then let values} = \text{values-of}(\text{type})\rho \text{ in} \\ &\quad \text{if result} \in \text{values} \\ ?? \quad &\quad \text{then c-exit}(\text{RETURN})(\rho) \text{ result} \\ 306 \quad &\quad \text{else mandatory-exception}(\text{RETURN-RANGE}) \\ ?? \quad &\text{else c-exit}(\text{RETURN})(\rho) \text{ result} \end{aligned}$$

annotations The expression of the return statement is evaluated. If the result is one of the values of the type defined by the return type of the function procedure, the result is returned; it is an exception if the result is not of the type defined.

6.5.6 With Statements

A with statement specifies a record variable and a statement sequence within which any field identifiers of that record variable need not be qualified by the designator of the record variable.

Concrete Syntax

with statement = "WITH", record designator, "DO", statement sequence, "END" ;

record designator = variable designator | value designator ;

Abstract Syntax

types

$With\text{-}statement :: design : Record\text{-}designator$
 $body : Statement\text{-}sequence;$

$Record\text{-}designator = Variable\text{-}designator \mid Value\text{-}designator$

Static Semantics

The designator shall denote a record variable or a record value. Within the statement sequence the field identifiers of the record variable or record value may be used without qualification.

functions

$wf\text{-}with\text{-}statement : With\text{-}statement \rightarrow Environment \rightarrow \mathbb{B}$
 $wf\text{-}with\text{-}statement(mk\text{-}With\text{-}statement(design, body))\rho \triangleq$
if $is\text{-}Variable\text{-}designator(design)$
145 then $wf\text{-}variable\text{-}designator(design)\rho \wedge$
145 let $type = t\text{-}variable\text{-}designator(design)\rho$ in
278 $is\text{-}record\text{-}type(type)\rho \wedge$
285 let $renv = field\text{-}types\text{-}of\text{-}record(type)\rho$ in
272 let $\rho_{block} = overwrite\text{-}var\text{-}environment(renv)\rho$ in
109 $wf\text{-}statement\text{-}sequence(body)\rho_{block}$
elseif $is\text{-}Value\text{-}designator(design)$
183 then $wf\text{-}value\text{-}designator(design)\rho \wedge$
214 $is\text{-}constant\text{-}expression(design)\rho \wedge$
152 let $type = t\text{-}expression(design)\rho$ in
278 $is\text{-}record\text{-}type(type)\rho \wedge$
285 let $renv = field\text{-}types\text{-}of\text{-}record(type)\rho$ in
218 let $vals = evaluate\text{-}constant\text{-}expression(design)\rho$ in
 let $venv = \{id \mapsto mk\text{-}Constant\text{-}value(renv(id), vals(id)) \mid id \in \text{dom } renv\}$ in
270 let $\rho_{block} = overwrite\text{-}const\text{-}environment(venv)\rho$ in
109 $wf\text{-}statement\text{-}sequence(body)\rho_{block}$
else false

annotations Check that the designator designates either a record variable or record value. Add the field identifiers with their corresponding types to the current environment to give the environment for the body of the with statement.

Dynamic Semantics

The record variable designator or value designator shall be evaluated before the statement sequence of the with statement is executed, and that evaluation shall establish a reference to the variable or value throughout the execution of the statement sequence of the with statement.

operations

```

m-with-statement : With-statement  $\rightarrow$  Environment  $\xrightarrow{o}$  ()
m-with-statement (mk-With-statement(desig, body)) $\rho \triangleq$ 
  if is-Variable-designator(desig)
145   then def mk-Record-variable(fields) = m-variable-designator(desig)  $\rho$ ;
272       let  $\rho_{block} = \text{overwrite-var-environment}(\text{fields})\rho$  in
109       m-statement-sequence(body)  $\rho_{block}$ 
183   else let type = t-value-designator(desig) $\rho$  in
285       let renv = field-types-of-record(type) $\rho$  in
183       def vals = m-value-designator(desig)  $\rho$ ;
       let fields = {id  $\mapsto$  mk-Constant-value(renv(id), value(id)) | id  $\in$  dom renv} in
270       let  $\rho_{block} = \text{overwrite-const-environment}(\text{fields})\rho$  in
109       m-statement-sequence(body)  $\rho_{block}$ 

```

annotations Add the field identifiers with their corresponding variables or values to the current environment to give the environment for the body of the with statement.

6.5.7 If Statements

An if statement is used to select one statement sequence for execution from alternatives, depending on the values of Boolean expressions.

Concrete Syntax

if statement = guarded statements, ["ELSE", statement sequence], "END" ;

guarded statements =
 "IF", Boolean expression, "THEN", statement sequence,
 { "ELSIF", Boolean expression, "THEN", statement sequence } ;

Boolean expression = expression ;

Abstract Syntax

types

If-statement :: *thens* : *Guarded-statement**
 elsep : [*Statement-sequence*];

Guarded-statement :: *guard* : *Expression*
 body : *Statement-sequence*

annotations An if statement always contains at least one statement sequence clause (which follows the first occurrence of a THEN symbol); thus after translation to the Abstract Syntax there is always a non-empty sequence of guarded statements. A missing ELSE subclause in the Concrete Syntax is denoted by nil in the abstract syntax.

Static Semantics

The type of the Boolean expressions shall be of the required Boolean type.

functions

$wf\text{-}if\text{-}statement : If\text{-}statement \rightarrow Environment \rightarrow \mathbb{B}$
 $wf\text{-}if\text{-}statement (mk\text{-}If\text{-}statement(thens, elsep))\rho \triangleq$
 $(\forall mk\text{-}Guarded\text{-}statement(guard, body) \in elems\ thens \cdot$
152 $wf\text{-}expression(guard)\rho \wedge$
152 $t\text{-}expression(guard)\rho = \text{BOOLEAN-TYPE} \wedge$
109 $wf\text{-}statement\text{-}sequence(body)\rho \wedge$
109 $(elsep = \text{nil} \vee wf\text{-}statement\text{-}sequence(elsep)\rho)$

Dynamic Semantics

The Boolean expressions of the sequence of guarded statements shall be evaluated in the order they occur until one of the Boolean expressions yields true; and then the associated statement sequence shall be executed. If all the Boolean expressions evaluate to false, the statement sequence of the ELSE subclause (if present) shall be executed. An if statement shall denote no action if all the Boolean expressions evaluate to the value false and there is no ELSE subclause.

NOTE — The evaluation of the Boolean expression could produce a transformation of the program state, i.e. have a side-effect.

operations

$m\text{-}if\text{-}statement : If\text{-}statement \rightarrow Environment \xrightarrow{o} ()$
 $m\text{-}if\text{-}statement (mk\text{-}If\text{-}statement(thens, elsep))\rho \triangleq$
if $thens = []$
120 then $execute\text{-}else\text{-}statements(elsep)\rho$
else let $mk\text{-}Guarded\text{-}statement(guard, body) = \text{hd}\ thens$ in
153 def $value = m\text{-}expression(guard)\rho$;
if $value$
109 then $m\text{-}statement\text{-}sequence(body)\rho$
120 else $m\text{-}if\text{-}statement(mk\text{-}If\text{-}statement(tl\ thens, elsep))\rho$

annotations

If there are no more elements remaining in the sequence of guarded statements then the else of the if statement (if present) is executed. If there are more elements remaining in the sequence of guarded statements, the Boolean expression of the first guarded statement in the sequence is evaluated, and if the result is true, the corresponding statement sequence is executed. If the result of the evaluation of the Boolean expression is false, then this element is removed from the sequence of guarded statements and the remainder of the sequence executed.

Auxiliary Definitions

operations

$execute\text{-}else\text{-}statements : [Statement\text{-}sequence] \rightarrow Environment \xrightarrow{o} ()$
 $execute\text{-}else\text{-}statements(elsep)\rho \triangleq$
if $elsep = \text{nil}$
then skip
109 else $m\text{-}statement\text{-}sequence(elsep)\rho$

annotations

If an else part is present, and no other component of the if statement has been executed, then the else part is executed.

6.5.8 Case Statements

A case statement is used to select one statement sequence for execution from alternatives, depending on the value of an expression.

Concrete Syntax

case statement = "CASE", case selector, "OF", case list, "END" ;

case selector = ordinal expression ;

case list = case, { case selector, case }, ["ELSE", statement sequence], ;

Abstract Syntax

types

Case-statement :: *expr* : *Expression*
 cases : *Case-set*
 elsep : [*Statement-sequence*]

annotations The number of elements in the cases component of the abstract representation of a case statement is equal to the number of case subclauses of the concrete representation.

Static Semantics

The case selector shall be of ordinal type and shall be expression-compatible with the type of each of the case labels. The values denoted by the case labels of the case label lists shall be distinct and of a type which is expression compatible with the type of the case selector. No value contained in a case label list shall occur in any other case label list of the same case statement.

The type of the case selector expression shall be expression compatible with the type of each of the case labels.

Case labels shall be constant expressions (or ranges of constant expressions).

functions

wf-case-statement : *Case-statement* \rightarrow *Environment* \rightarrow \mathbb{B}
wf-case-statement(*mk-Case-statement*(*expr*, *cases*, *elsep*)) $\rho \triangleq$
152 *wf-expression*(*expr*) $\rho \wedge$
123 $\forall case \in cases \cdot wf-case(case)\rho \wedge$
109 (*elsep* = nil \vee *wf-statement-sequence*(*elsep*) ρ) \wedge
122 *is-distinct-labels*(*cases*) $\rho \wedge$
152 let *type* = *t-expression*(*expr*) ρ in
280 *is-ordinal-type*(*type*) $\rho \wedge$
203 $\forall mk-Case(labels, -) \in cases \cdot t-set-definition(labels) = type$

annotations Check the case selector expression and its type. Check each of the case components and the else part if it is present.

Dynamic Semantics

The case selector shall be evaluated. The resulting value shall specify the execution of the statement sequence whose case label list contains that value. If the value of the case selector is not contained in any case label list, the statement sequence following the ELSE shall be executed if present - otherwise an exception shall be raised.

NOTE — A case label list contains a value if the value of a constant expression subclause of a case label list is equal to it, or if the value lies within the range specified by a pair of constant expression subclauses. The evaluation of the case selector could produce a transformation of the program state, i.e. have a side-effect.

operations

```

m-case-statement : Case-statement  $\rightarrow$  Environment  $\xrightarrow{o}$  ()
m-case-statement (mk-Case-statement(expr, cases, elsep)) $\rho \triangleq$ 
153   def i = m-expression(expr)  $\rho$ ;
203   if i  $\in \bigcup \{m\text{-set-definition}(c.\text{labels}) \rho \mid c \in \text{cases}\}$ 
203   then let c  $\in \text{cases}$  be st i  $\in m\text{-set-definition}(c.\text{labels}) \rho$  in
109     m-statement-sequence(c.body)  $\rho$ 
    elseif elsep  $\neq \text{nil}$ 
109   then m-statement-sequence(elsep)  $\rho$ 
306   else mandatory-exception(CASE-RANGE)

```

annotations The expression is evaluated and the result is used to determine which component of the case statement is to be executed.

Auxiliary Definitions

functions

```

is-distinct-labels : Case-set  $\rightarrow$  Environment  $\rightarrow \mathbb{B}$ 
is-distinct-labels (caseset) $\rho \triangleq$ 
   $\forall \text{case}_1 \in \text{caseset}, \text{case}_2 \in \text{caseset} .$ 
    case1 = case2  $\vee$ 
    let mk-Case(l1, -) = case1 in
    let mk-Case(l2, -) = case2 in
218   evaluate-constant-expression (l1) $\rho \cap \text{evaluate-constant-expression}$  (l2) $\rho = \{ \}$ 

```

annotations Check that the type of the expression is of ordinal type and that the values in the case labels are expression compatible with the type of the case selector expression. Check that each case label is distinct from the others and that they are disjoint.

6.5.8.1 Case

Concrete Syntax

```

case = [ case label list, ":", statement sequence ] ;

case label list = case label, { " ", case label } ;

case label = constant expression, [ "...", constant expression ] ;

```

Abstract Syntax

types

```

Case :: labels : Set-definition
       body : Statement-sequence

```

annotations Each case label list is translated into a set definition.

Static Semantics

The set of values defined by each of the case label lists shall be distinct from each other.

functions

$wf-case : Case \rightarrow Environment \rightarrow \mathbb{B}$

$wf-case (mk-Case(labels, body))\rho \triangleq$

203 $wf-set-definition (labels)\rho \wedge$

109 $wf-statement-sequence (body)\rho \wedge$

123 $wf-label (labels)\rho$

annotations Check the components of a case component.

Auxiliary Definitions

functions

$wf-label : Set-definition \rightarrow Environment \rightarrow \mathbb{B}$

$wf-label (labels)\rho \triangleq$

214 $is-constant-expression (labels)\rho \wedge$

123 $is-distinct-values (labels)\rho$

annotations Check that the values in a case label are distinct from the others and that they are disjoint.

functions

$is-distinct-values : Set-definition \rightarrow Environment \rightarrow \mathbb{B}$

$is-distinct-values (label)\rho \triangleq$

218 $\text{let } vals = \{evaluate-constant-expression (mem)\rho \mid mem \in label\} \text{ in}$

$\text{card } label = \text{card } vals \wedge$

?? $is-disjoint (vals)$

annotations Check that each label value or range of label values of a case labels is distinct from the others, and that the values are disjoint.

6.5.9 While Statements

A while statement specifies the execution of a statement sequence zero or more times, depending on the value of a Boolean expression.

Concrete Syntax

while statement = "WHILE", Boolean expression, "DO", statement sequence, "END" ;

Abstract Syntax

types

$While-statement :: expr : Expression$
 $body : Statement-sequence$

Static Semantics

The expression that controls the iteration shall be of the Boolean type.

functions

```
wf-while-statement : While-statement  $\rightarrow$  Environment  $\rightarrow \mathbb{B}$ 
wf-while-statement (mk-While-statement(expr, body)) $\rho \triangleq$ 
152   wf-expression (expr) $\rho \wedge$ 
152   t-expression (expr) $\rho = \text{BOOLEAN-TYPE} \wedge$ 
109   wf-statement-sequence (body) $\rho$ 
```

Dynamic Semantics

The Boolean expression shall be evaluated before each iteration. If the result of the evaluation is true, the statement sequence shall be executed, followed by the execution of the while statement again. If the result is false, the statement sequence shall not be executed and the execution of the while statement shall terminate.

NOTES

- 1 If the result of the first evaluation of the Boolean expression is false, the statement sequence of the while statement is not executed.
- 2 The evaluation of the Boolean expression could produce a transformation of the program state, i.e. have a side-effect.

operations

```
m-while-statement : While-statement  $\rightarrow$  Environment  $\xrightarrow{o}$  ()
m-while-statement (mk-While-statement(expr, body)) $\rho \triangleq$ 
153   def value = m-expression(expr)  $\rho$ ;
      if value
109   then (m-statement-sequence(body)  $\rho$  ;
124       m-while-statement(mk-While-statement(expr, body))  $\rho$  )
      else skip
```

annotations The execution of a while statement is defined by evaluating the expression, and if the result is true, executing the body of the while statement, followed by re-executing the while statement. If the expression evaluates to false, the execution of the while statement is terminated.

6.5.10 Repeat Statements

A repeat statement specifies the execution of a statement sequence one or more times, depending on the value of a Boolean expression.

Concrete Syntax

repeat statement = "REPEAT", statement sequence, "UNTIL", Boolean expression ;

Abstract Syntax

types

```
Repeat-statement :: body : Statement-sequence
                  expr : Expression
```

Static Semantics

The expression that controls the iteration shall be of the Boolean type.

functions

$wf-repeat-statement : Repeat-statement \rightarrow Environment \rightarrow \mathbb{B}$

$wf-repeat-statement (mk-Repeat-Statement(body, expr))\rho \triangleq$

109 $wf-statement-sequence (body)\rho \wedge$

152 $wf-expression (expr)\rho \wedge$

152 $t-expression (expr)\rho = \text{BOOLEAN-TYPE}$

Dynamic Semantics

The statement sequence shall be executed. The Boolean expression shall be evaluated; if the result is true, the repeat statement shall be executed again. If the result of the evaluation of the Boolean expression is false, the execution of the repeat statement shall terminate.

NOTES

1 The statement sequence is executed at least once.

2 The evaluation of the Boolean expression could produce a transformation of the program state, i.e. have a side-effect.

operations

$m-repeat-statement : Repeat-statement \rightarrow Environment \xrightarrow{o} ()$

$m-repeat-statement (mk-Repeat-statement(body, expr))\rho \triangleq$

109 $(m-statement-sequence(body) \rho ;$

153 $(\text{def } value = m-expression(expr) \rho ;$

if $\neg value$

125 then $m-repeat-statement(mk-Repeat-statement(body, expr)) \rho$

else skip))

annotations The execution of a repeat statement is defined by executing the body, evaluating the expression, and if this is not true re-executing the repeat statement. If the expression evaluates to true, the execution of the repeat statement is terminated.

6.5.11 Loop Statements

A loop statement specifies the repeated execution of a statement sequence. The statement sequence may contain zero or more exit statements, the execution of which terminates the execution of the loop statement.

Concrete Syntax

loop statement = "LOOP", statement sequence, "END" ;

Abstract Syntax

types

$Loop-statement :: body : Statement-sequence$

Static Semantics

functions

$wf-loop-statement : Loop-statement \rightarrow Environment \rightarrow \mathbb{B}$

$wf-loop-statement (mk-Loop-statement(body))\rho \triangleq$

109 $wf-statement-sequence (body)\rho$

Dynamic Semantics

operations

$$\begin{aligned}
& m\text{-loop-statement} : \text{Loop-statement} \rightarrow \text{Environment} \xrightarrow{o} () \\
& m\text{-loop-statement} (mk\text{-Loop-statement}(body)) \rho \triangleq \\
\text{??} \quad & (\text{def } \rho_{cont} = c\text{-fixe}(\{\text{TERMINATION} \mapsto \text{skip}\}) \rho; \\
109 \quad & m\text{-statement-sequence}(body) \rho_{cont} ; \\
126 \quad & m\text{-loop-statement}(mk\text{-Loop-statement}(body)) \rho)
\end{aligned}$$

6.5.12 Exit Statements

An exit statement may occur only within a loop statement when it specifies the termination of that loop statement.

Concrete Syntax

exit statement = "EXIT" ;

Abstract Syntax

types

Exit-statement :: EXIT

Static Semantics

An exit statement shall be contained within a loop statement.

functions

$$\begin{aligned}
& wf\text{-exit-statement} : \text{Exit-statement} \rightarrow \text{Environment} \rightarrow \mathbb{B} \\
& wf\text{-exit-statement} (mk\text{-Exit-statement}(\text{EXIT})) \rho \triangleq \\
& \quad \text{true}
\end{aligned}$$

annotations The check that the exit statement is contained inside a loop statement is part of the consistency check for the procedure/function/module that contains the exit statement.

Dynamic Semantics

The exit statement shall terminate the execution of a loop statement; the loop statement which is terminated is that loop statement which contains the exit statement but which does not contain another loop statement containing the exit statement.

operations

$$\begin{aligned}
& m\text{-exit-statement} : \text{Exit-statement} \rightarrow \text{Environment} \xrightarrow{o} () \\
& m\text{-exit-statement} (mk\text{-Exit-statement}(\text{EXIT})) \rho \triangleq \\
\text{??} \quad & c\text{-exit}(\text{TERMINATION}) \rho
\end{aligned}$$

6.5.13 For Statements

A for statement specifies that a statement sequence is to be repeatedly executed while a sequence of values are assigned, one-by-one, to a variable, called the control variable.

Concrete Syntax

for statement =
"FOR", control variable identifier, ":", initial value, "TO", final value, ["BY", step size], "DO", statement sequence, "END" ;
control variable identifier = identifier ;
initial value = ordinal expression ;
final value = ordinal expression ;
step size = constant expression ;

Abstract Syntax

types

For-statement :: *id* : *Identifier*
 initial : *Expression*
 final : *Expression*
 step : *Expression*
 body : *Statement-sequence*

annotations If the step subclause is not present in the Concrete Syntax, the translation process (from concrete to Abstract Syntax) will supply a step value of 1.

Static Semantics

The control variable shall be declared as a simple variable of ordinal type in a variable declaration part of the block that contains the for statement (see section xx). The type of the control variable shall be assignment compatible with the type of the initial expression. Either the type of the control variable or the host type of the type of the control variable and the type of the final expression shall be identical types. If the for statement does not contain a step clause, the step shall have the value 1. Otherwise the step shall be a constant expression of whole number type and be in an implementation defined range. The value of the step shall be non-zero. The for statement shall not contain a statement threatening the control variable.

Language Change The control variable is now assignment compatible with the initial value.

functions

wf-for-statement : *For-statement* \rightarrow *Environment* \rightarrow \mathbb{B}
wf-for-statement (*mk-For-statement*(*id*, *initial*, *final*, *step*, *body*)) $\rho \triangleq$
272 *is-variable* (*id*) $\rho \wedge$
?? *let type* = *t-entire-variable* (*mk-Entire-variable*(*id*)) ρ in
280 *is-ordinal-type* (*type*) $\rho \wedge$
152 *wf-expression* (*initial*) $\rho \wedge$
152 *wf-expression* (*final*) $\rho \wedge$
152 *wf-expression* (*step*) $\rho \wedge$
102 *is-assignment-compatible* (*type*, *t-expression*(*initial*) ρ) $\rho \wedge$
250 *is-comparable* (*host-type-of* (*type*) ρ , *t-expression* (*final*) ρ) $\rho \wedge$
214 *is-constant-expression* (*step*) $\rho \wedge$
152 *let stype* = *t-expression* (*step*) ρ in
281 *is-whole-number-type* (*stype*) $\rho \wedge$
218 *let by* = *evaluate-constant-expression* (*step*) ρ in
 by \neq 0 \wedge
 by \in *Step-range* \wedge
109 *wf-statement-sequence* (*body*) $\rho \wedge$
133 \neg *is-threatened-in-statement-seq* (*id*, *body*) ρ

annotations

Check each of the components of the for statement. The check that the control variable is declared in a variable declaration part of the block that contains the for statement is part of the consistency check for the procedure, function, or module that contains the for statement.

Dynamic Semantics

The for statement shall control the execution of a statement sequence while a sequence of values are assigned to the control variable.

The initial expression and the final expression shall be evaluated to give the initial and final values for the control variable. If the initial value is greater than the final value (in the case that the step is positive) or if the initial value is less than the final value (in the case that the step is negative), the execution of the for statement shall terminate.

Otherwise, a sequence of values shall be established. This sequence of values must be the longest sequence that satisfy the condition that the first element of the sequence of value shall be equal to the initial value; the last element of the sequence shall be less than or equal to the final value if the step is positive, or greater than or equal to the final value if the step is negative; and the difference between adjacent values in the sequence shall be equal to the step. For each element of the sequence in turn, the element shall be assigned to the control variable, and then the statement sequence shall be executed. The execution of the for statement shall terminate when the statement sequence has been executed for each value in the sequence.

After termination of the for statement the value of the control variable shall be undefined.

NOTES

- 1 The initial expression and the final expression are evaluated only once at the start of the execution of the for statement.
- 2 The statement sequence of the for statement is not be executed if the value of the initial expression is greater than the value of the final expression (in the case that the step is positive) or if the value of the initial expression is less than the value of the final expression (in the case that the step is negative).
- 3 The execution of a for statement may be terminated by the execution of an exit statement if the for statement is contained within a loop statement, or by a return statement.
- 4 The evaluation of the expressions that define the initial or final values could produce a transformation of the program state, i.e. have a side-effect.

operations

$$\begin{aligned} m\text{-for-statement} : \text{For-statement} &\rightarrow \text{Environment} \xrightarrow{o} () \\ m\text{-for-statement}(mk\text{-For-statement}(id, initial, final, step, body))\rho &\triangleq \\ \text{??} \quad \text{def } \rho_{cont} = c\text{-tix}(\{\text{TERMINATION} &\hspace{10em} \mapsto \\ \text{112}(\text{assign}(mk\text{-Entire-designator}(id), \text{UNDEFINED})\rho; c\text{-exit}(\text{TERMINATION})\rho\}))\rho; & \\ \text{146} \quad \text{def } cvar = m\text{-entire-designator}(mk\text{-Entire-designator}(id))\rho; & \\ \text{153} \quad \text{def } from = m\text{-expression}(initial)\rho; & \\ \text{153} \quad \text{def } to = m\text{-expression}(final)\rho; & \\ \text{153} \quad \text{def } by = m\text{-expression}(step)\rho; & \\ \text{129} \quad \text{let } control = \text{iteration-values}(from, to, by) \text{ in} & \\ \text{129} \quad \text{execute-for-statement}(cvar, control, body)\rho_{cont} & \end{aligned}$$

annotations

The initial expression and the final expression are evaluated in any order; the iteration sequence for the loop is built from the initial value, step, and final value and then the loop is executed using the iteration sequence.

Auxiliary Definitions

operations

$execute\text{-}for\text{-}statement : Variable \times Ordinal\text{-}value^* \times Statement\text{-}sequence \rightarrow Environment \xrightarrow{o} ()$

$execute\text{-}for\text{-}statement(cvar, control, body)\rho \triangleq$

if $control = []$

112 then $assign(cvar, UNDEFINED)\rho$

112 else $(assign(cvar, hd\ control)\rho ;$

109 $m\text{-}statement\text{-}sequence(body)\rho ;$

129 $execute\text{-}for\text{-}statement(cvar, tl\ control, body)\rho)$

annotations

If the iteration sequence is empty, terminate the execution of the loop and assign an undefined value to the control variable. If the iteration sequence is not empty, assign the next iteration value to the control variable, execute the body of the loop and execute the for statement again with an iteration sequence having the value used for the current iteration removed.

functions

$iteration\text{-}values : Ordinal\text{-}value \times Ordinal\text{-}value \times Ordinal\text{-}value \rightarrow Ordinal\text{-}value^*$

$iteration\text{-}values(from, to, by) \triangleq$

let $vals =$ if $by > 0$

then $iteration\text{-}up\text{-}sequence(from, to)$

else $iteration\text{-}down\text{-}sequence(from, to)$ in

$[vals(i) \mid i \in \text{inds } vals \cdot (i - 1) \bmod \text{abs } by = 0]$

annotations

This function constructs the sequence of values that may be assigned to the control variable. The sequence of values starts with the initial value. The last value in the sequence is less than or equal to the final value if the step is positive, or greater than or equal to the final value if the step is negative. The difference between adjacent values in the sequence is equal to the step.

If the initial value is greater than the final value, in the case that the step is positive, or if the initial value is less than the final value, in the case that the step is negative, the sequence of values is empty.

functions

$iteration\text{-}up\text{-}sequence : Ordinal\text{-}value \times Ordinal\text{-}value \rightarrow Ordinal\text{-}value^*$

$iteration\text{-}up\text{-}sequence(initial, final) \triangleq$

if $initial > final$

then $[]$

elseif $initial = final$

then $[initial]$

else $[initial] \frown iteration\text{-}up\text{-}sequence(succ(initial), final)$

annotations

Construct the sequence of consecutive values from the initial and final values.

functions

$iteration\text{-}down\text{-}sequence : Ordinal\text{-}value \times Ordinal\text{-}value \rightarrow Ordinal\text{-}value^*$

$iteration\text{-}down\text{-}sequence(first, last) \triangleq$

if $first < last$

then $[]$

elseif $first = last$

then $[first]$

else $[first] \frown iteration\text{-}down\text{-}sequence(pred(first), last)$

annotations

Construct a sequence of iteration values from the first and last values.

functions

succ : *Ordinal-value* \rightarrow *Ordinal-value*

```
succ (val)  $\triangleq$   
  if is-Number(val)  
  then val + 1  
  elseif is-B(val)  
  then true  
  else let mk-Enumerated-value(value, order) = val in  
    let i = index(value, order) in  
    mk-Enumerated-value(order(i + 1), order);
```

pred : *Ordinal-value* \rightarrow *Ordinal-value*

```
pred (val)  $\triangleq$   
  if is-Number(val)  
  then val - 1  
  elseif is-B(val)  
  then false  
  else let mk-Enumerated-value(value, order) = val in  
    let i = index(value, order) in  
    mk-Enumerated-value(order(i - 1), order)
```

6.5.14 Auxiliary Functions

6.5.14.1 Well-formed Control Variables

The control variable of each for statement in the body of a block shall be declared as a simple variable of ordinal type in a variable declaration part of the block that closest contains the for statement that uses it.

NOTE — A control variable cannot be an actual VAR parameter of any proper procedure or function procedure, because a control variable must be declared in the declaration part of a block in which it is used. The restriction that a control variable is of an ordinal type is checked by the well-formed condition for each for statement (see section 6.5.1.3).

A control variable shall not be imported by a module declaration.

A control variable shall not be exported (this restriction is checked by the well-formed condition for a module — see section 6.2.9).

A control variable shall not be threatened by a statement of the statement sequence of the for statement; (this restriction is checked by the well-formed condition for each for statement — see section 6.5.13).

A control variable shall not be threatened in a procedure declared in the declaration clause of the block that contains the for statement.

Threatening

A control variable shall be threatened in a proper procedure declaration if an identifier with the same spelling is not declared in the procedure heading, and the control variable is threatened in the block component of the procedure.

A control variable shall be threatened in a function procedure declaration if an identifier with the same spelling is not declared in the procedure heading, and the control variable is threatened in the block component of the procedure.

A control variable shall be threatened in a block if an identifier with the same spelling is not declared in a declaration of the block, and the control variable is threatened either by a procedure in a declaration of the block or by a statement of the statement sequence of the block.

A control variable shall be threatened in an assignment statement if it is the variable-designator of the assignment, if it is threatened by an expression in the variable designator component, or if it is threatened in the expression component of the assignment statement.

A control variable shall be threatened in a procedure call if it is threatened in an expression component of the procedure designator component of the procedure call or it is threatened in an argument of the procedure call.

A control variable shall be threatened in a return statement if the return statement is a function return statement and the control variable is threatened in the expression component of the return statement.

A control variable shall be threatened in a with statement if an identifier with the same spelling is not a field of the variable designated by the designator component of the with statement and the control variable is threatened in a statement of the statement sequence component of the with statement.

A control variable shall be threatened in an if statement if it is threatened by any Boolean expression components or if it is threatened in a statement of the statement sequence component.

A control variable shall be threatened in a case statement if it is threatened in the case selector or if it is threatened in a statement of the statement sequence component.

A control variable shall be threatened in a while statement if it is threatened in the Boolean expression component or if it is threatened in a statement of the statement sequence component.

A control variable shall be threatened in a repeat statement if it is threatened in a statement of the statement sequence component or if it is threatened in the Boolean expression component.

A control variable shall be threatened in a loop statement if it is threatened in a statement of the statement sequence component.

A control variable shall be threatened in a for statement if it is the control variable of the for statement, if it is threatened in the initial expression component, if it is threatened in the final expression component, or if it is threatened in a statement of the statement sequence component.

A control variable shall be threatened in a variable designator if it is threatened in an expression component of the variable designator.

A control variable shall be threatened in an expression if it is threatened as an element of the expression.

A control variable shall be threatened in a function call if it is threatened in the value designator component or if it is threatened in an argument of the function call.

A control variable shall be threatened as an argument to a procedure or function call if it is an actual parameter corresponding to a variable formal parameter or if it is threatened in an expression component of the argument.

NOTES

- 1 A control variable is not threatened by an empty statement or by an exit statement.
- 2 The standard procedures ADR, DEC, and INC are considered to take variable parameters.

Language Clarification

Control variables must be local and not exported or imported.

functions

$wf-control-variables : Declarations \times Statement^* \rightarrow Environment \rightarrow \mathbb{B}$

$wf-control-variables (decls, sts)\rho \triangleq$
 let $for-sts = x-statements-of-statement-seq(sts, For-statement)$ in
 let $for-ids = \{st.id \mid st \in for-sts\}$ in
 $\forall id \in for-ids .$
 $is-simple-variable(id, decls) \wedge$
 $\neg is-threatened-in-procedure(id, decls)\rho \wedge$
 $\neg is-imported-into-module(id, decls)$

annotations The result is true if and only if all the variables which are used as control variables are declared in the block, are not threatened by the statements or expressions contained in the block, are not threatened in procedures declared in the block, and are not imported by a module declared in the block.

functions

$is-simple-variable : Identifier \times Declarations \rightarrow \mathbb{B}$

$is-simple-variable(id, decls) \triangleq$
 $\exists mk-Variable-declaration(ids, -) \in elems\ decls \cdot id \in elems\ ids$

annotations The result is true if and only if there exists a variable declaration that contains a declaration of the variable.

annotations The identifier argument in the following operations is a control variable identifier and it designates a control variable.

functions

$is-threatened-in-procedure : Identifier \times Declarations \rightarrow Environment \rightarrow \mathbb{B}$

$is-threatened-in-procedure(id, decls)\rho \triangleq$
 $\exists proc \in elems\ decls \cap Proper-procedure-declaration \cdot$
 $is-threatened-in-proper-procedure(id, proc)\rho \vee$
 $\exists proc \in elems\ decls \cap Function-procedure-declaration \cdot$
 $is-threatened-in-function-procedure(id, proc)\rho$

annotations A control variable is threatened in the procedures declared in a block if there exists a procedure in the declarations component that threatens the control variable.

functions

$is-threatened-in-proper-procedure : Identifier \times Proper-procedure-declaration \rightarrow Environment \rightarrow \mathbb{B}$

$is-threatened-in-proper-procedure(id, procd)\rho \triangleq$
 let $mk-Proper-procedure-declaration(head, block) = procd$ in
 let $mk-Proper-procedure-heading(-, parms) = head$ in
 $id \notin identifiers-of-formal-parameter-list(parms) \wedge$
 $is-threatened-in-proper-procedure-block(id, block)\rho$

annotations A control variable is threatened in a proper procedure if an identifier with the same spelling is not declared in the procedure heading, and the control variable is threatened in the block component of the procedure.

functions

$is\text{-}threatened\text{-}in\text{-}function\text{-}procedure : Identifier \times Function\text{-}procedure\text{-}declaration \rightarrow Environment \rightarrow \mathbb{B}$

$is\text{-}threatened\text{-}in\text{-}function\text{-}procedure(id, fund)\rho \triangleq$
 $\quad \text{let } mk\text{-}Function\text{-}procedure\text{-}declaration(head, block) = fund \text{ in}$
 $\quad \text{let } mk\text{-}Function\text{-}procedure\text{-}heading(-, parms, -) = head \text{ in}$
 $\quad id \notin identifiers\text{-}of\text{-}formal\text{-}parameter\text{-}list(parms) \wedge$
 $\quad is\text{-}threatened\text{-}in\text{-}function\text{-}procedure\text{-}block(id, block)\rho$

annotations A control variable is threatened in a function procedure if an identifier with the same spelling is not declared in the procedure heading, and the control variable is threatened in the block component of the procedure.

functions

$is\text{-}threatened\text{-}in\text{-}proper\text{-}procedure\text{-}block : Identifier \times Proper\text{-}procedure\text{-}block \rightarrow Environment \rightarrow \mathbb{B}$

$is\text{-}threatened\text{-}in\text{-}proper\text{-}procedure\text{-}block(id, block)\rho \triangleq$
 $\quad \text{let } mk\text{-}Proper\text{-}procedure\text{-}block(decls, actions) = block \text{ in}$
 $\quad id \notin identifiers\text{-}declared\text{-}in(decls) \wedge$
 $\quad (is\text{-}threatened\text{-}in\text{-}procedure(id, decls)\rho \vee is\text{-}threatened\text{-}in\text{-}statement\text{-}seq(id, actions)\rho)$

annotations A control variable is threatened in a proper procedure block if an identifier with the same spelling is not declared in the declarations component of the block, and the control variable is threatened either by a procedure in the declarations component of the block or by the statement sequence of the block.

functions

$is\text{-}threatened\text{-}in\text{-}function\text{-}procedure\text{-}block : Identifier \times Function\text{-}procedure\text{-}block \rightarrow Environment \rightarrow \mathbb{B}$

$is\text{-}threatened\text{-}in\text{-}function\text{-}procedure\text{-}block(id, block)\rho \triangleq$
 $\quad \text{let } mk\text{-}Function\text{-}procedure\text{-}block(decls, actions) = block \text{ in}$
 $\quad id \notin identifiers\text{-}declared\text{-}in(decls) \wedge$
 $\quad (is\text{-}threatened\text{-}in\text{-}procedure(id, decls)\rho \vee is\text{-}threatened\text{-}in\text{-}statement\text{-}seq(id, actions)\rho)$

annotations A control variable is threatened in a function procedure block if an identifier with the same spelling is not declared in the declarations component of the block, and the control variable is threatened either by a procedure in the declarations component of the block or by the statement sequence of the block.

functions

$is\text{-}threatened\text{-}in\text{-}statement\text{-}seq : Identifier \times Statement^* \rightarrow Environment \rightarrow \mathbb{B}$

$is\text{-}threatened\text{-}in\text{-}statement\text{-}seq(id, sts)\rho \triangleq$
 $\quad \exists st \in \text{elems } sts \cdot$
 $\quad is\text{-}threatened\text{-}in\text{-}statement(id, st)\rho$

annotations A control variable is threatened in a statement sequence if at least one of the statements of the statement sequence threatens the control variable.

functions

$is\text{-}threatened\text{-}in\text{-}statement : Identifier \times Statement \rightarrow Environment \rightarrow \mathbb{B}$

$is\text{-}threatened\text{-}in\text{-}statement(id, st)\rho \triangleq$
 if $is\text{-}Empty\text{-}statement(st)$
 then false
 elseif $is\text{-}Assignment\text{-}statement(st)$
 then $is\text{-}threatened\text{-}in\text{-}assignment\text{-}statement(id, st)$
 elseif $is\text{-}Procedure\text{-}call(st)$
 then $is\text{-}threatened\text{-}in\text{-}procedure\text{-}call(id, st)$
 elseif $is\text{-}Return\text{-}statement(st)$
 then $is\text{-}threatened\text{-}in\text{-}return\text{-}statement(id, st)$
 elseif $is\text{-}With\text{-}statement(st)$
 then $is\text{-}threatened\text{-}in\text{-}with\text{-}statement(id, st)\rho$
 elseif $is\text{-}If\text{-}statement(st)$
 then $is\text{-}threatened\text{-}in\text{-}if\text{-}statement(id, st)\rho$
 elseif $is\text{-}Case\text{-}statement(st)$
 then $is\text{-}threatened\text{-}in\text{-}case\text{-}statement(id, st)\rho$
 elseif $is\text{-}While\text{-}statement(st)$
 then $is\text{-}threatened\text{-}in\text{-}while\text{-}statement(id, st)\rho$
 elseif $is\text{-}Repeat\text{-}statement(st)$
 then $is\text{-}threatened\text{-}in\text{-}repeat\text{-}statement(id, st)\rho$
 elseif $is\text{-}Loop\text{-}statement(st)$
 then $is\text{-}threatened\text{-}in\text{-}loop\text{-}statement(id, st)\rho$
 elseif $is\text{-}Exit\text{-}statement(st)$
 then false
 elseif $is\text{-}For\text{-}statement(st)$
 then $is\text{-}threatened\text{-}in\text{-}for\text{-}statement(id, st)\rho$
 else undefined

annotations A control variable is not threatened by an empty statement or by an exit statement.

functions

$is\text{-}threatened\text{-}in\text{-}assignment\text{-}statement : Identifier \times Assignment\text{-}statement \rightarrow \mathbb{B}$

$is\text{-}threatened\text{-}in\text{-}assignment\text{-}statement(id, st) \triangleq$
 let $mk\text{-}Assignment\text{-}statement(design, expr) = st$ in
 $design = mk\text{-}Entire\text{-}designator(id) \vee$
 $is\text{-}threatened\text{-}in\text{-}variable\text{-}designator(id, design) \vee$
 $is\text{-}threatened\text{-}in\text{-}expression(id, expr)$

annotations A control variable is threatened in an assignment statement if it is the variable designator of the assignment statement, or if it is threatened in any expression component of the assignment statement.

functions

$is\text{-}threatened\text{-}in\text{-}procedure\text{-}call : Identifier \times Procedure\text{-}call \rightarrow \mathbb{B}$

$is\text{-}threatened\text{-}in\text{-}procedure\text{-}call(id, st) \triangleq$
 let $mk\text{-}Procedure\text{-}call(design, aps) = st$ in
 $is\text{-}threatened\text{-}in\text{-}value\text{-}designator(id, design) \vee$
 $is\text{-}threatened\text{-}as\text{-}argument(id, aps)$

annotations A control variable is threatened in a procedure call if it is threatened as the procedure designator component of the procedure call or it is threatened in an argument of the procedure call.

functions

$is\text{-}threatened\text{-}in\text{-}return\text{-}statement : Identifier \times Return\text{-}statement \rightarrow \mathbb{B}$

$is\text{-}threatened\text{-}in\text{-}return\text{-}statement(id, st) \triangleq$

cases st :

$mk\text{-}Simple\text{-}return\text{-}statement(-) \rightarrow \text{false},$

$mk\text{-}Function\text{-}return\text{-}statement(-, expr) \rightarrow is\text{-}threatened\text{-}in\text{-}expression(id, expr)$

end

annotations

A control variable is threatened in a return statement if the return statement is a function return statement and the control variable is threatened in the expression component of the return statement.

functions

$is\text{-}threatened\text{-}in\text{-}with\text{-}statement : Identifier \times With\text{-}statement \rightarrow Environment \rightarrow \mathbb{B}$

$is\text{-}threatened\text{-}in\text{-}with\text{-}statement(id, st)\rho \triangleq$

let $mk\text{-}With\text{-}statement(design, body) = st$ in

if $is\text{-}Variable\text{-}designator(design)$

then let $type = t\text{-}variable\text{-}designator(design)\rho$ in

let $renv = field\text{-}types\text{-}of\text{-}record(type)\rho$ in

$id \notin \text{dom } renv \wedge$

$is\text{-}threatened\text{-}in\text{-}statement\text{-}seq(id, body)\rho$

elseif $is\text{-}Value\text{-}designator(design)$

then let $type = t\text{-}value\text{-}designator(design)\rho$ in

let $renv = field\text{-}types\text{-}of\text{-}record(type)\rho$ in

$id \notin \text{dom } renv \wedge$

$is\text{-}threatened\text{-}in\text{-}statement\text{-}seq(id, body)\rho$

else undefined

annotations

A control variable is threatened in a with statement if an identifier with the same spelling is not a field of the variable or value designated by the designator component of the with statement and the control variable is threatened in the statement sequence that is the body of the with statement.

$renv$ is the addition to the environment caused by the with statement. If the control variable identifier belongs to the domain of the mapping then within the with statement, it is to a field of the record that is being threatened, not the control variable.

functions

$is\text{-}threatened\text{-}in\text{-}if\text{-}statement : Identifier \times If\text{-}statement \rightarrow Environment \rightarrow \mathbb{B}$

$is\text{-}threatened\text{-}in\text{-}if\text{-}statement(id, st)\rho \triangleq$

let $mk\text{-}If\text{-}statement(thens, elsep) = st$ in

$(\exists mk\text{-}Guarded\text{-}statement(guard, body) \in \text{elems } thens .$

$is\text{-}threatened\text{-}in\text{-}expression(id, guard) \vee$

$is\text{-}threatened\text{-}in\text{-}statement\text{-}seq(id, body)\rho) \vee$

$(elsep \neq \text{nil} \wedge is\text{-}threatened\text{-}in\text{-}statement\text{-}seq(id, elsep)\rho)$

annotations

A control variable is threatened in an if statement if it is threatened by the Boolean expression component of a guarded statement, if it is threatened by the statement sequence component of a guarded statement, or if it is threatened in the statement sequence component of the ELSE clause.

functions

$is\text{-}threatened\text{-}in\text{-}case\text{-}statement : Identifier \times Case\text{-}statement \rightarrow Environment \rightarrow \mathbb{B}$

$is\text{-}threatened\text{-}in\text{-}case\text{-}statement(id, st)\rho \triangleq$
 let $mk\text{-}Case\text{-}statement(expr, cases, elsep) = st$ in
 137 $is\text{-}threatened\text{-}in\text{-}expression(expr) \vee$
 133 $(\exists mk\text{-}Case(-, body) \in cases \cdot is\text{-}threatened\text{-}in\text{-}statement\text{-}seq(id, body)\rho) \vee$
 133 $(elsep \neq nil \wedge is\text{-}threatened\text{-}in\text{-}statement\text{-}seq(id, elsep)\rho)$

annotations A control variable is threatened in a case statement if it is threatened in the case selector, if it is threatened in one of the statement sequences of the case clauses, or if it is threatened in the statement sequence of the else clause.

functions

$is\text{-}threatened\text{-}in\text{-}while\text{-}statement : Identifier \times While\text{-}statement \rightarrow Environment \rightarrow \mathbb{B}$

$is\text{-}threatened\text{-}in\text{-}while\text{-}statement(id, st)\rho \triangleq$
 let $mk\text{-}While\text{-}statement(expr, body) = st$ in
 137 $is\text{-}threatened\text{-}in\text{-}expression(id, expr) \vee$
 133 $is\text{-}threatened\text{-}in\text{-}statement\text{-}seq(id, body)\rho$

annotations A control variable is threatened in a while statement if it is threatened in the expression clause or if it is threatened in the statement sequence clause.

functions

$is\text{-}threatened\text{-}in\text{-}repeat\text{-}statement : Identifier \times Repeat\text{-}statement \rightarrow Environment \rightarrow \mathbb{B}$

$is\text{-}threatened\text{-}in\text{-}repeat\text{-}statement(id, st)\rho \triangleq$
 let $mk\text{-}Repeat\text{-}statement(body, expr) = st$ in
 133 $is\text{-}threatened\text{-}in\text{-}statement\text{-}seq(id, body)\rho \vee$
 137 $is\text{-}threatened\text{-}in\text{-}expression(id, expr)$

annotations A control variable is threatened in a repeat statement if it is threatened in the statement sequence clause or if it is threatened in the expression clause.

functions

$is\text{-}threatened\text{-}in\text{-}loop\text{-}statement : Identifier \times Loop\text{-}statement \rightarrow Environment \rightarrow \mathbb{B}$

$is\text{-}threatened\text{-}in\text{-}loop\text{-}statement(id, st)\rho \triangleq$
 let $mk\text{-}Loop\text{-}statement(body) = st$ in
 133 $is\text{-}threatened\text{-}in\text{-}statement\text{-}seq(id, body)\rho$

annotations A control variable is threatened in a loop statement if it is threatened in the statement sequence clause.

functions

$is\text{-}threatened\text{-}in\text{-}for\text{-}statement : Identifier \times For\text{-}statement \rightarrow Environment \rightarrow \mathbb{B}$

$is\text{-}threatened\text{-}in\text{-}for\text{-}statement(cid, st)\rho \triangleq$
 let $mk\text{-}For\text{-}statement(id, initial, final, -, body) = st$ in
 $cid = id \vee$
 137 $is\text{-}threatened\text{-}in\text{-}expression(cid, initial) \vee$
 137 $is\text{-}threatened\text{-}in\text{-}expression(cid, final) \vee$
 133 $is\text{-}threatened\text{-}in\text{-}statement\text{-}seq(cid, body)\rho$

annotations A control variable is threatened in a for statement if it is the control variable of the for statement, if it is threatened in the initial expression, if it is threatened in the final expression, or if it is threatened in the statement sequence that is the body of the for statement.

functions

$is\text{-}threatened\text{-}in\text{-}variable\text{-}designator : Identifier \times Variable\text{-}designator \rightarrow \mathbb{B}$

$is\text{-}threatened\text{-}in\text{-}variable\text{-}designator(id, vdesig) \triangleq$

cases $vdesig$:

$mk\text{-}Entire\text{-}designator(-) \rightarrow false,$

137 $mk\text{-}Indexed\text{-}designator(design, expr) \rightarrow is\text{-}threatened\text{-}in\text{-}variable\text{-}designator(id, design) \vee$
 $is\text{-}threatened\text{-}in\text{-}expression(id, expr),$

137 $mk\text{-}Selected\text{-}designator(design, -) \rightarrow is\text{-}threatened\text{-}in\text{-}variable\text{-}designator(id, design),$

137 $mk\text{-}Dereferenced\text{-}designator(design) \rightarrow is\text{-}threatened\text{-}in\text{-}variable\text{-}designator(id, design)$

end

annotations A control variable is threatened in a variable designator if it is threatened in any expression component of the variable designator.

functions

$is\text{-}threatened\text{-}in\text{-}expression : Identifier \times Expression \rightarrow \mathbb{B}$

$is\text{-}threatened\text{-}in\text{-}expression(id, expr) \triangleq$

if $is\text{-}Infix\text{-}expression(expr)$

137 then $is\text{-}threatened\text{-}in\text{-}infix\text{-}expression(id, expr)$

elseif $is\text{-}Prefix\text{-}expression(expr)$

137 then $is\text{-}threatened\text{-}in\text{-}prefix\text{-}expression(id, expr)$

elseif $is\text{-}Value\text{-}designator(expr)$

138 then $is\text{-}threatened\text{-}in\text{-}value\text{-}designator(id, expr)$

elseif $is\text{-}Function\text{-}call(expr)$

138 then $is\text{-}threatened\text{-}in\text{-}function\text{-}call(id, expr)$

elseif $is\text{-}Value\text{-}constructor(expr)$

138 then $is\text{-}threatened\text{-}in\text{-}value\text{-}constructor(id, expr)$

elseif $is\text{-}Constant\text{-}literal(expr)$

then false

else undefined

annotations A control variable is threatened in an expression if it is threatened as an element of the expression.

functions

$is\text{-}threatened\text{-}in\text{-}infix\text{-}expression : Identifier \times Infix\text{-}expression \rightarrow \mathbb{B}$

$is\text{-}threatened\text{-}in\text{-}infix\text{-}expression(id, expr) \triangleq$

let $mk\text{-}Infix\text{-}expression(left, -, right) = expr$ in

137 $is\text{-}threatened\text{-}in\text{-}expression(id, left) \vee$

137 $is\text{-}threatened\text{-}in\text{-}expression(id, right)$

annotations A control variable is threatened in an infix expression if it is threatened in an expression component of the infix expression.

functions

$is\text{-}threatened\text{-}in\text{-}prefix\text{-}expression : Identifier \times Prefix\text{-}expression \rightarrow \mathbb{B}$

$is\text{-}threatened\text{-}in\text{-}prefix\text{-}expression(id, pexpr) \triangleq$

let $mk\text{-}Prefix\text{-}expression(-, expr) = pexpr$ in

137 $is\text{-}threatened\text{-}in\text{-}expression(id, expr)$

annotations A control variable is threatened in an prefix expression if it is threatened in the expression component of the prefix expression.

functions

$is\text{-}threatened\text{-}in\text{-}value\text{-}designator : Identifier \times Value\text{-}designator \rightarrow \mathbb{B}$

$is\text{-}threatened\text{-}in\text{-}value\text{-}designator(id, vdesig) \triangleq$

cases $vdesig$:

138 $mk\text{-}Entire\text{-}value(-) \rightarrow \text{false},$
 137 $mk\text{-}Indexed\text{-}value(design, expr) \rightarrow is\text{-}threatened\text{-}in\text{-}value\text{-}designator(id, design) \vee$
 137 $is\text{-}threatened\text{-}in\text{-}expression(id, expr),$
 138 $mk\text{-}Selected\text{-}value(design, -) \rightarrow is\text{-}threatened\text{-}in\text{-}value\text{-}designator(id, design),$
 138 $mk\text{-}Dereferenced\text{-}value(design) \rightarrow is\text{-}threatened\text{-}in\text{-}value\text{-}designator(id, design)$
 end

annotations A control variable is threatened in a value designator if it is threatened in any expression component of the value designator.

functions

$is\text{-}threatened\text{-}in\text{-}function\text{-}call : Identifier \times Function\text{-}call \rightarrow \mathbb{B}$

$is\text{-}threatened\text{-}in\text{-}function\text{-}call(id, expr) \triangleq$

let $mk\text{-}Function\text{-}call(design, args) = expr$ in
 138 $is\text{-}threatened\text{-}in\text{-}value\text{-}designator(id, design) \vee$
 139 $is\text{-}threatened\text{-}as\text{-}argument(id, args)$

annotations A control variable is threatened in a function call if it is threatened as the value designator component or if it is threatened in an argument of the function call.

functions

$is\text{-}threatened\text{-}in\text{-}value\text{-}constructor : Identifier \times Value\text{-}constructor \rightarrow \mathbb{B}$

$is\text{-}threatened\text{-}in\text{-}value\text{-}constructor(id, expr) \triangleq$

cases $expr$:

138 $mk\text{-}Array\text{-}constructor(-, sval) \rightarrow is\text{-}threatened\text{-}in\text{-}structured\text{-}value(sval),$
 138 $mk\text{-}Record\text{-}constructor(-, sval) \rightarrow is\text{-}threatened\text{-}in\text{-}structured\text{-}value(sval),$
 138 $mk\text{-}Set\text{-}constructor(-, def) \rightarrow is\text{-}threatened\text{-}in\text{-}set\text{-}constructor(def)$
 end

annotations A control variable is threatened in a value constructor if it is threatened in any expression component of the value constructor.

functions

$is\text{-}threatened\text{-}in\text{-}structured\text{-}value : Identifier \times Structured\text{-}value \rightarrow \mathbb{B}$

$is\text{-}threatened\text{-}in\text{-}structured\text{-}value(id, sval) \triangleq$

137 $\exists mk\text{-}Element(val, -) \in \text{elems } sval \cdot is\text{-}threatened\text{-}in\text{-}expression(id, val)$

annotations A control variable is threatened in a structured value if it is threatened in any expression component of the structured value. Note that a control variable cannot be threatened in the replication component as it is a constant expression.

functions

$is\text{-}threatened\text{-}in\text{-}set\text{-}constructor : Identifier \times Set\text{-}definition \rightarrow \mathbb{B}$

$is\text{-}threatened\text{-}in\text{-}set\text{-}constructor(id, set) \triangleq$

$\exists mem \in set \cdot$

cases mem :

137 $mk\text{-}Singleton(expr) \rightarrow is\text{-}threatened\text{-}in\text{-}expression(id, expr),$
 137 $mk\text{-}Interval(min, max) \rightarrow is\text{-}threatened\text{-}in\text{-}expression(id, min) \vee$
 137 $is\text{-}threatened\text{-}in\text{-}expression(id, max)$
 end

annotations A control variable is threatened in a set constructor if it is threatened in any expression component of the set constructor.

functions

$is\text{-}threatened\text{-}as\text{-}argument : Identifier \times Actual\text{-}parameters \rightarrow \mathbb{B}$

$is\text{-}threatened\text{-}as\text{-}argument(id, aps) \triangleq$

$\exists ap \in \text{elems } aps \cdot$
 if $is\text{-}Variable\text{-}designator(ap)$
 then $mk\text{-}Entire\text{-}variable(id) = ap \vee$
 $is\text{-}threatened\text{-}in\text{-}variable\text{-}designator(id, ap)$
 elseif $is\text{-}Expression(ap)$
 then $is\text{-}threatened\text{-}in\text{-}expression(id, ap)$
 elseif $is\text{-}Type\text{-}parameter(ap)$
 then false
 else false

annotations A control variable is threatened as an argument to a procedure or function call if it is an actual parameter corresponding to a variable formal parameter or if it is threatened in an expression component of the argument.

functions

$is\text{-}imported\text{-}into\text{-}module : Identifier \times Declarations \rightarrow \mathbb{B}$

$is\text{-}imported\text{-}into\text{-}module(id, decls) \triangleq$

$\exists mk\text{-}Module\text{-}declaration(-, imports, -, -, -) \in \text{elems } decls \cdot$
 $id \in imports\text{-}of(imports)$

annotations Check if a control variable is imported by a module declaration of the block.

6.5.14.2 Well-formed Return Statements

functions

$wf\text{-}simple\text{-}return\text{-}statements : Statement^* \rightarrow \mathbb{B}$

$wf\text{-}simple\text{-}return\text{-}statements(sts) \triangleq$

let $returns = return\text{-}statements\text{-}of(sts, Return\text{-}statement)$ in
 $returns \subseteq Simple\text{-}return\text{-}Statement$;

$wf\text{-}function\text{-}return\text{-}statements : Typed \times Block\text{-}body \rightarrow Environment \rightarrow \mathbb{B}$

$wf\text{-}function\text{-}return\text{-}statements(rtype, sts)\rho \triangleq$

let $returns = return\text{-}statements\text{-}of(sts, Return\text{-}statement)$ in
 $card\ returns \geq 1 \wedge$
 $returns \subseteq Function\text{-}return\text{-}Statement \wedge$
 $\forall ret \in returns \cdot$
 let $etype = t\text{-}expression(ret.expr)\rho$ in
 $is\text{-}assignment\text{-}compatible(rtype, etype)\rho$

annotations Check that the return statement components of the statement sequence are all function return statements and that for each return statement, the type of the expression component of the return statement is assignment compatible with the return type of the function procedure.

functions

$return\text{-}statements\text{-}of : Statement^* \rightarrow Return\text{-}statement\text{-}set$

$return\text{-}statements\text{-}of(sts) \triangleq$

$x\text{-}statements\text{-}of\text{-}statement\text{-}seq(sts, Return\text{-}statement)$

annotations The return-statements of a statement sequence is the set containing all the return statements contained in the statement clauses of the statement sequence.

6.5.14.3 Well-formed Exit Statements

functions

$wf\text{-}exit\text{-}statements : Statement^* \rightarrow \mathbb{B}$

$wf\text{-}exit\text{-}statements(sts) \triangleq$

140 $free\text{-}exit\text{-}statements\text{-}of\text{-}statement\text{-}seq(sts) = \{ \}$

annotations This function returns true if and only if each exit statement of the statement sequence is textually contained within a loop statement.

A free exit statement is one which is not contained within a loop statement, thus the loop statements of a statement sequence are well-formed if the statement sequence contains no free exit statements.

functions

$free\text{-}exit\text{-}statements\text{-}of\text{-}statement\text{-}seq : Statement^* \rightarrow Exit\text{-}statement\text{-}set$

$free\text{-}exit\text{-}statements\text{-}of\text{-}statement\text{-}seq(sts) \triangleq$

140 $\bigcup \{ free\text{-}exit\text{-}statements\text{-}of\text{-}statement(st) \mid st \in \text{elems } sts \}$

annotations The free-exit-statements of a statement sequence is the set containing all the exit statements contained in the statement clauses of the statement sequence which are not contained in loop statements.

functions

$free\text{-}exit\text{-}statements\text{-}of\text{-}statement : Statement \rightarrow Exit\text{-}statement\text{-}set$

$free\text{-}exit\text{-}statements\text{-}of\text{-}statement(st) \triangleq$

if $is\text{-}Empty\text{-}statement(st)$

then $\{ \}$

elseif $is\text{-}Assignment\text{-}statement(st)$

then $\{ \}$

elseif $is\text{-}Procedure\text{-}call(st)$

then $\{ \}$

elseif $is\text{-}Return\text{-}statement(st)$

then $\{ \}$

elseif $is\text{-}With\text{-}statement(st)$

141 then $free\text{-}exit\text{-}statements\text{-}of\text{-}with\text{-}statement(st)$

elseif $is\text{-}If\text{-}statement(st)$

141 then $free\text{-}exit\text{-}statements\text{-}of\text{-}if\text{-}statement(st)$

elseif $is\text{-}Case\text{-}statement(st)$

141 then $free\text{-}exit\text{-}statements\text{-}of\text{-}case\text{-}statement(st)$

elseif $is\text{-}While\text{-}statement(st)$

141 then $free\text{-}exit\text{-}statements\text{-}of\text{-}while\text{-}statement(st)$

elseif $is\text{-}Repeat\text{-}statement(st)$

141 then $free\text{-}exit\text{-}statements\text{-}of\text{-}repeat\text{-}statement(st)$

elseif $is\text{-}Loop\text{-}statement(st)$

then $\{ \}$

elseif $is\text{-}Exit\text{-}statement(st)$

then $\{ st \}$

elseif $is\text{-}For\text{-}statement(st)$

142 then $free\text{-}exit\text{-}statements\text{-}of\text{-}for\text{-}statement(st)$

else undefined

annotations The free exit statements of an exit statement is a set containing that statement. The free exit statements of any other statement which is not composed of any parts that are themselves statements is the empty set. The free exit statements of a loop statement is the empty set. The free exit statements of any other statement which is composed of parts that are themselves statements is the free exit statements contained in the statement sequence component parts.

functions

free-exit-statements-of-with-statement : *With-statement* \rightarrow *Exit-statement-set*

free-exit-statements-of-with-statement(*st*) \triangleq
 let *mk-With-statement*(-, *body*) = *st* in
free-exit-statements-of-statement-seq(*body*)

annotations The free exit statements contained in a with statement are the free exit statements contained in the statement sequence clause of the with statement.

functions

free-exit-statements-of-if-statement : *If-statement* \rightarrow *Exit-statement-set*

free-exit-statements-of-if-statement(*st*) \triangleq
 let *mk-If-statement*(*thens*, *elsep*) = *st* in
 140 let *thens-set* = $\bigcup \{ \text{free-exit-statements-of-statement-seq}(gs.\text{body}) \mid gs \in \text{elems } \text{thens} \}$ in
 let *elsep-set* = if *elsep* = nil
 then { }
 140 else *free-exit-statements-of-statement-seq*(*elsep*) in
thens-set \cup *elsep-set*

annotations The free exit statements contained in an if statement are the free exit statements contained in the statement sequence component of the if statement.

functions

free-exit-statements-of-case-statement : *Case-statement* \rightarrow *Exit-statement-set*

free-exit-statements-of-case-statement(*st*) \triangleq
 let *mk-Case-statement*(-, *cases*, *elsep*) = *st* in
 140 let *cases-set* = $\bigcup \{ \text{free-exit-statements-of-statement-seq}(c.\text{body}) \mid c \in \text{cases} \}$ in
 let *elsep-set* = if *elsep* = nil
 then { }
 140 else *free-exit-statements-of-statement-seq*(*elsep*) in
cases-set \cup *elsep-set*

annotations The free exit statements contained in a case statement are the free exit statements contained in the statement sequence components of the case statement.

functions

free-exit-statements-of-while-statement : *While-statement* \rightarrow *Exit-statement-set*

free-exit-statements-of-while-statement(*st*) \triangleq
 let *mk-While-statement*(-, *body*) = *st* in
 140 *free-exit-statements-of-statement-seq*(*body*)

annotations The free exit statements contained in a while statement are the free exit statements contained in the statement sequence component of the while statement.

functions

free-exit-statements-of-repeat-statement : *Repeat-statement* \rightarrow *Exit-statement-set*

free-exit-statements-of-repeat-statement(*st*) \triangleq
 let *mk-Repeat-statement*(*body*, -) = *st* in
 140 *free-exit-statements-of-statement-seq*(*body*)

annotations The free exit statements contained in a repeat statement are the free exit statements contained in the statement sequence component of the repeat statement.

functions

free-exit-statements-of-for-statement : *For-statement* \rightarrow *Exit-statement-set*

free-exit-statements-of-for-statement (*st*) \triangleq
 let *mk-For-statement*(-, -, -, -, *body*) = *st* in
free-exit-statements-of-statement-seq (*body*)

140

annotations The free exit statements contained in a for statement are the free exit statements contained in the statement sequence component of the for statement.

6.5.14.4 Statement Classes of Statement Sequences

functions

x-statements-of-statement-seq : *Statement** \times *Statement-set* \rightarrow *Statement-set*

x-statements-of-statement-seq (*sts*, *stats*) \triangleq
 $\bigcup \{x\text{-statements-of-statement}(st, stats) \mid st \in \text{elems } sts\}$

142

annotations This function returns a set containing all the statements contained in the statement sequence which are members of the set defined by the second argument.

functions

x-statements-of-statement : *Statement* \times *Statement-set* \rightarrow *Statement-set*

x-statements-of-statement (*st*, *stats*) \triangleq
 let *ast* = if *st* \in *stats*
 then {*st*}
 else { } in
 let *rst* = if *is-With-statement*(*st*)
 then *x-statements-of-with-statement* (*st*, *stats*)
 elseif *is-If-statement*(*st*)
 then *x-statements-of-if-statement* (*st*, *stats*)
 elseif *is-Case-statement*(*st*)
 then *x-statements-of-case-statement* (*st*, *stats*)
 elseif *is-While-statement*(*st*)
 then *x-statements-of-while-statement* (*st*, *stats*)
 elseif *is-Repeat-statement*(*st*)
 then *x-statements-of-repeat-statement* (*st*, *stats*)
 elseif *is-Loop-statement*(*st*)
 then *x-statements-of-loop-statement* (*st*, *stats*)
 elseif *is-For-statement*(*st*)
 then *x-statements-of-for-statement* (*st*, *stats*)
 else { } in
ast \cup *bst*;

142

143

143

143

143

143

144

x-statements-of-with-statement : *With-statement* \times *Statement-set* \rightarrow *Statement-set*

x-statements-of-with-statement (*st*, *stats*) \triangleq
 let *mk-With-statement*(-, *body*) = *st* in
x-statements-of-statement-seq (*body*, *stats*)

142

annotations The control variable identifiers of a with statement are the control variable identifiers contained in the statement sequence clause of the with statement.

functions

$x\text{-statements-of-if-statement} : \text{If-statement} \times \text{Statement-set} \rightarrow \text{Statement-set}$

$x\text{-statements-of-if-statement}(st, stats) \triangleq$
let $mk\text{-If-statement}(thens, elsep) = st$ in
let $thens\text{-set} = \bigcup \{x\text{-statements-of-statement-seq}(gs.body, stats) \mid gs \in \text{elems } thens\}$ in
let $elsep\text{-set} = \text{if } elsep = \text{nil}$
then $\{\}$
else $x\text{-statements-of-statement-seq}(elsep, stats)$ in
 $thens\text{-set} \cup elsep\text{-set}$

annotations The control variable identifiers of an if statement are the control variable identifiers contained in the statement sequence components of the if statement.

functions

$x\text{-statements-of-case-statement} : \text{Case-statement} \times \text{Statement-set} \rightarrow \text{Statement-set}$

$x\text{-statements-of-case-statement}(st, stats) \triangleq$
let $mk\text{-Case-statement}(-, cases, elsep) = st$ in
let $cases\text{-set} = \bigcup \{x\text{-statements-of-statement-seq}(c.body, stats) \mid c \in cases\}$ in
let $elsep\text{-set} = \text{if } elsep = \text{nil}$
then $\{\}$
else $x\text{-statements-of-statement-seq}(elsep, stats)$ in
 $cases\text{-set} \cup elsep\text{-set}$

annotations The control variable identifiers of a case statement are the control variable identifiers contained in the statement sequence clauses of the case statement.

functions

$x\text{-statements-of-while-statement} : \text{While-statement} \times \text{Statement-set} \rightarrow \text{Statement-set}$

$x\text{-statements-of-while-statement}(st, stats) \triangleq$
let $mk\text{-While-statement}(-, body) = st$ in
 $x\text{-statements-of-statement-seq}(body, stats)$

annotations The control variable identifiers of a while statement are the control variable identifiers contained in the statement sequence clause of the while statement.

functions

$x\text{-statements-of-repeat-statement} : \text{Repeat-statement} \times \text{Statement-set} \rightarrow \text{Statement-set}$

$x\text{-statements-of-repeat-statement}(st, stats) \triangleq$
let $mk\text{-Repeat-statement}(body, -) = st$ in
 $x\text{-statements-of-statement-seq}(body, stats)$

annotations The control variable identifiers of a repeat statement are the control variable identifiers contained in the statement sequence clause of the repeat statement.

functions

$x\text{-statements-of-loop-statement} : \text{Loop-statement} \times \text{Statement-set} \rightarrow \text{Statement-set}$

$x\text{-statements-of-loop-statement}(st, stats) \triangleq$
let $mk\text{-Loop-statement}(body) = st$ in
 $x\text{-statements-of-statement-seq}(body, stats)$

annotations The control variable identifiers of a loop statement are the control variable identifiers contained in the statement sequence clause of the loop statement.

functions

$x\text{-statements-of-for-statement} : \text{For-statement} \times \text{Statement-set} \rightarrow \text{Statement-set}$

$x\text{-statements-of-for-statement}(st, stats) \triangleq$
let $mk\text{-For-statement}(-, -, -, -, body) = st$ in
 $x\text{-statements-of-statement-seq}(body, stats)$

142

annotations

The control variable identifiers of a for statement is the control variable identifier component of the for statement, together with the control variable identifiers contained in the statement sequence clause of the for statement.

6.6 Variable Designators

A variable designator denotes a variable, or a component of a variable.

Concrete Syntax

variable designator = entire designator | indexed designator | selected designator | dereferenced designator ;

Abstract Syntax

types

$$\begin{aligned} \text{Variable-designator} = & \text{Entire-designator} \\ & | \text{Indexed-designator} \\ & | \text{Selected-designator} \\ & | \text{Dereferenced-designator} \end{aligned}$$

Static Semantics

functions

$$\text{wf-variable-designator} : \text{Variable-designator} \rightarrow \text{Environment} \rightarrow \mathbb{B}$$
$$\text{wf-variable-designator}(vdesig)\rho \triangleq$$

cases $vdesig$:

146 $mk\text{-Entire-designator}(-) \rightarrow \text{wf-entire-designator}(vdesig)\rho,$
147 $mk\text{-Indexed-designator}(-, -) \rightarrow \text{wf-indexed-designator}(vdesig)\rho,$
148 $mk\text{-Selected-designator}(-, -) \rightarrow \text{wf-selected-designator}(vdesig)\rho,$
150 $mk\text{-Dereferenced-designator}(-) \rightarrow \text{wf-dereferenced-designator}(vdesig)\rho$
end;

$$t\text{-variable-designator} : \text{Variable-designator} \rightarrow \text{Environment} \rightarrow \text{Variable-typed}$$
$$t\text{-variable-designator}(vdesig)\rho \triangleq$$

cases $vdesig$:

146 $mk\text{-Entire-designator}(-) \rightarrow t\text{-entire-designator}(vdesig)\rho,$
147 $mk\text{-Indexed-designator}(-, -) \rightarrow t\text{-indexed-designator}(vdesig)\rho,$
149 $mk\text{-Selected-designator}(-, -) \rightarrow t\text{-selected-designator}(vdesig)\rho,$
150 $mk\text{-Dereferenced-designator}(-) \rightarrow t\text{-dereferenced-designator}(vdesig)\rho$
end

Dynamic Semantics

operations

$$m\text{-variable-designator} : \text{Variable-designator} \rightarrow \text{Environment} \xrightarrow{o} \text{Variable}$$
$$m\text{-variable-designator}(vdesig)\rho \triangleq$$

cases $vdesig$:

146 $mk\text{-Entire-designator}(-) \rightarrow m\text{-entire-designator}(vdesig)\rho,$
148 $mk\text{-Indexed-designator}(-, -) \rightarrow m\text{-indexed-designator}(vdesig)\rho,$
149 $mk\text{-Selected-designator}(-, -) \rightarrow m\text{-selected-designator}(vdesig)\rho,$
151 $mk\text{-Dereferenced-designator}(-) \rightarrow m\text{-dereferenced-designator}(vdesig)\rho$
end

6.6.1 Entire Designators

An entire designator consists of a qualified identifier referring to a variable, or to a formal parameter of a procedure.

Concrete Syntax

entire designator = qualified identifier ;

Abstract Syntax

types

Entire-designator :: *qid* : *Qualident*

Static Semantics

An entire designator shall be declared as a variable, or be a formal parameter of a procedure.

NOTE — Within the body of a with statement, a field identifier of a record variable which has been specified in the designator component of the with statement is also an entire designator — see section 6.5.6.

The type of an entire designator shall be defined by the declaration of the variable or formal parameter denoted by the designator.

functions

wf-entire-designator : *Entire-designator* \rightarrow *Environment* \rightarrow \mathbb{B}

wf-entire-designator(*mk-Entire-designator*(*qid*)) $\rho \triangleq$

272 *is-variable*(*qid*) $\rho \wedge$

?? *wf-qualident*(*qid*) ρ

annotations Check that the qualified identifier component was defined or declared as a variable, and that any qualification is by a module name.

;

t-entire-designator : *Entire-designator* \rightarrow *Environment* \rightarrow *Variable-typed*

t-entire-designator(*mk-Entire-designator*(*qid*)) $\rho \triangleq$

272 *type-of-variable*(*qid*) ρ

Dynamic Semantics

The result of the evaluation of an entire designator shall be the variable or formal parameter associated with the qualified identifier.

operations

m-entire-designator : *Entire-designator* \rightarrow *Environment* \xrightarrow{o} *Variable*

m-entire-designator(*mk-Entire-designator*(*qid*)) $\rho \triangleq$

272 *return associated-variable*(*qid*) ρ

6.6.2 Indexed Designators

An indexed designator is an array designator followed by one or more index expressions, and denotes a component variable of a variable of array type.

Concrete Syntax

indexed designator = array designator, "[", index expression, { ",", index expression }, "]" ;

array designator = variable designator ;

index expression = ordinal expression ;

ordinal expression = expression ;

In the concrete syntax, if the array designator is itself an indexed designator, an abbreviation can be used. In the abbreviated form, a single comma shall replace the sequence "]" [" that occurs in the full form. The abbreviated form and the full form shall be equivalent.

NOTE — If the indexed designator is written in the abbreviated form, the number of index expressions cannot exceed the dimension of the array.

TO DO — An example of a multi-dimensional array to illustrate the concrete syntax to be inserted.

Abstract Syntax

types

Indexed-designator :: *desig* : *Variable-designator*
expr : *Expression*

annotations Only the the full form of the concrete syntax is represented in the abstract syntax.

Static Semantics

A variable designator that is indexed shall be of an array type. The type of the index expression shall be assignment compatible with the index type of the array type. The type of the variable denoted by the indexed designator shall be the component type of the array type.

functions

wf-indexed-designator : *Indexed-designator* → *Environment* → **B**

wf-indexed-designator (*mk-Indexed-designator*(*desig*, *expr*)) $\rho \triangleq$

145 *wf-variable-designator* (*desig*) $\rho \wedge$

152 *wf-expression* (*expr*) $\rho \wedge$

145 let *type* = *t-variable-designator* (*desig*) ρ in

288 (*is-array-type* (*type*) $\rho \vee$ *is-open-array* (*type*) ρ) \wedge

284 let *itype* = *index-type-of* (*type*) ρ in

162 *is-assignment-compatible* (*itype*, *t-expression* (*expr*) ρ) ρ

annotations Check the designator and expression components of the indexed designator. Check that the variable denoted by the designator is an array type and that the type of the index expression is assignment compatible with the index type as given in the definition or declaration of the variable.

;

t-indexed-designator : *Indexed-designator* → *Environment* → *Variable-typed*

t-indexed-designator (*mk-Indexed-designator*(*desig*, -)) $\rho \triangleq$

145 let *type* = *t-variable-designator* (*desig*) ρ in

284 *component-type-of* (*type*) ρ

annotations Return the type of the component of the array as given in the definition or declaration of the variable.

Dynamic Semantics

The result of the evaluation of a variable designator shall be an array variable; the index expression shall be evaluated to give an ordinal value. If this ordinal value is a value of the index type, the indexed designator shall be the component of the array selected by the ordinal value.

It shall be an exception if the result of evaluating the index expression is not a value belonging to the range of values defined by the index type.

The order of evaluation of the array designator and the index expression of an indexed designator may be implementation dependent.

operations

$m\text{-indexed-designator} : \text{Indexed-designator} \rightarrow \text{Environment} \xrightarrow{o} \text{Variable}$

$m\text{-indexed-designator}(mk\text{-Indexed-designator}(desig, expr))\rho \triangleq$

145 def $mk\text{-Array-variable}(array) = m\text{-variable-designator}(desig)\rho$;

?? 153 def $index = m\text{-expression}(expr)\rho$;

if $index \in \text{dom } array$

?? then return $array(index)$

306 else $mandatory\text{-exception}(\text{INDEX-RANGE})$

annotations Evaluate the designator and the index expression — the evaluation order for the designator and the index expression given above is one of those permitted; the index expression may be evaluated before the designator or both may be evaluated in parallel. If the index is within the range of values defined by the array type then return the appropriate component; otherwise raise an exception.

6.6.3 Selected Designators

A selected designator is a record designator followed by a field identifier, and denotes a component variable of a variable of a record type.

Concrete Syntax

selected designator = record designator, ".", field identifier ;

field identifier = identifier ;

Abstract Syntax

types

$Selected\text{-designator} :: desig : \text{Variable-designator}$
 $id : \text{Identifier}$

Static Semantics

The variable designator shall be declared to be a record type, and the field identifier shall be one of the identifiers or a tag identifier of the field lists of that record-type.

The type of the selected designator shall be given by the type of the field or tag identifier.

functions

$wf\text{-selected-designator} : \text{Selected-designator} \rightarrow \text{Environment} \rightarrow \mathbb{B}$

$wf\text{-selected-designator}(mk\text{-Selected-designator}(desig, id))\rho \triangleq$

145 $wf\text{-variable-designator}(desig)\rho \wedge$

145 let $type = t\text{-variable-designator}(desig)\rho$ in

278 $is\text{-record-type}(type)\rho \wedge$

$id \in \text{dom } field\text{-types-of-record}(type)\rho$

annotations Check the designator component and that it designates a record structure. Check that identifier component is one of the field list identifiers of the record.

;

$t\text{-selected-designator} : \text{Selected-designator} \rightarrow \text{Environment} \rightarrow \text{Typed}$

$t\text{-selected-designator}(mk\text{-Selected-designator}(desig, id))\rho \triangleq$

145 let $type = t\text{-variable-designator}(desig)\rho$ in

285 let $types = field\text{-types-of-record}(type)\rho$ in

?? $types(id)$

annotations Return the type of the field corresponding to the field list identifier.

Dynamic Semantics

The variable designator component shall be evaluated to give a record variable. The selected designator shall be the component of the record variable selected by the field identifier. It shall be an exception if the selected designator belongs to a variant that has a tag field and the variant is not active. If the selected designator belongs to one or more variants without a tag field the variants shall become active.

operations

$m\text{-selected-designator} : \text{Selected-designator} \rightarrow \text{Environment} \xrightarrow{o} \text{Variable}$

$m\text{-selected-designator}(mk\text{-Selected-designator}(desig, id))\rho \triangleq$

145 def $mk\text{-Record-variable}(fields) = m\text{-variable-designator}(desig)\rho$;

?? let $component = fields(id)$ in

149 ($is\text{-Variant-variable}(component) \rightarrow$ if $is\text{-active-component}(component)$

 then return $component$

306 else $non\text{-mandatory-exception}(\text{INACTIVE-VARIANT})$,

150 $is\text{-Tagged-variable}(component) \rightarrow (make\text{-component-active}(component)$;

 return $component$) ,

others \rightarrow return $component$)

annotations Evaluate the designator and select the appropriate field of the record. If the field is a variant component of a variant fields and the tag for that variant fields component exists, then check that the value of the tag is such that it selects that field. If there is no explicit tag, set the implicit tag to select the field.

Static Semantics

functions

$is\text{-active-component} : \text{Variable} \rightarrow \mathbb{B}$

$is\text{-active-component}(component) \triangleq$

$is\text{-Variant-variable}(component) \wedge$

let $mk\text{-Variant-variable}(var, tagloc, tagvals) = component$ in

?? let $tagval = variable\text{-value}(tagloc)$ in

$tagval \in tagvals \wedge$

149 $is\text{-active-component}(var)$

annotations A field of a record may occur in a variant which is nested in several variants. First check that the outer variant is an active variant; this is done by checking that the value of the tag field is consistent with the field component that is referenced. The function is called recursively to check the next (inner) variant.

Dynamic Semantics

operations

```
make-component-active : Variable  $\xrightarrow{o}$  ()  
make-component-active (component)  $\triangleq$   
  if is-Tagged-variable(component)  
  then (let mk-Tagged-variable(var, tagvar, tagvals) = component in  
    112   let val  $\in$  tagvals in  
    150   assign(tagvar, val) ;  
    make-component-active(var) )  
  else skip
```

annotations A field of a record may occur in a variant which is nested in several variants. The outermost variant is made active by assigning an appropriate value to the tag field associated with this variant. The function is called recursively to make the next innermost variant active.

6.6.4 Dereferenced Designators

A dereferenced designator is a pointer designator followed by a dereferencing operator, and denotes the variable (if any) referenced by the value of the pointer designator.

Concrete Syntax

dereferenced designator = pointer designator, dereferencing operator ;

pointer designator = variable designator ;

Abstract Syntax

types

Dereferenced-designator :: *desig* : *Variable-designator*

Static Semantics

The pointer designator shall be of a pointer type.

The type of a dereferenced designator shall be the bound type specified in the declaration of the pointer type.

functions

```
wf-dereferenced-designator : Dereferenced-designator  $\rightarrow$  Environment  $\rightarrow \mathbb{B}$   
wf-dereferenced-designator (mk-Dereferenced-designator(desig)) $\rho \triangleq$   
145   wf-variable-designator(desig) $\rho \wedge$   
145   let type = t-variable-designator(desig) $\rho$  in  
279   is-pointer-type(type) $\rho$ ;  
  
t-dereferenced-designator : Dereferenced-designator  $\times$  Environment  $\rightarrow$  Typed  
t-dereferenced-designator (mk-Dereferenced-designator(desig)) $\rho \triangleq$   
145   let type = t-variable-designator(desig) $\rho$  in  
284   bound-type-of(type) $\rho$ 
```

Dynamic Semantics

The variable designator shall be evaluated to give a variable. If the value of that variable is neither the value of the pervasive identifier NIL, nor a variable value associated with non-existing storage then the dereferenced designator shall be that value.

It shall be an exception if the value of the variable is either the nil-value or a variable value associated with non-existing storage.

operations

$m\text{-dereferenced-designator} : Dereferenced\text{-designator} \rightarrow Environment \xrightarrow{o} Variable$

$m\text{-dereferenced-designator}(mk\text{-Dereferenced-designator}(desig))\rho \triangleq$

```
185   def var = m-variable-designator(desig) ρ;  
184   def val = value-of-a-variable(var) ;  
      ( exists (val) → return val,  
306   var = nil    → mandatory-exception(NIL-DEREFERENCE) ,  
306   others      → non-mandatory-exception(NONEXISTENT) )
```


6.7 Expressions

An expression denotes a computation for obtaining a value. However, an expression does not define a value if the computation does not terminate, or if an exception occurs during the computation such as might arise from using an undefined value or from dividing by zero.

Concrete Syntax

expression = simple expression, [relational operator, simple expression] ;

simple expression = [sign], term, { term operator, term } ;

term = factor, { factor operator, factor } ;

factor = "(", expression, ")" | not operator, factor | value designator | function call | value constructor | constant literal ;

There shall be four classes of operator precedence which, from highest to lowest, shall be: not operator, factor operators, term operators and signs, and relational operators. Within one expression, operators shall be applied in decreasing order of precedence. Two or more operators of the same precedence within one term, simple expression or expression shall be applied from left to right.

NOTE — Brackets can be used to nest expressions, and to override the operator precedence. Two operators may not be juxtaposed.

Abstract Syntax

types

Expression = *Infix-expression*
 | *Prefix-expression*
 | *Value-designator*
 | *Function-call*
 | *Value-constructor*
 | *Constant-literal*

annotations The abstract syntax of an expression denotes the evaluation of the constituent factors, terms, simple expressions and expressions given in the concrete syntax. The binding of operators to operands (denoted by brackets and operator precedence in the concrete syntax) is denoted by the structure of the abstract syntax.

The concrete syntax of an expression is equivalent to a tree, each node of which produces one abstract syntactic expression.

Static Semantics

functions

$wf-expression : Expression \rightarrow Environment \rightarrow \mathbb{B}$

$wf-expression(expr)\rho \triangleq$

154 $(is-Infix-expression(expr) \rightarrow wf-infix-expression(expr)\rho,$
179 $is-Prefix-expression(expr) \rightarrow wf-prefix-expression(expr)\rho,$
183 $is-Value-designator(expr) \rightarrow wf-value-designator(expr)\rho,$
189 $is-Function-call(expr) \rightarrow wf-function-call(expr)\rho,$
191 $is-Value-constructor(expr) \rightarrow wf-value-constructor(expr)\rho,$
207 $is-Constant-literal(expr) \rightarrow wf-constant-literal(expr)\rho);$

$t\text{-expression} : \text{Expression} \rightarrow \text{Environment} \rightarrow \text{Expression-typed}$

$t\text{-expression}(expr)\rho \triangleq$

154 $(is\text{-Infix-expression}(expr) \rightarrow t\text{-infix-expression}(expr)\rho,$
 179 $is\text{-Prefix-expression}(expr) \rightarrow t\text{-prefix-expression}(expr)\rho,$
 183 $is\text{-Value-designator}(expr) \rightarrow t\text{-value-designator}(expr)\rho,$
 189 $is\text{-Function-call}(expr) \rightarrow t\text{-function-call}(expr)\rho,$
 191 $is\text{-Value-constructor}(expr) \rightarrow t\text{-value-constructor}(expr)\rho,$
 207 $is\text{-Constant-literal}(expr) \rightarrow t\text{-constant-literal}(expr)\rho)$

Dynamic Semantics

operations

$m\text{-expression} : \text{Expression} \rightarrow \text{Environment} \xrightarrow{o} \text{Value}$

$m\text{-expression}(expr)\rho \triangleq$

154 $(is\text{-Infix-expression}(expr) \rightarrow m\text{-infix-expression}(expr)\rho,$
 179 $is\text{-Prefix-expression}(expr) \rightarrow m\text{-prefix-expression}(expr)\rho,$
 183 $is\text{-Value-designator}(expr) \rightarrow m\text{-value-designator}(expr)\rho,$
 189 $is\text{-Function-call}(expr) \rightarrow m\text{-function-call}(expr)\rho,$
 192 $is\text{-Value-constructor}(expr) \rightarrow m\text{-value-constructor}(expr)\rho,$
 207 $is\text{-Constant-literal}(expr) \rightarrow m\text{-constant-literal}(expr)\rho)$

6.7.1 Infix Expressions

An infix expression obtains a value by applying an infix operator to two operand values.

Concrete Syntax

factor operator = "*" | "/" | "REM" | "DIV" | "MOD" | and operator ;

and operator = "AND" | "&" ;

term operator = "+" | "-" | "OR" ;

relational operator = "=" | inequality operator | "<" | ">" | "<=" | ">=" | "IN" ;

inequality operator = "<>" | "#" ;

Both forms of and operator and inequality operator shall be provided and shall be equivalent.

Abstract Syntax

types

$\text{Infix-expression} :: \text{left} : \text{Expression}$
 $\phantom{\text{Infix-expression}} \text{op} : \text{Infix-operation}$
 $\phantom{\text{Infix-expression}} \text{right} : \text{Expression}$

annotations The components of an infix expression are a left operand, an infix operator and a right operand.

Static Semantics

Any constant used within the infix expression shall be within a range of values defined by the type of the infix expression. The result of an infix expression shall be of a scalar type or a set type.

functions

```
wf-infix-expression : Infix-expression → Environment →  $\mathbb{B}$ 
wf-infix-expression (mk-Infix-expression(left, op, right)) $\rho \triangleq$ 
152   wf-expression (left) $\rho \wedge$ 
152   wf-expression (right) $\rho \wedge$ 
152   let ltype = t-expression (left) $\rho$  in
152   let rtype = t-expression (right) $\rho$  in
155   wf-infix-operation (op, ltype, rtype) $\rho \wedge$ 
154   wf-constants-in-expression (left, right) $\rho$ ;

t-infix-expression : Infix-expression → Environment → Expression-typed
t-infix-expression (mk-Infix-expression(-, op, -) $\rho$ )  $\triangleq$ 
155   t-infix-operation (op)
```

Dynamic Semantics

operations

```
m-infix-expression : Infix-expression → Environment  $\xrightarrow{o}$  Value
m-infix-expression (mk-Infix-expression(op, left, right)) $\rho \triangleq$ 
  if is-Boolean-operation(op)
153   then def lvalue = m-expression(left)  $\rho$ ;
155         m-infix-operation(op, lvalue, right)  $\rho$ 
153   else def lvalue = m-expression(left)  $\rho$ ;
153         def rvalue = m-expression(right)  $\rho$ ;
155         m-infix-operation(op, lvalue, rvalue)  $\rho$ 
```

annotations In the case of an infix operator other than a Boolean operator, the evaluation order given above is one of those permitted, right to left or parallel order being allowed also, see 4.1.

In the case of a Boolean operator, whether or not the right expression is evaluated depends upon the value of the left expression; the left operand is evaluated first, and, if necessary, the right operand and finally the infix operation.

CHANGE — Arithmetic on variables of the address type is not allowed (except via functions imported from **SYSTEM**).

Auxiliary Definitions

functions

```
wf-constants-in-expression : Expression × Expression → Environment →  $\mathbb{B}$ 
wf-constants-in-expression (left, right) $\rho \triangleq$ 
152   let ltype = t-expression (left) $\rho$  in
152   let rtype = t-expression (right) $\rho$  in
  if ltype =  $\mathbb{Z}$ -TYPE ∨ ltype =  $\mathbb{R}$ -TYPE ∨ ltype =  $\mathbb{C}$ -TYPE
218   then let lvalue = evaluate-constant-expression (left) $\rho$  in
155         in-range (rtype, lvalue)
  elseif rtype =  $\mathbb{Z}$ -TYPE ∨ rtype =  $\mathbb{R}$ -TYPE ∨ rtype =  $\mathbb{C}$ -TYPE
218   then let rvalue = evaluate-constant-expression (right) $\rho$  in
155         in-range (ltype, rvalue)
  else true
```

annotations Constants in an infix expression are checked to be within the appropriate range for the type of that expression.

Auxiliary Definitions

functions

$$\begin{aligned}
 & in-range : Typed \times Value \rightarrow \mathbb{B} \\
 & in-range (type, value) \triangleq \\
 & \quad \text{if } type = \mathbb{C}\text{-TYPE} \\
 ?? \quad & \quad \text{then } re\text{-minimum}(type) \leq re\ value \wedge re\ value \leq re\text{-maximum}(type) \wedge \\
 ?? \quad & \quad \quad im\text{-minimum}(type) \leq im\ value \wedge im\ value \leq im\text{-maximum}(type) \\
 288 \quad & \quad \text{else } minimum(type) \leq value \wedge value \leq maximum(type)
 \end{aligned}$$

Abstract Syntax

types

$$\begin{aligned}
 Infix\text{-operation} = & Complex\text{-number-operation} \\
 & | Real\text{-number-operation} \\
 & | Whole\text{-number-operation} \\
 & | Boolean-operation \\
 & | Set-operation \\
 & | Relational-operation
 \end{aligned}$$

annotations For convenience of the description, infix operations are divided into six classes.

Static Semantics

functions

$$\begin{aligned}
 & wf\text{-infix-operation} : Infix\text{-operation} \times Expression\text{-typed} \times Expression\text{-typed} \rightarrow Environment \rightarrow \mathbb{B} \\
 & wf\text{-infix-operation}(op, ltype, rtype)\rho \triangleq \\
 156 \quad & (is\text{-Complex-number-operation}(op) \rightarrow wf\text{-complex-number-operation}(op, ltype, rtype)\rho, \\
 159 \quad & is\text{-Real-number-operation}(op) \rightarrow wf\text{-real-number-operation}(op, ltype, rtype)\rho, \\
 161 \quad & is\text{-Whole-number-operation}(op) \rightarrow wf\text{-whole-number-operation}(op, ltype, rtype)\rho, \\
 164 \quad & is\text{-Boolean-operation}(op) \rightarrow wf\text{-Boolean-operation}(op, ltype, rtype)\rho, \\
 165 \quad & is\text{-Set-operation}(op) \rightarrow wf\text{-set-operation}(op, ltype, rtype)\rho, \\
 167 \quad & is\text{-Relational-operation}(op) \rightarrow wf\text{-relational-operation}(op, ltype, rtype)\rho); \\
 & t\text{-infix-operation} : Infix\text{-operation} \rightarrow Expression\text{-typed} \\
 & t\text{-infix-operation}(op) \triangleq \\
 156 \quad & (is\text{-Complex-number-operation}(op) \rightarrow t\text{-complex-number-operation}(op), \\
 159 \quad & is\text{-Real-number-operation}(op) \rightarrow t\text{-real-number-operation}(op), \\
 161 \quad & is\text{-Whole-number-operation}(op) \rightarrow t\text{-whole-number-operation}(op), \\
 164 \quad & is\text{-Boolean-operation}(op) \rightarrow t\text{-Boolean-operation}(op), \\
 165 \quad & is\text{-Set-operation}(op) \rightarrow t\text{-set-operation}(op), \\
 167 \quad & is\text{-Relational-operation}(op) \rightarrow t\text{-relational-operation}(op))
 \end{aligned}$$

Dynamic Semantics

operations

$$\begin{aligned}
 & m\text{-infix-operation} : Infix\text{-operation} \times Value \times (Value \mid Expression) \rightarrow Environment \xrightarrow{o} Value \\
 & m\text{-infix-operation}(op, lval, rval)\rho \triangleq \\
 157 \quad & (is\text{-Complex-number-operation}(op) \rightarrow m\text{-complex-number-operation}(op, lval, rval),
 \end{aligned}$$

159 *is-Real-number-operation*(*op*) \rightarrow *m-real-number-operation*(*op*, *lval*, *rval*) ,
161 *is-Whole-number-operation*(*op*) \rightarrow *m-whole-number-operation*(*op*, *lval*, *rval*) ,
164 *is-Boolean-operation*(*op*) \rightarrow *m-Boolean-operation*(*op*, *lval*, *rval*) ρ ,
166 *is-Set-operation*(*op*) \rightarrow *m-set-operation*(*op*, *lval*, *rval*) ,
167 *is-Relational-operation*(*op*) \rightarrow *m-relational-operation*(*op*, *lval*, *rval*))

6.7.1.1 Complex Number Infix Operations

Four basic arithmetic infix operations are provided for operands of a complex number type, corresponding to addition, subtraction, multiplication, and division, denoted lexically by the symbols ‘+’, ‘-’, ‘*’, and ‘/’ respectively. The results of complex number infix operations are approximations to the corresponding mathematical operations.

Abstract Syntax

types

Complex-number-operation :: *op* : *Complex-number-operator*
otype : *Complex-number-type*

Static Semantics

A complex number infix operation with both operands as constant values of CC-type has a result which shall be that of CC-type; a complex number infix operation with one operand as constant value of CC-type has a result which shall be of the same type as the other operand, which shall be of a complex number type. Otherwise both operands shall be of the identical complex number type, and this shall be the type of the result.

NOTE — The following table gives the permitted types and the type of the result.

	complex type	long complex type	CC type
complex type	complex type	error	complex type
long complex type	error	long complex type	long complex type
CC type	complex type	long complex type	CC type

functions

wf-complex-number-operation :
Complex-number-operation \times *Expression-typed* \times *Expression-typed* \rightarrow *Environment* \rightarrow \mathbb{B}
wf-complex-number-operation (*mk-Complex-number-operation*(-, *otype*), *ltype*, *rtype*) $\rho \triangleq$
otype = *complex-result-type*(*ltype*, *rtype*);
t-complex-number-operation : *Complex-number-operation* \rightarrow *Expression-typed*
t-complex-number-operation (*mk-Complex-number-operation*(-, *otype*)) \triangleq
otype

Dynamic Semantics

A complex number infix operation shall be performed with an accuracy which is determined by the type of its operands. This accuracy shall not decrease from the Complex type to the Longcomplex type and from the Longcomplex type to the type of the complex number literal values.

It shall be an exception if a complex number infix operation leads to overflow, or division by zero. A complex number infix operation that leads to underflow shall produce an approximation to the mathematical result.

NOTES

1 Underflow does not cause an exception to occur.

2 ‘Gradual underflow’ is discussed in 8.2: It is different to ‘underflow’ see .

operations

```

m-complex-number-operation : Complex-number-operation × Value × Value  $\xrightarrow{o}$  Value
m-complex-number-operation (mk-Complex-number-operation(op, otype), lvalue, rvalue)  $\triangleq$ 
157   def c = m-complex-number-operator(op) (lvalue, rvalue);
157   get-complex-result(otype, c)

```

Auxiliary Definitions

functions

```

complex-result-type : Complex-number-type × Complex-number-type → Complex-number-type
complex-result-type (ltype, rtype)  $\triangleq$ 
  (ltype = rtype → ltype,
   ltype = C-TYPE → rtype,
   rtype = C-TYPE → ltype)

```

annotations The operation is undefined if its operands are not of the same type (unless one is a literal value).

operations

```

get-complex-result : Complex-number-type × C  $\xrightarrow{o}$  C
get-complex-result (type, r)  $\triangleq$ 
155   if in-range (type, r)
      then cases type:
??       COMPLEX-TYPE → return complex-approximation (r),
??       LONG-COMPLEX-TYPE → return long-complex-approximation (r),
??       C-TYPE → return complex-constant-approximation (r)
      end
306   else mandatory-exception(COMPLEX-OVERFLOW)

```

annotations The functions *complex-approximation*, *long-complex-approximation*, and *complex-constant-approximation* are implementation-defined and give approximations to the corresponding mathematical result. The functions *complex-approximation*, *long-complex-approximation*, and *complex-constant-approximation* are in the order of non-decreasing accuracy.

Abstract Syntax

types

Complex-number-operator = ADD | SUBTRACT | MULTIPLY | DIVIDE

annotations The above operators correspond to ‘+’, ‘-’, ‘*’, and ‘/’ respectively.

Dynamic Semantics

operations

```

m-complex-number-operator : Complex-number-operator → (Value × Value)  $\xrightarrow{o}$  Value
m-complex-number-operator (op)(lvalue, rvalue)  $\triangleq$ 
  cases op:
158   ADD → m-complex-add(lvalue, rvalue) ,
158   SUBTRACT → m-complex-subtract(lvalue, rvalue) ,
158   MULTIPLY → m-complex-multiply(lvalue, rvalue) ,

```

```
158      DIVIDE      → m-complex-divide(lvalue, rvalue)
      end;

m-complex-add : Value × Value  $\xrightarrow{o}$  Value
m-complex-add (lvalue, rvalue)  $\triangleq$ 
      return lvalue + rvalue ;

m-complex-subtract : Value × Value  $\xrightarrow{o}$  Value
m-complex-subtract (lvalue, rvalue)  $\triangleq$ 
      return lvalue − rvalue ;

m-complex-multiply : Value × Value  $\xrightarrow{o}$  Value
m-complex-multiply (lvalue, rvalue)  $\triangleq$ 
      return lvalue × rvalue ;

m-complex-divide : Value × Value  $\xrightarrow{o}$  Value
m-complex-divide (lvalue, rvalue)  $\triangleq$ 
      if rvalue ≠ 0
      then return lvalue/rvalue
306   else mandatory-exception(COMPLEX-ZERO-DIVISION)
```

annotations The operations +, −, ×, and / which are used in the above definitions are the mathematical ones, which are of infinite precision. Hence the finite precision of actual implementations is mirrored by infinite precision followed by an approximation operation (performed by *get-complex-result*).

6.7.1.2 Real Number Infix Operations

Four basic arithmetic infix operations are provided for operands of a real number type, corresponding to addition, subtraction, multiplication, and division, denoted lexically by the symbols ‘+’, ‘−’, ‘*’, and ‘/’ respectively. The results of real number infix operations are approximations to the corresponding mathematical operations.

Abstract Syntax

```
types

Real-number-operation :: op : Real-number-operator
                        otype : Real-number-type
```

Static Semantics

A real number infix operation with both operands as constant values of RR-type has a result which shall be that of RR-type; a real number infix operation with one operand as constant value of RR-type has a result which shall be of the same type as the other operand, which shall be of a real number type. Otherwise both operands shall be of the identical real number type, and this shall be the type of the result.

Language Clarification

These rules are true for both constants and literal values.

NOTE — The following table gives the permitted types and the type of the result.

	real type	long real type	RR type
real type	real type	error	real type
long real type	error	long real type	long real type
RR type	real type	long real type	RR type

functions

$wf\text{-}real\text{-}number\text{-}operation : Real\text{-}number\text{-}operation \times Expression\text{-}typed \times Expression\text{-}typed \rightarrow Environment \rightarrow \mathbb{B}$

$wf\text{-}real\text{-}number\text{-}operation (mk\text{-}Real\text{-}number\text{-}operation(-, otype), ltype, rtype)\rho \triangleq$

159 $otype = real\text{-}result\text{-}type (ltype, rtype);$

$t\text{-}real\text{-}number\text{-}operation : Real\text{-}number\text{-}operation \rightarrow Expression\text{-}typed$

$t\text{-}real\text{-}number\text{-}operation (mk\text{-}Real\text{-}number\text{-}operation(-, otype)) \triangleq$
 $otype$

Dynamic Semantics

A real number infix operation shall be performed with an accuracy which is determined by the type of its operands. This accuracy shall not decrease from the Real type to the Longreal type and from the Longreal type to the type of the real number literal values.

It shall be an exception if a real number infix operation leads to overflow, or division by zero. A real number infix operation that leads to underflow shall produce an approximation to the mathematical result.

NOTE — Underflow does not cause an exception to occur.

operations

$m\text{-}real\text{-}number\text{-}operation : Real\text{-}number\text{-}operation \times Value \times Value \xrightarrow{o} Value$

$m\text{-}real\text{-}number\text{-}operation (mk\text{-}Real\text{-}number\text{-}operation(op, otype), lvalue, rvalue) \triangleq$

160 $\text{def } r = m\text{-}real\text{-}number\text{-}operator(op) (lvalue, rvalue);$

159 $\text{get}\text{-}real\text{-}result(otype, r)$

Auxiliary Definitions

functions

$real\text{-}result\text{-}type : Real\text{-}number\text{-}type \times Real\text{-}number\text{-}type \rightarrow Real\text{-}number\text{-}type$

$real\text{-}result\text{-}type (ltype, rtype) \triangleq$

$(ltype = rtype \rightarrow ltype,$

$ltype = \mathbb{R}\text{-}TYPE \rightarrow rtype,$

$rtype = \mathbb{R}\text{-}TYPE \rightarrow ltype)$

annotations The operation is undefined if its operands are not of the same type (unless one is a literal value).

operations

$\text{get}\text{-}real\text{-}result : Real\text{-}number\text{-}type \times \mathbb{R} \xrightarrow{o} \mathbb{R}$

$\text{get}\text{-}real\text{-}result (type, r) \triangleq$

155 $\text{if } in\text{-}range (type, r)$

$\text{then cases } type:$

?? $\mathbb{R}\text{-}TYPE \rightarrow \text{return } real\text{-}approximation (r),$

?? $\mathbb{L}\text{-}REAL\text{-}TYPE \rightarrow \text{return } long\text{-}real\text{-}approximation (r),$

?? $\mathbb{R}\text{-}TYPE \rightarrow \text{return } real\text{-}constant\text{-}approximation (r)$

end

306 $\text{else } mandatory\text{-}exception(\mathbb{R}\text{-}OVERFLOW)$

annotations The functions *real-approximation*, *long-real-approximation*, and *real-constant-approximation* are implementation-defined and give approximations to the corresponding mathematical result. The functions *real-approximation*, *long-real-approximation*, and *real-constant-approximation* are in the order of non-decreasing accuracy.

Abstract Syntax

types

$Real\text{-}number\text{-}operator = \text{ADD} \mid \text{SUBTRACT} \mid \text{MULTIPLY} \mid \text{DIVIDE}$

annotations The above operators correspond to '+', '-', '*', and '/' respectively.

Dynamic Semantics

operations

```

m-real-number-operator : Real-number-operator  $\rightarrow$  (Value  $\times$  Value)  $\xrightarrow{o}$  Value
m-real-number-operator (op) (lvalue, rvalue)  $\triangleq$ 
  cases op:
160   ADD       $\rightarrow$  m-real-add(lvalue, rvalue) ,
160   SUBTRACT  $\rightarrow$  m-real-subtract(lvalue, rvalue) ,
160   MULTIPLY  $\rightarrow$  m-real-multiply(lvalue, rvalue) ,
160   DIVIDE    $\rightarrow$  m-real-divide(lvalue, rvalue)
  end;

m-real-add : Value  $\times$  Value  $\xrightarrow{o}$  Value
m-real-add (lvalue, rvalue)  $\triangleq$ 
  return lvalue + rvalue ;

m-real-subtract : Value  $\times$  Value  $\xrightarrow{o}$  Value
m-real-subtract (lvalue, rvalue)  $\triangleq$ 
  return lvalue - rvalue ;

m-real-multiply : Value  $\times$  Value  $\xrightarrow{o}$  Value
m-real-multiply (lvalue, rvalue)  $\triangleq$ 
  return lvalue  $\times$  rvalue ;

m-real-divide : Value  $\times$  Value  $\xrightarrow{o}$  Value
m-real-divide (lvalue, rvalue)  $\triangleq$ 
  if rvalue  $\neq$  0
  then return lvalue / rvalue
306  else mandatory-exception(REAL-ZERO-DIVISION)

```

annotations The operations +, −, ×, and / which are used in the above definitions are the mathematical ones, which are of infinite precision. Hence the finite precision of actual implementations is mirrored by infinite precision followed by an approximation operation (performed by *get-real-result*).

6.7.1.3 Whole Number Infix Operations

Seven basic arithmetic infix operations are provided for operands of a whole number type, corresponding to addition, subtraction, multiplication, two forms of division, and two forms of remaindering. These operations are denoted lexically by the symbols '+', '-', '*', '/', 'DIV', 'REM', and 'MOD' respectively. The results of whole number infix operations are exact unless the result would be outside the range of the whole number type.

Abstract Syntax

types

$Whole\text{-}number\text{-}operation :: op : Whole\text{-}number\text{-}operator$
 $otype : Whole\text{-}number\text{-}type$

Static Semantics

A whole number infix operation with both operands as constant values of ZZ type has a result which shall be of ZZ type; a whole number infix operation with one operand that is a constant value of ZZ type has a result which shall be of the same type as the other operand, which shall be of a whole number type. Otherwise, both operands shall be of the identical whole number type, and this type shall be the type of the result.

Language Clarification

These rules are true for both constants and literal values.

NOTE — The following table gives the permitted types and the type of the result.

	signed type	unsigned type	ZZ type
signed type	signed type	error	signed type
unsigned type	error	unsigned type	unsigned type
ZZ type	signed type	unsigned type	ZZ type

functions

wf-whole-number-operation :

$Whole\text{-}number\text{-}operation \rightarrow (Expression\text{-}typed \times Expression\text{-}typed) \rightarrow Environment \rightarrow \mathbb{B}$

wf-whole-number-operation (*mk-Whole-number-operation*(-, *otype*))(*ltype*, *rtype*) $\rho \triangleq$

161 *otype* = *whole-number-result-type* (*ltype*, *rtype*);

t-whole-number-operation : *Whole-number-operation* $\rightarrow Expression\text{-}typed$

t-whole-number-operation (*mk-Whole-number-operation*(-, *otype*)) \triangleq
otype

Dynamic Semantics

A whole number infix operation with whole number operands shall produce the mathematically correct result if the result is within the range of the result type. It shall be an exception if the result is not within range of the result type.

operations

m-whole-number-operation : *Whole-number-operation* $\rightarrow (Value \times Value) \xrightarrow{o} Value$

m-whole-number-operation (*mk-Whole-number-operation*(*op*, *otype*), *lvalue*, *rvalue*) \triangleq

?? *def* *n* = *m-whole-number-operator*(*op*) (*lvalue*, *rvalue*);

162 *get-whole-result*(*otype*, *n*)

Auxiliary Definitions

functions

whole-number-result-type : *Whole-number-type* $\times Whole\text{-}number\text{-}type \rightarrow Whole\text{-}number\text{-}type$

whole-number-result-type (*ltype*, *rtype*) \triangleq

(*ltype* = *rtype* \rightarrow *ltype*,

ltype = $\mathbb{Z}\text{-TYPE}$ \rightarrow *rtype*,

rtype = $\mathbb{Z}\text{-TYPE}$ \rightarrow *ltype*)

annotations

The operation is undefined if the left and right operands for a whole number operation are not of the same type.

operations

$get_whole_result(type, n) : Whole_number_type \times \mathbb{Z} \xrightarrow{o} \mathbb{Z}$

```

 $\triangle$ 
155   if in-range (type, n)
      then return n
306   else mandatory-exception(WHOLE-OVERFLOW)
```

Abstract Syntax

types

```
Whole-number-operator = ADD
                        | SUBTRACT
                        | MULTIPLY
                        | DIVIDE
                        | REM
                        | DIV
                        | MOD
```

annotations The above operations correspond to ‘+’, ‘-’, ‘*’, ‘/’, ‘REM’, ‘DIV’, and ‘MOD’ respectively.

CHANGE — The operators ‘/’ and ‘REM’ have been introduced on the recommendation of Professor Wirth.

Dynamic Semantics

functions

```

m-whole-number-operator : Whole-number-operator  $\rightarrow$  (Value  $\times$  Value)  $\rightarrow$   $\mathbb{B}$ 
m-whole-number-operator (op)(lvalue, rvalue)  $\triangle$ 
  cases op :
163   ADD       $\rightarrow$  m-whole-add(lvalue, rvalue) ,
163   SUBTRACT  $\rightarrow$  m-whole-subtract(lvalue, rvalue) ,
163   MULTIPLY  $\rightarrow$  m-whole-multiply(lvalue, rvalue) ,
163   DIVIDE    $\rightarrow$  m-whole-divide(lvalue, rvalue) ,
163   REM       $\rightarrow$  m-whole-rem(lvalue, rvalue) ,
163   DIV       $\rightarrow$  m-whole-div(lvalue, rvalue) ,
164   MOD       $\rightarrow$  m-whole-mod(lvalue, rvalue)
end
```

The evaluation of ‘**lval** + **rval**’, ‘**lval** - **rval**’, and ‘**lval** * **rval**’, shall produce the mathematically correct result if the result is within the range of the result type, otherwise an exception shall occur.

The evaluation of ‘**lval** / **rval**’ shall cause an exception to occur if **rval** is zero; otherwise the result which is denoted by quotient shall be such that the identity

$$\mathbf{lval} = \mathbf{rval} * \mathbf{quotient} + \mathbf{remainder}$$

is satisfied for a value of **remainder** that is either zero or an integer of the same sign as **lval** and of smaller absolute value than **rval**.

NOTE — If **x** and **y** are whole numbers and **y** is not equal to zero, then $(-x)/y = x/(-y) = -(x/y)$

The evaluation of ‘**lval** REM **rval**’ shall cause an exception to occur if **rval** is zero; otherwise the result which is denoted by **remainder** shall be such that the identity

$$\mathbf{lval} = \mathbf{rval} * \mathbf{quotient} + \mathbf{remainder}$$

is satisfied for a value of **remainder** that is either zero or an integer of the same sign as **lval** and of smaller absolute value than **rval**.

The evaluation of '**lval** **DIV** **rval**' shall cause an exception to occur if **rval** is zero or negative; otherwise the result which is denoted by **quotient** shall be such that the identity

$$\text{lval} = \text{rval} * \text{quotient} + \text{modulus}$$

is satisfied for a value of **modulus** that is a non-negative integer less than **rval**.

The evaluation of '**lval** **MOD** **rval**' shall cause an exception to occur if **rval** is zero or negative; otherwise the result which is denoted by **modulus** shall be such that the identity

$$\text{lval} = \text{rval} * \text{quotient} + \text{modulus}$$

is satisfied for a value of **modulus** that is a non-negative integer less than **rval**.

NOTE — The result of these operations can be summarized by a table:

<i>op</i>	31 <i>op</i> 10	31 <i>op</i> (-10)	(-31) <i>op</i> 10	(-31) <i>op</i> (-10)
/	3	-3	-3	3
REM	1	1	-1	-1
DIV	3	Exception	-4	Exception
MOD	1	Exception	9	Exception

operations

$$m\text{-whole-add} : \text{Value} \times \text{Value} \xrightarrow{o} \text{Value}$$

$$m\text{-whole-add}(\text{lvalue}, \text{rvalue}) \triangleq$$

$$\text{return } \text{lvalue} + \text{rvalue} ;$$

$$m\text{-whole-subtract} : \text{Value} \times \text{Value} \xrightarrow{o} \text{Value}$$

$$m\text{-whole-subtract}(\text{lvalue}, \text{rvalue}) \triangleq$$

$$\text{return } \text{lvalue} - \text{rvalue} ;$$

$$m\text{-whole-multiply} : \text{Value} \times \text{Value} \xrightarrow{o} \text{Value}$$

$$m\text{-whole-multiply}(\text{lvalue}, \text{rvalue}) \triangleq$$

$$\text{return } \text{lvalue} \times \text{rvalue} ;$$

$$m\text{-whole-divide} : \text{Value} \times \text{Value} \xrightarrow{o} \text{Value}$$

$$m\text{-whole-divide}(\text{lvalue}, \text{rvalue}) \triangleq$$

if $\text{rvalue} \neq 0$

then let $\text{quotient} : \mathbb{Z}$ be st $\exists \text{remainder} : \mathbb{Z} \cdot \text{abs } \text{remainder} < \text{abs } \text{rvalue} \wedge$

$\text{sign}(\text{remainder}) = \text{sign}(\text{lvalue}) \wedge$

$\text{lvalue} = \text{quotient} \times \text{rvalue} + \text{remainder}$ in

return quotient

else $\text{mandatory-exception}(\text{WHOLE-ZERO-DIVISION}) ;$

$$m\text{-whole-rem} : \text{Value} \times \text{Value} \xrightarrow{o} \text{Value}$$

$$m\text{-whole-rem}(\text{lvalue}, \text{rvalue}) \triangleq$$

if $\text{rvalue} \neq 0$

then let $\text{remainder} : \mathbb{Z}$ be st $\exists \text{quotient} : \mathbb{Z} \cdot \text{abs } \text{remainder} < \text{abs } \text{rvalue} \wedge$

$\text{sign}(\text{remainder}) = \text{sign}(\text{lvalue}) \wedge$

$\text{lvalue} = \text{quotient} \times \text{rvalue} + \text{remainder}$ in

return remainder

else $\text{mandatory-exception}(\text{WHOLE-ZERO-REMAINDER}) ;$

```

m-whole-div : Value × Value  $\xrightarrow{o}$  Value
m-whole-div (lvalue, rvalue)  $\triangleq$ 
  if rvalue > 0
  then let quotient :  $\mathbb{Z}$  be st  $\exists$  modulus :  $\mathbb{N} \cdot$  lvalue = quotient × rvalue + modulus  $\wedge$ 
    modulus < rvalue in
    return quotient
306   else mandatory-exception(WHOLE-NONPOS-DIV) ;

m-whole-mod : Value × Value  $\xrightarrow{o}$  Value
m-whole-mod (lvalue, rvalue)  $\triangleq$ 
  if rvalue > 0
  then let modulus :  $\mathbb{N}$  be st  $\exists$  quotient :  $\mathbb{Z} \cdot$  lvalue = quotient × rvalue + modulus  $\wedge$ 
    modulus < rvalue in
    return modulus
306   else mandatory-exception(WHOLE-NONPOS-MOD)

```

6.7.1.4 Boolean Infix Operations

Two infix operations are provided for operands of the Boolean type, corresponding to logical conjunctions and logical disjunctions. These operations are denoted lexically by the symbols ‘**AND**’ and ‘**OR**’ respectively.

Abstract Syntax

types

Boolean-operation :: *op* : *Boolean-operator*

Static Semantics

Both operands shall be of the Boolean type and the result shall also be of the Boolean type.

functions

wf-Boolean-operation : *Boolean-operation* \rightarrow (*Expression-typed* × *Expression-typed*) \rightarrow *Environment* $\rightarrow \mathbb{B}$
wf-Boolean-operation (*mk-Boolean-operation*(-))(*ltype*, *rtype*) ρ \triangleq
ltype = BOOLEAN-TYPE \wedge *rtype* = BOOLEAN-TYPE;
t-Boolean-operation : *Boolean-operation* \rightarrow *Expression-typed*
t-Boolean-operation (*mk-Boolean-operation*(-)) \triangleq
 BOOLEAN-TYPE

Dynamic Semantics

operations

m-Boolean-operation : *Boolean-operation* \rightarrow (*Value* × *Expression*) \rightarrow *Environment* $\xrightarrow{o} \mathbb{B}$
m-Boolean-operation (*mk-Boolean-operation*(*op*))(*lvalue*, *right*) ρ \triangleq
165 *m-Boolean-operator*(*op*) (*lvalue*, *right*) ρ

Abstract Syntax

types

Boolean-operator = AND | OR

Dynamic Semantics

operations

$$\begin{aligned} & m\text{-Boolean-operator} : \text{Boolean-operator} \rightarrow (\text{Value} \times \text{Expression}) \rightarrow \text{Environment} \xrightarrow{o} \mathbb{B} \\ & m\text{-Boolean-operator}(op)(lvalue, right)\rho \triangleq \\ & \quad \text{cases } op: \\ 165 \quad & \text{AND} \rightarrow m\text{-Boolean-and}(lvalue, right)\rho, \\ 165 \quad & \text{OR} \rightarrow m\text{-Boolean-or}(lvalue, right)\rho \\ & \text{end} \end{aligned}$$

If the left hand operand to the **AND** operator is **FALSE**, the result shall be **FALSE** and the right hand operand shall not be evaluated. If the left operand to the **AND** operator is **TRUE**, the result shall be the value of the right hand operand.

If the left hand operand to the **OR** operator is **TRUE** the result shall be **TRUE** and the right hand operand shall not be evaluated. If the left hand operand to the **OR** operator is **FALSE**, the result shall be the value of the right hand operand.

operations

$$\begin{aligned} & m\text{-Boolean-and} : \text{Value} \times \text{Expression} \rightarrow \text{Environment} \xrightarrow{o} \mathbb{B} \\ & m\text{-Boolean-and}(lvalue, right)\rho \triangleq \\ & \quad \text{if } lvalue \\ 153 \quad & \text{then } m\text{-expression}(right)\rho \\ & \quad \text{else return false ;} \\ & m\text{-Boolean-or} : \text{Value} \times \text{Expression} \rightarrow \text{Environment} \xrightarrow{o} \mathbb{B} \\ & m\text{-Boolean-or}(lvalue, right)\rho \triangleq \\ & \quad \text{if } lvalue \\ & \quad \text{then return true} \\ 153 \quad & \text{else } m\text{-expression}(right)\rho \end{aligned}$$

6.7.1.5 Set Infix Operations

Four set infix operations are provided for operands of identical set types, corresponding to union, difference, intersection and symmetric difference. These operations are denoted lexically by the symbols ‘+’, ‘-’, ‘*’, and ‘/’ respectively.

Abstract Syntax

types

$$\begin{aligned} & \text{Set-operation} :: op : \text{Set-operator} \\ & \quad otype : \text{Typed} \end{aligned}$$

Static Semantics

The type of the result and the type of each operand of a set infix operator shall all be the same set type.

functions

$$\begin{aligned} & wf\text{-set-operation} : \text{Set-operation} \rightarrow (\text{Expression-typed} \times \text{Expression-typed}) \rightarrow \text{Environment} \rightarrow \mathbb{B} \\ & wf\text{-set-operation}(mk\text{-Set-operation}(-, otype))(ltype, rtype)\rho \triangleq \\ 279 \quad & is\text{-set-type}(ltype)\rho \wedge \\ & \quad ltype = otype \wedge rtype = otype; \\ & t\text{-set-operation} : \text{Set-operation} \rightarrow \text{Expression-typed} \\ & t\text{-set-operation}(mk\text{-Set-operation}(-, otype)) \triangleq \\ & \quad otype \end{aligned}$$

Dynamic Semantics

operations

```
m-set-operation : Set-operation → (Value × Value)  $\xrightarrow{o}$  Value  
m-set-operation (mk-Set-operation(op, -))(lvalue, rvalue)  $\triangleq$   
  let mk-Set-value(lvalues) = lvalue in  
  let mk-Set-value(rvalues) = rvalue in  
166 let res = m-set-operator(op) (lvalues, rvalues) in  
  return mk-Set-value(res)
```

annotations The actual set of values is contained in a composite object, the sets must be extracted in order to apply the operator, and then the result converted back to the appropriate composite type.

Abstract Syntax

types

Set-operator = UNION | DIFFERENCE | INTERSECTION | SYMMETRIC-DIFFERENCE

annotations The above operations correspond to ‘+’, ‘-’, ‘*’ and ‘/’ respectively.

Dynamic Semantics

operations

```
m-set-operator : Set-operator → (Value × Value)  $\xrightarrow{o}$  Value  
m-set-operator (op)(lvalues, rvalues)  $\triangleq$   
  cases op:  
166    UNION                      → m-set-union(lvalues, rvalues) ,  
166    DIFFERENCE                → m-set-difference(lvalues, rvalues) ,  
166    INTERSECTION              → m-set-intersection(lvalues, rvalues) ,  
167    SYMMETRIC-DIFFERENCE → m-set-symmetric-difference(lvalues, rvalues)  
  end
```

The set union operation shall produce the set containing elements from either the left operand or the right operand or both.

The set difference operation shall produce the set containing elements from the left operand but not from the right operand.

The set intersection operation shall produce the set containing elements which are both from the left and from the right operand.

The set symmetric difference operation shall produce the set containing elements which are from exactly one of the operands.

operations

```
m-set-union : Value × Value  $\xrightarrow{o}$  Value  
m-set-union (lvalues, rvalues)  $\triangleq$   
  return lvalues ∪ rvalues ;  
  
m-set-difference : Value × Value  $\xrightarrow{o}$  Value  
m-set-difference (lvalues, rvalues)  $\triangleq$   
  return lvalues − rvalues ;
```

$m\text{-set-intersection} : \text{Value} \times \text{Value} \xrightarrow{o} \text{Value}$
 $m\text{-set-intersection}(lvalues, rvalues) \triangleq$
 $\quad \text{return } lvalues \cap rvalues ;$
 $m\text{-set-symmetric-difference} : \text{Value} \times \text{Value} \xrightarrow{o} \text{Value}$
 $m\text{-set-symmetric-difference}(lvalues, rvalues) \triangleq$
 $\quad \text{return } (lvalues - rvalues) \cup (rvalues - lvalues)$

6.7.1.6 Relational Operations

Relational infix operations are provided for elementary types, corresponding to tests for equality and inequality (denoted lexically by the symbols ‘=’ and ‘<>’), and for ordering (in the case of scalar types, denoted lexically by the symbols ‘<’, ‘>’, ‘<=’, and ‘>=’) or set relationships (in the case of set types, denoted lexically by the symbols ‘<=’ and ‘>=’). A further relational operation is also provided, a test for set membership, denoted lexically by the symbol ‘**IN**’.

Abstract Syntax

types

$Relational\text{-}operation = Complex\text{-}comparison\text{-}operation$
 $\quad | Scalar\text{-}relational\text{-}operation$
 $\quad | Set\text{-}relational\text{-}operation$
 $\quad | Procedure\text{-}relational\text{-}operation$
 $\quad | Pointer\text{-}relational\text{-}operation$
 $\quad | Protection\text{-}relational\text{-}operation$

Static Semantics

functions

$wf\text{-}relational\text{-}operation : \text{Infix-operation} \times \text{Expression-typed} \times \text{Expression-typed} \rightarrow \text{Environment} \rightarrow \mathbb{B}$

$wf\text{-}relational\text{-}operation(op, ltype, rtype)\rho \triangleq$

$168 \quad (is\text{-}Complex\text{-}comparison\text{-}operation(op) \rightarrow wf\text{-}complex\text{-}comparison\text{-}operation(op, ltype, rtype)\rho,$
 $169 \quad is\text{-}Scalar\text{-}relational\text{-}operation(op) \rightarrow wf\text{-}scalar\text{-}relational\text{-}operation(op, ltype, rtype)\rho,$
 $172 \quad is\text{-}Set\text{-}relational\text{-}operation(op) \rightarrow wf\text{-}set\text{-}relational\text{-}operation(op, ltype, rtype)\rho,$
 $174 \quad is\text{-}Procedure\text{-}relational\text{-}operation(op) \rightarrow wf\text{-}procedure\text{-}relational\text{-}operation(op, ltype, rtype)\rho,$
 $175 \quad is\text{-}Pointer\text{-}relational\text{-}operation(op) \rightarrow wf\text{-}pointer\text{-}relational\text{-}operation(op, ltype, rtype)\rho,$
 $177 \quad is\text{-}Protection\text{-}relational\text{-}operation(op) \rightarrow wf\text{-}protection\text{-}relational\text{-}operation(op, ltype, rtype)\rho);$

$t\text{-}relational\text{-}operation : \text{Infix-operation} \rightarrow \text{Environment} \rightarrow \text{Expression-typed}$

$t\text{-}relational\text{-}operation(expr)\rho \triangleq$

$168 \quad (is\text{-}Complex\text{-}comparison\text{-}operation(op) \rightarrow t\text{-}complex\text{-}comparison\text{-}operation(op)\rho,$
 $169 \quad is\text{-}Scalar\text{-}relational\text{-}operation(op) \rightarrow t\text{-}scalar\text{-}relational\text{-}operation(op)\rho,$
 $172 \quad is\text{-}Set\text{-}relational\text{-}operation(op) \rightarrow t\text{-}set\text{-}relational\text{-}operation(op)\rho,$
 $174 \quad is\text{-}Procedure\text{-}relational\text{-}operation(op) \rightarrow t\text{-}procedure\text{-}relational\text{-}operation(op)\rho,$
 $175 \quad is\text{-}Pointer\text{-}relational\text{-}operation(op) \rightarrow t\text{-}pointer\text{-}relational\text{-}operation(op)\rho,$
 $177 \quad is\text{-}Protection\text{-}relational\text{-}operation(op) \rightarrow t\text{-}protection\text{-}relational\text{-}operation(op)\rho)$

Dynamic Semantics

operations

$m\text{-}relational\text{-}operation : \text{Infix-operation} \times \text{Value} \times (\text{Value} \mid \text{Expression}) \rightarrow \text{Environment} \xrightarrow{o} \text{Value}$

$m\text{-}relational\text{-}operation(op, lval, rval)\rho \triangleq$

$168 \quad (is\text{-}Complex\text{-}comparison\text{-}operation(op) \rightarrow m\text{-}complex\text{-}comparison\text{-}operation(op, ltype, rtype),$

$$\begin{aligned}
& \text{is-Scalar-relational-operation}(op) \rightarrow m\text{-scalar-relational-operation}(op, ltype, rtype), \\
& \text{is-Set-relational-operation}(op) \rightarrow m\text{-set-relational-operation}(op, ltype, rtype), \\
& \text{is-Procedure-relational-operation}(op) \rightarrow m\text{-procedure-relational-operation}(op, ltype, rtype), \\
& \text{is-Pointer-relational-operation}(op) \rightarrow m\text{-pointer-relational-operation}(op, ltype, rtype), \\
& \text{is-Protection-relational-operation}(op) \rightarrow m\text{-protection-relational-operation}(op, ltype, rtype)
\end{aligned}$$

Complex Comparison Operations

Two comparison operators are provided for complex types.

Abstract Syntax

types

$Complex\text{-comparison-operation} :: op : Complex\text{-comparison-operator}$

Static Semantics

Only the equality and inequality operators shall be applicable to operands of a complex type; other relational operators shall not be applied to operands of a complex type. The operands shall be of identical types, and the result shall be of the Boolean type.

TO DO — Identical types or comparable types?

functions

$$\begin{aligned}
& wf\text{-complex-comparison-operation} : \\
& \quad Complex\text{-comparison-operation} \rightarrow (Expression\text{-typed} \times Expression\text{-typed}) \rightarrow Environment \rightarrow \mathbb{B} \\
& wf\text{-complex-comparison-operation}(mk\text{-Complex-comparison-operation}(op))(ltype, rtype)\rho \triangleq \\
280 \quad & is\text{-scalar-type}(ltype)\rho \wedge \\
& \quad ltype = rtype \vee \\
?? \quad & ltype = \mathbb{C}\text{-type} \wedge is\text{-complex-type}(rtype)\rho \vee \\
?? \quad & rtype = \mathbb{C}\text{-type} \wedge is\text{-complex-type}(ltype)\rho; \\
& t\text{-complex-comparison-operation} : Complex\text{-comparison-operation} \rightarrow Expression\text{-typed} \\
& t\text{-complex-comparison-operation}(mk\text{-Complex-comparison-operation}(-)) \triangleq \\
& \quad \text{BOOLEAN-TYPE}
\end{aligned}$$

Dynamic Semantics

operations

$$\begin{aligned}
& m\text{-complex-comparison-operation} : Complex\text{-comparison-operation} \rightarrow (Value \times Value) \xrightarrow{o} Value \\
& m\text{-complex-comparison-operation}(mk\text{-Complex-comparison-operation}(op))(lvalue, rvalue) \triangleq \\
169 \quad & m\text{-complex-comparison-operator}(op)(lvalue, rvalue)
\end{aligned}$$

Abstract Syntax

types

$Complex\text{-comparison-operator} = EQ \mid NE$

annotations The above operations correspond to ‘=’ and ‘<>’ respectively.

NOTE — The operation ‘<>’ can be represented as ‘#’, see section 6.7.1

Dynamic Semantics

operations

```
m-complex-comparison-operator : Complex-comparison-operator  $\rightarrow$  (Value  $\times$  Value)  $\xrightarrow{o}$  Value
m-complex-comparison-operator (op)(lvalue, rvalue)  $\triangleq$ 
  cases op:
169      EQ  $\rightarrow$  m-complex-eq(lvalue, rvalue) ,
169      NE  $\rightarrow$  m-complex-ne(lvalue, rvalue)
  end;

m-complex-eq : Value  $\times$  Value  $\xrightarrow{o}$  Value
m-complex-eq (lvalue, rvalue)  $\triangleq$ 
  return lvalue = rvalue ;

m-complex-ne : Value  $\times$  Value  $\xrightarrow{o}$  Value
m-complex-ne (lvalue, rvalue)  $\triangleq$ 
  return lvalue  $\neq$  rvalue
```

NOTE — A consequence of the approximation inherent in complex types is that, for the result from these operations will depend upon the nature of the approximation.

Scalar Relational Operations

Six scalar relational operators are provided for scalar types.

Abstract Syntax

types

Scalar-relational-operation :: *op* : *Scalar-relational-operator*

Static Semantics

The equality and inequality operators shall be applied only to operands of elementary types. The operands shall be of types that are expression compatible, and the result of an equality or inequality operation shall be of the Boolean type.

The other relational operations shall be the less than, greater than, less than or equal, and greater than or equal operators. The operands of a scalar relational operator shall be of scalar types that are expression compatible, and the result of a scalar relational operation shall be of the Boolean type.

NOTE — Since the Boolean type is an ordinal type with FALSE less than TRUE then if *p* and *q* are operands of Boolean type, '*p* = *q*' denotes their equivalence and '*p* <= *q*' means *p* implies *q*.

functions

```
wf-scalar-relational-operation :
  Scalar-relational-operation  $\rightarrow$  (Expression-typed  $\times$  Expression-typed)  $\rightarrow$  Environment  $\rightarrow$   $\mathbb{B}$ 
wf-scalar-relational-operation (mk-Scalar-relational-operation(op))(ltype, rtype) $\rho$   $\triangleq$ 
280   is-scalar-type (ltype) $\rho$   $\wedge$ 
170   is-comparable (ltype, rtype) $\rho$ ;

t-scalar-relational-operation : Scalar-relational-operation  $\rightarrow$  Expression-typed
t-scalar-relational-operation (mk-Scalar-relational-operation(-))  $\triangleq$ 
  BOOLEAN-TYPE
```

Dynamic Semantics

operations

$m\text{-scalar-relational-operation} : \text{Scalar-relational-operation} \rightarrow (\text{Value} \times \text{Value}) \xrightarrow{o} \text{Value}$
 $m\text{-scalar-relational-operation}(mk\text{-Scalar-relational-operation}(op))(lvalue, rvalue) \triangleq$
 $m\text{-scalar-relational-operator}(op)(lvalue, rvalue)$

Abstract Syntax

types

$\text{Scalar-relational-operator} = \text{EQ} \mid \text{NE} \mid \text{LT} \mid \text{GT} \mid \text{LE} \mid \text{GE}$

annotations The above operations correspond to ‘=’, ‘<>’, ‘<’, ‘>’, ‘<=’ and ‘>=’ respectively.

NOTE — The operation ‘<>’ can be represented as ‘#’, see section 6.7.1

Auxiliary Functions

functions

$is\text{-comparable} : \text{Number-type} \times \text{Number-type} \rightarrow \text{Environment} \rightarrow \mathbb{B}$
 $is\text{-comparable}(ltype, rtype)\rho \triangleq$
 $\quad ltype = rtype$
 $\quad \vee$
 $\quad ltype = \mathbb{R}\text{-TYPE} \wedge is\text{-real-number-type}(rtype)\rho$
 $\quad \vee$
 $\quad rtype = \mathbb{R}\text{-TYPE} \wedge is\text{-real-number-type}(ltype)\rho$
 $\quad \vee$
 $\quad ltype = \mathbb{Z}\text{-TYPE} \wedge is\text{-whole-number-type}(rtype)\rho$
 $\quad \vee$
 $\quad rtype = \mathbb{Z}\text{-TYPE} \wedge is\text{-whole-number-type}(ltype)\rho$

Dynamic Semantics

operations

$m\text{-scalar-relational-operator} : \text{Scalar-relational-operator} \rightarrow (\text{Value} \times \text{Value}) \xrightarrow{o} \text{Value}$
 $m\text{-scalar-relational-operator}(op)(lvalue, rvalue) \triangleq$
 $\quad \text{cases } op:$
 $\quad \text{EQ} \rightarrow m\text{-scalar-eq}(lvalue, rvalue),$
 $\quad \text{NE} \rightarrow m\text{-scalar-ne}(lvalue, rvalue),$
 $\quad \text{LT} \rightarrow m\text{-scalar-lt}(lvalue, rvalue),$
 $\quad \text{GT} \rightarrow m\text{-scalar-gt}(lvalue, rvalue),$
 $\quad \text{LE} \rightarrow m\text{-scalar-le}(lvalue, rvalue),$
 $\quad \text{GE} \rightarrow m\text{-scalar-ge}(lvalue, rvalue)$
 $\quad \text{end};$
 $m\text{-scalar-eq} : \text{Value} \times \text{Value} \xrightarrow{o} \text{Value}$
 $m\text{-scalar-eq}(lvalue, rvalue) \triangleq$
 $\quad \text{return } scalar\text{-value}(lvalue) = scalar\text{-value}(rvalue) ;$
 $m\text{-scalar-ne} : \text{Value} \times \text{Value} \xrightarrow{o} \text{Value}$
 $m\text{-scalar-ne}(lvalue, rvalue) \triangleq$
 $\quad \text{return } scalar\text{-value}(lvalue) \neq scalar\text{-value}(rvalue) ;$

$m\text{-scalar-lt} : \text{Value} \times \text{Value} \xrightarrow{o} \text{Value}$
 $m\text{-scalar-lt}(lvalue, rvalue) \triangleq$
 $\text{return } \text{scalar-value}(lvalue) < \text{scalar-value}(rvalue) ;$

$m\text{-scalar-gt} : \text{Value} \times \text{Value} \xrightarrow{o} \text{Value}$
 $m\text{-scalar-gt}(lvalue, rvalue) \triangleq$
 $\text{return } \text{scalar-value}(lvalue) > \text{scalar-value}(rvalue) ;$

$m\text{-scalar-le} : \text{Value} \times \text{Value} \xrightarrow{o} \text{Value}$
 $m\text{-scalar-le}(lvalue, rvalue) \triangleq$
 $\text{return } \text{scalar-value}(lvalue) \leq \text{scalar-value}(rvalue) ;$

$m\text{-scalar-ge} : \text{Value} \times \text{Value} \xrightarrow{o} \text{Value}$
 $m\text{-scalar-ge}(lvalue, rvalue) \triangleq$
 $\text{return } \text{scalar-value}(lvalue) \geq \text{scalar-value}(rvalue)$

NOTE — A consequence of the approximation inherent in real types is that, for operands of these types, the result from the comparison operations will depend upon the nature of the approximation.

Auxiliary Definitions

functions

$scalar\text{-}value : \text{Value} \rightarrow \text{Number}$
 $scalar\text{-}value(value) \triangleq$
 $(is\text{-}Enumerated\text{-}value(value) \rightarrow \text{let } mk\text{-}Enumerated\text{-}value(id, ids) = value \text{ in}$
 $\quad\quad\quad index(id, ids),$
 $is\text{-}Char(value) \rightarrow collation\text{-}map(value),$
 $others \rightarrow value)$

Set Relational Operations

Five set relational operators are provided.

Abstract Syntax

types

$Set\text{-}relational\text{-}operation :: op : Set\text{-}relational\text{-}operator$

Static Semantics

The set relational operators shall be the set equality, set inequality, superset, subset and set membership operators. Both operands of a set equality, set inequality, superset and subset operator shall be of identical set type.

The right operand of the set membership operator shall be of a set type. The type of the left operand and the host type of the base type of the set type shall be identical.

The result of a set relational operation shall be of the Boolean type.

functions

$wf\text{-}set\text{-}relational\text{-}operation : Set\text{-}relational\text{-}operation \rightarrow (Expression\text{-}typed \times Expression\text{-}typed) \rightarrow Environment \rightarrow \mathbb{B}$
 $wf\text{-}set\text{-}relational\text{-}operation (mk\text{-}Set\text{-}relational\text{-}operation(op))(ltype, rtype)\rho \triangleq$
 $(op \in \{EQ, NE, SUPERSET, SUBSET\} \rightarrow is\text{-}set\text{-}type(ltype)\rho \wedge$
 $ltype = rtype,$
 $op = IN \rightarrow wf\text{-}membership\text{-}expression(ltype, rtype)\rho);$
 $t\text{-}set\text{-}relational\text{-}operation : Set\text{-}relational\text{-}operation \rightarrow Expression\text{-}typed$
 $t\text{-}set\text{-}relational\text{-}operation (mk\text{-}Set\text{-}relational\text{-}operation(-)) \triangleq$
 $BOOLEAN\text{-}TYPE$

Dynamic Semantics

operations

$m\text{-}set\text{-}relational\text{-}operation : Set\text{-}relational\text{-}operation \rightarrow (Value \times Value) \xrightarrow{o} Value$
 $m\text{-}set\text{-}relational\text{-}operation (mk\text{-}Set\text{-}relational\text{-}operation(op))(lvalue, rvalue) \triangleq$
 $(op \in \{EQ, NE, SUPERSET, SUBSET\} \rightarrow \text{let } mk\text{-}Set\text{-}value(lvalues) = lvalue \text{ in}$
 $\text{let } mk\text{-}Set\text{-}value(rvalues) = rvalue \text{ in}$
 $\text{def } res = m\text{-}set\text{-}relational\text{-}operator(op)(lvalues, rvalues);$
 $\text{return } res,$
 $op = IN \rightarrow \text{let } mk\text{-}Set\text{-}value(rvalues) = rvalue \text{ in}$
 $\text{def } res = m\text{-}set\text{-}relational\text{-}operator(op)(lvalue, rvalues);$
 $\text{return } res)$

annotations The actual set of values is contained in a composite object, so the sets must be extracted in order to apply the operator.

Auxiliary Definitions

functions

$wf\text{-}membership\text{-}expression : Expression\text{-}typed \times Expression\text{-}typed \rightarrow Environment \rightarrow \mathbb{B}$
 $wf\text{-}membership\text{-}expression(ltype, rtype)\rho \triangleq$
 $is\text{-}ordinal\text{-}type(ltype)\rho \wedge$
 $is\text{-}set\text{-}type(rtype)\rho \wedge$
 $\text{let } htype = host\text{-}type\text{-}of(base\text{-}type\text{-}of(rtype)\rho)\rho \text{ in}$
 $ltype = htype$

Abstract Syntax

types

$Set\text{-}relational\text{-}operator = EQ \mid NE \mid SUPERSET \mid SUBSET \mid IN$

annotations The above operations correspond to '=', '<>', '>=', '<=' and 'IN' respectively.

Dynamic Semantics

operations

$m\text{-}set\text{-}relational\text{-}operator : Set\text{-}relational\text{-}operator \rightarrow (Value \times Value) \xrightarrow{o} Value$
 $m\text{-}set\text{-}relational\text{-}operator(op)(lvalue, rvalue) \triangleq$
 $\text{cases } op:$
 $EQ \rightarrow m\text{-}set\text{-}eq(lvalue, rvalue),$

```

173      NE      → m-set-ne(lvalue, rvalue) ,
173      SUPERSET → m-superset(lvalue, rvalue) ,
173      SUBSET   → m-subset(lvalue, rvalue) ,
173      IN       → m-in(lvalue, rvalue)
      end

```

The set equality operation shall give a result of true if and only if the two sets contain the same values.

The set inequality operation shall give a result of true if and only if the two sets are not equal.

The superset operation shall give a result of true if and only if all the values in the right operand are also in the left operand.

The subset operation shall give a result of true if and only if all the values in the left operand are also in the right operand.

The set membership operation shall give a result of true if and only if the value of the left operand is a member of the set denoted by the right operand.

NOTE — A consequence of the above rules is that ‘1000 IN DigitSet{0..9}’ produces a result of FALSE and no exception occurs.

Example

Given the declarations:

```

TYPE BaseType = ...
   SetType = SET OF BaseType;
VAR  s, inverse : SetType;

```

Then the inverse of the set **s** can be calculated as follows:

```

inverse := SetType{MIN(BaseType)..MAX(BaseType)} - s;

```

operations

$m\text{-set-eq} : \text{Value} \times \text{Value} \xrightarrow{o} \text{Value}$
 $m\text{-set-eq}(lvalues, rvalues) \triangleq$
 return $rvalues = lvalues$;

$m\text{-set-ne} : \text{Value} \times \text{Value} \xrightarrow{o} \text{Value}$
 $m\text{-set-ne}(lvalues, rvalues) \triangleq$
 return $rvalues \neq lvalues$;

$m\text{-superset} : \text{Value} \times \text{Value} \xrightarrow{o} \text{Value}$
 $m\text{-superset}(lvalues, rvalues) \triangleq$
 return $rvalues \subseteq lvalues$;

$m\text{-subset} : \text{Value} \times \text{Value} \xrightarrow{o} \text{Value}$
 $m\text{-subset}(lvalues, rvalues) \triangleq$
 return $lvalues \subseteq rvalues$;

$m\text{-in} : \text{Value} \times \text{Value} \xrightarrow{o} \text{Value}$
 $m\text{-in}(lvalue, rvalues) \triangleq$
 return $lvalue \in rvalues$

Procedure Relational Operations

Values of structurally identical procedure types can be compared for equality or inequality.

Abstract Syntax

types

Procedure-relational-operation :: *op* : *Procedure-relational-operator*

Static Semantics

The procedure relational operators shall be the procedure equality and procedure inequality operators. The types of the operands of a procedure relational operator shall be structurally identical.

The result of a procedure relational operation shall be of the Boolean type.

functions

wf-procedure-relational-operation :

Procedure-relational-operation \rightarrow (*Expression-typed* \times *Expression-typed*) \rightarrow *Environment* $\rightarrow \mathbb{B}$

wf-procedure-relational-operation (*mk-Procedure-relational-operation*(-))(*ltype*, *rtype*) $\rho \triangleq$

(*is-Type-name*(*ltype*) \wedge *is-Type-name*(*rtype*) \rightarrow *ltype* = *rtype* \wedge

279 *is-procedure-type*(*ltype*) ρ ,

271 *is-Type-name*(*ltype*) \wedge *is-Procedure-type*(*rtype*) \rightarrow *structure-of*(*ltype*) ρ = *rtype.proc*,

271 *is-Procedure-type*(*ltype*) \wedge *is-Type-name*(*rtype*) \rightarrow *ltype.proc* = *structure-of*(*rtype*) ρ ,

is-Procedure-type(*ltype*) \wedge *is-Procedure-type*(*rtype*) \rightarrow *ltype.proc* = *rtype.proc*);

t-procedure-relational-operation : *Procedure-relational-operation* \rightarrow *Expression-typed*

t-procedure-relational-operation (*mk-Procedure-relational-operation*(-)) \triangleq

BOOLEAN-TYPE

Dynamic Semantics

operations

m-procedure-relational-operation : *Procedure-relational-operation* \rightarrow (*Value* \times *Value*) \xrightarrow{o} *Value*

m-procedure-relational-operation (*mk-Procedure-relational-operation*(*op*))(*lvalue*, *rvalue*) \triangleq

cases *op*:

175 *mk-Function-procedure-value*(*lden*) \rightarrow let *mk-Function-procedure-value*(*rden*) = *rvalue* in
m-procedure-relational-operator(*op*)(*lden*, *rden*),

175 *mk-Propert-procedure-value*(*lden*) \rightarrow let *mk-Propert-procedure-value*(*rden*) = *rvalue* in
m-procedure-relational-operator(*op*)(*lden*, *rden*)

end

Abstract Syntax

types

Procedure-relational-operator = EQ | NE

annotations The above operators correspond to ‘=’ and ‘<>’ respectively.

Language Clarification

The equality and inequality operators are applicable to procedures.

Dynamic Semantics

operations

$m\text{-procedure-relational-operator} : \text{Procedure-relational-operator} \rightarrow (\text{Value} \times \text{Value}) \xrightarrow{o} \text{Value}$
 $m\text{-procedure-relational-operator}(op)(lvalue, rvalue) \triangleq$
cases op :
175 EQ $\rightarrow m\text{-proc-eq}(lvalue, rvalue)$,
175 NE $\rightarrow m\text{-proc-ne}(lvalue, rvalue)$
end

The procedure equality operation shall give the result **TRUE** if and only if the values of both operands denote the same procedure text.

The procedure non-equality operation shall give the result **TRUE** unless the values of both operands denote the same procedure text.

operations

$m\text{-proc-eq} : \text{Value} \times \text{Value} \xrightarrow{o} \text{Value}$
 $m\text{-proc-eq}(lvalue, rvalue) \triangleq$
return $lvalue = rvalue$;

 $m\text{-proc-ne} : \text{Value} \times \text{Value} \xrightarrow{o} \text{Value}$
 $m\text{-proc-ne}(lvalue, rvalue) \triangleq$
return $lvalue \neq rvalue$

Pointer Relational Operations

Values of pointer and opaque types can be compared for equality or inequality.

Abstract Syntax

types

$\text{Pointer-relational-operation} :: op : \text{Pointer-relational-operator}$

Static Semantics

The pointer relational operators shall be the pointer equality and pointer inequality operators. The types of the operands of a pointer relational operator shall be identical.

The result of a pointer relational operation shall be of the Boolean type.

NOTE — Since opaque types are similar to pointer types, pointer equality and inequality can be applied to values of opaque types.

functions

$wf\text{-pointer-relational-operation} :$
 $\text{Pointer-relational-operation} \rightarrow (\text{Expression-typed} \times \text{Expression-typed}) \rightarrow \text{Environment} \rightarrow \mathbb{B}$
 $wf\text{-pointer-relational-operation}(mk\text{-Pointer-relational-operation}(-))(ltype, rtype)\rho \triangleq$
 $(is\text{-Type-name}(ltype) \wedge is\text{-Type-name}(rtype) \rightarrow ltype = rtype \wedge$
279 $(is\text{-pointer-type}(ltype)\rho \vee is\text{-opaque-type}(ltype)\rho),$
279 $is\text{-Type-name}(ltype) \wedge rtype = \text{NIL-TYPE} \rightarrow is\text{-pointer-type}(ltype)\rho,$
279 $ltype = \text{NIL-TYPE} \wedge is\text{-Type-name}(rtype) \rightarrow is\text{-pointer-type}(rtype)\rho,$
 $ltype = \text{NIL-TYPE} \wedge rtype = \text{NIL-TYPE} \rightarrow \text{true});$

$t\text{-pointer-relational-operation} : \text{Pointer-relational-operation} \rightarrow \text{Expression-typed}$

$t\text{-pointer-relational-operation}(\text{mk-Pointer-relational-operation}(-)) \triangleq$
 BOOLEAN-TYPE

annotations Opaque types are indistinguishable from pointer types with respect to the meaning function. The well-formed conditions forbid the dereferencing of a value of an opaque type outside of the implementation module in which the opaque type is declared (see section ??).

Dynamic Semantics

operations

$m\text{-pointer-relational-operation} : \text{Pointer-relational-operation} \rightarrow (\text{Value} \times \text{Value}) \xrightarrow{o} \text{Value}$

$m\text{-pointer-relational-operation}(\text{mk-Pointer-relational-operation}(op))(lvalue, rvalue) \triangleq$

let $\text{mk-Pointer-value}(plvalue) = lvalue$ in

let $\text{mk-Pointer-value}(prvalue) = rvalue$ in

176 $m\text{-pointer-relational-operator}(op)(plvalue, prvalue)$

Abstract Syntax

types

$\text{Pointer-relational-operator} = \text{EQ} \mid \text{NE}$

Dynamic Semantics

operations

$m\text{-pointer-relational-operator} : \text{Pointer-relational-operator} \rightarrow (\text{Value} \times \text{Value}) \xrightarrow{o} \mathbb{B}$

$m\text{-pointer-relational-operator}(op)(lvalue, rvalue) \triangleq$

cases op :

176 $\text{EQ} \rightarrow m\text{-pointer-eq}(lvalue, rvalue),$

176 $\text{NE} \rightarrow m\text{-pointer-ne}(lvalue, rvalue)$

end

annotations The above operations correspond to ‘=’ and ‘<>’ respectively.

Dynamic Semantics

The pointer equality operation shall give the result true if and only if the values of both operands denote the same variable or they have the same value denoted by the pervasive identifier **NIL**.

The pointer non-equality operation shall give the result true if and only if the values of both operands do not denote the same variable, and they do not have the value denoted by the pervasive identifier **NIL**.

NOTE — Since pointer values can be constructed via **ADDRESS**, the semantics is defined in terms of the underlying model rather than the objects held in the store.

operations

$m\text{-pointer-eq} : \text{Value} \times \text{Value} \xrightarrow{o} \text{Value}$

$m\text{-pointer-eq}(plvalue, prvalue) \triangleq$

return $plvalue = prvalue$;

$m\text{-pointer-ne} : \text{Value} \times \text{Value} \xrightarrow{o} \text{Value}$

$m\text{-pointer-ne}(plvalue, prvalue) \triangleq$

return $plvalue \neq prvalue$

Protection Relational Operations

Values of protection types can be compared.

The infix relational operators equal, unequal, greater than or equal, and less than or equal shall be defined on values of Protection type.

Abstract Syntax

types

Protection-relational-operation :: *op* : *Protection-relational-operator*

Static Semantics

The protection relational operators shall be the protection equal, unequal, greater than or equal, and less than or equal operators. The types of the operands of a protection relational operator shall be of Protection type.

The result of a protection relational operation shall be of the Boolean type.

functions

wf-protection-relational-operation :
 Protection-relational-operation \rightarrow (*Expression-typed* \times *Expression-typed*) \rightarrow *Environment* $\rightarrow \mathbb{B}$
wf-protection-relational-operation (*mk-Protection-relational-operation*(-))(*ltype*, *rtype*) $\rho \triangleq$
 ltype = PROTECTION-TYPE \wedge *rtype* = PROTECTION-TYPE;
t-protection-relational-operation : *Protection-relational-operation* \rightarrow *Expression-typed*
t-protection-relational-operation (*mk-Protection-relational-operation*(-)) \triangleq
 BOOLEAN-TYPE

Dynamic Semantics

operations

m-protection-relational-operation : *Protection-relational-operation* \rightarrow (*Value* \times *Value*) \xrightarrow{o} *Value*
m-protection-relational-operation (*mk-Protection-relational-operation*(*op*))(*lvalue*, *rvalue*) \triangleq
177 *m-protection-relational-operator*(*op*) (*lvalue*, *rvalue*)

Abstract Syntax

types

Protection-relational-operator = EQ | NE | LE | GE

annotations The above operations correspond to '=', '<>', '<=' and '>=' respectively.

Dynamic Semantics

operations

m-protection-relational-operator : *Protection-relational-operator* \rightarrow (*Value* \times *Value*) \xrightarrow{o} *Value*
m-protection-relational-operator (*op*)(*lvalue*, *rvalue*) \triangleq
 cases *op*:
178 EQ \rightarrow *m-prot-eq*(*lvalue*, *rvalue*) ,
178 NE \rightarrow *m-prot-ne*(*lvalue*, *rvalue*) ,

```

178      LE  $\rightarrow$  m-prot-le(lvalue, rvalue) ,
178      GE  $\rightarrow$  m-prot-ge(lvalue, rvalue)
      end

```

The protection equality operation shall give the result true if and only if the values of both operands are the same protection value.

The protection non-equality operation shall give the result true if and only if the values of both operands are not the same protection value.

The order operations shall return a result depending on an implementation defined order on the protection values. The ordering shall be such that a value of protection type, P1, shall compare less than another value, P2, if and only if P1 permits all the interrupts permitted by P2 but P2 does not permit all the interrupts permitted by P1.

operations

m-prot-eq : *Value* \times *Value* \xrightarrow{o} *Value*
m-prot-eq (*lvalue*, *rvalue*) \triangleq
 return *lvalue* = *rvalue* ;

m-prot-ne : *Value* \times *Value* \xrightarrow{o} *Value*
m-prot-ne (*lvalue*, *rvalue*) \triangleq
 return *lvalue* \neq *rvalue* ;

m-prot-le : *Value* \times *Value* \xrightarrow{o} *Value*
m-prot-le (*lvalue*, *rvalue*) \triangleq
 return *lvalue* $\leq_{\text{protection}}$ *rvalue* ;

m-prot-ge : *Value* \times *Value* \xrightarrow{o} *Value*
m-prot-ge (*lvalue*, *rvalue*) \triangleq
 return *lvalue* $\geq_{\text{protection}}$ *rvalue*

annotations The operators $\leq_{\text{protection}}$ and $\geq_{\text{protection}}$ are implementation defined.

6.7.2 Prefix Expressions

A prefix expression obtains a value by applying a prefix operator to an operand value.

Concrete Syntax

sign = ["+" | "-"] ;
 not operator = "NOT" | "^" ;

The two forms of the not operator shall be equivalent.

Abstract Syntax

types

Prefix-expression :: *op* : *Prefix-operation*
 expr : *Expression*

Static Semantics

The result of a prefix operation shall be of the same type as the expression.

functions

$$wf\text{-}prefix\text{-}expression : Prefix\text{-}expression \rightarrow Environment \rightarrow \mathbb{B}$$
$$wf\text{-}prefix\text{-}expression (mk\text{-}Prefix\text{-}expression(op, expr))\rho \triangleq$$

152 $wf\text{-}expression(expr)\rho \wedge$
let $etype = t\text{-}expression(expr)\rho$ in
179 $wf\text{-}prefix\text{-}operation(op, etype)\rho;$

$$t\text{-}prefix\text{-}expression : Prefix\text{-}expression \rightarrow Environment \rightarrow Expression\text{-}typed$$
$$t\text{-}prefix\text{-}expression (mk\text{-}Prefix\text{-}expression(op, -))\rho \triangleq$$

179 $t\text{-}prefix\text{-}operation(op)$

Dynamic Semantics

The expression shall be evaluated first and then the prefix operator applied.

operations

$$m\text{-}prefix\text{-}expression : Prefix\text{-}expression \rightarrow Environment \xrightarrow{o} Value$$
$$m\text{-}prefix\text{-}expression (mk\text{-}Prefix\text{-}expression(op, expr))\rho \triangleq$$

153 $def\ value = m\text{-}expression(expr)\rho;$
179 $m\text{-}prefix\text{-}operation(op, value)\rho$

Abstract Syntax

types

$$Prefix\text{-}operation = Arithmetic\text{-}prefix\text{-}operation \mid Boolean\text{-}prefix\text{-}operation$$

Static Semantics

functions

$$wf\text{-}prefix\text{-}operation : Prefix\text{-}operation \times Expression\text{-}typed \rightarrow \mathbb{B}$$
$$wf\text{-}prefix\text{-}operation (op, etype) \triangleq$$

180 $(is\text{-}Arithmetic\text{-}prefix\text{-}operation(op) \rightarrow wf\text{-}arithmetic\text{-}prefix\text{-}operation (op, etype),$
181 $is\text{-}Boolean\text{-}prefix\text{-}operation(op) \rightarrow wf\text{-}Boolean\text{-}prefix\text{-}operation (op, etype));$

$$t\text{-}prefix\text{-}operation : Prefix\text{-}operation \rightarrow Expression\text{-}typed$$
$$t\text{-}prefix\text{-}operation (op) \triangleq$$

180 $(is\text{-}Arithmetic\text{-}prefix\text{-}operation(op) \rightarrow wf\text{-}arithmetic\text{-}prefix\text{-}operation (op)\rho,$
181 $is\text{-}Boolean\text{-}prefix\text{-}operation(op) \rightarrow wf\text{-}Boolean\text{-}prefix\text{-}operation (op)\rho)$

Dynamic Semantics

operations

$$m\text{-}prefix\text{-}operation : Prefix\text{-}operation \times Value \xrightarrow{o} Value$$
$$m\text{-}prefix\text{-}operation (op, value) \triangleq$$

180 $(is\text{-}Arithmetic\text{-}prefix\text{-}operation(op) \rightarrow m\text{-}arithmetic\text{-}prefix\text{-}operation(op, value),$
182 $is\text{-}Boolean\text{-}prefix\text{-}operation(op) \rightarrow m\text{-}Boolean\text{-}prefix\text{-}operation(op, value))$

6.7.2.1 Arithmetic Prefix Operations

Two arithmetic prefix operations are provided, corresponding to an identity operation (prefix plus) and to negation (prefix minus). These operations are denoted lexically by the symbols ‘+’ and ‘-’ respectively. The results of complex number and real number prefix operations are approximations to the corresponding mathematical operations, while the results of whole number prefix operations are exact, within implementation defined ranges.

Abstract Syntax

types

$$\begin{aligned} \textit{Arithmetic-prefix-operation} &:: \textit{op} : \textit{Arithmetic-prefix-operator} \\ &\quad \textit{otype} : \textit{Number-type} \end{aligned}$$

Static Semantics

The operand of an arithmetic prefix operation shall be a value of a complex number type, a real number type or a whole number type, except that the prefix minus operator shall not be applied to a value of the unsigned type.

NOTE — The semantics disallow expressions of the form $(-a+b)$ when a and b are of the unsigned type.

functions

$$\begin{aligned} \textit{wf-arithmetic-prefix-operation} &: \textit{Arithmetic-prefix-operation} \times \textit{Expression-typed} \rightarrow \mathbb{B} \\ \textit{wf-arithmetic-prefix-operation}(\textit{mk-Arithmetic-prefix-operation}(\textit{op}, \textit{otype}), \textit{etype}) &\triangleq \\ 180 \quad \textit{otype} = \textit{arithmetic-prefix-operation-result-type}(\textit{op}, \textit{etype}); \\ \textit{t-arithmetic-prefix-operation} &: \textit{Arithmetic-prefix-operation} \rightarrow \textit{Expression-typed} \\ \textit{t-arithmetic-prefix-operation}(\textit{mk-Arithmetic-prefix-operation}(-, \textit{otype})) &\triangleq \\ \textit{otype} \end{aligned}$$

Dynamic Semantics

operations

$$\begin{aligned} \textit{m-arithmetic-prefix-operation} &: \textit{Arithmetic-prefix-operation} \times \textit{Value} \xrightarrow{o} \textit{Value} \\ \textit{m-arithmetic-prefix-operation}(\textit{mk-Arithmetic-prefix-operation}(\textit{op}, \textit{otype}), \textit{value}) &\triangleq \\ 77 \quad \textit{def val} = \textit{m-arithmetic-prefix-operator}(\textit{op}, \textit{value}); \\ 162 \quad (\textit{is-Whole-number-type}(\textit{otype}) \rightarrow \textit{get-whole-result}(\textit{otype}, \textit{val}), \\ 159 \quad \textit{is-Real-number-type}(\textit{otype}) \rightarrow \textit{get-real-result}(\textit{otype}, \textit{val}), \\ 157 \quad \textit{is-Complex-number-type}(\textit{otype}) \rightarrow \textit{get-complex-result}(\textit{otype}, \textit{val})) \end{aligned}$$

Auxiliary Definitions

functions

$$\begin{aligned} \textit{arithmetic-prefix-operation-result-type} &: \textit{Arithmetic-prefix-operator} \times \textit{Typed} \rightarrow \textit{Typed} \\ \textit{arithmetic-prefix-operation-result-type}(\textit{op}, \textit{type}) &\triangleq \\ \textit{if is-Number-type}(\textit{type}) & \\ \textit{then if } \textit{type} = \text{UNSIGNED-TYPE} \wedge \textit{op} = \text{MINUS} & \\ \textit{then undefined} & \\ \textit{else } \textit{type} & \\ \textit{else undefined} & \end{aligned}$$

annotations Return the type of a prefix operation. The result is undefined if the prefix minus operator is applied to the unsigned type or if it is applied to a value of a non-numeric type.

Abstract Syntax

types

$Arithmetic-prefix-operator = PLUS \mid MINUS$

annotations The above operators correspond to ‘+’ and ‘-’ respectively.

Dynamic Semantics

functions

```

m-arithmetic-prefix-operator : Arithmetic-prefix-operator × Value  $\xrightarrow{o}$  Value
m-arithmetic-prefix-operator (op, value)  $\triangleq$ 
  cases op :
??   PLUS  → m-plus (value),
??   MINUS → m-minus (value)
      end

```

Dynamic Semantics

The prefix plus operation shall have no effect on the value and the prefix minus operation shall change the sign of the value.

NOTE — An exception may occur with the prefix minus operation for numeric types having non-symmetric ranges.

operations

```

m-plus : Value  $\xrightarrow{o}$  Value
m-plus (value)  $\triangleq$ 
  return value ;

m-minus : Value  $\xrightarrow{o}$  Value
m-minus (value)  $\triangleq$ 
  return − value

```

6.7.2.2 Boolean Prefix Operations

One Boolean prefix operation is provided, corresponding to logical negation. This operation is denoted lexically by the symbol ‘**NOT**’ or ‘~’.

Abstract Syntax

types

$Boolean-prefix-operation :: op : Boolean-prefix-operator$

Static Semantics

The operand of the **NOT** operator shall be of the Boolean type and the result shall also be of the Boolean type.

functions

```

wf-Boolean-prefix-operation : Boolean-prefix-operation → Type →  $\mathbb{B}$ 
wf-Boolean-prefix-operation (mk-Boolean-prefix-operation (-))(etype)  $\triangleq$ 
  etype = BOOLEAN-TYPE;

```

$t\text{-Boolean-prefix-operation} : \text{Boolean-prefix-operation} \rightarrow \text{Expression-typed}$
 $t\text{-Boolean-prefix-operation}(\text{mk-Boolean-prefix-operation}(-)) \triangleq$
 BOOLEAN-TYPE

Dynamic Semantics

operations

$m\text{-Boolean-prefix-operation} : \text{Boolean-prefix-operation} \rightarrow \mathbb{B} \xrightarrow{o} \mathbb{B}$
 $m\text{-Boolean-prefix-operation}(\text{mk-Boolean-prefix-operation}(op), value) \triangleq$
 $m\text{-not}(value)$

182

Abstract Syntax

types

$\text{Boolean-prefix-operator} = \text{NOT}$

annotations The above operator corresponds to ‘**NOT**’ or ‘ \sim ’.

Dynamic Semantics

The **NOT** operator shall give **TRUE** for the operand value **FALSE** and **FALSE** for the operand value **TRUE**.

operations

$m\text{-not} : \text{Value} \xrightarrow{o} \text{Value}$
 $m\text{-not}(value) \triangleq$
 $\text{return } \neg value$

6.7.3 Value Designators

A value designator appears in an expression to denote the value of an object such as a constant identifier or a variable identifier, or the value of a component of such an object.

Concrete Syntax

value designator = entire value | indexed value | selected value | dereferenced value ;

Abstract Syntax

types

$\text{Value-designator} = \text{Entire-value}$
 $\quad \quad \quad | \text{Indexed-value}$
 $\quad \quad \quad | \text{Selected-value}$
 $\quad \quad \quad | \text{Dereferenced-value}$

Static Semantics

functions

$wf\text{-value-designator} : \text{Value-designator} \rightarrow \text{Environment} \rightarrow \mathbb{B}$

$wf\text{-value-designator}(vd)\rho \triangleq$

cases vd :

```
184      mk-Entire-value(-)      → wf-entire-value(vd)ρ,
186      mk-Indexed-value(-, -)  → wf-indexed-value(vd)ρ,
187      mk-Selected-value(-, -) → wf-selected-value(vd)ρ,
188      mk-Dereferenced-value(-) → wf-dereferenced-value(vd)ρ
end
```

functions

$t\text{-value-designator} : \text{Value-designator} \rightarrow \text{Environment} \rightarrow \text{Typed}$

$t\text{-value-designator}(vd)\rho \triangleq$

cases vd :

```
184      mk-Entire-value(-)      → t-entire-value(vd)ρ,
186      mk-Indexed-value(-, -)  → t-indexed-value(vd)ρ,
187      mk-Selected-value(-, -) → t-selected-value(vd)ρ,
188      mk-Dereferenced-value(-) → t-dereferenced-value(vd)ρ
end
```

Dynamic Semantics

operations

$m\text{-value-designator} : \text{Value-designator} \rightarrow \text{Environment} \xrightarrow{o} \text{Value}$

$m\text{-value-designator}(vd)\rho \triangleq$

cases :

```
184      mk-Entire-value(-)      → m-entire-value(vd) ρ,
186      mk-Indexed-value(-, -)  → m-indexed-value(vd) ρ,
187      mk-Selected-value(-, -) → m-selected-value(vd) ρ,
188      mk-Dereferenced-value(-) → m-dereferenced-value(vd) ρ
end
```

6.7.3.1 Entire Values

An entire value consists of a qualified identifier referring to a value.

Concrete Syntax

entire value = qualified identifier ;

Abstract Syntax

types

$\text{Entire-value} :: \text{qid} : \text{Qualident}$

annotations Within the body of a with statement an entire value may also be the field identifier of a record value which has been specified in the designator component of the with statement.

Static Semantics

An entire designator shall denote a value.

The type of an entire value shall be defined by the declaration of the designator that denotes the value or by the declaration of the variable containing the value.

functions

```
wf-entire-value : Entire-value → Environment → B
wf-entire-value (mk-Entire-value(qid))ρ  $\triangleq$ 
270 (is-constant (qid)ρ ∨ is-variable (qid)ρ ∨ is-procedure (qid)ρ) ∧
?? wf-qualident (qid)ρ;

t-entire-value : Entire-value → Environment → Expression-typed
t-entire-value (mk-Entire-value(qid))ρ  $\triangleq$ 
276 let object = access-environment (qid)ρ in
270 let rtype = (is-Constant-value(object) → type-of-constant (qid)ρ,
272 is-Variable-typed(object) → type-of-variable (qid)ρ,
273 is-Procedure-typed(object) → type-of-procedure (qid)ρ) in
283 host-type-of (rtype)ρ
```

Dynamic Semantics

The result of the evaluation of an entire value shall be the value associated with the qualified identifier.

The value of a value designator shall be the value of the variable. It shall be an exception if the value of the variable is undefined.

NOTES

- 1 The evaluation of a designator can have a side-effect via a function designator in a subscript expression.
- 2 Only structured objects can have components which may be undefined, see 6.5.3

operations

```
m-entire-value : Entire-value → Environment  $\xrightarrow{o}$  Value
m-entire-value (mk-Entire-value(qid))ρ  $\triangleq$ 
276 let object = access-environment (qid)ρ in
( is-Constant-value(object) → return object.value,
184 is-Variable(object) → value-of-a-variable(object) ρ,
is-Procedure-value(object) → return object)
```

Auxiliary Definitions

operations

```
value-of-a-variable : Variable  $\xrightarrow{o}$  Value
value-of-a-variable (var)  $\triangleq$ 
184 def val = variable-value(var) ;
if val = nil
306 then mandatory-exception(UNDEFINED-VALUE)
else return val ;
```

```

variable-value : Variable  $\xrightarrow{o}$  [Value]
variable-value (v)  $\triangleq$ 
  cases v:
298    mk-Elementary-variable(loc, -)       $\rightarrow$  contents(loc) ,
    mk-Array-variable(vars)               $\rightarrow$  dcl mp : Value  $\xrightarrow{m}$  Value := { };
    for all i  $\in$  inds vars
184    do def ival = variable-value(vars(i)) ;
    mp := mp  $\uparrow$  { i  $\mapsto$  ival } return mp ,
    mk-Record-variable(fields)             $\rightarrow$  dcl mp : Value  $\xrightarrow{m}$  Value := { };
    for all id  $\in$  dom fields
184    do def ival = variable-value(fields(id)) ;
    if ival  $\neq$  nil
    then mp := mp  $\uparrow$  { id  $\mapsto$  ival }
    else skip return mp ,
184    mk-Tag-variable(var, -, -)           $\rightarrow$  variable-value(var) ,
184    mk-Variant-Variable(var, tagloc, tagvals)  $\rightarrow$  def tval = variable-value(tagloc) ;
    if tval  $\in$  tagvals
184    then variable-value(var)
    else return nil ,
184    mk-Tagged-Variable(var, tagvar, tagvals)  $\rightarrow$  def tval = variable-value(tagvar) ;
    if tval  $\in$  tagvals
184    then variable-value(var)
    else return nil
  end

```

6.7.3.2 Indexed Values

An indexed value is an array value followed by one or more index expressions, and denotes a component value of a value of array type.

Concrete Syntax

indexed value = array value, left bracket, index expression, { ",", index expression }, right bracket ;

array value = value designator ;

In the concrete syntax, if the indexed value is itself an indexed value an abbreviation may be used. In the abbreviated form, a single comma shall replace the sequence '] [' that occurs in the full form. The abbreviated form and the full form shall be equivalent.

NOTE — If the indexed designator is written in the abbreviated form, the number of index expressions cannot exceed the dimension of the array.

TO DO — An example of a multi-dimensional array to illustrate the concrete syntax to be inserted.

Abstract Syntax

types

Indexed-value :: *desig* : *Value-designator*
expr : *Expression*

annotations The abstract syntax only uses the full form.

Static Semantics

The value designator shall be a value of an array type. The type of the index expression shall be assignment compatible with the index type of the array.

The type of an indexed value shall be the component type of the array.

functions

```

wf-indexed-value : Indexed-value → Environment → B
wf-indexed-value (mk-Indexed-value(design, expr))ρ  $\triangleq$ 
183   wf-value-designator(design)ρ ∧
152   wf-expression(expr)ρ ∧
183   let type = t-value-designator(design)ρ in
278   is-array-type(type)ρ ∧
284   let itype = index-type-of(type)ρ in
152   assignment-compatible(itype, t-expression(expr)ρ)ρ;

t-indexed-value : Indexed-value → Environment → Expression-typed
t-indexed-value (mk-Indexed-value(design, -))ρ  $\triangleq$ 
183   let type = t-value-designator(design)ρ in
284   component-type-of(type)ρ

```

Dynamic Semantics

The result of the evaluation of the value designator shall be an array value; the result of the evaluation of the index expression component shall be an ordinal value. If this ordinal value is a value of the index type of the array value, the indexed value shall be the value of the component of the array value selected by the ordinal value. It shall be an exception if the ordinal value is not a value belonging to the range of values defined by the index type of the array.

The order of evaluation of the value designator and the index expression of an indexed value shall be implementation dependent.

operations

```

m-indexed-value : Indexed-value → Environment  $\xrightarrow{o}$  Value
m-indexed-value (mk-Indexed-value(design, expr))ρ  $\triangleq$ 
183   def array = m-value-designator(design)ρ;
153   def index = m-expression(expr)ρ;
   if index ∈ dom array
   then return array(index)
306   else mandatory-exception(INDEX-RANGE)

```

annotations The evaluation order of the array value and the index expression of the indexed value given above is one of those permitted, right-to-left or parallel being allowed also, see section 4.1.

6.7.3.3 Selected Values

A selected value is a record value designator followed by a field identifier, and denotes a component value of a value of a record type.

Concrete Syntax

selected value = record value, ".", field identifier ;

record value = value designator ;

Abstract Syntax

types

Selected-value :: *desig* : Value-designator
 id : Identifier

Static Semantics

The value designator shall be a value of a record type, and the field identifier shall be one of the identifiers of the identifier list or the tag identifier of a field list of the record-type.

The type of a selected value shall be given by the type of the field identifier.

functions

wf-selected-value : *Selected-value* \rightarrow *Environment* \rightarrow \mathbb{B}
wf-selected-value (*mk-Selected-value*(*desig*, *id*)) $\rho \triangleq$
183 *wf-value-designator* (*desig*) $\rho \wedge$
183 let *type* = *t-value-designator* (*desig*) ρ in
278 *is-record-type* (*type*) $\rho \wedge$
 id \in dom *field-types-of-record* (*type*) ρ ;

t-selected-value : *Selected-value* \rightarrow *Environment* \rightarrow *Expression-typed*
t-selected-value (*mk-Selected-value*(*desig*, *id*)) $\rho \triangleq$
183 let *type* = *t-value-designator* (*desig*) ρ in
285 let *types* = *field-types-of-record* (*type*) ρ in
 types (*id*)

Dynamic Semantics

The value designator component shall be evaluated to give a record value. The selected value shall be the value of the component of the record value selected by the field identifier.

It shall be an exception if an attempt is made to access a value from a non-active component of a variant record.

operations

m-selected-value : *Selected-value* \rightarrow *Environment* \xrightarrow{o} *Value*
m-selected-value (*mk-Selected-value*(*desig*, *id*)) $\rho \triangleq$
183 def *recvar* = *m-value-designator*(*desig*) ρ ;
 if *id* \in dom *recvar*
 then return *recvar*(*id*)
306 else *non-mandatory-exception*(INACTIVE-VARIANT)

6.7.3.4 Dereferenced Values

A dereferenced value is a pointer value followed by a dereferencing operator, and denotes the value (if any) referenced by the pointer value.

Concrete Syntax

dereferenced value = pointer value, dereferencing operator ;

pointer value = value designator ;

Abstract Syntax

types

$Dereferenced-value :: design : Value-designator$

Static Semantics

The value designator shall be a value of a pointer type.

The type of a dereferenced value shall be the bound type specified in the declaration of the pointer type.

functions

$wf-dereferenced-value : Dereferenced-value \rightarrow Environment \rightarrow \mathbb{B}$

```
183   $wf-dereferenced-value (mk-Dereferenced-value(design))\rho \triangleq$   
183   $wf-value-designator(design)\rho \wedge$   
279   $let\ type = t-value-designator(design)\rho\ in$   
279   $is-pointer-type(type)\rho;$ 
```

$t-dereferenced-value : Dereferenced-value \rightarrow Environment \rightarrow Expression-typed$

```
183   $t-dereferenced-value (mk-Dereferenced-value(design))\rho \triangleq$   
284   $let\ type = t-value-designator(design)\rho\ in$   
284   $bound-type-of(type)\rho$ 
```

Dynamic Semantics

The value designator shall be evaluated to give a value. If the value is neither the value of the pervasive identifier **NIL** nor undefined then the dereferenced value shall be the value associated with the variable denoted by that value. It shall be an exception if the value is either the value of the pervasive identifier **NIL** or undefined.

NOTE — Since the only constant of a pointer type is **NIL**, no dereferenced values can be obtained from a constant.

operations

$m-dereferenced-value : Dereferenced-value \rightarrow Environment \xrightarrow{o} Value$

```
183   $m-dereferenced-value (mk-Dereferenced-value(design))\rho \triangleq$   
184   $def\ var = m-value-designator(design)\rho;$   
184   $(\ exists\ (var) \rightarrow value-of-a-variable(var) ,$   
306   $var = nil \rightarrow mandatory-exception(NIL-REFERENCE) ,$   
306   $others \rightarrow non-mandatory-exception(NONEXISTENT) )$ 
```

6.7.4 Function Calls

A function call specifies the evaluation of the parameters of the parameter list (if present) and the activation of the block associated with the function designator, and yields the value of the result of the activation upon completion of the activation.

Concrete Syntax

function call = function designator, "(", [actual parameter list], ")" ;

Abstract Syntax

types

Function-call :: *desig* : *Function-designator*
 args : *Actual-parameters*

annotations An empty parameter list in the concrete syntax of the call is represented in the abstract syntax by an empty sequence of actual parameters.

Static Semantics

If a function has any formal parameters, the function call shall contain a list of actual parameters that shall be bound to their corresponding formal parameters defined in the function declaration. The correspondence shall be established by the positions of the parameters in the lists of actual and formal parameters respectively. The number of actual parameters shall be equal to the number of formal parameters.

functions

wf-function-call : *Function-call* \rightarrow *Environment* \rightarrow \mathbb{B}
wf-function-call (*mk-Function-call*(*desig*, *aps*)) $\rho \triangleq$
190 *wf-function-designator* (*desig*, *args*) $\rho \wedge$
190 let *type* = *t-function-designator* (*desig*, *args*) ρ in
279 *is-function-procedure-type* (*type*) $\rho \wedge$
219 *wf-actual-parameters* (*args*) ρ
;

t-function-call : *Function-call* \rightarrow *Environment* \rightarrow *Expression-typed*
t-function-call (*mk-Function-call*(*desig*, *args*)) $\rho \triangleq$
190 *t-function-designator* (*desig*, *args*) ρ

Dynamic Semantics

A function call shall specify the evaluation of the actual parameters, the binding of the results of this evaluation to the corresponding formal parameters, and the activation of the block associated with the function procedure. The order of evaluation, accessing and binding of the actual-parameters is implementation dependent.

Actual parameters shall be evaluated before the activation of the block.

operations

m-function-call : *Function-call* \rightarrow *Environment* \xrightarrow{o} *Value*
m-function-call (*mk-Function-call*(*desig*, *args*)) $\rho \triangleq$
191 def *f* = *m-function-designator*(*desig*) ρ ;
219 def *argvals* = *m-actual-parameters*(*args*) ρ ;
 (*is-Function-value*(*f*) \rightarrow *f*(*argvals*) ,
 is-Standard-function-designator(*f*) \rightarrow *m-standard-function-designator*(*f*) (*argvals*) ρ ,
 is-System-function-designator(*f*) \rightarrow *m-system-function-designator*(*f*) (*argvals*) ρ ,
 is-Coroutine-function-designator(*f*) \rightarrow *m-coroutine-function-designator*(*f*) (*argvals*) ρ)

6.7.4.1 Function Designators

Concrete Syntax

function designator = value designator ;

Abstract Syntax

types

$$\begin{aligned} \text{Function-designator} &= \text{Value-designator} \\ &\quad | \text{Standard-function} \\ &\quad | \text{System-function} \\ &\quad | \text{Coroutine-function}; \\ \\ \text{Standard-function} &:: id : \text{Identifier} \\ &\quad \text{desig} : \text{Standard-function-designator}; \\ \\ \text{System-function} &:: qid : \text{Qualident} \\ &\quad \text{desig} : \text{System-function-designator} ; \\ \\ \text{Coroutine-function} &:: qid : \text{Qualident} \\ &\quad \text{desig} : \text{Coroutine-function-designator} \end{aligned}$$

Static Semantics

functions

$$\begin{aligned} &wf\text{-function-designator} : \text{Function-designator} \times \text{Actual-parameters} \rightarrow \text{Environment} \rightarrow \mathbb{B} \\ &wf\text{-function-designator}(fdesig, args)\rho \triangleq \\ 183 \quad &(\text{is-Value-designator}(fdesig) \rightarrow wf\text{-value-designator}(fdesig)\rho \wedge \\ 220 \quad &\quad \text{is-parameters-match}(desig)(args)\rho, \\ &\text{is-Standard-function}(fdesig) \rightarrow \text{let } mk\text{-Standard-function}(id, desig) = fdesig \text{ in} \\ ?? \quad &\quad \text{is-standard-function}(id)\rho \wedge \\ 235 \quad &\quad wf\text{-standard-function-designator}(desig)(args)\rho, \\ &\text{is-System-function}(fdesig) \rightarrow \text{let } mk\text{-System-function}(qid, desig) = fdesig \text{ in} \\ ?? \quad &\quad \text{is-system-function}(qid)\rho \wedge \\ 321 \quad &\quad wf\text{-system-function-designator}(desig)(args)\rho, \\ &\text{is-Coroutine-function}(fdesig) \rightarrow \text{let } mk\text{-Coroutine-function}(qid, desig) = fdesig \text{ in} \\ ?? \quad &\quad \text{is-coroutine-function}(qid)\rho \wedge \\ 344 \quad &\quad wf\text{-coroutine-function-designator}(desig)(args)\rho) \end{aligned}$$

functions

$$\begin{aligned} &t\text{-function-designator} : \text{Function-designator} \times \text{Actual-parameters} \rightarrow \text{Environment} \rightarrow \text{Expression-typed} \\ &t\text{-function-designator}(fdesig, args)\rho \triangleq \\ ?? \quad &(\text{is-Value-designator}(fdesig) \rightarrow \\ &\quad \text{let } mk\text{-Function-procedure-structure}(-, return) = \text{structure-of-a-value}(fdesig)\rho \text{ in} \\ &\quad \text{return}, \\ &\text{is-Standard-function}(fdesig) \rightarrow \\ &\quad \text{let } mk\text{-Standard-function}(-, desig) = fdesig \text{ in} \\ 235 \quad &\quad t\text{-standard-function-designator}(desig)(args)\rho, \\ &\text{is-System-function}(fdesig) \rightarrow \\ &\quad \text{let } mk\text{-System-function}(-, desig) = fdesig \text{ in} \\ 321 \quad &\quad t\text{-system-function-designator}(desig)(args)\rho, \\ &\text{is-Coroutine-function}(fdesig) \rightarrow \\ &\quad \text{let } mk\text{-Coroutine-function}(-, desig) = fdesig \text{ in} \\ 344 \quad &\quad t\text{-coroutine-function-designator}(desig)(args)\rho) \end{aligned}$$

Dynamic Semantics

operations

$m\text{-function-designator} : \text{Function-designator} \rightarrow \text{Environment} \xrightarrow{o} \text{Value}$

$m\text{-function-designator}(fdesig, args)\rho \triangleq$

```
183  ( is-Value-designator(fdesig)  → m-value-designator(fdesig) ρ,
    is-Standard-function(fdesig) → let mk-Standard-function(-, desig) = fdesig in
                                   return desig,
    is-System-function(fdesig)   → let mk-System-function(-, desig) = fdesig in
                                   return desig,
    is-Coroutine-function(fdesig) → let mk-Coroutine-function(-, desig) = fdesig in
                                   return desig)
```

6.7.5 Value Constructors

A value constructor gives a value of an array, record, or set type by specifying a list of values to be given to the elements of such an entity. The syntax requires a type identifier, and so value constructors cannot be used to specify values of variables of anonymous type. However, untyped value constructors can be used as the elements of a value constructor.

Concrete Syntax

value constructor = array constructor | record constructor | set constructor ;

CHANGE — Array and record constructors are not defined in *Programming in Modula-2*.

Abstract Syntax

types

$\text{Value-constructor} = \text{Array-constructor} \mid \text{Record-constructor} \mid \text{Set-constructor}$

Static Semantics

functions

$wf\text{-value-constructor} : \text{Value-constructor} \rightarrow \text{Environment} \rightarrow \mathbb{B}$

$wf\text{-value-constructor}(vc)\rho \triangleq$

cases vc :

```
192  mk-Array-constructor(-, -) → wf-array-constructor(vc)ρ,
196  mk-Record-constructor(-, -) → wf-record-constructor(vc)ρ,
202  mk-Set-constructor(-, -)   → wf-set-constructor(vc)ρ
end;
```

$t\text{-value-constructor} : \text{Value-constructor} \rightarrow \text{Environment} \rightarrow \text{Expression-typed}$

$t\text{-value-constructor}(vc)\rho \triangleq$

cases vc :

```
192  mk-Array-constructor(-, -) → t-array-constructor(vc)ρ,
196  mk-Record-constructor(-, -) → t-record-constructor(vc)ρ,
202  mk-Set-constructor(-, -)   → t-set-constructor(vc)ρ
end
```


Dynamic Semantics

operations

$m\text{-value-creator} : \text{Value-creator} \rightarrow \text{Environment} \xrightarrow{o} \text{Value}$
 $m\text{-value-creator}(vc)\rho \triangleq$
cases vc :
192 $mk\text{-Array-creator}(-, -) \rightarrow m\text{-array-creator}(vc)\rho,$
197 $mk\text{-Record-creator}(-, -) \rightarrow m\text{-record-creator}(vc)\rho,$
202 $mk\text{-Set-creator}(-, -) \rightarrow m\text{-set-creator}(vc)\rho$
end

6.7.5.1 Array Constructors

An array constructor gives a value of an array type by specifying a value for each element of the array.

NOTE — An array constructor does not give a value for an open array type.

Concrete Syntax

array constructor = type identifier, array constructed value ;

Abstract Syntax

types

$\text{Array-creator} :: qid : \text{Qualident}$
 $\text{def} : \text{Array-definition}$

Static Semantics

The qualified identifier shall denote an array type. The type of an array constructor shall be the array type denoted by the qualified identifier.

functions

$wf\text{-array-creator} : \text{Array-creator} \rightarrow \text{Environment} \rightarrow \mathbb{B}$
 $wf\text{-array-creator}(mk\text{-Array-creator}(qid, def))\rho \triangleq$
?? $wf\text{-qualident}(qid)\rho \wedge$
let $type = \text{type-of}(qid)\rho$ in
193 $wf\text{-array-definition}(type, def)\rho;$
 $t\text{-array-creator} : \text{Array-creator} \rightarrow \text{Environment} \rightarrow \text{Expression-typed}$
 $t\text{-array-creator}(mk\text{-Array-creator}(qid, -))\rho \triangleq$
271 $\text{type-of}(qid)\rho$

Dynamic Semantics

The ordered sequence of component values shall denote the component values of the array constructor in increasing order of the index value.

operations

$m\text{-array-creator} : \text{Array-creator} \rightarrow \text{Environment} \xrightarrow{o} \text{Array-value}$
 $m\text{-array-creator}(mk\text{-Array-creator}(qid, def))\rho \triangleq$
271 let $type = \text{type-of}(qid)\rho$ in
193 $evaluate\text{-array-definition}(type, def)\rho$

NOTE — The components of an array definition may be evaluated in any order.

Array Definitions

An array definition gives a value of a one-dimensional array type.

Concrete Syntax

array constructed value = left brace, repeated component, { ",", repeated component }, right brace ;

Abstract Syntax

types

Array-definition :: *vals* : *Repeated-elements*;

Repeated-elements = *Repeated-element**

Static Semantics

Each repeated element shall represent one or more occurrences of a value. The total number of values represented by the element sequence shall be equal to the number of values in the array.

functions

wf-array-definition : *Typed* × *Array-definition* → *Environment* → \mathbb{B}

wf-array-definition (*type*, *def*) $\rho \triangleq$

288 *is-array-type* (*type*) $\rho \wedge$ *size-of-array-definition* (*def*) $\rho =$ *size-of-array* (*type*) $\rho \wedge$

let *ctype* = *component-type-of* (*type*) ρ in

let *mk-Array-definition* (*vals*) = *def* in

195 $\forall val \in \text{elems } vals \cdot \text{wf-repeated-element}(ctype, val)\rho$

annotations Check that the number of components of the array definition is the same as the number of elements defined by the array type.

Dynamic Semantics

The ordered sequence of values produced by the element sequence shall denote the element values of the array definition in increasing order of the index value. It shall be an exception if any element is not a value of the component type.

operations

evaluate-array-definition : *Typed* × *Array-definition* → *Environment* \xrightarrow{o} *Value*

evaluate-array-definition (*type*, *def*) $\rho \triangleq$

let *mk-Array-definition* (*vals*) = *def* in

284 let *ctype* = *component-type-of* (*type*) ρ in

?? let *evals* = *evaluate-repeated-elements* (*ctype*, *vals*) ρ in

194 return *construct-array-value* (*type*, *evals*) ρ

Auxiliary Functions

functions

size-of-array-definition : *Array-definition* → *Environment* → \mathbb{N}

size-of-array-definition (*def*) $\rho \triangleq$

let *mk-Array-definition* (*vals*) = *def* in

number-of-repetitions (*vals*) ρ

annotations Return the number of components in an array definition.

functions

$number-of-repetitions : Repeated-elements \rightarrow Environment \rightarrow \mathbb{N}$
 $number-of-repetitions(vals)\rho \triangleq$
 if $vals = []$
 then 0
 else let $mk-Repeated-element(-, by) = hd\ vals$ in
 198 $evaluate-constant-expression(by)\rho + number-of-repetitions(tl\ vals)\rho$

annotations Return the number of elements in an array definition after expanding any repetitions.

functions

$construct-array-value : Expression-typed \times Value^* \rightarrow Environment \rightarrow Array-value$
 $construct-array-value(type, vals)\rho \triangleq$
 284 let $itype = index-type-of(type)\rho$ in
 291 let $indx = ordered-values-of(itype)\rho$ in
 $\{indx(i) \mapsto vals(i) \mid i \in inds\ indx\}$

annotations Construct an array value from an evaluated array definition.

operations

$evaluate-repeated-elements : Typed \times Repeated-elements \rightarrow Environment \xrightarrow{o} Value^*$
 $evaluate-repeated-elements(type, vals)\rho \triangleq$
 if $vals = []$
 then []
 else let $i \in inds\ vals$ in
 194 def $fvals = evaluate-repeated-elements(type, front(vals, i - 1))\rho$;
 195 def $mvals = evaluate-repeated-element(type, vals(i))\rho$;
 194 def $bvals = evaluate-repeated-elements(type, rest(vals, i + 1))\rho$;
 return $fvals \curvearrowright mvals \curvearrowright bvals$

annotations Evaluate the components of a array definition.

Repeated Elements

Concrete Syntax

repeated component = component of structure, ["BY", repetition factor] ;

repetition factor = constant expression ;

Abstract Syntax

types

$Repeated-element :: val : Element$
 $by : Expression$

annotations If the repetition factor is not present, the translation process will supply a repetition factor of 1.

Static Semantics

The absence of a repetition factor shall be equivalent to a repetition factor whose value is 1. If a repetition factor is present, it shall be a constant expression of whole number type and shall have a value which is not less than zero.

Each element shall be an expression, an array definition, a record definition, or a set definition. If it is an expression, the type of the expression shall be assignment compatible with the component type of the type of the array definition.

functions

```
wf-repeated-element : Typed × Repeated-element → Environment →  $\mathbb{B}$ 
wf-repeated-element (type, rval)  $\triangleq$ 
  let mk-Repeated-element(val, by) = rval in
196   wf-element (type, val)  $\rho$  ∧
152   wf-expression (by)  $\rho$  ∧
214   is-constant-expression (by)  $\rho$  ∧
282   is-whole-number-type (t-expression (by)  $\rho$ )  $\rho$  ∧
218   evaluate-constant-expression (by)  $\rho \geq 0$ 
```

Dynamic Semantics

The element shall be evaluated to give a value. The value of a repeated element shall be a sequence of N occurrences of that value, where N is the value of the repetition factor.

operations

```
evaluate-repeated-element : Typed × Repeated-element → Environment  $\xrightarrow{o}$  Value*
evaluate-repeated-element (type, rval)  $\rho \triangleq$ 
  let mk-Repeated-element(val, by) = rval in
196   def eval = evaluate-element(type, val)  $\rho$ ;
153   def num = m-expression(by)  $\rho$ ;
  return [eval | i ∈ {1, ..., num}]
```

Elements

Concrete Syntax

component of structure = expression | array constructed value | record constructed value | set constructed value ;

Abstract Syntax

types

Element = *Expression* | *Array-definition* | *Record-definition* | *Set-definition*

Static Semantics

The element shall be an expression, an array definition, a record definition, or a set definition.

functions

$$\begin{aligned}
 & wf_element : Typed \times Element \rightarrow Environment \rightarrow \mathbb{B} \\
 & wf_element (type, def) \rho \triangleq \\
 152 \quad & (is_Expression(def) \rightarrow wf_expression(def) \rho \wedge \\
 162 \quad & \quad \quad is_assignment_compatible (type, t_expression (def) \rho) \rho, \\
 193 \quad & is_Array_definition(def) \rightarrow wf_array_definition (type, def) \rho, \\
 197 \quad & is_Record_definition(def) \rightarrow wf_record_definition (type, def) \rho, \\
 203 \quad & is_Set_definition(def) \rightarrow wf_set_definition (def) \rho \wedge \\
 203 \quad & \quad \quad wf_type_and_set_definition (type, def) \rho)
 \end{aligned}$$

Dynamic Semantics

operations

$$\begin{aligned}
 & evaluate_element : Typed \times Element \rightarrow Environment \xrightarrow{o} Value \\
 & evaluate_element (type, val) \rho \triangleq \\
 153 \quad & (is_Expression(val) \rightarrow m_expression(val) \rho, \\
 193 \quad & is_Array_definition(val) \rightarrow evaluate_array_definition (type, val) \rho, \\
 198 \quad & is_Record_definition(val) \rightarrow evaluate_record_definition (type, val) \rho, \\
 203 \quad & is_Set_definition(val) \rightarrow \text{def } res = m_set_definition (val) \rho; \\
 204 \quad & \quad \quad evaluate_set_definition (type, res) \rho)
 \end{aligned}$$

6.7.5.2 Record Constructors

A record constructor gives a value of a record type.

Concrete Syntax

record constructor = type identifier, record constructed value ;

Abstract Syntax

types

$$\begin{aligned}
 & Record_constructor :: qid : Qualident \\
 & \quad \quad \quad def : Record_definition
 \end{aligned}$$

Static Semantics

The qualified identifier shall denote a record type. The type of a record constructor shall be the record type denoted by the qualified identifier.

functions

$$\begin{aligned}
 & wf_record_constructor : Record_constructor \rightarrow Environment \rightarrow \mathbb{B} \\
 & wf_record_constructor (mk_Record_constructor (qid, def)) \rho \triangleq \\
 ?? \quad & wf_qualident (qid) \rho \wedge \\
 271 \quad & \text{let } type = type_of (qid) \rho \text{ in} \\
 197 \quad & wf_record_definition (type, def) \rho; \\
 & t_record_constructor : Record_constructor \rightarrow Environment \rightarrow Expression_typed \\
 & t_record_constructor (mk_Record_constructor (qid, -)) \rho \triangleq \\
 271 \quad & type_of (qid) \rho
 \end{aligned}$$

Dynamic Semantics

The ordered sequence of component values shall denote the corresponding component values of the record constructor. It shall be an exception if any element is not a value of the corresponding component type for the selected variant (if present).

operations

$m\text{-record-creator} : \text{Record-creator} \rightarrow \text{Environment} \xrightarrow{o} \text{Record-value}$
 $m\text{-record-creator}(mk\text{-Record-creator}(qid, def))\rho \triangleq$
271 $\text{let } type = \text{type-of}(qid)\rho \text{ in}$
198 $\text{evaluate-record-definition}(type, def) \rho$

Record Definitions

A record definition gives a value of a record type.

Concrete Syntax

record constructed value = left brace, [repeated component, { ",", repeated component }], right brace ;

Each element shall specify a value of the component type of the record type or the value of the tag corresponding to a tagless variant.

Abstract Syntax

types

$\text{Record-definition} :: \text{vals} : \text{Elements};$
 $\text{Elements} = \text{Element}^*$

Static Semantics

The elements of the record definition shall be specified in an order corresponding to the order of the fields defined in the declaration of the type of the record. Where an element of the record definition corresponds to a tag field, the value of that element shall be denoted by a constant expression whose value corresponds to the chosen variant. Each element shall either be an expression, an array definition, a record definition, or a set definition. If it is an expression, the type of the expression shall be assignment compatible with the corresponding component of the chosen variant of the type of the record.

functions

$wf\text{-record-definition} : \text{Typed} \times \text{Record-definition} \rightarrow \text{Environment} \rightarrow \mathbb{B}$
 $wf\text{-record-definition}(type, def)\rho \triangleq$
298 $is\text{-record-type}(type)\rho \wedge size\text{-of-record-definition}(def)\rho = size\text{-of-record}(type, def)\rho \wedge$
271 $\text{let } mk\text{-Record-definition}(vals) = def \text{ in}$
199 $\text{let } mk\text{-Record-structure}(struc) = structure\text{-of}(type)\rho \text{ in}$
199 $match\text{-fields-list}(struc, vals)\rho$

Dynamic Semantics

The ordered sequence of component values shall denote the corresponding values of the record constructor. It shall be an exception if any element is not a value of the corresponding component type for the selected variant (if present).

operations

$evaluate_record_definition : Typed \times Record_definition \rightarrow Environment \xrightarrow{o} Record_value$
 $evaluate_record_definition (type, def) \rho \text{let } mk_Record_definition(vals) = def \text{ in}$
 $\text{let } mk_Record_constructor(struc) = structure_of(type) \rho \text{ in}$
 $construct_fields_list_value(struc, vals) \rho$

NOTE — The components of a record definition may be evaluated in any order.

Auxiliary Definitions

functions

$size_of_record_definition : Record_definition \rightarrow Environment \rightarrow \mathbb{N}$
 $size_of_record_definition (def) \rho \triangleq$
 $\text{let } mk_Record_definition(vals) = def \text{ in}$
 $\text{len } vals;$

 $size_of_record : Typed \times Record_definition \rightarrow Environment \rightarrow \mathbb{N}$
 $size_of_record (type, def) \rho \triangleq$
 $\text{let } mk_Record_structure(struc) = structure_of(type) \rho \text{ in}$
 $\text{let } mk_Record_definition(vals) = def \text{ in}$
 $size_of_fields_list(struc, vals) \rho;$

 $size_of_fields_list : Fields_list_structure \times Elements \rightarrow Environment \rightarrow \mathbb{N}$
 $size_of_fields_list (fsl, vals) \rho \triangleq$
 $\text{if } fsl = [] \vee vals = []$
 $\text{then } 0$
 $\text{else let } size = size_of_fields(\text{hd } fsl, vals) \rho \text{ in}$
 $\text{let } rem = rest(vals, size + 1) \text{ in}$
 $size + size_of_fields_list(\text{tl } fsl, rem) \rho$

annotations Calculate the number of components in a fields-list.

functions

$size_of_fields : Fields_structure \times Elements \rightarrow Environment \rightarrow \mathbb{N}$
 $size_of_fields (fs, vals) \rho \triangleq$
 $(is_Fixed_fields_structure(fs) \rightarrow size_of_fixed_fields(fs),$
 $is_Variant_fields_structure(fs) \rightarrow size_of_variant_fields(fs, vals) \rho)$

annotations Calculate the number of components in a fields.

functions

$size_of_fixed_fields : Fixed_fields_structure \rightarrow \mathbb{N}$
 $size_of_fixed_fields (ffs) \triangleq$
 $\text{let } mk_Fixed_fields_structure(ids, -) = ffs \text{ in}$
 $\text{len } ids$

annotations Calculate the number of components in a fixed-fields.

functions

$size\text{-}of\text{-}variant\text{-}fields : Variant\text{-}fields\text{-}structure \times Elements \rightarrow Environment \rightarrow \mathbb{N}$

$size\text{-}of\text{-}variant\text{-}fields(vstruc, vals)\rho \triangleq$
 let $mk\text{-}Variant\text{-}fields\text{-}structure(-, -, variants, other) = vstruc$ in
 218 let $tagval = evaluate\text{-}constant\text{-}expression(hd\ vals)\rho$ in
 if $\exists i \in inds\ variants \cdot tagval \in variants(i).labels$
 then let $i \in inds\ variants$ be st $tagval \in variants(i).labels$ in
 199 $size\text{-}of\text{-}variant(variants(i), tl\ vals)\rho + 1$
 elseif $other \neq nil$
 198 then $size\text{-}of\text{-}fields\text{-}list(other, tl\ vals)\rho + 1$
 else 0

annotations Calculate the number of components in a variant component of a record structure, selected by the value of the tag field.

functions

$size\text{-}of\text{-}variant : Variant\text{-}structure \times Elements \rightarrow Environment \rightarrow \mathbb{N}$

$size\text{-}of\text{-}variant(vstruc, vals)\rho \triangleq$
 let $mk\text{-}Variant\text{-}structure(-, fields) = vstruc$ in
 198 $size\text{-}of\text{-}fields\text{-}list(fields, vals)\rho$

annotations Calculate the number of components in a variant component of a record structure.

functions

$match\text{-}fields\text{-}list : Fields\text{-}list\text{-}structure \times Elements \rightarrow Environment \rightarrow \mathbb{B}$

$match\text{-}fields\text{-}list(fsl, vals)\rho \triangleq$
 if $fsl = []$
 then $vals = []$
 199 else $match\text{-}fields(hd\ fsl, vals) \wedge$
 198 let $size = size\text{-}of\text{-}fields(hd\ fsl, vals)$ in
 198 let $rvals = rest(vals, size + 1)$ in
 199 $match\text{-}fields\text{-}list(tl\ fsl, rvals)$

annotations Check that the structure defined by a fields-list matches the structure given by an expression sequence component of a structured record value.

functions

$match\text{-}fields : Fields\text{-}structure \times Elements \rightarrow Environment \rightarrow \mathbb{B}$

$match\text{-}fields(fs, vals)\rho \triangleq$
 199 $(is\text{-}Fixed\text{-}fields\text{-}structure(fs) \rightarrow match\text{-}fixed\text{-}fields(fs, vals)\rho,$
 200 $is\text{-}Variant\text{-}fields\text{-}structure(fs) \rightarrow match\text{-}variant\text{-}fields(fs, vals)\rho)$

annotations Check that the structure defined by a fields matches the structure given by an expression sequence component of a structured record value.

functions

$match\text{-}fixed\text{-}fields : Fixed\text{-}fields\text{-}structure \times Elements \rightarrow Environment \rightarrow \mathbb{B}$

$match\text{-}fixed\text{-}fields(ffs, vals)\rho \triangleq$
 let $mk\text{-}Fixed\text{-}fields\text{-}structure(ids, type) = ffs$ in
 196 $\forall i \in inds\ ids \cdot wf\text{-}element(type, vals(i))\rho$

annotations Check that the type of each expression in the record constructor is assignment compatible with the type of the corresponding component of a record type.

functions

$match\text{-}variant\text{-}fields : Variant\text{-}fields\text{-}structure \times Elements \rightarrow Environment \rightarrow \mathbb{B}$

```

match-variant-fields (vfs, vals) ρ  $\triangleq$ 
  let mk-Variant-fields-structure(-, tagt, variants, other) = vfs in
214   is-constant-expression (hd vals) ρ ∧
196   wf-element (tagt, hd vals) ρ ∧
  let tagv = evaluate-constant-expression(hd vals) ρ in
  if  $\exists i \in \text{inds } variants \cdot tagv \in variants(i).labels$ 
  then let  $i \in \text{inds } variants$  be st  $tagv \in variants(i).labels$  in
200     match-variant (variants(i), tl vals) ρ
  elseif other  $\neq$  nil
199   then match-fields-list (other, tl vals) ρ
  else undefined

```

annotations

Check that the type of each expression in the record constructor is assignment compatible with the type of the corresponding component of the record type; also check that the expression that corresponds to any tag field is a constant expression. The result is undefined if the structured value does not match the type.

functions

$match\text{-}variant : Variant\text{-}structure \times Elements \rightarrow Environment \rightarrow \mathbb{B}$

```

match-variant (variant, vals) ρ  $\triangleq$ 
  let mk-Variant-structure(-, fields) = variant in
  match-fields-list(fields, vals) ρ

```

annotations

Check the expressions in the record constructor that correspond to a variant component.

operations

$construct\text{-}fields\text{-}list\text{-}value : Fields\text{-}list\text{-}structure \times Elements \rightarrow Environment \xrightarrow{o} Record\text{-}value$

```

construct-fields-list-value (fsl, vals) ρ  $\triangleq$ 
  if fsl = []
  then return { }
200   else def rval = construct-fields-value(hd fsl, vals) ρ;
        let size = card dom rval in
std     let rem = rest(vals, size + 1) in
200     def lval = construct-fields-list-value(tl fsl, rem) ρ;
        return rval  $\cup$  lval

```

annotations

Build a record value corresponding to a fields-list component of a record from an expression sequence.

;

$construct\text{-}fields\text{-}value : Fields\text{-}structure \times Elements \rightarrow Environment \xrightarrow{o} Record\text{-}value$

```

construct-fields-value ((fs, vals)) ρ  $\triangleq$ 
200   ( is-Fixed-fields-structure(fs)  $\rightarrow$  construct-fixed-fields-value(fs, vals) ρ,
201     is-Variant-fields-structure(fs)  $\rightarrow$  construct-variant-fields-value(fs, vals) ρ)

```

annotations

Build a record value corresponding to a fields component of a record from an expression sequence.

;

$construct\text{-}fixed\text{-}fields\text{-}value : Fixed\text{-}fields\text{-}structure \times Expression^* \rightarrow Environment \xrightarrow{o} Record\text{-}value$

```

construct-fixed-fields-value (fs, vals) ρ  $\triangleq$ 
  let mk-Fixed-fields-structure(ids, type) = fs in
  (dcl rv : Value  $\xrightarrow{m}$  Value := { } ;

```

```

    for all  $i \in \text{inds } ids$ 
196   do def  $ival = \text{evaluate-element}(type, vals(i)) \rho$ ;
       $rv := rv \uparrow \{ids(i) \mapsto ival\}$ 

  annotations      Build a record value corresponding to a fixed-fields component of a record from an expression
                    sequence.

;

 $\text{construct-variant-fields-value} : \text{Variant-fields-structure} \times \text{Elements} \rightarrow \text{Environment} \xrightarrow{o} \text{Record-value}$ 
 $\text{construct-variant-fields-value}(vfs, vals) \rho \triangleq$ 
  let  $mk\text{-Variant-fields-structure}(id, -, variants, other) = vfs$  in
153  def  $tagval = m\text{-expression}(\text{hd } vals) \rho$ ;
  let  $tag = \text{if } id \neq \text{nil}$ 
      then  $\{id \mapsto tagval\}$ 
      else  $\{\}$  in
  if  $\exists i \in \text{inds } variants \cdot tagval \in variants(i).labels$ 
  then let  $i \in \text{inds } variants$  be st  $tagval \in variants(i).labels$  in
201    def  $vval = \text{construct-variant-value}(variants(i), \text{tl } vals) \rho$ ;
    return  $vval \cup tag$ 
  ?? else def  $fval = \text{construct-fields-list-value}(other, \text{tl } vals) \rho$ ;
    return  $fval \cup tag$ 

  annotations      Build a record value corresponding to a variant-fields component of a record from an expression
                    sequence.

;

 $\text{construct-variant-value} : \text{Variant-structure} \times \text{Elements} \rightarrow \text{Environment} \xrightarrow{o} \text{Record-value}$ 
 $\text{construct-variant-value}(vs, vals) \rho \text{let } mk\text{-Variant-structure}(-, fields) = vs \text{ in}$ 
200  $\text{construct-fields-list-value}(fields, vals) \rho$ 

  annotations      Build a record value corresponding to a variant component of a record from an expression
                    sequence.

```

6.7.5.3 Set Constructors

A set constructor gives the values of the members of a set by specifying the individual elements or ranges of values which define elements.

Concrete Syntax

set constructor = type identifier, set constructed value ;

NOTE — The set type of a set constructor is given explicitly.

CHANGE — There is no default type (of BITSET) cf *Programming in Modula-2*.

Abstract Syntax

types

$\text{Set-constructor} :: qid : \text{Qualident}$
 $\text{def} : \text{Set-definition}$

Static Semantics

The qualified identifier shall denote a set type. The type of a set constructor shall be the set type denoted by the qualified identifier. The type of the elements shall be assignment compatible with the base type of the set type.

NOTES

- 1 The construct '{ }' preceded by a set type name denotes the empty set of the specified set type. An empty set is a valid value of any set type.
- 2 Since the base type of a set type contains at least one value, there are at least two values of a set type.

functions

```
wf-set-constructor : Set-constructor → Environment →  $\mathbb{B}$ 
wf-set-constructor (mk-Set-constructor(qid, def)) $\rho \triangleq$ 
77   wf-qualident (qid) $\rho \wedge$ 
203   wf-set-definition (def) $\rho \wedge$ 
271   let type = type-of (qid) $\rho$  in
203   wf-type-and-set-definition (type, def) $\rho$ ;

t-set-constructor : Set-constructor → Environment → Expression-typed
t-set-constructor (mk-Set-constructor(qid, -)) $\rho \triangleq$ 
271   type-of (qid) $\rho$ 
```

CHANGE — The types of the elements of a set constructor are assignment compatible with the base type of the type identifier of the set constructor.

Dynamic Semantics

The value of a set constructor shall be the value of the set definition. It shall be an exception if a member of the set definition has a value which is not a value of the base type of the set type.

operations

```
m-set-constructor : Set-constructor → Environment  $\xrightarrow{o}$  Set-value
m-set-constructor (mk-Set-constructor(qid, def)) $\rho \triangleq$ 
283   let type = base-type-of (qid) $\rho$  in
203   def res = m-set-definition (def)  $\rho$ ;
204   evaluate-set-definition (type, res)  $\rho$ 
```

Set Definitions

A set definition denotes a set of values.

Concrete Syntax

set constructed value = left brace, [member, { " , member }], right brace ;

Abstract Syntax

types

Set-definition = *Member-set*

Static Semantics

The type of the set members shall be identical and this type shall be assignment compatible with the base type of the type of the set.

NOTES

- 1 The construct ‘{ }’ denotes the empty set. An empty set is a valid value of any set type.
- 2 Since the base type of a set type contains at least one value, there are at least two values of a set type.

functions

$wf\text{-}set\text{-}definition : Set\text{-}definition \rightarrow Environment \rightarrow \mathbb{B}$

$wf\text{-}set\text{-}definition(def)\rho \triangleq$

204 $\forall member \in def \cdot wf\text{-}member(member)\rho \wedge$

204 $\forall mem_1, mem_2 \in def \cdot t\text{-}member(mem_1)\rho = t\text{-}member(mem_2)\rho$

Language Clarification

The type of the elements of a set definition must be an ordinal type which is assignment compatible with the base type of the type of the set.

functions

$t\text{-}set\text{-}definition : Set\text{-}definition \rightarrow Environment \rightarrow Expression\text{-}typed$

$t\text{-}set\text{-}definition(def)\rho \triangleq$

let $mem \in def$ in

204 $t\text{-}member(mem)\rho$

annotations The type of a *Set-definition* is the type of its members, not a set type.

Dynamic Semantics

The value of a set definition shall be the union of the values of the members of the set definition. It shall be an exception if a member of the set definition has a value which is not a value of the base type of the type of the set.

operations

$m\text{-}set\text{-}definition : Set\text{-}definition \rightarrow Environment \xrightarrow{o} Value\text{-}set$

$m\text{-}set\text{-}definition(def)\rho \triangleq$

(dcl $setv : Value\text{-}set := \{ \}$;

for all $member \in def$

204 do def $val = m\text{-}member(member)\rho$;

$setv := setv \cup \{val\}$

return $\bigcup setv$)

Auxiliary Functions

functions

$wf\text{-}type\text{-}and\text{-}set\text{-}definition : Typed \times Set\text{-}definition \rightarrow Environment \rightarrow \mathbb{B}$

$wf\text{-}type\text{-}and\text{-}set\text{-}definition(type, def)\rho \triangleq$

$is\text{-}set\text{-}type(type)\rho \wedge$

let $btype = base\text{-}type\text{-}of(type)\rho$ in

203 $is\text{-}assignment\text{-}compatible(btype, t\text{-}set\text{-}definition(def)\rho)\rho \wedge$

operations

```
evaluate-set-definition : Typed × Value-set → Environment  $\xrightarrow{o}$  Value-set  
evaluate-set-definition (type, vals) $\rho \triangleq$   
290   let values = values-of (type) $\rho$  in  
      if vals  $\subseteq$  values  
      then return mk-Set-value(vals)  
306   else mandatory-exception(SET-RANGE)
```

Members

Constituent members of a set are denoted by a single value or a range of values.

Concrete Syntax

member = interval | singleton ;

Abstract Syntax

types

Member = *Interval* | *Singleton*

Static Semantics

functions

```
wf-member : Member → Environment →  $\mathbb{B}$   
wf-member (mem) $\rho \triangleq$   
205   (is-Interval(mem) → wf-interval(mem) $\rho$ ,  
206   is-Singleton(mem) → wf-singleton(mem) $\rho$ );  
  
t-member : Member → Environment → Expression-typed  
t-member (mem) $\rho \triangleq$   
205   (is-Interval(mem) → t-interval(mem) $\rho$ ,  
206   is-Singleton(mem) → t-singleton(mem) $\rho$ )
```

Dynamic Semantics

operations

```
m-member : Member → Environment  $\xrightarrow{o}$  Value-set  
m-member (mem) $\rho \triangleq$   
205   ( is-Interval(mem) → m-interval(mem)  $\rho$ ,  
206   is-Singleton(mem) → m-singleton(mem)  $\rho$ )
```

Interval

Constituent members of a set may be denoted by a range of values.

Concrete Syntax

interval = ordinal expression, "...", ordinal expression ;

Abstract Syntax

types

$$\begin{aligned} \text{Interval} &:: \text{min} : \text{Expression} \\ &\quad \text{max} : \text{Expression} \end{aligned}$$

Static Semantics

Both expressions of an interval shall be of identical type which shall be an ordinal type.

The type of an interval shall be the type possessed by each expression of the interval.

functions

$$\begin{aligned} &wf\text{-interval} : \text{Interval} \rightarrow \text{Environment} \rightarrow \mathbb{B} \\ &wf\text{-interval}(mk\text{-Interval}(\text{min}, \text{max}))\rho \triangleq \\ 152 \quad &wf\text{-expression}(\text{min})\rho \wedge \\ 152 \quad &wf\text{-expression}(\text{max})\rho \wedge \\ 152 \quad &t\text{-expression}(\text{min})\rho = t\text{-expression}(\text{max})\rho \wedge \\ 152 \quad &\text{let } type = t\text{-expression}(\text{min})\rho \text{ in} \\ 280 \quad &is\text{-ordinal-type}(type)\rho; \\ &t\text{-interval} : \text{Interval} \rightarrow \text{Environment} \rightarrow \text{Expression-typed} \\ &t\text{-interval}(mk\text{-Interval}(\text{min}, \text{max}))\rho \triangleq \\ 152 \quad &t\text{-expression}(\text{min})\rho \end{aligned}$$

Dynamic Semantics

An interval of the form ‘ $\mathbf{x}.. \mathbf{y}$ ’, where \mathbf{x} and \mathbf{y} are expressions, shall denote no members if the value of \mathbf{x} is greater than the value of \mathbf{y} ; otherwise, it shall denote one or more members that have the values in the closed interval from the value of \mathbf{x} to the value of \mathbf{y} .

operations

$$\begin{aligned} &m\text{-interval} : \text{Interval} \rightarrow \text{Environment} \xrightarrow{o} \text{Value-set} \\ &m\text{-interval}(mk\text{-Interval}(\text{min}, \text{max}))\rho \triangleq \\ 153 \quad &\text{def } vmin = m\text{-expression}(\text{min})\rho; \\ 153 \quad &\text{def } vmax = m\text{-expression}(\text{max})\rho; \\ &\text{return } \{x \mid x \in \{vmin, \dots, vmax\}\} \end{aligned}$$

annotations The left to right evaluation given above is one of the orders permitted. The two expressions may be evaluated in any order, including in parallel, see 4.1.

Singleton

A member of a set may be denoted by a single value.

Concrete Syntax

singleton = ordinal expression ;

Abstract Syntax

types

$$\text{Singleton} :: \text{expr} : \text{Expression}$$

Static Semantics

The expression in a singleton shall be of ordinal type.

The type of a singleton shall be the type of the expression.

functions

```
wf-singleton : Singleton → Environment →  $\mathbb{B}$ 
wf-singleton (mk-Singleton(expr)) $\rho \triangleq$ 
152   wf-expression (expr) $\rho \wedge$ 
152   let type = t-expression (expr) $\rho$  in
280   is-ordinal-type (type) $\rho$ ;

t-singleton : Singleton → Environment → Expression-typed
t-singleton (mk-Singleton(expr)) $\rho \triangleq$ 
152   t-expression (expr) $\rho$ 
```

Dynamic Semantics

A singleton shall denote the member that has the value of the expression.

operations

```
m-singleton : Singleton → Environment  $\xrightarrow{o}$  Value-set
m-singleton (mk-Singleton(expr)) $\rho \triangleq$ 
153   def value = m-expression(expr)  $\rho$ ;
      return { value }
```

6.7.6 Constant Literals

Constant literals are literals denoting constant values.

NOTE — There are no literals of the Complex or Long Complex type. But constant values of these types may be formed using the standard function **CMPLX**; for example ‘**CMPLX**(0.0,1.0)’, see section 6.9.3.5.

Concrete Syntax

constant literal = whole number literal | real literal | string literal | character literal ;

Abstract Syntax

types

Constant-literal = *Whole-number-literal* | *Real-literal* | *String-literal* | **NIL-VALUE**

annotations In the translation from the concrete to abstract syntax, character literals are represented as strings of length one.

As the identifier **NIL** may be redefined, it does not appear in the concrete syntax.

Static Semantics

functions

```
wf-constant-literal : Value-designator  $\rightarrow$  Environment  $\rightarrow$   $\mathbb{B}$   
wf-constant-literal(val) $\rho \triangleq$   
  cases val :  
    208 mk-Whole-number-literal(-)  $\rightarrow$  wf-whole-number-literal(val) $\rho$ ,  
    209 mk-Real-literal(-)  $\rightarrow$  wf-real-literal(val) $\rho$ ,  
    210 mk-String-literal(-)  $\rightarrow$  wf-string-literal(val) $\rho$ ,  
      NIL-VALUE  $\rightarrow$  true  
  end
```

functions

```
t-constant-literal : Value-designator  $\rightarrow$  Environment  $\rightarrow$  Typed  
t-constant-literal(val) $\rho \triangleq$   
  cases val :  
    208 mk-Whole-number-literal(-)  $\rightarrow$  t-whole-number-literal(val) $\rho$ ,  
    209 mk-Real-literal(-)  $\rightarrow$  t-real-literal(val) $\rho$ ,  
    210 mk-String-literal(-)  $\rightarrow$  t-string-literal(val) $\rho$ ,  
      NIL-VALUE  $\rightarrow$  nil -TYPE  
  end
```

Dynamic Semantics

operations

```
m-constant-literal : Value-designator  $\rightarrow$  Environment  $\xrightarrow{o}$  Value  
m-constant-literal(val) $\rho \triangleq$   
  cases val :  
    208 mk-Whole-number-literal(-)  $\rightarrow$  m-whole-number-literal(val)  $\rho$ ,  
    209 mk-Real-literal(-)  $\rightarrow$  m-real-literal(val)  $\rho$ ,  
    210 mk-String-literal(-)  $\rightarrow$  m-string-literal(val)  $\rho$ ,  
    ?? NIL-VALUE  $\rightarrow$  nil-value()  
  end
```

6.7.6.1 Whole Number Literals

Whole number literals can be denoted by means of a decimal, octal or hexadecimal notation.

Concrete Syntax

whole number literal = decimal number | octal number | hexadecimal number ;

decimal number = digit, { digit } ;

digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;

octal number = octal digit, { octal digit }, "B" ;

octal digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" ;

hexadecimal number = digit, { hex digit }, "H" ;

hex digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" | "A" | "B" | "C" | "D" | "E" | "F" ;

The radix of the first form of whole number literal shall be 10, the second form shall be 8 and the last form shall be 16. No separators shall appear within a whole number literal.

NOTE — The additional requirement of no separators must be made since this is not a consequence of the concrete syntax.

Abstract Syntax

types

Whole-number-literal :: *value* : \mathbb{N}

Static Semantics

There shall be an implementation-defined maximum value for literals of whole number types. This value shall be at least as large as the maximum value of any whole number type.

NOTE — The maximum value for whole number literal denotations has no denotation as an identifier.

functions

wf-whole-number-literal : *Whole-number-literal* \rightarrow *Environment* $\rightarrow \mathbb{B}$

wf-whole-number-literal (*mk-Whole-number-literal*(*value*)) $\rho \triangleq$
value $\in \{0, \dots, \text{maximum}(\mathbb{Z}\text{-TYPE})\};$

t-whole-number-literal : *Whole-number-literal* \rightarrow *Environment* \rightarrow *Expression-typed*

t-whole-number-literal (*mk-Whole-number-literal*(-)) $\rho \triangleq$
 $\mathbb{Z}\text{-TYPE}$

Dynamic Semantics

operations

m-whole-number-literal : *Whole-number-literal* \rightarrow *Environment* $\xrightarrow{o} \mathbb{Z}$

m-whole-number-literal (*mk-Whole-number-literal*(*value*)) $\rho \triangleq$
 return *value*

6.7.6.2 Real Literals

Real literal values are denoted by a conventional decimal notation.

Concrete Syntax

real literal = digit, { digit }, ".", { digit }, [scale factor] ;

scale factor = "E", ["+" | "-"], digit, { digit } ;

A real literal shall be written in decimal notation with an optional scale factor. The scale factor shall give the power of 10 to which the remaining part of the real literal is multiplied by to produce its value. No separators shall appear within a real literal.

NOTE — The additional requirement of no separators must be made since this is not a consequence of the concrete syntax. A real literal always contains a decimal point.

Abstract Syntax

types

Real-literal :: *value* : \mathbb{R}

Static Semantics

There shall be an implementation-defined maximum value for literals of real types. This value shall be at least as large as the maximum value of any real type.

NOTE — The maximum value for real literal denotations has no denotation as an identifier.

functions

wf-real-literal : *Real-literal* \rightarrow *Environment* $\rightarrow \mathbb{B}$

wf-real-literal(*mk-Real-literal*(*value*)) $\rho \triangleq$

$0 \leq \text{value} \wedge$

288 $\text{value} \leq \text{maximum}(\mathbb{R}\text{-TYPE});$

t-real-literal : *Real-literal* \rightarrow *Environment* \rightarrow *Expression-typed*

t-real-literal(*mk-Real-literal*(-)) $\rho \triangleq$

$\mathbb{R}\text{-TYPE}$

Dynamic Semantics

operations

m-real-literal : *Real-literal* \rightarrow *Environment* $\xrightarrow{o} \mathbb{R}$

m-real-literal(*mk-Real-literal*(*value*)) $\rho \triangleq$

?? $\text{return } \text{real-constant-approximation}(\text{value})$

6.7.6.3 String Literals

String literals denote a sequence of characters.

Concrete Syntax

quoted string = `""`, { national character - `""` }, `"""` | `'''`, { national character - `'''` }, `'''` ;

string concatenate symbol = `""` ;

NOTES

1 The string literal syntax is not an alternative to the structured value constructor because the compatibility rules for literals are more relaxed than those for concrete string constants defined using structured value constructors.

2 String constants cannot be indexed in a character expression.

CHANGE — Catenation of string literals and identifiers denoting string constants is allowed.

Abstract Syntax

types

String-literal = *Single-string**

Static Semantics

functions

```

$$wf\text{-}string\text{-}literal : String\text{-}literal \rightarrow Environment \rightarrow \mathbb{B}$$

$$wf\text{-}string\text{-}literal(cl)\rho \triangleq$$
211 
$$\forall ss \in \text{elems } cl \cdot wf\text{-}single\text{-}string(ss)\rho;$$

$$t\text{-}string\text{-}literal : String\text{-}literal \rightarrow Environment \rightarrow Expression\text{-}typed$$

$$t\text{-}string\text{-}literal(cl)\rho \triangleq$$
77 
$$mk\text{-}String\text{-}type(string\text{-}length(cl)\rho)$$

```

Dynamic Semantics

operations

```

$$m\text{-}string\text{-}literal : String\text{-}literal \rightarrow Environment \xrightarrow{o} \{\mathbb{N} \mapsto \text{char}\}$$

$$m\text{-}string\text{-}literal(cl)\rho$$

$$\triangleq$$
210 
$$\begin{aligned} &\text{def } str = \text{catenate}(cl)\rho; \\ &\text{return } make\text{-}string\text{-}value(str) \end{aligned}$$

```

Auxiliary Functions

operations

```

$$\text{catenate} : Single\text{-}string^* \rightarrow Environment \xrightarrow{o} \text{char}^*$$

$$\text{catenate}(ss)\rho \triangleq$$

$$\begin{aligned} &\text{if } ss = [] \\ &\text{then return } [] \\ \small{211} &\text{else def } s = m\text{-}single\text{-}string(\text{hd } ss)\rho; \\ \small{210} &\text{def } f = \text{catenate}(\text{tl } ss)\rho; \\ &\text{return } s \curvearrowright f; \end{aligned}$$

$$make\text{-}string\text{-}value : \text{char}^* \rightarrow (\mathbb{N} \xrightarrow{m} \text{char})$$

$$make\text{-}string\text{-}value(str) \triangleq$$

$$\begin{aligned} &\text{if } str = [] \\ &\text{then } \{0 \mapsto \text{END-OF-STRING-CHAR}\} \\ &\text{else } \{(i-1) \mapsto str(i) \mid i \in \text{inds } str\} \end{aligned}$$

```

Single Strings

A single string denotes a sequence of characters.

Concrete Syntax

single string = quoted string | string identifier ;

Abstract Syntax

types

$Single\text{-}string = Character\text{-}string \mid String\text{-}identifier$

Static Semantics

functions

$$wf\text{-}single\text{-}string : Single\text{-}string \rightarrow Environment \rightarrow \mathbb{B}$$

$$wf\text{-}single\text{-}string(ss)\rho \triangleq$$

$$211 \quad (is\text{-}Character\text{-}string(ss) \rightarrow wf\text{-}character\text{-}string(ss)\rho,$$

$$212 \quad is\text{-}String\text{-}identifier(ss) \rightarrow wf\text{-}string\text{-}identifier(ss)\rho)$$

Dynamic Semantics

operations

$$m\text{-}single\text{-}string : Single\text{-}string \rightarrow Environment \xrightarrow{o} Value$$

$$m\text{-}single\text{-}string(ss)\rho \triangleq$$

$$?? \quad (is\text{-}Character\text{-}string(ss) \rightarrow m\text{-}character\text{-}string(ss)\rho,$$

$$?? \quad is\text{-}String\text{-}identifier(ss) \rightarrow m\text{-}string\text{-}identifier(ss)\rho)$$

Character Strings

Character strings denote a sequence of characters.

Concrete Syntax

quoted string = `""`, { national character - `""` }, `""` | `'''`, { national character - `'''` }, `'''` ;

The sequence of characters in a string shall be denoted by themselves but preceded and succeeded by either a simple quote `'` or a double quote `''`

The value denoted by `''''` (the empty string) which is defined by an implementation, shall be the value which the standard string library for the implementation uses as a string terminator.

NOTES

- 1 One string literal cannot contain both a single and double quote characters (since one or the other will mark the end of the string) and cannot cross a line boundary.
- 2 The empty string is expression compatible with a value of a type denoted by the pervasive identifier `CHAR`.
- 3 The component type of a concrete string type must be identical to the type denoted by the pervasive identifier `CHAR`; it cannot be a subrange. This is because it must be possible to assign a value of `CHAR`, which has been defined by an implementation as its string terminating character, to a component of a concrete string variable.

CHANGE — The empty string is expression and assignment compatible with the type denoted by the pervasive identifier `CHAR`.

Abstract Syntax

types

$$Character\text{-}string :: val : Char^*$$

Static Semantics

functions

$$wf\text{-}character\text{-}string : Character\text{-}string \rightarrow Environment \rightarrow \mathbb{B}$$

$$wf\text{-}character\text{-}string(mk\text{-}Character\text{-}string(-))\rho \triangleq$$

true

Dynamic Semantics

functions

$$\begin{aligned}
& m\text{-character-string} : \text{Character-string} \rightarrow \text{Environment} \rightarrow \text{char}^* \\
& m\text{-character-string}(mk\text{-Character-string}(cl))\rho \triangleq \\
& \quad cl
\end{aligned}$$

String Identifiers

String identifiers denote a string constant.

Concrete Syntax

string identifier = qualified identifier ;

Abstract Syntax

types

$$String\text{-}identifier :: qid : Qualident$$

Static Semantics

functions

$$\begin{aligned}
& wf\text{-}string\text{-}identifier : String\text{-}identifier \rightarrow Environment \rightarrow \mathbb{B} \\
& wf\text{-}string\text{-}identifier(mk\text{-}String\text{-}identifier(qid))\rho \triangleq \\
27a \quad & is\text{-}constant(qid)\rho \wedge type\text{-}of(qid)\rho = S\text{-}type
\end{aligned}$$

Dynamic Semantics

functions

$$\begin{aligned}
& m\text{-}string\text{-}identifier : String\text{-}identifier \rightarrow Environment \rightarrow \text{char}^* \\
& m\text{-}string\text{-}identifier(mk\text{-}String\text{-}identifier(qid))\rho \triangleq \\
77 \quad & associated\text{-}constant\text{-}value(qid)\rho
\end{aligned}$$

6.7.6.4 Character Literals

A character literal denotes a value of type **CHAR**.

Concrete Syntax

quoted character = `""`, national character - `""`, `'''` | `'''`, national character - `'''`, `'''` ;

character number literal = octal digit, { octal digit }, `"C"` ;

A character literal shall denote a value of type **CHAR**. The character within either a single quote ‘`’`’ or a double quote ‘`”`’ shall be the value denoted. A sequence of octal digits followed by ‘`C`’ shall denote a literal of type **CHAR** whose ordinal value shall be given by the value of the octal digit sequence.

NOTE — A character literal may also be regarded as a string literal of length 1.

Language Clarification

Character number literals are of type SS-type.

Abstract Syntax

annotations	There is no abstract syntax, static semantics, or dynamic semantics for character literals as they are handled by the formal definition as strings of length one.
-------------	---

6.7.6.5 Pointer Literals

A pointer literal denotes a value of a pointer type. There is only one such literal, denoted by the pervasive identifier **NIL**, and this is expression compatible with the value of any user-defined pointer type.

6.7.7 Constant Expressions

The constituents of a constant expression are constants. A constant expression is thus one that can be completely evaluated by a mere textual scan without executing the program. Constant expressions are of particular use in constant declarations.

Concrete Syntax

constant expression = expression ;

Abstract Syntax

annotations	The abstract syntax for <i>constantexpression</i> is the same as for <i>expression</i> .
-------------	--

Static Semantics

An infix expression is constant if and only if its two operands are both constant expressions.

A prefix expression is constant if and only if its operand is a constant expression.

A constant expression shall not contain a reference to a variable, unless that variable is an actual parameter to a standard function that is allowed in constant expressions.

A function call may be contained in a constant expression if and only if the function is a standard function (except that **SIZE** of an open array value is not allowed) or the functions **CAST**, **ROTATE**, **SHIFT** and **TSIZE** from the module **SYSTEM**, and its parameters are constant expressions.

An array constructor is constant if and only if its elements are denoted by constant expressions.

A record constructor is constant if and only if its elements are denoted by constant expressions.

A set constructor is constant if and only if its elements are denoted by constant expressions.

It shall be an error for the value of a constant whole number expression to exceed the maximum value for whole number literals.

It shall be an error for the value of a constant real expression to exceed the maximum value for real literals.

The value of a constant expression shall be the same as that of an equivalent non-constant expression. If the evaluation of an expression would lead to an exception, then the evaluation of the same constant expression is an error.

The ordered sequence of component values shall denote the corresponding component values of the record constructor. It shall be an exception if any element is not a value of the corresponding component type for the selected variant (if any).

NOTE — The constant expression ‘TRUE OR (1/0=1)’ is legal and evaluates to TRUE since the right hand operand is not evaluated.

functions

is-constant-expression : *Expression* → *Environment* → \mathbb{B}

is-constant-expression (*expr*) $\rho \triangleq$
214 (*is-Infix-expression* (*expr*) → *is-constant-infix-expression* (*expr*) ρ ,
214 *is-Prefix-expression* (*expr*) → *is-constant-prefix-expression* (*expr*) ρ ,
214 *is-Value-designator* (*expr*) → *is-constant-value-designator* (*expr*) ρ ,
215 *is-Function-call* (*expr*) → *is-constant-function-call* (*expr*) ρ ,
216 *is-Value-constructor* (*expr*) → *is-constant-value-constructor* (*expr*) ρ ,
 is-Constant-literal (*expr*) → true)

annotations The function checks whether an expression is a constant expression.

A whole number is a constant expression.

A real number is a constant expression.

A string is a constant expression.

functions

is-constant-infix-expression : *Infix-expression* → *Environment* → \mathbb{B}

is-constant-infix-expression (*mk-Infix-expression* (*left*, -, *right*)) $\rho \triangleq$
214 *is-constant-expression* (*left*) $\rho \wedge$
214 *is-constant-expression* (*right*) ρ

annotations Check that both expression components are constant expressions.

functions

is-constant-prefix-expression : *Prefix-expression* → *Environment* → \mathbb{B}

is-constant-prefix-expression (*mk-Prefix-expression* (-, *expr*)) $\rho \triangleq$
214 *is-constant-expression* (*expr*) ρ

annotations Check that the expression component is a constant expression.

functions

is-constant-value-designator : *Value-designator* → *Environment* → \mathbb{B}

is-constant-value-designator (*vd*) $\rho \triangleq$
 cases *vd* :
215 *mk-Entire-value* (-) → *is-constant-entire-value* (*vd*) ρ ,
215 *mk-Indexed-value* (-, -) → *is-constant-indexed-value* (*vd*) ρ ,
215 *mk-Selected-value* (-, -) → *is-constant-selected-value* (*vd*) ρ ,
215 *mk-Dereferenced-value* (-) → *is-constant-dereferenced-value* (*vd*) ρ
 end

functions

$is_constant_entire_value : Entire_value \rightarrow Environment \rightarrow \mathbb{B}$

$is_constant_entire_value (mk_Entire_value(qid))\rho \triangleq$

270 $is_constant(qid)\rho \vee$

272 $is_proper_procedure(qid)\rho \vee$

273 $is_function_procedure(qid)\rho$

annotations Check that the qualified identifier denotes a constant or a proper procedure or a function procedure.

functions

$is_constant_indexed_value : Indexed_value \rightarrow Environment \rightarrow \mathbb{B}$

$is_constant_indexed_value (mk_Indexed_value(design, expr))\rho \triangleq$

214 $is_constant_value_designator(design)\rho \wedge$

214 $is_constant_expression(expr)\rho$

annotations Check that the designator denotes a constant and that the index expression is a constant expression.

functions

$is_constant_selected_value : Selected_value \rightarrow Environment \rightarrow \mathbb{B}$

$is_constant_selected_value (mk_Selected_value(design, -))\rho \triangleq$

214 $is_constant_value_designator(design)\rho$

annotations Check that the designator denotes a constant.

functions

$is_constant_dereferenced_value : Dereferenced_value \rightarrow Environment \rightarrow \mathbb{B}$

$is_constant_dereferenced_value (mk_Dereferenced_value(-))\rho \triangleq$

false

annotations A dereferenced value cannot be a constant expression.

functions

$is_constant_function_call : Function_call \rightarrow Environment \rightarrow \mathbb{B}$

$is_constant_function_call (mk_Function_call(fd, args))\rho \triangleq$

215 $is_constant_function_designator(fd) \wedge$

216 $\forall arg \in elems\ args \cdot is_constant_actual_parameter(arg)\rho;$

$is_constant_function_designator : Value_designator \rightarrow Environment \rightarrow \mathbb{B}$

$is_constant_function_designator (fd)\rho \triangleq$

215 $(is_Standard_function(fd) \rightarrow is_constant_standard_function(fd)\rho,$

216 $is_System_function(fd) \rightarrow is_constant_system_function(fd)\rho,$

others \rightarrow false);

$is_constant_standard_function : Standard_function \rightarrow Environment \rightarrow \mathbb{B}$
 $is_constant_standard_function (mk_Standard_function(-, vd))\rho \triangleq$
 $is_Abs_designator(vd) \vee$
 $is_Cap_designator(vd) \vee$
 $is_Chr_designator(vd) \vee$
 $is_Cmplx_designator(vd) \vee$
 $is_Float_designator(vd) \vee$
 $is_High_designator(vd) \vee$
 $is_Im_designator(vd) \vee$
 $is_Int_designator(vd) \vee$
 $is_Length_designator(vd) \vee$
 $is_Lfloat_designator(vd) \vee$
 $is_Max_designator(vd) \vee$
 $is_Min_designator(vd) \vee$
 $is_Odd_designator(vd) \vee$
 $is_Ord_designator(vd) \vee$
 $is_Re_designator(vd) \vee$
 $is_Size_designator(vd) \wedge \neg is_open_array(vd.type)\rho \vee$
 $is_Trunc_designator(vd) \vee$
 $is_Val_designator(vd);$

$is_constant_system_function : System_function \rightarrow Environment \rightarrow \mathbb{B}$
 $is_constant_system_function (mk_System_function(-, fd))\rho \triangleq$
 $is_Cast_designator(fd) \vee$
 $is_Rotate_designator(fd) \vee$
 $is_Shift_designator(fd) \vee$
 $is_Tsize_designator(fd);$

$is_constant_actual_parameter : Actual_parameter \rightarrow Environment \rightarrow \mathbb{B}$
 $is_constant_actual_parameter (ap)\rho \triangleq$
 $(is_Expression(ap) \rightarrow is_constant_expression(ap)\rho,$
 $is_Type_parameter(ap) \rightarrow true)$

annotations Check that the function call is to a standard function or a system function that is defined to be a constant function.

functions

$is_constant_value_constructor : Value_constructor \rightarrow Environment \rightarrow \mathbb{B}$
 $is_constant_value_constructor (vc)\rho \triangleq$
 $cases\ vc :$
 $mk_Array_constructor(-, -) \rightarrow is_constant_array_constructor(vc)\rho,$
 $mk_Record_constructor(-, -) \rightarrow is_constant_record_constructor(vc)\rho,$
 $mk_Set_constructor(-, -) \rightarrow is_constant_set_constructor(vc)\rho$
 end

functions

$is_constant_array_constructor : Array_constructor \rightarrow Environment \rightarrow \mathbb{B}$
 $is_constant_array_constructor (mk_Array_constructor(-, def))\rho \triangleq$
 $is_constant_array_definition(def)\rho;$
 $is_constant_array_definition : Array_definition \rightarrow Environment \rightarrow \mathbb{B}$
 $is_constant_array_definition (mk_Array_definition(vals))\rho \triangleq$
 $\forall val \in elems\ vals \cdot is_constant_repeated_element(val)\rho;$

$is_constant_repeated_element : Repeated_element \rightarrow Environment \rightarrow \mathbb{B}$
 $is_constant_repeated_element (mk_Repeated_element(val, by))\rho \triangleq$
 $is_constant_element (val)\rho;$

$is_constant_element : Element \rightarrow Environment \rightarrow \mathbb{B}$
 $is_constant_element (el)\rho \triangleq$
 $(is_Expression(el) \rightarrow is_constant_expression (el)\rho,$
 $is_Array_definition(el) \rightarrow is_constant_array_definition (el)\rho,$
 $is_Record_definition(el) \rightarrow is_constant_record_definition (el)\rho,$
 $is_Set_definition(el) \rightarrow is_constant_set_definition (el)\rho)$

annotations Check that the array definition is a constant expression.

Check that each of the repeated element components of an array definition is a constant expression.

Check that element and repetition factor are constant expressions.

functions

$is_constant_record_constructor : Record_constructor \rightarrow Environment \rightarrow \mathbb{B}$
 $is_constant_record_constructor (mk_Record_constructor(-, def))\rho \triangleq$
 $is_constant_record_definition (def)\rho;$

$is_constant_record_definition : Record_definition \rightarrow Environment \rightarrow \mathbb{B}$
 $is_constant_record_definition (mk_Record_definition(vals))\rho \triangleq$
 $\forall val \in elems\ vals \cdot is_constant_element (val)\rho$

annotations Check that a record constructor is a constant expression by checking that the record definition component is a constant expression. A record definition is a constant expression if each of the elements is a constant expression.

functions

$is_constant_set_constructor : Set_constructor \rightarrow Environment \rightarrow \mathbb{B}$
 $is_constant_set_constructor (mk_Set_constructor(-, def))\rho \triangleq$
 $is_constant_set_definition (def)\rho;$

$is_constant_set_definition : Set_definition \rightarrow Environment \rightarrow \mathbb{B}$
 $is_constant_set_definition (def)\rho \triangleq$
 $\forall member \in def \cdot is_constant_member (member)\rho;$

$is_constant_member : Member \rightarrow Environment \rightarrow \mathbb{B}$
 $is_constant_member (mem)\rho \triangleq$
 $(is_Singleton(mem) \rightarrow is_constant_singleton (mem)\rho,$
 $is_Interval(mem) \rightarrow is_constant_interval (mem)\rho);$

$is_constant_singleton : Singleton \rightarrow Environment \rightarrow \mathbb{B}$
 $is_constant_singleton (mk_Singleton(expr))\rho \triangleq$
 $is_constant_expression (expr)\rho;$

$is_constant_interval : Interval \rightarrow Environment \rightarrow \mathbb{B}$
 $is_constant_interval (mk_Interval(min, max))\rho \triangleq$
 $is_constant_expression (min)\rho \wedge$
 $is_constant_expression (max)\rho$

annotations	Check that the set definition is a constant expression by checking that each of the members of the set is a constant expression. A singleton member is constant if it is a constant expression, and an interval is constant if each of its component expressions are constant expressions.
-------------	--

6.7.7.1 Constant Expression Evaluation

functions

evaluate-constant-expression : *Expression* \rightarrow *Environment* \rightarrow *Value*

evaluate-constant-expression (*expr*) $\rho \triangleq$

291	let $\rho_{cont} = exit(\text{NOT-DEFINED})\text{nil } \rho$ in
153	let ($-, result$) = (<i>m-expression</i> (<i>expr</i>) (ρ_{cont}) σ_{empty}) in
	<i>result</i>

annotations	As a constant expression can be evaluated with just a textual scan, the evaluation of a constant expression is defined to be the meaning of an expression with an empty state.
-------------	--

Language Clarification

Constant expressions can be procedure values. Structured values of procedure values are also allowed.

6.8 Procedure Activation

The execution of a proper procedure or function procedure call causes the activation and execution of the block associated with the procedure. If the procedure was declared with formal parameters, the call will have an actual parameter list and these parameters are substituted in place of the corresponding formal parameters defined in the procedure declaration. The correspondence is established by the positions of the parameters in the lists of actual and formal parameters respectively.

6.8.1 Actual Parameters

Concrete Syntax

```
actual parameters = "(" [ actual parameter list ] ")" ;
actual parameter list = actual parameter { "," actual parameter } ;
actual parameter = variable designator | expression | type parameter ;
```

Abstract Syntax

$Actual\text{-}parameters = Actual\text{-}parameter^*$

$Actual\text{-}parameter = Variable\text{-}designator \mid Expression \mid Type\text{-}parameter$

$Arguments = Argument^*$

$Argument = Variable \mid Value \mid Typed$

Static Semantics

$wf\text{-}actual\text{-}parameters : Actual\text{-}parameters \rightarrow Environment \rightarrow \mathbb{B}$

$wf\text{-}actual\text{-}parameters(aps)\rho \triangleq$
219 $\forall ap \in \text{elems } aps \cdot wf\text{-}actual\text{-}parameter(ap)\rho$

$wf\text{-}actual\text{-}parameter : Actual\text{-}parameter \rightarrow Environment \rightarrow \mathbb{B}$

$wf\text{-}actual\text{-}parameter(ap)\rho \triangleq$
145 $(is\text{-}Variable\text{-}designator(ap) \rightarrow wf\text{-}variable\text{-}designator(ap)\rho,$
152 $is\text{-}Expression(ap) \rightarrow wf\text{-}expression(ap)\rho,$
220 $is\text{-}Type\text{-}parameter(ap) \rightarrow wf\text{-}type\text{-}parameter(ap)\rho)$

annotations Check each of the actual parameters.

Dynamic Semantics

$m\text{-}actual\text{-}parameters : Actual\text{-}parameters \rightarrow Environment \xrightarrow{o} Arguments$

$m\text{-}actual\text{-}parameters((aps)\rho) \triangleq$
 if $aps = []$
 then return $[]$
220 else def $mhd = m\text{-}actual\text{-}parameter(hd\text{ }aps)\rho,$
219 $mtl = m\text{-}actual\text{-}parameters(tl\text{ }aps)\rho;$
 return $[mhd] \curvearrowright mtl$

annotations Evaluate each of the actual parameters. The order of evaluation, accessing, and binding of the actual parameters is implementation-dependent.

$m\text{-actual-parameter} : \text{Actual-parameter} \rightarrow \text{Environment} \xrightarrow{o} \text{Argument}$

$m\text{-actual-parameter}(ap)\rho \triangleq$
145 $(\text{is-Variable-designator}(ap) \rightarrow m\text{-variable-designator}(ap)\rho,$
153 $\text{is-Expression}(ap) \rightarrow m\text{-expression}(ap)\rho,$
220 $\text{is-Type-parameter}(ap) \rightarrow m\text{-type-parameter}(ap)\rho)$

6.8.1.1 Type Parameters

Certain standard procedures can take types as parameters.

Concrete Syntax

type parameter = type identifier ;

Abstract Syntax

$\text{Type-parameter} :: \text{type} : \text{Qualident}$

Static Semantics

$wf\text{-type-parameter} : \text{Type-parameter} \rightarrow \text{Environment} \rightarrow \mathbb{B}$
 $wf\text{-type-parameter}(mk\text{-Type-parameter}(qid))\rho \triangleq$
?? $wf\text{-qualident}(qid)\rho \wedge$
270 $is\text{-type}(qid)\rho$

Dynamic Semantics

$m\text{-type-parameter} : \text{Type-parameter} \rightarrow \text{Environment} \xrightarrow{o} \text{Typed}$
 $m\text{-type-parameter}(mk\text{-Type-parameter}(qid))\rho \triangleq$
271 $\text{return type-of}(qid)\rho$

6.8.2 Parameter Matching

The formal parameters shall match the actual parameters.

Formal parameters shall match actual parameters if the number of formal parameters is equal to the number of actual parameters and the formal parameters as defined in the procedure declaration correspond to the actual parameters. The correspondence is established by the positions of the parameters in the lists of formal and actual parameters respectively.

Each formal parameter shall have a type that is parameter compatible to the type of the corresponding actual parameter.

$is\text{-parameters-match} : \text{Formal-parameters-typed} \times \text{Actual-parameters} \rightarrow \text{Environment} \rightarrow \mathbb{B}$
 $is\text{-parameters-match}(fpl, apl)\rho \triangleq$
 $\text{len } fpl = \text{len } apl \wedge$
 $\forall i \in \text{inds } fpl \cdot is\text{-parameter-compatible}(fpl(i), apl(i))\rho$

annotations	Check that the number of formal parameters is equal to the number of actual parameters, and that each of the formal parameters has a type that is parameter compatible with the actual parameter.
-------------	---

6.8.3 Argument Binding

Dynamic Semantics

The value which is the result of evaluating the expression that is the argument shall be assigned to the formal parameter variable. If the formal parameter is of a type that is one of the storage types that is exported from the module SYSTEM, then the argument associated with the parameter is cast to a type constructed from one or more of the smallest addressable units and the resulting value assigned to the formal parameter.

On activation of the block, the current value of the expression is assigned to each variable that is denoted by the formal parameter. The expression is evaluated prior to the procedure activation, and the resulting value is assigned to the formal parameter which constitutes a local variable of the procedure. If the formal parameter is an array, then the actual parameter must be an expression consisting of just a value designator that designates an array.

The order in which the arguments are evaluated is not defined by this International Standard; any order is permissible.

$bind_arguments : Formal_parameter_list \times Arguments \rightarrow Environment \xrightarrow{o} Environment$

```

bind\_arguments (parms, args)  $\rho \triangleq$ 
  if parms = []
  then return  $\rho$ 
  else let  $i : \mathbb{N}$  be st  $i \in inds\ parms$  in
std    let remp = del (parms, i),
std    rema = del (args, i) in
221    def penvi = bind\_argument (parms (i), args (i))  $\rho$ ,
221    penv = bind\_arguments (remp, rema)  $\rho$ ;
272    return overwrite\_var\_environment (penvi  $\cup$  penv)  $\rho$ 

```

$bind_argument : Formal_parameter \times Argument \rightarrow Environment \xrightarrow{o} Environment$

```

bind\_argument (parm, arg)  $\rho \triangleq$ 
  cases parm:
221    mk\_Value\_parameter\_specification (id, ftype)  $\rightarrow$  bind\_argument\_to\_value\_parameter (id, ftype, arg)  $\rho$ ,
222    mk\_Variable\_parameter\_specification (id, ftype)  $\rightarrow$  bind\_argument\_to\_variable\_parameter (id, ftype, arg)  $\rho$ 
end

```

annotations	The argument is bound to the parameter according to whether the formal parameter is a value parameter or a variable parameter.
-------------	--

$bind_argument_to_value_parameter : Identifier \times Formal_type \times Argument \rightarrow Environment \xrightarrow{o} Environment$

```

bind\_argument\_to\_value\_parameter (id, ftype, arg)  $\rho \triangleq$ 
90  let typed = d\_formal\_type (ftype)  $\rho$  in
106  if is\_system\_parameter (typed)  $\rho$ 
    then let locv = cast\_to\_loc (typed, arg)  $\rho$  in
272    return overwrite\_var\_environment ({id  $\mapsto$  locv})  $\rho$ 
272  else return overwrite\_var\_environment ({id  $\mapsto$  arg})  $\rho$ 

```

$bind_value : Variable_typed \times Value \rightarrow Environment \xrightarrow{o} Variable$

```

bind\_value (vtype, arg)  $\rho \triangleq$ 
78  ( vtype  $\in Typed \rightarrow$  def var = m\_typed (vtype)  $\rho$ ;
112    (assign (var, arg)  $\rho$ ;
    return var ),

```

222 $vtype \in \text{Open-array-type}(type) \rightarrow \text{let } mk\text{-Open-array}(ctype) = vtype \text{ in}$
 $\text{build-array}(ctype, arg) \rho$

$build\text{-array} : Typed \times Value \rightarrow Environment \xrightarrow{o} Variable$

$build\text{-array}(ctype, arg) \rho \triangleq$
 51d $\text{let } indx = \{0, \dots, \text{card } arg - 1\} \text{ in}$
 222 $\text{let } offs = \text{mins}(arg) \text{ in}$
 $\text{def } array = \text{bind-array-value}(ctype, indx, arg, offs) \rho;$
 $\text{return } mk\text{-Array-variable}(array)$

$bind\text{-array-value} : Typed \times *\text{-set}\mathbb{N} \times Value \times \mathbb{N} \rightarrow Environment \xrightarrow{o} Variable$

$bind\text{-array-value}((ctype, indx, arg, offset) \rho) \triangleq$
 $\text{if } indx = \{ \}$
 $\text{then return } \{ \}$
 $\text{else let } i : \mathbb{N} \text{ be st } i \in indx \text{ in}$
 $\text{let } remx = indx - \{i\} \text{ in}$
 221 $\text{def } v_i = \text{bind-value}(ctype, arg(i + offset)) \rho,$
 222 $m = \text{bind-array-value}(ctype, remx, arg, offset) \rho;$
 $\text{let } m_i = \{i \mapsto v_i\} \text{ in}$
 $\text{return } m_i \cup m$

$bind\text{-argument-to-variable-parameter} : Identifier \times Formal\text{-type} \times Argument \rightarrow Environment \xrightarrow{o} Environment$

$bind\text{-argument-to-variable-parameter}(id, ftype, arg) \rho \triangleq$
 90 $\text{let } typed = d\text{-formal-type}(ftype) \rho \text{ in}$
 106 $\text{if } is\text{-system-parameter}(typed) \rho$
 222 $\text{then def } locv = \text{storage-as-loc}(typed, arg) \rho;$
 272 $\text{return } \text{overwrite-var-environment}(\{id \mapsto locv\}) \rho$
 272 $\text{else return } \text{overwrite-var-environment}(\{id \mapsto arg\}) \rho$

annotations If the formal parameter is of a type that is one of the storage types that are exported from the module SYSTEM, then the argument associated with the parameter is bound to basic storage which is one or more smallest addressable units and this storage gives an alternative representation to the value(s) found in the actual parameter. If the formal parameter is not of a storage type that is exported from SYSTEM, then the identifier of the formal parameter is bound to the variable.

$storage\text{-as-loc} : Typed \times Argument \rightarrow Environment \xrightarrow{o} Variable$

$storage\text{-as-loc}(typed, arg) \rho \triangleq$
 221 $\text{def } var = \text{bind-value}(typed, arg) \rho;$
 300 $\text{set-aliases}(var, arg)$

annotations Allocate storage for the system storage type and alias it with the storage associated with the variable parameter.

6.9 Pervasive Identifiers

6.9.1 Elementary Data Types

The following elementary data types shall exist.

a) Signed type

A value of signed type shall be an element of an implementation-defined finite subset of the whole numbers. The signed type is one of two required whole number types, and shall be denoted by the pervasive type identifier **INTEGER**. If x is a value of signed type then:

$$x \in \{\text{MIN}(\text{INTEGER}), \dots, \text{MAX}(\text{INTEGER})\}$$

The ordinal number of a non-negative value of signed type shall be the value itself.

b) Unsigned Type

A value of unsigned type shall be an element of an implementation-defined finite subset of the non-negative whole numbers. The unsigned type is one of two required whole number types, and shall be denoted by the pervasive type identifier **CARDINAL**. If x is a value of unsigned type then:

$$x \in \{0, \dots, \text{MAX}(\text{CARDINAL})\}$$

The ordinal number of a value of unsigned type shall be the value itself.

c) Boolean Type

A value of Boolean type shall be one of the logical truth values *true* or *false*. The Boolean type shall be denoted by the pervasive type identifier **BOOLEAN**. The values shall be the enumeration of truth values denoted by the pervasive constant identifiers **FALSE** and **TRUE**. The ordinal numbers of the truth values denoted by **FALSE** and **TRUE** shall be the values 0 and 1 respectively.

d) Character Type

A value of character type shall be an element of a finite and ordered set of characters. The character type shall be denoted by the pervasive type identifier **CHAR**. The values shall be the enumeration of a set of implementation-defined characters, some possibly without graphic representations. The ordinal numbers of the required character values shall be values of unsigned type, that are defined by the implementation.

NOTES

1 The ordering relationship between any two character values is the same as that between their ordinal numbers; see 6.7.1.6 ie. If x and y are any two character values then:

$$x < y \Rightarrow \text{VAL}(\text{CARDINAL}, x) < \text{VAL}(\text{CARDINAL}, y)$$

2 This International Standard does not define the ordinal number of any of the character values, but it does require that the ordering sequence has certain properties; see 6.9.5.

3 A character value denoted by an octal digit sequence followed by 'C' denotes the character whose *ordinal position* is given by the digit sequence; see 6.7.6.4. This is not necessarily the same as the internal code used to represent the character.

e) Real Types

A value of real type shall be an element of one of two implementation-defined finite subsets of the real numbers.

Real: — this type shall be denoted by the pervasive type identifier **REAL**. If x is a real value then:

$$\text{min-real-value} \leq x \leq \text{max-real-value}$$

Long Real: — this type shall be denoted by the pervasive type identifier **LONGREAL**. If x is a long real value then:

$$\text{min-long-real-value} \leq x \leq \text{max-long-real-value}$$

f) Complex Types

A value of complex type shall be an element of one of two implementation-defined finite subsets of the complex numbers.

Complex: — this type shall be denoted by the pervasive type identifier **COMPLEX**. If x is a complex value then:

$$\text{min-real-value} \leq \text{re } x \leq \text{max-real-value} \wedge \text{min-real-value} \leq \text{im } x \leq \text{max-real-value}$$

Long Complex: — this type shall be denoted by the pervasive type identifier **LONGCOMPLEX**. If x is a long complex value then:

$$\text{min-long-real-value} \leq \text{re } x \leq \text{max-long-real-value} \wedge \text{min-long-real-value} \leq \text{im } x \leq \text{max-long-real-value}$$

CHANGE — The complex types are not in *Programming in Modula-2*.

g) Procedure Types

A value of procedure type shall be a level 0 procedure denotation. The procedure shall have no formal parameters. This type shall be denoted by the pervasive identifier **PROC**.

h) Protection Types

A value of protection type shall be one of the values *interruptible* or *uninterruptible*, or one of an implementation defined set of values that include the values *interruptible* and *uninterruptible*. The protection type shall be denoted by the pervasive identifier **PROTECTION**. The value *interruptible* shall be denoted by the pervasive constant identifier **INTERRUPTIBLE** and the value *uninterruptible* shall be denoted by the pervasive constant identifier **UNINTERRUPTIBLE**.

CHANGE — The proposals for protection differ from those in *Programming in Modula-2* which were specific to the PDP-11 interrupt structure.

NOTES

1 A value of protection type, P1 is less than another value, P2, if and only if P1 permits all the interrupts included by P2 but P2 does not permit all the interrupts included by P1.

2 It is likely that the interrupt masking structure of the system underlying the implementation will be richer than the two protection values called for here. As an extension, an implementation may add values (and denotations for those values) to the type **PROTECTION** to reflect the richer structure. The underlying interrupt structure may be totally or partially ordered: an ordinal-like type or a set-like type for the extended **PROTECTION** type would then result. The relational operators on protection have been chosen to be those which are common both to ordinal and to set types.

Extensions to protection shall be implementation defined.

i) Coroutine Types

A value of coroutine type shall be a coroutine denotation that describes the execution state of a coroutine. There are no pervasive identifiers which denote values of coroutine type.

Denotation Types

a) ZZ type

A value of ZZ type shall be an element of an implementation-defined finite subset of the whole numbers denoted as specified in section 6.7.6 by the syntax for a whole number literal. There is no required type identifier for the ZZ type.

If x is of ZZ type then:

$$\min\text{-ZZ-value} \leq x \leq \max\text{-ZZ-value}$$

b) RR type

A value of RR type shall be an element of an implementation-defined finite subset of the real numbers denoted as specified in section 6.7.6 by the syntax for a real literal. There is no required type identifier for the RR type.

c) CC type

A value of CC type shall be an element of an implementation-defined finite subset of the complex numbers denoted by using the standard function **CMPLX** on two arguments of RR type. There is no required type identifier for the CC type.

d) SS Type

A value of SS type shall be a sequence of characters, denoted as specified in section 6.7.6 by the syntax for a string literal. There is no required type identifier for the SS type.

6.9.2 Standard Procedures

The standard procedures are **DEC**, **DISPOSE**, **ENTER**, **EXCL**, **HALT**, **INC**, **INCL**, **LEAVE** and **NEW**.

6.9.2.1 Standard Procedure Designators

Abstract Syntax

types

$$\begin{aligned} \text{Standard-procedure-designator} = & \text{Dec-designator} \\ & | \text{Dispose-designator} \\ & | \text{Enter-designator} \\ & | \text{Excl-designator} \\ & | \text{Halt-designator} \\ & | \text{Inc-designator} \\ & | \text{Incl-designator} \\ & | \text{Leave-designator} \\ & | \text{New-designator} \end{aligned}$$

Static Semantics

functions

$$\text{wf-standard-procedure-designator} : \text{Standard-procedure-designator} \rightarrow \text{Arguments} \rightarrow \text{Environment} \rightarrow \mathbb{B}$$

$$\text{wf-standard-procedure-designator}(\text{desig})(\text{aps})\rho \triangleq$$

$$\begin{aligned} 226 \quad & (\text{is-Dec-designator}(\text{desig}) \rightarrow \text{wf-dec-designator}(\text{desig})(\text{aps})\rho, \\ 228 \quad & \text{is-Dispose-designator}(\text{desig}) \rightarrow \text{wf-dispose-designator}(\text{desig})(\text{aps})\rho, \\ 229 \quad & \text{is-Enter-designator}(\text{desig}) \rightarrow \text{wf-enter-designator}(\text{desig})(\text{aps})\rho, \\ 230 \quad & \text{is-Excl-designator}(\text{desig}) \rightarrow \text{wf-excl-designator}(\text{desig})(\text{aps})\rho, \\ 230 \quad & \text{is-Halt-designator}(\text{desig}) \rightarrow \text{wf-halt-designator}(\text{desig})(\text{aps})\rho, \\ 231 \quad & \text{is-Inc-designator}(\text{desig}) \rightarrow \text{wf-inc-designator}(\text{desig})(\text{aps})\rho, \\ 232 \quad & \text{is-Incl-designator}(\text{desig}) \rightarrow \text{wf-incl-designator}(\text{desig})(\text{aps})\rho, \\ 233 \quad & \text{is-Leave-designator}(\text{desig}) \rightarrow \text{wf-leave-designator}(\text{desig})(\text{aps})\rho, \\ 234 \quad & \text{is-New-designator}(\text{desig}) \rightarrow \text{wf-new-designator}(\text{desig})(\text{aps})\rho \end{aligned}$$

Dynamic Semantics

operations

$$m\text{-standard-procedure-designator} : \text{Standard-procedure-designator} \rightarrow \text{Arguments} \rightarrow \text{Environment} \xrightarrow{o} ()$$

$$m\text{-standard-procedure-designator}(desig)\rho \triangleq$$

226 $(\text{is-Dec-designator}(desig) \rightarrow m\text{-dec-designator}(desig)(args)\rho,$
 228 $\text{is-Dispose-designator}(desig) \rightarrow m\text{-dispose-designator}(desig)(args)\rho,$
 229 $\text{is-Enter-designator}(desig) \rightarrow m\text{-enter-designator}(desig)(args)\rho,$
 230 $\text{is-Excl-designator}(desig) \rightarrow m\text{-excl-designator}(desig)(args)\rho,$
 230 $\text{is-Halt-designator}(desig) \rightarrow m\text{-halt-designator}(desig)(args)\rho,$
 231 $\text{is-Inc-designator}(desig) \rightarrow m\text{-inc-designator}(desig)(args)\rho,$
 232 $\text{is-Incl-designator}(desig) \rightarrow m\text{-incl-designator}(desig)(args)\rho,$
 233 $\text{is-Leave-designator}(desig) \rightarrow m\text{-leave-designator}(desig)(args)\rho,$
 234 $\text{is-New-designator}(desig) \rightarrow m\text{-new-designator}(desig)(args)\rho)$

6.9.2.2 The Procedure DEC

Abstract Syntax

types

$$\text{Dec-designator} :: \text{rtype} : \text{Typed}$$

Static Semantics

A call of **DEC** shall have one or two actual parameters. The first parameter shall be a variable which is of an ordinal type. If there is a second parameter, it shall be an expression of a whole number type. If the second parameter is a constant expression of **ZZ** type and the first parameter is of a signed type, the value of the constant expression shall be less than or equal to the maximum value of the signed type. If the second parameter is a constant expression of **ZZ** type and the first parameter is not of a signed type, the value of the constant expression of **ZZ** type shall be greater than or equal to zero.

functions

$$wf\text{-dec-designator} : \text{Dec-designator} \rightarrow \text{Actual-parameters} \rightarrow \text{Environment} \rightarrow \mathbb{B}$$

$$wf\text{-dec-designator}(mk\text{-Dec-designator}(\text{rtype}))(\text{aps})\rho \triangleq$$

cases aps :

227 $[\text{var}] \rightarrow wf\text{-dec-inc-var}(\text{rtype}, \text{var})\rho,$
 227 $[\text{var}, \text{expr}] \rightarrow wf\text{-dec-inc-var}(\text{rtype}, \text{var})\rho \wedge$
 227 $wf\text{-dec-inc-value}(\text{rtype}, \text{expr})\rho$
 end

Dynamic Semantics

A call of **DEC** with one parameter shall assign to the variable the predecessor of its current value. It shall be an exception if the predecessor lies outside the range of the type of the variable or there is no predecessor.

A call of **DEC** with two parameters shall assign to the variable the N^{th} predecessor of its current value, where N is the value of the second actual parameter. It shall be an exception if the N^{th} predecessor lies outside the range of the type of the variable or there is no N^{th} predecessor.

operations

$$m\text{-dec-designator} : \text{Dec-designator} \rightarrow \text{Arguments} \rightarrow \text{Environment} \xrightarrow{o} ()$$

$$m\text{-dec-designator}(mk\text{-Dec-designator}(\text{rtype}))(\text{args})\rho \triangleq$$

cases args :

227 $[\text{var}] \rightarrow m\text{-inc-dec}(\text{rtype}, \text{var}, -1)\rho,$

```

227   [var, val] → m-inc-dec(rtype, var, -val) ρ
      end;

m-inc-dec : Typed × Variable × Value → Environment →o ()
m-inc-dec (rtype, var, step)  $\triangleq$ 
283   let vtype = if host-type-of (rtype) ρ = SIGNED-TYPE
                then SIGNED-TYPE
                else UNSIGNED-TYPE in
77    let val = converted-result(vtype, variable-value (var)) ρ in
77    let change = converted-result(vtype, step) ρ in
77    let rval = converted-result(rtype, val - change) in
112   assign(var, rval) ρ

```

Language Clarification

Programming in Modula-2 does not fully specify the types of the parameters of **DEC**.

Auxiliary Definitions

functions

```

wf-dec-inc-var : Typed × Actual-parameter → Environment →  $\mathbb{B}$ 
wf-dec-inc-var (rtype, var) ρ  $\triangleq$ 
  is-Variable-designator(var) ∧
145   rtype = t-variable-designator(var) ρ ∧
280   is-ordinal-type (rtype) ρ

```

annotations Check whether the first parameter of a call of **DEC** or **INC** is valid.

functions

```

wf-dec-inc-value : Typed × Actual-parameter → Environment →  $\mathbb{B}$ 
wf-dec-inc-value (rtype, expr) ρ  $\triangleq$ 
  is-Expression(expr) ∧
283   let vtype = host-type-of (rtype) ρ in
152   let etype = t-expression (expr) ρ in
      etype ∈ {UNSIGNED-TYPE, SIGNED-TYPE} ∨
      (etype =  $\mathbb{Z}$ -TYPE ∧
218   let val = evaluate-constant-expression (expr) ρ in
288   vtype = SIGNED-TYPE ∧ val ≤ maximum (SIGNED-TYPE) ρ
      ∨
      vtype ≠ SIGNED-TYPE ∧ val ≥ 0)

```

annotations Check whether the two parameters of a call of **DEC** or **INC** are valid. This includes checking that the second parameter is within range when it is a constant expression of **ZZ** type. The constraints are present because signed arithmetic will be used if the first parameter is of a signed type and unsigned arithmetic will be used in all other cases.

6.9.2.3 The Procedure **DISPOSE**

Abstract Syntax

types

Dispose-designator :: rtype : Typed

Static Semantics

A call of **DISPOSE** shall have one or more actual parameters. The first, and possibly only parameter, shall be a variable that is of a pointer type.

If the second and subsequent parameters are present, they shall be constant expressions, and in this case the bound type of the first parameter shall be a record type.

The identifier **DEALLOCATE** shall have a defining occurrence within the scope of the call of **DISPOSE**. This defining occurrence shall associate **DEALLOCATE** with a proper procedure that has two parameters. The first formal parameter of this proper procedure shall be a variable parameter of the address type and the second formal parameter shall be a value parameter of the unsigned type.

functions

$$\begin{aligned}
 &wf\text{-}dispose\text{-}designator : Dispose\text{-}designator \rightarrow Actual\text{-}parameters \rightarrow Environment \rightarrow \mathbb{B} \\
 &wf\text{-}dispose\text{-}designator(mk\text{-}Dispose\text{-}designator(rtype))(aps)\rho \triangleq \\
 &\quad \text{len } aps \geq 1 \wedge \\
 &\quad \text{let } [var] \curvearrowright exprs = aps \text{ in} \\
 &\quad is\text{-}Variable\text{-}designator(var) \wedge \\
145 &\quad rtype = t\text{-}variable\text{-}designator(var)\rho \wedge \\
279 &\quad is\text{-}pointer\text{-}type(rtype)\rho \wedge \\
272 &\quad is\text{-}proper\text{-}procedure(DEALLOCATE)\rho \wedge \\
273 &\quad associated\text{-}procedure(DEALLOCATE)\rho = storage\text{-}call \wedge \\
&\quad (expr = [] \vee \\
284 &\quad expr \neq [] \wedge \text{let } btype = bound\text{-}type\text{-}of(rtype)\rho \text{ in} \\
271 &\quad \text{let } rs = structure\text{-}of(btype)\rho \text{ in} \\
&\quad is\text{-}Record\text{-}structure(rs) \wedge \\
333 &\quad is\text{-}tags\text{-}in\text{-}field\text{-}list(rs, exprs) \wedge \\
&\quad \forall expr \in \text{elems } exprs \cdot is\text{-}Expression(expr) \wedge \\
152 &\quad wf\text{-}expression(expr)\rho \wedge \\
214 &\quad is\text{-}constant\text{-}expression(expr)\rho)
 \end{aligned}$$

Dynamic Semantics

A call ‘**DISPOSE(P)**’ shall have the same effect as the call ‘**DEALLOCATE(P,S)**’ (where **P** is a variable of the pointer type **T** and **S** is the size of the bound type given in the declaration of **T**).

A call ‘**DEALLOCATE(P,SYSTEM.TSIZE(B,< expression-list >))**’ (where **P** is a pointer variable whose bound type is the type **B**) shall have the same effect as the call ‘**DISPOSE(P,< expression-list >)**’.

operations

$$\begin{aligned}
 &m\text{-}dispose\text{-}designator : Dispose\text{-}designator \rightarrow Arguments \rightarrow Environment \xrightarrow{o} () \\
 &m\text{-}dispose\text{-}designator(mk\text{-}Dispose\text{-}designator(rtype))(args)\rho \triangleq \\
 &\quad \text{let } [var] \curvearrowright vals = args \text{ in} \\
273 &\quad \text{let } f = associated\text{-}procedure(DEALLOCATE)\rho \text{ in} \\
335 &\quad \text{let } size = tsize(rtype.type, vals)\rho \text{ in} \\
&\quad \text{return } f([var, size])
 \end{aligned}$$

Auxiliary Definitions

values

$$\begin{aligned}
 &storage\text{-}call = \\
 &\quad \text{let } args = [mk\text{-}Variable\text{-}formal\text{-}typed(ADDRESS\text{-}TYPE), \\
 &\quad \quad mk\text{-}Value\text{-}formal\text{-}typed(UNSIGNED\text{-}TYPE)] \text{ in} \\
 &\quad mk\text{-}Proper\text{-}procedure\text{-}structure(args)
 \end{aligned}$$

6.9.2.4 The Procedure ENTER

Abstract Syntax

types

Enter-designator ::

Static Semantics

A call of **ENTER** shall have one actual parameter which shall be an expression of the standard type **PROTECTION**.

functions

$wf_enter_designator : Enter_designator \rightarrow Actual_parameters \rightarrow Environment \rightarrow \mathbb{B}$
 $wf_enter_designator(mk_Enter_designator())(aps)\rho \triangleq$
 $\quad \text{len } aps = 1 \wedge$
 $\quad \text{let } [expr] = aps \text{ in}$
 $\quad is_Expression(expr) \wedge$
 $\quad t_expression(expr)\rho = \text{PROTECTION-TYPE}$

Dynamic Semantics

A call of **ENTER** shall set the current protection to the value of the expression and save the previous value. It shall be an exception if the value of the expression is less restrictive than the current protection.

operations

$m_enter_designator : Enter_designator \rightarrow Arguments \rightarrow Environment \xrightarrow{o} ()$
 $m_enter_designator(mk_Enter_designator())(arg)\rho \triangleq$
 $\quad \text{let } [val] = arg \text{ in}$
 $\quad \text{if } is_less_restrictive_protection(val)$
 $\quad \text{then } mandatory_exception(\text{LESSPROTECTION})$
 $\quad \text{else } set_protection(val)$

CHANGE — The procedure **ENTER** is not in *Programming in Modula-2*.

6.9.2.5 The Procedure EXCL

Abstract Syntax

types

Excl-designator :: *rtype* : *Typed*

Static Semantics

A call of **EXCL** shall have two actual parameters. The first actual parameter shall be a variable which is of a set type and the second actual parameter shall be an expression. The type of the expression shall be the host type of the base type of the set type.

functions

$wf\text{-}excl\text{-}designator : Excl\text{-}designator \rightarrow Actual\text{-}parameters \rightarrow Environment \rightarrow \mathbb{B}$
 $wf\text{-}excl\text{-}designator (mk\text{-}Excl\text{-}designator(rtype))(aps)\rho \triangleq$
 $\quad \text{len } aps = 2 \wedge$
 $\quad \text{let } [var, expr] = aps \text{ in}$
 $\quad is\text{-}Variable\text{-}designator(var) \wedge$
 $\quad is\text{-}Expression(expr) \wedge$
145 $\quad rtype = t\text{-}variable\text{-}designator(var)\rho \wedge$
279 $\quad is\text{-}set\text{-}type(rtype)\rho \wedge$
283 $\quad t\text{-}expression(expr)\rho = host\text{-}type\text{-}of(base\text{-}type\text{-}of(rtype)\rho)\rho$

Dynamic Semantics

A call of **EXCL** shall assign to the set variable the value that is obtained by removing the value of the expression from the current value of the set variable. It shall be an exception if the value of the expression is not a value of the base type of the type of the set variable.

operations

$m\text{-}excl\text{-}designator : Excl\text{-}designator \rightarrow Arguments \rightarrow Environment \xrightarrow{o} ()$
 $m\text{-}excl\text{-}designator (mk\text{-}Excl\text{-}designator(rtype))(args)\rho \triangleq$
 $\quad \text{let } [var, val] = args \text{ in}$
 $\quad \text{let } mk\text{-}Set\text{-}structure(btype) = structure\text{-}of(rtype)\rho \text{ in}$
290 $\quad \text{let } range = values\text{-}of(btype)\rho \text{ in}$
 $\quad \text{if } val \in range$
?? $\quad \text{then let } value = variable\text{-}value(var) \text{ in}$
112 $\quad \quad assign(var, value - \{val\}) \rho$
306 $\quad \text{else } mandatory\text{-}exception(EXCLNOTBASE)$

6.9.2.6 The Procedure HALT

Abstract Syntax

types

$Halt\text{-}designator ::$

Static Semantics

A call of **HALT** shall have no actual parameters.

functions

$wf\text{-}halt\text{-}designator : Halt\text{-}designator \rightarrow Actual\text{-}parameters \rightarrow Environment \rightarrow \mathbb{B}$
 $wf\text{-}halt\text{-}designator (mk\text{-}Halt\text{-}designator())(aps)\rho \triangleq$
 $\quad \text{len } aps = 0$

Dynamic Semantics

A call of **HALT** shall terminate the execution of the program (but see section 7.4).

operations

$m\text{-}halt\text{-}designator : Halt\text{-}designator \rightarrow Arguments \rightarrow Environment \xrightarrow{o} ()$
 $m\text{-}halt\text{-}designator (mk\text{-}Halt\text{-}designator())([])\rho \triangleq$
?? $\quad c\text{-}exit(HALT) \rho$

6.9.2.7 The Procedure **INC**

Abstract Syntax

types

$Inc\text{-}designator :: rtype : Typed$

Static Semantics

A call of **INC** shall have one or two actual parameters. The first parameter shall be a variable which is of an ordinal type. If there is a second parameter, it shall be an expression of a whole number type. If the second parameter is a constant expression of ZZ type and the first parameter is of a signed type, the value of the constant expression shall be less than or equal to the maximum value of the signed type. If the second parameter is a constant expression of ZZ type and the first parameter is not of a signed type, the value of the constant expression shall be greater than or equal to zero.

functions

$wf\text{-}inc\text{-}designator : Inc\text{-}designator \rightarrow Actual\text{-}parameters \rightarrow Environment \rightarrow \mathbb{B}$
 $wf\text{-}inc\text{-}designator (mk\text{-}Inc\text{-}designator(rtype))(aps)\rho \triangleq$
cases aps :
227 $[var] \rightarrow wf\text{-}dec\text{-}inc\text{-}var(rtype, var)\rho,$
227 $[var, expr] \rightarrow wf\text{-}dec\text{-}inc\text{-}var(rtype, var)\rho \wedge$
227 $wf\text{-}dec\text{-}inc\text{-}value(rtype, expr)\rho$
end

Dynamic Semantics

A call of **INC** with one parameter shall assign to the variable the successor of its current value. It shall be an exception if the successor lies outside the range of the type of the variable or there is no successor.

A call of **INC** with two parameters shall assign to the variable the N^{th} successor of its current value, where N is the value of the second actual parameter. It shall be an exception if the N^{th} successor lies outside the range of the type of the variable or there is no N^{th} successor.

operations

$m\text{-}inc\text{-}designator : Inc\text{-}designator \rightarrow Arguments \rightarrow Environment \xrightarrow{o} ()$
 $m\text{-}inc\text{-}designator (mk\text{-}Inc\text{-}designator(rtype))(args)\rho \triangleq$
cases $args$:
227 $[var] \rightarrow m\text{-}inc\text{-}dec(rtype, var, 1)\rho,$
227 $[var, val] \rightarrow m\text{-}inc\text{-}dec(rtype, var, val)\rho$
end

Language Clarification

Programming in Modula-2 does not fully specify the types of the parameters of **INC**.

6.9.2.8 The Procedure INCL

Abstract Syntax

types

$Incl-designator :: rtype : Typed$

Static Semantics

A call of **INCL** shall have two actual parameters. The first actual parameter shall be a variable which is of a set type and the second actual parameter shall be an expression. The type of the expression and the host type of the base type of the set type shall be identical types.

functions

$wf-incl-designator : Incl-designator \rightarrow Actual-parameters \rightarrow Environment \rightarrow \mathbb{B}$

$wf-incl-designator(mk-Incl-designator(rtype))(aps)\rho \triangleq$
let $aps = 2 \wedge$
let $[var, expr] = aps$ in
 $is-Variable-designator(var) \wedge$
 $is-Expression(expr) \wedge$
145 $rtype = t-variable-designator(var)\rho \wedge$
279 $is-set-type(rtype)\rho \wedge$
283 $t-expression(expr)\rho = host-type-of(base-type-of(rtype)\rho)\rho$

Dynamic Semantics

A call of **INCL** shall assign to the set variable the value that is obtained by including the value of the expression in the current value of the set variable. It shall be an exception if the value of the expression is not a value of the base type of the type of the set variable.

operations

$m-incl-designator : Incl-designator \rightarrow Arguments \rightarrow Environment \xrightarrow{o} ()$

$m-incl-designator(mk-Incl-designator(rtype))(args)\rho \triangleq$
let $[var, val] = args$ in
271 let $mk-Set-structure(btype) = structure-of(rtype)\rho$ in
290 let $range = values-of(btype)\rho$ in
if $val \in range$
?? then let $value = variable-value(var)$ in
112 $assign(var, value \cup \{val\})\rho$
306 else $mandatory-exception(INCLNOTBASE)$

6.9.2.9 The Procedure LEAVE

Abstract Syntax

types

$Leave-designator ::$

Static Semantics

A call of **LEAVE** shall have one actual parameter which shall be an expression of the standard type **PROTECTION**.

functions

$wf\text{-}leave\text{-}designator : Leave\text{-}designator \rightarrow Actual\text{-}parameters \rightarrow Environment \rightarrow \mathbb{B}$

$wf\text{-}leave\text{-}designator(mk\text{-}Leave\text{-}designator())(aps)\rho \triangleq$

$\text{len } aps = 1 \wedge$

$\text{let } [expr] = aps \text{ in}$

$is\text{-}Expression(expr) \wedge$

152 $t\text{-}expression(expr)\rho = \text{PROTECTION-TYPE}$

Dynamic Semantics

A call of **LEAVE** shall restore the current protection to the value which it held on the corresponding call of **ENTER**. It shall be an exception if the value of the expression is not equal to the current protection on the call of **LEAVE**.

NOTE — The requirement for the value of the parameter to **LEAVE** to equal the current protection provides a consistency check when **ENTER** and **LEAVE** are called explicitly in protected modules.

operations

$m\text{-}leave\text{-}designator : Leave\text{-}designator \rightarrow Arguments \rightarrow Environment \xrightarrow{o} ()$

$m\text{-}leave\text{-}designator(mk\text{-}Leave\text{-}designator())(arg)\rho \triangleq$

$\text{let } [val] = arg \text{ in}$

?? $\text{if } is\text{-}current\text{-}protection(val)$

304 $\text{then } restore\text{-}protection()$

306 $\text{else } mandatory\text{-}exception(\text{PROTECTIONERROR})$

NOTE — The current protection may be lowered temporarily by calling the procedure **COROUTINES.LISTEN**.

CHANGE — The procedure **LEAVE** is not in *Programming in Modula-2*.

6.9.2.10 The Procedure **NEW**

Abstract Syntax

types

$New\text{-}designator :: rtype : Typed$

Static Semantics

A call of **NEW** shall have one or more actual parameters. The first, and possibly only parameter, shall be a variable that is of a pointer type.

If the second and subsequent parameters are present, they shall be constant expressions, and in this case the bound type of the first parameter shall be a record type.

The identifier **ALLOCATE** shall have a defining occurrence within the scope of the call of **NEW**. This defining occurrence shall associate **ALLOCATE** with a proper procedure that has two parameters. The first formal parameter of this proper procedure shall be a variable parameter of the address type and the second formal parameter shall be a value parameter of the unsigned type.

```

wf-new-designator : New-designator  $\rightarrow$  Actual-parameters  $\rightarrow$  Environment  $\rightarrow \mathbb{B}$ 
wf-new-designator (mk-New-designator(rtype))(aps) $\rho \triangleq$ 
  len aps  $\geq 1 \wedge$ 
  let [var]  $\curvearrowright$  exprs = aps in
  is-Variable-designator(var)  $\wedge$ 
145  rtype = t-variable-designator(var) $\rho \wedge$ 
279  is-pointer-type(rtype) $\rho \wedge$ 
272  is-proper-procedure(ALLOCATE) $\rho \wedge$ 
273  associated-procedure(ALLOCATE) $\rho = storage-call \wedge$ 
  (exprs = [])
   $\vee$ 
284  let btype = bound-type-of(rtype) $\rho$  in
271  (let rs = structure-of(btype) $\rho$  in
    is-Record-structure(rs)  $\wedge$ 
    len exprs  $\geq 1 \wedge$ 
333  is-tags-in-field-list(rs, exprs)  $\wedge$ 
     $\forall expr \in \text{elems } exprs .$ 
      is-Expression(expr)  $\wedge$ 
252  wf-expression(expr) $\rho \wedge is-constant-expression(expr)\rho$ )

```

Dynamic Semantics

A call ‘**NEW(P)**’ shall have the same effect as the call ‘**ALLOCATE(P,S)**’ (where **P** is a variable of the pointer type **T** and **S** is the size of the bound type given in the declaration of **T**).

A call ‘**ALLOCATE(P,SYSTEM.TSIZE(B,< expression-list >))**’ (where **P** is a pointer variable whose bound type is the type **B**) shall have the same effect as the call ‘**NEW(P,< expression-list >)**’.

operations

```

m-new-designator : New-designator  $\rightarrow$  Arguments  $\rightarrow$  Environment  $\xrightarrow{o}$  ()
m-new-designator (mk-New-designator(rtype))(args) $\rho \triangleq$ 
  let [var]  $\curvearrowright$  vals = args in
273  let f = associated-procedure(ALLOCATE) $\rho$  in
335  let size = tsize(rtype.type, vals) $\rho$  in
  f([var, size])

```

6.9.3 Standard Functions

The standard functions are **ABS**, **CAP**, **CHR**, **CMPLX**, **FLOAT**, **HIGH**, **IM**, **INT**, **LENGTH**, **LFLOAT**, **MAX**, **MIN**, **ODD**, **ORD**, **PROT**, **RE**, **SIZE**, **TRUNC** and **VAL**.

6.9.3.1 Standard Function Designators

Abstract Syntax

types

$$\begin{aligned} \text{Standard-function-designator} = & \text{Abs-designator} \\ & | \text{Cap-designator} \\ & | \text{Chr-designator} \\ & | \text{Cmplx-designator} \\ & | \text{Float-designator} \\ & | \text{High-designator} \\ & | \text{Im-designator} \\ & | \text{Int-designator} \\ & | \text{Length-designator} \\ & | \text{Lfloat-designator} \\ & | \text{Max-designator} \\ & | \text{Min-designator} \\ & | \text{Odd-designator} \\ & | \text{Ord-designator} \\ & | \text{Prot-designator} \\ & | \text{Re-designator} \\ & | \text{Size-designator} \\ & | \text{Trunc-designator} \\ & | \text{Val-designator} \end{aligned}$$

Static Semantics

functions

$$\text{wf-standard-function-designator} : \text{Standard-function-designator} \rightarrow \text{Actual-parameters} \rightarrow \text{Environment} \rightarrow \mathbb{B}$$

$$\text{wf-standard-function-designator}(\text{desig})(\text{aps})\rho \triangleq$$

237	$\text{is-Abs-designator}(\text{desig})$	$\rightarrow \text{wf-abs-designator}(\text{desig})(\text{aps})\rho,$
238	$\text{is-Cap-designator}(\text{desig})$	$\rightarrow \text{wf-cap-designator}(\text{desig})(\text{aps})\rho,$
239	$\text{is-Chr-designator}(\text{desig})$	$\rightarrow \text{wf-chr-designator}(\text{desig})(\text{aps})\rho,$
240	$\text{is-Cmplx-designator}(\text{desig})$	$\rightarrow \text{wf-cmplx-designator}(\text{desig})(\text{aps})\rho,$
240	$\text{is-Float-designator}(\text{desig})$	$\rightarrow \text{wf-float-designator}(\text{desig})(\text{aps})\rho,$
241	$\text{is-High-designator}(\text{desig})$	$\rightarrow \text{wf-high-designator}(\text{desig})(\text{aps})\rho,$
242	$\text{is-Im-designator}(\text{desig})$	$\rightarrow \text{wf-im-designator}(\text{desig})(\text{aps})\rho,$
243	$\text{is-Int-designator}(\text{desig})$	$\rightarrow \text{wf-int-designator}(\text{desig})(\text{aps})\rho,$
244	$\text{is-Length-designator}(\text{desig})$	$\rightarrow \text{wf-length-designator}(\text{desig})(\text{aps})\rho,$
245	$\text{is-Lfloat-designator}(\text{desig})$	$\rightarrow \text{wf-lfloat-designator}(\text{desig})(\text{aps})\rho,$
246	$\text{is-Max-designator}(\text{desig})$	$\rightarrow \text{wf-max-designator}(\text{desig})(\text{aps})\rho; ,$
247	$\text{is-Min-designator}(\text{desig})$	$\rightarrow \text{wf-min-designator}(\text{desig})(\text{aps})\rho,$
248	$\text{is-Odd-designator}(\text{desig})$	$\rightarrow \text{wf-odd-designator}(\text{desig})(\text{aps})\rho,$
249	$\text{is-Ord-designator}(\text{desig})$	$\rightarrow \text{wf-ord-designator}(\text{desig})(\text{aps})\rho,$
249	$\text{is-Prot-designator}(\text{desig})$	$\rightarrow \text{wf-prot-designator}(\text{desig})(\text{aps})\rho,$
250	$\text{is-Re-designator}(\text{desig})$	$\rightarrow \text{wf-re-designator}(\text{desig})(\text{aps})\rho,$
251	$\text{is-Size-designator}(\text{desig})$	$\rightarrow \text{wf-size-designator}(\text{desig})(\text{aps})\rho,$
252	$\text{is-Trunc-designator}(\text{desig})$	$\rightarrow \text{wf-trunc-designator}(\text{desig})(\text{aps})\rho,$
253	$\text{is-Val-designator}(\text{desig})$	$\rightarrow \text{wf-val-designator}(\text{desig})(\text{aps})\rho;$

$t\text{-standard-function-designator} : \text{Standard-function-designator} \rightarrow \text{Actual-parameters} \rightarrow \text{Environment} \rightarrow \text{Expression-typed}$

$t\text{-standard-function-designator}(\text{desig})(\text{aps})\rho \triangleq$

237	$(\text{is-Abs-designator}(\text{desig}) \rightarrow t\text{-abs-designator}(\text{desig})(\text{aps})\rho,$
238	$\text{is-Cap-designator}(\text{desig}) \rightarrow t\text{-cap-designator}(\text{desig})(\text{aps})\rho,$
239	$\text{is-Chr-designator}(\text{desig}) \rightarrow t\text{-chr-designator}(\text{desig})(\text{aps})\rho,$
240	$\text{is-Cmplx-designator}(\text{desig}) \rightarrow t\text{-cmplx-designator}(\text{desig})(\text{aps})\rho,$
240	$\text{is-Float-designator}(\text{desig}) \rightarrow t\text{-float-designator}(\text{desig})(\text{aps})\rho,$
241	$\text{is-High-designator}(\text{desig}) \rightarrow t\text{-high-designator}(\text{desig})(\text{aps})\rho,$
243	$\text{is-Int-designator}(\text{desig}) \rightarrow t\text{-int-designator}(\text{desig})(\text{aps})\rho,$
242	$\text{is-Im-designator}(\text{desig}) \rightarrow t\text{-im-designator}(\text{desig})(\text{aps})\rho,$
244	$\text{is-Length-designator}(\text{desig}) \rightarrow t\text{-length-designator}(\text{desig})(\text{aps})\rho,$
245	$\text{is-Lfloat-designator}(\text{desig}) \rightarrow t\text{-lfloat-designator}(\text{desig})(\text{aps})\rho,$
246	$\text{is-Max-designator}(\text{desig}) \rightarrow t\text{-max-designator}(\text{desig})(\text{aps})\rho,$
247	$\text{is-Min-designator}(\text{desig}) \rightarrow t\text{-min-designator}(\text{desig})(\text{aps})\rho,$
248	$\text{is-Odd-designator}(\text{desig}) \rightarrow t\text{-odd-designator}(\text{desig})(\text{aps})\rho,$
249	$\text{is-Ord-designator}(\text{desig}) \rightarrow t\text{-ord-designator}(\text{desig})(\text{aps})\rho,$
249	$\text{is-Prot-designator}(\text{desig}) \rightarrow t\text{-prot-designator}(\text{desig})(\text{aps})\rho,$
250	$\text{is-Re-designator}(\text{desig}) \rightarrow t\text{-re-designator}(\text{desig})(\text{aps})\rho,$
251	$\text{is-Size-designator}(\text{desig}) \rightarrow t\text{-size-designator}(\text{desig})(\text{aps})\rho,$
252	$\text{is-Trunc-designator}(\text{desig}) \rightarrow t\text{-trunc-designator}(\text{desig})(\text{aps})\rho,$
253	$\text{is-Val-designator}(\text{desig}) \rightarrow t\text{-val-designator}(\text{desig})(\text{aps})\rho)$

Dynamic Semantics

operations

$m\text{-standard-function-designator} : \text{Standard-function-designator} \rightarrow \text{Arguments} \rightarrow \text{Environment} \rightarrow \mathbb{B}$

$m\text{-standard-function-designator}(\text{desig})(\text{args})\rho \triangleq$

237	$(\text{is-Abs-designator}(\text{desig}) \rightarrow m\text{-abs-designator}(\text{desig})(\text{args})\rho,$
238	$\text{is-Cap-designator}(\text{desig}) \rightarrow m\text{-cap-designator}(\text{desig})(\text{args})\rho,$
239	$\text{is-Chr-designator}(\text{desig}) \rightarrow m\text{-chr-designator}(\text{desig})(\text{args})\rho,$
240	$\text{is-Cmplx-designator}(\text{desig}) \rightarrow m\text{-cmplx-designator}(\text{desig})(\text{args})\rho,$
241	$\text{is-Float-designator}(\text{desig}) \rightarrow m\text{-float-designator}(\text{desig})(\text{args})\rho,$
242	$\text{is-High-designator}(\text{desig}) \rightarrow m\text{-high-designator}(\text{desig})(\text{args})\rho,$
244	$\text{is-Int-designator}(\text{desig}) \rightarrow m\text{-int-designator}(\text{desig})(\text{args})\rho,$
243	$\text{is-Im-designator}(\text{desig}) \rightarrow m\text{-im-designator}(\text{desig})(\text{args})\rho,$
244	$\text{is-Length-designator}(\text{desig}) \rightarrow m\text{-length-designator}(\text{desig})(\text{args})\rho,$
245	$\text{is-Lfloat-designator}(\text{desig}) \rightarrow m\text{-lfloat-designator}(\text{desig})(\text{args})\rho,$
246	$\text{is-Max-designator}(\text{desig}) \rightarrow m\text{-max-designator}(\text{desig})(\text{args})\rho,$
247	$\text{is-Min-designator}(\text{desig}) \rightarrow m\text{-min-designator}(\text{desig})(\text{args})\rho,$
248	$\text{is-Odd-designator}(\text{desig}) \rightarrow m\text{-odd-designator}(\text{desig})(\text{args})\rho,$
249	$\text{is-Ord-designator}(\text{desig}) \rightarrow m\text{-ord-designator}(\text{desig})(\text{args})\rho,$
250	$\text{is-Prot-designator}(\text{desig}) \rightarrow m\text{-prot-designator}(\text{desig})(\text{args})\rho,$
250	$\text{is-Re-designator}(\text{desig}) \rightarrow m\text{-re-designator}(\text{desig})(\text{args})\rho,$
251	$\text{is-Size-designator}(\text{desig}) \rightarrow m\text{-size-designator}(\text{desig})(\text{args})\rho,$
252	$\text{is-Trunc-designator}(\text{desig}) \rightarrow m\text{-trunc-designator}(\text{desig})(\text{args})\rho,$
254	$\text{is-Val-designator}(\text{desig}) \rightarrow m\text{-val-designator}(\text{desig})(\text{args})\rho)$

6.9.3.2 The Function ABS

Abstract Syntax

types

$\text{Abs-designator} :: \text{rtype} : \text{Expression-typed}$

Static Semantics

A call of **ABS** shall have one actual parameter which shall be an expression of the signed type, a constant expression of ZZ type, or an expression of a real number type.

The value returned by a call of **ABS** shall be of the same type as the expression.

NOTE — An expression which simply consists of a value designator that is of a subrange-type is regarded as being of the host-type of the subrange-type. A call of **ABS** applied to such an expression returns a value of the host-type.

functions

$wf-abs-designator : Abs-designator \rightarrow Actual-parameters \rightarrow Environment \rightarrow \mathbb{B}$

$wf-abs-designator(mk-Abs-designator(rtype))(aps)\rho \triangleq$

$\text{len } aps = 1 \wedge$

$\text{let } [expr] = aps \text{ in}$

$is-Expression(expr) \wedge$

$(rtype = \text{SIGNED-TYPE} \vee rtype = \text{Z-TYPE} \vee is-Real-number-type(rtype)) \wedge$

152 $t-expression(expr)\rho = rtype;$

$t-abs-designator : Abs-designator \rightarrow Actual-parameters \rightarrow Environment \rightarrow Expression-typed$

$t-abs-designator(mk-Abs-designator(rtype))(aps)\rho \triangleq$

$rtype$

Dynamic Semantics

A call of **ABS** shall return the absolute value of the expression. It shall be an exception if this value lies outside the range of the type of the expression.

NOTE — An exception will occur when the value of the expression is negative and the corresponding positive value does not belong to the type of the expression — this can only occur if the type has a non-symmetric range.

operations

$m-abs-designator : Abs-designator \rightarrow Arguments \rightarrow Environment \xrightarrow{o} Value$

$m-abs-designator(mk-Abs-designator(rtype))(args)\rho \triangleq$

$\text{let } [arg] = args \text{ in}$

$(arg \geq 0 \rightarrow \text{return } arg ,$

162 $rtype = \text{SIGNED-TYPE} \rightarrow get-whole-result(rtype, -arg) ,$

162 $rtype = \text{Z-TYPE} \rightarrow get-whole-result(rtype, -arg) ,$

159 $is-Real-number-type(rtype) \rightarrow get-real-result(rtype, -arg))$

Language Clarification

Programming in Modula-2 does not specify the type of the parameter of **ABS**. This International Standard does not permit **ABS** to be applied to a parameter of the unsigned type.

6.9.3.3 The Function CAP

Abstract Syntax

types

$Cap-designator ::$

Static Semantics

A call of **CAP** shall have one actual parameter which shall be an expression of the character type.

The value returned by a call of **CAP** shall be of the character type.

functions

```
wf-cap-designator : Cap-designator → Actual-parameters → Environment → B
wf-cap-designator (mk-Cap-designator())(aps)ρ  $\triangle$ 
  len aps = 1 ∧
  let [expr] = aps in
  is-Expression(expr) ∧
152  t-expression(expr)ρ = CHARACTER-TYPE;

t-cap-designator : Cap-designator → Actual-parameters → Environment → Typed
t-cap-designator (mk-Cap-designator())(aps)ρ  $\triangle$ 
  CHARACTER-TYPE
```

Dynamic Semantics

If the value of the expression is a lower-case letter, a call of **CAP** shall return the corresponding upper-case letter; otherwise the value of the expression shall be returned.

operations

```
m-cap-designator : Cap-designator → Arguments → Environment  $\xrightarrow{o}$  Value
m-cap-designator (mk-Cap-designator(-, -))(args)ρ  $\triangle$ 
  let [ch] = args in
  if ch ∈ dom capitalisations
77  then return capitalisations(ch)
  else return ch
```

Language Clarification

Programming in Modula-2 only defines **CAP** when the parameter is a letter, although some examples assume that **CAP** is defined for all values of character type. This International Standard defines **CAP** for all values of character type.

6.9.3.4 The Function CHR

Abstract Syntax

types

Chr-designator ::

Static Semantics

A call of **CHR** shall have one actual parameter which shall be an expression of a whole number type.

The value returned by a call of **CHR** shall be of the character type.

functions

$wf\text{-}chr\text{-}designator : Chr\text{-}designator \rightarrow Actual\text{-}parameters \rightarrow Environment \rightarrow B$

$wf\text{-}chr\text{-}designator(mk\text{-}Chr\text{-}designator())(aps)\rho \triangleq$

$\text{len } aps = 1 \wedge$

$\text{let } [expr] = aps \text{ in}$

$is\text{-}Expression(expr) \wedge$

252 $is\text{-}whole\text{-}number\text{-}type(t\text{-}expression(expr)\rho)\rho;$

$t\text{-}chr\text{-}designator : Chr\text{-}designator \rightarrow Actual\text{-}parameters \rightarrow Environment \rightarrow Typed$

$t\text{-}chr\text{-}designator(mk\text{-}Chr\text{-}designator())(aps)\rho \triangleq$

CHARACTER-TYPE

Dynamic Semantics

The call ‘**CHR**(**x**)’ shall have the same effect as the call ‘**VAL**(**CHAR**, **x**)’.

NOTE — An exception will occur if the expression is a whole number literal value which is less than zero or if the value of the expression exceeds the ordinal number of the maximum value of the character type.

operations

$m\text{-}chr\text{-}designator : Chr\text{-}designator \rightarrow Arguments \rightarrow Environment \xrightarrow{o} Value$

$m\text{-}chr\text{-}designator(mk\text{-}Chr\text{-}designator())(args)\rho \triangleq$

$\text{let } [value] = args \text{ in}$

254 $converted\text{-}result(CHARACTER\text{-}TYPE, value) \rho$

6.9.3.5 The Function CMPLX

Abstract Syntax

types

$Cmplx\text{-}designator :: rtype : Expression\text{-}typed$

Static Semantics

A call of **CMPLX** shall have two actual parameters which shall both be expressions.

The type of the result shall be of complex type if both expressions are of real type, or if one is of real type and one is a constant expression of RR type. The type of the result shall be of long complex type if both expressions are of long real type, or one is of long real type and one is a constant expression of RR type. If both parameters are constant expressions of RR type, the result shall be of CC type.

NOTE — The following table gives the permitted types and the type of the result.

	real type	long real type	RR type
real type	complex type	error	complex type
long real type	error	long complex type	long complex type
RR type	complex type	long complex type	CC type

functions

```

wf-cmplx-designator : Cmplx-designator → Actual-parameters → Environment →  $\mathbb{B}$ 
wf-cmplx-designator (mk-Cmplx-designator(rtype))(aps)  $\rho \triangleq$ 
  len aps = 2  $\wedge$ 
  let [re-expr, im-expr] = aps in
152   is-Expression(re-expr)  $\wedge$  is-Real-number-type(t-expression(re-expr) $\rho$ )  $\wedge$ 
152   is-Expression(im-expr)  $\wedge$  is-Real-number-type(t-expression(im-expr) $\rho$ )  $\wedge$ 
152   rtype = complex-result-type(t-expression(re-expr) $\rho$ , t-expression(im-expr) $\rho$ );

t-cmplx-designator : Cmplx-designator → Actual-parameters → Environment → Expression-typed
t-cmplx-designator (mk-Cmplx-designator(rtype))(aps)  $\rho \triangleq$ 
  rtype

```

Dynamic Semantics

The call '**CMPLX**(**x**, **y**)' shall return a value of the complex type whose real part is **x** and whose imaginary part is **y**.

NOTE — An exception will occur if the value lies outside the range of the result type.

operations

```

m-cmplx-designator : Cmplx-designator → Arguments → Environment  $\xrightarrow{o}$  Value
m-cmplx-designator (mk-Cmplx-designator(rtype))(args)  $\rho \triangleq$ 
  let [x, y] = args in
157   get-complex-result(rtype, x + iy)

```

CHANGE — The function **CMPLX** is not in *Programming in Modula-2*.

6.9.3.6 The Function **FLOAT**

Abstract Syntax

types

Float-designator ::

Static Semantics

A call of **FLOAT** shall have one actual parameter which shall be an expression of a number type.

The value returned by a call of **FLOAT** shall be of the real type.

functions

```

wf-float-designator : Float-designator → Actual-parameters → Environment →  $\mathbb{B}$ 
wf-float-designator (mk-Float-designator())(aps)  $\rho \triangleq$ 
  len aps = 1  $\wedge$ 
  let [expr] = aps in
  is-Expression(expr)  $\wedge$ 
282   is-number-type(t-expression(expr) $\rho$ ) $\rho$ ;

t-float-designator : Float-designator → Actual-parameters → Environment → Typed
t-float-designator (mk-Float-designator())(aps)  $\rho \triangleq$ 
  REAL-TYPE

```

Dynamic Semantics

The call ‘**FLOAT**(**x**)’ shall have the same effect as the call ‘**VAL**(**REAL**,**x**)’.

NOTE — An exception will occur if the value of the expression lies outside the range of the real type.

operations

$$m\text{-float-designator} : \text{Float-designator} \rightarrow \text{Arguments} \rightarrow \text{Environment} \xrightarrow{o} \text{Value}$$

$$m\text{-float-designator}(mk\text{-Float-designator}())(args)\rho \triangleq$$

$$\text{let } [value] = args \text{ in}$$

$$254 \quad converted\text{-result}(\text{REAL-TYPE}, value)\rho$$

6.9.3.7 The Function HIGH

Abstract Syntax

types

High-designator ::

Static Semantics

A call of **HIGH** shall have one actual parameter. The parameter shall either be a variable designator which is an open array parameter or be an indexed designator which denotes an array and which has an array designator that is an open array parameter.

The value returned by a call of **HIGH** shall be of the unsigned type.

functions

$$wf\text{-high-designator} : \text{High-designator} \rightarrow \text{Actual-parameters} \rightarrow \text{Environment} \rightarrow \mathbb{B}$$

$$wf\text{-high-designator}(mk\text{-High-designator}())(aps)\rho \triangleq$$

$$\text{len } aps = 1 \wedge$$

$$\text{let } [var] = aps \text{ in}$$

$$is\text{-Variable-designator}(var) \wedge$$

$$242 \quad wf\text{-high-argument}(var)\rho;$$

$$t\text{-high-designator} : \text{High-designator} \rightarrow \text{Actual-parameters} \rightarrow \text{Environment} \rightarrow \text{Typed}$$

$$t\text{-high-designator}(mk\text{-High-designator}())(aps)\rho \triangleq$$

$$\text{UNSIGNED-TYPE}$$

Dynamic Semantics

A call of **HIGH** shall return the upper bound of the actual parameter. It shall be an exception if this value lies outside the range of the unsigned type.

NOTE — An implementor should consider the result type of **HIGH** when choosing the range of the unsigned type.

Example

Given:

TYPE colour=(red, yellow, green);

VAR a:ARRAY [-7..-2] OF ARRAY BOOLEAN
OF ARRAY colour OF REAL;

PROCEDURE p(f:ARRAY OF ARRAY OF ARRAY OF REAL);

then, for a call ‘p(a)’

expression	result
HIGH(f)	5
HIGH(f[0])	1
HIGH(f[0,0])	2
HIGH(f[0,0,0])	error

operations

```

m-high-designator : High-designator → Arguments → Environment  $\xrightarrow{o}$  Value
m-high-designator (mk-High-designator())(args) $\rho \triangleq$ 
  let [array] = args in
  let value = card (dom array) − 1 in
162   get-whole-result(UNSIGNED-TYPE, value)

```

CHANGE — This International Standard permits multi-dimensional open array parameters whereas these are not allowed by *Programming in Modula-2*. This International Standard permits HIGH to be applied only to an open array parameter, it does not permit HIGH to be applied to an ordinary array variable.

Auxiliary Definitions

functions

```

wf-high-argument : Variable-designator → Environment →  $\mathbb{B}$ 
wf-high-argument (arg) $\rho \triangleq$ 
146   (is-Entire-designator(arg) → let type = t-entire-designator (arg) $\rho$  in
288   is-open-array (type) $\rho$ ,
242   is-Indexed-designator(arg) → wf-high-argument (arg.desig) $\rho$ )

```

6.9.3.8 The Function IM

Abstract Syntax

types

Im-designator :: *rtype* : *Expression-typed*

Static Semantics

A call of **IM** shall have one actual parameter which shall be an expression. If the expression is of the complex type, the value returned shall be of the real type; if the expression is of the long complex type, the value returned shall be of the long real type; otherwise, the expression shall be a constant expression of CC type and the value returned shall be of the RR type

functions

```

wf-im-designator : Im-designator → Actual-parameters → Environment →  $\mathbb{B}$ 
wf-im-designator (mk-Im-designator(rtype))(aps) $\rho \triangleq$ 
  len aps = 1 ∧
  let [expr] = aps in
  is-Expression(expr) ∧
152   let etype = t-expression (expr) $\rho$  in
  ??   is-complex-type (etype) $\rho$  ∧
243   rtype = complex-component-type (etype);

t-im-designator : Im-designator → Actual-parameters → Environment → Expression-typed
t-im-designator (mk-Im-designator(rtype))(aps) $\rho \triangleq$ 
  rtype

```

Auxiliary Definitions

functions

$complex-component-type : Complex-number-type \rightarrow Real-number-type$

$complex-component-type (type) \triangleq$

cases $type$:

COMPLEX-TYPE \rightarrow REAL-TYPE,

LONG-COMPLEX-TYPE \rightarrow LONG-REAL-TYPE,

C-TYPE $\rightarrow \mathbb{R}$ -TYPE

end

Dynamic Semantics

The call '**IM**(**x**)' shall return the imaginary part of the complex number **x**.

NOTE — An exception will occur if the value lies outside the range of values of the result type.

operations

$m-im-designator : Im-designator \rightarrow Arguments \rightarrow Environment \xrightarrow{o} Value$

$m-im-designator (mk-Im-designator(rtype))(args)\rho \triangleq$

let $[value] = args$ in

?? $real-result-type(rtype, im\ value)$

CHANGE — The function **IM** is not in *Programming in Modula-2*.

6.9.3.9 The Function INT

Abstract Syntax

types

$Int-designator ::$

Static Semantics

A call of **INT** shall have one actual parameter which shall be an expression of ordinal type or of a real number type.

The value returned by a call of **INT** shall be of the signed type.

functions

$wf-int-designator : Int-designator \rightarrow Actual-parameters \rightarrow Environment \rightarrow \mathbb{B}$

$wf-int-designator (mk-Int-designator())(aps)\rho \triangleq$

len $aps = 1 \wedge$

let $[expr] = aps$ in

$is-Expression(expr) \wedge$

152 let $etype = t-expression(expr)\rho$ in

280 $is-ordinal-type(etype)\rho \vee is-real-number-type(etype);$

$t-int-designator : Int-designator \rightarrow Actual-parameters \rightarrow Environment \rightarrow Typed$

$t-int-designator (mk-Int-designator())(aps)\rho \triangleq$

SIGNED-TYPE

Dynamic Semantics

The call ‘**INT**(**x**)’ shall have the same effect as ‘**VAL**(**INTEGER**, **x**)’.

operations

$m\text{-int-designator} : \text{Int-designator} \rightarrow \text{Arguments} \rightarrow \text{Environment} \xrightarrow{o} \text{Value}$
 $m\text{-int-designator}(mk\text{-Int-designator}())(args)\rho \triangleq$
let $[value] = args$ in
254 $converted\text{-result}(\text{SIGNED-TYPE}, value)\rho$

CHANGE — The function **INT** is not in *Programming in Modula-2*.

6.9.3.10 The Function **LENGTH**

Abstract Syntax

types

$Length\text{-designator} ::$

Static Semantics

A call of **LENGTH** shall have one actual parameter which shall be an expression of a string type.

The value returned by a call of **LENGTH** shall be of the unsigned type.

functions

$wf\text{-length-designator} : Length\text{-designator} \rightarrow Actual\text{-parameters} \rightarrow Environment \rightarrow \mathbb{B}$
 $wf\text{-length-designator}(mk\text{-Length-designator}())(aps)\rho \triangleq$
len $aps = 1 \wedge$
let $[expr] = aps$ in
 $is\text{-Expression}(expr) \wedge$
152 let $etype = t\text{-expression}(expr)\rho$ in
282 $is\text{-string-type}(etype)\rho \vee$
278 $is\text{-array-type}(etype)\rho \wedge$
284 $component\text{-type-of}(etype)\rho = \text{CHARACTER-TYPE};$
 $t\text{-length-designator} : Length\text{-designator} \rightarrow Actual\text{-parameters} \rightarrow Environment \rightarrow Typed$
 $t\text{-length-designator}(mk\text{-Length-designator}())(aps)\rho \triangleq$
UNSIGNED-TYPE

Dynamic Semantics

The call ‘**LENGTH**(**x**)’ shall return the length of the string **x**.

operations

$m\text{-length-designator} : Length\text{-designator} \rightarrow \text{Arguments} \rightarrow \text{Environment} \xrightarrow{o} \text{Value}$
 $m\text{-length-designator}(mk\text{-Length-designator}())(args)\rho \triangleq$
let $[value] = args$ in
if $\text{END-OF-STRING-CHAR} \in \text{rng } value$
s1d then let $end = \text{mins}(\{i \in \text{dom } value \mid value(i) = \text{END-OF-STRING-CHAR}\})$ in
return end
else return $(\text{card dom } value)$

CHANGE — The function `LENGTH` is not in *Programming in Modula-2*.

6.9.3.11 The Function `LFLOAT`

Abstract Syntax

types

Lfloat-designator ::

Static Semantics

A call of `LFLOAT` shall have one actual parameter which shall be an expression of a number type.

The value returned by a call of `LFLOAT` shall be of the long real type.

functions

wf-lfloat-designator : *Lfloat-designator* \rightarrow *Actual-parameters* \rightarrow *Environment* \rightarrow **B**

wf-lfloat-designator (*mk-Lfloat-designator*())(*aps*) $\rho \triangleq$

len *aps* = 1 \wedge

let [*expr*] = *aps* in

is-Expression(*expr*) \wedge

252 *is-number-type* (*t-expression* (*expr*) ρ);

t-lfloat-designator : *Lfloat-designator* \rightarrow *Actual-parameters* \rightarrow *Environment* \rightarrow *Typed*

t-lfloat-designator (*mk-Lfloat-designator*())(*aps*) $\rho \triangleq$

LONG-REAL-TYPE

Dynamic Semantics

The call '`LFLOAT(x)`' shall have the same effect as the call '`VAL(LONGREAL, x)`'.

NOTE — An exception will occur if the value of the expression lies outside the range of the long real type.

operations

m-lfloat-designator : *Lfloat-designator* \rightarrow *Arguments* \rightarrow *Environment* \xrightarrow{o} *Value*

m-lfloat-designator (*mk-Lfloat-designator*())(*args*) $\rho \triangleq$

let [*value*] = *args* in

254 *converted-result*(LONG-REAL-TYPE, *value*) ρ

CHANGE — The function `LFLOAT` is not in *Programming in Modula-2*.

6.9.3.12 The Function `MAX`

Abstract Syntax

types

Max-designator ::

Static Semantics

A call of `MAX` shall have one actual parameter which shall be a qualified identifier which is the identifier of an ordinal type.

If the host-type of the type named by the actual parameter is a scalar type, the value returned by a call of **MAX** shall be regarded as a literal value; otherwise the value returned shall be of the host-type of the type named by the actual parameter.

Language Clarification

The function **MAX** may be applied to any scalar type (including **REAL** and **LONGREAL**).

functions

```

wf-max-designator : Max-designator → Actual-parameters → Environment → ℬ
wf-max-designator (mk-Max-designator())(aps)ρ  $\triangleq$ 
  len aps = 1 ∧
  let [mk-Type-parameter(qid)] = aps in
270  is-type(qid)ρ ∧
271  let type = type-of(qid)ρ in
280  is-scalar-type(type)ρ ∨ is-bitnum-type(type)ρ;

t-max-designator : Max-designator → Actual-parameters → Environment → Expression-typed
t-max-designator (mk-Max-designator())(aps)ρ  $\triangleq$ 
  let [mk-Type-parameter(qid)] = aps in
  let type = type-of(qid)ρ in
  let host = host-type-of(type)ρ in
  (host ∈ {UNSIGNED-TYPE, SIGNED-TYPE} → ℤ-TYPE,
   host ∈ {REAL-TYPE, LONG-REAL-TYPE} → ℝ-TYPE,
   others → host)

```

Dynamic Semantics

A call of **MAX** shall return the largest value of the type named by the actual parameter.

NOTE — Example

Given:

TYPE percentage=CARDINAL[0..100];

TYPE countdown=INTEGER[-9..-1];

TYPE colours=(red, yellow, green);

TYPE coward=colours[yellow..yellow];

operations

then:

expression	value	type
MAX(percentage)	100	ZZ-type
MAX(countdown)	-1	ZZ-type
MAX(colours)	green	colours
MAX(coward)	yellow	colours

‘MAX(CARDINAL)’ and ‘MAX(INTEGER)’ are also of ZZ type.

```

m-max-designator : Max-designator → Arguments → Environment  $\xrightarrow{o}$  Value
m-max-designator (mk-Max-designator())(args)ρ  $\triangleq$ 
  let [type] = args in
??  maximum(type) ρ

```

6.9.3.13 The Function MIN

Abstract Syntax

types

Min-designator ::

Static Semantics

A call of **MIN** shall have one actual parameter which shall be a qualified identifier which is the identifier of an ordinal type.

If the host-type of the type named by the actual parameter is a scalar type, the value returned by a call of **MIN** shall be regarded as a literal value; otherwise the value returned shall be of the host-type of the type named by the actual parameter.

Language Clarification

The function **MIN** may be applied to any scalar type (including **REAL** and **LONGREAL**).

functions

$wf-min-designator : Min-designator \rightarrow Actual-parameters \rightarrow Environment \rightarrow \mathbb{B}$

$wf-min-designator(mk-Min-designator())(aps)\rho \triangleq$

len $aps = 1 \wedge$

let $[mk-Type-parameter(qid)] = aps$ in

270 $is-type(qid)\rho \wedge$

271 let $type = type-of(qid)\rho$ in

280 $is-scalar-type(type)\rho \vee is-bitnum-type(type)\rho;$

$t-min-designator : Min-designator \rightarrow Actual-parameters \rightarrow Environment \rightarrow Expression-typed$

$t-min-designator(mk-Min-designator())(aps)\rho \triangleq$

let $[mk-Type-parameter(qid)] = aps$ in

271 let $type = type-of(qid)\rho$ in

283 let $host = host-type-of(type)\rho$ in

$(host \in \{UNSigned-TYPE, Signed-TYPE\} \rightarrow \mathbb{Z}\text{-TYPE},$

$host \in \{REAL\text{-TYPE}, LONG\text{-REAL}\text{-TYPE}\} \rightarrow \mathbb{R}\text{-TYPE},$

others $\rightarrow host)$

Dynamic Semantics

A call of **MIN** shall return the smallest value of the type named by the actual parameter.

NOTE — Example

Given:

TYPE percentage=CARDINAL[0..100];

TYPE countdown=INTEGER[-9..-1];

TYPE colours=(red, yellow, green);

TYPE coward=colours[yellow..yellow];

then:

expression	value	type
MIN(percentage)	0	ZZ type
MIN(countdown)	-9	ZZ type
MIN(colours)	red	colours
MIN(coward)	yellow	colours

'MIN(CARDINAL)' and 'MIN(INTEGER)' are also of ZZ type.

operations

$m-min-designator : Min-designator \rightarrow Arguments \rightarrow Environment \xrightarrow{o} Value$

$m-min-designator(mk-Min-designator())(args)\rho \triangleq$

let $[type] = args$ in

?? $minimum(type)\rho$

6.9.3.14 The Function ODD

Abstract Syntax

types

Odd-designator ::

Static Semantics

A call of **ODD** shall have one actual parameter which shall be an expression of a whole number type.

The value returned by a call of **ODD** shall be of the Boolean type.

functions

wf-odd-designator : *Odd-designator* \rightarrow *Actual-parameters* \rightarrow *Environment* $\rightarrow \mathbb{B}$
wf-odd-designator (*mk-Odd-designator*())(*aps*) $\rho \triangleq$
 len *aps* = 1 \wedge
 let [*expr*] = *aps* in
 is-Expression(*expr*) \wedge
252 *is-whole-number-type* (*t-expression* (*expr*) ρ);

t-odd-designator : *Odd-designator* \rightarrow *Actual-parameters* \rightarrow *Environment* \rightarrow *Typed*
t-odd-designator (*mk-Odd-designator*())(*aps*) $\rho \triangleq$
 BOOLEAN-TYPE

Dynamic Semantics

A call of **ODD** shall return the value **TRUE** if the absolute value of the actual parameter is odd; otherwise the value **FALSE** shall be returned.

operations

m-odd-designator : *Odd-designator* \rightarrow *Arguments* \rightarrow *Environment* \xrightarrow{o} *Value*
m-odd-designator (*mk-Odd-designator*())(*args*) $\rho \triangleq$
 let [*arg*] = *args* in
 return $\exists n \in \mathbb{Z} \cdot 2 \times n + 1 = \text{arg}$

6.9.3.15 The Function ORD

Abstract Syntax

types

Ord-designator ::

Static Semantics

A call of **ORD** shall have one actual parameter which shall be an expression of an ordinal type.

The value returned by a call of **ORD** shall be of the unsigned type.

functions

$wf\text{-}ord\text{-}designator : Ord\text{-}designator \rightarrow Actual\text{-}parameters \rightarrow Environment \rightarrow \mathbb{B}$

$wf\text{-}ord\text{-}designator(mk\text{-}Ord\text{-}designator())(aps)\rho \triangleq$

$\text{len } aps = 1 \wedge$

$\text{let } [expr] = aps \text{ in}$

$is\text{-}Expression(expr) \wedge$

250 $is\text{-}ordinal\text{-}type(t\text{-}expression(expr)\rho);$

$t\text{-}ord\text{-}designator : Ord\text{-}designator \rightarrow Actual\text{-}parameters \rightarrow Environment \rightarrow Typed$

$t\text{-}ord\text{-}designator(mk\text{-}Ord\text{-}designator())(aps)\rho \triangleq$

UNSIGNED-TYPE

Dynamic Semantics

The call ‘**ORD**(**x**)’ shall have the same effect as the call ‘**VAL**(**CARDINAL**, **x**)’.

NOTE — An exception will occur if the ordinal number corresponding to the value of the expression lies outside the range of the unsigned type.

operations

$m\text{-}ord\text{-}designator : Ord\text{-}designator \rightarrow Arguments \rightarrow Environment \xrightarrow{o} Value$

$m\text{-}ord\text{-}designator(mk\text{-}Ord\text{-}designator())(args)\rho \triangleq$

$\text{let } [value] = args \text{ in}$

254 $converted\text{-}result(UNSIGNED\text{-}TYPE, value) \rho$

6.9.3.16 The PROT Function

Abstract Syntax

types

$Prot\text{-}designator ::$

Static Semantics

A call of **PROT** shall have no parameters.

The value returned by a call of **PROT** shall be of the protection type.

functions

$wf\text{-}prot\text{-}designator : Prot\text{-}designator \rightarrow Actual\text{-}parameters \rightarrow Environment \rightarrow \mathbb{B}$

$wf\text{-}prot\text{-}designator(mk\text{-}Prot\text{-}designator())(aps)\rho \triangleq$

$\text{len } aps = 0;$

$t\text{-}prot\text{-}designator : Prot\text{-}designator \rightarrow Actual\text{-}parameters \rightarrow Environment \rightarrow Typed$

$t\text{-}prot\text{-}designator(mk\text{-}Prot\text{-}designator())(aps)\rho \triangleq$

PROTECTION-TYPE

Dynamic Semantics

A call of **PROT** shall return the value of the current protection.

operations

$$m\text{-prot-designator} : \text{Prot-designator} \rightarrow \text{Arguments} \rightarrow \text{Environment} \xrightarrow{o} \text{Value}$$

$$m\text{-prot-designator}(mk\text{-Prot-designator}())(args)\rho \triangleq$$

$$current\text{-protection}()$$

303

CHANGE — The function PROT is not in *Programming in Modula-2*.

6.9.3.17 The Function RE

Abstract Syntax

types

$$Re\text{-designator} :: rtype : \text{Expression-typed}$$

Static Semantics

A call of **RE** shall have one actual parameter which shall be an expression. If the expression is of the complex type, the value returned shall be of the real type; if the expression is of the long complex type, the value returned shall be of the long real type; otherwise, the expression shall be a constant expression of CC type and the value returned shall be of the RR type

functions

$$wf\text{-re-designator} : Re\text{-designator} \rightarrow \text{Actual-parameters} \rightarrow \text{Environment} \rightarrow \mathbb{B}$$

$$wf\text{-re-designator}(mk\text{-Re-designator}(rtype))(aps)\rho \triangleq$$

$$\begin{aligned} & \text{len } aps = 1 \wedge \\ & \text{let } [expr] = aps \text{ in} \\ & is\text{-Expression}(expr) \wedge \\ 152 & \text{let } etype = t\text{-expression}(expr)\rho \text{ in} \\ 77 & is\text{-complex-type}(etype)\rho \wedge \\ 243 & rtype = complex\text{-component-type}(etype); \end{aligned}$$

$$t\text{-re-designator} : Re\text{-designator} \rightarrow \text{Actual-parameters} \rightarrow \text{Environment} \rightarrow \text{Expression-typed}$$

$$t\text{-re-designator}(mk\text{-Re-designator}(rtype))(aps)\rho \triangleq$$

$$rtype$$

Dynamic Semantics

The call '**RE(x)**' shall return the real part of the complex number **x**.

NOTE — An exception will occur if the value lies outside the range of values of the result type.

operations

$$m\text{-re-designator} : Re\text{-designator} \rightarrow \text{Arguments} \rightarrow \text{Environment} \xrightarrow{o} \text{Value}$$

$$m\text{-re-designator}(mk\text{-Re-designator}(rtype))(args)\rho \triangleq$$

$$\begin{aligned} & \text{let } [value] = args \text{ in} \\ 77 & real\text{-result-type}(rtype, re\ value) \end{aligned}$$

CHANGE — The function RE is not in *Programming in Modula-2*.

6.9.3.18 The Function SIZE

Abstract Syntax

types

$Size\text{-}designator :: type : Expression\text{-}typed$

Static Semantics

A call of **SIZE** shall have one actual parameter which shall be either an entire designator which is not an open array parameter or a qualified identifier which is the identifier of a type.

The value returned by a call of **SIZE** shall be a whole number literal value.

NOTES

- 1 If the parameter to **SIZE** is a variable, it must be an entire designator. Thus, it cannot be a component of an array, a component of a record, or the variable referenced by a pointer variable.
- 2 A call of **SIZE** is permitted in a constant expression.

functions

$wf\text{-}size\text{-}designator : Size\text{-}designator \rightarrow Actual\text{-}parameters \rightarrow Environment \rightarrow \mathbb{B}$

$wf\text{-}size\text{-}designator (mk\text{-}Size\text{-}designator(type))(aps)\rho \triangleq$

$\text{len } aps = 1 \wedge$

$\text{cases } aps :$

$[mk\text{-}Entire\text{-}designator()] \rightarrow type = t\text{-}entire\text{-}designator(ap)\rho \wedge$

$\neg is\text{-}open\text{-}array(type)\rho,$

$[mk\text{-}Type\text{-}parameter(qid)] \rightarrow is\text{-}type(qid)\rho \wedge$

$type = type\text{-}of(qid)\rho$

$\text{end};$

$t\text{-}size\text{-}designator : Size\text{-}designator \rightarrow Actual\text{-}parameters \rightarrow Environment \rightarrow Expression\text{-}typed$

$t\text{-}size\text{-}designator (mk\text{-}Size\text{-}designator(type))(aps)\rho \triangleq$

$\mathbb{Z}\text{-TYPE}$

Dynamic Semantics

A call of **SIZE** where the parameter is a type shall return the number of locations (see 7.1.2.1) which would be occupied by a variable of that type. If the parameter is a record type with variant components, the value returned shall be the maximum number of locations occupied. It shall be an exception if the value returned lies outside the range of the whole number literal values.

A call of **SIZE** where the parameter is an entire designator shall have the same effect as a call of **SIZE** with an actual parameter which is the type of the entire designator.

An implementation shall provide sufficient information to determine the size of all types.

NOTE — It is illegal for the parameter to **SIZE** to be the name of a formal parameter which is an open array parameter. The **SIZE** of any other formal parameter is equal to the **SIZE** of the type of the formal parameter.

operations

$m\text{-}size\text{-}designator : Size\text{-}designator \rightarrow Arguments \rightarrow Environment \xrightarrow{o} Value$

$m\text{-}size\text{-}designator (mk\text{-}Size\text{-}designator(type))(args)\rho \triangleq$

$\text{let } size = tsize(type, [])\rho \text{ in}$

$get\text{-}whole\text{-}result(\mathbb{Z}\text{-TYPE}, size)$

6.9.3.19 The Function TRUNC

Abstract Syntax

types

Trunc-designator ::

Static Semantics

A call of **TRUNC** shall have one actual parameter which shall be an expression of a real number type.

The value returned by a call of **TRUNC** shall be of the unsigned type.

functions

wf-trunc-designator : *Trunc-designator* \rightarrow *Actual-parameters* \rightarrow *Environment* $\rightarrow \mathbb{B}$

wf-trunc-designator (*mk-Trunc-designator* ()) (*aps*) $\rho \triangleq$

len *aps* = 1 \wedge

let [*expr*] = *aps* in

is-Expression (*expr*) \wedge

152 *is-Real-number-type* (*t-expression* (*expr*) ρ);

t-trunc-designator : *Trunc-designator* \rightarrow *Actual-parameters* \rightarrow *Environment* \rightarrow *Typed*

t-trunc-designator (*mk-Trunc-designator* ()) (*aps*) $\rho \triangleq$

UNSIGNED-TYPE

Dynamic Semantics

The call ‘**TRUNC** (**x**)’ shall have the same effect as the call ‘**VAL**(**CARDINAL**, **x**)’.

NOTE — An exception will occur if the integral part of the expression lies outside the range of the unsigned type.

operations

m-trunc-designator : *Trunc-designator* \rightarrow *Arguments* \rightarrow *Environment* \xrightarrow{o} *Value*

m-trunc-designator (*mk-Trunc-designator* ()) (*args*) $\rho \triangleq$

let [*value*] = *args* in

254 *converted-result*(UNSIGNED-TYPE, *value*) ρ

6.9.3.20 The Function VAL

Abstract Syntax

types

Val-designator ::

Static Semantics

A call of **VAL** shall have two actual parameters. The first parameter shall be a qualified identifier which is the identifier of a type, called the ‘result-type’, and the second parameter shall be an expression. The result-type shall be an ordinal type or a real number type. The type of the expression shall be an ordinal type or a real number type. At least one of the following shall hold:

- a) The type of the expression and the host type of the result-type are identical types.

- b) The type of the expression and the result-type are both number types.
- c) The host type of the result-type is a whole number type.
- d) The type of the expression is a whole number type.

If the result-type is a subrange type, the value returned by a call of **VAL** shall be of the host type of the subrange type; otherwise, it shall be of the result-type.

NOTE — If t is some enumerated type, in the following table: \checkmark denotes a valid combination of types, and \times denotes an invalid combination:

The type of the expression	result type						
	CARDINAL*	INTEGER*	REAL	LONGREAL	CHAR*	BOOLEAN*	the type t^*
CARDINAL	\checkmark	\checkmark	\checkmark	\checkmark	\checkmark	\checkmark	\checkmark
INTEGER	\checkmark	\checkmark	\checkmark	\checkmark	\checkmark	\checkmark	\checkmark
ZZ-type	\checkmark	\checkmark	\checkmark	\checkmark	\checkmark	\checkmark	\checkmark
REAL	\checkmark	\checkmark	\checkmark	\checkmark	\times	\times	\times
LONGREAL	\checkmark	\checkmark	\checkmark	\checkmark	\times	\times	\times
RR-type	\checkmark	\checkmark	\checkmark	\checkmark	\times	\times	\times
CHAR \dagger	\checkmark	\checkmark	\times	\times	\checkmark	\times	\times
BOOLEAN	\checkmark	\checkmark	\times	\times	\times	\checkmark	\times
the type t	\checkmark	\checkmark	\times	\times	\times	\times	\checkmark

* or a subrange of this type

\dagger or SS-type of length 1

functions

$wf\text{-}val\text{-}designator : Val\text{-}designator \rightarrow Actual\text{-}parameters \rightarrow Environment \rightarrow \mathbb{B}$

$wf\text{-}val\text{-}designator(mk\text{-}Val\text{-}designator())(aps)\rho \triangleq$

$\text{len } aps = 2 \wedge$

$\text{let } [type, expr] = aps \text{ in}$

$is\text{-}Expression(expr) \wedge$

152 $\text{let } etype = t\text{-}expression(expr)\rho \text{ in}$

280 $is\text{-}scalar\text{-}type(etype)\rho \wedge$

283 $\text{let } htype = host\text{-}type\text{-}of(type)\rho \text{ in}$

$(htype = etype$

\vee

281 $is\text{-}number\text{-}type(type)\rho \wedge is\text{-}number\text{-}type(etype)\rho$

\vee

281 $is\text{-}whole\text{-}number\text{-}type(htype)\rho$

\vee

281 $is\text{-}whole\text{-}number\text{-}type(etype)\rho);$

$t\text{-}val\text{-}designator : Val\text{-}designator \rightarrow Actual\text{-}parameters \rightarrow Environment \rightarrow Typed$

$t\text{-}val\text{-}designator(mk\text{-}Val\text{-}designator())(aps)\rho \triangleq$

$\text{let } [type, -] = aps \text{ in}$

283 $host\text{-}type\text{-}of(type)\rho$

Dynamic Semantics

If the result-type is a real number type, a call of **VAL** shall return the value of the result-type which has the same numerical value as the value of the expression. It shall be an exception if the value of the expression lies outside the range of the result-type.

If the host type of the result-type is a whole number type and the type of the expression is a number type, a call of **VAL** shall return the integral part of the value of the expression. It shall be an exception if the integral part lies outside the range of the result-type.

If the result-type or the type of the expression or both of these types are not number types, a call of **VAL** shall return the value of the result-type which has an ordinal number equal to the ordinal number of the value of the expression. It shall be an exception if the value of the expression is a numerical value that is negative or if the ordinal number of the value of the expression lies outside the range of the ordinal numbers of the result-type.

NOTE — **VAL** does not round: If *r* is of the real type, the expressions ‘**VAL**(**INTEGER**,*r*)’ and ‘**VAL**(**CARDINAL**,*r*)’ both have the value of *r* truncated.

Example

Given: then:

TYPE Days=(Sun,Mon,Tue,Wed,Thu,Fri,Sat);

TYPE WeekDays=[Mon..Fri];

VAR i, j, z:**INTEGER**;

VAR r:**REAL**;

and:

i:= 42; j:= -1; z:= 0; r:= -2.7;

expression	value	type
VAL (CARDINAL ,i)	42	CARDINAL
VAL (CARDINAL ,j)	Exception	
VAL (CARDINAL ,r)	Exception	
VAL (INTEGER ,i)	42	INTEGER
VAL (INTEGER ,r)	-2	INTEGER
VAL (REAL ,i)	42.0	REAL
VAL (REAL ,TRUE)	Error	
VAL (LONGREAL ,r)	-2.7	LONGREAL
VAL (CHAR ,z)	0C	CHAR
VAL (CHAR ,r)	Error	
VAL (BOOLEAN ,0)	FALSE	BOOLEAN
VAL (Days,5)	Fri	Days
VAL (WeekDays,5)	Fri	Days
VAL (WeekDays,z)	Exception	

operations

$m\text{-val-designator} : \text{Val-designator} \rightarrow \text{Arguments} \rightarrow \text{Environment} \xrightarrow{o} \text{Value}$

$m\text{-val-designator}(mk\text{-Val-designator}())(args)\rho \triangleq$

let [*type*, *value*] = *args* in

254 *converted-result*(*type*, *value*) ρ

CHANGE — The definition of **VAL** given in *Programming in Modula-2* has been extended.

6.9.4 Auxiliary Definitions

operations

$converted\text{-result} : \text{Typed} \times \text{Value} \rightarrow \text{Environment} \xrightarrow{o} \text{Value}$

$converted\text{-result}(type, value)\rho \triangleq$

283 let *rtype* = *host-type-of* (*type*) ρ in

if *is-Real-number-type*(*rtype*)

159 then *get-real-result*(*rtype*, *scalar-value* (*value*))

281 elseif *is-whole-number-type* (*rtype*) ρ

255 then let *tvalue* = *trunc* (*scalar-value* (*value*)) in

162 *get-whole-result*(*rtype*, *tvalue*)

255 else let *tvalue* = *trunc* (*scalar-value* (*value*)) in — ordinal type other than whole number type

291 let *values* = *ordered-values-of* (*rtype*) ρ in

if *tvalue* + 1 \in inds *values*

?? then let *value* = *values* (*tvalue* + 1) ρ in

290 if *value* \in *values-of* (*type*) ρ

then return *value*

306 else *mandatory-exception*(CONVERSIONERROR)

306 else *mandatory-exception*(CONVERSIONERROR)

annotations Carry out safe conversion; convert the argument to a corresponding value of another type.

functions

$$\text{trunc} : \mathbb{R} \rightarrow \mathbb{Z}$$

$$\begin{aligned} \text{trunc}(r) &\triangleq \\ &\text{if } r \geq 0 \\ &\text{then let } n \in \mathbb{N} \text{ be st } n \leq r < n + 1 \text{ in} \\ &\quad n \\ 255 \quad &\text{else } -\text{trunc}(-r) \end{aligned}$$

annotations Return the integral part of a given numerical value.

6.9.5 Constant Definitions

values

$$\begin{aligned} \text{required-lower-case-letters} &= [a', b', c', d', e', f', g', h', i', j', k', l', m', \\ &\quad n', o', p', q', r', s', t', u', v', w', x', y', z']; \\ \text{required-upper-case-letters} &= [A', B', C', D', E', F', G', H', I', J', K', L', M', \\ &\quad N', O', P', Q', R', S', T', U', V', W', X', Y', Z']; \\ \text{required-capitalisations} &= \{\text{required-lower-case-letters}(i) \xrightarrow{m} \text{required-upper-case-letters}(i) \mid \\ &\quad i \in \text{inds required-lower-case-letters}\}; \\ \text{implementation-defined-capitalisations} &= : \text{char} \xrightarrow{m} \text{char}; \\ \text{capitalisations} &= \text{required-capitalisations} \cup \text{implementation-defined-capitalisations} \end{aligned}$$

NOTE — Defining *capitalisations* in this way avoids a dependency on the collating sequence of the character values. *implementation-defined-capitalisations* is a means by which implementations may have extra capitalisation rules (e.g. ‘ γ ’ \rightarrow ‘ Γ ’ or ‘ \ddot{u} ’ \rightarrow ‘ \ddot{U} ’).

values

$$\begin{aligned} \text{digits} &= [0', 1', 2', 3', 4', 5', 6', 7', 8', 9']; \\ \text{other-required-characters} &= \{c \mid c \in \text{required symbols}\}; \\ \text{required-characters} &= \text{elems required-lower-case-letters} \cup \\ &\quad \text{elems required-upper-case-letters} \cup \\ &\quad \text{elems digits} \cup \\ &\quad \text{other-required-characters}; \\ \text{collation-map} &= \text{required-collation-map} \cup \\ &\quad \text{implementation-defined-collation-map}; \\ \text{required-collation-map} &= \text{required-characters} \xrightarrow{m} \mathbb{N}; \\ \text{implementation-defined-collation-map} &= \text{Char} \xrightarrow{m} \mathbb{N} \end{aligned}$$

NOTE — required symbols is defined in 5.7.

Properties

$$\begin{aligned} \forall i \in \{1, \dots, 9\} \cdot \\ &\quad \text{required-collation-map}(\text{digits}(i + 1)) = \text{required-collation-map}(\text{digits}(i)) + 1 \\ \forall i \in \{1, \dots, 25\} \cdot \\ &\quad \text{required-collation-map}(\text{required-upper-case-letters}(i + 1)) > \text{required-collation-map}(\text{required-upper-case-letters}(i)) \\ \forall i \in \{1, \dots, 25\} \cdot \\ &\quad \text{required-collation-map}(\text{required-lower-case-letters}(i + 1)) > \text{required-collation-map}(\text{required-lower-case-letters}(i)) \end{aligned}$$

6.10 The Environment – Auxiliary Functions

6.10.1 Domains

6.10.1.1 Environments

types

```
Environment :: consts   : Constants
                types    : Types
                strucs   : Structures
                vars     : Variables
                procs    : Procedures
                mods     : Modules
                level    :  $[\mathbb{N}]$ 
                domain :  $[Protection\text{-}domain]$ 
                dens     :  $[Procdens]$ 
                conts    :  $[Continuation]$  ;
```

$Constants = Identifier \xrightarrow{m} Constant\text{-}value$

annotations The declaration of an enumerated type will cause the identifiers that denote the values defined by that type to be added to the constant component of the environment.

types

$Types = Identifier \xrightarrow{m} Typed;$

$Structures = Type\text{-}name \xrightarrow{m} Structure;$

$Variables = Identifier \xrightarrow{m} (Variable\text{-}typed \mid Variable);$

$Procedures = Identifier \xrightarrow{m} Procedure\text{-}id;$

$Modules = Identifier \xrightarrow{m} Module\text{-}environment;$

$Procdens = Procedure\text{-}id \xrightarrow{m} (Procedure\text{-}typed \mid Program\text{-}cont);$

$Continuation = Exit \xrightarrow{m} Program\text{-}cont$

annotations The environment associates an identifier with an object and with the properties of that object.

For an identifier that denotes a constant object the property is its type.

For an identifier that denotes a type, the object is either a basic type or a unique type name. For objects that are of a structured type the environment associates the type name with the structure of that type, and for elementary types the environment associates the type name with the details of their construction.

For an identifier that denotes a variable, when checking the well-formed properties of a program, the property of the object is its type; when giving a meaning to the program the object is the associated variable.

For an identifier that denotes a procedure (either standard or defined by a declaration), when checking the well-formed properties of a program, the property of the object is its type; when giving a meaning to the program the object is the parameters, rules for binding formal to actual parameters, and the declarations and body of the procedure.

For an identifier that denotes a module, the object is the environment defined by the declarations contained in the module as restricted by any exports.

The environment contains a count of the level of static nesting of procedures.

The environment contains the denotations for procedures which are program continuations.

The environment contains the static continuations; the continuations for exit, return, and halt.

6.10.1.2 Constant Values

types

$$\begin{array}{l} \textit{Constant-value} :: \textit{type} : \textit{Expression-typed} \\ \textit{value} : \textit{Value} \end{array}$$

6.10.1.3 Types

types

$$\textit{Typed} = \textit{Basic-type} \mid \textit{Type-name} \mid \textit{System-storage-type};$$

$$\textit{Type-name} = \textit{proc-type} \mid \textit{TOKEN}$$

annotations The declaration of a new type will create a new type name that is used to denote that new type. This type name is unique within a program.

types

$$\textit{Basic-type} = \textit{Number-type} \mid \text{BOOLEAN-TYPE} \mid \text{CHARACTER-TYPE} \mid \text{NIL-TYPE} \mid \text{PROTECTION-TYPE} \mid \text{COROUTINE-TYPE};$$

$$\textit{Number-type} = \textit{Whole-number-type} \mid \textit{Real-number-type};$$

$$\textit{Whole-number-type} = \text{UNSIGNED-TYPE} \mid \text{SIGNED-TYPE} \mid \mathbb{Z}\text{-TYPE};$$

$$\textit{Real-number-type} = \text{REAL-TYPE} \mid \text{LONG-REAL-TYPE} \mid \mathbb{R}\text{-TYPE};$$

$$\textit{String-type} :: \textit{length} : \mathbb{N};$$

System-storage-type = LOC-TYPE | MACHINE-ADDRESS-TYPE | BITNUM-TYPE

6.10.1.4 Structures

types

Structure = *Array-structure* | *Record-structure* | *Procedure-structure* | *Set-structure*
 | *Pointer-structure* | OPAQUE | *Enumerated-structure* | *Subrange-structure*;

Array-structure :: *itype* : *Typed*
 ctype : *Typed*

annotations The structure of an array type are the type names or basic types of the index and component types.

types

Record-structure :: *fields* : *Fields-list-structure*

annotations The structure of an array type are the type names or basic types of the components of the record.

types

Fields-list-structure = *Fields-structure**;

Fields-structure = *Fixed-fields-structure* | *Variant-fields-structure*;

Fixed-fields-structure :: *ids* : *Identifier**
 type : *Typed* ;

Variant-fields-structure :: *tag* : [*Identifier*]
 tagt : *Typed*
 variants : *Variant-structure**
 other : *Fields-list-structure* ;

Variant-structure :: *labels* : *Set-value*
 fields : *Fields-list-structure* ;

Procedure-structure = *Proper-procedure-structure* | *Function-procedure-structure*;

Proper-procedure-structure :: *parms* : *Formal-parameters-typed*

annotations The structure of a proper procedure are the type names or basic types of the formal parameters of the procedure.

types

Function-procedure-structure :: *parms* : *Formal-parameters-typed*
 return : [*Typed*]

annotations The structure of a function procedure are the type names or basic types of the formal parameters of the procedure, together with the the type names or basic types of the result. The return type is optional because forward use of a type name is allowed in the context of a procedure type.

types

Formal-parameters-typed = *Formal-parameter-typed**;

Formal-parameter-typed = *Value-formal-typed* | *Variable-formal-typed*;

Value-formal-typed :: *type* : *Variable-typed* ;

Variable-formal-typed :: *type* : *Variable-typed* ;

Set-structure :: *btype* : *Typed*

annotations The structure of a set type is the type name or basic type of the base type.

types

Pointer-structure :: *type* : [*Typed*]

annotations The structure of a pointer type is the the type name or basic type of the bound type. The bound type is optional because forward use of a type name is allowed in the context of a pointer.

types

Enumerated-structure :: *values* : *Identifier**

annotations The structure of an enumerated type is the sequence of identifiers defined by the type, the order of the identifiers in the sequence is identical to their order in the type definition.

types

Subrange-structure :: *rtype* : *Typed*
 range : *Set-value*

annotations The structure of a subrange type is the range type and a set of values defined by the lower and upper bounds of the type definition.

types

Value-structure = *Structure* | *Open-array-typed* | *Procedure-typed* | *String-type*

6.10.1.5 Variable Types

types

$$\textit{Variable-typed} = [\textit{Typed}] \mid \textit{Open-array-typed}$$

annotations A variable can either be declared or can be a parameter of a procedure, thus it will either be associated with its type name or its basic type, or can be an open array parameter. The type is optional here because forward use of a type name is allowed in the context of a procedure type.

types

$$\textit{Open-array-typed} :: \textit{type} : \textit{Component-typed} ;$$
$$\textit{Component-typed} = \textit{Variable-typed}$$

6.10.1.6 Expression Types

types

$$\textit{Expression-typed} = \textit{Typed} \mid \textit{String-type} \mid \textit{Procedure-typed} \mid \textit{Open-array-typed}$$

annotations The result of an expression can be:

- a value of a basic type or a value whose type has been defined in a declaration;
- a string constant;
- a procedure constant.

6.10.1.7 Procedure Types

types

$$\textit{Procedure-typed} = \textit{Procedure-const} \mid \textit{Standard-procedure};$$
$$\textit{Procedure-const} :: \textit{proc} : \textit{Procedure-structure} \\ \textit{level} : \mathbb{N} ;$$
$$\textit{Standard-procedure} = \text{STANDARD-PROPER-PROCEDURE} \mid \text{STANDARD-FUNCTION-PROCEDURE}$$

6.10.1.8 Module Environments

types

$$\textit{Module-environment} = \textit{Unqualified-environment} \mid \textit{Qualified-environment};$$

Unqualified-environment :: *env* : *Environment* ;

Qualified-environment :: *env* : *Environment*

6.10.1.9 Variables

types

Variable = *Elementary-variable* | *Array-variable* | *Record-variable* | *Tag-variable* | *Variant-variable* | *Tagged-variable*;

Loc = An infinite set of tokens;

Elementary-variable :: *loc* : *Loc*
 range : [*Set-value*]

annotations If the identifier associated with an elementary variable is of subrange type, then the range component contains the possible set of values that can be assigned to the variable.

types

Array-variable :: *vars* : *Array-components* ;

Array-components = *Ordinal-value* \xrightarrow{m} *Variable*

inv *ac* \triangleq
 $\forall a \in ac \cdot$
 $\forall rx, ry \in \text{rng } a \cdot$
is-same-variable-type(*rx*, *ry*)

annotations The components of an array variable must all be of the same type.

types

Record-variable :: *fields* : *Record-components* ;

Record-components = *Identifier* \xrightarrow{m} *Variable*;

Tag-variable :: *var* : *Elementary-variable*
 variants : *Variant-component-set*

annotations A tag variable contains information about the variant components that are associated with it. Thus any change to the value of the tag which will cause a different component to be selected can be reflected by setting the variables of the old component to undefined.

types

Variant-variable :: *var* : *Variable*
 tagloc : *Elementary-variable*
 tagvals : *Value-set*

annotations A variant variable contains information about the explicit tag associated with the variant fields of which the variable is a component; thus before accessing the value of the variable the tag can be checked to see if its value is such that the variant that contains the variable is active.

types

Tagged-variable :: *var* : *Variable*
 tagvar : *Tag-variable*
 tagvals : *Value-set*

annotations A tagged variable contains information about the implicit tag associated with the variant fields of which the variable is a component; thus before accessing the value of the variable the tag can be checked to see if its value is such that the variant that contains the variable is active.

types

Variant-component :: *labels* : *Value-set*
 fields : *Record-components*

6.10.1.10 Protection Domains

types

Protection-domain :: *entry* : *Program-cont*
 leave : *Program-cont*
 val : *Value*

6.10.1.11 Procedure Denotations

types

Procedure-id = *TOKEN*;

Program-cont = *Cmdcont* \rightarrow *Coroutine-env* \rightarrow *Statecont*

6.10.1.12 Local Continuations

types

Exit = RETURN | TERMINATION | HALT

6.10.2 Environments

types

$$\begin{aligned} \textit{Environment} &:: \textit{env} : \textit{Associations} \\ &\quad \textit{strucs} : \textit{Structures} \end{aligned}$$

annotations The environment associates an identifier with an object and with the properties of that object.

For an identifier that denotes a type, the object is either a basic type or a unique type name. For objects that are of a structured type the environment associates the type name with the structure of that type, and for elementary types the environment associates the type name with the details of their construction.

types

$$\textit{Associations} = \textit{Static-environment} \mid \textit{Dynamic-environment};$$
$$\textit{Structures} = \textit{Type-name} \xrightarrow{m} \textit{Structure};$$
$$\begin{aligned} \textit{Static-environment} &:: \textit{ids} : \textit{Identifier} \xrightarrow{m} \textit{Property} \\ &\quad \textit{level} : [\mathbb{N}]; \end{aligned}$$
$$\textit{Property} = \textit{Constant-value} \mid \textit{Typed} \mid \textit{Variable-typed} \mid \textit{Procedure-typed} \mid \textit{Environment}$$

The static environment associates an identifier with the properties of an object.

For an identifier that denotes a constant object the property is its type and its value.

For an identifier that denotes a variable the property of the object is its type.

For an identifier that denotes a procedure (either standard or defined by a declaration the property of the object is its type.

For an identifier that denotes a module, the object is the environment defined by the declarations contained in the module as restricted by any exports.

The environment contains a count of the level of static nesting of procedures.

The declaration of an enumerated type will cause the identifiers that denote the values defined by that type to be added to the constant component of the environment.

types

$$\begin{aligned} \textit{Dynamic-environment} &:: \textit{consts} : \textit{Identifier} \xrightarrow{m} \textit{Object} \\ &\quad \textit{domain} : \textit{Protection-domain} \\ &\quad \textit{dens} : \textit{Procdens} \\ &\quad \textit{conts} : \textit{Continuation} \end{aligned}$$

The dynamic environment associates an identifier with an object.

types

$$Object = Constant-value \mid Typed \mid Variable \mid Procedure-id \mid Environment$$

For an identifier that denotes a constant object the object is its type and value.

For an identifier that denotes a variable the object is the associated variable.

For an identifier that denotes a procedure the object is the parameters, rules for binding formal to actual parameters, and the declarations and body of the procedure.

For an identifier that denotes a module, the object is the environment defined by the declarations contained in the module as restricted by any exports.

The environment contains the denotations for procedures which are program continuations.

The environment contains the static continuations; the continuations for exit, return, and halt.

types

$$Procdens = Procedure-id \xrightarrow{m} Program-cont;$$

$$Continuation = Exit \xrightarrow{m} Program-cont$$

6.10.3 Constant Values

types

$$\begin{array}{l} Constant-value :: type : Expression-typed \\ \quad \quad \quad value : Value \end{array}$$

6.10.3.1 Types

types

$$Typed :: id : Types ;$$

$$Types = Basic-type \mid Type-name \mid System-storage-type \mid proc-type;$$

$$Type-name = TOKEN$$

annotations	The declaration of a new type will create a new type name that is used to denote that new type. This type name is unique within a program.
-------------	--

types

$$Basic-type = Number-type \mid Boolean-type \mid character-type \mid nil-type \mid protection-type \mid coroutine-type;$$

$Number\text{-}type = Whole\text{-}number\text{-}type \mid Real\text{-}number\text{-}type;$

$Whole\text{-}number\text{-}type = unsigned\text{-}type \mid signed\text{-}type \mid \mathbb{Z}\text{-}type;$

$Real\text{-}number\text{-}type = real\text{-}type \mid long\text{-}real\text{-}type \mid \mathbb{R}\text{-}type;$

$String\text{-}type :: length : \mathbb{N} ;$

$System\text{-}storage\text{-}type = loc\text{-}type \mid machine\text{-}address\text{-}type \mid bitnum\text{-}type$

6.10.3.2 Structures

types

$Structure = Array\text{-}structure \mid Record\text{-}structure \mid Procedure\text{-}structure \mid Set\text{-}structure$
 $\mid Pointer\text{-}structure \mid Opaque\text{-}structure \mid Enumerated\text{-}structure \mid Subrange\text{-}structure;$

$Array\text{-}structure :: itype : Typed$
 $ctype : Typed$

annotations The structure of an array type are the type names or basic types of the index and component types.

types

$Record\text{-}structure :: fields : Fields\text{-}list\text{-}structure$

annotations The structure of an array type are the type names or basic types of the components of the record.

types

$Fields\text{-}list\text{-}structure = Fields\text{-}structure^*;$

$Fields\text{-}structure = Fixed\text{-}fields\text{-}structure \mid Variant\text{-}fields\text{-}structure;$

$Fixed\text{-}fields\text{-}structure :: ids : Identifier^*$
 $type : Typed ;$

$Variant\text{-}fields\text{-}structure :: tag : [Identifier]$
 $tagt : Typed$
 $variants : Variant\text{-}structure^*$
 $other : Fields\text{-}list\text{-}structure ;$

$Variant\text{-}structure :: labels : Set\text{-}value$
 $fields : Fields\text{-}list\text{-}structure ;$

Procedure-structure = *Proper-procedure-structure* | *Function-procedure-structure*;

Proper-procedure-structure :: *parms* : *Formal-parameters-typed*

annotations The structure of a proper procedure are the type names or basic types of the formal parameters of the procedure.

types

Function-procedure-structure :: *parms* : *Formal-parameters-typed*
 return : [*Typed*]

annotations The structure of a function procedure are the type names or basic types of the formal parameters of the procedure, together with the the type names or basic types of the result. The return type is optional because forward use of a type name is allowed in the context of a procedure type.

types

Formal-parameters-typed = *Formal-parameter-typed**;

Formal-parameter-typed = *Value-formal-typed* | *Variable-formal-typed*;

Value-formal-typed :: *type* : *Variable-typed* ;

Variable-formal-typed :: *type* : *Variable-typed* ;

Set-structure :: *btype* : *Typed*

annotations The structure of a set type is the type name or basic type of the base type.

types

Pointer-structure :: *type* : [*Typed*]

annotations The structure of a pointer type is the the type name or basic type of the bound type. The bound type is optional because forward use of a type name is allowed in the context of a pointer.

types

Opaque-structure :: *opaque* : ;

Enumerated-structure :: *values* : *Identifier**

annotations The structure of an enumerated type is the sequence of identifiers defined by the type, the order of the identifiers in the sequence is identical to their order in the type definition.

types

Subrange-structure :: *rtype* : *Typed*
range : *Set-value*

annotations The structure of a subrange type is the range type and a set of values defined by the lower and upper bounds of the type definition.

types

Value-structure = *Structure* | *Open-array-typed* | *Procedure-typed* | *String-type*

6.10.3.3 Variable Types

types

Variable-typed = [*Typed*] | *Open-array-typed*

annotations A variable can either be declared or can be a parameter of a procedure, thus it will either be associated with its type name or its basic type, or can be an open array parameter. The type is optional here because forward use of a type name is allowed in the context of a procedure type.

types

Open-array-typed :: *type* : *Component-typed* ;

Component-typed = *Variable-typed*

6.10.3.4 Expression Types

types

Expression-typed = *Typed* | *String-type* | *Procedure-typed* | *Open-array-type*

annotations The result of an expression can be:

- a value of a basic type or a value whose type has been defined in a declaration;
- a string constant;
- a procedure constant.

6.10.3.5 Procedure Types

types

Procedure-typed = *Procedure-const* | *Standard-procedure*;

Procedure-const :: *proc* : *Procedure-structure*
level : \mathbb{N} ;

Standard-procedure = *standard-proper-procedure* | *standard-function-procedure*

6.10.3.6 Module Environments

types

Module-environment = *Unqualified-environment* | *Qualified-environment* ;

Unqualified-environment :: *env* : *Environment* ;

Qualified-environment :: *env* : *Environment*

6.10.3.7 Variables

types

Variable = *Elementary-variable* | *Array-variable* | *Record-variable* | *Tag-variable* | *Variant-variable* | *Tagged-variable* ;

Loc = An infinite set of tokens;

Elementary-variable :: *loc* : *Loc*
range : [*Set-value*]

annotations If the identifier associated with an elementary variable is of subrange type, then the range component contains the possible set of values that can be assigned to the variable.

Array-variable :: *vars* : *Array-components*

types

Array-components = *Ordinal-value* \xrightarrow{m} *Variable*

$\text{inv } ac \triangleq$
 $\forall a \in ac \cdot$
 $\forall rx, ry \in \text{rng } a \cdot$
 $\text{is-same-variable-type}(rx, ry)$

annotations The components of an array variable must all be of the same type.

types

Record-variable :: *fields* : *Record-components* ;

Record-components = *Identifier* \xrightarrow{m} *Variable*;

Tag-variable :: *var* : *Elementary-variable*
 variants : *Variant-component-set*

annotations A tag variable contains information about the variant components that are associated with it. Thus any change to the value of the tag which will cause a different component to be selected can be reflected by setting the variables of the old component to undefined.

types

Variant-variable :: *var* : *Variable*
 tagloc : *Elementary-variable*
 tagvals : *Value-set*

annotations A variant variable contains information about the explicit tag associated with the variant fields of which the variable is a component; thus before accessing the value of the variable the tag can be checked to see if its value is such that the variant that contains the variable is active.

types

Tagged-variable :: *var* : *Variable*
 tagvar : *Tag-variable*
 tagvals : *Value-set*

annotations A tagged variable contains information about the implicit tag associated with the variant fields of which the variable is a component; thus before accessing the value of the variable the tag can be checked to see if its value is such that the variant that contains the variable is active.

types

Variant-component :: *labels* : *Value-set*
 fields : *Record-components*

6.10.3.8 Protection Domains

types

Protection-domain :: *entry* : *Program-cont*
 leave : *Program-cont*
 val : *Value*

6.10.3.9 Procedure Denotations

types

Procedure-id = *TOKEN*;

Program-cont = *Cmdcont* \rightarrow *Coroutine-env* \rightarrow *Statecont*

6.10.3.10 Local Continuations

types

$$Exit = return \mid termination \mid halt$$

6.10.4 Operations on the Environment

6.10.4.1 Functions to Access the Constants Component of the Environment

functions

$$is_constant : Qualident \times Environment \rightarrow \mathbb{B}$$

$$\begin{aligned} is_constant(qid)\rho &\triangleq \\ \text{let } mk\text{-}(id, \rho_{mod}) &= unqualify(qid)\rho \text{ in} \\ id &\in \text{dom } \rho_{mod}.consts \end{aligned}$$

annotations Check that a qualident denotes a constant.

functions

$$type\text{-}of\text{-}constant : Qualident \rightarrow Environment \rightarrow Typed$$

$$\begin{aligned} type\text{-}of\text{-}constant(qid)\rho &\triangleq \\ \text{let } mk\text{-}(id, \rho_{mod}) &= unqualify(qid)\rho \text{ in} \\ \text{let } mk\text{-}Constant\text{-}value &(type, -) = \rho_{mod}.consts(id) \text{ in} \\ type & \end{aligned}$$

annotations The result is the unique type of a qualident that denotes a constant.

functions

$$associated\text{-}constant\text{-}value : Qualident \rightarrow Environment \rightarrow Value$$

$$\begin{aligned} associated\text{-}constant\text{-}value(qid)\rho &\triangleq \\ \text{let } mk\text{-}(id, \rho_{mod}) &= unqualify(qid)\rho \text{ in} \\ \text{let } mk\text{-}Constant\text{-}value &(-, value) = \rho_{mod}.consts(id) \text{ in} \\ value & \end{aligned}$$

annotations The result is the constant value associated with a qualident introduced in a constant declaration.

functions

$$overwrite\text{-}const\text{-}environment : Constants \times Environment \rightarrow Environment$$

$$\begin{aligned} overwrite\text{-}const\text{-}environment(consts)\rho &\triangleq \\ \text{let } \rho_{left} &= \text{remove-from-environment}(\text{dom } consts)\rho \text{ in} \\ \text{let } nconsts &= \rho_{left}.consts \upharpoonright consts \text{ in} \\ \rho_{left} \upharpoonright \{s\text{-}consts \mapsto nconsts\} & \end{aligned}$$

annotations The result is a new environment with the constants added.

6.10.4.2 Functions to Access the Types Component of the Environment

functions

$$is_type : Qualident \rightarrow Environment \rightarrow \mathbb{B}$$

$$\begin{aligned} is_type(qid)\rho &\triangleq \\ \text{let } mk\text{-}(id, \rho_{mod}) &= unqualify(qid)\rho \text{ in} \\ id &\in \text{dom } \rho_{mod}.types \end{aligned}$$

annotations Check that a qualident denotes a type.

functions

type-of : *Qualident* \rightarrow *Environment* \rightarrow [*Typed*]

type-of (*qid*) $\rho \triangleq$
 let *mk*-(*id*, ρ_{mod}) = *unqualify*(*qid*) ρ in
 let *typem* = $\rho_{mod}.types$ in
 if *id* \in dom *typem*
 then *typem*(*id*)
 else NIL

annotations The result is the type name or basic type associated with a qualident that denotes a type.

functions

overwrite-type-environment : *Types* \rightarrow *Environment* \rightarrow *Environment*

overwrite-type-environment (*types*) $\rho \triangleq$
 let $\rho_{left} = \text{remove-from-environment}(\text{dom } types)\rho$ in
 let *ntypes* = $\rho_{left}.types \upharpoonright types$ in
 $\rho_{left} \upharpoonright \{s-types \mapsto ntypes\}$

annotations Add a new type to the environment

6.10.4.3 Functions to Access the Structures Component of the Environment

functions

structure-of : *Expression-typed* \rightarrow *Environment* \rightarrow *Structure*

structure-of (*type*) $\rho \triangleq$
 if *type* \in *Type-name*
 then *s-strucs*(ρ)(*type*)
 else *type*

annotations The result is the structure associated with a type object.

functions

associated-structure : *Qualident* \rightarrow *Environment* \rightarrow [*Structure*]

associated-structure (*qid*) $\rho \triangleq$
 let *mk*-(*id*, ρ_{mod}) = *unqualify*(*qid*) ρ in
 let *type* = $\rho_{mod}.types(id)$ in
structure-of(*type*) ρ_{mod}

annotations The result is the structure of a qualident that denotes a type.

functions

overwrite-struc-environment : *Structures* \times *Environment* \rightarrow *Environment*

overwrite-struc-environment (*strucs*) $\rho \triangleq$
 let $\rho_{left} = \text{remove-from-environment}(\text{dom } strucs)\rho$ in
 let *nstrucs* = $\rho_{left}.strucs \upharpoonright strucs$ in
 $\rho_{left} \upharpoonright \{s-strucs \mapsto nstrucs\}$

annotations Add a new structure to the environment.

6.10.4.4 Functions to Access the Variables Component of the Environment

functions

$is_variable : Qualident \times Environment \rightarrow \mathbb{B}$

$is_variable(qid)\rho \triangleq$
 let $mk-(id, \rho_{mod}) = unqualify(qid)\rho$ in
 $id \in \text{dom } \rho_{mod}.vars$

annotations Check if a qualident denotes a variable.

functions

$type_of_variable : Qualident \rightarrow Environment \rightarrow Typed$

$type_of_variable(qid)\rho \triangleq$
 let $mk-(id, \rho_{mod}) = unqualify(qid)\rho$ in
 $\rho_{mod}.vars(id)$

annotations The result is the type name or basic type of a qualident that denotes a variable.

functions

$associated_variable : Qualident \rightarrow Environment \rightarrow (Variable_typed \mid Variable)$

$associated_variable(qid)\rho \triangleq$
 let $mk-(id, \rho_{mod}) = unqualify(qid)\rho$ in
 $\rho_{mod}.vars(id)$

annotations The result is the variable associated with a qualident that denotes a variable.

functions

$overwrite_var_environment : Variables \times Environment \rightarrow Environment$

$overwrite_var_environment(vars)\rho \triangleq$
 let $\rho_{left} = \text{remove-from-environment}(\text{dom } vars)\rho$ in
 let $nvars = \rho_{left}.vars \uparrow vars$ in
 $\rho_{left} \uparrow \{s\text{-}vars \mapsto nvars\}$

annotations Add a variable together with its type to the environment.

6.10.4.5 Functions to Access the Procedures Component of the Environment

functions

$is_procedure : Qualident \times Environment \rightarrow \mathbb{B}$

$is_procedure(qid)\rho \triangleq$
 let $mk-(id, \rho_{mod}) = unqualify(qid)\rho$ in
 $id \in \text{dom } \rho_{mod}.procs$

annotations Check if a qualident denotes a variable.

functions

$is_proper_procedure : Qualident \times Environment \rightarrow \mathbb{B}$

$is_proper_procedure(id)\rho \triangleq$
 let $mk-(id, \rho_{mod}) = unqualify(qid)\rho$ in
 let $obj = \rho_{mod}.procs(id)$ in
 $obj \in \text{Proper-procedure-structure} \cup \{\text{standard-proper-procedure}\}$

annotations Check if a qualident denotes a proper procedure.

functions

$is-function-procedure : Qualident \times Environment \rightarrow \mathbb{B}$
 $is-function-procedure(id)\rho \triangleq$
 let $mk-(id, \rho_{mod}) = unqualify(qid)\rho$ in
 let $obj = \rho_{mod}.procs(id)$ in
 $obj \in Function-procedure-structure \cup \{standard-function-procedure\}$

annotations Check if a qualident denotes a function procedure.

functions

$type-of-procedure : Qualident \rightarrow Environment \rightarrow Typed$
 $type-of-procedure(qid)\rho \triangleq$
 let $mk-(id, \rho_{mod}) = unqualify(qid)\rho$ in
 let $procid = \rho_{mod}.procs(id)$ in
 $\rho_{mod}.dens(procid)$

annotations The result is the type of a qualident that denotes a procedure.

functions

$associated-procedure : Qualident \rightarrow Environment \rightarrow (Procedure-typed \mid Procedure-value)$
 $associated-procedure(qid)\rho \triangleq$
 let $mk-(id, \rho_{mod}) = unqualify(qid)\rho$ in
 let $procid = \rho_{mod}.procs(id)$ in
 $\rho_{mod}.dens(procid)$

annotations The result is the proper procedure associated with a qualident.

functions

$overwrite-proc-environment : Procedures \times Environment \rightarrow Environment$
 $overwrite-proc-environment(procs)\rho \triangleq$
 let $\rho_{left} = remove-from-environment(dom\ procs)\rho$ in
 let $nprocs = s-procs(\rho_{left}) \uparrow procs$ in
 $\rho_{left} \uparrow \{s-procs \mapsto nprocs\}$

annotations Add a procedure and its associated qualident to the environment.

functions

$procedures-of : Environment \rightarrow Procedures$
 $procedures-of(\rho) \triangleq$
 $\rho.procs$

6.10.4.6 Functions to Access the Modules Component of the Environment

functions

$is-module : Qualident \times Environment \rightarrow \mathbb{B}$
 $is-module(qid)\rho \triangleq$
 let $mk-(id, \rho_{mod}) = unqualify(qid)\rho$ in
 $id \in dom\ \rho_{mod}.mods$

annotations Checks if a qualident denotes a module.

functions

associated-module : *Qualident* \rightarrow *Environment* \rightarrow *Module-environment*

associated-module (*qid*) $\rho \triangleq$
 let *mk-*(*id*, ρ_{mod}) = *unqualify*(*qid*) ρ in
 $\rho_{mod}.mods(id)$

annotations Return the module associated with a qualident.

functions

overwrite-mod-environment : *Modules* \times *Environment* \rightarrow *Environment*

overwrite-mod-environment (*mods*) $\rho \triangleq$
 let $\rho_{left} = \text{remove-from-environment}(\text{dom } mods)\rho$ in
 let $nmods = \rho_{left}.mods \upharpoonright mods$ in
 $\rho_{left} \upharpoonright \{s-mods \mapsto nmods\}$

annotations Add a module and its associated qualident to the environment.

functions

construct-module-environment : *Identifier* \times *Environment* \rightarrow *Environment*

construct-module-environment (*name*, ρ) \triangleq
 let $menv = \{name \mapsto mk\text{-}Qualified\text{-environment}(\rho)\}$ in
 $mk\text{-}Environment(\{\}, \{\}, \{\}, \{\}, \{\}, \{\}, menv, 0, \{\}, \{\}, \{\})$

6.10.4.7 Functions to Access the Procedure Level Component of the Environment

functions

current-level : *Environment* $\rightarrow \mathbb{N}$

current-level (ρ) \triangleq
 $\rho.level$

annotations Return the current nesting level of the environment.

functions

level-of : *Procedure-const* $\rightarrow \mathbb{N}$

level-of (*proc*) \triangleq
 $proc.level$

annotations Return the current level of a procedure.

functions

new-level : *Environment* \rightarrow *Environment*

new-level (ρ) \triangleq
 $\rho \upharpoonright maps\text{-}level \mapsto \rho.level + 1$

annotations Increase the nesting level of procedure declarations by 1.

functions

is-level-zero-proc : *Procedure-value* \rightarrow *Environment* $\rightarrow \mathbb{B}$

is-level-zero-proc (*proc*) $\rho \triangleq$
 is not yet defined

annotations Check that the procedure is declared at level zero.

6.10.4.8 Functions to Access the Protection Domain Component of the Environment

functions

$$\begin{aligned}
 &add-protection : [Protection-domain] \rightarrow Environment \rightarrow Environment \\
 &add-protection (pdom) \rho \triangleq \\
 &\quad \text{if } \rho.domain \neq \text{NIL} \wedge \\
 &\quad \quad pdom = \text{NIL} \\
 &\quad \text{then } \rho \\
 &\quad \text{else } \mu(\rho, s-domain \mapsto pdom)
 \end{aligned}$$

TO DO — Resolve whether it is acceptable that only the inner most access procedure is called in nested domains of protection.

functions

$$\begin{aligned}
 &protection-domain : Environment \rightarrow [Protection-domain] \\
 &protection-domain \rho \triangleq \\
 &\quad \rho.domain
 \end{aligned}$$

functions

$$\begin{aligned}
 &is-protected-domain : Environment \rightarrow \mathbb{B} \\
 &is-protected-domain(\rho) \triangleq \\
 &\quad \rho.domain \neq \text{NIL}
 \end{aligned}$$

6.10.4.9 Functions to Access the Procedure Continuation Component of the Environment

functions

$$\begin{aligned}
 &overwrite-den-environment : Procdens \rightarrow Environment \rightarrow Environment \\
 &overwrite-den-environment (pden) \rho \triangleq \\
 &\quad \text{let } \rho_{left} = \text{remove-from-environment}(\text{dom } pden) \rho \text{ in} \\
 &\quad \text{let } ndens = \rho_{left}.dens \upharpoonright pden \text{ in} \\
 &\quad \rho_{left} \upharpoonright \{s-dens \mapsto ndens\}
 \end{aligned}$$

functions

$$\begin{aligned}
 &allocate-procedure-id : Environment \rightarrow Procedure-id \\
 &allocate-procedure-id() \rho \triangleq \\
 &\quad \text{let } pid \in Procedure-id - \text{dom } \rho.pcont \text{ in} \\
 &\quad pid
 \end{aligned}$$

functions

$$\begin{aligned}
 &associated-procedure-cont : Procedure-id \rightarrow Environment \rightarrow Program-cont \\
 &associated-procedure-cont (procid) \rho \triangleq \\
 &\quad \rho.pcont(procid)
 \end{aligned}$$

6.10.4.10 Functions to Access the Environment

types

$$Object = Typed \mid Structure \mid Variable-typed \mid Variable \mid Procedure-typed \mid Procedure-value \mid Module-environment$$

functions

access-environment : *Qualident* \times *Environment* \rightarrow *Object*

access-environment (*qid*) $\rho \triangleq$
 let *mk-*(*id*, ρ_{mod}) = *unqualify*(*qid*) ρ in
 let *mk-Environment* (*consts*, *types*, -, *vars*, *procs*, *mods*, -, -, *pconts*, *lconts*) = ρ_{mod} in
 (*id* \in dom *consts* \rightarrow *consts*(*id*),
id \in dom *types* \rightarrow *types*(*id*),
id \in dom *vars* \rightarrow *vars*(*id*),
id \in dom *procs* \rightarrow *procs*(*id*),
id \in dom *mods* \rightarrow *mods*(*id*),
id \in dom *pconts* \rightarrow *pconts*(*id*),
id \in dom *lconts* \rightarrow *lconts*(*id*))

annotations Look up a qualident in the environment, and return the associated object and its properties.

functions

associated-value : *Qualident* \rightarrow *Environment* \rightarrow *Value*

associated-value (*qid*) $\rho \triangleq$
 let *mk-*(*id*, ρ_{mod}) = *unqualify*(*qid*) ρ in
access-environment(*id*) ρ_{mod}

functions

identifiers-in-scope : *Environment* \rightarrow *Identifier-set*

identifiers-in-scope (ρ) \triangleq
 let *mk-Environment* (*consts*, *types*, -, *vars*, *procs*, *mods*, -, -, -, -) = ρ in
 dom *consts* \cup dom *types* \cup dom *vars* \cup dom *procs* \cup dom *mods*

annotations The identifiers in scope is the union of all of the domains of various (mapping) components of the environment.

6.10.4.11 Operations to Update Environments

functions

merge-environment : *Environment* \times *Environment* \rightarrow *Environment*

merge-environment (ρ_a, ρ_b) \triangleq
 let *mk-Environment* (*nconsts*, *ntypes*, *nstrucs*, *nvars*, *nprocs*, *nmods*, *nlevel*, *doms*, *pconts*, *lconts*) = ρ_a in
 let *mk-Environment* (*consts*, *types*, *strucs*, *vars*, *procs*, *mods*, -, -, -, -) = ρ_b in
mk-Environment (*nconsts* \cup *consts*, *ntypes* \cup *types*, *nstrucs* \cup *strucs*, *nvars* \cup *vars*, *nprocs* \cup *procs*,
nmods \cup *mods*, *nlevel*, *doms*, *pconts*, *lconts*)
 pre *identifiers-in-scope*(ρ_a) \cap *identifiers-in-scope*(ρ_b) = { }

annotations Merge two environments.

functions

merge-environments : *Environment-set* \rightarrow *Environment*

merge-environments (*envs*) \triangleq
 if *envs* = { }
 then { }
 else let $\rho \in envs$ in
 let ρ_{rest} = *merge-environments*(*envs* - ρ) in
merge-environment(ρ_{rest}, ρ)

pre *is-disjoint*($\{\text{identifiers-in-scope}(\rho) \mid \rho \in \text{envs}\}$)

annotations merge all the environments contained in a set of environments.

functions

overwrite-environment : *Environment* \rightarrow *Environment* \rightarrow *Environment*

overwrite-environment (*env*) $\rho \triangleq$
 let *mk-Environment* (*econsts*, *etypes*, *estruc*s, *evars*, *eprocs*, *emods*, *elevel*, *edom*s, *epcont*s, *elcont*s) = *env* in
 let *mk-Environment* (*consts*, *types*, *strucs*, *vars*, *procs*, *mods*, -, -, -, -) = ρ in
mk-Environment (*consts* \dagger *econsts*, *types* \dagger *etypes*, *strucs* \dagger *estruc*s, *vars* \dagger *evars*, *procs* \dagger *eprocs*,
mods \dagger *emods*, *elevel*, *edom*s, *epcont*s, *elcont*s)

annotations Overwrite the environment ρ with the environment *env*.

functions

restrict-environment : *Identifier-set* \rightarrow *Environment* \rightarrow *Environment*

restrict-environment (*ids*) $\rho \triangleq$
 let *mk-Environment* (*consts*, *types*, *strucs*, *vars*, *procs*, *mods*, *level*, *dom*s, *pcont*s, *lcont*s) = ρ in
mk-Environment (*ids* \triangleleft *consts*, *ids* \triangleleft *types*, *strucs*, *ids* \triangleleft *vars*, *ids* \triangleleft *procs*, *ids* \triangleleft *mods*, *level*, *dom*s, *pcont*s, *lcont*s)

annotations *identifiers-in-scope*(*restrict-environment*(*ids*) ρ) = *ids*

functions

remove-from-environment : *Identifier-set* \rightarrow *Environment* \rightarrow *Environment*

remove-from-environment (*ids*) $\rho \triangleq$
 let *mk-Environment* (*consts*, *types*, *strucs*, *vars*, *procs*, *mods*, *level*, *dom*s, *pcont*s, *lcont*s) = ρ in
mk-Environment (*ids* \triangleleft *consts*, *ids* \triangleleft *types*, *strucs*, *ids* \triangleleft *vars*, *ids* \triangleleft *procs*, *ids* \triangleleft *mods*, *level*, *dom*s, *pcont*s, *lcont*s)

annotations *identifiers-in-scope*(*restrict-environment*(*ids*) ρ) = *ids*

functions

combine-definition-and-imported-environments : *Environment* \times *Environment* \rightarrow *Environment*

combine-definition-and-imported-environments (*denv*, *ienv*) \triangleq
 let *opaques* = *opaque-types-of* (*denv*) in
 let *procs* = *procedures-of* (*denv*) in
 let *idef* = *remove-from-environment*(*dom* *procs* \cup *opaques*)*denv* in
merge-environment(*idef*, *ienv*)

annotations combine the environments of a definition module and the associated implementation module.
 Note that the declarations for opaque types and procedures are taken from the implementation module.

Language Clarification

The declarations for opaque types and procedures are taken from the implementation module.

6.10.4.12 New Type Names

functions

generate-type-name : *Environment* \rightarrow *Type-name*

generate-type-name (ρ) \triangleq
 let *type* \in (*Type-name* - *dom* ρ .*strucs*) in
type

annotations Generate a type name that has not been used elsewhere in the current environment.

6.10.5 Unqualify a Qualident

functions

```

unqualify : Qualident → Environment → (Identifier × Environment)
unqualify (qid) ρ  $\triangleq$ 
  let  $\rho_{env} = merge\_environment(\rho)\rho_{std\_ids}$  in ( $qid \in Identifier \rightarrow (qid, \rho_{env})$ ,
     $qid \in Qualified \rightarrow$  let  $mk\_Qualified(id, qual) = qid$  in
      let  $\rho_{mod} = \rho_{env.mods}$  in
        if  $id \in \text{dom } \rho_{mod}$ 
        then  $unqualify(qual, \rho_{mod}(id))$ 
        else  $error("Incorrectly qualified identifier")$ )

```

annotations The result is the environment of the module containing the declaration of the identifier component of the qualified identifier, together with that identifier.

TO DO — A fix to make certain that the standard identifier environment is scanned last, change when this section is re-organised.

6.10.6 Types

6.10.6.1 Structured Types

A structured type is an array or record type.

functions

```

is-structured-type : Expression-typed → Environment →  $\mathbb{B}$ 
is-structured-type (type) ρ  $\triangleq$ 
  is-array-type (type) ρ  $\vee$ 
  is-record-type (type) ρ

```

annotations Check that a type is a structured type.

functions

```

is-array-type : Expression-typed → Environment →  $\mathbb{B}$ 
is-array-type is-array-type (type) ρ  $\triangleq$ 
  structure-of (type) ρ  $\in Array\_structure$ 

```

annotations Check that a type name denotes an array type.

functions

```

is-record-type : Expression-typed → Environment →  $\mathbb{B}$ 
is-record-type (type) ρ  $\triangleq$ 
  structure-of (type) ρ  $\in Record\_structure$ 

```

annotations Check that a type name denotes a record type.

6.10.6.2 Elementary Types

An elementary type is a procedure, pointer, opaque, set, or scalar type.

functions

$is\text{-}elementary\text{-}type : Expression\text{-}typed \rightarrow Environment \rightarrow \mathbb{B}$

$is\text{-}elementary\text{-}type (type)\rho \triangleq$
 $is\text{-}procedure\text{-}type (type)\rho \vee$
 $is\text{-}set\text{-}type (type)\rho \vee$
 $is\text{-}pointer\text{-}type (type)\rho \vee$
 $is\text{-}opaque\text{-}type (type)\rho \vee$
 $is\text{-}scalar\text{-}type (type)\rho$

annotations Check that a type is an elementary type.

functions

$is\text{-}procedure\text{-}type : Expression\text{-}typed \rightarrow Environment \rightarrow \mathbb{B}$

$is\text{-}procedure\text{-}type (type)\rho \triangleq$
 $is\text{-}proper\text{-}procedure\text{-}type (type)\rho \wedge$
 $is\text{-}function\text{-}procedure\text{-}type (type)\rho$

annotations Check that a type denotes a procedure type.

functions

$is\text{-}proper\text{-}procedure\text{-}type : Expression\text{-}typed \rightarrow Environment \rightarrow \mathbb{B}$

$is\text{-}proper\text{-}procedure\text{-}type (type)\rho \triangleq$
 $structure\text{-}of (type)\rho \in Proper\text{-}procedure\text{-}structure$

annotations Check that a type denotes a procedure type.

functions

$is\text{-}function\text{-}procedure\text{-}type : Expression\text{-}typed \rightarrow Environment \rightarrow \mathbb{B}$

$is\text{-}function\text{-}procedure\text{-}type (type)\rho \triangleq$
 $structure\text{-}of (type)\rho \in Function\text{-}procedure\text{-}structure$

annotations Check that a type denotes a procedure type.

functions

$is\text{-}set\text{-}type : Expression\text{-}typed \rightarrow Environment \rightarrow \mathbb{B}$

$is\text{-}set\text{-}type (type)\rho \triangleq$
 $structure\text{-}of (type)\rho \in Set\text{-}structure$

annotations Check that a type name denotes a set type.

functions

$is\text{-}pointer\text{-}type : Expression\text{-}typed \rightarrow Environment \rightarrow \mathbb{B}$

$is\text{-}pointer\text{-}type (type)\rho \triangleq$
 $structure\text{-}of (type)\rho \in Pointer\text{-}structure$

annotations Check that a type denotes a pointer type.

functions

$is\text{-}opaque\text{-}type : Expression\text{-}typed \rightarrow Environment \rightarrow \mathbb{B}$

$is\text{-}opaque\text{-}type (type)\rho \triangleq$
 $structure\text{-}of (type)\rho = opaque$

annotations Check that a type name denotes an opaque type.

6.10.6.3 Scalar Types

A Scalar type is an ordinal or a real type.

functions

$is\text{-}scalar\text{-}type : Expression\text{-}typed \rightarrow Environment \rightarrow \mathbb{B}$

$$is\text{-}scalar\text{-}type(type)\rho \triangleq \\ is\text{-}real\text{-}number\text{-}type(type)\rho \vee \\ is\text{-}ordinal\text{-}type(type)\rho$$

annotations Check that a type name denotes a scalar type.

6.10.6.4 Real Number Types

functions

$is\text{-}real\text{-}number\text{-}type : Expression\text{-}typed \rightarrow Environment \rightarrow \mathbb{B}$

$$is\text{-}real\text{-}number\text{-}type(type)\rho \triangleq \\ type \in Real\text{-}number\text{-}type$$

annotations Check that a type name denotes a real number type.

6.10.6.5 Ordinal Types

An ordinal type is a character, enumerated, subrange, or whole number type.

functions

$is\text{-}ordinal\text{-}type : Expression\text{-}typed \rightarrow Environment \rightarrow \mathbb{B}$

$$is\text{-}ordinal\text{-}type(type)\rho \triangleq \\ is\text{-}character\text{-}type(type)\rho \vee \\ is\text{-}enumerated\text{-}type(type)\rho \vee \\ is\text{-}subrange\text{-}type(type)\rho \vee \\ is\text{-}whole\text{-}number\text{-}type(type)\rho$$

annotations Check that a type is an ordinal type.

functions

$is\text{-}character\text{-}type : Expression\text{-}typed \rightarrow Environment \rightarrow \mathbb{B}$

$$is\text{-}character\text{-}type(type)\rho \triangleq \\ type = character\text{-}type$$

annotations Check that a type is a character type.

functions

$is\text{-}enumerated\text{-}type : Expression\text{-}typed \rightarrow Environment \rightarrow \mathbb{B}$

$$is\text{-}enumerated\text{-}type(type)\rho \triangleq \\ structure\text{-}of(type)\rho \in Enumerated\text{-}structure$$

annotations Check that a type is an enumerated type.

functions

$is-subrange-type : Expression-typed \rightarrow Environment \rightarrow \mathbb{B}$

$is-subrange-type (type) \rho \triangleq$
 $structure-of (type) \rho \in Subrange-structure$

annotations Check that a type is a subrange type.

6.10.6.6 Whole Number Types

A whole number type is either the unsigned type, the signed type or the type of a whole number literal value.

functions

$is-whole-number-type : Expression-typed \rightarrow Environment \rightarrow \mathbb{B}$

$is-whole-number-type (type) \rho \triangleq$
 $is-unsigned-type (type) \rho \vee$
 $is-signed-type (type) \rho \vee$
 $is-ZZ-type (type) \rho$

annotations Check that a type object denotes a whole number type.

functions

$is-unsigned-type : Expression-typed \rightarrow Environment \rightarrow \mathbb{B}$

$is-unsigned-type (type) \rho \triangleq$
 $type = unsigned-type$

annotations Check that a type denotes the unsigned type.

functions

$is-signed-type : Expression-typed \rightarrow Environment \rightarrow \mathbb{B}$

$is-signed-type (type) \rho \triangleq$
 $type = signed-type$

annotations Check that a type denotes the signed type.

functions

$is-ZZ-type : Expression-typed \rightarrow Environment \rightarrow \mathbb{B}$

$is-ZZ-type (type) \rho \triangleq$
 $type = \mathbb{Z}-type$

annotations Check that a type denotes the ZZ type.

6.10.6.7 Number Types

functions

$is-number-type : Expression-typed \rightarrow Environment \rightarrow \mathbb{B}$

$is-number-type (type) \rho \triangleq$
 $is-real-number-type (type) \rho \vee$
 $is-whole-number-type (type) \rho$

annotations Check that a type name denotes a real number type or a whole number type.

6.10.6.8 String Types

functions

$is-string-type : Expression-typed \rightarrow Environment \rightarrow \mathbb{B}$

$is-string-type(type)\rho \triangleq$
 $type \in String-type$

annotations Check that a type name denotes a string type.

6.10.6.9 System Storage Types

functions

$is-system-storage-type : Expression-typed \rightarrow Environment \rightarrow \mathbb{B}$

$is-system-storage-type(type)\rho \triangleq$
 $is-loc-type(type)\rho \vee$
 $is-address-type(type)\rho \vee$
 $is-bin-type(type)\rho$

annotations Check that a type name denotes a system storage type.

functions

$is-loc-type : Expression-typed \rightarrow Environment \rightarrow \mathbb{B}$

$is-loc-type(type)\rho \triangleq$
 $type = loc-type$

annotations Check that a type name denotes a storage location type.

functions

$is-address-type : Expression-typed \rightarrow Environment \rightarrow \mathbb{B}$

$is-address-type(type)\rho \triangleq$
 $type = address-type$

annotations Check that a type name denotes an address type.

functions

$is-bin-type : Expression-typed \rightarrow Environment \rightarrow \mathbb{B}$

$is-bin-type(type)\rho \triangleq$
 $type = binnum-type$

annotations Check that a type name denotes a binary number type.

functions

$is-machine-address-type : Expression-typed \rightarrow Environment \rightarrow \mathbb{B}$

$is-machine-address-type(type)\rho \triangleq$
 $type = machine-address-type$

annotations Check that a type name denotes a machine address type.

6.10.7 Unit Types

A unit type is the range type of a subrange, the host type of a subrange, the base type of a set, the component type of an array, the index type of an array, or the bound type of a pointer.

6.10.7.1 Range Type

The range type of a subrange type is the previously defined ordinal type from which the subrange type is directly derived.

functions

$$\begin{aligned} \text{range-type-of} &: \text{Typed} \rightarrow \text{Environment} \rightarrow \text{Typed} \\ \text{range-type-of}(\text{type})\rho &\triangleq \\ &\text{let } \text{mk-Subrange-structure}(\text{rtype}, -) = \text{structure-of}(\text{type})\rho \text{ in} \\ &\text{rtype} \end{aligned}$$

annotations The result is the range type of a subrange type.

6.10.7.2 Host Type

The host type of a subrange type is the previously defined ordinal type from which the subrange type is indirectly derived by defining a subrange of a previously defined subrange. It is a character type, enumerated type, or whole number type.

functions

$$\begin{aligned} \text{host-type-of} &: \text{Typed} \rightarrow \text{Environment} \rightarrow \text{Typed} \\ \text{host-type-of}(\text{type})\rho &\triangleq \\ &\text{if } \text{is-subrange-type}(\text{type})\rho \\ &\text{then let } \text{rtype} = \text{range-type-of}(\text{type})\rho \text{ in} \\ &\quad \text{host-type-of}(\text{rtype})\rho \\ &\text{else } \text{type} \end{aligned}$$

annotations The result is the host type of a subrange type.

6.10.7.3 Base Type

The base type of a set type is the previously defined ordinal type from which the members of the set type directly belong.

functions

$$\begin{aligned} \text{base-type-of} &: \text{Typed} \rightarrow \text{Environment} \rightarrow \text{Typed} \\ \text{base-type-of}(\text{type})\rho &\triangleq \\ &\text{let } \text{mk-Set-structure}(\text{btype}) = \text{structure-of}(\text{type})\rho \text{ in} \\ &\text{btype} \end{aligned}$$

annotations The result is the base type of a set type.

6.10.7.4 Component Type

The component type of an array type is the type of the components of an array structure.

functions

```
component-type-of : Expression-typed  $\rightarrow$  Environment  $\rightarrow$  Typed  
component-type-of (type) $\rho \triangleq$   
  let struc = structure-of(type) $\rho$  in  
  cases struc :  
    mk-Array-structure ( $\cdot$ , ctype)  $\rightarrow$  ctype,  
    mk-Open-array-typed (type)  $\rightarrow$  type  
  end
```

annotations The result is the component type of an array.

6.10.7.5 Index Type

The index type of an array type is the type of the index of an array structure.

functions

```
index-type-of : Expression-typed  $\rightarrow$  Environment  $\rightarrow$  Typed  
index-type-of (type) $\rho \triangleq$   
  let struc = structure-of(type) $\rho$  in  
  cases struc :  
    mk-Array-structure (itype,  $\cdot$ )  $\rightarrow$  itype,  
    mk-Open-array-typed ( $\cdot$ )  $\rightarrow$  unsigned-type  
  end
```

annotations The result is the index type of an array.

6.10.7.6 Bound Type

The bound type of a pointer type is the type of the variable pointed to by a value of the pointer type.

functions

```
bound-type-of : Typed  $\rightarrow$  Environment  $\rightarrow$  Typed  
bound-type-of (type) $\rho \triangleq$   
  let mk-Pointer-structure (btype) = structure-of(type) $\rho$  in  
  btype
```

annotations The result is the bound type of a pointer.

6.10.8 Other Type Functions

6.10.8.1 Return Types

functions

```
return-types-of : Expression-typed  $\rightarrow$  Environment  $\rightarrow$  Typed  
return-types-of (type) $\rho \triangleq$   
  let mk-Function-procedure-structure ( $\cdot$ , return) = structure-of(type) in  
  return
```

6.10.8.2 Parameter Types

functions

```

parameter-types-of : Expression-typed  $\rightarrow$  Environment  $\rightarrow$  Formal-parameters-typed
parameter-types-of (type)  $\rho \triangleq$ 
  let struc = structure-of (type) in
  cases struc :
    mk-Proper-procedure-structure (parms)  $\rightarrow$  parms,
    mk-Function-procedure-structure (parms, -)  $\rightarrow$  parms
  end

```

6.10.8.3 Auxiliary Functions for Records

functions

```

field-types-of-record : Expression-typed  $\rightarrow$  Environment  $\rightarrow$  (Identifier  $\xrightarrow{m}$  Typed)
field-types-of-record (type)  $\rho \triangleq$ 
  let record = structure-of-type (type)  $\rho$  in
  let mk-Record-structure (fields) = record in
  fields-of-fields-list (fields)

```

annotations The result is a mapping whose domain is the field identifiers of the fixed part of a record together with the field identifiers of the variant component(s). In the mapping each field identifier is mapped to its declared type.

functions

```

fields-of-fields-list : Fields-list-structure  $\rightarrow$  (Identifier  $\xrightarrow{m}$  Typed)
fields-of-fields-list (fields)  $\triangleq$ 
   $\bigcup \{ \text{fields-of-fields}(\text{field}) \mid$ 
     $\text{field} \in \text{elems fields} \}$ 

```

annotations Construct a mapping that is the union of the mappings constructed from each of the fields-structure components of the field-list-structure.

functions

```

fields-of-fields : Fields-structure  $\rightarrow$  (Identifier  $\xrightarrow{m}$  Typed)
fields-of-fields (field)  $\triangleq$ 
  cases field :
    mk-Fixed-fields-structure ()  $\rightarrow$  fields-of-fixed-fields(field),
    mk-Variant-fields-structure ()  $\rightarrow$  fields-of-variant-fields(field)
  end

```

annotations Construct a mapping from either a fixed-fields-structure or a variant-fields-structure component of a field-list-structure.

functions

```

fields-of-fixed-fields : Fixed-fields-structure  $\rightarrow$  (Identifier  $\xrightarrow{m}$  Typed)
fields-of-fixed-fields (field)  $\triangleq$ 
  let mk-Fixed-fields-structure (ids, type) = field in
  { id  $\mapsto$  type |
    id  $\in$  elems ids }

```

annotations Construct a mapping whose domain is the field identifiers of all the variant components of a record, and in which each field identifier is mapped to its declared type.

functions

$fields\text{-}of\text{-}variant\text{-}fields : Variant\text{-}fields\text{-}structure \rightarrow (Identifier \xrightarrow{m} Typed)$
 $fields\text{-}of\text{-}variant\text{-}fields (field) \triangleq$
 let $mk\text{-}Variant\text{-}fields\text{-}structure (tag, tagt, variant, other) = field$ in
 (if $tag \neq NIL$
 then $\{tag \mapsto tagt\}$
 else $\{ \}$) $\cup fields\text{-}of\text{-}variants(variant) \cup fields\text{-}of\text{-}fields\text{-}list(other)$

annotations Construct a mapping whose domain is the tag identifier, the field identifiers of the variant components of a record and the field identifiers of the ELSE component, and in which each field identifier is mapped to its declared type.

functions

$fields\text{-}of\text{-}variants : Variant\text{-}structure^* \rightarrow (Identifier \xrightarrow{m} Typed)$
 $fields\text{-}of\text{-}variants (variants) \triangleq$
 $\bigcup \{fields\text{-}of\text{-}variant(variant) \mid$
 $variant \in \text{elems } variants\}$

annotations Construct a mapping that relates the identifiers belonging to every variant component of a record type with their declared types.

functions

$fields\text{-}of\text{-}variant : Variant\text{-}structure \rightarrow (Identifier \xrightarrow{m} Typed)$
 $fields\text{-}of\text{-}variant (variant) \triangleq$
 let $mk\text{-}Variant\text{-}structure(-, fields) = variant$ in
 $fields\text{-}of\text{-}fields\text{-}list(fields)$

annotations Construct a mapping that relates the field identifiers of a particular variant component of a record type with their declared types.

6.10.8.4 Auxiliary Functions for Values

functions

$is\text{-}value : Qualident \rightarrow Environment \rightarrow \mathbb{B}$
 $is\text{-}value (qid)\rho \triangleq$
 $is\text{-}constant(qid)\rho \vee$
 $is\text{-}variable(qid)\rho \vee$
 $is\text{-}procedure(qid)\rho$

functions

$type\text{-}of\text{-}value : Qualident \rightarrow Environment \rightarrow Typed$
 $type\text{-}of\text{-}value (qid)\rho \triangleq$
 let $object = access\text{-}environment(qid)\rho$ in
 $(object \in Constant\text{-}value \rightarrow \text{let } mk\text{-}Constant\text{-}value (type, -) = object \text{ in}$
 $type\text{-}of(type)\rho,$
 $object \in Variable\text{-}typed \rightarrow type\text{-}of(object)\rho,$
 $object \in Procedure\text{-}typed \rightarrow type\text{-}of(object)\rho)$

6.10.8.5 Auxiliary Functions for Arrays

functions

$is_zero_array_type : Typed \rightarrow Environment \rightarrow \mathbb{B}$
 $is_zero_array_type (type) \rho \triangleq$
 $is_array_type (type) \rho \wedge$
 let $itype = index_type_of (type) \rho$ in
 $host_type_of (itype) = unsigned_type \wedge$
 $minimum(itype) = 0$

annotations Check that a type is an array type with signed index starting at 0.

functions

$lower_bound_of : Typed \rightarrow Environment \rightarrow Ordinal_value$
 $lower_bound_of (type) \rho \triangleq$
 let $itype = index_type_of (type) \rho$ in
 $minimum(itype)$
pre $is_array_type (type) \rho$

annotations The result is the lower bound of an array.

functions

upper-bound-of : *Typed* \rightarrow *Environment* \rightarrow *Ordinal-value*
upper-bound-of (*type*) $\rho \triangleq$
 let *itype* = *index-type-of* (*type*) ρ in
 maximum (*itype*)
pre *is-array-type* (*type*) ρ

annotations The result is the upper bound of an array.

functions

size-of-array : *Typed* \rightarrow *Environment* $\rightarrow \mathbb{N}$
size-of-array (*type*) $\rho \triangleq$
 upper-bound-of (*type*) $\rho - \text{lower-bound-of}(\text{type})\rho + 1$
pre *is-array-type* (*type*) ρ

6.10.8.6 Auxiliary Functions for Open Arrays

functions

is-open-array : *Expression-typed* $\rightarrow \mathbb{B}$
is-open-array (*type*) $\rho \triangleq$
 type \in *Open-array-typed*

annotations Check that a type name denotes a record type.

6.10.8.7 Auxiliary Functions for Opaque Types

functions

opaque-types-of : *Environment* \rightarrow *Identifier-set*
opaque-types-of (ρ) \triangleq
 let *type* = $\rho.\text{types}$ in
 {*id* \in dom *type* |
 is-opaque-type (*id*)}

annotations The opaque types of an environment are those types whose structure is opaque.

6.10.9 Values Associated with a Type

functions

maximum : *Typed* \rightarrow *Environment* \rightarrow *Value*
maximum (*type*) $\rho \triangleq$
 (*type* \in *Basic-type* \rightarrow *maximum-of-basic-type* (*type*),
 type \in *Type-name* \rightarrow let *struc* = *structure-of* (*type*) ρ in
 if *struc* \in *Enumerated-structure*
 then *maximum-of-enumerated* (*struc*)
 else *maximum-of-subrange* (*struc*))
pre *is-scalar-type* (*type*) ρ

annotations Return the largest value of a type.

functions

$maximum-of-basic-type : Basic-type \rightarrow Value$
 $maximum-of-basic-type (type) \triangleq$
 $(type = unsigned-type \rightarrow max-unsigned-value,$
 $type = signed-type \rightarrow max-signed-value,$
 $type = \mathbb{Z}\text{-type} \rightarrow max-zz-value,$
 $type = real-type \rightarrow max-real-value,$
 $type = long-real-type \rightarrow max-long-real-value,$
 $type = \mathbb{R}\text{-type} \rightarrow max-rr-value,$
 $type = Boolean-type \rightarrow TRUE,$
 $type = character-type \rightarrow \text{let } collating-sequence = map-to-seq(characters) \text{ in}$
 $\quad \text{last } collating-sequence)$

annotations Return the largest value of a basic type.

functions

$maximum-of-enumerated : Enumerated-structure \rightarrow Value$
 $maximum-of-enumerated (struc) \triangleq$
 $\text{let } mk-Enumerated-structure(values) = struc \text{ in}$
 $mk-Enumerated-value(\text{last } values, values)$

annotations Return the largest value of an enumerated type, this is denoted by the last identifier that occurs in the sequence of identifiers of the enumerated type.

functions

$maximum-of-subrange : Subrange-structure \rightarrow Value$
 $maximum-of-subrange (struc) \triangleq$
 $\text{let } mk-Subrange-structure(-, range) = struc \text{ in}$
 $\text{let } mk-Set-value(value) = range \text{ in}$
 $maxs(value)$

annotations Return the largest value of a subrange type.

functions

$minimum : Typed \rightarrow Environment \rightarrow Value$
 $minimum (type) \rho \triangleq$
 $(type \in Basic-type \rightarrow minimum-of-basic-type(type),$
 $type \in Type-name \rightarrow \text{let } struc = structure-of(type) \rho \text{ in}$
 $\quad \text{if } struc \in Enumerated-structure$
 $\quad \text{then } minimum-of-enumerated(struc)$
 $\quad \text{else } minimum-of-subrange(struc))$
 $\text{pre } is-scalar-type(typed) \rho$

annotations Return the smallest value of a type.

functions

minimum-of-basic-type : *Basic-type* \rightarrow *Value*

minimum-of-basic-type (*type*) \triangleq

(*type* = *unsigned-type* \rightarrow 0,
type = *signed-type* \rightarrow *min-signed-value*,
type = \mathbb{Z} -*type* \rightarrow 0,
type = *real-type* \rightarrow *min-real-value*,
type = *long-real-type* \rightarrow *min-long-real-value*,
type = \mathbb{R} -*type* \rightarrow *min-rr-value*,
type = *Boolean-type* \rightarrow FALSE,
type = *character-type* \rightarrow let *collating-sequence* = *map-to-seq*(*characters*) in
 hd collating-sequence)

annotations Return the smallest value of a basic type.

functions

minimum-of-enumerated : *Enumerated-structure* \rightarrow *Value*

minimum-of-enumerated (*struc*) \triangleq

let *mk-Enumerated-structure*(*values*) = *struc* in
 mk-Enumerated-value(*hd values*, *values*)

annotations Return the smallest value of an enumerated type, this is denoted by the first identifier that occurs in the sequence of identifiers of the enumerated type.

functions

minimum-of-subrange : *Subrange-structure* \rightarrow *Environment* \rightarrow *Value*

minimum-of-subrange (*struc*) \triangleq

let *mk-Subrange-structure*(-, *range*) = *struc* in
 let *mk-Set-value*(*value*) = *range* in
 mins(*value*)

annotations Return the smallest value of a subrange type.

functions

values-of : *Typed* \rightarrow *Environment* \rightarrow *Value-set*

values-of (*type*) ρ \triangleq

(*type* \in *Basic-type* \rightarrow *values-of-basic-type*(*type*),
type \in *Type-name* \rightarrow let *struc* = *structure-of*(*type*) in
 if *struc* \in *Enumerated-structure*
 then *values-of-enumerated*(*struc*)
 else *values-of-subrange*(*struc*))

pre *is-scalar-type*(*type*) ρ

annotations Return the set of values associated with a type.

functions

values-of-basic-type : *Basic-type* \rightarrow *Value-set*

values-of-basic-type (*type*) \triangleq

(*type* \in *Number-type* \rightarrow {*x* \in \mathbb{R} |
 minimum(*typed*) $\leq x \leq$ *maximum*(*typed*)},
type = *Boolean-type* \rightarrow {FALSE, TRUE},
type = *character-type* \rightarrow let *collating-sequence* = *map-to-seq*(*characters*) in
 collating-sequence)

annotations Return the set of values associated with a basic type.

functions

$values-of-enumerated : Enumerated-structure \rightarrow Value-set$

$values-of-enumerated (struc) \triangleq$
 $\text{let } mk-Enumerated-structure(values) = struc \text{ in}$
 $\{ mk-Enumerated-value(values(i), values) \mid$
 $i \in \text{inds } values \}$

annotations Return the set of value associated with an enumerated type, this is the set of values containing the identifiers that occurs in the sequence of identifiers of the enumerated type.

functions

$values-of-subrange : Subrange-structure \rightarrow Value-set$

$values-of-subrange (struc) \triangleq$
 $\text{let } mk-Subrange-structure(-, range) = struc \text{ in}$
 $\text{let } mk-Set-value(value) = range \text{ in}$
 $value$

annotations Return the set of value associated with an subrange type.

functions

$ordered-values-of : Typed \rightarrow Environment \rightarrow Value^*$
 $ordered-values-of (type)\rho \triangleq$
 $\text{let } sv \in [Value] \text{ be st } i < j \Rightarrow sv(i) < sv(j) \wedge$
 $\text{inds } sv = \{1, \dots, \text{card } values-of(type)\rho\} \wedge$
 $\text{elems } sv = values-of(type)\rho \text{ in}$
 sv

annotations Return a sequence of values associated with a type, the sequence is ordered by the values it is constructed from.

6.10.10 Local Continuations

functions

$exit : Exit \rightarrow Environment \Rightarrow$
 $exit(ex)\rho \triangleq$
 $\rho.\text{conts}(ex)$

functions

$tixe : Continuation \rightarrow Environment \rightarrow Declcont \rightarrow Cmdcont$
 $tixe(cont)\rho \triangleq$
 $\text{let } oldcont = \rho.\text{conts} \text{ in}$
 $\text{let } newcont = oldcont \uparrow cont \text{ in}$
 $\text{let } \rho_{new} = \mu(\rho, s\text{-conts} \mapsto newcont) \text{ in}$
 $\text{return}(\rho_{new})$

values

proc-struct = *mk-Proper-procedure-structure*([])

$\rho_{std-ids} : \rightarrow Environment$

$\rho_{std-ids}(\rho_{std-ids}) \triangleq$
 $mk-Environment(\{NIL \mapsto mk-Constant(NIL-TYPE, NIL-VALUE),$
 $FALSE \mapsto mk-Constant(BOOLEAN-TYPE, FALSE),$
 $TRUE \mapsto mk-Constant(BOOLEAN-TYPE, TRUE),$
 $INTERRUPTIBLE \mapsto mk-Constant(PRIORITY-TYPE, INTERRUPTIBLE),$
 $UNINTERRUPTIBLE \mapsto mk-Constant(PRIORITY-TYPE, UNINTERRUPTIBLE),$
 $\text{-- Types}\{BOOLEAN \mapsto BOOLEAN-TYPE,$
 $CARDINAL \mapsto UNSIGNED-TYPE,$
 $CHAR \mapsto CHARACTER-TYPE,$
 $LONGREAL \mapsto LONG-REAL-TYPE,$
 $INTEGER \mapsto SIGNED-TYPE,$
 $REAL \mapsto REAL-TYPE,$
 $PROC \mapsto PROC-TYPE,$
 $PROTECTION \mapsto PROTECTION-TYPE\},$
 $\text{-- Structures}\{PROC-TYPE \mapsto proc-struct\},$
 $\text{-- Variables}\{ \},$
 $\text{-- Procedures}\{DEC \mapsto STANDARD-PROPER-PROCEDURE,$
 $DISPOSE \mapsto STANDARD-PROPER-PROCEDURE,$
 $ENTER \mapsto STANDARD-PROPER-PROCEDURE,$
 $EXCL \mapsto STANDARD-PROPER-PROCEDURE,$
 $HALT \mapsto STANDARD-PROPER-PROCEDURE,$
 $INC \mapsto STANDARD-PROPER-PROCEDURE,$
 $INCL \mapsto STANDARD-PROPER-PROCEDURE,$
 $LEAVE \mapsto STANDARD-PROPER-PROCEDURE,$
 $NEW \mapsto STANDARD-PROPER-PROCEDURE,$
 $ABS \mapsto STANDARD-FUNCTION-PROCEDURE,$
 $CAP \mapsto STANDARD-FUNCTION-PROCEDURE,$
 $CHR \mapsto STANDARD-FUNCTION-PROCEDURE,$
 $FLOAT \mapsto STANDARD-FUNCTION-PROCEDURE,$
 $HIGH \mapsto STANDARD-FUNCTION-PROCEDURE,$
 $LENGTH \mapsto STANDARD-FUNCTION-PROCEDURE,$
 $LFLOAT \mapsto STANDARD-FUNCTION-PROCEDURE,$
 $MAX \mapsto STANDARD-FUNCTION-PROCEDURE,$
 $MIN \mapsto STANDARD-FUNCTION-PROCEDURE,$
 $PROT \mapsto STANDARD-FUNCTION-PROCEDURE,$
 $ODD \mapsto STANDARD-FUNCTION-PROCEDURE,$
 $ORD \mapsto STANDARD-FUNCTION-PROCEDURE,$
 $SIZE \mapsto STANDARD-FUNCTION-PROCEDURE,$
 $TRUNC \mapsto STANDARD-FUNCTION-PROCEDURE,$
 $VAL \mapsto STANDARD-FUNCTION-PROCEDURE,$
 $INT \mapsto STANDARD-FUNCTION-PROCEDURE\},$
 $\text{-- Modules}\{ \},$
 $\text{-- Protection domain} NIL,$
 $\text{-- level}\}$
 $0,$
 $\text{-- Procedure denotations}\{ \},$
 $\text{-- Local continuations}\{ \})$

functions

$\rho_{empty} : \rightarrow Environment$

$\rho_{empty} () \triangleq$
 $mk-Environment(\{\},$
 $-- Types\{\},$
 $-- Structures\{\},$
 $-- Variables\{\},$
 $-- Procedures\{\},$
 $-- Modules\{\},$
 $-- LevelNIL,$
 $-- Protection\ domainNIL,$
 $-- Procedure\ denotations\{\},$
 $-- Local\ continuations\{\})$

6.10.12 Constants

values

$max-unsigned-value$ = An implementation defined unsigned value;

$max-signed-value$ = An implementation defined signed value;

$max-zz-value$ = An implementation defined constant value;

$max-real-value$ = An implementation defined real value;

$max-long-real-value$ = An implementation defined long real value;

$max-rr-value$ = An implementation defined real constant value;

$min-unsigned-value$ = An implementation defined unsigned value;

$min-signed-value$ = An implementation defined signed value;

$min-zz-value$ = An implementation defined constant value;

$min-real-value$ = An implementation defined real value;

$min-long-real-value$ = An implementation defined long real value;

$min-rr-value$ = An implementation defined real constant value;

$step-range$ = An implementation defined set of values for the step of for loops

TO DO — 1. Complete the reorganisation of this section by moving all functions that access the environment into section 6.10 and simplify access to the environment.

2. If an identifier is not found in the environment, it should be looked up in *ρ_{std-ids}*, a more elegant solution is necessary than is currently incorporated here.

6.11 Storage and Auxiliary Functions

6.11.1 The Storage Model

6.11.1.1 The State

types

$$\begin{aligned} \text{Storage} :: & \text{store} && : \text{Loc} \xrightarrow{m} \text{Storable-value} \\ & \text{aliases} && : \text{Loc-pair-set} \\ & \text{io} && : \text{Stream-id} \xrightarrow{m} \text{Stream} \\ & \text{coroutines} : \text{Coroutine-id} \xrightarrow{m} \text{Coroutine-storage} \\ & \text{handlers} && : \text{Interrupt-source} \xrightarrow{m} \text{Handler} ; \end{aligned}$$
$$\text{Storable-value} = \text{Elementary-value} \mid \text{UNDEFINED};$$
$$\begin{aligned} \text{Loc-pair} :: & \text{first} && : \text{Loc} \\ & \text{second} && : \text{Loc} \end{aligned}$$

6.11.1.2 Values

types

$$\text{Elementary-value} = \text{Basic-value} \mid \text{Enumerated-value} \mid \text{Set-value} \mid \text{Pointer-value} \mid \text{Procedure-value} \mid \text{Protection};$$
$$\text{Value} = \text{Elementary-value} \mid \text{Array-value} \mid \text{Record-value};$$
$$\text{Array-value} = \text{Ordinal-value} \xrightarrow{m} \text{Value};$$
$$\text{Ordinal-value} = \text{Number} \mid \text{Bool} \mid \text{Enumerated-value};$$
$$\text{Record-value} = \text{Identifier} \xrightarrow{m} \text{Value};$$
$$\text{Basic-value} = \text{Number} \mid \text{String-value} \mid \text{NIL-VALUE} \mid \mathbb{B};$$
$$\text{Number} = \mathbb{R};$$
$$\text{String-value} = \mathbb{N} \xrightarrow{m} \text{char};$$
$$\begin{aligned} \text{Enumerated-value} :: & \text{value} : \text{Identifier} \\ & \text{order} : \text{Identifier}^*; \end{aligned}$$
$$\text{Set-value} :: \text{value} : \text{Ordinal-value-set} ;$$
$$\text{Pointer-value} :: \text{value} : (\text{Loc} \mid \text{NIL-VALUE}) ;$$
$$\text{Procedure-value} = \text{Procedure-id};$$
$$\text{Arguments} = \text{Value}^*;$$
$$\text{Protection} = \{\text{INTERRUPTIBLE}, \text{UNINTERRUPTIBLE}\} \cup \text{Other-protection};$$
$$\text{Other-protection} = \text{implementation-defined};$$
$$\text{Interrupt-source} = \text{implementation-defined}$$

annotations *Other-protection* is an implementation defined set of values together with a partial order. The order relation shall be such that:

$$\forall p \in \text{Protection} \cdot \text{INTERRUPTIBLE} \leq p \leq \text{UNINTERRUPTIBLE}$$

and all values in Protection shall be comparable with the two values INTERRUPTIBLE and UNINTERRUPTIBLE.

Interrupt-source is an implementation defined set of values

values

interrupt-source-type = implementation-defined

annotations An implementation defined type consistent with the set of values chosen to be consistent with interrupt source.

6.11.1.3 Coroutine Storage

types

Coroutine-storage :: *protection* : *Protection*^{*}
 locs : *Loc-set*

6.11.1.4 Handlers

types

Handler :: *id* : *Coroutine-id*
 from : [*Loc*]

6.11.2 The Coroutine Environment

types

Coroutine-env :: *map* : *Coroutine-id* \xrightarrow{m} *Program-cont*
 caller : *Coroutine-id*;
Coroutine-id = token | PROGRAM-set

6.11.3 The Continuations

types

Program-cont = *Cmdcont* \rightarrow *Coroutine-env* \rightarrow *Statecont*;
Cmdcont = *Coroutine-env* \rightarrow *Statecont*;
Statecont = *State* \rightarrow *Answer*;
Answer = *Stream-id* \xrightarrow{m} *Stream*

6.11.4 Operations on Storage

6.11.4.1 Operations to Initialise and Terminate the State

operations

initialise-state : *Streams*
initialise-state (*str*)
 ext wr *store* : *Loc* \xrightarrow{m} *Storable-value*
 wr *aliases* : *Loc-pair-set*
 wr *io* : *Stream-id* \xrightarrow{m} *Stream*

post $store = \{ \} \wedge$
 $aliases = \{ \} \wedge$
 $io = str$

annotations Initialises the state with the input streams of the program. Note that there is no storage allocated in the storage model.

operations

$terminate-state () \text{ result-of-program-execution} : Streams$
 ext wr $store : Loc \xrightarrow{m} Storable-value$
 $wr aliases : Loc-pair-set$
 $wr io : Stream-id \xrightarrow{m} Stream$
 post $store = \{ \} \wedge$
 $aliases = \{ \} \wedge$
 $io = \{ \} \wedge$
 $result-of-program-execution = \overleftarrow{io}$

annotations This operation returns the final streams and de-allocates all storage in the storage model.

6.11.4.2 Operations to Allocate and Access Storage

operations

$generate-location () \text{ } r : Loc$
 ext wr $store : Loc \xrightarrow{m} Storable-value$
 post let $r \in Loc$ be st $r \notin \text{dom } store$ in
 $store = \overleftarrow{store} \cup \{ r \mapsto \text{UNDEFINED} \}$

annotations Generate a new storage location (one currently not in the domain of the storage map) and set its value to undefined.

;

$exists (var : Loc) \text{ } r : \mathbb{B}$
 ext rd $store : Loc \xrightarrow{m} Storable-value$
 post $r \Leftrightarrow var \in \text{dom } store$

annotations Check the existence of a storage location.

;

$change-value (var : Loc, val : Storable-value)$
 ext wr $store : Loc \xrightarrow{m} Storable-value$
 pre $var \in \text{dom } store$
 post let $aliased = \{ other \mid mk-Loc-pair(var, other) \in aliases \}$ in
 $store = \overleftarrow{store} \uparrow \{ var \mapsto val \} \uparrow \{ loc \mapsto side-effect-of-change(var, val, loc) \mid loc \in aliased \}$

annotations Change the value of a storage location. That storage location could be aliased with other storage locations caused by passing a variable using a variable parameter of a storage type exported from system, this will cause an implementation defined change to these storage locations.

;

$contents (loc : Loc) \text{ } r : Value$
 ext rd $store : Loc \xrightarrow{m} Storable-value$

pre $loc \in \text{dom } store$
 post $r = store(loc)$

annotations Access the value of a storage location associated with a variable.

functions

$side-effect-of-change : Variable \times Value \times Loc \rightarrow Storable-value$
 $side-effect-of-change (var, val, loc) \triangleq$
An-implementation-defined-value

6.11.4.3 Operations to Deallocate Storage

operations

$deallocate-storage : Identifier-set \rightarrow Environment \xrightarrow{o} ()$
 $deallocate-storage (ids) \rho \triangleq$
277 let $denv = restrict-environment (ids) \rho$ in
299 $deallocate(denv)$

annotations De-allocate the storage associated with a set of identifiers.
;
 $deallocate : Environment \xrightarrow{o} ()$
 $deallocate (denv) \triangleq$
 let $mk-Environment(-, -, -, vars, -, mods, -) = denv$ in
 let $locs = \bigcup \{locs-of(var) \mid var \in rng\ vars\}$ in
299 $(deallocate-locs(vars) ;$
299 $deallocate-modules(mods))$

annotations De-allocate the storage denoted by variables defined in an environment.
;
 $deallocate-parameters : Identifier-set \rightarrow Environment \xrightarrow{o} ()$
 $deallocate-parameters (ids) \rho \triangleq$
 let $penv = restrict-environment(ids) \rho$ in
 let $mk-Environment(-, -, -, vars, -, mods, -) = penv$ in
 let $locs = \bigcup \{locs-of(var) \mid var \in rng\ vars\}$ in
299 $(deallocate-locs(locs) ;$
300 $remove-aliases(mods))$

annotations De-allocate the storage denoted by parameters.
;
 $deallocate-locs (locs : Loc-set)$
ext wr $store : Loc \xrightarrow{m} Storable-value$
post $store = locs \triangleleft \overline{store}$

annotations De-allocate the storage defined by a set of locations.
;
 $deallocate-modules : Module-environment \xrightarrow{o} ()$
 $deallocate-modules (mods) \triangleq$
 let $mk-Module-environment(env) = mods$ in
 if $env = \rho_{empty}$
 then skip
299 else $deallocate(env)$

annotations De-allocate the storage of a module.

functions

$locs-of : Variable \rightarrow Loc\text{-}set$

$locs-of(v) \triangleq$

cases v :

$mk\text{-}Elementary\text{-}variable(loc, -) \rightarrow \{loc\},$
 $mk\text{-}Array\text{-}variable(vars) \rightarrow \bigcup \{locs-of(var) \mid var \in \text{rng } vars\},$
 $mk\text{-}Record\text{-}variable(fields) \rightarrow \bigcup \{locs-of(field) \mid field \in \text{rng } fields\},$
 $mk\text{-}Tag\text{-}variable(var, -, -) \rightarrow locs-of(var),$
 $mk\text{-}Variant\text{-}variable(var, tagloc, -) \rightarrow locs-of(var) \cup \{tagloc\},$
 $mk\text{-}Tagged\text{-}variable(var, tagloc, -) \rightarrow locs-of(var) \cup \{tagloc\}$

end

annotations Return the set of locations associated with a variable.

6.11.4.4 Storage Aliasing

functions

$alias-set : Variable \times Variable \rightarrow Loc\text{-}pair\text{-}set$

$alias-set(vara, varb) \triangleq$

let $locsa = locs-of(vara)$ in
 let $locsb = locs-of(varb)$ in
 $\{mk\text{-}Loc\text{-}pair(fst, snd) \mid fst, snd \in locsa \cup locsb\}$

annotations Return the set of alias pairs that could occur if two variable share storage.

operations

$set\text{-}aliases(vara, varb : Variable) \ r : Value$

ext $\lambda aliases : Loc\text{-}pair\text{-}set$

300 post $aliases = alias-set(vara, varb) \cup \overleftarrow{aliases}$

annotations Update the alias information with the set of alias pairs that could occur if two variable share storage.

;

$remove\text{-}aliases(locs : Locs\text{-}set)$

ext wr $aliases : Loc\text{-}pair\text{-}set$

post $aliases = \{pair \in \overleftarrow{aliases} \mid first(pair) \in locs \wedge second(pair) \in locs\}$

annotations Remove alias pairs from storage.

;

$alias\text{-}machine\text{-}address(loc : Locs, val : Value)$

ext wr $aliases : Loc\text{-}pair\text{-}set$

post let $maddr = implementation\text{-}defined\text{-}machine\text{-}address(val)$ in

$aliases = \{mk\text{-}Loc\text{-}pair(loc, maddr)\} \cup \overleftarrow{aliases}$

annotations Alias a machine address with a location.

6.11.4.5 Continuations

TO DO — Fix up definition of *terminate-actions* in *stop*

values

$stop = \lambda c \cdot \lambda \sigma \cdot STOP \circ \lambda c \cdot \lambda \sigma \cdot terminate\text{-}actions;$

$cenv = mk\text{-}Coroutine\text{-}env(\{\}, PROGRAM)$

operations

$initialise : Coroutine\text{-}id \times Program\text{-}cont \xrightarrow{o} ()$

$initialise(id, proc) \triangleq$
 $\lambda c \cdot \lambda e \cdot \text{let } mk\text{-}Coroutine\text{-}env(map, caller) = e \text{ in}$
 $\text{return } mk\text{-}Coroutine\text{-}env(map \upharpoonright \{id \mapsto proc\}, caller)$

annotations Initialise a coroutine by adding to the coroutine environment.

;

$resume : Identifier \xrightarrow{o} ()$

$resume(id) \triangleq$
 $\lambda c \cdot \lambda e \cdot \text{let } mk\text{-}Coroutine\text{-}env(map, caller) = e \text{ in}$
 $\text{let } map' = map \upharpoonright \{caller \mapsto c\} \text{ in}$
 $\text{return } map'(id)(map', id)$

annotations Resume a coroutine by saving the current execution of the coroutine that is currently being executed in the environment, and continuing the execution of the coroutine to be resumed.

;

$allocate\text{-}coroutine\text{-}id : \xrightarrow{o} Coroutine\text{-}id$

$allocate\text{-}coroutine\text{-}id \triangleq$
 $\lambda c \cdot \lambda e \cdot \text{let } mk\text{-}Coroutine\text{-}env(map, caller) = e \text{ in}$
 $\text{let } id \in Coroutine\text{-}Id - \text{dom } map \text{ in}$
 $\text{return } id$

annotations Allocate an unused coroutine identifier.

;

$reserve\text{-}coroutine\text{-}workspace : Coroutine\text{-}id \times Loc \times \mathbb{N} \xrightarrow{o} ()$

$reserve\text{-}coroutine\text{-}workspace(cid, space, size) \triangleq$

301 $\text{def } store = coroutine\text{-}locs(space, size) ;$
 $\text{let } cloc = mk\text{-}Coroutine\text{-}storage([], store) \text{ in}$
 302 $\text{add-to-store}(cid, cloc)$

annotations Construct coroutine storage and add it to the state. The storage consists of an empty protection stack and the storage for the variables and control information for a coroutine.

;

$coroutine\text{-}locs : Loc \times \mathbb{N} \xrightarrow{o} Loc\text{-}set$

$coroutine\text{-}locs(space, size)\text{return-an-implementation-defined-set-of-Loc's. ;}$

$is\text{-}adequate\text{-}work\text{-}space : Loc \times \mathbb{N} \xrightarrow{o} \mathbb{B}$

$is\text{-}adequate\text{-}work\text{-}space(space, size)$

;

$attach : \mathbb{N} \xrightarrow{o} ()$

$attach(source) \triangleq$
 $\lambda c \cdot \lambda e \cdot \lambda \sigma \cdot \text{let } mk\text{-}Coroutine\text{-}env(-, caller) = e \text{ in}$
 $\text{let } mk\text{-}State(-, -, -, -, handlers) = \sigma \text{ in}$
 $\text{let } new\text{-}handler = mk\text{-}Handler(caller, NIL) \text{ in}$
 $\text{let } new\text{-}handlers = handlers \upharpoonright \{source \mapsto new\text{-}handler\} \text{ in}$
 $\text{let } \sigma_{updated} = \sigma \upharpoonright \{handlers \mapsto new\text{-}handlers\} \text{ in}$
 $\text{return } c \ e \ \sigma_{updated}$

annotations Attach the calling coroutine to a source of interrupts.

;

$detach : \mathbb{N} \xrightarrow{o} ()$

$detach(source) \triangleq$
 $\lambda c \cdot \lambda e \cdot \lambda \sigma \cdot \text{let } mk\text{-Coroutine-env}(-, caller) = e \text{ in}$
 $\text{let } mk\text{-State}(-, -, -, -, handlers) = \sigma \text{ in}$
 $\text{let } new\text{-handlers} = remove(source, caller, handlers) \text{ in}$
 $\text{let } \sigma_{updated} = \sigma \uparrow handlers \mapsto new\text{-handlers} \text{ in}$
 $\text{return } c \ e \ \sigma_{updated}$

annotations Detach the calling coroutine from a source of interrupts.

functions

$remove : \mathbb{N} \rightarrow Coroutine\text{-}id \rightarrow Handlers \rightarrow Handlers$

$remove(source, caller, handlers) \triangleq$
 $\text{if } source \in \text{dom } handlers \wedge \text{let } hand = handlers(source) \text{ in } hand.id = caller$
 $\text{then } \{source\} \triangleleft handlers$
 $\text{else } handlers$

operations

$add\text{-to-store} : Coroutine\text{-}id \times Coroutine\text{-}storage \xrightarrow{o} ()$

$add\text{-to-store}(cid, cloc) \triangleq$
 $\lambda c \cdot \lambda e \cdot \lambda \sigma \cdot \text{let } mk\text{-State}(-, -, -, coroutines, -) = \sigma \text{ in}$
 $\text{let } new\text{-coroutines} = coroutines \uparrow \{cid \mapsto cloc\} \text{ in}$
 $\text{let } \sigma_{updated} = \sigma \uparrow \{coroutines \mapsto new\text{-coroutines}\} \text{ in}$
 $\text{return } c \ e \ \sigma_{updated}$

annotations Add the storage for a coroutine to the state.

;

$current\text{-coroutine} : \xrightarrow{o} Coroutine\text{-}id$

$current\text{-coroutine}() \triangleq$
 $\lambda c \cdot \lambda e \cdot \lambda \sigma \cdot \text{let } mk\text{-Coroutine-env}(-, caller) = e \text{ in}$
 $\text{return } caller$

annotations Return the identity of the currently executing coroutine.

;

$target\text{-for-interrupted-coroutine} : Loc \xrightarrow{o} ()$

$target\text{-for-interrupted-coroutine}(loc) \triangleq$
 $\lambda c \cdot \lambda e \cdot \lambda \sigma \cdot \text{let } mk\text{-Coroutine-env}(-, caller) = e \text{ in}$
 $\text{let } mk\text{-State}(-, -, -, -, handlers) = \sigma \text{ in}$
 $\text{let } source \in \text{dom } handlers \text{ be st } handlers(source) = caller \text{ in}$
 $\text{let } mk\text{-Handler}(id, -) = handlers(source) \text{ in}$
 $\text{let } new\text{-handler} = mk\text{-Handler}(id, loc) \text{ in}$
 $\text{let } new\text{-handlers} = handlers \uparrow \{source \mapsto new\text{-handler}\} \text{ in}$
 $\text{let } \sigma_{updated} = \sigma \uparrow \{handlers \mapsto new\text{-handlers}\} \text{ in}$
 $\text{return } c \ e \ \sigma_{updated}$

annotations Set a handler for an interrupt.

;

is-attached : $\xrightarrow{o} \mathbb{B}$

is-attached () \triangleq
 $\lambda c \cdot \lambda e \cdot \lambda \sigma \cdot \text{let } mk\text{-Coroutine-env}(-, caller) = e \text{ in}$
 $\text{let } mk\text{-State}(-, -, -, -, handlers) = \sigma \text{ in}$
 $\text{return } \exists source \in \text{dom } handlers \cdot$
 $handlers(source) = caller$

annotations Check that the currently executing coroutine is attached to a source of interrupts.

;

transfer-to-handler : $\mathbb{N} \xrightarrow{o} ()$

transfer-to-handler (source) \triangleq
 $\lambda c \cdot \lambda e \cdot \lambda \sigma \cdot \text{let } mk\text{-Coroutine-env}(-, caller) = e \text{ in}$
 $\text{let } mk\text{-State}(-, -, -, -, handlers) = \sigma \text{ in}$
 $\text{let } mk\text{-Handler}(id, from) = handlers(source) \text{ in}$
 $(assign(from, caller) \rho;$
 $resume(id))$

112

301

annotations An event connected to the coroutine calling **IOTRANSFER** will cause the coroutine to resume at the statement following the call to **IOTRANSFER** and place the identity of the coroutine that was interrupted in the second argument.

TO DO — add following text and check:

It shall be an exception if a **RETURN** statement is explicitly or implicitly executed in the activation of the procedure which is given as a parameter to **NEWCOROUTINE**.

It shall not be an exception to explicitly or implicitly execute a **RETURN** statement if the procedure is called normally (i.e. not as a result of a call to **NEWCOROUTINE** and a subsequent **TRANSFER**).

6.11.4.6 Accessing the Protection Component of the State

operations

current-protection : $\xrightarrow{o} Protection$

current-protection () \triangleq
 $\lambda c \cdot \lambda e \cdot \lambda \sigma \cdot \text{let } mk\text{-Coroutine-env}(-, caller) = e \text{ in}$
 $\text{let } mk\text{-State}(-, -, -, cr, -) = \sigma \text{ in}$
 $\text{let } mk\text{-Coroutine}(protection, -) = cr(caller) \text{ in}$
 $\text{return } protection$

annotations Read the state and return the current protection of the caller.

;

is-less-restrictive-protection : $Protection \xrightarrow{o} \mathbb{B}$

is-less-restrictive-protection (val) \triangleq
 $\lambda c \cdot \lambda e \cdot \lambda \sigma \cdot \text{let } mk\text{-Coroutine-env}(-, caller) = e \text{ in}$
 $\text{let } mk\text{-State}(-, -, -, cr, -) = \sigma \text{ in}$
 $\text{let } mk\text{-Coroutine}(protection, -) = cr(caller) \text{ in}$
 $\text{return } val \leq_{protection} protection \wedge val \neq protection$

annotations Read the state and return *true* iff the parameter is less restrictive than the current protection of the caller. The operator $\leq_{protection}$ is implementation defined.

;

is-current-protection : *Protection* \xrightarrow{o} \mathbb{B}

is-current-protection (*val*) \triangleq

$\lambda c \cdot \lambda e \cdot \lambda \sigma \cdot \text{let } mk\text{-}State(-, -, -, cr, -) = \sigma \text{ in}$
 $\text{let } mk\text{-}Coroutine\text{-}env(-, caller) = e \text{ in}$
 $\text{let } mk\text{-}State(-, -, -, cr, -) = \sigma \text{ in}$
 $\text{let } mk\text{-}Coroutine(protection, -) = cr(caller) \text{ in}$
 $\text{return } val = protection$

annotations Read the state and return *true* iff the parameter is the same value as the current protection for the caller.

;

set-protection : *Protection* \xrightarrow{o} ()

set-protection (*val*) \triangleq

$\lambda c \cdot \lambda e \cdot \lambda \sigma \cdot \text{let } mk\text{-}Coroutine\text{-}env(-, caller) = e \text{ in}$
 $\text{let } mk\text{-}State(-, -, -, cr, -) = \sigma \text{ in}$
 $\text{let } mk\text{-}Coroutine\text{-}storage(protection, locs) = cr(caller) \text{ in}$
 $\text{let } new\text{-}crstore = mk\text{-}Coroutine\text{-}storage([val] \curvearrowright protection, locs) \text{ in}$
 $\text{let } new\text{-}cr = cr \uparrow \{ caller \mapsto new\text{-}crstore \} \text{ in}$
 $\text{let } \sigma_{updated} = \mu(\sigma, caller \mapsto new\text{-}cr) \text{ in}$
 $\text{return } c \ e \ \sigma_{updated}$

annotations Save the previous protection and add the new protection to the state.

;

restore-protection : \xrightarrow{o} ()

restore-protection () \triangleq

$\lambda c \cdot \lambda e \cdot \lambda \sigma \cdot \text{let } mk\text{-}Coroutine\text{-}env(-, caller) = e \text{ in}$
 $\text{let } mk\text{-}State(-, -, -, cr, -) = \sigma \text{ in}$
 $\text{let } mk\text{-}Coroutine\text{-}storage(protection, locs) = cr(caller) \text{ in}$
 $\text{if } protection \neq []$
 $\text{then let } new\text{-}crstore = mk\text{-}Coroutine\text{-}storage(tl \ protection, locs) \text{ in}$
 $\text{let } new\text{-}cr = cr \uparrow \{ caller \mapsto new\text{-}crstore \} \text{ in}$
 $\text{let } \sigma_{updated} = \sigma \uparrow \{ caller \mapsto new\text{-}cr \} \text{ in}$
 $\text{return } c \ e \ \sigma_{updated}$

306

$\text{else } mandatory\text{-}exception(NOPROTECTION) (e) \ \sigma$

annotations Remove the previous protection from the state. It is an exception if there is no previous protection saved.

TO DO —

- a) Change all of the implicit operations in this section to continuation style.
- b) The exception handling context must be included as part of the state of a coroutine.

6.12 Exceptions

TO DO — This section to be completed. It will contain the functions and types necessary to ensure a consistent treatment of Language (Clause 6 and 7) exceptions and those raised in separate modules.

It will contain the VDM for raising exceptions used by section 7.3, plus other relevant functions.

The *Must-detect-exception* and *May-detect-exception* types will be augmented with other values as the fourth draft is assembled.

Think up better names for *Must-detect-exception* and *May-detect-exception*.

6.12.1 Exception Reporting

types

Language-exception = *Must-detect-exception* | *May-detect-exception*;

Must-detect-exception = ASSIGN-RANGE
| CASE-RANGE
| COMPLEX-OVERFLOW
| COMPLEX-ZERO-DIVISION
| INDEX-RANGE
| NIL-DEREFERENCE
| NOT-ATTACHED
| PASSIVE-PROGRAM
| REAL-OVERFLOW
| REAL-ZERO-DIVISION
| RETURN-RANGE
| SET-RANGE
| SMALL-WORKSPACE
| TAG-RANGE
| UNDEFINED-VALUE
| WHOLE-NONPOS-DIV
| WHOLE-NONPOS-MOD
| WHOLE-OVERFLOW
| WHOLE-ZERO-DIVISION
| WHOLE-ZERO-REMAINDER

annotations

ASSIGN-RANGE	if	assignment value out of range of target
TAG-RANGE	if	tag value out of range
RETURN-RANGE	if	returned result of a function is out of range
CASE-RANGE	if	case selector out of range
INDEX-RANGE	if	array indexed out of range
NIL-DEREFERENCE	if	attempt to dereference nil
NOT-ATTACHED	if	the caller is not attached to a source of interrupts
SMALL-WORKSPACE	if	size of the supplied workspace is smaller than the minimum size required

;

May-detect-exception = INACTIVE-VARIANT
| NONEXISTENT
| ADDRESS-ARITHMETIC
| LOWREALEXCEPTION
| LOWLONGEXCEPTION
| PROCESS-ERROR

annotations

INACTIVE-VARIANT	if	access to non-active variant
NONEXISTENT	if	attempt to access a non-existing variable
ADDRESS-ARITHMETIC	if	Invalid use of DIFADR , SUBADR or ADDADR
LOWREALEXCEPTION	if	invalid LowReal operation
LOWLONGEXCEPTION	if	invalid LowLong operation
PROCESS-ERROR	if	invalid Processes operation

6.12.1.1 Exceptions that shall be reported

operations

mandatory-exception : *Must-detect-exception* \xrightarrow{o} ()

mandatory-exception (*ex*) is not yet defined

6.12.1.2 Exceptions that may be reported

An implementation shall define which of the exceptions which it is not mandatory to detect, are actually detected. An implementation need not define the circumstances when the exception is detected and when it is not.

operations

non-mandatory-exception : *May-detect-exception* \xrightarrow{o} ()

non-mandatory-exception (*ex*) is not yet defined

6.12.1.3 Messages associated with exceptions

The implementation shall define the character string associated with each of the exceptions raised in clause 6 (which may be accessed using the **GetMessage** procedure — see 7.3.4).

functions

implementation-defined-exception-string : *Language-exception* \rightarrow char*

implementation-defined-exception-string (*ex*) \triangleq
is not yet defined

6.13 A Summary of Notation

6.13.1 Types

6.13.1.1 The basic types

The set of Boolean values is written as \mathbb{B}

$$\mathbb{B} = \{\text{true}, \text{false}\}$$

The set of all natural numbers is written as \mathbb{N}

$$\mathbb{N} = \{0, 1, 2, \dots\}$$

The set of all positive natural numbers is written as \mathbb{N}_1

$$\mathbb{N}_1 = \{0, 1, 2, \dots\}$$

$$\mathbb{N}_1 = \mathbb{N} - \{0\}$$

The set of all integers is written as \mathbb{Z}

$$\mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$$

$$\mathbb{Z} = \{-n \mid n \in \mathbb{N}\} \cup$$

$$\{n \mid n \in \mathbb{N}\}$$

The set of all rational numbers is written as \mathbb{Q} , and the set of all real numbers is written as \mathbb{R} .

Note that: $\mathbb{N}_1 \subset \mathbb{N} \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R}$

The set of all characters is written as `char`, and a set consisting of an infinite number of tokens is written `token`.

6.13.1.2 Type definitions

$$A = B$$

Define another (new) name for the set of objects B . The two names are synonymous in the sense that the members of each set cannot be distinguished. A is a set of elements such that:

$$x \in A \Leftrightarrow x \in B$$

$$A = \text{NAME}$$

Define a set consisting of the single object denoted by the constant `NAME`.

$$x \in A \Leftrightarrow x = \text{NAME}$$

$$A = B\text{-set}$$

The `-set` constructor produces the set of all finite sub-sets, so for a set B the B -set is the set of all possible sub-sets of B – thus an element of B -set is a sub-set of B .

$$x \in A \Leftrightarrow x \subseteq B$$

$$A = B^*$$

Given any set B , the `*` constructor builds the set of all sequences which have components from the set B . An element of the set B^* is a sequence whose elements come from the set A .

$$a \in A \Rightarrow \forall i \in \text{inds } a \cdot a(i) \in B$$

$$A = B^+$$

Given any set B , the $^+$ constructor builds the set of all none-empty sequences which have components from the set B .

$$a \in A \Rightarrow \forall i \in \text{inds } a \cdot a(i) \in B \wedge \text{len } a > 0$$

$$M = D \xrightarrow{m} R$$

Define M to be the set of possible mappings from the domain set D to the range set R .

$$M = D \xleftarrow{m} R$$

Define M to be the set of possible one-to-one mappings from the domain set D to the range set R .

$$m \in M \Rightarrow \text{dom } m \subset D \wedge \text{rng } m \subset R \wedge \text{card dom } m = \text{card rng } m$$

$$P = S \times T$$

Define P to be the product set consisting of all possible pairs of elements, with the first element of the pair drawn from the set S and the second element from the set T .

$$P = \{(s, t) \mid s \in S \wedge t \in T\}$$

$$P = Q \mid R$$

Define P to be the union of the two sets Q and R .

$$p \in P \Leftrightarrow p \in Q \vee$$

$$p \in R$$

$$A = [B]$$

Define the elements of A to be either in B or ‘missing’.

$$A = [B] \Leftrightarrow A = B \mid \text{nil}$$

compose B of

$sel1 : Set1$

$sel2 : Set2$

$Set3$

end

Defines A to be a composite object, the name of the type of the composite object is B , the first component of which comes from the set $Set1$, the second component from the set $Set2$ and the third component from the set $Set3$. The names $sel1$ and $sel2$ are selectors.

$$A = \{mk\text{-}B(x, y, z) \mid x \in Set1 \wedge y \in Set2 \wedge z \in Set3\}$$

$$mk\text{-}B = Set1 \times Set2 \times Set3 \rightarrow A$$

The following is also valid:

```

B =  compose B of
      sel1   : Set1
      sel2   : Set2
              Set3
      end

```

and can be abbreviated to:

```

B :: sel1 : Set1
    sel2 : Set2
        Set3

```

is not yet defined

The type has still to be defined; the choice has been postponed to a later step in the development.

6.13.2 Logic

6.13.2.1 Propositional operators

The propositional operators (in order of decreasing priority):

$\neg A$

Not A ; the negation of the proposition A .

$A \wedge B$

A and B ; the conjunction of the propositions A and B .

$A \vee B$

A or B ; the disjunction of the propositions A and B .

$A \Rightarrow B$

Implication, A implies B ; if A then B .

$A \Leftrightarrow B$

Equivalence of the two propositions A and B ; A if and only if B .

6.13.2.2 Quantifiers

$\forall x \in X \cdot P(x)$

The universal quantifier; for all elements in the set X , the property P holds.

$\exists x \in X \cdot P(x)$

The existential quantifier; there exists one or more elements in the set X such that the property P holds.

$\exists! x \in X \cdot P(x)$

There exists exactly one element in the set X such that the property P holds.

6.13.2.3 Equivalent expressions

$\exists i \in D \cdot \neg P(i) = \neg (\forall i \in D \cdot P(i))$

And

$\forall i \in D \cdot P(i) = \neg (\exists i \in D \cdot \neg P(i))$

$a \wedge b$	$= b \wedge a$	"Commutativity"
$a \vee b$	$= b \vee a$	
$a \wedge (b \wedge c)$	$= (a \wedge b) \wedge c$	"Associativity"
$a \vee (b \vee c)$	$= (a \vee b) \vee c$	
$a \wedge (b \vee c)$	$= (a \wedge b) \vee (a \wedge c)$	"Distributivity"
$a \vee (b \wedge c)$	$= (a \vee b) \wedge (a \vee c)$	
$a \wedge \text{true}$	$= a$	
$a \wedge \text{false}$	$= \text{false}$	
$a \vee \text{false}$	$= a$	
$a \vee \text{true}$	$= \text{true}$	
$\neg (a \vee b)$	$= \neg a \wedge \neg b$	"de Morgan's Laws"
$\neg (a \wedge b)$	$= \neg a \vee \neg b$	
$\neg \neg a$	$= a$	
$a \Rightarrow b$	$= \neg a \vee b$	
$\text{true} \Rightarrow a$	$= a$	
$\text{false} \Rightarrow a$	$= \text{true}$	
$a \Leftrightarrow b$	$= (a \Rightarrow b) \wedge (b \Rightarrow a)$	

6.13.3 Sets

6.13.3.1 Set expressions

$\{\}$

The empty set. The set with no members.

$\{x1, x2, \dots, xn\}$

Set enumeration. The explicit description of a set, the set with exactly the listed elements as members.

$\{x \in S \mid P(x)\}$

Set comprehension. The implicit description of a set, the set containing those elements of the set S that satisfy the property (predicate) P .

$\{i, \dots, j\}$

Set range expression. A subset of the integers, all the integers between i and j inclusive.

$\{i, \dots, j\} \triangleq \{n \in \mathbb{N} \mid i \leq n \wedge n \leq j\}$

6.13.3.2 Set prefix operators

$\text{card } S$

The cardinality of a set. The number of elements in the set S .

$\mathcal{F} S$

The set of all finite subsets of the set S .

$\mathcal{F} x \triangleq \{s \mid s \subseteq S \wedge \exists n \in \mathbb{N} \cdot \text{card } s < n\}$

$\bigcup SS$

The distributed union operator. The union of all the sets contained in SS ; SS must be a set of sets.

$\bigcup SS \triangleq \{e \mid \exists s \in SS \cdot e \in s\}$

$\bigcap SS$

The distributed intersection operator. The intersection of all the sets contained in SS ; SS must be a set of sets.

$$\bigcap SS \triangleq \{e \mid \forall s \in SS \cdot e \in s\}$$

6.13.3.3 Set infix operators

$$S1 \cup S2$$

Set union. The set of elements, all of which are members of either $S1$ or $S2$ (or both).

$$S1 \cup S2 \triangleq \{e \mid e \in S1 \vee e \in S2\}$$

$$S1 \cap S2$$

Set intersection. The set of elements, all of which are members of both $S1$ and $S2$.

$$S1 \cap S2 \triangleq \{e \mid e \in S1 \wedge e \in S2\}$$

$$S1 - S2$$

Set difference. The set containing those elements of $S1$ that are not in $S2$.

$$S1 - S2 \triangleq \{e \mid e \in S1 \wedge e \notin S2\}$$

$$S1 \subseteq S2$$

The subset relation. All of the elements of $S1$ are also elements of $S2$.

$$S1 \subseteq S2 \triangleq \forall x \in S1 \cdot x \in S2$$

$$S1 \subset S2$$

The proper subset relation. The set $S1$ is a subset of the set $S2$ and $S2$ contains at least one element that is not in $S1$.

$$S1 \subset S2 \triangleq S1 \subseteq S2 \wedge S1 \neq S2$$

$$e \in S$$

Set membership. Tests if the element e is a member of the set S .

$$e \notin S$$

Negation of set membership. Tests if the element e is not a member of the set S .

6.13.4 Sequences

6.13.4.1 Sequence expressions

$$[]$$

The empty sequence. The sequence with no elements.

$$[e1, e2, \dots, en]$$

Sequence enumeration. An explicit sequence description (definition), the sequence with exactly the given elements in the order shown.

$$[e(i) \mid i \in S \cdot P(e(i))]$$

Sequence comprehension. The implicit description of a sequence, the sequence containing those elements indexed by the set S that satisfy the property (predicate) P .

Figure 1 — The set operations.

$l(i, \dots, j)$

Subsequence expression. The i^{th} through j^{th} element of the sequence l .

6.13.4.2 Sequence prefix operators

$\text{hd } l$

The head of a sequence. The first element of the sequence l ; the sequence must not be empty.

$\text{tl } l$

The tail of a sequence. The sequence obtained by removing the first element of l ; the sequence must not be empty.

$\text{last } l$

The last element of a sequence. The sequence must not be empty.

$\text{front } l$

The front of a sequence. The sequence obtained by removing all but the last element of l . The sequence must not be empty.

$\text{len } l$

Length of a sequence. The number of elements in the sequence l .

$\text{len } l \triangleq \text{if } l = [] \text{ then } 0 \text{ else } 1 + \text{len } \text{tl } l$

$\text{inds } l$

The set of indices of a sequence. The set of integers that can be used as valid indices for the sequence l .

$\text{inds } l \triangleq \{1, \dots, \text{len } l\}$

$\text{elems } l$

The set of elements that make up the sequence l .

$\text{elems } l \triangleq \{l(i) \mid 1 \leq i \leq \text{len } l\}$

$\text{conc } l$

The distributed concatenation operator. The elements of l , which must be sequences, are concatenated together.

$\text{conc } l \triangleq \text{if } l = [] \text{ then } [] \text{ else hd } l \curvearrowright \text{conc tl } l$

6.13.4.3 Sequence infix operators

$l1 \curvearrowright l2$

Sequence concatenation. The sequence containing the elements of $l1$ followed by the elements of $l2$.

$s \sqsubseteq l$

Sequence membership. Test if s is a subsequence of l .

$s \sqsubseteq l \triangleq \exists c, d \in X^* . c \curvearrowright s \curvearrowright d = l$

6.13.4.4 Sequence application

$l(i)$

Subscripting a sequence. The element of l with index i ($1 \leq i \leq \text{len } l$).

$l(i) \triangleq \text{if } i = 1 \text{ then hd } l \text{ else tl } l(i - 1)$

Figure 2 — The sequence operations.

6.13.5 Maps

A map, m consists of pairs of elements from a set D and from a set R , and a rule that associates with each element of D a unique element of R .

6.13.5.1 Map expressions

$\{\}$

The empty map. The mapping with no pairs.

$\{d1 \mapsto r1, d2 \mapsto r2, \dots, dn \mapsto rn\}$

Map enumeration. Explicit map definition, the map with the given pairs – the order of the pairs is not important.

$$\{d \mapsto f(d) \mid P(d)\}$$

Map comprehension. Implicit mapping definition, the mapping whose pairs are such that their first elements satisfy P , and for each pair, the second element is derived from the first by applying f .

6.13.5.2 Map prefix and postfix operators

$\text{dom } m$

The domain of a map m . The set of first elements of the pairs in the map m .

$\text{rng } m$

The range of a map m . The set of second elements of the pairs in the map m .

$$\text{rng } m \triangleq \{m(d) \mid d \in \text{dom } m\}$$

$\text{merge } M$

The merge of a set of maps M . The union of all the maps in the set M .

m^{-1}

Map inversion. The inverse of the mapping m ; the mapping must be one-to-one.

$$m^{-1}(x) \triangleq \text{let } y \text{ be st } m(y) = x \text{ in } y$$

6.13.5.3 Map infix operators

$m1 \cup m2$

Map union. A mapping with the information from the mapping $m1$ and the information from $m2$; the domains of the two maps must be disjoint.

$m1 \dagger m2$

Map overwrite. A mapping with all of the information from the mapping $m1$ and that information from $m2$ which is not overwritten by $m2$.

$$m1 \dagger m2 \triangleq \{d \mapsto r \mid d \in \text{dom } m2 \wedge r = m2(d) \vee d \in (\text{dom } m1 - \text{dom } m2) \wedge r = m1(d)\}$$

$S \triangleleft m$

Map domain restriction. A mapping with the pairs of mapping m whose first elements are also in the set S .

$$S \triangleleft m \triangleq \{d \mapsto m(d) \mid d \in (S \cap \text{dom } m)\}$$

$S - m$

Map domain subtraction. The mapping which maps those domain elements of m that are not in the set S into the same range of elements as m .

$$S - m \triangleq \{d \mapsto m(d) \mid d \in (\text{dom } m - S)\}$$

$S \triangleright m$

Map range restriction. The mapping which maps those domain elements of m into the range of elements that are in the set S .

$$m \triangleright S \triangleq \{d \mapsto m(d) \mid m(d) \in (S \cap \text{rng } m)\}$$

$S-m$

Map range subtraction. The mapping which maps those domain elements of m into the range of elements that are not in the set S .

$$m-S \triangleq \{d \mapsto m(d) \mid m(d) \in (\text{rng } m - S)\}$$

m^i

Map iteration. The mapping obtained by applying m to its argument i times; the range of m must be contained in its domain.

$$m^i \triangleq \text{if } i = 0 \text{ then } x \text{ else } m \uparrow (i - 1)m(x)$$

$m \circ n$

Map composition. The mapping that is equivalent to first applying the map n and then applying m to the result. The range of n must be contained in the domain of m .

$$m \circ n \triangleq m(n(x))$$

6.13.5.4 Map application

$m(d)$

Map application. The element of the range that corresponds to the element, d , of the domain.

Figure 3 — The map operations.

6.13.6 Records

6.13.6.1 Record expressions

Given the following composite object description:

$B :: \text{sel1} : \text{Set1}$
 $\text{sel2} : \text{Set2}$
 Set3

if $x \in B$ and $s1 \in \text{Set1}$ then

$$\mu(x, sel1 \mapsto s1).sel1 = s1$$

and

$$\mu(x, sel1 \mapsto s1).sel1 = s1$$

$$\mu(x, sel1 \mapsto s1).sel2 = x.s2$$

Figure 4 — The record operations.

7 System Modules

A system module does not have an actual definition or implementation provided by the user, but is built into the implementation to give access to features which cannot otherwise be expressed.

7.1 The Module SYSTEM

The system module **SYSTEM** gives access to the addressing and the underlying storage of variables.

7.1.1 The pseudo definition module of SYSTEM

The interface to **SYSTEM** behaves as if the following were its definition module:

TO DO — Improve comments in Definition Module.

```
DEFINITION MODULE SYSTEM;
  (* In what follows, <imp def> means an implementation-defined value *)
CONST
  LOCSPERBYTE    = <imp.def>  (* May not be present in all implementations *)
  LOCSPERWORD    = <imp.def>
  BITSPERBITSET  = <imp.def>

TYPE
  BITNUM;
  LOC;          (* The smallest addressable unit of storage *)
  ADDRESS = POINTER TO LOC;
  BYTE = ARRAY[0..LOCSPERBYTE-1] OF LOC; (* May not be present in all implementations *)
  WORD = ARRAY[0..LOCSPERWORD-1] OF LOC; (* NB. Cannot be indexed *)
  BITSET = SET OF BITNUM[0..BITSPERBITSET-1];
  MACHINEADDRESS = RECORD
    <an implementation-defined structure>
  END;

PROCEDURE ADDADR(addr : ADDRESS; offset : CARDINAL): ADDRESS;
  (* Return (offset + addr) or raise an exception *)

PROCEDURE ADDRESSVALUE(val : <some type>; ...): ADDRESS;
  (* Return an address constructed from the supplied parameters *)

PROCEDURE ADR(VAR v : <anytype>): ADDRESS;
  (* Return the address of v *)

PROCEDURE CAST(<type>; val : <anytype>): <type>;
  (* Present the expression denoted by val as the type <type> or *)
  (* raise an exception. Note that the presentation produces no *)
  (* extra instructions (but the evaluation of val may do so)    *)

PROCEDURE DIFADR(faddr,saddr : ADDRESS): INTEGER;
  (* Return (faddr - saddr) or raise an exception *)

PROCEDURE ROTATE(val : <a bit-set type or a system storage type>;
  shift : INTEGER): <type of first parameter>;
  (* Return a value such that for each bit b of result *)
  (* bit[b] = bit[(i - shift) mod bitsize(val)] of val *)
```

```

PROCEDURE SHIFT(val    : <a bit-set type or a system storage type>;
                shift  : INTEGER): <type of first parameter>;
    (* Return val Shifted up/down with zeros added as necessary *)

PROCEDURE SUBADR(addr : ADDRESS; offset : CARDINAL): ADDRESS;
    (* Return (addr - offset) or raise an exception *)

PROCEDURE TSIZE(<type>; ...): CARDINAL;
    (* Return the no of LOCS in <type>. (The extra parameters *)
    (* are used to distinguish variants in a variant record) *)
END SYSTEM.

```

CHANGE —

- a) The type LOC has been introduced as a mechanism to resolve the problem that, in many implementations of Modula-2, both **BYTE** and **WORD** occur in **SYSTEM**. Its introduction allows a consistent handling of both of these types, and also enables further **WORD**-like types to be introduced in an implementation, for example, where $\text{TSIZE}(\text{CARDINAL}) \neq \text{TSIZE}(\text{INTEGER})$.
- b) The type **BITNUM** has been introduced as a mechanism to resolve the problem of which size to choose for **BITSET**, and which representation of a basic data-type it should correspond to. Its introduction allows **BITSET**-like types to be introduced in an implementation.
- c) Variables of type **ADDRESS** are no longer (expression-) compatible with **CARDINAL** i.e. they can no longer occur in expressions that include arithmetic operators.

New functions **ADDADR**, **SUBADR**, and **DIFADR** have been introduced for address arithmetic.

These changes are to accommodate those machines where the interpretation of an address as a **CARDINAL** is not possible.

An implementation may extend the module **SYSTEM** by including additional types, procedures variables and constants appropriate to the facilities of the target hardware. The names **BYTE** and **LOCS PER BYTE** are reserved in **SYSTEM** for that purpose. These names shall only be used for appropriate target hardware entities.

Any procedure or function that is added to the module **SYSTEM** cannot be assigned to a procedure variable.

7.1.2 Types

The module **SYSTEM** provides types for manipulating machine words, bit-sets and addresses.

7.1.2.1 System Storage Types

A location is the smallest addressable unit of storage of an implementation, and shall occupy one unit (as defined by the **TSIZE** function). The location type shall have an implementation-dependent representation, and shall be denoted by the type identifier **LOC**.

The only operation directly defined on the location type shall be assignment and uses of the **SYSTEM** functions **SHIFT** and **ROTATE**.

Parameters of location type shall be treated specially; any type for which **TSIZE** returns the value 1 may be passed when such a parameter is declared.

Parameters of type **ARRAY OF LOC** shall be treated specially; any type except for **Z-TYPE**, **R-TYPE** or **C-TYPE** may be passed when such a parameter is declared. The translator shall construct an upper bound determined by the size of the item passed.

NOTE — There are special rules affecting parameter compatibility for system storage types. These are defined in Section 6.4.3, Parameter Compatibility.

The type **WORD** shall be defined in terms of **LOC** as follows:

```
LOCSPERWORD = <imp.def> (* an implementation-defined value *);  
WORD = ARRAY[0..LOCSPERWORD-1] OF LOC;
```

The type **BYTE**, if present, may be defined in terms of **LOC** as follows: if **TSIZE**(**BYTE**) = 1 then

```
LOCSPERBYTE = 1;  
BYTE = LOC;
```

and if **TSIZE**(**BYTE**) > 1 then

```
LOCSPERBYTE = <imp.def> (* an implementation-defined value *);  
BYTE = ARRAY[0..LOCSPERBYTE-1] OF LOC;
```

The array structure of the type **WORD** and **BYTE** shall not be visible outside of the module — variables of type **BYTE** and **WORD** shall not be subscripted.

7.1.2.2 Address Type

The address type shall be a type that has the following declaration:

```
TYPE ADDRESS = POINTER TO LOC;
```

and corresponds to the address of the **LOC**.

As well as the operations that are normally applicable to variables of pointer types, the only other valid operations on variables of the address type shall be the system function procedures **ADDADR**, **SUBADR**, and **DIFADR**. In the case of **ADDADR** and **SUBADR** the result type of these procedures shall be a value of address type; in the case of **DIFADR** the result type shall be of signed type.

The address type shall be assignment compatible with all pointer types.

A parameter of address type shall be parameter compatible with all pointer types, thus a formal parameter (either value or variable) of address type is compatible with an actual parameter of any pointer type and vice-versa.

7.1.2.3 Machine Address Type

A machine address type shall be a record type with an implementation defined structure. Values of this type may be used in the optional expression that follows the identifier in a variable declaration to fix the storage location of that variable.

7.1.2.4 Bit Number Type

A value of bit number type shall be an element of an implementation-defined finite subset of the non-negative whole numbers. The bit number type shall be a subrange of a whole number type, and shall be denoted by the type identifier **BITNUM**. If x is a value of bit number type then

$$x \in \{0, \dots, \mathbf{MAX}(\mathbf{BITNUM})\}$$

If the number of bits in a **LOC** is n and the **TSIZE** of the largest scalar type is m , then

$$\mathbf{MAX}(\mathbf{BITNUM}) \geq n \times m$$

The ordinal number of a value of bit number type shall be the value itself.

7.1.2.5 Bit-Set Type

A set type which has a member type of either bit number type or a subrange whose host type is bit number type shall have a type called bit-set type.

The type BITSET shall be defined in terms of BITNUM as follows:

```
BITSPERBITSET = <imp.def> (* an implementation-defined value *);
BITSET = SET OF BITNUM[0..BITSPERBITSET-1];
```

Values of type bit-set shall be considered as sequences of bits, numbered according to the whole numbers in the member type of the bit-set. For a value of type bit-set considered as a sequence, bit number n is set (i.e. has the value 1) if value n is in the set denoted by the value, and bit number n is unset or clear (has the value 0) if the value n is not in the set denoted by the value. An implementation may restrict the lower and upper bounds of the interval used to define a set of BITNUM. For example the lower bound may be restricted to 0 and the upper bound to a value that is a multiple of the number of bits in a LOC.

NOTE — With this interpretation the following set operations are equivalent to bit operations:

+	union	bitwise or operation on two bit sequences
-	set difference	the bits of the first sequence that are not in the second
*	intersection	bitwise and operation on two bit sequences
/	symmetric set difference	bitwise exclusive-or operation on two bit sequences
INCL(s,x)		set bit x (to 1)
EXCL(s,x)		clear bit x (to 0)
=	equality	are the two bit sequences equal?
<>	inequality	are the two bits sequences different?
<=	less than or equal	is the first bit sequence included in the second?
>=	greater than or equal	does the first bit sequence include the second?
x IN s	membership	is bit number x set to 1?

Examples

With the declarations given below

```
TYPE BITBYTE = SET OF BITNUM[0..7] ;
...
VAR byte : BITBYTE;
```

A value of type BITBYTE is equivalent to the following sequence of bits:

									byte
bit number	7	6	5	4	3	2	1	0	

The assignment

```
byte := BITBYTE{0,3,7};
```

will give:

	1	0	0	0	1	0	0	1	byte
bit number	7	6	5	4	3	2	1	0	

and the call:

```
INCL(byte,4);
```

will give:

	1	0	0	1	1	0	0	1	byte
bit number	7	6	5	4	3	2	1	0	

TEMPORARY NOTE — Should $\text{CAST}(\text{BITSET}, 1) = \text{BITSET}\{0\}$ be true? Note that this would make machines ‘little-endian’, which is against the philosophy of the definition given above, and the definitions of **SHIFT** and **ROTATE**.

7.1.3 The **SYSTEM** Functions

Abstract Syntax

types

$$\begin{aligned} \text{System-function-designator} = & \text{Addadr-designator} \\ & | \text{Addressvalue-designator} \\ & | \text{Adr-designator} \\ & | \text{Cast-designator} \\ & | \text{Difadr-designator} \\ & | \text{Rotate-designator} \\ & | \text{Shift-designator} \\ & | \text{Subadr-designator} \\ & | \text{Tsize-designator} \end{aligned}$$

Static Semantics

functions

$\text{wf-system-function-designator} : \text{System-function-designator} \rightarrow \text{Actual-parameters} \rightarrow \text{Environment} \rightarrow \text{Typed}$

$\text{wf-system-function-designator}(\text{sfd})(\text{aps})\rho \triangleq$

cases sfd :

```
??   mk-Addadr-designator()      → wf-addadr(sfd)ρ,
??   mk-Addressvalue-designator(-) → wf-addressvalue(sfd)ρ,
??   mk-Adr-designator()        → wf-adr(sfd)ρ,
??   mk-Cast-designator(-, -)    → wf-cast(sfd)ρ,
??   mk-Difadr-designator()      → wf-difadr(sfd)ρ,
??   mk-Rotate-designator(-)     → wf-rotate(sfd)ρ,
??   mk-Shift-designator(-)      → wf-shift(sfd)ρ,
??   mk-Subadr-designator()      → wf-subadr(sfd)ρ,
??   mk-Tsize-designator()       → wf-tsize(sfd)ρ
end;
```

$\text{t-system-function-designator} : \text{System-function-designator} \rightarrow \text{Actual-parameters} \rightarrow \text{Environment} \rightarrow \text{Typed}$

$\text{t-system-function-designator}(\text{sfd})(\text{aps})\rho \triangleq$

cases sfd :

```
??   mk-Addadr-designator()      → t-addadr(sfd)ρ,
??   mk-Addressvalue-designator(-) → t-addressvalue(sfd)ρ,
??   mk-Adr-designator()        → t-adr(sfd)ρ,
??   mk-Cast-designator(-, -)    → t-cast(sfd)ρ,
??   mk-Difadr-designator()      → t-difadr(sfd)ρ,
??   mk-Rotate-designator(-)     → t-rotate(sfd)ρ,
??   mk-Shift-designator(-)      → t-shift(sfd)ρ,
??   mk-Subadr-designator()      → t-subadr(sfd)ρ,
??   mk-Tsize-designator()       → t-tsize(sfd)ρ
end
```

Dynamic Semantics

operations

$m\text{-system-function-designator} : \text{System-function-designator} \rightarrow \text{Actual-parameters} \rightarrow \text{Environment} \xrightarrow{o} \text{Value}$

$m\text{-system-function-designator}(sfd)(aps)\rho \triangleq$

cases sfd :

```

??      mk-Addadr-designator()      → m-addadr(sfd) ρ,
323     mk-Addressvalue-designator(-) → m-addressvalue(sfd) ρ,
??      mk-Adr-designator()         → m-adr(sfd) ρ,
??      mk-Cast-designator(-, -)    → m-cast(sfd) ρ,
??      mk-Difadr-designator()      → m-difadr(sfd) ρ,
??      mk-Rotate-designator(-)     → m-rotate(sfd) ρ,
??      mk-Shift-designator(-)      → m-shift(sfd) ρ,
??      mk-Subadr-designator()      → m-subadr(sfd) ρ,
??      mk-Tsize-designator()       → m-tsize(sfd) ρ
end

```

7.1.3.1 The ADDADR Function

ADDADR is a function procedure that calculates a new value of type address by adding the first parameter, which is an address, to the second parameter, which is an offset. The addition is carried out in an implementation-defined manner.

Abstract Syntax

types

$Addadr ::$

Static Semantics

A call of **ADDADR** shall have two actual parameters. The first parameter shall be an expression whose result type is the address type. The second parameter shall be an expression whose result type is the unsigned type. The result type shall be of type address.

functions

$wf\text{-addadr-designator} : Addadr\text{-designator} \rightarrow \text{Actual-parameters} \rightarrow \text{Environment} \rightarrow \mathbb{B}$

$wf\text{-addadr-designator}(mk\text{-Addadr}())(aps)\rho \triangleq$

len $aps = 2 \wedge$

let $[addr, offset] = aps$ in

152 $is\text{-Expression}(addr) \wedge t\text{-expression}(addr)\rho = \text{ADDRESS-TYPE} \wedge$

152 $is\text{-Expression}(offset) \wedge t\text{-expression}(offset)\rho = \text{UNSIGNED-TYPE};$

$t\text{-addadr-designator} : Addadr\text{-designator} \rightarrow \text{Actual-parameters} \rightarrow \text{Environment} \rightarrow \text{Typed}$

$t\text{-addadr-designator}(mk\text{-Addadr}())(aps)\rho \triangleq$

ADDRESS-TYPE

Dynamic Semantics

A call of **ADDADR** shall return an address calculated by adding the first parameter to the second in an implementation-defined manner.

It may be an exception if the **ADDADR** function increments an address out of address range, or if the address space is non-contiguous.

operations

$$m\text{-addadr-designator} : \text{Addadr-designator} \rightarrow \text{Arguments} \rightarrow \text{Environment} \xrightarrow{o} \text{Value}$$

$$m\text{-addadr-designator} (mk\text{-Addadr}())(args)\rho \triangleq$$

$$\text{let } [addr, offset] = args \text{ in}$$

$$\text{if } can\text{-add}(addr, offset)\rho$$

$$\text{then } add\text{-offset}(addr, offset)\rho$$

$$\text{else } non\text{-mandatory-exception}(\text{ADDRESS-ARITHMETIC})$$

Auxiliary Definitions

operations

$$can\text{-add} : \text{Variable} \times \mathbb{N} \times \text{Environment} \xrightarrow{o} \mathbb{B}$$

$$can\text{-add}(var, offset)\rho \triangleq$$

implementation-defined;

$$add\text{-offset} : \text{Variable} \times \mathbb{N} \times \text{Environment} \xrightarrow{o} \text{Value}$$

$$add\text{-offset}(var, offset)\rho \triangleq$$

implementation-defined — The result is an implementation-defined value of address type

7.1.3.2 The ADDRESSVALUE Function

ADDRESSVALUE is a function procedure that converts its argument(s) to a value of address type.

Abstract Syntax

types

$$\text{Addressvalue} ::$$

Static Semantics

A call of ADDRESSVALUE shall have an implementation-defined number of actual parameters.

functions

$$wf\text{-addressvalue-designator} : \text{Addressvalue-designator} \rightarrow \text{Actual-parameters} \rightarrow \text{Environment} \rightarrow \mathbb{B}$$

$$wf\text{-addressvalue-designator} (mk\text{-Addressvalue}())(aps)\rho \triangleq$$

Implementation-defined;

$$t\text{-addressvalue-designator} : \text{Addressvalue-designator} \rightarrow \text{Actual-parameters} \rightarrow \text{Environment} \rightarrow \text{Typed}$$

$$t\text{-addressvalue-designator} (mk\text{-Addressvalue}())(aps)\rho \triangleq$$

ADDRESS-TYPE

Dynamic Semantics

The argument(s) are converted to an address in an implementation-defined manner.

operations

$$m\text{-addressvalue} : \text{Addressvalue-designator} \rightarrow \text{Arguments} \rightarrow \text{Environment} \xrightarrow{o} \text{Value}$$

$$m\text{-addressvalue} (mk\text{-Addressvalue}(), type)(args)\rho \triangleq$$

is not yet defined

— Implementation-defined

CHANGE — A new function ADDRESSVALUE has been introduced to replace assignment-compatibility between ADDRESS and INTEGER and CARDINAL

7.1.3.3 The ADR Function

ADR is a function procedure that takes a variable parameter and returns its address.

Abstract Syntax

types

Adr-designator ::

Static Semantics

A call of ADR shall have one actual parameter which shall be a variable parameter. The parameter may be dereferenced, selected, or subscripted. The result type shall be an address.

functions

wf-adr-designator : *Adr-designator* \rightarrow *Actual-parameters* \rightarrow *Environment* \rightarrow \mathbb{B}

wf-adr-designator (*mk-Adr*())(*aps*) $\rho \triangleq$

len *aps* = 1 \wedge

let [*ap*] = *aps* in

is-Variable-designator(*ap*);

t-adr-designator : *Adr-designator* \rightarrow *Actual-parameters* \rightarrow *Environment* \rightarrow *Typed*

t-adr-designator (*mk-System-function-designator*(ADR, NIL))(*aps*) $\rho \triangleq$

ADDRESS-TYPE

Dynamic Semantics

A call of ADR shall return the address of the variable.

operations

m-adr-designator : *Adr-designator* \rightarrow *Arguments* \rightarrow *Environment* \xrightarrow{o} *Value*

m-adr-designator (*mk-Adr*())(*args*) $\rho \triangleq$

let [*arg*] = *args* in

?? *address-of*(*arg*) ρ

Auxiliary Definitions

functions

address-of : *Variable* \times *Environment* \xrightarrow{o} *Value*

address-of (*arg*) \triangleq

is not yet defined

Properties

If

```

TYPE
  Val      = ... ;
  PtoVal = POINTER TO Val;
...
VAR
  v : Val;
  p : PtoVal;
...
p := CAST(PtoVal,ADR(v))

```

Then $p\uparrow$ and v designate the same variable.

7.1.3.4 The CAST Function

CAST is a function procedure that has a type name as the first parameter and a value of any non-literal type as the second. It returns a value of the type associated with the type name that is the first parameter.

Abstract Syntax

types

$$\begin{array}{l} \textit{Cast-designator} :: \textit{ttype} : \textit{Type} \\ \phantom{\textit{Cast-designator} ::} \textit{stype} : \textit{Type} \end{array}$$

Static Semantics

functions

$$\begin{array}{l} \textit{wf-cast-designator} : \textit{Cast-designator} \rightarrow \textit{Actual-parameters} \rightarrow \textit{Environment} \rightarrow \mathbb{B} \\ \textit{wf-cast-designator}(\textit{mk-Cast-designator}(\textit{ttype}, \textit{stype}))(\textit{aps})\rho \triangleq \\ \quad \text{len } \textit{aps} = 2 \wedge \\ \quad \text{let } [qid, \textit{expr}] = \textit{aps} \text{ in} \\ 270 \quad \textit{is-type}(qid)\rho \wedge \\ 271 \quad \textit{type-of}(qid)\rho = \textit{ttype} \wedge \\ \quad \textit{is-Expression}(\textit{expr}) \wedge \\ 152 \quad \textit{t-expression}(\textit{expr})\rho = \textit{stype} \wedge \\ 152 \quad \textit{t-expression}(\textit{expr})\rho \notin \{\mathbb{Z}\text{-TYPE}, \mathbb{R}\text{-TYPE}, \mathbb{C}\text{-TYPE}\} \end{array}$$

functions

$$\begin{array}{l} \textit{t-cast-designator} : \textit{Cast-designator} \rightarrow \textit{Actual-parameters} \rightarrow \textit{Environment} \rightarrow \textit{Typed} \\ \textit{t-cast-designator}(\textit{mk-Cast-designator}(\textit{ttype}, \textit{stype}))(\textit{aps})\rho \triangleq \\ 283 \quad \textit{host-type-of}(\textit{type})\rho \end{array}$$

Language Clarification

The result type of the value to be ‘cast’ cannot be a number literal.

Dynamic Semantics

The value of the second argument shall be converted, using a possibly unsafe conversion, to a value of the type denoted by the first argument.

If the number of storage units that the value occupies is the same as the number of storage units used by a variable of the type described by the first parameter, then the bit pattern representation of the new value shall be the same

as the bit pattern representation of the old value; otherwise the new value and the old value shall have the same bit pattern representation for a size that is equal to the smaller of the number of storage units.

There are alignment problems with some applications of **CAST** that could cause an exception to occur, implementations may raise an exception in these circumstances.

It may be possible that the value produced as a result of the **CAST** function may not be a valid value for the target type, an implementation may raise an exception in those circumstances either at that point in the program, or later.

operations

```

m-cast-designator : Cast-designator → Arguments → Environment → Value
m-cast-designator (mk-Cast-designator(ttype, stype))(args)ρ  $\triangleq$ 
  let [-, val] = args in
??   return cast-value(ttype, val)

```

Auxiliary Definitions

functions

```

cast-value : Typed × Value → Value
cast-value (type, val)  $\triangleq$ 
  is not yet defined

```

Properties

The meaning of **CAST**(*TargetType*, *SourceExpression*) is as follows:

Let **ARRAY**[0..*t*] **OF** **LOC** correspond to the *TargetType* where *t* = **SIZE**(*TargetType*)− 1

Let **ARRAY**[0..*v*] **OF** **LOC** correspond to the *SourceExpression* where *v* = **SIZE**(Typeof*SourceExpression*) − 1

The meaning of **CAST** is a value of type *TargetType* that when interpreted as an **ARRAY**[0..*t*] **OF** **LOC** has the same values as the elements from 0 to min(*t*,*v*) of the value of the *SourceExpression* interpreted as an **ARRAY**[0..*v*] **OF** **LOC** and if *t*>*v*, the remaining elements of the array from *v* to *t* are undefined.

If (assuming **MAX**(**BITNUM**) is large enough):

```

(* n is the number of bits in a LOC *)
CONST BITSperxType = TSIZE(xType)*n ;

VAR x : xType;

TYPE BITxType = SET OF BITNUM[0..BITSperxType-1]

TSIZE(anyType)=TSIZE(xType)

then:

CAST(BITxType,CAST(anyType,x)) = CAST(BITxType,x)

and

CAST(xTYPE,x) = x

```

CHANGE — The new function **CAST** has been introduced for unsafe type conversion.

7.1.3.5 The DIFADR Function

DIFADR is a function procedure that calculates the offset of one address from another. The offset is calculated by subtracting the second address from the first in an implementation-defined manner.

Abstract Syntax

types

Difadr-designator ::

Static Semantics

A call of **DIFADR** shall have two actual parameters. Both parameters shall be expressions of the address type. The result type shall be of the signed type.

functions

```

wf-difadr-designator : Difadr-designator → Actual-parameters → Environment →  $\mathbb{B}$ 
wf-difadr-designator (mk-Difadr-designator())(aps)  $\rho \triangleq$ 
  len aps = 2  $\wedge$ 
  let [fstaddr, secaddr] = aps in
152   is-Expression(fstaddr)  $\wedge$  t-expression (fstaddr)  $\rho$  = ADDRESS-TYPE  $\wedge$ 
152   is-Expression(secaddr)  $\wedge$  t-expression (secaddr)  $\rho$  = ADDRESS-TYPE

```

functions

```

t-difadr-designator : Difadr-designator → Actual-parameters → Environment → Typed
t-difadr-designator (mk-Difadr-designator())(aps)  $\rho \triangleq$ 
  SIGNED-TYPE

```

Dynamic Semantics

A call of **DIFADR** shall return a signed number calculated by subtracting the second parameter from the first in an implementation-defined manner.

It may be an exception if the result of the **DIFADR** function is out of range of the signed type, or if the address space is non-contiguous.

operations

```

m-difadr-designator : Difadr-designator → Arguments → Environment  $\xrightarrow{o}$  Value
m-difadr-designator (mk-Difadr-designator())(args)  $\rho \triangleq$ 
  let [fstaddr, secaddr] = args in
??   if is-comparable-addresses(fstaddr, secaddr)  $\rho$ 
327   then def offset = address-difference(fstaddr, secaddr)  $\rho$ ;
162       get-whole-result(SIGNED-TYPE, offset)
306   else non-mandatory-exception(ADDRESS-ARITHMETIC)

```

Auxiliary Definitions

operations

```

is-comparable-addresses : Variable  $\times$  Variable → Environment  $\xrightarrow{o}$   $\mathbb{B}$ 
is-comparable-addresses (fvar, svar)  $\rho \triangleq$ 
??    $\exists n \in \mathbb{N} \cdot \text{add-offset}(svar, n) = fvar \vee$ 
??    $\exists m \in \mathbb{N} \cdot \text{add-offset}(fvar, m) = svar;$ 

```


$$\text{address-difference} : \text{Variable} \times \text{Variable} \rightarrow \text{Environment} \xrightarrow{o} \mathbb{Z}$$

$$\text{address-difference}(fvar, svar)\rho \triangleq$$

```

??   if  $\exists n \in \mathbb{N} \cdot \text{add-offset}(svar, n) = fvar$ 
??   then let  $offset$  be st  $\text{add-offset}(svar, offset) = fvar$  in
      return  $offset$ 
??   else let  $offset$  be st  $\text{add-offset}(fvar, offset) = svar$  in
      return  $- offset$ 

```

Properties

Given the declarations:

```

VAR
  adr  : ADDRESS;
  n    : CARDINAL;

```

the following properties shall hold if an exception is not raised:

SUBADR(ADDADR(adr,n),n) = adr

ADDADR(SUBADR(adr,n),n) = adr

DIFADR(ADDADR(adr,n),adr) = n

DIFADR(adr,SUBADR(adr,n)) = n

7.1.3.6 The ROTATE Function

ROTATE is a function procedure which has as a first parameter an expression whose result type is either the bit-set type or a system storage type. With the interpretation of a value of bit-set type as a sequence of bits, **ROTATE** rotates the bit sequence up or down by the number of places given by the absolute value of the second parameter. The direction is down if the sign of the value of the second parameter is negative, otherwise the direction is up.

Abstract Syntax

types

Rotate-designator :: *rtype* : *Typed*

Static Semantics

A call of **ROTATE** shall have two actual parameters. The first parameter which shall be an expression of a bit-set type or a system storage type, and the second parameter shall be an expression of the signed type.

The result type shall be the type of the first parameter.

An implementation may impose restrictions on the system storage types which are allowed as the first parameter.

functions

$wf\text{-}rotate\text{-}designator : Rotate\text{-}designator \rightarrow Actual\text{-}parameters \rightarrow Environment \rightarrow \mathbb{B}$

$wf\text{-}rotate\text{-}designator(mk\text{-}Rotate\text{-}designator(type))(aps)\rho \triangleq$
 $\quad \text{len } aps = 2 \wedge$

$\quad \text{let } [expr, shift] = aps \text{ in}$

152 $\quad is\text{-}Expression(expr) \wedge type = t\text{-}expression(expr)\rho \wedge$

266 $\quad (base\text{-}type\text{-}of(type)\rho = \text{BIT-SET-TYPES} \vee is\text{-}system\text{-}storage(type)\rho) \wedge$

152 $\quad is\text{-}Expression(shift) \wedge t\text{-}expression(shift)\rho = \text{SIGNED-TYPE};$

$t\text{-}rotate\text{-}designator : Rotate\text{-}designator \rightarrow Actual\text{-}parameters \rightarrow Environment \rightarrow Typed$

$t\text{-}rotate\text{-}designator(mk\text{-}Rotate\text{-}designator(type))(aps)\rho \triangleq$
 $\quad type$

Dynamic Semantics

If the first parameter is of the system storage type, it shall be cast to the bit-set type.

The elements of the set of values that is the value of the first actual parameter to the procedure call shall all be increased by the value of the second argument, if it is positive, or decreased by the absolute value of the second argument, if it is negative. If any of the new values so produced does not belong to the range of values defined by the base type of the set, it is scaled by a value based on the minimum value of the base type, and the size of the range of the base type.

NOTE — With the interpretation of a value of type bit-set as a sequence of bits, the rotate operation can be considered as a rotate of the bits upwards, or a rotate of the bits downwards according to the value of the second operand.

operations

$m\text{-}rotate\text{-}designator : Rotate\text{-}designator \rightarrow Arguments \rightarrow Environment \xrightarrow{o} Value$

$m\text{-}rotate\text{-}designator(mk\text{-}Rotate\text{-}designator(type))(args)\rho \triangleq$

$\quad \text{let } [val, rot] = args \text{ in}$

283 $\quad \text{if } base\text{-}type\text{-}of(type) = \text{BIT-SET-TYPES}$

289 $\quad \text{then let } min = minimum(type)\rho \text{ in}$

288 $\quad \quad \text{let } max = maximum(type)\rho \text{ in}$

329 $\quad \quad \text{return } rotate(min, max, val, shift)$

?? $\quad \text{else let } bval = cast\text{-}to\text{-}bit\text{-}set(val) \text{ in}$

329 $\quad \quad \text{let } res = rotate(0, bit\text{-}size\text{-}of(bval) - 1, bval, rot) \text{ in}$

?? $\quad \quad \text{let } fval \text{ be st } cast\text{-}to\text{-}bitset(fval) = res \text{ in return } fval$

Auxiliary Definitions

functions

$rotate : Value \times Value \times Value \times Value \rightarrow Value$

$rotate(min, max, val, rot) \triangleq$

$\quad \text{let } d = max - min + 1 \text{ in}$

$\quad \text{let } result = \{((x + rot - min) \bmod d) + min \mid x \in val\} \text{ in}$

$\quad mk\text{-}Set\text{-}value(result)$

CHANGE — A new function ROTATE has been introduced for manipulating bit-set values. It does not rely on which end of a (binary) word is numbered 0.

7.1.3.7 The SHIFT Function

SHIFT is a function procedure which has as a first parameter an expression whose result type is either the bit-set type

or a system storage type. With the interpretation of a value of bit-set type as a sequence of bits, **SHIFT** shifts the bit sequence up or down by the number of places given by the absolute value of the second parameter, adding zeros as necessary. The direction is down if the sign of the value of the second parameter is negative, otherwise the direction is up.

Abstract Syntax

types

$Shift\text{-}designator :: rtype : Typed$

Static Semantics

A call of **SHIFT** shall have two actual parameters. The first parameter which shall be an expression whose result is a bit-set type or a system storage type, and the second parameter shall be an expression of the signed type.

The result type shall be the type of the first parameter.

An implementation may impose restrictions on the system storage types which are allowed as the first parameter.

functions

```

wf-shift-designator : Shift-designator → Actual-parameters → Environment → B
wf-shift-designator (mk-Shift-designator(type))(aps)ρ  $\triangleq$ 
  len aps = 2 ∧
  let [expr, shift] = aps in
  is-Expression(expr) ∧ type = is-expression(expr)ρ ∧
  (base-type-of(type)ρ = BIT-SET-TYPES ∨ is-system-type(type)ρ) ∧
  is-Expression(shift) ∧ t-expression(shift)ρ = SIGNED-TYPE

```

functions

```

t-shift-designator : Shift-designator → Actual-parameters → Environment → Typed
t-shift-designator (mk-Shift-designator(type))(aps)ρ  $\triangleq$ 
  host-type-of(type)ρ

```

Dynamic Semantics

The set of values that is the value of the expression that is the first actual parameter to the procedure call shall all be increased by value of the second argument, if it is positive, or decreased by the value of the second argument, if it is negative. If any of the new values so produced do not belong to the range of values defined by the base type of the set, they are discarded.

NOTE — With the interpretation of a value of type bit-set as a sequence of bits, the shift operation can be considered as a logical shift up operation or a logical shift down operation according to the value of the second operand.

operations

```

m-shift-designator : Shift-designator → Arguments → Environment  $\xrightarrow{o}$  Value
m-shift-designator (mk-Shift-designator(type))(args)ρ  $\triangleq$ 
  let [val, shift] = args in
  if base-type-of(type) = BIT-SET-TYPES
  then let min = minimum(type)ρ in
       let max = maximum(type)ρ in
       return shift(min, max, val, shift)
  else let bval = cast-to-bit-set(val) in
       let res = shift(0, bit-size-of(bval) - 1, bval, shift) in
       let fval be st cast-to-bitset(fval) = res in return fval

```

Auxiliary Definitions

functions

$$\begin{aligned} & \textit{shift} : \textit{Value} \times \textit{Value} \times \textit{Value} \times \textit{Value} \rightarrow \textit{Value} \\ & \textit{shift}(\textit{min}, \textit{max}, \textit{val}, \textit{shift}) \triangleq \\ & \quad \text{let } d = \textit{max} - \textit{min} + 1 \text{ in} \\ & \quad \text{let } \textit{result} = \{x + \textit{shift} \mid x \in \textit{val} \wedge \textit{min} \leq x + \textit{shift} \wedge x + \textit{shift} \leq \textit{max}\} \text{ in} \\ & \quad \textit{mk-Set-value}(\textit{result}) \end{aligned}$$

CHANGE — The new function **SHIFT** has been introduced for manipulating bit-set values. It does not rely on which end of a (binary) word is numbered 0.

7.1.3.8 The SUBADR Function

SUBADR is a function procedure that calculates a new value of type address by subtracting the second parameter, which is an offset, from the first parameter which is an address. The subtraction is carried out in an implementation-defined manner.

Abstract Syntax

types

Subadr-designator ::

Static Semantics

A call of **SUBADR** shall have two actual parameters. The first parameter shall be an expression whose result type is address. The second parameter shall be an expression of unsigned type. The result type shall be of type address.

functions

$$\begin{aligned} & \textit{wf-subadr-designator} : \textit{Subadr-designator} \rightarrow \textit{Actual-parameters} \rightarrow \textit{Environment} \rightarrow \mathbb{B} \\ & \textit{wf-subadr-designator}(\textit{mk-Subadr-designator}())(\textit{aps})\rho \triangleq \\ & \quad \text{len } \textit{aps} = 2 \wedge \\ & \quad \text{let } [\textit{addr}, \textit{offset}] = \textit{aps} \text{ in} \\ 152 \quad & \textit{is-Expression}(\textit{addr}) \wedge \textit{t-expression}(\textit{addr})\rho = \text{ADDRESS-TYPE} \wedge \\ 152 \quad & \textit{is-Expression}(\textit{offset}) \wedge \textit{t-expression}(\textit{offset})\rho = \text{UNSIGNED-TYPE} \end{aligned}$$

functions

$$\begin{aligned} & \textit{t-subadr-designator} : \textit{Subadr-designator} \rightarrow \textit{Actual-parameters} \rightarrow \textit{Environment} \rightarrow \textit{Typed} \\ & \textit{t-subadr-designator}(\textit{mk-Subadr-designator}())(\textit{aps})\rho \triangleq \\ & \quad \text{ADDRESS-TYPE} \end{aligned}$$

Dynamic Semantics

A call of **SUBADR** shall return an address calculated by subtracting the second parameter from the first in an implementation-defined manner.

It may be an exception if the **SUBADR** function decrements an address out of address range, or if the address space is non-contiguous.

operations

```

m-subadr-designator : Subadr-designator → Arguments → Environment → Value
m-subadr-designator (mk-Subadr-designator())(args) $\rho \triangleq$ 
  let [addr, offset] = args in
323   if can-add(addr, offset)  $\rho$ 
332   then subtract-offset(addr, offset)  $\rho$ 
306   else non-mandatory-exception(ADDRESS-ARITHMETIC)

```

Auxiliary Definitions

operations

```

can-subtract : Variable ×  $\mathbb{N}$  × Environment  $\xrightarrow{o}$   $\mathbb{B}$ 
can-subtract (var, offset) $\rho \triangleq$ 
  implementation-defined;

subtract-offset : Variable ×  $\mathbb{N}$  → Environment  $\xrightarrow{o}$  Value
subtract-offset (var, offset) $\rho \triangleq$ 
??   let nvar be st add-offset (nvar, offset) = var in
      nvar

```

7.1.3.9 The TSIZE Function

TSIZE is a function procedure which has two versions. The first has a type name as a parameter; it returns the size in terms of the smallest addressable unit of storage. The second version has a type of a record as the first parameter and one or more constant expressions as the remaining parameters; the values of these constant expressions must be associated with tags of any variant components. It returns the size of the variant record in terms of the smallest addressable unit of storage that would be needed to store the record with the tag fields having the values supplied in the call.

Abstract Syntax

types

Tsize-designator ::

Static Semantics

The first, and possibly only parameter, shall be a qualident denoting the name of a type.

If the second and subsequent parameters are present, they shall be constant expressions and the first parameter shall denote a record type. The types of the constant expressions shall correspond to the types of the tag fields contained in the record type (and not to the values of tag fields contained in record types contained in the record).

The result shall be of ZZ-type.

functions

$wf\text{-}tsize\text{-}designator : Tsize\text{-}designator \rightarrow Actual\text{-}parameters \rightarrow Environment \rightarrow \mathbb{B}$

$wf\text{-}tsize\text{-}designator(mk\text{-}Tsize())(aps)\rho \triangleq$

$\text{len } aps \geq 1 \wedge$

$\text{let } [qid] \curvearrowright exprs = aps \text{ in}$

270 $is\text{-}type(qid)\rho \wedge$

$(exprs = [] \vee$

271 $(\text{let } rs = structure\text{-}of(qid)\rho \text{ in}$

$is\text{-}Record\text{-}structure(rs) \wedge$

$\text{len } exprs \geq 1 \wedge$

?? $is\text{-}tags\text{-}in\text{-}fields\text{-}list(rs, exprs) \wedge$

214 $\forall expr \in exprs \cdot is\text{-}constant\text{-}expression(expr)\rho))$

functions

$t\text{-}tsize\text{-}designator : Tsize\text{-}designator \rightarrow Actual\text{-}parameters \rightarrow Environment \rightarrow Typed$

$t\text{-}tsize\text{-}designator(mk\text{-}Tsize\text{-}designator())(aps)\rho \triangleq$

$\mathbb{Z}\text{-TYPE}$

Dynamic Semantics

A call of **TSIZE** shall return an unsigned number based on the number of smallest addressable units required by a variable of the type given by the first (and possibly only) parameter.

operations

$m\text{-}tsize\text{-}designator : Tsize\text{-}designator \rightarrow Arguments \rightarrow Environment \rightarrow Value$

$m\text{-}tsize\text{-}designator(mk\text{-}Tsize\text{-}designator())(args)\rho \triangleq$

$\text{let } [arg] \curvearrowright vals = args \text{ in}$

335 $\text{let } val = tsize(arg, vals) \text{ in}$

162 $get\text{-}whole\text{-}result(\mathbb{Z}\text{-TYPE}, val)$

Auxiliary Functions

functions

$is\text{-}tags\text{-}in\text{-}field\text{-}list : Field\text{-}list \times Expression^* \rightarrow Environment \rightarrow \mathbb{B}$

$is\text{-}tags\text{-}in\text{-}field\text{-}list(fsl, exprs)\rho \triangleq$

$\text{if } fsl = []$

$\text{then } exprs = []$

333 $\text{else } is\text{-}tags\text{-}in\text{-}fields(\text{hd } fsl, exprs)\rho \wedge$

334 $\text{let } size = number\text{-}of\text{-}tags\text{-}in\text{-}field\text{-}list(\text{hd } fsl, exprs)\rho \text{ in}$

std $\text{let } reexprs = rest(exprs, size + 1) \text{ in}$

333 $is\text{-}tags\text{-}in\text{-}field\text{-}list(\text{tl } fsl, reexprs)\rho$

annotations

Check that the tag fields contained in a fields-list match the values given by an expression sequence.

functions

$is\text{-}tags\text{-}in\text{-}fields : Fields \times Expression^* \rightarrow Environment \rightarrow \mathbb{B}$

$is\text{-}tags\text{-}in\text{-}fields(fs, exprs) \triangleq$

$(is\text{-}Fixed\text{-}fields(fs) \rightarrow \text{true},$

334 $is\text{-}Variant\text{-}fields(fs) \rightarrow is\text{-}tags\text{-}in\text{-}variant\text{-}fields(fs, exprs)\rho)$

annotations

Check that the tag fields contained in a fields match the values given by an expression sequence.

functions

$is\text{-}tags\text{-}in\text{-}variant\text{-}fields : Variant\text{-}fields \times Expression^* \rightarrow Environment \rightarrow \mathbb{B}$

```

is-tags-in-variant-fields (vfs, exprs) ρ  $\triangleq$ 
  let vfs = mk-Variant-fields(-, tagt, variant, other) in
253  t-expression (hd exprs) ρ = host-type-of (tagt) ρ ∧
218  let tagv = evaluate-constant-expression (hd exprs) ρ in
  if  $\exists i \in \text{inds } variant \cdot tagv \in variant(i).label$ 
  then let  $i \in \text{inds } variant$  be st  $tagv \in variant(i).label$  in
334    is-tags-in-variant (variant(i), tl exprs) ρ
  elseif other  $\neq$  NIL
333  then is-tags-in-field-list (other, tl exprs) ρ
??  else error("tag value does not select a component")

```

annotations Check that the type of the expression in the sequence of tag values is assignment-compatible with the type of the corresponding tag component of the record type; check that the expression that corresponds to any tag field is a constant expression.

functions

$is\text{-}tags\text{-}in\text{-}variant : Variant \times Expression^* \rightarrow Environment \rightarrow \mathbb{B}$

```

is-tags-in-variant (variant, exprs)  $\triangleq$ 
  let mk-Variant(-, fields) = variant in
333  is-tags-in-field-list (fields, exprs) ρ

```

annotations Check the tag values in a sequence of values correspond to any tag fields in a variant component.

functions

$number\text{-}of\text{-}tags\text{-}in\text{-}field\text{-}list : Fields\text{-}list \times Expression^* \rightarrow Environment \rightarrow \mathbb{N}$

```

number-of-tags-in-field-list (fsl, exprs) ρ  $\triangleq$ 
  if fsl = []  $\vee$  exprs = []
  then 0
334  else let size = number-of-tags-in-fields (hd fsl, exprs) ρ in
      let reexprs = rest (exprs, size + 1) in
334  size + number-of-tags-in-field-list (tl fsl, reexprs) ρ

```

annotations Calculate the number of tags in a fields-list.

functions

$number\text{-}of\text{-}tags\text{-}in\text{-}fields : Fields \times Expression^* \rightarrow Environment \rightarrow \mathbb{N}$

```

number-of-tags-in-fields (fs, exprs) ρ  $\triangleq$ 
  (is-Fixed-fields(fs)  $\rightarrow$  0,
334  is-Variant-fields(fs)  $\rightarrow$  number-of-tags-in-variant-fields (fs, exprs) ρ)

```

annotations Calculate the number of tags in a fields.

functions

$number\text{-}of\text{-}tags\text{-}in\text{-}variant\text{-}fields : Variant\text{-}fields \times Expression^* \rightarrow Environment \rightarrow \mathbb{N}$

```

number-of-tags-in-variant-fields (vstruc, exprs) ρ  $\triangleq$ 
  let mk-Variant-fields(-, -, variant, other) = vstruc in
  let tagval = evaluate-constant-expression (hd exprs) ρ in
  if  $\exists i \in \text{inds } variant \cdot tagval \in variant(i).label$ 
  then let  $i \in \text{inds } variant$  be st  $tagval \in variant(i).label$  in
335  number-of-tags-in-variant (variant(i), tl exprs) ρ + 1
  elseif other  $\neq$  nil
334  then number-of-tags-in-field-list (other, tl exprs) ρ + 1
??  else error("tag value does not select a component")

```

annotations Calculate the number of tags in a variant component of a record structure, selected by the value of the tag field.

functions

$number-of-tags-in-variant : Variant \times Expression^* \rightarrow Environment \rightarrow \mathbb{N}$

$number-of-tags-in-variant(vstruc, exprs)\rho \triangleq$

let $mk\text{-}Variant(-, fields) = vstruc$ in

334 $number-of-tags-in-field-list(fields, tl\ exprs)\rho$

annotations Calculates the number of tags in a variant component of a record structure.

functions

$tsize : Typed \times Value^* \rightarrow Environment \rightarrow \mathbb{N}$

$tsize(type, vals)\rho \triangleq$

An implementation-defined unsigned value

7.1.4 The SYSTEM Environment

The environment for **SYSTEM** is given below:

types

$\rho_{system} = mk\text{-}Environment(\$
 — *Constants*
 either $\{LOCSERBYTE \xrightarrow{m} locsperbyte\text{-}constant\text{-}value \text{ or } \{\},$
 $LOCSERWORD \xrightarrow{m} locsperword\text{-}constant\text{-}value,$
 $BITSPERBITSET \xrightarrow{m} bitsperbitset\text{-}constant\text{-}value\},$
 — *Types*
 either $\{BYTE \xrightarrow{m} \text{BYTE-TYPE or } \{\},$
 $WORD \xrightarrow{m} \text{WORD-TYPE},$
 $BITSET \xrightarrow{m} \text{BITSET-TYPE},$
 $LOC \xrightarrow{m} \text{LOC-TYPE},$
 $ADDRESS \xrightarrow{m} \text{ADDRESS-TYPE},$
 $BITNUM \xrightarrow{m} \text{BIT-SET-TYPES}\},$
 $MACHINEADDRESS \xrightarrow{m} \text{implementation defined structure},$
 — *Structures*
 $\{ \text{BYTE-TYPE} \xrightarrow{m} \text{byte-array-structure},$
 $\text{WORD-TYPE} \xrightarrow{m} \text{word-array-structure},$
 $\text{BITSET-TYPE} \xrightarrow{m} \text{bitset-structure},$
 $\text{BYTE-RANGE} \xrightarrow{m} \text{byte-range-structure},$
 $\text{ADDRESS-TYPE} \xrightarrow{m} \text{address-type-structure}\},$
 $\{\},$
 — *Functions*
 $ADR \xrightarrow{m} \text{SYSTEM-FUNCTION-PROCEDURE},$
 $ADDADR \xrightarrow{m} \text{SYSTEM-FUNCTION-PROCEDURE},$
 $ADDRESSVALUE \xrightarrow{m} \text{SYSTEM-FUNCTION-PROCEDURE},$
 $CAST \xrightarrow{m} \text{SYSTEM-FUNCTION-PROCEDURE},$
 $DIFADR \xrightarrow{m} \text{SYSTEM-FUNCTION-PROCEDURE},$
 $ROTATE \xrightarrow{m} \text{SYSTEM-FUNCTION-PROCEDURE},$
 $SHIFT \xrightarrow{m} \text{SYSTEM-FUNCTION-PROCEDURE},$
 $SUBADR \xrightarrow{m} \text{SYSTEM-FUNCTION-PROCEDURE},$
 $TSIZE \xrightarrow{m} \text{SYSTEM-FUNCTION-PROCEDURE},$
 — *Modules*


```

{ },
-- Level
NIL,
-- Protectiondomain
NIL,
-- Denotations
{ },
-- Continuations
{ })

```

values

```

locperbyte-constant-value = mk-Constant-value( $\mathbb{Z}$ -TYPE, mk-Whole-number-literal(locperbyte));

locperword-constant-value = mk-Constant-value( $\mathbb{Z}$ -TYPE, mk-Whole-number-literal(locperword));

bitsperbitset-constant-value = mk-Constant-value( $\mathbb{Z}$ -TYPE, mk-Whole-number-literal(bitsperbitset));

byte-array-structure = mk-Array-structure(byte-range, LOC-TYPE);

word-array-structure = mk-Array-structure(word-range, LOC-TYPE);

bitset-structure = mk-Set-structure(bitset-range);

byte-range-structure = mk-Subrange-structure(UNSIGNED-TYPE, {0, ..., locperbyte - 1});

address-type-structure = mk-Pointer-structure(Loc-type)

```

7.2 The Module COROUTINES

COROUTINES shall be a system module.

7.2.1 The pseudo definition module of COROUTINES

The interface to COROUTINES behaves as if the following were its definition module.

```
DEFINITION MODULE COROUTINES;

FROM SYSTEM IMPORT ADDRESS;

TYPE
  COROUTINE; (* Values of this type are created dynamically by NEWCOROUTINE
              and identify the coroutine in subsequent operations *)

PROCEDURE ATTACH(source: <implementation defined>);
  (* Connect the calling coroutine to the source of interrupts. *)

PROCEDURE DETACH(source: <implementation defined>);
  (* Disconnect the calling coroutine from the source of interrupts *)

PROCEDURE HANDLER(source: <implementation defined>): COROUTINE;
  (* Return the coroutine which is connected to the source of interrupts.
     The result is undefined if IsATTACHED(source) = FALSE *)

PROCEDURE IOTRANSFER
  (   to: COROUTINE;   (* identity of destination coroutine *)
  VAR from: COROUTINE; (* identity of interrupted coroutine *)
  );
  (* The calling coroutine must be connected to a source of interrupts. *)

PROCEDURE IsATTACHED(source: <implementation defined>): BOOLEAN;
  (* Return TRUE if the source of interrupts is connected to any coroutine,
     otherwise return FALSE. *)

PROCEDURE LISTEN;
  (* Momentarily set current protection to INTERRUPTIBLE. The protection
     is restored on completion of the execution of LISTEN. *)

PROCEDURE NEWCOROUTINE
  (   body: PROC;           (* code of coroutine *)
    workspace: ADDRESS;     (* pointer to workspace of coroutine *)
    size: CARDINAL;         (* workspace size in LOCs *)
    initprotection: PROTECTION; (* initial protection of coroutine *)
  VAR cr: COROUTINE         (* identity of the new coroutine *)
  );

PROCEDURE SELF(): COROUTINE;
  (* Deliver identity of calling coroutine *)

PROCEDURE TRANSFER
  (VAR self: COROUTINE; (* identity of the calling coroutine *)
   to: COROUTINE        (* identity of the destination coroutine *)
  );
```

END COROUTINES.

CHANGE —

- a) Coroutines have been moved from the module **SYSTEM**.
- b) The type **PROCESS** is now called **COROUTINE**. The change from **PROCESS** to **ADDRESS** which occurred in the third edition of the Report has not been adopted.
- c) Protection is explicitly dealt with as a parameter to **NEWCOROUTINE**.
- d) Variables of the type denoted by the type identifier **COROUTINE** may be treated as if the type is "opaque" (assignment and test for equality are defined).

7.2.2 The COROUTINES Procedures

types

Coroutine-procedure-designator = ATTACH
| DETACH
| IOTRANSFER
| LISTEN
| NEWCOROUTINE
| TRANSFER

Static Semantics

functions

wf-coroutine-procedure-designator : *Coroutine-procedure-designator* \rightarrow *Actual-parameters* \rightarrow *Environment* $\rightarrow \mathbb{B}$

wf-coroutine-procedure-designator(*sfd*)(*aps*) $\rho \triangleq$

cases *sfd* :

339 ATTACH \rightarrow *wf-attach*(*aps*) ρ ,
339 DETACH \rightarrow *wf-detach*(*aps*) ρ ,
340 IOTRANSFER \rightarrow *wf-iotransfer*(*aps*) ρ ,
341 LISTEN \rightarrow *wf-listen*(*aps*) ρ ,
342 NEWCOROUTINE \rightarrow *wf-newcoroutine*(*aps*) ρ ,
343 TRANSFER \rightarrow *wf-transfer*(*aps*) ρ

end

Dynamic Semantics

operations

m-system-procedure-designator : *System-function-designator* \rightarrow *Actual-parameters* \rightarrow *Environment* \xrightarrow{o} ()

m-system-procedure-designator(*sfd*)(*aps*) $\rho \triangleq$

cases *sfd* :

339 ATTACH \rightarrow *m-attach*(*aps*) ρ ,
340 DETACH \rightarrow *m-detach*(*aps*) ρ ,
340 IOTRANSFER \rightarrow *m-iotransfer*(*aps*) ρ ,
341 LISTEN \rightarrow *m-listen*(*aps*) ρ ,
342 NEWCOROUTINE \rightarrow *m-newcoroutine*(*aps*) ρ ,
343 TRANSFER \rightarrow *m-transfer*(*aps*) ρ

end

7.2.2.1 The ATTACH Procedure

Static Semantics

A call of **ATTACH** shall have one actual parameter. The actual parameter shall be a value which is of an interrupt source type. The parameter defines a source of interrupts in an implementation defined manner.

functions

$$\begin{aligned} &wf_attach : Actual_parameters \rightarrow Environment \rightarrow \mathbb{B} \\ &wf_attach(aps)\rho \triangleq \\ &\quad \text{len } aps = 1 \wedge \\ &\quad \text{let } [source] = aps \text{ in} \\ &\quad is_Expression(source) \wedge \\ 152 \quad &is_Interrupt_source_type(t_expression(source)\rho) \end{aligned}$$

annotations The *Interrupt-source-type* is an implementation defined type — see section 6.10.1.3

Dynamic Semantics

A call of **ATTACH** shall cause an interrupt source to be connected with the calling coroutine in an implementation defined manner. If another coroutine was connected to the source before the call of **ATTACH**, it shall no longer be connected. More than one source of interrupts may be connected to one coroutine.

operations

$$\begin{aligned} &m_attach : Arguments \rightarrow Environment \xrightarrow{o} () \\ &m_attach(arg)\rho \triangleq \\ &\quad \text{let } [source] = arg \text{ in} \\ 451 \quad &attach(source) \end{aligned}$$

7.2.2.2 The DETACH Procedure

Static Semantics

A call of **DETACH** shall have one actual parameter. The actual parameter shall be a value which is of the interrupt source type. The parameter defines a source of interrupts in an implementation defined manner.

functions

$$\begin{aligned} &wf_detach : Actual_parameters \rightarrow Environment \rightarrow \mathbb{B} \\ &wf_detach(aps)\rho \triangleq \\ &\quad \text{len } aps = 1 \wedge \\ &\quad \text{let } [source] = aps \text{ in} \\ &\quad is_Expression(source) \wedge \\ 152 \quad &is_Interrupt_source_type(t_expression(source)\rho) \end{aligned}$$

annotations The value *Interrupt-source-type* is an implementation defined type — see section 6.10.1.3

Dynamic Semantics

If the calling coroutine was connected to the source of interrupts, a call of **DETACH** shall cause the coroutine to be no longer connected to it.

operations

$$m\text{-detach} : Arguments \rightarrow Environment \xrightarrow{o} ()$$

$$m\text{-detach}(arg)\rho \triangleq$$

$$\text{let } [source] = arg \text{ in}$$

$$detach(source)$$

451

7.2.2.3 The IOTRANSFER Procedure

Static Semantics

A call of **IOTRANSFER** shall have two actual parameters. The first actual parameter shall be an expression which is of the coroutine type and the second actual parameter shall be a variable which is of the coroutine type.

functions

$$wf\text{-iotransfer} : Actual\text{-}parameters \rightarrow Environment \rightarrow \mathbb{B}$$

$$wf\text{-iotransfer}(aps)\rho \triangleq$$

$$\text{len } aps = 2 \wedge$$

$$\text{let } [to, from] = aps \text{ in}$$

$$is\text{-Expression}(to) \wedge$$

$$t\text{-expression}(to)\rho = \text{COROUTINE-TYPE} \wedge$$

$$is\text{-Variable-designator}(from) \wedge$$

$$t\text{-variable-designator}(from)\rho = \text{COROUTINE-TYPE}$$

152

145

Dynamic Semantics

A call of **IOTRANSFER** shall transfer control to the coroutine denoted by the first argument. An interrupt from one of the sources connected to the coroutine shall suspend the execution of the currently executing coroutine and shall cause the resumption of the coroutine that is connected to the source. On resumption the second argument shall contain the identity of the interrupted coroutine. It shall be an exception if the calling coroutine is not connected to a source of interrupts.

NOTE — The first argument is a value returned by **NEWCOROUTINE** or **SELF**.

operations

$$m\text{-iotransfer} : Arguments \rightarrow Environment \xrightarrow{o} ()$$

$$m\text{-iotransfer}(args)\rho \triangleq$$

$$\text{let } [to, from] = args \text{ in}$$

$$\text{def } attached = is\text{-attached}();$$

$$\text{if } attached$$

$$\text{then } (target\text{-for-interrupted-coroutine}(from);$$

$$resume(to))$$

$$\text{else } mandatory\text{-exception}(\text{NOT-ATTACHED})$$

302

302

301

306

Interrupts

An interrupt request may be generated by the system underlying a Modula-2 implementation to indicate that an external condition has arisen which requires the attention of the program. Acceptance of such a request involves the interruption of the normal sequence of execution and needs to be postponed if the program is not prepared to handle the request, if it is a condition of relatively low priority, or if acceptance would lead to interference with the section of program which is under execution. Standard procedures, which control the disablement of interrupts, may be used as access control procedures in protected modules.

NOTE — Interrupt requests may originate from hardware device controllers or may be generated by an underlying operating system depending on the run-time environment of an implementation.

The current protection shall be a component of the state of a computation. All maskable interrupts shall be disabled if the current protection is uninterruptible. All interrupts shall be enabled if the current protection is interruptible. The current protection shall be initialised to an implementation-defined value.

operations

$$interrupt! : \mathbb{N} \xrightarrow{\circ} ()$$

$$interrupt!(source) \triangleq$$

303 $transfer\text{-}to\text{-}handler(source)$

7.2.2.4 The LISTEN Procedure

Static Semantics

A call of **LISTEN** shall have no actual parameters.

functions

$$wf\text{-}listen : Actual\text{-}parameters \rightarrow Environment \rightarrow \mathbb{B}$$

$$wf\text{-}listen(aps)\rho \triangleq$$

$$\text{len } aps = 0$$

Dynamic Semantics

A call of **LISTEN** shall momentarily set the current protection to **INTERRUPTIBLE**. The protection shall be restored on completion of the execution of **LISTEN**.

operations

$$m\text{-}listen : Arguments \rightarrow Environment \xrightarrow{\circ} ()$$

$$m\text{-}listen(args)\rho \triangleq$$

is not yet defined

— Momentarily set the current protection to **INTERRUPTIBLE**

7.2.2.5 The NEWCOROUTINE Procedure

Static Semantics

A call of **NEWCOROUTINE** shall have five actual parameters.

The first actual parameter shall be an expression which is of the procedure type and shall denote the coroutine. The procedure supplied as the actual parameter shall be declared at level 0.

The second actual parameter shall be an expression which is of the address type and is a pointer to the workspace of the coroutine.

The third actual parameter shall be an expression which is of unsigned type and is the size of the workspace of the coroutine.

The fourth actual parameter shall be an expression which is of protection type and is the initial protection of the coroutine.

The fifth actual parameter shall be a variable which is of coroutine type and shall contain the identity of the new coroutine on the completion of the call.

functions

$wf_newcoroutine : Actual_parameters \rightarrow Environment \rightarrow \mathbb{B}$

$wf_newcoroutine(aps)\rho \triangleq$

$\text{len } aps = 5 \wedge$

$\text{let } [proc, space, size, prio, cr] = aps \text{ in}$

$is_Expression(proc) \wedge$

159 $is_level_zero_proc(t_expression(proc)\rho) \wedge$

$is_Expression(space) \wedge$

152 $t_expression(space)\rho = \text{ADDRESS-TYPE} \wedge$

$is_Expression(size) \wedge$

152 $t_expression(Size)\rho = \text{UNSIGNED-TYPE} \wedge$

$is_Expression(prio) \wedge$

152 $t_expression(prio)\rho = \text{PROTECTION-TYPE} \wedge$

$is_Variable_designator(cr) \wedge$

77 $t_variable(cr)\rho = \text{COROUTINE-TYPE}$

Dynamic Semantics

A call of **NEWCOROUTINE** shall create a new coroutine and initialises its workspace. The coroutine is not activated. It shall be an exception if the size of the workspace given is smaller than the minimum size of the workspace required by the coroutine.

When control is first transfered to the coroutine, it shall begin execution at the start of the procedure. When any coroutine (including the main coroutine) begins execution, the exception handling context shall be such that the effect of raising an exception shall be to cause exceptional termination of the program.

The minimum workspace size shall be implementation defined. The implementation shall check for workspace overflow.

NOTE —

- a) The use made of workspace is implementation dependent.
- b) The check for workspace overflow may be carried out when a procedure is activated (see section 6.5.4).
- c) Coroutine workspace allows the possibility of aliasing. By explicitly addressing the workspace it may be possible to alter the values of local variables or control information such as procedure return addresses. Several consequences follow:
 - 1) Because the use of workspace is implementation dependent, such a program cannot be given a meaning by this standard.
 - 2) Depending on the implementation, it may not be possible to perform a static check that a for loop control variable in a coroutine is not threatened in this way.

There is no check for overlapping workspace so that standard programs may reuse coroutine workspace providing two or more coroutines never use the same workspace concurrently.

- d) After the first call of **NEWCOROUTINE**, there will be two coroutines in existence: the one created by the call and the (pre-existing) one created by the implementation. The work space for the original, or main, coroutine is supplied by the implementation. The body of this coroutine is the initialisation code of all modules which are not declared inside a procedure (i.e. all modules at level 0).

operations

$m_newcoroutine : Arguments \rightarrow Environment \xrightarrow{o} ()$

$m_newcoroutine(args)\rho \triangleq$

$\text{let } [proc, space, size, prio, cr] = args \text{ in}$

301 $\text{def } v = is_adequate_work_space(space, size) ;$

```

    if  $v$ 
??   then (let  $cont = return-cont-exception(proc)\rho$  in
301       def  $token = allocate-coroutine-id()$  ;
301       reserve-coroutine-workspace( $token, space, size$ ) ;
112       assign( $cr, token$ )  $\rho$ ;
301       initialise( $token, cont$ ) )
306   else mandatory-exception(SMALL-WORKSPACE)

```

TO DO — Put in a choice between two functions to deal with **HALT**

- a) A function which installs a *c-tire* call so that **HALT** in a coroutine triggers the termination actions in the context of that coroutine.
- b) A function which enables the termination actions to be done in the context of the main program.

Language Clarification

The execution of a **RETURN** statement in a procedure which is the body of a coroutine will give rise to an exception (see section 6.11.3).

7.2.2.6 The TRANSFER Procedure

Static Semantics

A call of **TRANSFER** shall have two actual parameters. The first actual parameter shall be a variable which is of the coroutine type and the second actual parameter shall be an expression which is of the coroutine type and shall be the identity of the destination coroutine.

functions

```

wf-transfer : Actual-parameters  $\rightarrow$  Environment  $\rightarrow \mathbb{B}$ 
wf-transfer( $aps$ ) $\rho \triangleq$ 
  len  $aps = 2 \wedge$ 
  let [ $self, to$ ] =  $aps$  in
  is-Variable-designator( $self$ )  $\wedge$ 
145  t-variable-designator( $self$ ) $\rho = \text{COROUTINE-TYPE} \wedge$ 
  is-Expression( $to$ )  $\wedge$ 
152  t-expression( $to$ ) $\rho = \text{COROUTINE-TYPE}$ 

```

Dynamic Semantics

A call of **TRANSFER** shall assign the identity of the calling coroutine to the variable denoted by the first argument and shall transfer control to the coroutine denoted by its second argument.

NOTE — If at the end of its previous execution (if any) that coroutine transferred control to another by a call of **IOTRANSFER** then it will appear as if an interrupt has occurred.

operations

```

m-transfer : Arguments  $\rightarrow$  Environment  $\xrightarrow{o} ()$ 
m-transfer( $arg$ ) $\rho \triangleq$ 
  (let [ $var, val$ ] =  $arg$  in
302   def  $self = current-coroutine()$  ;
??   def  $cont = contents-of(val)$  ;
112   assign( $var, self$ ) ;
301   resume( $cont$ ) )

```


7.2.3 The COROUTINES Functions

types

$$\begin{aligned} \text{Coroutine-function-designator} = & \text{HANDLER} \\ & | \text{ISATTACHED} \\ & | \text{SELF} \end{aligned}$$

Static Semantics

functions

$$\text{wf-coroutine-function-designator} : \text{Coroutine-procedure-designator} \rightarrow \text{Actual-parameters} \rightarrow \text{Environment} \rightarrow \mathbb{B}$$

$$\begin{aligned} \text{wf-coroutine-function-designator}(\text{sfd})(\text{aps})\rho & \triangleq \\ \text{cases } \text{sfd} : & \\ 344 \quad \text{HANDLER} & \rightarrow \text{wf-handler}(\text{aps})\rho, \\ 345 \quad \text{ISATTACHED} & \rightarrow \text{wf-isattached}(\text{aps})\rho, \\ 346 \quad \text{SELF} & \rightarrow \text{wf-self}(\text{aps})\rho \\ \text{end} & \end{aligned}$$

functions

$$\text{t-coroutine-function-designator} : \text{Coroutine-function-designator} \rightarrow \text{Actual-parameters} \rightarrow \text{Environment} \rightarrow \text{Typed}$$

$$\begin{aligned} \text{t-coroutine-function-designator}(\text{sfd})(\text{aps})\rho & \triangleq \\ \text{cases } \text{sfd} : & \\ 345 \quad \text{HANDLER} & \rightarrow \text{t-handler}(\text{aps})\rho, \\ 345 \quad \text{ISATTACHED} & \rightarrow \text{t-isattached}(\text{aps})\rho, \\ 346 \quad \text{SELF} & \rightarrow \text{t-self}(\text{aps})\rho \\ \text{end} & \end{aligned}$$

Dynamic Semantics

operations

$$\text{m-coroutine-function-designator} : \text{System-function-designator} \rightarrow \text{Actual-parameters} \rightarrow \text{Environment} \xrightarrow{o} ()$$

$$\begin{aligned} \text{m-coroutine-function-designator}(\text{sfd})(\text{aps})\rho & \triangleq \\ \text{cases } \text{sfd} : & \\ 345 \quad \text{HANDLER} & \rightarrow \text{m-handler}(\text{aps})\rho, \\ 345 \quad \text{ISATTACHED} & \rightarrow \text{m-isattached}(\text{aps})\rho, \\ 346 \quad \text{SELF} & \rightarrow \text{m-self}(\text{aps})\rho \\ \text{end} & \end{aligned}$$

7.2.3.1 The HANDLER Function

Static Semantics

A call of **HANDLER** shall have one actual parameter. The actual parameter shall be a value which is of an interrupt source type. The parameter defines a source of interrupts in an implementation defined manner.

functions

$$\text{wf-handler} : \text{Actual-parameters} \rightarrow \text{Environment} \rightarrow \mathbb{B}$$

$$\begin{aligned} \text{wf-handler}(\text{aps})\rho & \triangleq \\ \text{len } \text{aps} = 1 \wedge & \\ \text{let } [\text{source}] = \text{aps} \text{ in} & \\ \text{is-Expression}(\text{source}) \wedge & \\ 152 \quad \text{is-Interrupt-source-type}(\text{t-expression}(\text{source})\rho) & \end{aligned}$$

annotations The value *Interrupt-source-type* is an implementation defined type — see section 6.11.1.2.

functions

$t\text{-handler} : \text{Actual-parameters} \rightarrow \text{Environment} \rightarrow \text{Typed}$

$t\text{-handler}(aps)\rho \triangleq$
COROUTINE-TYPE

Dynamic Semantics

A call of **HANDLER** shall return the identity of the coroutine which is connected to the source of interrupts denoted by the argument.

operations

$m\text{-handler} : \text{Arguments} \rightarrow \text{Environment} \xrightarrow{o} ()$

$m\text{-handler}(arg)\rho \triangleq$
let $[source] = arg$ in

?? $\text{def handlers} = \text{coroutine-handlers}();$

?? $\text{handlers}(source)$

pre $source \in \text{dom handlers}$

7.2.3.2 The IsATTACHED Function

Static Semantics

A call of **IsATTACHED** shall have one actual parameter. The actual parameter shall be a value which is of an interrupt source type. The parameter defines a source of interrupts in an implementation defined manner.

functions

$wf\text{-isattached} : \text{Actual-parameters} \rightarrow \text{Environment} \rightarrow \mathbb{B}$

$wf\text{-isattached}(aps)\rho \triangleq$
len $aps = 1 \wedge$
let $[source] = aps$ in
 $is\text{-Expression}(source) \wedge$

152 $is\text{-Interrupt-source-type}(t\text{-expression}(source)\rho)$

annotations The value *Interrupt-source-type* is an implementation defined type – see section 6.11.1.2.

functions

$t\text{-isattached} : \text{Actual-parameters} \rightarrow \text{Environment} \rightarrow \text{Typed}$

$t\text{-isattached}(aps)\rho \triangleq$
BOOLEAN-TYPE

Dynamic Semantics

A call of **IsATTACHED** shall return true if a coroutine is connected to the source of interrupts denoted by the argument.

operations

$m\text{-isattached} : \text{Arguments} \rightarrow \text{Environment} \xrightarrow{o} \text{Value}$

$m\text{-isattached}(arg)\rho \triangleq$
let $[source] = arg$ in

?? $\text{def handlers} = \text{coroutine-handlers}();$

return $source \in \text{dom } handlers$

7.2.3.3 The SELF Function

Static Semantics

A call of **SELF** shall have no actual parameters.

functions

$$\begin{aligned} wf\text{-}self &: \text{Actual-parameters} \rightarrow \text{Environment} \rightarrow \mathbb{B} \\ wf\text{-}self(aps)\rho &\triangleq \\ &\quad \text{len } aps = 0 \end{aligned}$$

functions

$$\begin{aligned} t\text{-}self &: \text{Actual-parameters} \rightarrow \text{Environment} \rightarrow \text{Typed} \\ t\text{-}self(aps)\rho &\triangleq \\ &\quad \text{COROUTINE-TYPE} \end{aligned}$$

Dynamic Semantics

A call of **SELF** shall deliver the identity of the calling coroutine.

operations

$$\begin{aligned} m\text{-}self &: \text{Actual-parameters} \rightarrow \text{Environment} \xrightarrow{o} \text{Value} \\ m\text{-}self(args)\rho &\triangleq \\ &\quad \text{current-coroutine}() \end{aligned}$$

302

7.2.4 The COROUTINE Environment

The environment for **COROUTINE** is given below:

values

```
 $\rho_{coroutine} = mk\text{-}Environment($   
   $-- Constants$   
   $\{\}$ ,  
   $-- Types$   
   $\{COROUTINE \xrightarrow{m} COROUTINE\text{-}TYPE\}$ ,  
   $-- Structures$   
   $\{\}$ ,  
   $-- Variables$   
   $\{\}$ ,  
   $-- Procedures$   
   $\{ATTACH \xrightarrow{m} COROUTINE\text{-}PROPER\text{-}PROCEDURE,$   
     $DETACH \xrightarrow{m} COROUTINE\text{-}PROPER\text{-}PROCEDURE,$   
     $IOTRANSFER \xrightarrow{m} COROUTINE\text{-}PROPER\text{-}PROCEDURE,$   
     $LISTEN \xrightarrow{m} COROUTINE\text{-}PROPER\text{-}PROCEDURE,$   
     $NEWCOROUTINE \xrightarrow{m} COROUTINE\text{-}PROPER\text{-}PROCEDURE,$   
     $TRANSFER \xrightarrow{m} COROUTINE\text{-}PROPER\text{-}PROCEDURE,$   
     $HANDLER \xrightarrow{m} COROUTINE\text{-}FUNCTION\text{-}PROCEDURE,$   
     $ISATTACHED \xrightarrow{m} COROUTINE\text{-}FUNCTION\text{-}PROCEDURE,$   
     $SELF \xrightarrow{m} COROUTINE\text{-}FUNCTION\text{-}PROCEDURE\}$ ,  
   $-- Modules$   
   $\{\}$ ,  
   $-- Level$   
   $NIL,$   
   $-- Protectiondomain$   
   $NIL,$   
   $-- Denotations$   
   $\{\}$ ,  
   $-- Continuations$   
   $\{\}, ) : Environment$ 
```

7.3 The Module EXCEPTIONS

The system module **EXCEPTIONS** provides facilities for identifying exceptions, for reporting their occurrence by raising an exception, and for setting nested contexts in which exceptions may be handled.

Unless an exception handling context has been set, the raising of an exception causes exceptional termination of the program. Termination procedures may then discover the nature of the exception.

If an exception is raised while an exception handling context is set, program execution is taken back to the point at which the context was entered.

TO DO — Add a formal definition of this module.

7.3.1 The pseudo definition module of EXCEPTIONS

The interface to **EXCEPTIONS** behaves as if the following were its definition module.

DEFINITION MODULE EXCEPTIONS;

TYPE

EXCEPTION =

```
(noException,
  (* for separate modules and user-allocated exceptions: *)
  notLanguageException,
  (* for aggregated language exceptions: *)
  indexException, rangeException, caseSelectException,
  invalidLocation, functionException,
  wholeValueException, wholeDivException,
  realValueException, realDivException,
  protException,
  (* for system module exceptions: *)
  sysException, coException, exException, termException,
  (* for general program exceptions or internal library exceptions: *)
  generalException
);
```

PROCEDURE ExceptionValue(): EXCEPTION;

```
(* For the current coroutine,
  if it is in normal execution state, returns the value noException;
  if it is in exceptional execution state because of the
  raising of a non-language exception, returns the value
  notLanguageException;
  if it is in exceptional execution state because of the raising
  of a language exception, returns the corresponding value
*)
```

PROCEDURE RAISEGENERALEXCEPTION(message: ARRAY OF CHAR);

```
(* Associates the given value of message with the current context
  and raises an exception such that the corresponding value
  returned by ExceptionValue is generalException
*)
```

PROCEDURE GetMessage(VAR text: ARRAY OF CHAR);

```
(* For the current coroutine,
  if it is in exceptional execution state, returns the
  possibly truncated string associated with the current context,
```

```

        otherwise, in normal execution state, returns the null string.
    *)

PROCEDURE PUSHHANDLER;
    (* Sets a nested exception handling context for the current coroutine
       as the current exception handling context, and returns
       in the state of normal execution.
       Execution reverts to the statement following the call if an
       exception is raised in the state of normal execution
       or if the procedure RETRY is called in the state of
       exceptional execution.
    *)

PROCEDURE RETRY;
    (* If an exception handling context is set and the calling
       coroutine is in the state of exceptional execution, execution
       reverts to the statement following the corresponding call of
       the procedure PUSHHANDLER, in the state of normal execution.
       Otherwise, an exception is raised.
    *)

PROCEDURE ACKNOWLEDGE;
    (* If an exception handling context is set and the calling
       coroutine is in the state of exceptional execution,
       it is placed in the state of normal execution.
       Otherwise, an exception is raised.
    *)

PROCEDURE POPHANDLER();
    (* Removes the nested handling context of the current coroutine,
       reverting to any context that may have been current before the
       corresponding call of the procedure PUSHHANDLER.
       If the procedure is called in the state of exceptional execution,
       the exception is then raised again.
    *)

(* Facilities for use in the implementation of library modules *)

TYPE
    ExceptionSource;
    (* values of this type are used within library modules
       to identify the source of raised exceptions
    *)
    ExceptionNumber = [2 .. MAX(CARDINAL)];

PROCEDURE AllocateSource(VAR newSource: ExceptionSource);
    (* Allocates a unique value of type ExceptionSource
       or raises an exception if a unique value cannot be allocated.
    *)

PROCEDURE RAISE(
    source: ExceptionSource;
    number: ExceptionNumber;
    message: ARRAY OF CHAR
);
    (* Any nested handling context already in the state of exceptional
       execution is removed.
    *)

```

Associates the given values of source, number and message with the current context and enters the state of exceptional execution. In the absence of an exception handling context set by the coroutine, the program is terminated exceptionally. Otherwise, execution reverts to the point at which the current context was entered.

*)

```
PROCEDURE Number(source: ExceptionSource): CARDINAL;
(* For the current coroutine,
   if it is in normal execution state, returns the value zero.
   Otherwise, in the exceptional execution state,
   if the given source is not the source of the raised exception,
   returns the value one,
   and if the given source is the source of the exception,
   returns the value associated with the exception at the time
   it was raised.
*)
```

END EXCEPTIONS.

7.3.2 The procedure **ExceptionValue**

The procedure **ExceptionValue** is an enquiry function that allows the caller to determine if no exception has been raised, if an exception has been raised by a separate library module, or if a language exception has been raised.

NOTE — The language is regarded as one source of exceptions. Similar enquiry functions are exported by separate library modules that are sources of defined exceptions.

Static Semantics

A call of **ExceptionValue** shall have no actual parameters.

Dynamic Semantics

A call of **ExceptionValue** shall return **noException** if the calling coroutine is in the state of normal execution, and shall return **notLanguageException** if the calling coroutine is in the state of exceptional execution because of the raising of an exception by a separate library module. In other cases, where the calling coroutine is in the state of exceptional execution because of the raising of a language exception, a call of **ExceptionValue** shall return the value of type **EXCEPTION** that is associated with the raised exception.

TO DO — Define the mapping between the exceptions described in the language definition and the values of the enumeration type **EXCEPTION**; review the values of the enumeration when the above mapping is considered.

7.3.3 The procedure **RAISEGENERALEXCEPTION**

The procedure **RAISEGENERALEXCEPTION** provides a simple method of explicitly raising a general exception in a program.

Static Semantics

A call of **RAISEGENERALEXCEPTION** shall have one actual parameter which is an expression and which is of a type parameter-compatible with an open array of **CHAR**.

Dynamic Semantics

If the calling coroutine is in the state of exceptional execution, and one or more exception-handling contexts are set, a call of **RAISEGENERALEXCEPTION** shall remove successive contexts until one is found in the state of normal execution or until all exception-handling contexts have been removed.

The exception message shall be set to the value of the actual parameter, interpreted as a character string and limited to an implementation-defined length. The value of type **EXCEPTION** that is associated with the raised exception shall be **generalException**.

If an exception-handling context is set, execution of the coroutine shall resume from the statement following the corresponding call of the procedure **PUSHHANDLER** in the state of exceptional execution. If no exception-handling context is set, the program shall be terminated exceptionally.

7.3.4 The procedure GetMessage

The procedure **GetMessage** allows the program to obtain the message passed when an exception is raised. This may give further information about the nature of the exception for use by the program or in the construction of other messages.

Static Semantics

A call of **GetMessage** shall have one actual parameter which is an array variable having components of the character type.

Dynamic Semantics

A call of **GetMessage** shall assign a string value to the variable given as actual parameter. If the calling coroutine is in the state of exceptional execution, the string value shall be formed from the message stored when the corresponding exception was raised, but with a length limited by the capacity of the actual parameter.

In the state of normal execution, the string assigned to the variable given as actual parameter shall be the null string.

7.3.5 The procedure PUSHHANDLER

The procedure **PUSHHANDLER** is used to set a new exception handling context for the calling coroutine which lasts until there is a corresponding call of the procedure **POPHANDLER**. There is an initial return in the state of normal execution. Subsequently, the raising of an exception causes execution of the coroutine to revert to the statement following the call of **PUSHHANDLER**, in a state of exceptional execution. A call of the procedure **RETRY** will then cause execution to revert again to the statement following the call of **PUSHHANDLER**, in the state of normal execution.

Static Semantics

A call of **PUSHHANDLER** shall have no actual parameters.

Dynamic Semantics

A call of **PUSHHANDLER** shall create a new exception handling context nested within any previously created exception handling contexts for the calling coroutine. The procedure shall return from the call in the state of normal execution.

NOTES

1 On return from a procedure or a separate module body, if there has been a call of **PUSHHANDLER** without a corresponding call of **POPHANDLER**, the extra exception handling contexts are removed and an exception is raised. This prevents the possibility of the procedure that had called **PUSHHANDLER** no longer being active on the raising of an exception.

2 If implicit or explicit calls to the standard procedure **ENTER** are made in the new context for which there have not been corresponding calls of **LEAVE** when an exception is raised, these should be made explicitly as part of the task of handling the exception.

7.3.6 The procedure **RETRY**

The procedure **RETRY** is used to restart execution of the statements of an exception handling context after an exception has been handled.

Static Semantics

A call of **RETRY** shall have no actual parameters.

Dynamic Semantics

A call of **RETRY**, made in the state of exceptional execution when an exception-handling context is set, shall cause execution of the coroutine to resume from the statement following the corresponding call of the procedure **PUSHHANDLER** in the state of normal execution.

If made in the state of normal execution, or if no handling context has been set by the calling coroutine, an exception shall be raised. The value of type **EXCEPTION** that is associated with the raised exception shall be **exException**.

7.3.7 The procedure **ACKNOWLEDGE**

The procedure **ACKNOWLEDGE** is used to switch from the state of exceptional execution to the state of normal execution to indicate that an exception has been handled without retrying the statements of the exception handling context.

Static Semantics

A call of **ACKNOWLEDGE** shall have no actual parameters.

Dynamic Semantics

A call of **ACKNOWLEDGE**, made in the state of exceptional execution when an exception-handling context is set, shall place the calling coroutine in the state of normal execution.

If made in the state of normal execution, or if no handling context has been set by the calling coroutine, an exception shall be raised. The value of type **EXCEPTION** that is associated with the raised exception shall be **exException**.

7.3.8 The procedure **POPHANDLER**

The procedure **POPHANDLER** is used to remove an exception handling context set by **PUSHHANDLER**. If called in the state of exceptional execution, the same exception is then raised again.

Static Semantics

A call of **POPHANDLER** shall have no actual parameters.

Dynamic Semantics

A call of **POPHANDLER** shall normally remove the current exception handling context for the calling coroutine so as to restore any handling context that was in force prior to the corresponding call of **PUSHHANDLER**. If the call is made in a state of normal execution, there shall be a return from the call. Otherwise, called in the state of exceptional execution, the effect shall be for the exception to be raised again.

In the event that no exception handling context has been set in the calling procedure, a call of **POPHANDLER** shall cause an exception to be raised. The value of type **EXCEPTION** that is associated with the raised exception shall be **exException**.

7.3.9 The procedure **AllocateSource**

The procedure **AllocateSource** is a server for previously unallocated values of the opaque type **ExceptionSource**. Values of this type are used to identify a source of exceptions, such as a library module, when a specific exception is reported with the procedure **Raise**. **AllocateSource** is normally called once during initialisation of a separate module and the resulting value is then used in all calls of **Raise** for exceptions that may be handled by clients of the module.

Static Semantics

A call of **AllocateSource** shall have one actual parameter which is a variable of type **ExceptionSource**.

Dynamic Semantics

A call of **AllocateSource** shall allocate a value of type **ExceptionSource** that is unique within the program and assign this value to the variable given as the actual parameter. If a unique value cannot be allocated an exception shall be raised. The value of type **EXCEPTION** that is associated with the raised exception shall be **exException**.

7.3.10 The procedure **RAISE**

The procedure **RAISE** is used by a specified source to report the detection of a specified exception. Unless an exception handling context has been set by the calling coroutine, this will cause the program to be terminated exceptionally.

The semantics of exception reporting are the same for exceptions raised by the language implementation as for exceptions raised by explicit use of the procedure **RAISE**.

Static Semantics

A call of **Raise** shall have three actual parameters. The first actual parameter shall be an expression of type **ExceptionSource**, the second actual parameter shall be an expression of a type that is assignment compatible with the type **ExceptionNumber**, and the third actual parameter shall be an expression which is of a type parameter compatible with an open array of **CHAR**.

Dynamic Semantics

On a call of **RAISE**, if the value of the second actual parameter is not in the range of values of the type **ExceptionNumber**, a language exception shall be raised. The value of type **EXCEPTION** that is associated with the raised exception shall be **exException**.

Otherwise, if the calling coroutine is in the state of exceptional execution, and one or more exception handling contexts are set, a call of **RAISE** shall remove successive contexts until one is found in the state of normal execution or until all exception handling contexts have been removed.

The current exception source shall be set to the value of the first parameter.

The current exception number shall be set to the value of the second actual parameter.

The current exception message shall be set to the value of the third actual parameter, interpreted as a character string and limited to an implementation-defined length.

If an exception handling context is set, execution of the coroutine shall resume from the statement following the corresponding call of the procedure **PUSHHANDLER** in the state of exceptional execution. If no exception handling context is set, the program shall be terminated exceptionally.

7.3.11 The procedure **Number**

The procedure **Number** is used by library modules that are sources of defined exceptions to implement appropriate enquiry functions.

Static Semantics

A call of **Number** shall have one actual parameter which shall be an expression of type **ExceptionSource**.

Dynamic Semantics

If the calling coroutine is in the state of normal execution, a call of **Number** shall return the value zero. If the calling coroutine is in the state of exceptional execution and the value given as actual parameter is not equal to the current exception source, a call of **Number** shall return the value one. Otherwise, the current exception number shall be returned.

7.3.12 Examples of use

The following program fragments illustrate the use of the facilities of the system module **EXCEPTIONS**.

```
PROCEDURE Inverse(x: REAL): REAL;
VAR
  res: REAL;
BEGIN
  PUSHHANDLER;
  CASE ExceptionValue() OF
    |noException:
      res := 1.0/x
    |realDivException:
      ACKNOWLEDGE;
      res := 0.0
  END;
  POPHANDLER;
  RETURN res
END Inverse;

FROM EXCEPTIONS IMPORT
  ExceptionValue, EXCEPTION, RAISEGENERALEXCEPTION,
  PUSHHANDLER, POPHANDLER, RETRY;
FROM LibModule IMPORT
  LibExceptionValue, LibException, Fly, ReplaceRubberBand;

PROCEDURE KeepFlying();
```

```

BEGIN
  PUSHHANDLER;
  CASE ExceptionValue() OF
  |noException:
    (* statements in normal execution *)
    (* ... *)
    PUSHHANDLER;
    CASE LibExceptionValue() OF
    |libNoException:
      Fly;
    |notLibException:
      (* an exception, but not one defined by LibModule *)
      (* pass it to previous context *)
    |brokenRubberBand:
      ReplaceRubberBand;
      RETRY;
    END;
  POPHANDLER
  (* ... *)
  |notLanguageException:
    (* some other library exception *)
    (* pass it to the next context as a general exception *)
    RAISEGENERALEXCEPTION("unknown library exception")
  |indexException :
    (* take recovery action if possible *);
    RETRY
  (* other cases, as required *)
  ELSE
    (* pass to previous context *)
  END;
  POPHANDLER
END KeepFlying;

```

A module such as `LibModule` used in the previous example would include exports such as

```

TYPE
  LibException =
    (libNoException, notLibException,
     brokenRubberBand, rubberBandNotWound);

PROCEDURE LibExceptionValue(): LibException;

```

The latter could be implemented as

```

PROCEDURE LibExceptionValue(): LibException;
BEGIN
  RETURN VAL(LibException, EXCEPTIONS.Number(savedSource))
END LibExceptionValue;

```

where `savedSource` is a variable internal to the module which is initialised by a call of `ExceptionSource`.

Internally, exceptions would be raised by calls such as

```

RAISE(savedSource, ORD(brokenRubberBand), "while winding");

```

7.4 The Module Termination

Termination is a required system module, the purpose of which is to provide a facility whereby any module needing to clean up its internal state or close external channel connections before the program terminates may register its own routine to do this . It also provides predicates for monitoring the change in phase between initialization and termination of a program.

NOTES

1 The initialization code of a program completes execution on occurrence of the first of the following:

- a) reaching the end of the program module body,
- b) execution of the standard procedure **HALT**,
- c) execution of a **RETURN** statement in the program module body,
- d) raising an exception when no handler is current.

2 **HALT** is not equivalent to returning from the program module since the domain exit procedures are not invoked; see 6.1.11 and 6.1.1

7.4.1 The pseudo definition module of Termination

The interface to **Termination** behaves as if the following were its definition module.

```
DEFINITION MODULE Termination ;
```

```
PROCEDURE IsTerminating ( ) : BOOLEAN ;
```

```
  (* This predicate yields TRUE iff the initialization code of the  
    program module has completed executing, otherwise FALSE. *)
```

```
PROCEDURE HasHalted ( ) : BOOLEAN ;
```

```
  (* This predicate yields TRUE iff a call to HALT has been made, otherwise FALSE. *)
```

```
PROCEDURE RegisterCleanup (Cleanup : PROC) ;
```

```
  (* This procedure sets Cleanup to be the first routine to run after the  
    completion of all initialization code, that is, before any code which may  
    have been previously installed by some earlier call to this routine. *)
```

```
END Termination.
```

types

Termination-designator = *Termination-procedure-designator* | *Termination-function-designator*;

Termination-procedure-designator = REGISTERCLEANUP;

Termination-function-designator = ISTERMINATING
| HASHALTED

Static Semantics

wf-termination-designator : *Termination-designator* \rightarrow *Actual-parameters* \rightarrow *Environment* $\rightarrow \mathbb{B}$

wf-termination-designator(*sfd*)(*aps*) $\rho \triangleq$
 cases *sfd* :
 357 ISTERMINATING \rightarrow *wf-isterminating*(*aps*) ρ ,
 358 HASHALTED \rightarrow *wf-hashalted*(*aps*) ρ ,
 358 REGISTERCLEANUP \rightarrow *wf-registercleanup*(*aps*) ρ
 end

Declaration Semantics

t-Termination-function-designator : *Termination-function-designator* \rightarrow *Actual-parameters* \rightarrow *Environment* \rightarrow *Typed*

t-Termination-function-designator(*sfd*)(*aps*) $\rho \triangleq$
 BOOLEAN-TYPE

7.4.2 Dynamic semantics

state *Termination* of

initialising : \mathbb{B}
halting : \mathbb{B}
finals : *Proper-procedure-value**

inv *mk-Termination*(*initialising*, *halting*, -) \triangleq *halting* $\Rightarrow \neg$ *initialising*
 end

init-Termination : \xrightarrow{o} ()

init-Termination() \triangleq
 (*initialising* := true; *halting* := false; *finals* := [])

note-HALT : \xrightarrow{o} ()

note-HALT() \triangleq
 (*halting* := true; *initialising* := false)

annotations This operation is only used by the pervasive procedure **HALT**.

termination-actions : \xrightarrow{o} ()

termination-actions() \triangleq
 is not yet defined
 See section 7.4.6.

m-termination-designator : *Termination-designator* \rightarrow *Actual-parameters* \rightarrow *Environment* \xrightarrow{o} ()

m-termination-designator(*tfd*)(*aps*) $\rho \triangleq$
 cases *tfd* :
 358 ISTERMINATING \rightarrow *m-isterminating*(*aps*) ρ ,
 358 HASHALTED \rightarrow *m-hashalted*(*aps*) ρ ,
 358 REGISTERCLEANUP \rightarrow *m-registercleanup*(*aps*) ρ
 end

7.4.3 The Isterminating Function

A call of **IsTerminating** shall have no parameters.

$wf\text{-}is\text{-}terminating : Actual\text{-}parameters \rightarrow Environment \rightarrow \mathbb{B}$

$wf\text{-}is\text{-}terminating(aps)\rho \triangleq$
 $aps = []$

A call of **IsTerminating** shall return **TRUE** if the initialization code of the program module has completed executing, and **FALSE** otherwise.

$m\text{-}is\text{-}terminating : \rightarrow Environment \xrightarrow{o} \mathbb{B}$

$m\text{-}is\text{-}terminating() \triangleq$
 $\text{return } \neg \text{initialising}$

7.4.4 The HasHalted Function

A call of **HasHalted** shall have no parameters.

$wf\text{-}hashalted : Actual\text{-}parameters \rightarrow Environment \rightarrow \mathbb{B}$

$wf\text{-}hashalted(aps)\rho \triangleq$
 $aps = []$

A call of **HasHalted** shall return **TRUE** if an explicit call to **HALT** has been made, and **FALSE** otherwise.

$m\text{-}hashalted : \rightarrow Environment \xrightarrow{o} \mathbb{B}$

$m\text{-}hashalted() \triangleq$
 $\text{return } halting$

7.4.5 The RegisterCleanup Procedure

A call of **RegisterCleanup** shall have one parameter which is a level 0 parameterless procedure.

$wf\text{-}registercleanup : Actual\text{-}parameters \rightarrow Environment \rightarrow \mathbb{B}$

$wf\text{-}registercleanup(aps)\rho \triangleq$
 $\text{len } aps = 1 \wedge$
 $\text{let } [proc] = aps \text{ in}$
 $is\text{-}Expression(proc) \wedge$
152 $\text{let } type = t\text{-}expression(proc)\rho \text{ in}$
279 $is\text{-}proper\text{-}procedure\text{-}type(type)\rho \wedge is\text{-}level\text{-}zero\text{-}proc(type)\rho$

If there is sufficient storage, and provided that a call of **IsTerminating** would return **FALSE**, a call of **RegisterCleanup** shall insert the procedure given as actual parameter at the head of a list of procedures to be executed when the initialization code of the program module has completed executing, otherwise it shall be an exception.

$m\text{-}registercleanup : Arguments \rightarrow Environment \xrightarrow{o} ()$

$m\text{-}registercleanup(arg)\rho \triangleq$
359 $\text{def } is\text{-}room = can\text{-}register();$
 $\text{if } is\text{-}room \wedge \text{initialising}$
 $\text{then def } old\text{-}list = finals;$
 $finals := arg \curvearrowright old\text{-}list$
306 $\text{else } mandatory\text{-}exception(\text{CANNOT-REGISTER})$

TO DO — Decide whether we want these semantics or the alternative (elision of ‘ $\wedge \text{initialising}$ ’)

If there is sufficient storage, a call of `RegisterCleanup` shall insert the procedure given as actual parameter at the head of a stack of procedures to be executed when the initialization code of the program module has completed executing, otherwise it shall be an exception.

can-register : $\xrightarrow{o} \mathbb{B}$

can-register () \triangleq
implementation-defined

annotations Return whether there is enough space to register a cleanup procedure

NOTES

1 Completion of the initialization code of a program, either by normal return or by one of the other situations noted above, results in calling each routine registered by `RegisterCleanup`, and if the ‘Pop-Call’ or ‘Call-Pop’ option is chosen (see 7.4.6), the unstacking of such registered clean-up code in an implementation-defined order, until the stack is empty, whereafter the Modula-2 machine is closed down.

2 A termination procedure installed by `RegisterCleanup` should be written so that it will behave correctly if called more than once, or if reentered. This might happen if, for example, a termination procedure directly or indirectly halts, or if a coroutine that is resumed as a result of an interrupt during termination also halts.

<p>7.4.6.1 The brutalist's</p> <pre> <i>termination-actions</i> : \xrightarrow{o} () <i>termination-actions</i> () \triangleq if <i>initialising</i> then (<i>initialising</i> := false; for <i>mk-Procedure-value</i> (<i>f</i>) in <i>finals</i> do <i>f</i>([])) else skip </pre>	<p>7.4.6.2 'Pop-Call'</p> <pre> <i>termination-actions</i> : \xrightarrow{o} () <i>termination-actions</i> () \triangleq if <i>finals</i> = [] then skip else (<i>initialising</i> := false; def <i>mk-Procedure-value</i> (<i>f</i>) = hd <i>finals</i>; def <i>tail</i> = tl <i>finals</i>; (<i>finals</i> := <i>tail</i>; <i>f</i>([]); <i>termination-actions</i>())) </pre>
<p>7.4.6.3 'Call-Pop'</p> <pre> <i>termination-actions</i> : \xrightarrow{o} () <i>termination-actions</i> () \triangleq if <i>finals</i> = [] then skip else (<i>initialising</i> := false; def <i>mk-Procedure-value</i> (<i>f</i>) = hd <i>finals</i>; def <i>tail</i> = tl <i>finals</i>; (<i>f</i>([]); <i>finals</i> := <i>tail</i>; <i>termination-actions</i>())) </pre>	<p>7.4.6.4 Don's suggestion</p> <pre> <i>termination-actions</i> : \xrightarrow{o} () <i>termination-actions</i> () \triangleq (<i>initialising</i> := false; for <i>mk-Procedure-value</i> (<i>f</i>) in <i>finals</i> do <i>f</i>([])) </pre>
<p>7.4.6.5 Roger's suggestion</p> <pre> <i>termination-actions</i> : \xrightarrow{o} () <i>termination-actions</i> () \triangleq (if <i>initialising</i> then (dcl <i>termproc</i> : \mathbb{N} := 0; <i>initialising</i> := false;) else <i>finals</i> := <i>del</i>(<i>finals</i>, <i>termproc</i>); for <i>i</i> = 1 to len <i>finals</i> do if \neg <i>halted</i> (<i>i</i>) then let <i>mk-Procedure-value</i> (<i>f</i>) = <i>finals</i> (<i>i</i>) in (<i>termproc</i> := <i>i</i>; <i>f</i>([])) else skip) </pre>	

NOTE — Explicit manipulation of continuations might improve some of the definitions above.

TO DO — Pick one.

8 Required Separate Modules

8.1 The Storage Module

The Storage module provides facilities for allocating and deallocating of storage at execution time.

8.1.1 The definition module of Storage

```
DEFINITION MODULE Storage;

  (* This module provides facilities for allocating and
     deallocating storage at execution time.
  *)

  FROM SYSTEM IMPORT ADDRESS;

  PROCEDURE ALLOCATE(VAR v:ADDRESS; n:CARDINAL);
  (* The procedure ALLOCATE allocates an area of storage
     which is n LOCs long and assigns the address of this
     area to v. If there is insufficient storage to do this,
     the value NIL is assigned to v.
  *)

  PROCEDURE DEALLOCATE(VAR v:ADDRESS; n:CARDINAL);
  (* The procedure DEALLOCATE frees an area of storage
     which is n LOCs long whose address is given by the
     value of v. The value NIL is then assigned to v.
  *)

END Storage .
```

8.1.2 Dynamic semantics

When a program starts, it shall have no areas of storage allocated to it.

```
state Storage of
  heap : Loc-set-set
  inv (mk-Storage(heap))  $\triangle$ 
     $\forall x, y \in \text{heap} \cdot x \cap y = \{ \}$ ;
  init (mk-Storage(heap))  $\triangle$ 
    heap = { }
end
```

The procedure ALLOCATE

A call of ALLOCATE shall allocate to the program an area of consecutive storage locations that are not currently allocated to the program. The area shall have a size in LOCs that is equal to the value of the expression n . If it is not possible to allocate this storage, the nil value shall be assigned to the variable v ; otherwise, the address of the area shall be assigned to the variable v .

$ALLOCATE : Variable \times \mathbb{N} \xrightarrow{o} Environment \Rightarrow$

$ALLOCATE(v, n)\rho \triangleq$

if $n = 0 \vee \neg is-available(n)$

then assign(v , NIL-VALUE) ρ

else let $locs \in Loc\text{-}set$ be st

card $locs = n \wedge$

$is-consecutive-locs(locs) \wedge$

$is-well-aligned(locs) \wedge$

$\forall loc \in locs \cdot \neg exists(loc) \text{ in let } loc \in locs \text{ be st } loc = base-of-locs(locs) \text{ in allocate-locs}(locs);$

add-to-heap($locs$);

assign(v , mk-Pointer-value(loc)) ρ

$is-available(n : \mathbb{N}) b : \mathbb{B}$

ext rd $heap : Loc\text{-}set\text{-}set$

rd $store : Loc \xrightarrow{m} Storable\text{-}value$

pre true

post $b \Leftrightarrow implementation\text{-}dependent\text{-}expression$

$is-consecutive-locs : Loc\text{-}set \rightarrow \mathbb{B}$

$is-consecutive-locs(locs) \triangleq$

let $n = \text{card } locs$ in $\exists loc \in locs \cdot$

$\{add\text{-}offset(loc, i) \mid i \in \{0, \dots, n-1\}\} = \{address\text{-}of(loc) \mid loc \in locs\}$

$is-well-aligned : Loc\text{-}set \rightarrow \mathbb{B}$

$is-well-aligned(locs) \triangleq$

$implementation\text{-}defined\text{-}expression$

$base-of-locs : Loc\text{-}set \rightarrow Loc$

$base-of-locs(locs) \triangleq$

$implementation\text{-}defined\text{-}expression$

$allocate-locs(locs : Loc\text{-}set)$

ext wr $store : Loc \xrightarrow{m} Storable\text{-}value$

pre $\forall loc \in locs \cdot loc \notin \text{dom } store$

post $store = \overline{store} \cup \{loc \mapsto \text{UNDEFINED} \mid loc \in locs\}$

$add\text{-}to\text{-}heap(locs : Loc\text{-}set)$

ext wr $heap : Loc\text{-}set\text{-}set$

pre $\forall loc \in locs \cdot loc \notin \bigcup heap$

post $heap = \overline{heap} \cup \{locs\}$

The procedure DEALLOCATE

A call of DEALLOCATE shall cause an area of consecutive storage locations to be unallocated from the program. The area shall have a size in LOCs that is equal to the value of the expression n , and shall have an address that is equal to the current value of the variable v . The call of DEALLOCATE shall assign the nil value to the variable v . The area to be unallocated shall be an area of storage that is currently allocated to the program; otherwise, an exception shall occur.

$DEALLOCATE : Variable \times \mathbb{N} \xrightarrow{o} Environment \Rightarrow$

$DEALLOCATE(v, n)\rho \triangleq$

if $n = 0$

then assign(v , NIL-VALUE) ρ

else let $loc = \text{value-of-a-variable}(v)$ in if $loc = \text{NIL-VALUE}$

then exception(value-of-the-pointer-variable-is-the-NIL-value)

else if $loc \notin \bigcup heap$

then exception(variable-points-to-an-unallocated-storage-location)

else let $locs \in heap$ be st $loc \in locs$ in let $unwanted_addrs = \{add_offset(loc, i) \mid i \in \{0, \dots, n - 1\}\}$ in if $unwanted_addrs \neq \{address_of(loc) \mid loc \in locs\}$

then exception(area-to-be-unallocated-is-not-the-same-as-a-currently-allocated-area)

else deallocate-locs($locs$) ;

remove-from-heap($locs$) ;

assign(v , NIL-VALUE) ρ

$remove_from_heap(locs : Loc\text{-}set)$

ext wr $heap : Loc\text{-}set\text{-}set$

pre $locs \in heap$

post $heap = \overline{heap} - \{locs\}$

8.2 Low level real modules

Two modules are provided to enable access to the underlying properties of type **REAL** and type **LONGREAL**. The two modules share common concepts, functions and values and hence both modules are considered together.

The module **LowReal** gives access to the underlying properties of **REAL**, while **LowLong** gives access to the same properties for **LONGREAL**.

If an implementation provides facilities for dynamically changing the properties of type **REAL** or **LONGREAL**, then the constant values defined in these modules shall refer to the default properties.

8.2.1 The definition module of LowReal

The definition module shall be

```
DEFINITION MODULE LowReal;
```

```
CONST
```

```
  Radix =      <implementation-defined integer value>;
  Places =     <implementation-defined integer value>;
  ExpoMin =    <implementation-defined integer value>;
  ExpoMax =    <implementation-defined integer value>;
  Large =      <implementation-defined REAL value>;
  Small =      <implementation-defined REAL value>;
  IEEE =       <implementation-defined BOOLEAN value>;
  ISO =        <implementation-defined BOOLEAN value>;
  Rounds =     <implementation-defined BOOLEAN value>;
  GUnderflow = <implementation-defined BOOLEAN value>;
  Exception =  <implementation-defined BOOLEAN value>;
  Extend =     <implementation-defined BOOLEAN value>;
  NModes =     <implementation-defined integer value>;
```

```
TYPE Modes = SET OF 0 .. NModes;
```

```
  LowException = (LowNoException, NotLowException, LowRealException, LowLongException);
```

```
PROCEDURE LowExceptionValue(): LowException;
```

```
PROCEDURE exponent(x: REAL): INTEGER;
```

```
PROCEDURE fraction(x: REAL): REAL;
```

```
PROCEDURE sign(x: REAL): REAL;
```

```
PROCEDURE succ(x: REAL): REAL;
```

```
PROCEDURE ulp(x: REAL): REAL;
```

```
PROCEDURE pred(x: REAL): REAL;
```

```
PROCEDURE intpart(x: REAL): REAL;
```

```
PROCEDURE fractpart(x: REAL): REAL;
```

```
PROCEDURE scale(x: REAL; n: INTEGER): REAL;
```

```
PROCEDURE trunc(x: REAL; n: INTEGER): REAL;
```

```

PROCEDURE round(x: REAL; n: INTEGER): REAL;

PROCEDURE Synthesise(expo: INTEGER; sig: REAL): REAL;

PROCEDURE SetMode(M: Modes);

PROCEDURE GetMode(): Modes;

END LowReal.

```

The implementation module shall be provided by the processor with the semantics given in clause 8.2.3.

8.2.2 The definition module of LowLong

The definition module shall be

```

DEFINITION MODULE LowLong;

CONST
    Radix =      <implementation-defined integer value>;
    Places =     <implementation-defined integer value>;
    ExpoMin =    <implementation-defined integer value>;
    ExpoMax =    <implementation-defined integer value>;
    Large =      <implementation-defined LONGREAL value>;
    Small =      <implementation-defined LONGREAL value>;
    IEEE =       <implementation-defined BOOLEAN value>;
    ISO =        <implementation-defined BOOLEAN value>;
    Rounds =     <implementation-defined BOOLEAN value>;
    GUnderflow = <implementation-defined BOOLEAN value>;
    Exception =  <implementation-defined BOOLEAN value>;
    Extend =     <implementation-defined BOOLEAN value>;
    NModes =     <implementation-defined integer value>;

TYPE Modes = SET OF 0 .. NModes;
    LowException = (LowNoException, NotLowException, LowRealException, LowLongException);

PROCEDURE LowExceptionValue(): LowException;

PROCEDURE exponent(x: LONGREAL): INTEGER;

PROCEDURE fraction(x: LONGREAL): LONGREAL;

PROCEDURE sign(x: LONGREAL): LONGREAL;

PROCEDURE succ(x: LONGREAL): LONGREAL;

PROCEDURE ulp(x: LONGREAL): LONGREAL;

PROCEDURE pred(x: LONGREAL): LONGREAL;

PROCEDURE intpart(x: LONGREAL): LONGREAL;

PROCEDURE fractpart(x: LONGREAL): LONGREAL;

PROCEDURE scale(x: LONGREAL; n: INTEGER): LONGREAL;

```

```

PROCEDURE trunc(x: LONGREAL; n: INTEGER): LONGREAL;

PROCEDURE round(x: LONGREAL; n: INTEGER): LONGREAL;

PROCEDURE Synthesise(expo: INTEGER; sig: LONGREAL): LONGREAL;

PROCEDURE SetMode(M: Modes);

PROCEDURE GetMode(): Modes;

END LowLong.

```

The implementation module shall be provided by the processor with the semantics given in clause 8.2.3.

8.2.3 Semantics

The bodies of the two modules **LowReal** and **LowLong** shall have the semantics defined below. For the exceptions defined below, an implementation shall either raise the exception **LowRealException** (for the module **LowReal**, or the exception **LowLongException** for the module **LowLong**), or an implementation-defined predefined exception shall occur.

NOTES

- 1 The predefined exceptions are listed in clause 6.12.1, and the exception could depend upon the context.
- 2 If an implementation chooses to use a predefined exception, then this exception could fail to be detected (see the list in clause 6.12.1).
- 3 Since type **REAL** or type **LONGREAL** could be fixed-point or floating-point, or something quite different, the semantics of these module cannot be defined to the same precision of the main language. Hence VDM-SL is not used in this clause.
- 4 A more precise specification of functions which conform to the natural language specification given here appears in *ISO/IEC 10967: Information technology—Programming languages—Language compatible arithmetic*. Functions of a similar name correspond and the following correspondences hold:

Modula-2	LCAS
Radix	r
Places	p
GUnderflow	$denorm$
Small	$fmin_N$
ExpoMin	$emin$
ExpoMax	$emax$
MAX(REAL)	$fmax$

For the identifiers **X** listed below, the module **LowReal** shall provide **LowReal.X** and the module **LowLong** shall provide **LowLong.X**:

Radix --- : The integer giving the implementation-defined value of the radix used to represent the corresponding real values.

Places --- : The implementation-defined integer giving the number of radix places to store values of the corresponding real type.

NOTE — Some implementations may choose to compute expressions to greater precision than that used to store values. The value **Places**, and the other facilities in these modules refer only to the representation used to store values.

ExpoMin --- : The implementation-defined integer value of the exponent minimum.

ExpoMax --- : The implementation-defined integer value of the exponent maximum.

NOTE — An implementation is permitted to choose values such that **ExpoMin** = **ExpoMax**, which will presumably be the case for a fixed point representation.

Large --- : The implementation-defined largest value of the corresponding real type.

NOTE — On some systems this may be a machine representation of infinity.

Small --- : The implementation-defined smallest positive value of the corresponding real type, represented to maximal precision.

NOTE — If an implementation has stored values strictly between 0.0 and **Small**, then presumably the implementation supports gradual underflow.

IEEE --- : An implementation-defined **BOOLEAN** value which is **TRUE** if and only if the corresponding real type conforms to the IEEE 754 or IEEE 854 floating point standards.

NOTE — If **IEEE** then the value of **Radix** is 2 for IEEE 754, and 2 or 10 for IEEE 854.

ISO --- : An implementation-defined **BOOLEAN** value which is **TRUE** if and only if the corresponding real type conforms to the forthcoming ISO standard (ISO/IEC 10967).

NOTE — It is expected that the proposed ‘*Language compatible arithmetic*’ standard will be agreed at about the same time as this International Standard.

Rounds --- : An implementation-defined **BOOLEAN** value which is **TRUE** if and only if each operation produces a result which is one of the nearest values of the corresponding real type to the mathematical result.

NOTE — If **Rounds**, the result is not defined if the mathematical result lies mid-way between two values of the real type.

GUnderflow --- : An implementation-defined **BOOLEAN** value which is **TRUE** if and only if there are values of the corresponding real type between 0.0 and **Small**.

Exception --- : An implementation-defined **BOOLEAN** value which is **TRUE** if and only if all operations which attempt to produce real values out of range results in a detected exception.

Extend --- : An implementation-defined **BOOLEAN** value which is **TRUE** if and only if expressions of the corresponding real type are computed to higher precision than the stored values.

NOTE — If **Extend**, then values greater than **Large** can be computed in an expression but not stored in variables.

NModes --- : The implementation-defined integer value giving one less than the number of bit positions needed for the status flags for mode control.

exponent --- : Gives the exponent value of **x** and hence lies between **ExpoMin** and **ExpoMax**. Gives an exception if **x** = 0.0.

fraction --- : Gives the significand or mantissa part of **x**; hence the relationship: **x** = **fraction(x)** * **Radix** ** **exponent(x)** should hold.

sign --- : Gives 1.0 if **x** is greater than 0.0, gives -1.0 if **x** is less than 0.0, and gives either 1.0 or -1.0 if **x** is 0.0.

NOTE — The uncertainty about the handling of 0.0 is to allow for systems which distinguish between 0.0 and -0.0 (such as IEEE 754 systems).

succ --- : Gives the next value of the corresponding real type greater than **x** (if such a value exists), otherwise an exception is raised.

ulp --- : Gives the value of the corresponding real type equal to the last digit of **x** (if such a value exists), otherwise an exception is raised.

pred --- : Gives the next value of the corresponding real type less than **x** (if such a value exists), otherwise an exception is raised.

intpart --- : Gives the integer part of **x**. For negative values, this is **-intpart(abs(x))**.

fractpart --- : Gives the fractional part of **x**. This satisfies **fractpart(x)+intpart(x)=x**.

scale --- : Gives **x * Radix ** n** if such a value exists otherwise an exception is raised.

trunc --- : Gives the value the first **n** places of **x**. An exception is raised if **n** is less than or equal to zero.

round --- : Gives the value of **x** rounded to the first **n** places. An exception is raised if such a value does not exist or if **n** is less than or equal to zero.

Synthesis --- : This produces a result of the corresponding real type from the given significand and exponent values.

SetMode --- : The procedure resets status flags appropriate to the underlying implementation of the corresponding real type.

NOTES

1 The effect of **SetMode** on the operations on the corresponding real type on any other ready process is not defined. Many implementations of floating point provide options for setting status flags within the system which controls details of the handling of the type. Although two procedures are provided, one for each real type, the effect may be the same. Typical effects that can be obtained by this means are:

- a) Ensure overflow causes an exception;
- b) Allow underflow to cause an exception;
- c) Control the rounding;
- d) Allow special values to be produced (NaNs in IEEE);
- e) Ensure special value access will cause an exception;

Since these effects are so varied, the values of type **Modes** used is not specified.

2 Implementations are not required to preserve the status flags (if any) with the co-routine state.

GetMode --- : Gives the mode in the form set by **SetMode**, or the default if **SetMode** is not used.

NOTE — The value obtained by **GetMode** is not necessarily the value set by **SetMode**, since a call of **SetMode** might attempt to set flags which cannot be set by program.

8.3 Classification of values of the character type

The full set of values of the character type (the elementary type denoted by the pervasive identifier **CHAR**) is implementation-defined. The module **CharClass** allows a program to determine the classification of a given value of the character type of an implementation in a way that does not rely on there being a known literal representation of all members of a class.

8.3.1 The Module CharClass

The module **CharClass** provides predicates to test if a given value of the character type in an implementation is classified as a numeric, a letter, an upper-case letter, a lower-case letter, or a value representing a control function.

8.3.1.1 The definition module of CharClass

```
DEFINITION MODULE CharClass;

(* Classification of values of the pervasive type CHAR *)

PROCEDURE IsNumeric(ch: CHAR): BOOLEAN;
  (* Tests if the parameter is classified as a numeric character *)

PROCEDURE IsLetter(ch: CHAR): BOOLEAN;
  (* Tests if the parameter is classified as a letter *)

PROCEDURE IsUpper(ch: CHAR): BOOLEAN;
  (* Tests if the parameter is classified as an upper case letter *)

PROCEDURE IsLower(ch: CHAR): BOOLEAN;
  (* Tests if the parameter is classified as a lower case letter *)

PROCEDURE IsControl(ch: CHAR): BOOLEAN;
  (* Tests if the parameter represents a control function *)

END CharClass.
```

8.3.1.2 The dynamic semantics of CharClass

TECHNICAL NOTE — The following definitions refer to sequences of character values defined in 6.9.5 or to sets of values built up from lexical definitions in 5.7. The latter are distinguished by a different font, eg. *national numeric*.

The procedure IsNumeric

A call of **IsNumeric** shall return the value **TRUE** if the value of the parameter **ch** is a member of an implementation-defined set of numeric characters which shall include the decimal digits, and the value **FALSE** otherwise.

$IsNumeric : \text{char} \rightarrow \mathbb{B}$

$IsNumeric(ch) \triangleq$
 $ch \in \text{elems } \textit{digits} \cup \{c \mid c \in \textit{national numeric}\}$

The procedure IsLetter

A call of **IsLetter** shall return the value **TRUE** if the value of the parameter **ch** is a member of an implementation-defined set of letters which shall include the required letters, and the value **FALSE** otherwise.

$IsLetter : \text{char} \rightarrow \mathbb{B}$

$IsLetter(ch) \triangleq$

$ch \in \text{elems } required\text{-}lower\text{-}case\text{-}letters \cup \text{elems } required\text{-}upper\text{-}case\text{-}letters \cup \{c \mid c \in \text{letter}\}$

The procedure **IsUpper**

A call of **IsUpper** shall return the value **TRUE** if the value of the parameter **ch** is a member of an implementation-defined set of upper case letters which shall include the required upper case letters, and the value **FALSE** otherwise.

$IsUpper : \text{char} \rightarrow \mathbb{B}$

$IsUpper(ch) \triangleq$

$ch \in \text{rng } capitalisations$

The procedure **IsLower**

A call of **IsLower** shall return the value **TRUE** if the value of the parameter **ch** is a member of an implementation-defined set of lower case letters which shall include the required lower case letters, and the value **FALSE** otherwise.

$IsLower : \text{char} \rightarrow \mathbb{B}$

$IsLower(ch) \triangleq$

$ch \in \text{dom } capitalisations$

The procedure **IsControl**

A call of **IsControl** shall return the value **TRUE** if the value of the parameter **ch** is a member of an implementation-defined set of characters representing control functions, and the value **FALSE** otherwise.

$IsControl : \text{char} \rightarrow \mathbb{B}$

$IsControl(ch) \triangleq$

is not yet defined

TO DO — Complete the definition of *IsControl*.

9 Standard Libraries

9.1 The Mathematical Libraries

The mathematical libraries provide the common mathematical functions and two real constants.

The module `RealMath` provides the constants and functions for type `REAL`, while the module `LongMath` provides similar constants and functions for type `LONGREAL`.

The module `ComplexMath` provides functions for type `COMPLEX`, while the module `LongComplexMath` provides similar functions for type `LONGCOMPLEX`.

9.1.1 The definition modules

9.1.1.1 The definition module of `RealMath`

```
DEFINITION MODULE RealMath;

CONST
  PI    = 3.1415926535897932384626433832795028841972;
  Exp1  = 2.7182818284590452353602874713526624977572;

TYPE
  RMathLibException = (RMathNoException, NotRMathException, RMathException);

PROCEDURE sqrt(x: REAL): REAL;

PROCEDURE exp(x: REAL): REAL;

PROCEDURE ln(x: REAL): REAL;

PROCEDURE sin(x: REAL): REAL;

PROCEDURE cos(x: REAL): REAL;

PROCEDURE tan(x: REAL): REAL;

PROCEDURE arcsin(x: REAL): REAL;

PROCEDURE arccos(x: REAL): REAL;

PROCEDURE arctan(x: REAL): REAL;

PROCEDURE power(x, y: REAL): REAL;

PROCEDURE round(x: REAL): INTEGER;

PROCEDURE MathException(): RMathLibException;

END RealMath.
```

The implementation module shall be provided with the semantics given in 9.1.2.

9.1.1.2 The definition module of LongMath

```
DEFINITION MODULE LongMath;

CONST
  PI    = 3.1415926535897932384626433832795028841972;
  Expl  = 2.7182818284590452353602874713526624977572;

TYPE
  RMathLibException = (RMathNoException, NotRMathException, RMathException);

PROCEDURE sqrt(x: LONGREAL): LONGREAL;

PROCEDURE exp(x: LONGREAL): LONGREAL;

PROCEDURE ln(x: LONGREAL): LONGREAL;

PROCEDURE sin(x: LONGREAL): LONGREAL;

PROCEDURE cos(x: LONGREAL): LONGREAL;

PROCEDURE tan(x: LONGREAL): LONGREAL;

PROCEDURE arcsin(x: LONGREAL): LONGREAL;

PROCEDURE arccos(x: LONGREAL): LONGREAL;

PROCEDURE arctan(x: LONGREAL): LONGREAL;

PROCEDURE power(x, y: LONGREAL): LONGREAL;

PROCEDURE round(x: LONGREAL): INTEGER;

PROCEDURE MathException(): RMathLibException;

END LongMath.
```

The implementation module shall be provided with the semantics given in 9.1.2.

9.1.1.3 The definition module of ComplexMath

```
DEFINITION MODULE ComplexMath;

TYPE
  CMathLibException = (CMathNoException, NotCMathException, CMathException);

PROCEDURE abs (cmplx: COMPLEX) : REAL;
(* returns the length of a complex number*)

PROCEDURE arg (cmplx : COMPLEX) : REAL;
(* returns the angle a complex number subtends to the positive real axis *)

PROCEDURE exp (cmplx : COMPLEX) : COMPLEX;
(* returns the complex exponential of the complex number *)
```

```

PROCEDURE ln (cmplx : COMPLEX) : COMPLEX;
(* returns the principal value of the natural log of the complex number *)

PROCEDURE polarToComplex (abs, arg : REAL) : COMPLEX;
(* returns the complex number with the given polar coordinates *)

PROCEDURE sqrt (cmplx : COMPLEX) : COMPLEX;
(* returns the principal square root of the complex number*)

PROCEDURE MathException(): CMathLibException;

END ComplexMath.

```

The implementation module shall be provided with the semantics given in 9.1.3.

9.1.1.4 The definition module of LongComplexMath

```

DEFINITION MODULE LongComplexMath;

TYPE
  CMathLibException = (CMathNoException, NotCMathException, CMathException);

PROCEDURE abs (cmplx: LONGCOMPLEX) : LONGREAL;
(* returns the length of a long complex number*)

PROCEDURE arg (cmplx : LONGCOMPLEX) : LONGREAL;
(* returns the angle a long complex number subtends to the positive real
axis *)

PROCEDURE exp (cmplx : LONGCOMPLEX) : LONGCOMPLEX;
(* returns the complex exponential of the long complex number *)

PROCEDURE ln (cmplx : LONGCOMPLEX) : LONGCOMPLEX;
(* returns the principal value of the natural log of the long complex
number *)

PROCEDURE polarToComplex (abs, arg : LONGREAL) : LONGCOMPLEX;
(* returns the long complex number with the given polar coordinates *)

PROCEDURE sqrt (cmplx : LONGCOMPLEX) : LONGCOMPLEX;
(* returns the principal square root of the long complex number*)

PROCEDURE MathException(): CMathLibException;

END LongComplexMath.

```

The implementation module shall be provided with the semantics given in 9.1.3.

9.1.2 Semantics of RealMath and LongMath

The semantics of the two modules shall be the same, except that when module **RealMath** refers to type **REAL**, the corresponding function in **LongMath** refers to type **LONGREAL**, and when a function in **RealMath** raises the exception **RealMathException**, the corresponding exception is raised in **LongMath**.

NOTE — The above statement is merely to avoid needless repetition of the semantics for **LongMath**.

Dynamic Semantics

NOTE — The functions `sin`, `cos`, `tan` and the absolute operator used in the definitions below are the mathematical functions which are of infinite precision. Hence the finite precision of actual implementations is modelled by defining a result of infinite precision followed by an approximation operation (performed by *working-approximation*).

9.1.2.1 The constants

The constant `PI` shall provide an implementation-defined approximation to the mathematical constant π . The constant `Exp1` shall provide an implementation-defined approximation to the mathematical constant e .

NOTE — Due to the approximations involved, `sin(PI)` may not equal zero exactly; similarly with `Exp1` and `exp(1)`.

9.1.2.2 Square root

The result of the square root function shall be an implementation-defined approximation to the positive signed square root of the argument. It shall be an exception if the argument is negative.

operations

```
 $sqrt : \mathbb{R} \xrightarrow{o} \mathbb{R}$   
 $sqrt(x) \triangleq$   
  if  $x \geq 0$   
  then let  $r : \mathbb{R}$  be st  $r \geq 0 \wedge$   
     $r^2 = x$  in  
??      return working-approximation( $r$ )  
306    else mandatory-exception(NEGATIVE-SQRT-ARG)
```

9.1.2.3 Exponential

The result of the exponential function shall be an implementation-defined approximation to the mathematical constant e raised to the power of the argument.

```
 $exp : \mathbb{R} \xrightarrow{o} \mathbb{R}$   
 $exp(x) \triangleq$   
  let  $r = e^x$  in  
??  return working-approximation( $r$ )
```

9.1.2.4 Natural logarithm

The result of the natural logarithm function shall be an implementation-defined approximation to the natural logarithm of the argument. It shall be an exception if the the argument is zero or negative.

```
 $ln : \mathbb{R} \xrightarrow{o} \mathbb{R}$   
 $ln(x) \triangleq$   
  if  $x > 0$   
  then let  $r : \mathbb{R}$  be st  $e^r = x$  in  
??      return working-approximation( $r$ )  
306    else mandatory-exception(NONPOSITIVE-LN-ARG)
```

9.1.2.5 Sine

The result of the sine function shall be an implementation-defined approximation to the sine of the argument.

```

sin : ℝ  $\xrightarrow{o}$  ℝ
sin (x)  $\triangleq$ 
  let r = sin(x) in
?? return working-approximation (r)

```

9.1.2.6 Cosine

The result of the cosine function shall be an implementation-defined approximation to the cosine of the argument.

```

cos : ℝ  $\xrightarrow{o}$  ℝ
cos (x)  $\triangleq$ 
  let r = cos(x) in
?? return working-approximation (r)

```

9.1.2.7 Tangent

The result of the tangent function shall be an implementation-defined approximation to the tangent of the argument.

```

tan : ℝ  $\xrightarrow{o}$  ℝ
tan (x)  $\triangleq$ 
  let r = tan(x) in
?? return working-approximation (r)

```

9.1.2.8 Arcsine

The result of the arcsine function shall be an implementation-defined approximation to the arcsine of the argument. It shall be an exception if the absolute value of the argument is greater than one.

```

arcsin : ℝ  $\xrightarrow{o}$  ℝ
arcsin (x)  $\triangleq$ 
  if 1 ≥ x ≥ -1
  then let r : ℝ be st sin(r) = x ∧
    π/2 < r ≤ π/2 in
?? return working-approximation (r)
306 else mandatory-exception(ARCSIN-ARG-MAGNITUDE)

```

9.1.2.9 Arccosine

The result of the arccosine function shall be an implementation-defined approximation to the arccosine of the argument. It shall be an exception if the absolute value of the argument is greater than 1.

```

arccos : ℝ  $\xrightarrow{o}$  ℝ
arccos (x)  $\triangleq$ 
  if 1 ≥ x ≥ -1
  then let r : ℝ be st cos(r) = x ∧
    0 ≤ r ≤ π in
?? return working-approximation (r)
306 else mandatory-exception(ARCCOS-ARG-MAGNITUDE)

```

9.1.2.10 Arctangent

The result of the arctangent function shall be an implementation-defined approximation to the arctangent of the argument.


```

arctan :  $\mathbb{R} \xrightarrow{o} \mathbb{R}$ 
arctan (x)  $\triangleq$ 
  let r :  $\mathbb{R}$  be st  $\tan(r) = x \wedge$ 
     $-\pi < r \leq +\pi$  in
?? return working-approximation (r)

```

9.1.2.11 Power

The result of the power function shall be an implementation-defined approximation to the result obtained by raising the first argument to the power of the second argument. It shall be an exception if the value of the first argument is zero or negative.

```

power :  $\mathbb{R} \times \mathbb{R} \xrightarrow{o} \mathbb{R}$ 
power (x, y)  $\triangleq$ 
  if x > 0
  then let r =  $x^y$  in
??   return working-approximation (r)
306 else mandatory-exception(NONPOSITIVE-POWER-ARG)

```

annotations This function is mathematically equivalent to $\exp(y * \ln(x))$ but may be computed differently.

9.1.2.12 Round

The result of the round function shall be the integer obtained by rounding the argument. The result shall be implementation-defined if the argument is midway between two integer values. It shall be an exception if the mathematical result is not within the range of type **INTEGER**.

```

round :  $\mathbb{R} \xrightarrow{o} \mathbb{Z}$ 
round (x)  $\triangleq$ 
  let n :  $\mathbb{Z}$  be st  $|n - x| \leq 0.5$  in
?? return get-whole-result (INTEGER-TYPE, n)

```

annotations The result of *round*(1.5) is either 1 or 2. An implementation may choose either value. The exception referred to above is defined by *get-whole-result*.

9.1.3 Semantics of ComplexMath and LongComplexMath

The semantics of the two modules shall be the same, except that when module **ComplexMath** refers to type **COMPLEX**, the corresponding function in **LongComplexMath** refers to type **LONGCOMPLEX**, and when a function in **ComplexMath** raises the exception **CMathException**, the corresponding exception is raised in **LongComplexMath**.

NOTE — The above statement is merely to avoid needless repetition of the semantics for **LongComplexMath**.

Dynamic Semantics

NOTE — The functions sin, cos, arctan and the absolute operator used in the definitions below are the mathematical functions which are of infinite precision. Hence the finite precision of actual implementations is modelled by defining a result of infinite precision followed by an approximation operation (performed by *working-approximation* or *working-complex-approximation*).

TO DO — Modify the definitions, where appropriate, to produce *principal* values.

9.1.3.1 Complex modulus

A call to **abs** shall produce an implementation defined approximation to the Modulus, (or length, or absolute value) of the complex number

NOTE — An overflow exception may occur in this computation, even when the complex number is itself well defined.

```

abs :  $\mathbb{C} \xrightarrow{o} \mathbb{R}$ 
abs(x)  $\triangleq$ 
?? let r :  $\mathbb{R}$  be st  $r^2 = (\text{re } x)^2 + (\text{im } x)^2$  in return working-approximation( $|r|$ )

```

9.1.3.2 Complex angle

A call to **arg** shall produce an implementation defined approximation to the angle the complex number makes with the positive real axis in the complex plane.

```

arg :  $\mathbb{C} \xrightarrow{o} \mathbb{R}$ 
arg(x)  $\triangleq$ 
  let re = re x,
      im = im x in
?? ( re > 0  $\rightarrow$  return working-approximation(arctan(im/re)),
??   re < 0  $\rightarrow$  return working-approximation(if im  $\geq$  0 then  $\pi - \text{arctan}(\text{im}/\text{re})$  else  $-\pi - \text{arctan}(\text{im}/\text{re})$ ),
      re = 0  $\rightarrow$  if im = 0
306         then mandatory-exception(zero-arg-parameters)
??         else return working-approximation(if im > 0 then  $\pi/2$  else  $3\pi/2$ ) )

```

9.1.3.3 Complex exponential

A call to **exp** shall produce an implementation defined approximation to the mathematical constant e raised to the power of the argument.

```

exp :  $\mathbb{C} \xrightarrow{o} \mathbb{R}$ 
exp(x)  $\triangleq$ 
  let re = re x,
      im = im x in
  let c :  $\mathbb{C}$  be st  $\text{re } c = e^{\text{re}} \times \cos(\text{im}) \wedge$ 
      im c =  $e^{\text{re}} \times \sin(\text{im})$  in
?? return working-complex-approximation(c)

```

9.1.3.4 Complex natural logarithm

A call to **ln** shall produce an implementation defined approximation to the principal value or the natural logarithm of the argument.

```

exp :  $\mathbb{C} \xrightarrow{o} \mathbb{R}$ 
exp(x)  $\triangleq$ 
  let c :  $\mathbb{C}$  be st  $e^c = x$  in
?? return working-complex-approximation(c)

```

9.1.3.5 Polar representation

A call to **polarToComplex** with parameters **abs** and **arg** shall produce an implementation defined approximation to the complex number having Modulus **abs** and argument **arg**.

$$polarToComplex : \mathbb{R} \times \mathbb{R} \xrightarrow{o} \mathbb{C}$$

$$polarToComplex(abs, arg) \triangleq$$

$$\text{let } c : \mathbb{C} \text{ be st } \text{rec} = abs \times \cos(arg) \wedge$$

$$\text{im } c = abs \times \sin(arg) \text{ in}$$

```
?? return working-complex-approximation(c)
```

9.1.3.6 Complex square root

A call of ‘**sqrt**(**x**)’ shall produce an implementation defined approximation to the principal square root of the complex number **x**. That is, the result is the complex number whose argument has minimum absolute value and which multiplied by itself is approximately equal to **x**.

operations

$$sqrt : \mathbb{C} \xrightarrow{o} \mathbb{C}$$

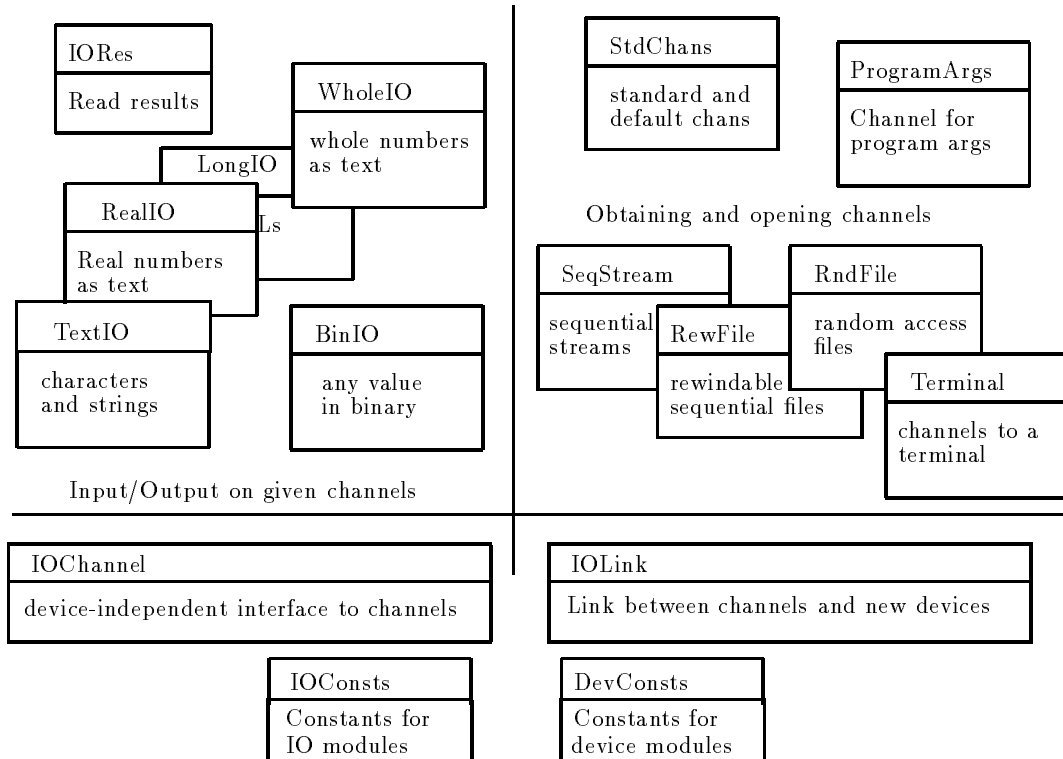
$$sqrt(x) \triangleq$$

$$\text{let } c : \mathbb{C} \text{ be st } c^2 = x \text{ in}$$

```
?? return working-complex-approximation(c)
```

9.2 The Input/Output Library

The input/output library defined in this clause allows for the reading and writing of data streams over one or more *channels*. Channels are connected to sources of input data and destinations of output data known as *devices* or *device instances*. There is a separation between modules that are concerned with device-independent operations, such as reading and writing, and modules concerned with device-dependent operations, such as making connections to named files. This separation allows the library to be extended to work with new devices. The module structure of the library is depicted in the following figure.



Channels already open to standard sources and destinations may be identified using functions provided by the module **StdChans**. This module also allows identification and selection of a set of channels used by default for input and output operations.

The modules **TextIO**, **WholeIO**, **RealIO**, and **LongIO**, allow the reading and writing of high-level units of data using text operations on channels specified explicitly by a parameter. These high-level units include characters and strings, whole numbers and real numbers in decimal notation. The module **BinIO** allows the reading and writing of arbitrary data types using binary operations on explicitly specified channels. The related module **IORes** allows the program to determine whether the last operation to read data from a specified channel found data in the required format.

Corresponding to the **TextIO** group of modules is a group of modules **STextIO**, **SWholeIO**, **SRealIO**, **SLongIO**, **SBinIO** and **SIORes**. The procedures of this group do not take parameters identifying a channel. They operate on the default input and output channels as identified by the module **StdChans**.

The module **IOConsts** defines constants and types used by **IORes** and **SIORes**.

The device modules **SeqStream**, **RewFile**, **RndFile**, and **Terminal** allow new channels to be opened to named streams, to rewindable sequential files, to random access files, and to a terminal device. The device module **ProgramArgs** provides an open channel from which program arguments may be read. Device specific operations, such as positioning within a random access file, are defined by the corresponding device module.

The module **DevConsts** defines constants and types used in the device module procedures that open channels.

The primitive device-independent operations on channels are provided by the module **IOChannel**. Text operations produce data streams as sequences of characters and line marks. Binary operations produce data streams as sequences of storage locations. The library allows devices to support both text and binary operations on a single channel but this behaviour is not required for the devices defined in this clause.

The module **IOChannel** defines general input/output library exceptions which may be raised when using any device. Device errors, such as a hardware read/write error, are reported by raising one of the general exceptions and providing an implementation-defined error number. Exceptions associated with device-specific operations are defined by the appropriate device module.

The final module of the input/output library is the module **IOLink**. This module allows the user of the library to implement new device modules for use with channels.

NOTE — Partial implementations of the input/output library may provide only modules selected from the group **STextIO**, **SWholeIO**, **SRealIO**, and **SLongIO**, normally with **SIORes** and **IOConsts**. If any other module is provided, the module **IOChannel** must also be provided, in accord with the import dependencies between definition modules of the library.

TO DO — Simplify the comments in definition modules once the dynamic semantics sections have been completed.

9.2.0.1 Common Definitions

types

Char is not yet defined;

Read-result = ALL-RIGHT | OUT-OF-RANGE | WRONG-FORMAT |
END-OF-INPUT | END-OF-LINE | UNKNOWN;

Open-result = OPENED | NO-SUCH-FILE | FILE-EXISTS |
WRONG-FLAGS | OUT-OF-CHANNELS;

Chan-id = token;

Dev-id = token;

Std-dev-name = RNDFILE | REWFILE | SEQSTREAM | PROGRAMARGS | TERMINAL;

New-dev-name = token;

Dev-name = *Std-dev-name* | *New-dev-name*;

Chan is not yet defined;

File = $\mathbb{N} \xrightarrow{m} Loc$;

Filepos = implementation-defined;

Characteristic = READ | WRITE | OLD | TEXT | BINARY | ECHO | INTERACTIVE;

Operation = LOOK | SKIP | SKIPLOOK | WRITELN | WRITETEXT | WRITEBIN |
READTEXT | READBIN | NAME | FLUSH | RESET | FREE;

Inop = *Dev-id* $\xrightarrow{o} ([Loc \mid Char] \times \mathbb{N})$;

Outop = *Dev-id* $\times (Loc \mid Char) \xrightarrow{o} \mathbb{N}$;

Insop = *Dev-id* $\xrightarrow{o} ((Loc^* \mid Char^*) \times \mathbb{N})$;

Outsop = *Dev-id* $\times (Loc^* \mid Char^*) \xrightarrow{o} \mathbb{N}$;

Nameop = *Dev-id* $\times \mathbb{N} \xrightarrow{o} Char^*$;

Op = *Dev-id* $\xrightarrow{o} ()$;

$Device\text{-}op = Inop \mid Outop \mid Insop \mid Outsop \mid Nameop \mid Op;$

compose *Device* of

id : *Dev-name*
properties : *Characteristic-set*
ops : $\{Operation \mapsto Device\text{-}op\}$
error : \mathbb{Z}
result : *Read-result*

end

state *IOLib* of

channels : $Chan\text{-}id \xleftrightarrow{m} Chan$
devices : $Chan\text{-}id \xleftrightarrow{m} Dev\text{-}id$
devtable : $Dev\text{-}id \xleftrightarrow{m} Device$
files : $Dev\text{-}id \xrightarrow{m} File$
write-position : $Dev\text{-}id \xrightarrow{m} \mathbb{N}$
stdin : *Chan-id*
stdout : *Chan-id*
stderr : *Chan-id*
nullchan : *Chan-id*
badchan : *Chan-id*

inv *mk-IOLib* (*channels*, *devices*, *devtable*, *files*, *write-position*, -, -, -, -) \triangleq
 $(ch \in \text{dom } channels \Leftrightarrow ch \in \text{dom } devices) \wedge$
 $(d \in \text{rng } devices \Leftrightarrow d \in \text{dom } devtable) \wedge$
 $(d \in \text{dom } files \Leftrightarrow d \in \text{dom } write\text{-}position)$

end

9.2.1 Standard and Default Channels

Standard channels do not have to be opened by the program since they are already open and ready for use. The values used to identify these channels are constant throughout the execution of the program. Under an operating system, they will normally be connected to sources and destinations specified before the program is run. On a stand-alone system, they may be connected to a console terminal. No method is provided of closing a standard channel.

Default channels are channels whose identities have been stored as those to be used by default for input and output operations. Initially these correspond to the standard channels but the values may be varied to obtain the effect of redirection.

9.2.1.1 The module StdChans

The module **StdChans** defines function procedures that identify channels already open to implementation-defined sources and destinations of standard input, standard output, and standard error output. Access to a *null* channel is provided to allow unwanted output to be suppressed. This module also allows identification and selection of the channels used by default for input and output operations.

9.2.1.1.1 The definition module of StdChans

DEFINITION MODULE StdChans;

(* Standard and default channels *)

IMPORT IOChannel;

TYPE

```

ChanId = IOChannel.ChanId;
(* Values of this type are used to identify channels *)

(* The pre-opened channel values. These channels cannot be closed. *)

PROCEDURE StdInChan(): ChanId;
(* Returns a value identifying the implementation-defined standard source
for program input *)

PROCEDURE StdOutChan(): ChanId;
(* Returns a value identifying the implementation-defined standard source
for program output *)

PROCEDURE StdErrChan(): ChanId;
(* Returns a value identifying the implementation-defined standard
destination for program error messages. *)

(* The null device throws away all data written to it and gives an
immediate end of input indication on reading: *)

PROCEDURE NullChan(): ChanId;
(* Returns a value identifying a channel open to the null device *)

(* The default channel values. *)

PROCEDURE InChan(): ChanId;
(* Returns the identity of the current default input channel,
as used by input procedures that do not take a channel parameter.
Initially this is the value returned by the procedure StdInChan. *)

PROCEDURE OutChan(): ChanId;
(* Returns the identity of the current default output channel,
as used by output procedures that do not take a channel parameter.
Initially this is the value returned by the procedure StdOutChan. *)

PROCEDURE ErrChan(): ChanId;
(* Returns the identity of the current default error message channel.
Initially this is the value returned by the procedure StdErrChan. *)

PROCEDURE SetInChan(cid: ChanId);
(* Sets the current default input channel identity to that given by the
value of the parameter cid. *)

PROCEDURE SetOutChan(cid: ChanId);
(* Sets the current default output channel identity to that given by the
value of the parameter cid. *)

PROCEDURE SetErrChan(cid: ChanId);
(* Sets the current default error channel identity to that given by the
value of the parameter cid. *)

END StdChans.

```

9.2.1.1.2 The dynamic semantics of StdChans

The procedure StdInChan

A call of **StdInChan** shall return a value identifying a channel open to the implementation-defined standard source for program input.

The procedure **StdOutChan**

A call of **StdOutChan** shall return a value identifying a channel open to the implementation-defined standard destination for program output.

The procedure **StdErrChan**

A call of **StdErrChan** shall return a value identifying a channel open to the implementation-defined standard destination for program error messages.

The procedure **NullChan**

A call of **NullChan** shall return a value identifying a channel open to the null device. The null device shall support all operations by discarding all data written to it and by giving an immediate end of input indication on reading.

The procedure **InChan**

A call of **InChan** shall return the identity of the current default input channel. This shall be the channel currently used by input procedures that do not take a channel parameter. Initially this shall be the value returned by the procedure **StdInChan**.

The procedure **OutChan**

A call of **OutChan** shall return the identity of the current default output channel. This shall be the channel currently used by output procedures that do not take a channel parameter. Initially this shall be the value returned by the procedure **StdOutChan**.

The procedure **ErrChan**

A call of **ErrChan** shall return the identity of the current default output channel for program error messages. Initially this shall be the value returned by the procedure **StdErrChan**.

The procedure **SetInChan**

A call of **SetInChan** shall set the current default input channel to that identified by the value of the parameter **cid**.

The procedure **SetOutChan**

A call of **SetOutChan** shall set the current default output channel to that identified by the value of the parameter **cid**.

The procedure **SetErrChan**

A call of **SetErrChan** shall set the current default output channel for error messages to that identified by the value of the parameter **cid**.

9.2.2 Reading and Writing of Data over Specified Channels

The module **TextIO** allows the reading and writing of characters, character strings, and line marks using text operations. The reading and writing of whole numbers in decimal text form is allowed by the module **WholeIO**. The modules **RealIO** and **LongIO** provide for the input and output of real numbers in decimal text form.

The module **BinIO** allows for the reading and writing of data using binary operations.

The input procedures of these modules are sufficient for use where the format of the input data is known. However, in case their use is inconsistent with the format of the input data, they have the effect of setting a read result for the used channel. The module **IORes** allows the read result for the most recent input operation to be obtained.

In all cases, channels are selected explicitly by the value of a parameter passed to the procedures of these modules.

TO DO — Consider provision for writing and reading data in octal and hexadecimal notation representing numbers or bit patterns.

9.2.2.1 The module TextIO

The module **TextIO** allows the reading and writing of characters, character strings, and line marks using text operations.

9.2.2.1.1 The definition module of TextIO

```
DEFINITION MODULE TextIO;
```

```
(* Character and string text operations *)
```

```
FROM IOChannel IMPORT  
  ChanId;
```

```
(* The following procedures do not read past line marks: *)
```

```
PROCEDURE ReadChar(cid: ChanId; VAR ch: CHAR);  
  (* If possible, removes a character from the input stream and assigns  
  the corresponding value to the parameter ch.  
  The read result is set to allRight, endOfLine or endOfInput. *)
```

```
PROCEDURE ReadLine(cid: ChanId; VAR s: ARRAY OF CHAR);  
  (* Removes any remaining characters before the next line mark copying  
  as many as can be accommodated to the parameter s as a string value.  
  The read result is set to the value allRight, outOfRange, endOfLine,  
  or endOfInput. *)
```

```
PROCEDURE ReadString(cid: ChanId; VAR s: ARRAY OF CHAR);  
  (* Removes and copies only those characters before the next line mark  
  that can be accommodated to the parameter s as a string value.  
  The read result is set to the value allRight, endOfLine, or endOfInput. *)
```

```
PROCEDURE ReadToken(cid: ChanId; VAR s: ARRAY OF CHAR);  
  (* Skips leading spaces and then removes characters before the next  
  space or line mark copying as many as can be accommodated to the  
  parameter s as a string value.  
  The read result is set to the value allRight, outOfRange, endOfLine,  
  or endOfInput. *)
```

```

(* The following procedure reads past the next line mark *)

PROCEDURE ReadLn(cid: ChanId);
  (* Removes successive items from the input stream up to and including
  the next line mark or until the end of input is reached.
  The read result is set to the value allRight, or endOfInput. *)

(* Output procedures *)

PROCEDURE WriteChar(cid: ChanId; ch: CHAR);
  (* Writes the parameter ch to the output stream *)

PROCEDURE WriteLn(cid: ChanId);
  (* Writes a line mark to the output stream *)

PROCEDURE WriteString(cid: ChanId; s: ARRAY OF CHAR);
  (* Writes the string value of the parameter s to the output stream *)

END TextIO.

```

9.2.2.1.2 The dynamic semantics of TextIO

The procedure ReadChar

If there is a character next in the input stream, a call of **ReadChar** shall remove the character from the stream and assign the corresponding value to the parameter **ch**. Otherwise, the value of the parameter **ch** is not defined. The read result for the channel shall be set to the value

allRight if a character is read;
endOfLine if no character is read, the next item being a line mark;
endOfInput if no character is read, the input stream having ended.

$ReadChar : Chan-id \xrightarrow{o} [Char]$

$ReadChar(chid) \triangleq$
442 $def\ did = device(chid);$
405 $read-at-most(did, 1)$

The procedure ReadLine

A call of **ReadLine** shall read a string formed from any remaining characters before the next line mark. As much of the string as can be accommodated shall be copied to the parameter **s** as a string value. The read result for the channel shall be set to the value

allRight if the string is not empty and is accommodated in **s**;
outOfRange if the string is not empty but is not accommodated in **s**;
endOfLine if the string is empty, the next item being a line mark;
endOfInput if the string is empty, the input stream having ended.

$ReadLine : Chan-id \times \mathbb{N}_1 \xrightarrow{o} Char^*$

$ReadLine(chid, size) \triangleq$
442 $def\ did = device(chid);$
405 $def\ line = read-while(did, \lambda x \cdot true);$

```

    if  $line \neq []$ 
    then if  $\text{len } line \leq size$ 
435       then (  $set\text{-}result(did, ALL\text{-}RIGHT)$  ;
               return  $line$  )
435       else (  $set\text{-}result(did, OUT\text{-}OF\text{-}RANGE)$  ;
std        return  $front(line, size)$  )
    else return []

```

The procedure ReadString

A call of **ReadString** shall read a string formed from any remaining characters before the next line mark or up to the capacity of the parameter **s**. The string shall be copied to the parameter **s** as a string value. The read result for the channel shall be set to the value

allRight if the string is not empty and is accommodated in **s**;
endOfLine if the string is empty, the next item being a line mark;
endOfInput if the string is empty, the input stream having ended.

$ReadString : Chan\text{-}id \times \mathbb{N}_1 \xrightarrow{o} Char^*$

$ReadString(chid, size) \triangleq$

```

442  def  $did = device(chid)$  ;
405  read-at-most( $did, size$ )

```

The procedure ReadToken

A call of **ReadToken** shall skip any leading spaces and read a string formed from any remaining characters before the next space or line mark. As much of the string as can be accommodated shall be copied to the parameter **s** as a string value. The read result for the channel shall be set to the value

allRight if the string is not empty and is accommodated in **s**;
outOfRange if the string is not empty but is not accommodated in **s**;
endOfLine if the string is empty, the next item being a line mark;
endOfInput if the string is empty, the input stream having ended.

$ReadToken : Chan\text{-}id \times \mathbb{N}_1 \xrightarrow{o} Char^*$

$ReadToken(chid, size) \triangleq$

```

442  def  $did = device(chid)$  ;
a86  def  $sps\text{-}tok = read\text{-}while(did, \lambda x \cdot isa\text{-}token(x))$  ;
444  let  $sps \curvearrowright tok = sps\text{-}tok$  be st  $tok = no\text{-}leading\text{-}space(sps\text{-}tok)$  in
    if  $tok = []$ 
    then if  $\text{len } tok \leq size$ 
435       then (  $set\text{-}result(did, ALL\text{-}RIGHT)$  ;
               return  $tok$  )
435       else (  $set\text{-}result(did, OUT\text{-}OF\text{-}RANGE)$  ;
std        return  $front(tok, size)$  )
    else return []

```

$isa\text{-}token : Char^* \rightarrow \mathbb{B}$

$isa\text{-}token(s) \triangleq$

```

444  let  $sps \curvearrowright tok = s$  be st  $tok = no\text{-}leading\text{-}space(s)$  in
444   $\forall t \in \text{elems } tok \cdot \neg isa\text{-}space(t)$ 

```

The procedure ReadLn

A call of **ReadLn** shall read successive items from the input stream up to and including the next line mark or until the end of input is reached. The read result for the channel shall be set to the value

allRight if a line mark is read;
endOfInput if no line mark is read, the input stream having ended.

ReadLn : *Chan-id* \xrightarrow{o} ()

ReadLn (*chid*) \triangleq

```
442  def did = device(chid);
405  def line = read-while(did,  $\lambda x$  . true);
436  if result(did) = END-OF-LINE
406  then remove(did)
      else skip
```

The procedure **WriteChar**

A call of **WriteChar** shall write the character corresponding to the parameter **ch** to the output stream.

WriteChar : *Chan-id* \times *Char* \xrightarrow{o} ()

WriteChar (*chid*, *ch*) \triangleq

```
442  def did = device(chid);
405  write(did, ch)
```

The procedure **WriteLn**

A call of **WriteLn** shall write a line mark to the output stream.

WriteLn : *Chan-id* \xrightarrow{o} ()

WriteLn (*chid*) \triangleq

```
442  def did = device(chid);
405  write-eol(did)
```

The procedure **WriteString**

A call of **WriteString** shall write the string value of the parameter **s** to the output stream.

WriteString : *Chan-id* \times *Char*^{*} \xrightarrow{o} ()

WriteString (*chid*, *chs*) \triangleq

```
442  def did = device(chid);
      for ch : Char in chs
405  do write(did, ch)
```

9.2.2.2 The module **WholeIO**

The module **WholeIO** allows the reading and writing of whole numbers in decimal text form.

9.2.2.2.1 The definition module of **WholeIO**

DEFINITION MODULE **WholeIO**;

(* Input and output of whole numbers in text form *)

```

FROM IOChannel IMPORT
  ChanId;

(* the text form of a signed whole number is
   ["+" | "-"], decimal digit, {decimal digit} *)

PROCEDURE ReadInt(cid: ChanId; VAR int: INTEGER);
  (* Skips leading spaces and removes any remaining characters
   that form part of a signed whole number.
   A corresponding value is assigned to the parameter int.
   The read result is set to the value allRight, outOfRange, wrongFormat,
   endOfLine, or endOfInput. *)

PROCEDURE WriteInt(cid: ChanId; int: INTEGER; width: CARDINAL);
  (* Writes the value of the parameter int in text form in a field of the
   given minimum width. *)

(* the text form of an unsigned whole number is
   decimal digit, {decimal digit} *)

PROCEDURE ReadCard(cid: ChanId; VAR card: CARDINAL);
  (* Skips leading spaces and removes any remaining characters
   that form part of an unsigned whole number.
   A corresponding value is assigned to the parameter card.
   The read result is set to the value allRight, outOfRange, wrongFormat,
   endOfLine, or endOfInput. *)

PROCEDURE WriteCard(cid: ChanId; card: CARDINAL; width: CARDINAL);
  (* Writes the value of the parameter card in text form in a field of the
   given minimum width. *)

END WholeIO.

```

9.2.2.2.2 The dynamic semantics of WholeIO

The text form of a signed whole number shall be

["+" | "-"], decimal digit, {decimal digit}

isa-signed-number : $Char^* \rightarrow \mathbb{B}$

isa-signed-number(*s*) \triangleq

388 *isa-number*(*s*) \vee
444 let *sps* \curvearrowright *sign-digits* = *s* be st *sign-digits* = *no-leading-space*(*s*) in
77 let *sign* \curvearrowright *digits* = *sign-digits* be st *isa-sign*(*sign*) in
444 $\forall d \in \text{elems } \textit{digits} \cdot \textit{isa-numeral}(d)$

annotations Return *true* if the sequence is a signed number *or could form part of a signed number*. Return *false* iff the sequence could not be a signed number.

The text form of an unsigned whole number shall be

decimal digit, {decimal digit}

isa-number : $Char^* \rightarrow \mathbb{B}$

isa-number(*s*) \triangleq

444 let *sps* \curvearrowright *digits* = *s* be st *digits* = *no-leading-space*(*s*) in
444 $\forall d \in \text{elems } \textit{digits} \cdot \textit{isa-numeral}(d)$

annotations	Return <i>true</i> if the sequence is an unsigned number <i>or could form part of an unsigned number</i> . Return <i>false</i> iff the sequence could not be an unsigned number.
-------------	---

The procedure **ReadInt**

A call of **ReadInt** shall skip any leading spaces and read characters that form part of a signed whole number. The read result for the channel shall be set to the value

allRight	if a signed whole number is read and the value is in the range of the pervasive type INTEGER —the value shall be assigned to the parameter int ;
outOfRange	if a signed whole number is read but the value is out of range of the type INTEGER —the value MAX(INTEGER) or MIN(INTEGER) shall be assigned to int according to the sign of the number;
wrongFormat	if there are characters read or to be read but these are not in the format of a signed whole number —the value of int is not defined;
endOfLine	if no characters are read, the next item being a line mark —the value of int is not defined;
endOfInput	if no characters are read, the input having ended —the value of int is not defined.

$ReadInt : Chan-id \xrightarrow{o} [\mathbb{Z}]$

$ReadInt(chid) \triangleq$

```

442 def did = device(chid);
405 def sps-sign-digits = read-while(chid, λ x · isa-signed-number(x));
444 let sps ⇧ sign-digits = sps-sign-digits be st sign-digits = no-leading-space(sps-sign-digits) in
?? let sign ⇧ digits = sign-digits be st sign = [] ∨ isa-sign(sign) in
if digits = []
436 then def res = result(did);
      (if res = ALL-RIGHT ∨ sign ≠ []
435       then set-result(did, WRONG-FORMAT)
       else skip;
       return nil )
?? else let value = if isa-minus(sign)
444       then - numeric-value(digits)
444       else numeric-value(digits) in
let res = if value > max-signed-value ∨ value < min-signed-value
          then OUT-OF-RANGE
          else ALL-RIGHT in
435 (set-result(did, res);
    return if res = ALL-RIGHT
          then value
          else if value > 0 then max-signed-value else min-signed-value )

```

The procedure **WriteInt**

A call of **Writeint** shall write the value of the parameter **int** in text form with leading spaces as required to make the number of characters written at least that given by the value of the parameter **width**. A sign shall be written only for negative values. In the special case of a value of zero for **width**, exactly one leading space shall be written.

$WriteInt : Chan \times \mathbb{Z} \times \mathbb{N} \xrightarrow{o} ()$

$WriteInt(chid, value, width) \triangleq$

```

442 def did = device(chid);

```

```

let number, sign, digits : Char* be st number = sign ~ digits ^
                                     sign = if value < 0 then minus-char else [] ^
                                     numeric-value (digits) = | value | in
444
let s = if width = 0
444     then spaces (1) ~ sign ~ digits
         elseif len sign ~ digits ≥ width
         then sign ~ digits
444     else spaces (width - len sign ~ digits) ~ sign ~ digits in
405 write-chars (did, s)

```

The procedure ReadCard

A call of **ReadCard** shall skip any leading spaces and read characters that form part of an unsigned whole number. The read result for the channel shall be set to the value

allRight	if an unsigned whole number is read and the value is in the range of the pervasive type CARDINAL —the value shall be assigned to the parameter card ;
outOfRange	if a signed whole number is read but the value is out of range of the values of type CARDINAL —the value MAX(CARDINAL) shall be assigned to card ;
wrongFormat	if there are characters read or to be read but these are not in the format of an unsigned whole number —the value of card is not defined;
endOfLine	if no characters are read, the next item being a line mark —the value of card is not defined;
endOfInput	if no characters are read, the input having ended —the value of card is not defined.

ReadCard : Chan-id \xrightarrow{o} [N]

ReadCard (chid) \triangleq

```

442 def did = device(chid) ;
405 def sps-digits = read-while(did, λ x · isa-number(x)) ;
444 let sps ~ digits = sps-digits be st digits = no-leading-space (sps-digits) in
    if digits = []
436 then def res = result(did) ;
        (if res = ALL-RIGHT
435         then set-result(did, WRONG-FORMAT)
         else skip;
         return nil )
444 else let value = numeric-value (digits) in
        let res = if value > max-unsigned-value
                    then OUT-OF-RANGE
                    else ALL-RIGHT in
435 (set-result(did, res) ;
    return if res = ALL-RIGHT
            then value
            else max-unsigned-value )

```

The procedure WriteCard

A call of **WriteCard** shall write the value of the parameter **card** in text form with leading spaces as required to make the number of characters written at least that given by the value of the parameter **width**. In the special case of a value of zero for **width**, exactly one leading space shall be written.

```

WriteCard : Chan × ℕ × ℕ  $\xrightarrow{o}$  ()
WriteCard (chid, value, width)  $\triangleq$ 
442   def did = device(chid);
444   let digits : Char* be st numeric-value (digits) = value in
     let s = if width = 0
444         then spaces (1)  $\frown$  digits
         elseif len digits  $\geq$  width
         then digits
444         else spaces (width - len digits)  $\frown$  digits in
405   write-chars(did, s)

```

9.2.2.3 The module RealIO

The module `RealIO` allows the reading and writing of real numbers in decimal text form. Real number parameters are of the pervasive type `REAL`.

9.2.2.3.1 The definition module of RealIO

```

DEFINITION MODULE RealIO;

(* Input and output of real numbers in decimal text form *)

FROM IOChannel
  IMPORT ChanId;

(* the text form of a signed fixed-point real number is
   ["+" | "-"], decimal digit, {decimal digit},
   [".", {decimal digit}]
*)

(* the text form of a signed floating-point real number is
   signed fixed-point real number,
   "E", ["+" | "-"], decimal digit, {decimal digit}
*)

PROCEDURE ReadReal(cid: ChanId; VAR real: REAL);
  (* Skips leading spaces and removes any remaining characters
   that form part of a signed fixed or floating point number.
   A corresponding value is assigned to the parameter real.
   The read result is set to the value allRight, outOfRange, wrongFormat,
   endOfLine, or endOfInput. *)

PROCEDURE WriteFloat(
  cid: ChanId; real: REAL; sigFigs: CARDINAL; width: CARDINAL
);
  (* Writes the value of the parameter real in floating-point text form
   with sigFigs significant figures in a field of the given minimum width. *)

  (* Examples of floating point output:
   value:      3923009   39.23009   0.0003923009
   sigFigs
   1           4E+6      4E+1       4E-4
   2           3.9E+6    3.9E+1     3.9E-4
   5           3.9230E+6 3.9230E+1 3.9230E-4
   *)

```



```

PROCEDURE WriteEng(
  cid: ChanId; real: REAL; sigFigs: CARDINAL; width: CARDINAL
);
(* As for WriteFloat except that the number is scaled with one to
three digits in the whole number part and with an exponent that is
a multiple of three. *)

(* Examples of floating point engineering output:
value:      3923009   39.23009   0.0003923009
sigFigs
1           4E+6           40           400E-6
2           3.9E+6          39           390E-6
5           3.9230E+6       39.230       392.30E-6
*)

PROCEDURE WriteFixed(
  cid: ChanId; real: REAL; place: INTEGER; width: CARDINAL
);
(* Writes the value of the parameter real in fixed-point text form,
rounded to the given place relative to the decimal point,
in a field of the given minimum width. *)

(* Examples of fixed point output:
value:      3923009   3.923009   0.0003923009
places
-5           3920000           0           0
-2           3923010           0           0
-1           3923009           4           0
0            3923009.           4.           0.
1            3923009.0          3.9           0.0
4            3923009.0000        3.9230        0.0004
*)

PROCEDURE WriteReal(cid: ChanId; real: REAL; width: CARDINAL);
(* Writes the value of real as WriteFixed if the sign and magnitude
can be shown in the given width, or otherwise as WriteFloat.
The number of places or significant digits depend on the given width. *)

END RealIO.

```

9.2.2.3.2 The dynamic semantics of RealIO

The text form of a signed fixed-point real number shall be

```

["+" | "-"], decimal digit, {decimal digit},
["."], {decimal digit}

```

The text form of a signed floating-point real number shall be

```

signed fixed-point real number,
"E", ["+" | "-"], decimal digit, {decimal digit}

```

The procedure ReadReal

A call of **ReadReal** shall skip any leading spaces and read characters that form part of a signed fixed or floating point number. The read result for the channel shall be set to the value

allRight	if a signed whole number is read and the value is in the range of the pervasive type REAL —the value shall be assigned to the parameter real ;
outOfRange	if a signed real number is read but the value is out of range of the values of type REAL —the value MAX(REAL) or MIN(REAL) shall be assigned to real according to the sign of the number;
wrongFormat	if there are characters read or to be read but these are not in the format of a signed real number —the value of real is not defined;
endOfLine	if no characters are read, the next item being a line mark —the value of real is not defined;
endOfInput	if no characters are read, the input having ended —the value of real is not defined.

The procedure **WriteFloat**

A call of **WriteFloat** shall write the value of the parameter **real** in floating-point text form with leading spaces as required to make the number of characters written at least that given by the value of the parameter **width**. A sign shall be written only for negative values. In the special case of a value of zero for **width**, exactly one leading space shall be written.

One significant digit shall be included in the whole number part. The signed exponent part shall be included only if the exponent value is not 0. If the value of the parameter **sigFigs** is greater than 0, that number of significant digits shall be included, otherwise an implementation-defined number of significant digits shall be included. The decimal point shall not be included if there are no significant digits in the fractional part.

The procedure **WriteEng**

A call of **WriteEng** shall write the value of the parameter **real** in floating-point text form as for **WriteFloat**, except that the number shall be scaled with one to three digits in the whole number part and with an exponent that is a multiple of three.

The procedure **WriteFixed**

A call of **WriteFixed** shall write the value of the parameter **real** in fixed-point text form with leading spaces as required to make the number of characters written at least that given by the value of the parameter **width**. A sign shall be written only for negative values. In the special case of a value of zero for **width**, exactly one leading space shall be written.

At least one digit shall be included in the whole number part. The value shall be rounded to the given value of **place** relative to the decimal point which shall be suppressed if **place** is less than 0.

The procedure **WriteReal**

If the sign and magnitude of the parameter **real** can be expressed in a field given by the parameter **width**, a call of **WriteReal** shall behave as a call of **WriteFixed**, with the number of decimal places limited to those that can be shown in the given width. Otherwise, **WriteReal** shall behave as **WriteFloat** with a number of significant digits of at least one, limited to those that can be included together with the sign and exponent part in the given width.

In the special case of a width of 0, the effect shall be as for a call of **WriteFloat** with the parameter **sigFigs** equal to 0 and the parameter **width** also equal to 0.

9.2.2.4 The module LongIO

The module LongIO allows the reading and writing of real numbers in decimal text form. Real number parameters are of the pervasive type LONGREAL.

9.2.2.4.1 The definition module of LongIO

```
DEFINITION MODULE LongIO;
```

```
(* Input and output of real numbers in decimal text form *)
```

```
FROM IOChannel
  IMPORT ChanId;
```

```
(* the text form of a signed fixed-point real number is
   ["+" | "-"], decimal digit, {decimal digit},
   [".", {decimal digit}]
*)
```

```
(* the text form of a signed floating-point real number is
   signed fixed-point real number,
   "E", ["+" | "-"], decimal digit, {decimal digit}
*)
```

```
PROCEDURE ReadLong(cid: ChanId; VAR long: LONGREAL);
  (* Skips leading spaces and removes any remaining characters
   that form part of a signed fixed or floating point number.
   A corresponding value is assigned to the parameter long.
   The read result is set to the value allRight, outOfRange, wrongFormat,
   endOfLine, or endOfInput. *)
```

```
PROCEDURE WriteLongFloat(
  cid: ChanId; long: LONGREAL; sigFigs: CARDINAL; width: CARDINAL
);
  (* Writes the value of the parameter real in floating-point text form
   with sigFigs significant figures in a field of the given minimum width. *)

  (* Examples of floating point output:
   value:      3923009   39.23009   0.0003923009
   sigFigs
   1           4E+6      4E+1       4E-4
   2           3.9E+6    3.9E+1     3.9E-4
   5           3.9230E+6 3.9230E+1 3.9230E-4
  *)
```

```
PROCEDURE WriteLongEng(
  cid: ChanId; long: LONGREAL; sigFigs: CARDINAL; width: CARDINAL
);
  (* As for WriteLongFloat except that the number is scaled with one to
   three digits in the whole number part and with an exponent that is
   a multiple of three. *)

  (* Examples of floating point engineering output:
   value:      3923009   39.23009   0.0003923009
   sigFigs
   1           4E+6      40         400E-6
  *)
```

2	3.9E+6	39	390E-6
5	3.9230E+6	39.230	392.30E-6

*)

```

PROCEDURE WriteLongFixed(
  cid: ChanId; long: LONGREAL; place: INTEGER; width: CARDINAL
);
(* Writes the value of the parameter long in fixed-point text form,
rounded to the given place relative to the decimal point,
in a field of the given minimum width. *)

(* Examples of fixed point output:
value:      3923009    3.923009    0.0003923009
places
-5          3920000          0          0
-2          3923010          0          0
-1          3923009          4          0
0           3923009.          4.          0.
1           3923009.0         3.9         0.0
4           3923009.0000      3.9230      0.0004
*)

PROCEDURE WriteLong(cid: ChanId; long: LONGREAL; width: CARDINAL);
(* Writes the value of long as WriteLongFixed if the sign and magnitude
can be shown in the given width, or otherwise as WriteLongFloat.
The number of places or significant digits depend on the given width. *)

END LongIO.
```

9.2.2.4.2 The dynamic semantics of LongIO

The text form of a signed fixed-point real number shall be

["+" | "-"], decimal digit, {decimal digit},
[".", {decimal digit}]

The text form of a signed floating-point real number shall be

signed fixed-point real number,
"E", ["+" | "-"], decimal digit, {decimal digit}

The procedure ReadLong

A call of **ReadLong** shall skip any leading spaces and read characters that form part of a signed fixed or floating point number. The read result for the channel shall be set to the value

allRight	if a signed whole number is read and the value is in the range of the pervasive type LONGREAL —the value shall be assigned to the parameter long ;
outOfRange	if a signed real number is read but the value is out of range of the values of type LONGREAL —the value MAX(LONGREAL) or MIN(LONGREAL) shall be assigned to long according to the sign of the number
wrongFormat	if there are characters read or to be read but these are not in the format of a signed real number —the value of long is not defined;
endOfLine	if no characters are read, the next item being a line mark —the value of long is not defined;
endOfInput	if no characters are read, the input having ended —the value of long is not defined.

The procedure **WriteLongFloat**

A call of **WriteLongFloat** shall write the value of the parameter **long** in floating-point text form with leading spaces as required to make the number of characters written at least that given by the value of the parameter **width**. A sign shall be written only for negative values. In the special case of a value of zero for **width**, exactly one leading space shall be written.

One significant digit shall be included in the whole number part. The signed exponent part shall be included only if the exponent value is not 0. If the value of the parameter **sigFigs** is greater than 0, that number of significant digits shall be included, otherwise an implementation-defined number of significant digits shall be included. The decimal point shall not be included if there are no significant digits in the fractional part.

The procedure **WriteLongEng**

A call of **WriteLongEng** shall write the value of the parameter **long** in floating-point text form as for **WriteLongFloat**, except that the number shall be scaled with one to three digits in the whole number part and with an exponent that is a multiple of three.

The procedure **WriteLongFixed**

A call of **WriteLongFixed** shall write the value of the parameter **long** in fixed-point text form with leading spaces as required to make the number of characters written at least that given by the value of the parameter **width**. A sign shall be written only for negative values. In the special case of a value of zero for **width**, exactly one leading space shall be written.

At least one digit shall be included in the whole number part. The value shall be rounded to the given value of **place** relative to the decimal point which shall be suppressed if **place** is less than 0.

The procedure **WriteLong**

If the sign and magnitude of the parameter **long** can be expressed in a field given by the parameter **width**, a call of **WriteLong** shall behave as a call of **WriteLongFixed**, with the number of decimal places limited to those that can be shown in the given width. Otherwise, **WriteLong** shall behave as **WriteLongFloat** with a number of significant digits of at least one, limited to those that can be included together with the sign and exponent part in the given width.

In the special case of a width of 0, the effect shall be as for a call of **WriteLongFloat** with the parameter **sigFigs** equal to 0 and the parameter **width** also equal to 0.

9.2.2.5 The module BinIO

The module **BinIO** allows for the reading and writing of data using binary operations.

9.2.2.5.1 The definition module of BinIO

```
DEFINITION MODULE BinIO;
```

```
(* Reading and writing data using binary operations *)
```

```
FROM IOChannel IMPORT
```

```
  ChanId;
```

```
FROM SYSTEM IMPORT
```

```
  LOC;
```

```
PROCEDURE Read(cid: ChanId; VAR to: ARRAY OF LOC);
```

```

    (* Reads storage units and assigns to successive components of the
    parameter to.
    The read result is set to the value allRight, wrongFormat, or
    endOfInput *)

PROCEDURE Write(cid: ChanId; from: ARRAY OF LOC);
    (* Writes storage units from successive components of the parameter
    from *)

END BinIO.

```

9.2.2.5.2 The dynamic semantics of BinIO

The procedure Read

A call of **Read** shall read successive storage units from the channel and assign them to successive components of the parameter **to**. The read result for the channel shall be set to the value

allRight if items are read for all components;
wrongFormat if some items are read but not for all components;
endOfInput if no items are read, the input having ended.

$Read : Chan-id \times \mathbb{N}_1 \xrightarrow{o} Loc^*$

$Read(chid, n) \triangleq$
 442 $def\ did = device(chid);$
 443 $let\ op = device-op(did, BINREAD)\ in$
 406 $def\ mk-(locs, res) = get(chid, op, n);$
 $let\ nlocs = len\ locs\ in$
 $(if\ nlocs < n \wedge nlocs > 0$
 435 $then\ set-result(did, WRONG-FORMAT)$
 $else\ skip;$
 $return\ locs\)$

The procedure Write

A call of **Write** shall write successive components of the parameter **from** to the channel as storage units without interpretation.

$Write : Chan-id \times Loc^* \xrightarrow{o} ()$

$Write(chid, from) \triangleq$
 442 $def\ did = device(chid);$
 443 $let\ binwop = device-op(did, BINWRITE)\ in$
 397 $binwop(chid, from)$

9.2.2.6 The module IOConsts

The module **IOConsts** defines the type used to express read results. Programs do not normally need to import from **IOConsts** directly since client modules define identifiers that correspond to those defined by this module.

9.2.2.6.1 The definition module of IOConsts

```

DEFINITION MODULE IOConsts;

```

```

(* Types and constants for input/output modules *)

(* The following type is used to classify the result of an input operation: *)

TYPE
  ReadResultEnum = (
    notKnown,
    (* no read result is set *)
    allRight,
    (* data is as expected or as required *)
    outOfRange,
    (* data cannot be represented *)
    wrongFormat,
    (* data not in expected format *)
    endOfLine,
    (* end of line seen before expected data *)
    endOfInput
    (* end of input seen before expected data *)
  );

END IOConsts.

```

9.2.2.7 The module IORes

The module `IORes` allows the program to determine whether the last operation to read data from a specified input channel found data in the required format.

9.2.2.7.1 The definition module of IORes

```

DEFINITION MODULE IORes;

(* Obtain read results on specified channels *)

IMPORT
  IOConsts;
FROM IOChannel IMPORT
  ChanId;

TYPE
  ReadResultEnum = IOConsts.ReadResultEnum;

(*
  ReadResultEnum = (
    notKnown,
    (* no read result is set *)
    allRight,
    (* data is as expected or as required *)
    outOfRange,
    (* data cannot be represented *)
    wrongFormat,
    (* data not in expected format *)
    endOfLine,
    (* end of line seen before expected data *)
    endOfInput
    (* end of input seen before expected data *)
  )

```

```

    );
*)

PROCEDURE ReadResult(cid: ChanId): ReadResultEnum;
    (* Returns the result for the last read operation on the channel *)

END IORes.

```

9.2.2.7.2 The dynamic semantics of IORes

The procedure ReadResult

A call of **ReadResult** shall return the stored read result for the channel identified by the parameter **cid**.

$ReadResult : Chan-id \xrightarrow{o} Read-result$

```

ReadResult(chid)  $\triangleq$ 
442   def did = device(chid);
436   result(did)

```


9.2.3 Reading and Writing of Data over Default Channels

The modules `STextIO`, `SWholeIO`, `SRealIO`, `SLongIO`, `SBinIO`, and `SIORes` provide procedures that operate on default input and output channels and so do not take a parameter identifying a channel.

9.2.3.1 The module `STextIO`

The module `STextIO` corresponds to `TextIO`.

9.2.3.1.1 The definition module of `STextIO`

```
DEFINITION MODULE STextIO;
```

```
(* Character and string text operations on default channels *)
```

```
(* The following procedures do not read past line marks: *)
```

```
PROCEDURE ReadChar(VAR ch: CHAR);
```

```
(* If possible, removes a character from the input stream and assigns  
the corresponding value to the parameter ch.
```

```
The read result is set to allRight, endOfLine or endOfInput. *)
```

```
PROCEDURE ReadLine(VAR s: ARRAY OF CHAR);
```

```
(* Removes any remaining characters before the next line mark copying  
as many as can be accommodated to the parameter s as a string value.
```

```
The read result is set to the value allRight, outOfRange, endOfLine,  
or endOfInput. *)
```

```
PROCEDURE ReadString(VAR s: ARRAY OF CHAR);
```

```
(* Removes and copies only those characters before the next line mark  
that can be accommodated to the parameter s as a string value.
```

```
The read result is set to the value allRight, endOfLine, or endOfInput. *)
```

```
PROCEDURE ReadToken(VAR s: ARRAY OF CHAR);
```

```
(* Skips leading spaces and then removes characters before the next  
space or line mark copying as many as can be accommodated to the  
parameter s as a string value.
```

```
The read result is set to the value allRight, outOfRange, endOfLine,  
or endOfInput. *)
```

```
(* The following procedure reads past the next line mark *)
```

```
PROCEDURE ReadLn;
```

```
(* Removes successive items from the input stream up to and including  
the next line mark or until the end of input is reached.
```

```
The read result is set to the value allRight, or endOfInput. *)
```

```
(* Output procedures *)
```

```
PROCEDURE WriteChar(ch: CHAR);
```

```
(* Writes the parameter ch to the output stream *)
```

```
PROCEDURE WriteLn;
```

```
(* Writes a line mark to the output stream *)
```

```

PROCEDURE WriteString(s: ARRAY OF CHAR);
  (* Writes the string value of the parameter s to the output stream *)

END STextIO.

```

9.2.3.1.2 The dynamic semantics of STextIO

The procedures of the module `STextIO` shall behave as the corresponding procedures of the module `TextIO` except that input shall be taken from the default input channel and output shall be sent over the default output channel.

9.2.3.2 The module SWholeIO

The module `SWholeIO` corresponds to `WholeIO`.

9.2.3.2.1 The definition module of SWholeIO

```

DEFINITION MODULE SWholeIO;

(* Input and output of whole numbers in text form over default channels *)

(* the text form of a signed whole number is
   ["+" | "-"], decimal digit, {decimal digit} *)

PROCEDURE ReadInt(VAR int: INTEGER);
  (* Skips leading spaces and removes any remaining characters
     that form part of a signed whole number.
     A corresponding value is assigned to the parameter int.
     The read result is set to the value allRight, outOfRange, wrongFormat,
     endOfLine, or endOfInput. *)

PROCEDURE WriteInt(int: INTEGER; width: CARDINAL);
  (* Writes the value of the parameter int in text form in a field of the
     given minimum width. *)

(* the text form of an unsigned whole number is
   decimal digit, {decimal digit} *)

PROCEDURE ReadCard(VAR card: CARDINAL);
  (* Skips leading spaces and removes any remaining characters
     that form part of an unsigned whole number.
     A corresponding value is assigned to the parameter card.
     The read result is set to the value allRight, outOfRange, wrongFormat,
     endOfLine, or endOfInput. *)

PROCEDURE WriteCard(card: CARDINAL; width: CARDINAL);
  (* Writes the value of the parameter card in text form in a field of the
     given minimum width. *)

END SWholeIO.

```

9.2.3.2.2 The dynamic semantics of SWholeIO

The procedures of the module `SWholeIO` shall behave as the corresponding procedures of the module `WholeIO` except that input shall be taken from the default input channel and output shall be sent over the default output channel.

9.2.3.3 The module SRealIO

The module `SRealIO` corresponds to `RealIO`.

9.2.3.3.1 The definition module of SRealIO

```
DEFINITION MODULE SRealIO;

(* Input and output of real numbers in decimal text form over default channels *)

PROCEDURE ReadReal(VAR real: REAL);
  (* Skips leading spaces and removes any remaining characters
   that form part of a signed fixed or floating point number.
   A corresponding value is assigned to the parameter real.
   The read result is set to the value allRight, outOfRange, wrongFormat,
   endOfLine, or endOfInput. *)

PROCEDURE WriteFloat(real: REAL; sigFigs: CARDINAL; width: CARDINAL);
  (* Writes the value of the parameter real in floating-point text form
   with sigFigs significant figures in a field of the given minimum width. *)

PROCEDURE WriteEng(real: REAL; sigFigs: CARDINAL; width: CARDINAL);
  (* As for WriteFloat except that the number is scaled with one to
   three digits in the whole number part and with an exponent that is
   a multiple of three. *)

PROCEDURE WriteFixed(real: REAL; place: INTEGER; width: CARDINAL);
  (* Writes the value of the parameter real in fixed-point text form,
   rounded to the given place relative to the decimal point,
   in a field of the given minimum width. *)

PROCEDURE WriteReal(real: REAL; width: CARDINAL);
  (* Writes the value of real as WriteFixed if the sign and magnitude
   can be shown in the given width, or otherwise as WriteFloat.
   The number of places or significant digits depend on the given width. *)

END SRealIO.
```

9.2.3.3.2 The dynamic semantics of SRealIO

The procedures of the module `SRealIO` shall behave as the corresponding procedures of the module `RealIO` except that input shall be taken from the default input channel and output shall be sent over the default output channel.

9.2.3.4 The module SLongIO

The module `SLongIO` corresponds to `LongIO`.

9.2.3.4.1 The definition module of SLongIO

```
DEFINITION MODULE SLongIO;

(* Input and output of real numbers in decimal text form over default channels *)

PROCEDURE ReadLong(VAR long: LONGREAL);
  (* Skips leading spaces and removes any remaining characters
  that form part of a signed fixed or floating point number.
  A corresponding value is assigned to the parameter long.
  The read result is set to the value allRight, outOfRange, wrongFormat,
  endOfLine, or endOfInput. *)

PROCEDURE WriteLongFloat(long: LONGREAL; sigFigs: CARDINAL; width: CARDINAL);
  (* Writes the value of the parameter real in floating-point text form
  with sigFigs significant figures in a field of the given minimum width. *)

PROCEDURE WriteLongEng(long: LONGREAL; sigFigs: CARDINAL; width: CARDINAL);
  (* As for WriteLongFloat except that the number is scaled with one to
  three digits in the whole number part and with an exponent that is
  a multiple of three. *)

PROCEDURE WriteLongFixed(long: LONGREAL; place: INTEGER; width: CARDINAL);
  (* Writes the value of the parameter long in fixed-point text form,
  rounded to the given place relative to the decimal point,
  in a field of the given minimum width. *)

PROCEDURE WriteLong(long: LONGREAL; width: CARDINAL);
  (* Writes the value of long as WriteLongFixed if the sign and magnitude
  can be shown in the given width, or otherwise as WriteLongFloat.
  The number of places or significant digits depend on the given width. *)

END SLongIO.
```

9.2.3.4.2 The dynamic semantics of SLongIO

The procedures of the module `SLongIO` shall behave as the corresponding procedures of the module `LongIO` except that input shall be taken from the default input channel and output shall be sent over the default output channel.

9.2.3.5 The module SBinIO

The module `SBinIO` corresponds to `BinIO`.

9.2.3.5.1 The definition module of SBinIO

```
DEFINITION MODULE SBinIO;

(* Reading and writing data using binary operations over default channels *)

FROM SYSTEM IMPORT
  LOC;

PROCEDURE Read(VAR to: ARRAY OF LOC);
  (* Reads storage units and assigns to successive components of the
```

```

parameter to.
The read result is set to the value allRight, wrongFormat, or
endOfInput *)

PROCEDURE Write(from: ARRAY OF LOC);
  (* Writes storage units from successive components of the parameter
  from *)

END SBinIO.

```

9.2.3.5.2 The dynamic semantics of SBinIO

The procedures of the module **SBinIO** shall behave as the corresponding procedures of the module **BinIO** except that input shall be taken from the default input channel and output shall be sent over the default output channel.

9.2.3.6 The module SIORes

The module **SIORes** corresponds to **IORes**.

9.2.3.6.1 The definition module of SIORes

```

DEFINITION MODULE SIORes;

(* Obtain read results on the default channel *)

IMPORT
  IOConsts;

TYPE
  ReadResultEnum = IOConsts.ReadResultEnum;

(*
  ReadResultEnum = (
    notKnown,
    (* no read result is set *)
    allRight,
    (* data is as expected or as required *)
    outOfRange,
    (* data cannot be represented *)
    wrongFormat,
    (* data not in expected format *)
    endOfLine,
    (* end of line seen before expected data *)
    endOfInput
    (* end of input seen before expected data *)
  );
*)

PROCEDURE ReadResult(): ReadResultEnum;
  (* Returns the result for the last read operation
  on the default input channel *)

END SIORes.

```

9.2.3.6.2 The dynamic semantics of SIORes

The procedure ReadResult

A call of **ReadResult** shall return the stored read result for the default input channel.

NOTE — The existence of the module **IOConsts** allows the definition module **SIORes** to be independent of **IORes** and **IOChannel**.

9.2.3.7 Auxiliary definitions used in the device independent modules

operations

```

read-at-most : Dev-id × ℕ  $\xrightarrow{o}$  Char*
read-at-most (did, n)  $\triangleq$ 
443   let ramop = device-op (did, READTEXT) in
405   ramop (did, n) ;

read-while : Dev-id × (Char* → B)  $\xrightarrow{o}$  Char*
read-while (did, f)  $\triangleq$ 
405   read-until-false ([], did, f) ;

read-until-false : Char* × Chan-id × (Char* → B)  $\xrightarrow{o}$  Char*
read-until-false (ins, did, continue-reading)  $\triangleq$ 
443   let rflop = device-op (did, LOOK) in
405   def ch = rflop (did, 1) ;
436   def res = result (did) ;
      if res = ALL-RIGHT
      then let ans = ins  $\curvearrowright$  [ch] in
405         if continue-reading (ans)
406         then (remove (chid) ;
405             read-until-false (ans, chid, continue-reading))
      else return ins
      else return ins ;

write : Chan-id × Char  $\xrightarrow{o}$  ()
write (chid, ch)  $\triangleq$ 
      let dev = devices (chid) in
      let wop = dev.ops (WRITETEXT) in
405   wop (chid, [ch], 1)

annotations      Write a character to the output stream.
;

write-chars : Chan-id × Char*  $\xrightarrow{o}$  ()
write-chars (chid, chars)  $\triangleq$ 
      let dev = devices (chid) in
      let wtop = dev.ops (WRITETEXT) in
405   wtop (chid, chars, len chars)

annotations      Write a string to the output stream.
;

write-eol : Chan-id  $\xrightarrow{o}$  ()
write-eol (chid)  $\triangleq$ 
      let dev = devices (chid) in
      let weop = dev.ops (WRITELN) in
405   weop (chid)

```

```

    annotations      Write an end of line marker to the output stream.
;

get : Dev-id × Device-op × ℕ  $\xrightarrow{o}$  ((Loc* | Char*) × Read-result)
get (did, drop, n)  $\triangleq$ 
406   def mk- (ins, r) = drop(did, n) ;
      if r = ALL-RIGHT ∧ len ins < n
406   then def mk- (retry, rr) = get(did, drop, n - len ins) ;
      return mk- (ins ⋈ retry, rr)
      else return mk- (ins, r)
annotations      Call a device ‘input’ procedure. Retry if some, but not enough, data returned.
;

remove : Dev-id  $\xrightarrow{o}$  ()
remove (did, n)  $\triangleq$ 
443   let remop = device-op (did, SKIP) in
406   remop(did)

```

9.2.4 Obtaining Channels from Device Modules

Separate device modules are defined for obtaining new channels connected to sequential streams, rewindable sequential files, random access files, and a terminal device. The request to obtain a channel is made by calling the appropriate *open* procedure, supplying information to identify the source or destination to which a connection is to be made. The required input/output operations are specified using combinations of flags which are defined in terms of constants imported from the module `IOChannel`.

The open procedures return values which indicate the success, or otherwise, of the request. These values originate in the module `DevConsts`.

Each of these device modules defines a predicate allowing a check to be made that a given channel was opened by that module and a *close* procedure to break the connection and release the channel. Procedures are also provided for device-dependent operations, such as setting the read/write position on a random access file.

A further device module is defined to allow access to the program arguments over a pre-opened channel.

9.2.4.1 Implementation-defined functions used in the device modules

functions

$isa-Char : Loc^* \rightarrow \mathbb{B}$

$isa-Char(s) \triangleq$
implementation-defined

annotations Return whether an external representation of data is representable as a character.

;

$emit : Char^* \rightarrow Loc^*$

$emit(its) \triangleq$
implementation-defined

annotations Return the external representation of a sequence of characters

;

$pos-to-filepos : Chan-id \times \mathbb{N} \xrightarrow{o} Filepos$

$pos-to-filepos(chid, n) \triangleq$

407 if $\exists fp : Filepos \cdot filepos-to-pos(chid, fp) = n$
407 then return $\iota fp : Filepos \cdot filepos-to-pos(chid, fp) = n$

306 else *mandatory-exception*(POSRANGE) ;

$filepos-to-pos : Chan-id \times Filepos \rightarrow \mathbb{N}$

$filepos-to-pos(chid, fp) \triangleq$
implementation-defined;

$is_legal_filename : Char^* \rightarrow \mathbb{B}$

$is_legal_filename(name) \triangleq$
implementation-defined;

$is_existing_file : Char^* \rightarrow \mathbb{B}$

$is_existing_file(name) \triangleq$
implementation-defined

407 pre $is_legal_filename(name)$

9.2.4.2 The module DevConsts

The module **DevConsts** defines common types and values for use with open procedures. Programs do not normally need to import from **DevConsts** directly since device modules define identifiers that correspond to those defined by this module.

9.2.4.2.1 The definition module of DevConsts

DEFINITION MODULE DevConsts;

(* Common types and values for device open requests and results *)

(* Device modules may allow combinations of the following flags to be given when a channel is opened. An open procedure may not accept or honour all combinations. *)

TYPE

```
FlagEnum = (  
  readFlag,  
    (* input operations are requested/available *)  
  writeFlag,  
    (* output operations are requested/available *)  
  oldFlag,  
    (* a file may/must/did exist before the channel was opened *)  
  textFlag,  
    (* text operations are requested/available *)  
  binaryFlag,  
    (* binary operations are requested/available *)  
  interactiveFlag,  
    (* interactive use is requested/applies *)  
  echoFlag  
    (* echoing by interactive device on removal of characters from input  
    stream requested/applies *)  
);  
FlagSet = SET OF FlagEnum;
```

(* Singleton values of FlagSet, to allow for example, read+write. *)

CONST

```
read = FlagSet{readFlag};
```

```

    (* input operations are requested/available *)
write = FlagSet{writeFlag};
    (* output operations are requested/available *)
old = FlagSet{oldFlag};
    (* a file may/must/did exist before the channel was opened *)
text = FlagSet{textFlag};
    (* text operations are requested/available *)
binary = FlagSet{binaryFlag};
    (* binary operations are requested/available *)
interactive = FlagSet{interactiveFlag};
    (* interactive use is requested/applies *)
echo = FlagSet{echoFlag};
    (* echoing by interactive device on removal of characters from input
    stream requested/applies *)

(* Possible results of open requests: *)

TYPE
OpenResultEnum =
    (opened,
        (* the open succeeded as requested *)
    wrongNameFormat,
        (* given name is in the wrong format for the implementation *)
    wrongFlags,
        (* given flags include a value that does not apply to the device *)
    tooManyOpen,
        (* this device cannot support any more open channels *)
    outOfChans,
        (* no more channels can be allocated *)
    wrongPermissions,
        (* file or directory permissions do not allow request *)
    noRoomOnDevice,
        (* storage limits on the device prevent the open *)
    noSuchFile,
        (* a needed file does not exist *)
    fileExists,
        (* a file of the given name already exists when a new one is required *)
    wrongFileType,
        (* the file is of the wrong type to support the required operations *)
    noTextOperations,
        (* text operations have been requested but are not supported *)
    noBinaryOperations,
        (* binary operations have been requested but are not supported *)
    noMixedOperations,
        (* text and binary operations have been requested but they
        are not supported in combination *)
    otherProblem
        (* open failed for some other reason *)
    );

```

END DevConsts.

9.2.4.2.2 Dynamic semantics

In a call of a device module open procedure with a request parameter of type **FlagSet** and a result parameter of type **OpenResultEnum**:

If the result is **opened**, the following operations shall be provided for the opened channel for the combinations of request flags shown:

	read	write	
text	text input	text output	—as defined for the device
binary	binary input	binary output	—as defined for the device

NOTE — The meaning of the other flags is given for the open operations to which they apply.

If the result is other than **opened**, the channel parameter shall be assigned the value identifying the *bad* channel on which no input/output operations are provided. The result shall be chosen according to the following table:

wrongNameFormat	if the given name is not in the format defined for the implementation
wrongFlags	if the given flags include a value that does not apply to the device
tooManyOpen	if the device cannot support any more open channels
outOfChans	if no more channels can be allocated
wrongPermissions	if file or directory permissions do not allow the request to be met
noRoomOnDevice	if storage limits on the device do not allow the request to be met
noSuchFile	if a needed file does not exist
fileExists	if a file of the given name already exists when a new one is required
wrongFileType	if the named file is of the wrong type to support the required operations
noTextOperations	if text operations have been requested but are not supported by the device
noBinaryOperations	if binary operations have been requested but are not supported by the device
noMixedOperations	if text and binary operations have been requested but they are not supported in combination by the device
otherProblem	if the open failed for a reason other than the above

9.2.4.3 The module SeqStream

The module **SeqStream** allows new channels to be obtained that are connected to a named external source and/or destination for independent sequential data streams.

9.2.4.3.1 The definition module of SeqStream

```
DEFINITION MODULE SeqStream;

(* Independent sequential data streams *)

IMPORT
    IOChannel, DevConsts;
FROM DevConsts IMPORT
    FlagEnum;

TYPE
    ChanId = IOChannel.ChanId;
    FlagSet = DevConsts.FlagSet;
    OpenResultEnum = DevConsts.OpenResultEnum;

(* Accepted singleton values of FlagSet: *)

CONST
    read = FlagSet{readFlag};
    (* input operations are requested/available *)
    write = FlagSet{writeFlag};
    (* output operations are requested/available *)
```

```

old = FlagSet{oldFlag};
(* a file may/must/did exist before the channel was opened *)
text = FlagSet{textFlag};
(* text operations are requested/available *)
binary = FlagSet{binaryFlag};
(* binary operations are requested/available *)

PROCEDURE Open(
  VAR cid: ChanId;
    name: ARRAY OF CHAR;
    flags: FlagSet;
  VAR res: OpenResultEnum
);
(* The read flag implies old;
without the binary flag, text is implied.
If successful, assigns to the parameter cid the identity of a channel
open to a source/destination of the given name and assigns the value
opened to the parameter res.
If a channel cannot be opened as required, the value of the parameter
res indicates the reason and cid identifies the bad channel. *)

PROCEDURE IsSeqStream(cid: ChanId): BOOLEAN;
(* Tests if the channel is open to a sequential stream *)

PROCEDURE Close(VAR cid: ChanId);
(* If the channel is not open to a sequential stream, the exception
wrongDevice is raised. Otherwise, the channel is closed and the value
identifying the bad channel is assigned to the parameter cid *)

END SeqStream.

```

9.2.4.3.2 The dynamic semantics of SeqStream

In a request to open a sequential stream, the flags **read**, **write**, **old**, **text**, and **binary** shall apply. If **binary** is not included in the request parameter **flags**, inclusion of **text** shall be implied.

The procedure **Open**

In a call of **Open**, inclusion of **read** in the parameter **flags** shall imply inclusion of **old** and a source of the given name shall already exist for the call to succeed.

If **write** is included, a destination of the given name shall not already exist unless the flag **old** is given or implied.

If the call is successful, the parameter **cid** shall identify a channel open to a source and/or destination of the given name and the value **opened** shall be assigned to the parameter **res**.

If a channel cannot be opened as required, the value of the parameter **res** shall indicate the reason and **cid** shall identify the bad channel.

NOTE — Distinct modes in combination with **text** and/or **binary** are given by the following equivalent sets of flags:

read from an existing source:	read	old	read+old
write to a new destination:	write		
write to a new or old destination:	write+old		
read/write an existing source/destination:	read+write	read+write+old	

The procedure IsSeqStream

A call of **IsSeqStream** shall return **TRUE** if the channel is open to a sequential stream and **FALSE** otherwise.

The procedure Close

In a call of **Close**, if the channel identified by the parameter **cid** is not open to a sequential stream, the exception **wrongDevice** shall be raised. Otherwise, the channel shall be closed and the value identifying the bad channel shall be assigned to **cid**.

9.2.4.4 The module RewFile

The module **RewFile** allows new channels to be opened to rewindable stored files. If opened for both writing and reading, data written to the file may be read back from the start of the file. Rewriting from the start of the file causes the previous contents to be lost.

9.2.4.4.1 The definition module of RewFile

```
DEFINITION MODULE RewFile;
```

```
(* Rewindable sequential files *)
```

```
IMPORT
```

```
    IOChannel, DevConsts;
```

```
FROM DevConsts IMPORT
```

```
    FlagEnum;
```

```
TYPE
```

```
    ChanId = IOChannel.ChanId;
```

```
    FlagSet = DevConsts.FlagSet;
```

```
    OpenResultEnum = DevConsts.OpenResultEnum;
```

```
(* Accepted singleton values of FlagSet: *)
```

```
CONST
```

```
    read = FlagSet{readFlag};
```

```
    (* input operations are requested/available *)
```

```
    write = FlagSet{writeFlag};
```

```
    (* output operations are requested/available *)
```

```
    old = FlagSet{oldFlag};
```

```
    (* a file may/must/did exist before the channel was opened *)
```

```
    text = FlagSet{textFlag};
```

```
    (* text operations are requested/available *)
```

```
    binary = FlagSet{binaryFlag};
```

```
    (* binary operations are requested/available *)
```

```
PROCEDURE OpenWrite(
```

```
    VAR cid: ChanId;
```

```
    name: ARRAY OF CHAR;
```

```
    flags: FlagSet;
```

```
    VAR res: OpenResultEnum
```

```
);
```

```
(* The write flag is implied;
```

```
without the binary flag, text is implied.
```

```

    If successful,
    assigns to the parameter cid the identity of a channel open to a stored file
    of the given name and assigns the value opened to the parameter res.
    The file is of zero length.
    Output mode is selected.
    If a channel cannot be opened as required, the value of the parameter
    res indicates the reason and cid identifies the bad channel. *)

PROCEDURE OpenAppend(
    VAR cid: ChanId;
        name: ARRAY OF CHAR;
        flags: FlagSet;
    VAR res: OpenResultEnum
);
    (* The write and old flags are implied;
    without the binary flag, text is implied.
    If successful,
    assigns to the parameter cid the identity of a channel open to a stored file
    of the given name and assigns the value opened to res.
    Output mode is selected and the write position corresponds to the length of
    the file.
    If a channel cannot be opened as required, the value of the parameter res
    indicates the reason and cid identifies the bad channel. *)

PROCEDURE OpenRead(
    VAR cid: ChanId;
        name: ARRAY OF CHAR;
        flags: FlagSet;
    VAR res: OpenResultEnum
);
    (* The read and old flags are implied;
    without the binary flag, text is implied.
    If successful,
    assigns to the parameter cid the identity of a channel open to a stored file
    of the given name and assigns the value opened to res.
    Input mode is selected.
    If a channel cannot be opened as required, the value of the parameter res
    indicates the reason and cid identifies the bad channel. *)

PROCEDURE IsRewFile(cid: ChanId): BOOLEAN;
    (* Tests if the channel is open to a rewindable sequential file *)

PROCEDURE Reread(cid: ChanId);
    (* If the channel is not open to a rewindable file, the exception
    wrongDevice is raised. Otherwise,
    if successful, sets the read position to the start of the file and selects
    input mode.
    If the operation cannot be performed, perhaps because of insufficient
    permissions, neither input mode nor output mode is selected. *)

PROCEDURE Rewrite(cid: ChanId);
    (* If the channel is not open to a rewindable file, the exception
    wrongDevice is raised. Otherwise,
    if successful, truncates the file to zero length and selects output mode.
    If the operation cannot be performed, perhaps because of insufficient
    permissions, neither input mode nor output mode is selected.
    *)

```

```

PROCEDURE Close(VAR cid: ChanId);
  (* If the channel is not open to a rewindable file, the exception
  wrongDevice is raised. Otherwise,
  the channel is closed and the value identifying the bad channel
  is assigned to the parameter cid *)

END RewFile.

```

9.2.4.4.2 The dynamic semantics of RewFile

In a request to open a rewindable file, the flags **read**, **write**, **old**, **text**, and **binary** shall apply. If **binary** is not included in the request parameter **flags**, inclusion of **text** shall be implied.

Channels open to rewindable sequential files may be in input mode or in output mode.

In input mode, only input operations shall be available — ‘(IOChannel.Flags()*(read+write) = read)’ shall be **TRUE**, and an attempt to write over the channel shall raise the exception **notAvailable**.

In output mode, only output operations shall be available — ‘(IOChannel.Flags()*(read+write) = write)’ shall be **TRUE**, and an attempt to read from the channel shall raise the exception **notAvailable**.

The procedure OpenWrite

In a call of **OpenWrite**, inclusion of the **write** flag in the parameter **flags** shall be implied.
A destination of the given name shall not already exist unless the flag **old** is given.

If the **read** flag is included in the request, the **Reread** operation shall be available for the open to succeed.

If the call is successful, the parameter **cid** shall identify a channel open to a stored file of the given name and the value **opened** shall be assigned to the parameter **res**.

The file shall be of zero length.

Output mode shall be selected.

If a channel cannot be opened as required, the value of the parameter **res** shall indicate the reason and **cid** shall identify the bad channel.

NOTE — Distinct modes in combination with **text** and/or **binary** are given by the following equivalent sets of flags:

write to a new file:	write	
write to a new file or a truncated existing file:	old	write+old
write to a new file, need read permission:	write+read	read
write to a new or existing file, need read permission:	old+read	write+old+read

The procedure OpenAppend

In a call of **OpenAppend**, inclusion of the **write** and **old** flags in the parameter **flags** shall be implied and a destination of the given name may already exist.

If the **read** flag is included in the request, the **Reread** operation shall be available for the open to succeed.

If the call is successful, the parameter **cid** shall identify a channel open to a stored file of the given name and the value **opened** shall be assigned to the parameter **res**.

The write position shall correspond to the length of the file.

Output mode shall be selected.

If a channel cannot be opened as required, the value of the parameter **res** shall indicate the reason and **cid** shall identify the bad channel.

NOTE — Distinct modes in combination with **text** and/or **binary** are given by the following equivalent sets of flags:

write to a new or append to an existing file:	write	old	write+old	
write to a new or append to an existing file, need read permission:	read	write+read	old+read	write+old+read

The procedure **OpenRead**

In a call of **OpenRead**, inclusion of the **read** and **old** flags in the parameter **flags** shall be implied and a destination of the given name shall already exist for the open to succeed.

If the **write** flag is included in the request, the **Rewrite** operation shall be available for the open to succeed.

If the call is successful, the parameter **cid** shall identify a channel open to a stored file of the given name and the value **opened** shall be assigned to the parameter **res**.

Input mode shall be selected.

If a channel cannot be opened as required, the value of the parameter **res** shall indicate the reason and **cid** shall identify the bad channel.

NOTE — Distinct modes in combination with **text** and/or **binary** are given by the following equivalent sets of flags:

read from an existing file:	read	old	read+old	
read from an existing file, need write permission:	write	read+write	old+write	read+old+write

The procedure **IsRewFile**

A call of **IsRewFile** shall return **TRUE** if the channel is open to a rewindable sequential file and **FALSE** otherwise.

The procedure **Reread**

In a call of **Reread**, if the channel is not open to a rewindable sequential file, the exception **wrongDevice** shall be raised. Otherwise, if successful, the read position shall be set to the start of the file and input mode shall be selected. If the operation cannot be performed, perhaps because of insufficient permissions, neither input mode nor output mode shall be selected.

The procedure **Rewrite**

If the channel is not open to a rewindable sequential file, the exception **wrongDevice** shall be raised. Otherwise, if successful, the file shall be truncated to zero length and output mode shall be selected.

If the operation cannot be performed, perhaps because of insufficient permissions, neither input mode nor output mode shall be selected.

The procedure **Close**

In a call of **Close**, if the channel identified by the parameter **cid** is not open to a rewindable sequential file, the exception **wrongDevice** shall be raised. Otherwise, the channel shall be closed and the value identifying the bad channel shall be assigned to **cid**.

9.2.4.5 The module **RndFile**

The module **RndFile** allows new channels to be opened to random access files.

9.2.4.5.1 The definition module of **RndFile**

DEFINITION MODULE **RndFile**;


```

(* Random access files *)

IMPORT
    IOChannel, DevConsts;
FROM DevConsts IMPORT
    FlagEnum;
FROM SYSTEM IMPORT
    LOC;

TYPE
    ChanId = IOChannel.ChanId;
    FlagSet = DevConsts.FlagSet;
    OpenResultEnum = DevConsts.OpenResultEnum;

(* Accepted singleton values of FlagSet: *)

CONST
    read = FlagSet{readFlag};
    (* input operations are requested/available *)
    write = FlagSet{writeFlag};
    (* output operations are requested/available *)
    old = FlagSet{oldFlag};
    (* a file may/must/did exist before the channel was opened *)
    text = FlagSet{textFlag};
    (* text operations are requested/available *)
    binary = FlagSet{binaryFlag};
    (* binary operations are requested/available *)

PROCEDURE OpenOld(
    VAR cid: ChanId;
        name: ARRAY OF CHAR;
        flags: FlagSet;
    VAR res: OpenResultEnum
);
    (* The old flag is implied;
    without the write flag, read is implied;
    without the text flag, binary is implied.
    If successful,
    assigns to the parameter cid the identity of a channel open to a
    random access file of the given name and assigns the value opened
    to the parameter res.
    The read and/or write position is at the start of the file.
    If a channel cannot be opened as required, the value of the parameter
    res indicates the reason and cid identifies the bad channel. *)

PROCEDURE OpenClean(
    VAR cid: ChanId;
        name: ARRAY OF CHAR;
        flags: FlagSet;
    VAR res: OpenResultEnum
);
    (* The write flag is implied;
    without the text flag, binary is implied.
    If successful,
    assigns to the parameter cid the identity of a channel open to a random

```

```

    access file of the given name and assigns the value opened to res.
    The file is of zero length.
    If a channel cannot be opened as required, the value of the parameter
    res indicates the reason and cid identifies the bad channel. *)

PROCEDURE IsRndFile(cid: ChanId): BOOLEAN;
    (* Tests if the channel is open to a random access file *)

TYPE
    RndFileExceptionEnum =
        (rndFileNoException,
         (* there is no exception in this context *)
         notRndFileException,
         (* there is an exception in this context, but from another source *)
         posRange
         (* required new file position cannot be represented as a value
          of type FilePos *))
    );

PROCEDURE RndFileException(): RndFileExceptionEnum;
    (* Returns the RndFileExceptionEnum value for the current context *)

CONST
    FilePosSize = 4;
    (* implementation defined number greater than 0 *)

TYPE
    FilePos = ARRAY [1 .. FilePosSize] OF LOC;

PROCEDURE StartPos(cid: ChanId): FilePos;
    (* If the channel is not open to a random access file, the exception
    wrongDevice is raised. Otherwise,
    returns the position of the start of the file *)

PROCEDURE CurrentPos(cid: ChanId): FilePos;
    (* If the channel is not open to a random access file, the exception
    wrongDevice is raised. Otherwise,
    returns the current read/write position *)

PROCEDURE EndPos(cid: ChanId): FilePos;
    (* If the channel is not open to a random access file, the exception
    wrongDevice is raised. Otherwise,
    returns the first position at or after which there have been no writes *)

PROCEDURE NewPos(
    cid: ChanId;
    chunks: INTEGER;
    chunkSize: CARDINAL;
    from: FilePos
): FilePos;
    (* If the channel is not open to a random access file, the exception
    wrongDevice is raised. Otherwise,
    returns the position chunks*chunkSize relative to the parameter from
    or raises the exception posRange if the required position cannot
    be represented as a value of type FilePos. *)

```

```

PROCEDURE SetPos(cid: ChanId; pos: FilePos);
  (* If the channel is not open to a random access file, the exception
  wrongDevice is raised. Otherwise, sets the read/write position to the
  value given by the parameter pos. *)

PROCEDURE Close(VAR cid: ChanId);
  (* If the channel is not open to a random access file, the exception
  wrongDevice is raised. Otherwise,
  the channel is closed and the value identifying the bad channel
  is assigned to the parameter cid *)

END RndFile.

```

9.2.4.5.2 The dynamic semantics of RndFile

Channels opened by the module **RndFile** shall have an associated read/write position in the corresponding random-access file. The read/write position shall be at the start of the file after opening and after a reset operation on the channel. It shall be moved forward by the number of positions occupied by data that are taken from the file by an input operation or written to the file by an output operation.

A random-access file shall have a length corresponding to the position after the highest read/write position at which data have been written, or shall be zero if no data have been written to the file. If the read/write position is set at the current length, either implicitly on an input or output operation, or explicitly by a positioning operation, the effect of an input operation shall be as if the input stream had ended. A write at that position shall, if necessary, attempt to allocate more physical storage for the file.

In a request to open a random-access file, the flags **read**, **write**, **old**, **text**, and **binary** shall apply. If **text** is not included in the request parameter **flags**, inclusion of **binary** shall be implied.

$rndfile\text{-}merge\text{-}flags : Characteristic\text{-}set \times Characteristic\text{-}set \rightarrow ([Characteristic\text{-}set] \times Open\text{-}result)$

$rndfile\text{-}merge\text{-}flags(given, required) \triangleq$
 let $default = \text{if } TEXT \in given \text{ then } \{ \} \text{ else } \{ BINARY \}$ in
 if $given \subseteq rndfile\text{-}flags$
 then $mk\text{-}(given \cup required \cup default, OPENED)$
 else $mk\text{-}(nil, WRONG\text{-}FLAGS)$

TO DO — Extend definition to permit the denial of mixed text and binary working.

$rndfile\text{-}check\text{-}file : Char^* \times Characteristic\text{-}set \rightarrow Open\text{-}result$

$rndfile\text{-}check\text{-}file(name, flags) \triangleq$
 407 if $is\text{-}legal\text{-}filename(name)$
 then if $OLD \in flags$
 408 then if $is\text{-}existing\text{-}file(name)$
 then OPENED
 else NO-SUCH-FILE
 408 else if $is\text{-}existing\text{-}file(name)$
 then FILE-EXISTS
 else OPENED
 else WRONG-NAME-FORMAT

TO DO — Extend definition to handle other problems in `DevConsts.OpenResultEnum`

values

```
rndfile-ops = {LOOK  $\mapsto$  rndfile-look,  
                SKIP  $\mapsto$  rndfile-skip,  
                SKIPLOOK  $\mapsto$  rndfile-skiplook,  
                WRITELN  $\mapsto$  rndfile-writeln,  
                WRITETEXT  $\mapsto$  rndfile-writetext,  
                WRITEBIN  $\mapsto$  rndfile-writebin,  
                READTEXT  $\mapsto$  rndfile-readtext,  
                READBIN  $\mapsto$  rndfile-readbin,  
                FLUSH  $\mapsto$  rndfile-flush,  
                RESET  $\mapsto$  rndfile-reset,  
                FREE  $\mapsto$  rndfile-free};
```

```
rndfile-flags = {READ, WRITE, OLD, TEXT, BINARY}
```

The procedure **OpenOld**

In a call of **OpenOld**, inclusion of the **old** flag in the parameter **flags** shall be implied and a destination of the given name shall already exist for the open to succeed.

If the **write** flag is not included in the request, inclusion of the **read** flag shall be implied.

If the call is successful, the parameter **cid** shall identify a channel open to a random access file of the given name and the value **opened** shall be assigned to the parameter **res**.

If a channel cannot be opened as required, the value of the parameter **res** shall indicate the reason and **cid** shall identify the bad channel.

NOTE — Distinct modes in combination with **text** and/or **binary** are given by the following equivalent sets of flags:

read from an existing file:	read	old	read+old
write to an existing file:	write	write+old	
read/write an existing file:	read+write	read+write+old	

OpenOld : $Char^* \times * \text{-setCharacteristic} \xrightarrow{o} (Chan\text{-}id \times Open\text{-}result)$

OpenOld (*name*, *flags*) \triangleq

```
418 let res1 = rndfile-check-file (name, flags  $\cup$  {OLD}) in  
    if res1  $\neq$  OPENED  
    then return mk- (badchan, res1)  
418 else let mk- (oflags, res2) = rndfile-merge-flags (flags, mk- (if WRITE  $\notin$  flags then {READ,OLD} else {OLD})) in  
    if res2  $\neq$  OPENED  
    then return mk- (badchan, res2)  
std else let oldops = rndfile-ops  $\cup$  {NAME  $\mapsto$   $\lambda x \cdot front (name, x)$ } in  
440   def mk- (chid, did) = make-a-channel (RNDFILE, oflags, oldops);  
    if chid = badchan  
    then return mk- (badchan, OUT-OF-CHANNELS)  
    else ( || write-position (did) := 1,  
443           files (did) := read-file (name);  
           return mk- (chid, OPENED) )  
pre  $\neg \exists chid : Chan\text{-}id \cdot$   
    let did = devices (chid) in  
419    supplied-name (did) = name
```

```

supplied-name : Dev-id → Char*
supplied-name (did)  $\triangleq$ 
  let nameop = devtable (did).name in
420 let n = mins ({ n : ℕ | nameop (n + 1) = nameop (n) }) in
420 nameop (did, n)

```

The procedure **OpenClean**

In a call of **OpenClean**, inclusion of the **write** flag in the parameter **flags** shall be implied.
A destination of the given name shall not already exist unless the flag **old** is given.

If the call is successful, the parameter **cid** shall identify a channel open to a random access file of the given name and the value **opened** shall be assigned to the parameter **res**.
The file shall be of zero length.

If a channel cannot be opened as required, the value of the parameter **res** shall indicate the reason and **cid** shall identify the bad channel.

NOTE — Distinct modes in combination with **text** and/or **binary** are given by the following equivalent sets of flags:

write to a new file:	write	
write to a new file or a truncated existing file:	old	write+old
write to a new file, read permission is needed:	read	write+read
write to a new file or a truncated existing file, read permission is needed:	old+read	write+old+read

$OpenClean : Char^* \times Characteristic\text{-}set \xrightarrow{o} (Chan\text{-}id \times Open\text{-}result)$

```

OpenClean (name, flags)  $\triangleq$ 
418 let res1 = rndfile-check-file (name, flags) in
  if res1 ≠ OPENED
  then return mk- (badchan, res1)
418 else let mk- (cflags, res2) = rndfile-merge-flags (flags, {WRITE}) in
  if res2 ≠ OPENED
  then return mk- (badchan, res2)
std else let cleanops = rndfile-ops ∪ {NAME ↦ λ x . front (name, x)} in
440 def mk- (chid, did) = make-a-channel (RNDFILE, cflags, cleanops) ;
  if chid = badchan
  then return mk- (badchan, OUT-OF-CHANNELS)
  else ( || write-position (did) := 1,
         files (did) := { } ;
        return mk- (chid, OPENED) )
pre  $\neg \exists chid : Chan\text{-}id \cdot$ 
  let did = devices (chid) in
419 supplied-name (did) = name

```

The procedure **IsRndFile**

A call of **IsRndFile** shall return **TRUE** if the channel is open to a random access file and **FALSE** otherwise.

$IsRndFile : Chan\text{-}id \xrightarrow{o} \mathbb{B}$

```

IsRndFile (chid)  $\triangleq$ 
  if chid ∈ dom channels
  then let mk-Chan (devid, -) = channels (chid) in
    return devid = RNDFILE
  else return false

```

The procedure **RndFileException**

A call of **RndFileException** shall return **rndFileNoException** if there is no exception in the current context, **notRndFileException** if there is an exception raised in the current context from a source other than **RndFile**, or **posRange** if that exception is raised in the current context.

The procedure **StartPos**

A call of **StartPos** shall return the position of the start of the file.

$StartPos : Chan-id \xrightarrow{o} Filepos$

$StartPos(chid) \triangleq$

```
442 def did = device(chid);  
443 (vet(did, RNDFILE, { }));  
407 return pos-to-filepos(chid, 1) )
```

The procedure **CurrentPos**

A call of **CurrentPos** shall return the current read/write position in the file.

$CurrentPos : Chan-id \xrightarrow{o} Filepos$

$CurrentPos(chid) \triangleq$

```
442 def did = device(chid);  
443 (vet(did, RNDFILE, { }));  
407 return pos-to-filepos(chid, write-position(did)) )
```

The procedure **EndPos**

A call of **EndPos** shall return the first position in the file at or after which no data have been written.

$EndPos : Chan-id \xrightarrow{o} Filepos$

$EndPos(chid) \triangleq$

```
442 def did = device(chid);  
443 (vet(did, RNDFILE, { }));  
    let end = card dom files(did) in  
407 return pos-to-filepos(chid, end + 1) )
```

The procedure **NewPos**

A call of **NewPos** shall return the read/write position **chunks** \times **chunkSize** places relative to the position given by the value of the parameter **from** or shall raise the exception **posRange** if the required position cannot be represented as a value of type **FilePos**.

$NewPos : Chan-id \times \mathbb{Z} \times \mathbb{N} \times Filepos \xrightarrow{o} Filepos$

$NewPos(chid, chunks, chunksize, from) \triangleq$

```
442 def did = device(chid);  
443 (vet(did, RNDFILE, { }));  
407 let opos = filepos-to-pos(chid, from) in  
    let npos = opos + (chunks  $\times$  chunksize) in  
407 def nfpos = pos-to-filepos(chid, npos);  
    if npos  $\leq$  0  
306 then mandatory-exception(POSRANGE)  
    else return nfpos )
```

The procedure **SetPos**

A call of **SetPos** shall set the read/write position to the position given by the value of the parameter **pos**.

If this position is beyond the value returned by a call of **EndPos**, '**read** <= **IOChannel.Flags()**' shall be **FALSE** and a call of an input operation shall raise the exception **notAvailable** — the value of '**write** <= **IOChannel.Flags()**' shall correspond to the availability of output operations as defined for the implementation.

NOTE — Setting the read/write position beyond the value returned by **EndPos** does not itself affect the size of the file.

$SetPos : Chan-id \times Filepos \xrightarrow{o} ()$

$SetPos(chid, fpos) \triangleq$

```
442  def did = device(chid) ;
443  (vet(did, RNDFILE, { } ) ;
    let mk-Device(props, -, -, -) = devices(did) in
407  let pos = filepos-to-pos(chid, fpos) in
    (if pos > card dom write-position(did) ∧ READ ∈ props
     then devices(did) := μ(devices(did), properties ↦ props − READ)
     else skip;
     write-position(did) := pos))
```

The procedure **Close**

In a call of **Close**, if the channel identified by the parameter **cid** is not open to a random access file, the exception **wrongDevice** shall be raised. Otherwise, the channel shall be closed and the value identifying the bad channel shall be assigned to **cid**.

$Close : Chan-id \xrightarrow{o} Chan-id$

$Close(chid) \triangleq$

```
442  def did = device(chid) ;
425  (rndfile-free(did) ;
    channels := {chid} ⋈ channels;
    devices := {did} ⋈ devices;
    devtable := {did} ⋈ devtable;
    return badchan )
```

9.2.4.5.3 The Rndfile device procedures

TO DO — Separate the device procedures into those which must be provided and those which may raise the **NotAvailable** exception.

$rndfile-look : Inop$

$rndfile-look(did) \triangleq$

```
443  (vet(did, RNDFILE, {READ, TEXT}) ;
    let wpos = write-position(did) in
    let file = files(did) in
    if wpos > card dom file
435  then (set-result(did, END-OF-INPUT) ;
         return mk-(nil, END-OF-INPUT) )
??  else let ins = map-to-seq(file) in
std  let s = rest(ins, wpos) in
443  def choice = an-item(s) ;
    if choice = END-LINE
435  then (set-result(did, END-OF-LINE) ;
         return mk-(nil, END-OF-LINE) )
```

```

435     else (set-result(did, ALL-RIGHT) ;
           return mk- (choice, ALL-RIGHT) ))
pre did ∈ dom devices

rndfile-skip : Op
rndfile-skip (did)  $\triangle$ 
443  (vet(did, RNDFILE, {READ, TEXT}) ;
    let wpos = write-position (did) in
    if wpos > card dom files (did)
306  then mandatory-exception(SKIP-AT-END)
??  else let ins = map-to-seq (file) in
std   let s = rest (ins, wpos) in
443   def choice = an-item(s) ;
      (write-position (did) := wpos + len choice ;
435   set-result(did, ALL-RIGHT) ))
pre did ∈ dom devices

rndfile-skiplook : Inop
rndfile-skiplook (did)  $\triangle$ 
423  rndfile-skip(did) ;
422  rndfile-look(did)
pre did ∈ dom devices

rndfile-readtext : Insop
rndfile-readtext (did, maxchars)  $\triangle$ 
443  (vet(did, RNDFILE, {READ, TEXT}) ;
    let wpos = write-position (did) in
    let file = files (did) in
    if wpos ≥ card dom file
435  then (set-result(did, END-OF-INPUT) ;
        return mk- (nil, 0) )
??  else let ins = map-to-seq (file) in
std   let s = rest (ins, wpos) in
443   def mk- (ans, nchars) = chars-up-to(s, maxchars) ;
      let amount = len ans in
435   (set-result(did, ALL-RIGHT) ;
      write-position (did) := wpos + amount ;
      return mk- (ans, nchars) ))
pre did ∈ dom devices

rndfile-writeln : Op
rndfile-writeln (did)  $\triangle$ 
443  (vet(did, RNDFILE, {WRITE, TEXT}) ;
    let wpos = write-position (did) in
    let file = files (did) in
407  let bos = emit (END-LINE) in
445  let os = seq-to-map (wpos, bos) in
    let upf = file † os in
445  let newf = pad (upf) in
    (files (did) := newf ;
     write-position (did) := wpos + len bos ;
    ))
pre did ∈ dom devices

```



```

rndfile-writetext : Outsop
rndfile-writetext (did, outs)  $\triangle$ 
443 ( vet(did, RNDFILE, {WRITE, TEXT}) ;
    let wpos = write-position (did) in
    let file = files (did) in
407 let bos = emit (outs) in
445 let os = seq-to-map (wpos, bos) in
    let upf = file  $\dagger$  os in
445 let newf = pad (upf) in
    (files (did) := newf;
     write-position (did) := wpos + len bos;
    ))
pre did  $\in$  dom devices

```

```

rndfile-readbin : Insop
rndfile-readbin (did, maxlocs)  $\triangle$ 
443 ( vet(did, RNDFILE, {READ, BINARY}) ;
    let wpos = write-position (did) in
    let file = files (did) in
    if wpos  $\geq$  card dom file
435 then ( set-result(did, END-OF-INPUT) ;
         return mk-(nil, 0) )
?? else let ins = map-to-seq (file) in
std    let s = rest (ins, wpos) in
        let n = if len s < maxlocs then len s else maxlocs in
443    let ans = up-to (s, n) in
        let amount = len ans in
435    ( set-result(did, ALL-RIGHT) ;
      write-position (did) := wpos + amount;
      return mk-(ans, amount) ) )
pre did  $\in$  dom devices

```

```

rndfile-writebin : Outsop
rndfile-writebin (did, outs)  $\triangle$ 
443 ( vet(did, RNDFILE, {WRITE, BINARY}) ;
    let wpos = write-position (did) in
    let file = files (did) in
445 let os = seq-to-map (wpos, outs) in
    let upf = file  $\dagger$  os in
445 let newf = pad (upf) in
    (files (did) := newf;
     write-position (did) := wpos + len outs;
    ))
pre did  $\in$  dom devices

```

```

rndfile-flush : Op
rndfile-flush (did)  $\triangle$ 
443 ( vet(did, RNDFILE, {WRITE}) ;
    let file = files (did) in
?? let os = map-to-seq (file) in
419 let filename = supplied-name (did) in
443 write-file(filename, os) )
pre did  $\in$  dom devices

```

```

rndfile-reset : Op
rndfile-reset(did)  $\triangleq$ 
424 (rndfile-flush(did);
      write-position(did) := 1)
pre did  $\in$  dom devices

rndfile-free : Op
rndfile-free(did)  $\triangleq$ 
443 (vet(did, RNDFILE, {}));
424 rndfile-flush(did);
      || write-position := {did}  $\triangleleft$  write-position;
      files := {did}  $\triangleleft$  files)
pre did  $\in$  dom devices

```

9.2.4.6 The module Terminal

The module **Terminal** provides access to an interactive terminal.

9.2.4.6.1 The definition module of Terminal

```

DEFINITION MODULE Terminal;

(* Channels opened by this module are connected to a single terminal device;
typed characters are distributed between channels according to the sequence
of read requests. *)

IMPORT
  IOChannel, DevConsts;
FROM DevConsts IMPORT
  FlagEnum;

TYPE
  ChanId = IOChannel.ChanId;
  FlagSet = DevConsts.FlagSet;
  OpenResultEnum = DevConsts.OpenResultEnum;

(* Accepted singleton values of FlagSet: *)

CONST
  read = FlagSet{readFlag};
  (* input operations are requested/available *)
  write = FlagSet{writeFlag};
  (* output operations are requested/available *)
  text = FlagSet{textFlag};
  (* text operations are requested/available *)
  binary = FlagSet{binaryFlag};
  (* binary operations are requested/available *)
  echo = FlagSet{echoFlag};
  (* echoing by interactive device on reading of characters from input
  stream requested/applies *)

(* In line mode, items are echoed before being added to the input stream
and are added a line at a time.
In single character mode, items are added to the input stream one at a time

```

and are echoed as they are removed from the input stream by a read operation *)

```
PROCEDURE Open(VAR cid: ChanId; flags: FlagSet; VAR res: OpenResultEnum);
  (* Without the binary flag, text is implied.
  Without the echo flag, line mode is requested, otherwise single character
  mode is requested.
  If successful,
  assigns to the parameter cid the identity of a channel open to the terminal
  as requested and assigns the value opened to the parameter res.
  Otherwise, the value of the parameter res indicates the reason for failure
  and cid identifies the bad channel. *)

PROCEDURE IsTerminal(cid: ChanId): BOOLEAN;
  (* Tests if the channel is open to the terminal *)

PROCEDURE Close(VAR cid: ChanId);
  (* If the channel is not open to the terminal, the exception
  wrongDevice is raised.
  Otherwise, the channel is closed and the value identifying the bad channel
  is assigned to the parameter cid *)

END Terminal.
```

9.2.4.6.2 The dynamic semantics of Terminal

Channels connected to the terminal device shall be opened in line mode or in single-character mode. In line mode, items shall be echoed before being added to the input stream and shall be added a line at a time.

In single character mode, items shall be added to the input stream as they are typed and shall be echoed as they are removed from the input stream by a text read device operation — provided they have not already been echoed.

Typed characters shall be distributed between multiple channels according to the sequence of read requests.

NOTE — If all the channels open to the terminal are open in line mode, the terminal device operates exclusively in line mode. Similarly, if all the channels open to the terminal are open in single-character mode, the terminal device operates exclusively in single-character mode. Echoing only occurs on reading from a channel in single-character mode, and not on looking or skipping, in order to allow interactive input routines to suppress the echoing of unwanted or unexpected characters.

If an implementation allows it, there might be one or more channels open in line mode and one or more channels open in single-character mode. In that case, all echoing needs to be postponed until the treatment of characters can be determined according to the sequence of calls of input operations.

This behaviour allows programs that use the terminal in different modes to be written in a modular fashion there being no need to explicitly save and restore the state of the terminal device.

In a request to open a channel to the terminal device, the flags **read**, **write**, **text**, **binary** and **echo** shall apply. If **binary** is not included in the request parameter **flags**, inclusion of **text** shall be implied. If the **read** flag is not included in the request, inclusion of the **write** flag shall be implied.

The procedure Open

In a call of **Open**, if the **echo** flag is included in the request, single-character mode shall be available for the open to succeed. Without the **echo** flag, line mode shall be available on the opened channel.

If the call is successful, the parameter **cid** shall identify a channel open to the terminal in the requested mode and the value **opened** shall be assigned to the parameter **res**.

If a channel cannot be opened as required, the value of the parameter **res** shall indicate the reason and **cid** shall

identify the bad channel.

The procedure **IsTerminal**

A call of **IsTerminal** shall return **TRUE** if the channel is open to the terminal device and **FALSE** otherwise.

The procedure **Close**

In a call of **Close**, if the channel identified by the parameter **cid** is not open to the terminal device, the exception **wrongDevice** shall be raised. Otherwise, the channel shall be closed and the value identifying the bad channel shall be assigned to **cid**.

9.2.4.7 The module ProgramArgs

The module **ProgramArgs** provides a channel for which input is taken from any arguments given to the program.

9.2.4.7.1 The definition module of ProgramArgs

```
DEFINITION MODULE ProgramArgs;

IMPORT
  IOChannel;

TYPE
  ChanId = IOChannel.ChanId;

(* Initially, and after a reset operation on the channel, if there are
no program arguments a call of IsArg returns FALSE, otherwise
input is taken from the first argument to the program, as defined for
the implementation. *)

PROCEDURE ArgChan(): ChanId;
  (* The returned value identifies a channel for reading program arguments *)

PROCEDURE ArgGiven(): BOOLEAN;
  (* Tests if there is a current argument to read from.
  If not, read < IOChannel.Flags() will be FALSE and attempting to read from
  the argument channel will cause the exception notAvailable to be raised. *)

PROCEDURE NextArg;
  (* If there is another argument, causes subsequent input from the
  argument device to come from the start of the next argument.
  Otherwise there is no argument to read from - a call of ArgGiven will
  return FALSE *)

END ProgramArgs.
```

9.2.4.7.2 The dynamic semantics of ProgramArgs

The procedure **ArgChan**

A call of **ArgChan** shall return a value identifying a channel from which the implementation-defined program arguments may be read.

The procedure **ArgGiven**

A call of **ArgGiven** shall return **TRUE** if there is a current argument to read from and **FALSE** otherwise.

If there is no current argument, '**read < IOChannel.Flags()**' shall be **FALSE** and attempting to read from the argument channel shall cause the exception **notAvailable** to be raised.

The procedure **NextArg**

After a call of **NextArg**, if there is another argument, subsequent input shall be taken from the start of that argument, otherwise, a call of **ArgGiven** shall return **FALSE**.

NOTE — Provision of **NextArg** allows arguments that contain spaces or line marks.

9.2.5 The Interface to Channels for Device-Independent Operations

There are two interfaces to the actual operations provided by a device module for a particular channel. Device-dependent operations, which include opening and closing a channel, are defined in the device definition module. Access to operations provided in a device-independent manner is allowed by the module `IOChannel`.

9.2.5.1 The module `IOChannel`

The module `IOChannel` defines the type `ChanId` that is used to identify channels. The module allows device-independent access to operations that are provided by the device to which the channel is open.

9.2.5.1.1 The definition module of `IOChannel`

```
DEFINITION MODULE IOChannel;

(* Types and procedures forming the interface to channels for
device-independent data transfer modules *)

FROM IOConsts IMPORT
    ReadResultEnum;
FROM DevConsts IMPORT
    FlagSet;
FROM SYSTEM IMPORT
    ADDRESS;

TYPE
    ChanId;
    (* Values of this type are used to identify channels *)

(* There is one pre-defined value identifying a bad channel on which no
data transfer operations are available. It is to be used to initialize
variables of type ChanId: *)

PROCEDURE BadChan(): ChanId;
    (* Returns the value identifying the bad channel *)

(* For each of the following operations, if the device supports the
operation on the channel, the behaviour of the procedure conforms with the
description below. The full behaviour is defined for each device module.
If the device does not support the operation on the channel, the behaviour
of the procedure is to raise the exception notAvailable. *)

(* Text operations - these perform any required translation between
the internal and external representation of text. *)

PROCEDURE Look(cid: ChanId; VAR ch: CHAR; VAR res: ReadResultEnum);
    (* If there is a character as the next item in the given input stream,
    assigns its value to the parameter ch without removing it from the stream.
    Otherwise, the value of the parameter ch is not defined.
    The parameter res, and the stored read result, is set to the value
    allRight, endOfLine, or endOfInput. *)

PROCEDURE Skip(cid: ChanId);
    (* If the input has ended, the exception skipAtEnd is raised,
```

```

otherwise, the next character or line mark in the input is removed
and the stored read result is set to the value allRight. *)

PROCEDURE SkipLook(cid: ChanId; VAR ch: CHAR; VAR res: ReadResultEnum);
(* If the stream has ended, the exception skipAtEnd is raised,
otherwise, the next character or line mark in the input is removed.
If there is a character as the next item in the given input stream,
assigns its value to the parameter ch without removing it from the stream.
Otherwise, the value of the parameter ch is not defined.
The parameter res, and the stored read result, is set to the value
allRight, endOfLine, or endOfInput. *)

PROCEDURE TextRead(
cid: ChanId; to: ADDRESS; maxChars: CARDINAL; VAR charsRead: CARDINAL
);
(* Reads at most maxChars characters from the current line and assigns
corresponding values to successive locations, starting at the address
given by the parameter to, and continuing at increments corresponding
to the address difference between successive components of an ARRAY OF CHAR.
The number of characters read is assigned to the parameter charsRead.
The read result is set to the value allRight, endOfLine, or endOfInput. *)

PROCEDURE TextWrite(cid: ChanId; from: ADDRESS; chars: CARDINAL);
(* Writes a number of characters given by the value of the parameter chars,
starting at the address given by the parameter from and continuing at
increments corresponding to the address difference between successive
components of an array of CHAR. *)

PROCEDURE WriteLn(cid: ChanId);
(* Writes a line mark over the channel *)

(* Binary operations: *)

PROCEDURE BinRead(
cid: ChanId; to: ADDRESS; maxLocs: CARDINAL; VAR locsRead: CARDINAL
);
(* Reads at most maxLocs items and assigns corresponding values to
successive locations starting at the address given by the parameter to.
The number of items read is assigned to the parameter locsRead.
The read result is set to the value allRight, or endOfInput *)

PROCEDURE BinWrite(cid: ChanId; from: ADDRESS; locs: CARDINAL);
(* Writes a number of items given by the value of the parameter locs from
successive locations starting at the address given by the parameter from. *)

(* Common operations *)

PROCEDURE GetName(cid: ChanId; VAR s: ARRAY OF CHAR);
(* Copies to the parameter s a name associated with the channel,
possibly truncated depending on the capacity of s *)

PROCEDURE Reset(cid: ChanId);
(* Reset to a state defined by the device module *)

PROCEDURE Flush(cid: ChanId);
(* Flush any data buffered by the device module out to the destination *)

```

```

(* Access to read results *)

PROCEDURE SetReadResult(cid: ChanId; res: ReadResultEnum);
  (* Sets the read result value for the channel to the value res *)

PROCEDURE ReadResult(cid: ChanId): ReadResultEnum;
  (* Returns the stored read result value for the channel *)
  (* (This is initially the value notKnown) *)

(* Users can discover which flags actually apply to a channel. *)

PROCEDURE Flags(cid: ChanId): FlagSet;
  (* Returns the set of flags that apply to the given channel *)

(* The following exceptions are defined for this module and its clients *)

TYPE
  ChanExceptionEnum =
    (ChanNoException,
      (* there is no exception in this context *)
    notChanException,
      (* there is an exception in this context, but from another source *)
    wrongDevice,
      (* device specific operation on wrong device *)
    notAvailable,
      (* operation attempted that is not available on that channel *)
    skipAtEnd,
      (* attempt to skip data from a stream that has ended *)
    softDeviceError,
      (* device specific recoverable error *)
    hardDeviceError,
      (* device specific non-recoverable error *)
    textParseError,
      (* input data does not correspond to a character or line mark -
        optional detection *)
    notAChannel,
      (* given value does not identify a channel - optional detection *)
    );

PROCEDURE ChanException(): ChanExceptionEnum;
  (* Returns the ChanException value for the current context *)

(* When a device procedure detects a device error, it raises the exception
softDeviceError or hardDeviceError. If these exceptions are handled,
the following procedure may be used to discover an implementation-defined
error number for the channel. *)

TYPE
  DeviceErrNum = INTEGER;

PROCEDURE DeviceError(cid: ChanId): DeviceErrNum;
  (* If a device error exception has been raised, returns the error
number stored by the device module. *)

```


END IOChannel.

9.2.5.1.2 The dynamic semantics of IOChannel

The **bad** channel shall be a channel on which no data transfer operations are available. Enquiries on the bad channel shall indicate that no operations are available.

NOTE — The identity of the bad channel is to be used to initialize variables of type **ChanId**.

The procedure **BadChan**

The value returned by a call of **BadChan** shall identify the bad channel.

BadChan : \rightarrow *Chan-id*

BadChan () \triangleq
badchan

Text operations

For each of the following operations, if the associated device supports the operation on the channel, the behaviour of the procedure conforms with the given description. The full behaviour is defined for each device module. If the device does not support the operation on the channel, the behaviour of the procedure is to raise the exception **notAvailable**.

A text stream is a possibly empty sequence of items where each item is either a character or a line mark.

The text operations provided by a device module shall perform any necessary translation between the internal representation, as a sequence of characters and line marks, and the external representation used by the source or destination. The interpretation of control characters shall be implementation-defined. The exception **textParseError** may be raised if input data does not correspond to a character or line mark.

NOTE — This may involve, for example, translation to and from escape sequences used in a coded character set, mapping between the external and internal representation of lines, or the interpretation of format effectors.

The procedure **Look**

On a call of **Look**, if there is a character as the next item in the input stream, the corresponding value shall be assigned to the parameter **ch** without removing the character from the stream. Otherwise, the value of the parameter **ch** is not defined.

The parameter **res**, and the stored read result, shall be set to the value:

allRight if a character is seen next;
endOfLine if no character is seen, the next item being a line mark;
endOfInput if no character is seen, the input having ended.

Look : *Chan-id* \xrightarrow{o} (*[Char]* \times *Read-result*)

Look (*chid*) \triangleq

```
442 def did = device(chid) ;  
443 let lop = device-op(did, LOOK) in  
432 lop(did)
```

The procedure **Skip**

On a call of **Skip**, if the input stream has ended, the exception **skipAtEnd** shall be raised. Otherwise, the next character or line mark in the input shall be removed and the stored read result shall be set to the value **allRight**.

$Skip : Chan-id \xrightarrow{o}$

$Skip(chid) \triangleq$

```

442 def did = device(chid);
443 let skop = device-op(did, SKIP) in
444   skop(did)

```

The procedure SkipLook

On a call of **SkipLook**, if the input stream has ended, the exception **skipAtEnd** shall be raised. Otherwise, the next character or line mark in the input shall be removed. If there is a character as the next item in the input stream, the corresponding value shall be assigned to the parameter **ch** without removing the character from the stream. Otherwise, the value of the parameter **ch** is not defined.

The parameter **res**, and the stored read result, shall be set to the value:

allRight if a character is seen next;
endOfLine if no character is seen, the next item being a line mark;
endOfInput if no character is seen, the input having ended.

$SkipLook : Chan-id \xrightarrow{o} ([Char] \times Read-result)$

$SkipLook(chid) \triangleq$

```

442 def did = device(chid);
443 let slop = device-op(did, SKIPLOOK) in
444   slop(did)

```

The procedure TextRead

A call of **TextRead** shall read at most **maxChars** characters from the current line and assign corresponding values to successive locations, starting at the address given by the parameter **to**, and continuing at increments corresponding to the address difference between successive components of an array of component type **CHAR**.

The number of characters read shall be assigned to the parameter **charsRead**.

The read result shall be set to the value

allRight if '**maxChars** = **charsRead** = 0' or ('**maxChars** > 0' and '**charsRead** > 0');
endOfLine if '**maxChars** > 0' and '**charsRead** = 0', the next item being a line mark;
endOfInput if '**maxChars** > 0' and '**charsRead** = 0', the input having ended.

$TextRead : Chan-id \times \mathbb{N} \xrightarrow{o} (Char^* \times \mathbb{N})$

$TextRead(chid) \triangleq$

```

442 def did = device(chid);
443 let trop = device-op(did, TEXTREAD) in
444   def ans = trop(did, n);
   return ans, len ans

```

The procedure TextWrite

A call of **TextWrite** shall write a number of characters given by the value of the parameter **chars**, starting at the address given by the parameter **from** and continuing at increments corresponding to the address difference between successive components of an array of component type **CHAR**.

$TextWrite : Chan-id \times Char^* \times \mathbb{N} \xrightarrow{o} ()$

$TextWrite(chid, outs, n) \triangleq$

```

442 def did = device(chid);
443 let twop = device-op(did, TEXTWRITE) in
444   twop(did, outs, n)

```

The procedure **WriteLn**

A call of **WriteLn** shall write a line mark to the output stream.

$IOChannel\text{-}WriteLn : Chan\text{-}id \xrightarrow{o} ()$

$IOChannel\text{-}WriteLn(chid) \triangle$

```
442  def did = device(chid) ;
443  let wlop = device-op(did, WRITELN) in
434  wlop(did)
```

Binary operations

For each of the following operations, if the associated device supports the operation on the channel, the behaviour of the procedure conforms with the given description. The full behaviour is defined for each device module. If the device does not support the operation on the channel, the behaviour of the procedure is to raise the exception **notAvailable**.

The binary operations provided by a device module transfer data location by location with no translation or interpretation.

The procedure **BinRead**

A call of **BinRead** shall read at most **maxLocs** items and assign corresponding values to successive locations starting at the address given by the parameter **to**.

The number of items read shall be assigned to the parameter **locsRead**. The read result shall be set to the value

allRight if ('maxLocs = locsRead = 0') or ('maxLocs > 0 and 'locsRead > 0')

endOfInput if 'maxLocs' > 0' and 'locsRead = 0'

$BinRead : Chan\text{-}id \times \mathbb{N} \xrightarrow{o} (Loc^* \times \mathbb{N})$

$BinRead(chid, outs, n) \triangle$

```
442  def did = device(chid) ;
443  let brop = device-op(did, BINWRITE) in
434  def ans = brop(did, outs, n) ;
    return mk-(ans, len ans)
```

The procedure **BinWrite**

A call of **BinWrite** shall write a number of items given by the value of the parameter **locs** from successive locations starting at the address given by the parameter **from**.

$BinWrite : Chan\text{-}id \times Loc^* \times \mathbb{N} \xrightarrow{o} ()$

$BinWrite(chid, outs, n) \triangle$

```
442  def did = device(chid) ;
443  let bwop = device-op(did, BINWRITE) in
434  wlop(did, outs, n)
```

Common operations

For each of the following operations the behaviour of the procedure conforms with the given description. The full behaviour is defined for each device module.

The procedure **GetName**

A call of **GetName** shall copy to the parameter **s** as a string value a name associated with the channel, possibly truncated depending on the capacity of **s**.

$Getname : Chan-id \times \mathbb{N}_1 \xrightarrow{o} Char^*$

$Getname(chid, size) \triangleq$

```

442   def did = device(chid);
443   let namop = device-op(did, NAME) in
435   namop(did, size)

```

The procedure **Reset**

A call of **Reset** shall reset the device associated with the channel to a state defined by the device module.

$Reset : Chan-id \xrightarrow{o} ()$

$Reset(chid) \triangleq$

```

442   def did = device(chid);
443   let rsop = device-op(did, RESET) in
435   rsop(did)

```

The procedure **Flush**

A call of **Flush** shall flush any data buffered by the device module out to the destination.

$Flush : Chan-id \xrightarrow{o} ()$

$Flush(chid) \triangleq$

```

442   def did = device(chid);
443   let flsop = device-op(did, FLUSH) in
435   flsop(did)

```

Access to read results

Higher-level data input procedures, for units such as strings and numerals, may alter the read result for a channel to indicate success or a particular kind of failure of interpretation. The result can be recovered, if necessary, by the caller of the data input procedure.

The procedure **SetReadResult**

A call of **SetReadResult** shall set the read result value for the channel to the value given by the parameter **res**.

$SetReadResult : Chan-id \times Read-result \xrightarrow{o} ()$

$SetReadResult(chid, res) \triangleq$

```

442   def did = device(chid);
435   set-result(did, res)

```

$set-result : Dev-id \times Read-result \xrightarrow{o} ()$

$set-result(did, res) \triangleq$

```

  let dtbl = devtable(did) in
  devtable(did) :=  $\mu$ (dtbl, result  $\mapsto$  res)

```

The procedure **ReadResult**

A call of **ReadResult** shall return the stored read result value for the channel.

```

IOChannel-ReadResult : Chan-id  $\xrightarrow{o}$  Read-result
IOChannel-ReadResult (chid)  $\triangleq$ 
442   def did = device(chid) ;
436   result(did)

```

```

result : Dev-id  $\xrightarrow{o}$  Read-result
result (did)  $\triangleq$ 
  let mk-Device (-, -, -, -, res) = devtable (did) in
  return res

```

Channel enquiries

The procedure **Flags**

A call of **Flags** shall return the set of flags that currently apply to the given channel, as defined for the associated device.

```

Flags : Chan-id  $\xrightarrow{o}$  Characteristic-set
Flags (chid)  $\triangleq$ 
442   def did = device(chid) ;
      let mk-Device (-, props, -, -, -) = devtable (did) in
      return props

```

Exceptions and device errors

The procedure **ChanException**

A call of **ChanException** shall return **chanNoException** if there is no exception in the current context, **notChanException** if there is an exception raised in the current context from another source, or the value of **ChanExceptionEnum** corresponding to the raised exception.

The procedure **DeviceError**

If a device error exception has been raised during an operation on the identified channel, a call of **DeviceError** shall return the error number stored by the device module for that channel — otherwise the returned value is not defined.

NOTE — When a device procedure detects a device error, it raises the exception **softDeviceError** or **hardDeviceError**. If these exceptions are handled, the procedure **DeviceError** may be used to discover the implementation-defined error number stored by the device module for the channel that was in use when the device error occurred.

9.2.6 The Interface to Channels for New Device Modules

Additional device modules may be provided to allow the library to be used with other input sources and output destinations. These might include, for example, files opened with host-specific options or parameters or with host-specific behaviour, a windowing system, or a speech output device.

9.2.6.1 The module IOLink

The module `IOLink` allows new device modules to obtain channels and to install procedures that implement the device-independent operations on the channel.

9.2.6.1.1 The definition module of IOLink

```
DEFINITION MODULE IOLink;

(* Types and procedures giving the standard implementation of channels: *)

FROM IOChannel IMPORT
    ChanId, DeviceErrNum, ChanExceptionEnum;
FROM IOConsts IMPORT
    ReadResultEnum;
FROM DevConsts IMPORT
    FlagSet;
FROM SYSTEM IMPORT
    ADDRESS;

(* Devices need to identify themselves in order to allow a check to be made
that device-dependent operations are applied only for channels opened
to that device: *)

TYPE
    DeviceId;
    (* values of this type are used to identify new device modules and are
       normally obtained by them during their initialisation *)

PROCEDURE AllocateDeviceId(VAR did: DeviceId);
    (* Allocates a unique value of type DeviceId and assigns this
       value to the parameter did *)

(* a new device module open procedure obtains a channel by calling MakeChan *)

PROCEDURE MakeChan(did: DeviceId; VAR cid: ChanId);
    (* Attempts to make a new channel for the device module identified by did.
       If no more channels can be made, the identity of the bad channel is assigned
       to cid. Otherwise, the identity of a new channel is assigned to cid. *)

(* If a channel is allocated but the call of the device module open
procedure is not successful, and on a successful call of a device
module close procedure, the device module unmakes the channel and
returns the value identifying the bad channel to its client: *)

PROCEDURE UnMakeChan(did: DeviceId; VAR cid: ChanId);
    (* If the device module identified by the parameter did is not the module
       that made the channel identified by the parameter cid, the exception
```

wrongDevice is raised.

Otherwise, the channel is deallocated and the value identifying the bad channel is assigned to cid. *)

(* If the call of the device module open procedure is successful, the device module obtains a pointer to a device table for the channel, which will have been initialised by MakeChan. It then changes the fields of the device table to install its own values for the device data, supported operations, and flags, and returns to its client the identity of the channel. *)

TYPE

DeviceTablePtr = POINTER TO DeviceTable;

(* Values of this type are used to refer to device tables *)

(* Device modules supply procedures of the following types: *)

TYPE

LookProc = PROCEDURE(DeviceTablePtr, VAR CHAR, VAR ReadResultEnum);

SkipProc = PROCEDURE(DeviceTablePtr);

SkipLookProc = PROCEDURE(DeviceTablePtr, VAR CHAR, VAR ReadResultEnum);

TextReadProc = PROCEDURE(DeviceTablePtr, ADDRESS, CARDINAL, VAR CARDINAL);

TextWriteProc = PROCEDURE(DeviceTablePtr, ADDRESS, CARDINAL);

LnWriteProc = PROCEDURE(DeviceTablePtr);

BinReadProc = PROCEDURE(DeviceTablePtr, ADDRESS, CARDINAL, VAR CARDINAL);

BinWriteProc = PROCEDURE(DeviceTablePtr, ADDRESS, CARDINAL);

GetNameProc = PROCEDURE(DeviceTablePtr, VAR ARRAY OF CHAR);

ResetProc = PROCEDURE(DeviceTablePtr);

FlushProc = PROCEDURE(DeviceTablePtr);

FreeProc = PROCEDURE(DeviceTablePtr);

(* Carry out the operations involved in closing the corresponding channel, including flushing buffers, but do not unmake the channel.

This procedure is called for each open channel at program termination. *)

(* Device tables have:

a field in which the device module can store private data,

a field in which the value identifying the device module is stored,

a field in which the value identifying the channel is stored,

a field in which the read result is stored,

a field in which device error numbers are stored prior to the raising of a device error exception,

a field in which flags are stored indicating those which apply,

a field for each device procedure.

The fields are initialised by MakeChan to the values shown for the type: *)

TYPE

DeviceData = ADDRESS;

DeviceTable =

RECORD

cd: DeviceData;

(* the value NIL *)

did: DeviceId;

(* the value given in the call of MakeChan *)

cid: ChanId;

(* the identity of the channel *)

```

result: ReadResultEnum;
    (* the value notKnown *)
errNum: DeviceErrNum;
    (* undefined *)
flags: FlagSet;
    (* FlagSet{} *)
doLook: LookProc;
    (* raise exception notAvailable *)
doSkip: SkipProc;
    (* raise exception notAvailable *)
doSkipLook: SkipLookProc;
    (* raise exception notAvailable *)
doTextRead: TextReadProc;
    (* raise exception notAvailable *)
doTextWrite: TextWriteProc;
    (* raise exception notAvailable *)
doLnWrite: LnWriteProc;
    (* raise exception notAvailable *)
doBinRead: BinReadProc;
    (* raise exception notAvailable *)
doBinWrite: BinWriteProc;
    (* raise exception notAvailable *)
doGetName: GetNameProc;
    (* return the empty string *)
doReset: ResetProc;
    (* do nothing *)
doFlush: FlushProc;
    (* do nothing *)
doFree: FreeProc;
    (* do nothing *)
END;

```

(* The pointer to the device table for a channel is obtained using the following procedure: *)

```

PROCEDURE DeviceTablePtrValue(cid: ChanId; did: DeviceId): DeviceTablePtr;
    (* If the device module identified by the parameter did is not the module
       that made the channel identified by the parameter cid, the exception
       wrongDevice is raised.
       Otherwise, a pointer to the device table for the channel is returned. *)

```

(* A device module can ask if it opened a given channel. It does this to implement a corresponding enquiry function that is exported from the device module: *)

```

PROCEDURE IsDevice(cid: ChanId; did: DeviceId): BOOLEAN;
    (* Tests if the device module identified by the parameter did is the
       module that made the channel identified by the parameter cid. *)

```

(* Client modules may raise appropriate exceptions: *)

```

TYPE
    DevExceptionRange = [notAvailable .. textParseError];

```

```

PROCEDURE RAISEdevException(

```



```

cid: ChanId; did: DeviceId; x: DevExceptionRange; s: ARRAY OF CHAR
);
(* If the device module identified by the parameter did is not the module
that made the channel identified by the parameter cid, the exception
wrongDevice is raised. Otherwise the given exception is raised and
the string value of the parameter s is included in the exception message. *)

PROCEDURE IOException(): ChanExceptionEnum;
  (* Returns the ChanException value for the current context *)

(* On program termination, for all allocated channels, the device procedure
doFree is called. Device error exceptions are handled. The channels are
then deallocated. *)

END IOLink.

```

9.2.6.1.2 The dynamic semantics of IOLink

The procedure AllocateDeviceId

A call of **AllocateDeviceId** shall allocate a unique value of type **DeviceId** and assign this value to the parameter **did**.

$AllocateDeviceId : \xrightarrow{o} Dev\text{-}name$

$AllocateDeviceId () \triangleq$

```

  let dname : New-Dev-name be st  $\neg \exists d \in \text{dom } devtable \cdot devtable(d).id = dname$  in
  return dname

```

The procedure MakeChan

A call of **MakeChan** shall attempt to allocate a new channel. If no more channels can be allocated, the value identifying the bad channel shall be assigned to the parameter **cid**. Otherwise, the call shall assign a value identifying a new channel to **cid**.

$MakeChan : Dev\text{-}name \xrightarrow{o} Chan\text{-}id$

$MakeChan (dname) \triangleq$

```

440 def mk-(chid, did) = make-a-channel(dname, { }, { });
  return chid

```

values

$default\text{-}ops = \{op \mapsto unavailable \mid is\text{-}Device\text{-}op(op)\}$

$make\text{-}a\text{-}channel : Dev\text{-}name \times Characteristic\text{-}set \times (Operation \xrightarrow{m} Device\text{-}op) \xrightarrow{o} (Chan\text{-}id \times [Dev\text{-}id])$

$make\text{-}a\text{-}channel (dname, props, ops) \triangleq$

if $\text{card } \text{dom } channel \geq Implementation\text{-}defined\text{-}max\text{-}channels$

then return $mk\text{-}(badchan, nil)$

else let $chid \in Chan\text{-}id$ be st $chid \notin \text{dom } channel$ in

let $did : Dev\text{-}id$ be st $did \notin devices$ in

let $dev = mk\text{-}Device (dname, props, default\text{-}ops \upharpoonright ops, 0, UNKNOWN)$ in

let $chan = mk\text{-}Channel ()$ in

($\parallel channel := channel \cup \{chid \mapsto chan\},$

$devtable := devtable \cup \{did \mapsto dev\},$

$devices := devices \cup \{chid \mapsto did\};$

return $mk\text{-}(chid, did)$)

The procedure **UnMakeChan**

In a call of **UnMakeChan**, if the device module identified by the **did** is not the module that made the channel identified by **cid**, the exception **wrongDevice** shall be raised. Otherwise, the channel shall be deallocated and the value identifying the bad channel shall be assigned to **cid**.

The procedure **DeviceTablePtrValue**

In a call of **DeviceTablePtrValue**, if the device module identified by the **did** is not the module that made the channel identified by **cid**, the exception **wrongDevice** shall be raised. Otherwise, a pointer to the device table for the channel shall be returned.

The procedure **IsDevice**

A call of **IsDevice** shall return **TRUE** if the device module identified by **did** is the module that made the channel identified by **cid** and otherwise shall return **FALSE**.

$IsDevice : Chan-id \times Dev-name \rightarrow \mathbb{B}$

$IsDevice(chid, dname) \triangleq$
if $chid \in \text{dom } channels$
then let $dev = devices(chid)$ in
 return $dev.id = dname$
else return false

The procedure **RAISEdevException**

In a call of **RAISEdevException**, if the device module identified by the **did** is not the module that made the channel identified by **cid**, the exception **wrongDevice** shall be raised. Otherwise, the exception given by the paramter **x** shall be raised. The string value of the parameter **s** shall be included in the exception message.

The procedure **IOException**

A call of **IOException** shall return **chanNoException** if there is no exception in the current context, **notChanException** if there is an exception raised in the current context from another source, or the value of **ChanExceptionEnum** corresponding to the raised exception.

NOTE — A single value of **EXCEPTIONS.ExceptionSource** is used to identify the source of input/output library exceptions corresponding to **ChanExceptionEnum**. The procedure **IOException** is included so that this value need not be exported for **IOChannel.ChanException** to be provided in implementations that support **IOLink**.

9.2.6.1.3 A proforma for device module procedures

$device-Look : Dev-id \xrightarrow{o} ([Char] \times Read-result)$

$device-Look(did)$

pre $did \in \text{dom } devices$

post $device-Look = mk-(nil, \text{END-OF-INPUT}) \vee$

$device-Look = mk-(nil, \text{END-OF-LINE}) \vee$

407 $(device-Look = mk-(c, \text{ALL-RIGHT}) \wedge isa-Char(c))$

TO DO — Complete pre- and post-conditions for the other proforma device operations

$device-Skip : Dev-id \xrightarrow{o} ()$

$device-Skip(did) \triangleq$

is not yet defined

$device-SkipLook : Dev-id \xrightarrow{o} ([Char] \times Read-result)$

$device-SkipLook (did) \triangleq$
is not yet defined

$device-TextRead : Dev-id \times \mathbb{N} \xrightarrow{o} (Char^* \times \mathbb{N})$

$device-TextRead (did) \triangleq$
is not yet defined

$device-TextWrite : Dev-id \times Char^* \xrightarrow{o} ()$

$device-TextWrite (did) \triangleq$
is not yet defined

$device-WriteLn : Dev-id \xrightarrow{o} ()$

$device-WriteLn (did) \triangleq$
is not yet defined

$device-BinRead : Dev-id \times \mathbb{N} \xrightarrow{o} (Loc^* \times Read-result)$

$device-BinRead (did) \triangleq$
is not yet defined

$device-BinWrite : Dev-id \times Loc^* \xrightarrow{o} ()$

$device-BinWrite (did) \triangleq$
is not yet defined

$device-GetName : Dev-id \times \mathbb{N} \xrightarrow{o} Char^*$

$device-GetName (did) \triangleq$
is not yet defined

$device-Reset : Dev-id \xrightarrow{o} ()$

$device-Reset (did) \triangleq$
is not yet defined

$device-Flush : Dev-id \xrightarrow{o} ()$

$device-Flush (did) \triangleq$
is not yet defined

9.2.6.1.4 Module Initialisation

The module shall install a termination procedure which, for all allocated channels, calls the device procedure **doFree**. Device error exceptions shall be handled and acknowledged so that calls of all the **doFree** procedures are made and so that the calls are made in the state of normal execution. The channels shall then be deallocated.

9.2.6.2 Auxiliary definitions used by Device modules

$device : Chan-id \xrightarrow{o} Dev-id$

$device (chid) \triangleq$
if $chid \notin \text{dom } channels$
then *non-mandatory-exception*(NOT-A-CHANNEL)
else return $devices(chid)$

306

annotations	Return the device id corresponding to a channel id.
-------------	---

$$vet : Dev-id \times Dev-name \times \{Characteristic\} \xrightarrow{o} ()$$
$$vet (did, dname, props) \triangleq$$
$$\text{let } mk\text{-Device}(devname, properties, -, -) = dev \text{ in}$$

if $dname \neq devname$

306 then *mandatory-exception*(WRONG-DEVICE)

else if $props \subseteq properties$

then skip

```
306     else mandatory-exception(NOT-AVAILABLE)
```

```

pre  did ∈ dom devices

```

annotations	Check that the driver has created the channel and that it has the required properties
-------------	---

$$an-item : Loc^* \xrightarrow{o} [Char \mid \text{END-LINE}]$$
$$an-item(s) \triangleq$$

let $possible-items = \{front(s, i) \mid i \in \mathbb{N} \cdot is_Char(front(s, i)) \vee front(s, i) = \text{END-LINE}\}$ in

if *possible-items* = { }

```

306 then non-mandatory-exception(TEXT-PARSE-ERROR)

```

else let $choice : Char \mid \text{END-LINE}$ be st $\{choice\} \subseteq possible-items$ in

```

return choice

```

annotations	Check there is a character or line mark at the head of the input and return it.
-------------	---

$$chars-up-to : Loc^* \times \mathbb{N} \xrightarrow{o} (Char^* \times \mathbb{N})$$
$$chars\text{-}up\text{-}to\ (s, n) \triangleq$$

let $ans : Char^*$ be st $len\ ans < n \wedge$

$$\text{END-LINE} \notin \text{elems } ans \wedge$$
497 *isa-prefix* (*emit* (*ans*), *s*) in

```
if ans == []
```

```

306 then non-mandatory-exception(TEXT-PARSE-ERROR)

```

```
?? else return mk-(ans, length(ans))
```

annotations	Return up to n characters from the front of s .
-------------	---

$$up\text{-}to : X^* \times \mathbb{N} \rightarrow X^*$$
$$up-to(s, max) \triangleq$$

444 let $ans : X^*$ be st $isa_prefix(ans, s) \wedge len\ ans \leq max \wedge ans \neq []$ in

ans

annotations	Return a non empty subsequence up to length <i>max</i> from the front of <i>s</i> .
-------------	---

$$device-op : Dev-id \times Operation \rightarrow Device-op$$
$$device-op\ (did, oper) \triangleq$$

```
let dev = devtable (did) in
```

$$dev.op(oper)$$

annotations	Return the device procedure corresponding to the desired operation.
-------------	---

$$read_file : Char^* \xrightarrow{o} File$$

read-file (*name*) Read in a file

$$write_file : Char^* \times File \xrightarrow{o} ()$$

write-file (*name*, *f*) Write *f* to destination device

9.2.6.3 Auxiliary functions used in the definition of the IO library

functions

```

isa-space : Char* → ℬ
isa-space (s)  $\triangleq$ 
  elems s = {SPACE-CHARACTER}
  annotations      Return whether s consists entirely of space characters.
;

isa-prefix : Char* × Char* → ℬ
isa-prefix (s1, s2)  $\triangleq$ 
  ∃ i ∈ inds s2 · s1 = s2 (1, ..., i)
  annotations      Returns whether S1 is a prefix of S2
;

split : Char* × (Char → ℬ) → Char*
split (S, f)  $\triangleq$ 
  let first  $\curvearrowright$  rest = S be st
    (rest = [] ∧ ∀ x ∈ elems first · f(x)) ∨
    (¬ f(hd rest) ∧ ∀ x ∈ elems first · f(x)) in
  first
  annotations      Return the longest substring of S that satisfies f and starts at the beginning of S
;

no-leading-space : Char* → Char*
no-leading-space (S)
post ∃ x · x  $\curvearrowright$  no-leading-space = S ∧ isa-space (x)
  annotations      Return S stripped of leading white space.
;

isa-numeral : Char → ℬ
isa-numeral (c)  $\triangleq$ 
  c ∈ digits
  annotations      Is the character a numeral.
;

digit-value : Char → ℕ
digit-value (c)  $\triangleq$ 
  required-collation-map(c) − required-collation-map('0')
  annotations      Return the value of a numeral.
;

numeric-value : Char+ → ℕ
numeric-value (s)  $\triangleq$ 
  let c = hd s in
  444(digit-value (c) × 10 ↑ len tl s) + if tl s = [] then 0 else numeric-value (tl s)
  annotations      The numeric value of a string of numerals.
;

spaces : ℕ → Char*
spaces (n)
post let s ∈ Char* be st len s = n ∧ elems s = {SPACE} in
  spaces = s

```

```

    annotations      Return a sequence of  $n$  space characters.
;

seq-to-map :  $\mathbb{N} \times X^* \rightarrow (\mathbb{N} \xrightarrow{m} X)$ 
seq-to-map ( $n, S$ )  $\triangleq$ 
   $\{(x + n) \mapsto S(x) \mid x \in \text{inds } S\}$ ;

pad : File  $\rightarrow$  File
pad (file)  $\triangleq$ 
  let  $pad = \text{implementation-defined-padding-value}$  in
  let  $padding = \{n \mapsto pad \mid n \in \{0, \dots, \text{dom } file\}\}$  in
  padding  $\dagger$  file
    annotations      Fill any gaps in a file with implementation defined padding.

```

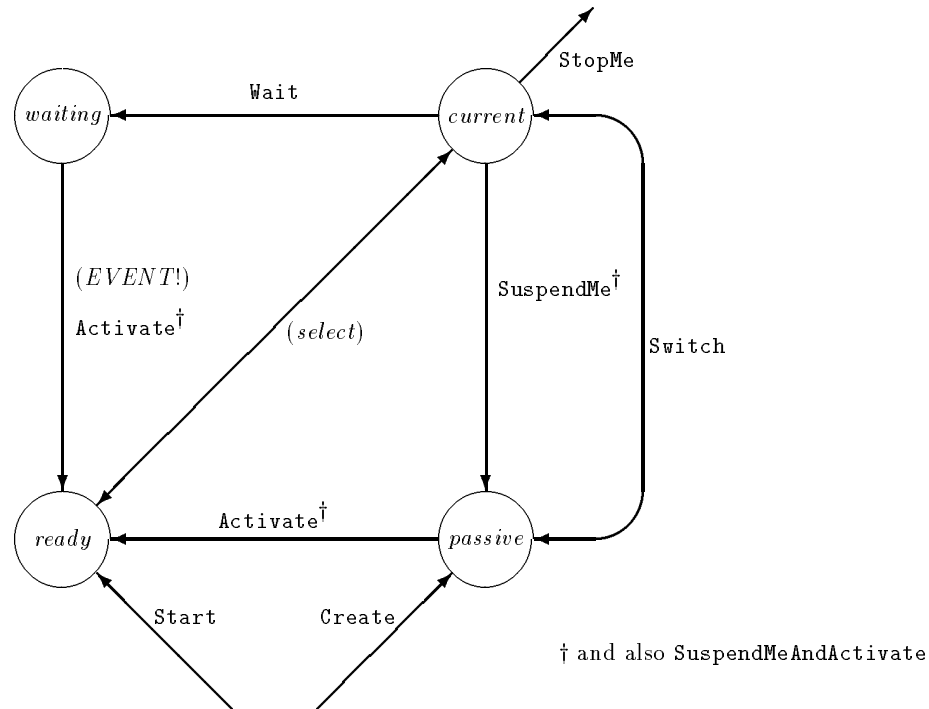
9.3 Concurrent Programming

9.3.1 The Processes Module

The Processes module allows the expression of concurrency within a Modula-2 program. In conjunction with the Semaphores module (which allows potentially parallel parts of the program to exclude each other in regions of interaction) it provides a basic set of facilities for use in concurrent programs.

NOTE — A concurrent program consists of a number of processes, each of which may potentially run in parallel with the others and is distinguishable from them. At any one time, a process may be in one of four states: It may be *ready*, that is, eligible to use the processor but not actually doing so: It may be *current* — using the processor: It may be *passive*, that is ineligible to use the processor until another process makes it eligible: Lastly, it may be *waiting* — ineligible to use the processor until the occurrence of one of the events for which it is waiting.

The relationships between the states together with the operations which lead to a change of state are depicted in the following figure. Operations which are in parentheses are not invocable directly by the program.



Two general styles of use are envisaged and both may be present in a single program. In the first (using `Switch`), there is a set of closely coupled processes which explicitly choose which of them shall run next and pass information between them as part of the choice. The intention is to provide a higher level coroutine-like facility. In the second style (using `Activate` and `SuspendMe`) the processes are written to be more independent of each other and the choice of which of them runs is left to the internal scheduler.

At all times, there must be at least one process which is using the processor or, if no process is eligible, there must be at least one process waiting on some external event.

It cannot be the case that there is a process which is eligible to use the processor which is more urgent than one of the processes currently doing so.

NOTE — A program which uses the Processes module should not make explicit use of coroutines (except, perhaps, in the implementation of the Processes module itself).

CHANGE — This module is not based on the Processes module described in *Programming in Modula-2*.

9.3.1.1 The definition module of Processes

```
DEFINITION MODULE Processes;
    (* This module allows concurrent algorithms to be      *)
    (* expressed using processes.                          *)
    (* A process is a unit of the program which has the    *)
    (* potential to run in parallel with other processes   *)
FROM    SYSTEM IMPORT ADDRESS;
TYPE    ProcessId;          (* Used to identify processes *)
        Parameter = ADDRESS; (* Used to pass data between processes *)
        Body      = PROC;
        Urgency   = INTEGER; (* A hint to the (internal) scheduler *)
        ProcException =      (* Exceptions raised by this module *)
        (ProcsNoException, notProcsException, passiveProgram, ProcessError);

PROCEDURE Create(
    ProcBody:   Body;
    ExtraSpace: CARDINAL;
    ProcUrg:    Urgency;
    ProcParams: Parameter;
    VAR Proc:   ProcessId
);
    (* This procedure creates a new process with the given *)
    (* parameters and urgency and with the supplied        *)
    (* procedure as its body. At least as much workspace   *)
    (* as indicated by ExtraSpace is allocated to it.      *)
    (* The identity of the new process is returned in Proc. *)
    (* The process will not run until activated.           *)

PROCEDURE Start(
    ProcBody:   Body;
    ExtraSpace: CARDINAL;
    ProcUrg:    Urgency;
    ProcParams: Parameter;
    VAR Proc:   ProcessId
);
    (* This procedure creates a new process (same as Create *)
    (* except that the process is able to run immediately) *)

PROCEDURE StopMe();
    (* This procedure terminates the calling process. The *)
    (* space used is recovered.                             *)

PROCEDURE SuspendMe();
    (* This procedure only returns when the calling process *)
    (* is activated by another process.                       *)

PROCEDURE Activate(Proc: ProcessId);
    (* This procedure allows another process (which must not *)
    (* be eligible to run) to run again.                      *)

PROCEDURE Switch(
    Proc: ProcessId;
    VAR Info: Parameter
);
    (* This procedure swaps from the calling process to the *)
    (* nominated process and passes a parameter. On return, *)
```



```

        (* Info will contain information from the process which *)
        (* chooses to switch to this one (or NIL if Activate or *)
        (* SuspendMeAndActivate is used instead of Switch).      *)

PROCEDURE SuspendMeAndActivate(Proc: ProcessId);
    (* Atomic sequence of SuspendMe() and Activate(Proc)      *)

PROCEDURE Attach(Source: CARDINAL);
    (* This procedure associates the calling process with a *)
    (* source of events                                     *)

PROCEDURE Detach(Source: CARDINAL);
    (* This procedure disassociates the calling process      *)
    (* from a source of events                                *)

PROCEDURE IsAttached(Source: CARDINAL) BOOLEAN;
    (* This procedure returns TRUE iff the source of events *)
    (* is associated with a process                           *)

PROCEDURE Handler(Source: CARDINAL) ProcessId;
    (* This procedure returns the identity of the process    *)
    (* which is associated with the source of events          *)

PROCEDURE Wait();
    (* This procedure will return when one of the associated*)
    (* sources has generated an event or when the calling    *)
    (* process is activated by another                        *)

PROCEDURE Me(): ProcessId;
    (* This procedure returns the identity of the process    *)

PROCEDURE MyParam(): Parameter;
    (* This procedure returns the value of the parameter     *)
    (* given when the calling process was created.           *)

PROCEDURE UrgencyOf(Proc: ProcessId): Urgency;
    (* This procedure returns the urgency of the process     *)
    (* given when the calling process was created.           *)

PROCEDURE ExceptionValue(): ProcException;
    (* This procedure returns whether an exception has been *)
    (* raised by this module, and if so, which one          *)

END Processes.

```

9.3.1.2 Dynamic semantics

Processes :: *ready* : PID-set
current : PID-set
passive : PID-set
waiting : PID-set
urgency : PID \xrightarrow{m} \mathbb{Z}
attached : Event-Source \xrightarrow{m} PID
params : PID \xrightarrow{m} Data
info : PID \xrightarrow{m} Data

$\text{inv } \text{mk-Processes}(R, C, P, W, U, A, -, -) \triangleq \text{is-pairwise-disjoint}(\{R, C, P, W\}) \wedge$
 $C = \{\} \Rightarrow R = \{\} \wedge$
 $C \cup W \neq \{\} \wedge$
 $\forall p \in C \cdot \forall q \in R \cdot U(p) \geq U(q) \wedge$
 $\text{rng } A \subseteq \text{dom } U = R \cup C \cup P \cup W$

TECHNICAL NOTE — *params* and *info* are used to model the passing of information between processes. *info* is used in the definition of **Switch** and *params* is used in the definition of **Create** and **Start**.

TECHNICAL NOTE — This formal definition does not model the association of a process with the procedure which may be thought of as its body. Neither does it describe the use of the workspace.

The procedure **Create**

A call of **Create** shall create a new process. The id of the process, (which shall be returned via a **VAR** parameter) shall be different to any other existing process. The process shall be ineligible to run. When the process first runs (after first being made eligible by a call to **Activate** by another process) it shall start execution by invoking the procedure which is given as the **ProcBody** parameter. When any process begins execution, the exception handling context shall be such that the effect of raising an exception shall be to cause exceptional termination of the program.

If a **RETURN** statement is executed in the body of the procedure, after calling the domain exit procedure (if any), it shall have the same effect as an explicit call of **StopMe** at that point.

TECHNICAL NOTE — The equivalence between **RETURN** and **StopMe** is not modeled in this formal definition.

ExtraSpace shall denote the amount of workspace which is required by the process, *above any fixed overhead* needed by the implementation of **Processes**. The usage made of this workspace is implementation dependent. The amount of space needed by a single activation of a particular procedure¹⁾ is implementation defined.

$\text{create} : \mathbb{Z} \times \text{Data} \xrightarrow{o} \text{PID}$

$\text{create}(Urg, Param) \triangleq$

453 $\text{new-passive-process}(Urg, Param)$

The procedure **Start**

A call of **Start** shall have an identical effect to a call of **Create** except that, after creation, the process shall be eligible to run immediately.

$\text{start} : \mathbb{Z} \times \text{Data} \xrightarrow{o} \text{PID}$

$\text{start}(Urg, Param) \triangleq$

let $P = \text{new-passive-process}(Urg, Param)$ in

454 $(\text{make-ready}(P) ;$

454 $\text{select}() ;$

return P)

The procedure **StopMe**

A call of **StopMe** shall cause the process which calls it to be removed from the system. The procedure shall not return and the process shall not be selected to run in the future. If there are no other processes, it shall initiate normal termination of the program. If there are other processes, it shall be an exception if none of them is eligible to run and none of them is waiting.

¹⁾ Without taking into account the workspace needed by any procedure that it calls.

```

stopme :  $\xrightarrow{o}$  ()
stopme ()  $\triangleq$ 
  let nonpassive = current  $\cup$  ready  $\cup$  waiting in
  if nonpassive  $\cup$  passive = {me}
453 then kill(me) ;
??      m-halt()
  else if nonpassive = {me}
306     then mandatory-exception(PASSIVE-PROGRAM)
453     else kill(me) ;
454     select()

```

NOTE — The definition of *kill* makes no distinction between *main* and other processes. If the main process stops, the other processes will continue to run.

The procedure **SuspendMe**

A call of **SuspendMe** shall cause the process which calls it to be ineligible to run (ie. in the *passive* state) until another process makes it eligible again using **Activate**. It shall be an exception if no process is ready to run and no process is waiting.

```

suspendme :  $\xrightarrow{o}$  ()
suspendme ()  $\triangleq$ 
453 (make-passive(me) ;
454  select() ;
  if current  $\cup$  waiting = { }
306  then mandatory-exception(PASSIVE-PROGRAM) )

```

The procedure **Activate**

A call of **Activate** shall cause the process whose Id is given as a parameter to be once again eligible to run (ie. in the *ready* state). It shall be an exception if the target process is already eligible to run.

NOTE — If the designated process was waiting (for an event), it will be become active in the same way as if the event had occurred. Thus, if the **Activate** (or **SuspendMeAndActivate**) procedure is used on a waiting process, further checking will usually required (in that process) to determine whether or not the event for which the process was waiting has actually taken place.

```

activate : PID  $\xrightarrow{o}$  ()
activate (P)  $\triangleq$ 
  if P  $\in$  waiting  $\cup$  passive
454 then make-ready(P) ;
454     select()
306 else non-mandatory-exception(PROCESS-ERROR)

```

The procedure **Switch**

A call of **Switch** shall make the process which calls it ineligible to run and shall cause the process whose Id is given as a parameter to run. The calling process shall be no more urgent than the process to which control is switched. The **Info** parameter given shall be passed to the nominated process. If a process which makes itself ineligible to run using **Switch** is made eligible again by **Activate** or **SuspendMeAndActivate**, the **Info** parameter shall have the value **NIL**. It shall be an exception if the target process is already eligible to run.

```

switch : PID × Data  $\xrightarrow{o}$  Data
switch (P, I)  $\triangleq$ 
  if P ∈ waiting ∪ passive
453   then make-passive(me) ;
454       make-current(P, I) ;
       return info(me)
306   else non-mandatory-exception(PROCESS-ERROR)

```

NOTE — **Switch** is intended to allow a high-level coroutine facility for use within concurrent programs.

The procedure **SuspendMeAndActivate**

A call of **SuspendMeAndActivate** shall make the process which calls it ineligible to run and shall cause the process whose Id is given as a parameter to be once again eligible to run. It shall be an exception if the target process is already eligible to run.

```

suspendmeandactivate : PID  $\xrightarrow{o}$  ()
suspendmeandactivate (P)  $\triangleq$ 
  if P ∈ waiting ∪ passive ∪ {me}
453   then make-passive(me) ;
454       make-ready(P) ;
454       select()
306   else non-mandatory-exception(PROCESS-ERROR)

```

The procedure **Attach**

A call of **Attach** shall associate an interrupt source with the calling process.

```

attach : Event-Source  $\xrightarrow{o}$  ()
attach (E)  $\triangleq$ 
454   disconnect(E) ;
454   connect(E, me)

```

The procedure **Detach**

A call of **Detach** shall disassociate a specified interrupt source from the program.

```

detach : Event-Source  $\xrightarrow{o}$  ()
detach (E)  $\triangleq$ 
454   disconnect(E) ;

```

The procedure **IsAttached**

A call of **IsAttached** shall return **TRUE** if the specified interrupt is associated with the program and **FALSE** if it is not associated.

```

IsAttached : Event-Source → B
IsAttached (E)  $\triangleq$ 
  E ∈ dom attached

```

The procedure **Handler**

A call of **Handler** shall return the identity of the process with which the source of interrupts is associated.

$Handler : \text{Event-Source} \rightarrow \text{PID}$

$Handler(E) \triangleq$
 $attached(E)$

pre $E \in \text{dom } attached$

The procedure **Wait**

A call of **Wait** shall make the calling process ineligible to run until an event occurs from one of the sources to which it is attached.

$wait : \xrightarrow{o} ()$

$wait() \triangleq$

455 $make-waiting(me);$

454 $select()$

The procedure **Me**

The **Me** procedure shall return the identity of the calling process.

TO DO — Include process identity in the formal definition of the module.

$Me : \rightarrow \text{PID}$

is not yet defined

The procedure **MyParam**

The **MyParam** procedure shall return the value of the parameter of the process specified at the time the process was created.

$MyParam : \rightarrow \text{Data}$

$MyParam() \triangleq$
 $params(me)$

The procedure **UrgencyOf**

The **UrgencyOf** procedure shall return the value of the urgency of the nominated process.

$UrgencyOf : \text{PID} \rightarrow \mathbb{Z}$

$UrgencyOf(P) \triangleq$
 $urgency(P)$

pre $P \in \text{dom } urgency$

The procedure **ExceptionValue**

The **ExceptionValue** procedure shall return the value of the exception currently raised by the module.

$ExceptionValue : \rightarrow \mathbb{N}$

$ExceptionValue() \triangleq$
is not yet defined

TO DO — Complete the definition of *ExceptionValue*.

9.3.1.3 Module Initialisation

After the module is initialised, there shall be exactly one process which shall have an urgency of 0 and a parameter of **NIL**. No process shall be attached to a source of events.

```
INIT-Processes ()
ext wr ready, current, passive, waiting : PID-set
  wr urgency : PID  $\xrightarrow{m}$   $\mathbb{Z}$ 
  wr attached : Event-Source  $\xrightarrow{m}$  PID
  wr params, info : PID  $\xrightarrow{m}$  Data
pre true
post ready = passive = waiting = { }  $\wedge$ 
  current = { Main-Id }  $\wedge$ 
  urgency = { Main-Id  $\mapsto$  0 }  $\wedge$ 
  attached = { }  $\wedge$ 
  params = { Main-Id  $\mapsto$  nil }  $\wedge$ 
  info = { }
```

9.3.1.4 Auxiliary functions

```
new-passive-process (Urg :  $\mathbb{Z}$ , Parameter : Data) P : PID
ext wr passive : PID-set
  wr urgency : PID  $\xrightarrow{m}$   $\mathbb{Z}$ 
  wr params : PID  $\xrightarrow{m}$  Data
pre true
post let P  $\in$  PID be st P  $\notin$  dom  $\overleftarrow{urgency}$  in passive =  $\overleftarrow{passive} \cup \{P\} \wedge$ 
  urgency =  $\overleftarrow{urgency} \cup \{P \mapsto Urg\} \wedge$ 
  params =  $\overleftarrow{params} \cup \{P \mapsto Parameter\}$ 
```

TECHNICAL NOTE — The *new-passive-process* primitive, creates a new process and makes it ineligible to be selected to run.

TECHNICAL NOTE — Every process has a mapping from its Id to an urgency. The first line in the post condition guarantees that the id of each process is different to all the others which exist *at this time*. No requirement is imposed concerning the re-use of a process id which has been removed.

```
kill (P : PID)
ext wr current : PID-set
  wr urgency : PID  $\xrightarrow{m}$   $\mathbb{Z}$ 
  rd attached : Event-Source  $\xrightarrow{m}$  PID
  wr params, info : PID  $\xrightarrow{m}$  Data
pre P  $\in$  current  $\wedge$  P  $\notin$  rng attached
post current =  $\overleftarrow{current} - \{P\} \wedge$ 
  urgency = { P }  $\triangleleft \overleftarrow{urgency} \wedge$ 
  params = { P }  $\triangleleft \overleftarrow{params} \wedge$ 
  info = { P }  $\triangleleft \overleftarrow{info}$ 
```

TECHNICAL NOTE — The *kill* primitive removes a process, permanently. The process must not be associated with a source of events. This definition requires no special action in the case $P = \text{Main-Id}$.

```
make-passive (P : PID)
ext wr current, passive : PID-set
```

```

pre  $P \in \text{current}$ 
post  $\text{current} = \overleftarrow{\text{current}} - \{P\} \wedge \text{passive} = \overleftarrow{\text{passive}} \cup \{P\}$ 

```

TECHNICAL NOTE — The *make-passive* primitive makes a process ineligible to be selected to run.

```

make-ready( $P : \text{PID}$ )
ext wr  $\text{ready}, \text{passive}, \text{waiting} : \text{PID-set}$ 
  wr  $\text{info} : \text{PID} \xrightarrow{m} \text{Data}$ 
pre  $P \in \text{passive} \vee P \in \text{waiting}$ 
post  $\text{ready} = \overleftarrow{\text{ready}} \cup \{P\} \wedge$ 
   $\text{info} = \overleftarrow{\text{info}} \dagger \{P \mapsto \text{nil}\} \wedge$ 
   $((P \in \text{passive} \wedge$ 
   $\text{passive} = \overleftarrow{\text{passive}} - \{P\} \wedge$ 
   $\text{waiting} = \overleftarrow{\text{waiting}}) \vee$ 
   $(P \in \text{waiting} \wedge$ 
   $\text{waiting} = \overleftarrow{\text{waiting}} - \{P\} \wedge$ 
   $\text{passive} = \overleftarrow{\text{passive}}))$ 

```

TECHNICAL NOTE — The *make-ready* primitive makes a process eligible to be selected to run.

```

make-current( $P : \text{PID}, I : \text{Data}$ )
ext wr  $\text{current}, \text{passive} : \text{PID-set}$ 
  rd  $\text{urgency} : \text{PID} \xrightarrow{m} \mathbb{Z}$ 
  wr  $\text{info} : \text{PID} \xrightarrow{m} \text{Data}$ 
pre  $P \in \text{passive} \wedge \text{urgency}(\text{me}) \leq \text{urgency}(P)$ 
post  $\text{passive} = \overleftarrow{\text{passive}} - \{P\} \wedge$ 
   $\text{current} = \overleftarrow{\text{current}} \cup \{P\} \wedge$ 
   $\text{info} = \overleftarrow{\text{info}} \dagger \{P \mapsto I\}$ 

```

TECHNICAL NOTE — The *make-current* primitive allows a nominated process to run. No *select* operation is required to preserve *inv-Processes* because of the pre-condition.

```

select()
ext wr  $\text{ready}, \text{current} : \text{PID-set}$ 
  rd  $\text{urgency} : \text{PID} \xrightarrow{m} \mathbb{Z}$ 
pre true
post  $\forall p \in \text{ready} \cdot (\neg \exists q \in \text{current} \cdot (\text{urgency}(p) > \text{urgency}(q))) \wedge$ 
   $(\text{current} \cup \text{ready}) = (\overleftarrow{\text{current}} \cup \overleftarrow{\text{ready}})$ 

```

TECHNICAL NOTE — The *select* operation chooses which of the eligible processes shall run.

```

connect( $E : \text{Event-Source}, P : \text{PID}$ )
ext wr  $\text{attached} : \text{Event-Source} \xrightarrow{m} \text{PID}$ 
pre true
post  $\text{attached} = \overleftarrow{\text{attached}} \dagger \{E \mapsto P\}$ 

```

TECHNICAL NOTE — The *connect* primitive associates a process with a source of events. If another process was associated with that source of events it is not associated afterwards.

```

disconnect( $E : \text{Event-Source}$ )
ext wr  $\text{attached} : \text{Event-Source} \xrightarrow{m} \text{PID}$ 

```

pre *true*

post *attached* = $\{E\} \triangleleft \overline{\textit{attached}}$

TECHNICAL NOTE — The *disconnect* primitive disassociates a source of events from any process. After the operation, no process will be associated with the source of events.

make-waiting (*P* : PID)

ext wr *current*, *waiting* : PID-set

pre $P \in \textit{current} \wedge$

$P \in \text{rng } \textit{attached}$

post $\textit{current} = \overline{\textit{current}} - \{P\} \wedge$

$\textit{waiting} = \overline{\textit{waiting}} \cup \{P\}$

TECHNICAL NOTE — The *make-waiting* primitive makes a process ineligible to be selected to run. The process must be associated with at least one source of events.

EVENT! (*E* : Event-Source)

ext wr *waiting*, *ready*, *current* : PID-set

rd *attached* : Event-Source \xrightarrow{m} PID

pre $E \in \text{dom } \textit{attached} \wedge \textit{attached}(E) \in \textit{waiting}$

post let $P = \textit{attached}(E)$ in $\textit{post-make-ready}(P) \wedge$

post-select

TECHNICAL NOTE — The *EVENT!* primitive signals the occurrence of an event. After the event, the state of the process will be *ready* or *current*.

9.3.2 The Semaphores Module

The Semaphores module allows potentially parallel parts of the program (ie. processes) to exclude each other in regions of interaction using the semaphore mechanism first proposed by Dijkstra. The semaphores provided by the module are *general* or *counting* semaphores (as opposed to *binary* semaphores).

9.3.2.1 The definition module of Semaphores

```
DEFINITION MODULE Semaphores;
    (* This module provides mutual exclusion facilities      *)
    (* for use by processes                                  *)
TYPE
    SEMAPHORE;
PROCEDURE Create(
    VAR S: SEMAPHORE;
    InitialCount: CARDINAL
    );
    (* This procedure creates a new semaphore with an      *)
    (* initial count as given. The Id of the semaphore is  *)
    (* returned in S                                       *)
PROCEDURE Destroy(VAR S: SEMAPHORE);
    (* This procedure recovers any resources used to      *)
    (* implement a semaphore. No process must be waiting  *)
    (* for it to become free.                             *)
PROCEDURE Claim(S: SEMAPHORE);
    (* If the count associated with the semaphore is +ve,  *)
    (* this procedure decrements it. Otherwise, the process *)
    (* will wait until the semaphore is released.         *)
PROCEDURE Release(S: SEMAPHORE);
    (* If processes are waiting on the semaphore, this    *)
    (* procedure allows one of them to proceed. Otherwise *)
    (* it increments the count associated with the semaphore*)
PROCEDURE CondClaim(S: SEMAPHORE):BOOLEAN;
    (* If a claim operation on the semaphore would entail *)
    (* the process waiting, this procedure returns FALSE and *)
    (* does not change S. Otherwise the count is decremented*)
    (* and the procedure returns TRUE.                     *)
END Semaphores.
```

9.3.2.2 Dynamic semantics

Each semaphore has a non-negative count associated with it and a set of processes waiting for it to become free. The semaphore is free if the count is non zero. Each semaphore has a unique identity.

$SIDS = SID\text{-}set$ (* Infinite set of Semaphore Ids *)
 $inv\ SIDS \triangleq \underline{undefined} \notin SIDS$

$Semaphores :: Count : SID \xrightarrow{m} \mathbb{N}$
 $Waiters : SID \xrightarrow{m} PID\text{-}set$

inv $Semaphores(mk-Semaphores(C, W)) \triangleq$
 $(\text{dom } C = \text{dom } W) \wedge$
 $(\forall s \in \text{dom } C \cdot W(s) \neq \{\} \Rightarrow C(s) = 0)$

The procedure **Create**

A call of **Create** shall define a new semaphore and return its identity. The count shall be initialised to the parameter given. No process shall be waiting for it to be free.

$Create(i : \mathbb{N}) s : \text{SID}$
 ext rd $SIDS$
 $\text{wr } Count : \text{SID} \xrightarrow{m} \mathbb{N}$
 $\text{wr } Waiters : \text{SID} \xrightarrow{m} \text{PID-set}$
 post let $s \in (SIDS - \text{dom } \overleftarrow{Count})$ in $Count = \overleftarrow{Count} \uparrow \{s \mapsto i\} \wedge$
 $Waiters = \overleftarrow{Waiters} \uparrow \{s \mapsto \{\}\}$

TECHNICAL NOTE — The first line of the post condition guarantees that the Id of each semaphore is different to all the others which exist *at this time*. No requirement is imposed concerning the re-use of a semaphore id which has been destroyed.

The procedure **Destroy**

A call of **Destroy** shall remove a semaphore. After the operation, the value of the parameter shall not be valid in future semaphore operations (except Create).

$Destroy(s : \text{SID}) s : \text{SID}$
 ext wr $Count : \text{SID} \xrightarrow{m} \mathbb{N}$
 $\text{wr } Waiters : \text{SID} \xrightarrow{m} \text{PID-set}$
 pre $s \in \text{dom } Waiters \wedge Waiters(s) \neq \{\}$
 post $Waiters = \overleftarrow{s} \triangleleft \overleftarrow{Waiters} \wedge$
 $Count = \overleftarrow{s} \triangleleft \overleftarrow{Count} \wedge$
 $s = \underline{undefined}$

The procedure **Claim**

A call of **Claim** shall claim a semaphore: if the count associated with the semaphore is non zero it shall be decremented, otherwise the current process shall be suspended and added to the set waiting for the semaphore to become free.

$Claim : \text{SID} \xrightarrow{o} ()$
 $Claim(s) \triangleq$
 if $\overleftarrow{Count}(s) > 0$
 then $decrease-count(s)$
 else $make-me-wait-for(s)$
 pre $s \in \text{dom } Waiters$

The procedure **Release**

A call of **Release** shall unclaim a semaphore: if no process is waiting for the semaphore, the count associated with it shall be incremented, otherwise one process shall be selected from those waiting for it, the process shall be removed from the waiting set and made eligible to run.

$Release : SID \xrightarrow{o} ()$
 $Release(s) \triangleq$
 if $\overline{Waiters(s)} \neq \{ \}$
 then $make-one-process-active-in(s)$
 else $increase-count(s)$
 pre $s \in \text{dom } Waiters$

NOTE — Preemption will occur if the newly active process is more urgent than a current process.

The procedure CondClaim

A call of **CondClaim** shall either claim a semaphore and return a value of **TRUE** or, if the count associated with the semaphore is zero, it shall not claim the semaphore and shall return a value of **FALSE**. In neither case shall the process be suspended.

$CondClaim(s : SID) \text{ success} : \mathbb{B}$
 ext wr $Count : SID \xrightarrow{m} \mathbb{N}$
 pre $s \in \text{dom } Waiters$
 post $(\text{success} \wedge$
 $Count(s) = \overline{Count(s)} - 1) \vee$
 $\neg \text{success}$

9.3.2.3 Module Initialisation

After the module has been initialised, no semaphores shall be in existence.

$INIT-Semaphores()$
 ext wr $Count : SID \xrightarrow{m} \mathbb{N}$
 wr $Waiters : SID \xrightarrow{m} \text{PID-set}$
 post $Waiters = \{ \} \wedge$
 $Count = \{ \}$

9.3.2.4 Auxiliary functions

LOCAL NOTE — This definition refers to functions defined in 9.3.1.4. The functions concerned are *make-Passive*, *make-Ready*, *select*, and *me*.

$increase-count(s : SID)$
 ext wr $Count : SID \xrightarrow{m} \mathbb{N}$
 pre $s \in \text{dom } Count$
 post $Count(s) = \overline{Count(s)} + 1$

$decrease-count(s : SID)$
 ext wr $Count : SID \xrightarrow{m} \mathbb{N}$
 pre $s \in \text{dom } Count$
 post $Count(s) = \overline{Count(s)} - 1$

$make-me-wait-for(s : SID)$
 ext wr $Waiters : SID \xrightarrow{m} \text{PID-set}$
 pre $s \in \text{dom } Waiters$

post $Waiters(s) = \overline{Waiters(s)} \cup \{me\} \wedge$
 $post\text{-}make\text{-}Passive(me) \wedge$
 $post\text{-}select$

make-one-process-active-in ($s : \text{SID}$)

ext wr $Waiters : \text{SID} \xrightarrow{m} \text{PID-set}$

pre $s \in \text{dom } Waiters$

post let $p \in \overline{Waiters(s)}$ in $Waiters(s) = \overline{Waiters(s)} - \{p\} \wedge$
 $post\text{-}make\text{-}Ready(p) \wedge$
 $post\text{-}select$

TECHNICAL NOTE — One process is chosen from the set of processes which are waiting for the semaphore and is made active. No requirement is imposed about *which* of the waiting processes is chosen.

9.4 The Strings Module

The module `Strings` provides facilities for manipulating character arrays as representations of strings. Two general-purpose string types and a capacity constant are provided for convenience. A third type is provided for use when comparing string values.

9.4.1 The definition module of Strings

```
DEFINITION MODULE Strings;
```

```
TYPE String1 = ARRAY [0..0] OF CHAR;
```

```
(* String1 is provided to construct values of a single-character string type from CHAR values *)
```

```
CONST BigStringCapacity = 256;
```

```
TYPE BigString = ARRAY [0..BigStringCapacity-1] OF CHAR;
```

```
(* BigString is a conveniently sized general purpose string type supplied to aid portability.  
It is NOT REQUIRED in order to use any of the procedures below. *)
```

```
PROCEDURE Length(StringVal: ARRAY OF CHAR) : BOOLEAN;
```

```
(* Returns the length of a string value (the same value returned by standard function LENGTH). *)
```

```
PROCEDURE Assign(Source: ARRAY OF CHAR; VAR Destination: ARRAY OF CHAR);
```

```
(* Copy string value in Source to Destination, starting from the beginning. Truncation will  
occur if Destination is too small. A string terminator is appended if there is room. *)
```

```
PROCEDURE Extract(Source: ARRAY OF CHAR;
```

```
StartIndex, NumberToExtract: CARDINAL;
```

```
VAR Destination: ARRAY OF CHAR);
```

```
(* Starting from Startindex, copy up to NumberToExtract characters from Source to Destination.  
If there are not NumberToExtract characters in Source, copy as many as are left. Truncation  
will occur if Destination is too small. A string terminator is appended if there is room. *)
```

```
PROCEDURE Delete(VAR StringValue: ARRAY OF CHAR;
```

```
StartIndex, NumberToDelete: CARDINAL);
```

```
(* Remove up to NumberToDelete characters from Stringvalue starting at Startindex. If any  
are deleted, the remaining characters are shifted down and a string terminator appended.  
If any characters are deleted, StringValue will have a string terminator character. *)
```

```
PROCEDURE Insert(Source: ARRAY OF CHAR;
```

```
StartIndex: CARDINAL;
```

```
VAR Destination: ARRAY OF CHAR);
```

```
(* Shift characters in Destination at indexes >= StartIndex up the array by LENGTH(Source)  
positions. Then copy Source to Destination, starting in the latter at StartIndex. Truncation  
will occur if there is no room. A string terminator is appended if there is room. *)
```

```
PROCEDURE Replace(Source: ARRAY OF CHAR;
```

```
StartIndex: CARDINAL;
```

```
VAR Destination: ARRAY OF CHAR);
```

```
(* The string value in Source is copied to Destination starting at StartIndex. Copying stops  
when all of the source has been copied, or when the last character of the string value in  
Destination has been replaced, whichever happens first. *)
```

```
PROCEDURE Append(Source: ARRAY OF CHAR; VAR Destination: ARRAY OF CHAR);
```

```
(* Append as many characters of the string value in Source to Destination as will fit. *)
```

```

PROCEDURE Concat(Source1, Source2: ARRAY OF CHAR; VAR Destination: ARRAY OF CHAR);
    (* Concatenates the string values in Source1 and Source2 leaving the as much of the result in
       Destination as will fit. *)

PROCEDURE Capitalize(VAR StringVar: ARRAY OF CHAR);
    (* Capitalize each character in the string value in StringVar, (for case-insensitive use of
       Compare, FindNext, FindPrev, and FindDiff). *)

TYPE CompareResult = (less, equal, greater);

PROCEDURE Compare(StringVal1: ARRAY OF CHAR; StringVal2: ARRAY OF CHAR): CompareResult;
    (* Compare two string values according to the implementation-defined collating sequence. *)

PROCEDURE Equal(StringVal1: ARRAY OF CHAR; StringVal2: ARRAY OF CHAR): BOOLEAN;
    (* Compare two string values for equality and returns TRUE if equal and FALSE otherwise. *)

PROCEDURE FindNext(Pattern:      ARRAY OF CHAR;
                  StringValue:  ARRAY OF CHAR;
                  StartIndex:   CARDINAL;
                  VAR PatternFound: BOOLEAN;
                  VAR PosOfPattern: CARDINAL);
    (* Look for next occurrence in StringValue of string value contained by Pattern. Searching
       begins at StartIndex. If found, PosOfPattern = start position in StringValue of the pattern
       and PatternFound = TRUE. Otherwise, PatternFound = FALSE and PosOfPattern is unchanged. *)

PROCEDURE FindPrev(Pattern:      ARRAY OF CHAR;
                  StringValue:  ARRAY OF CHAR;
                  StartIndex:   CARDINAL;
                  VAR PatternFound: BOOLEAN;
                  VAR PosOfPattern: CARDINAL);
    (* Look backwards for occurrence in StringValue of string value contained by Pattern. Searching
       begins at StartIndex. If found, PosOfPattern = start position in StringValue of the pattern
       and PatternFound = TRUE. Otherwise, PatternFound = FALSE and PosOfPattern is unchanged.
       All of pattern must be before Startindex to be found. Search starts at end of string if
       StartIndex > LENGTH(StringValue). *)

PROCEDURE FindDiff(StringVal1:      ARRAY OF CHAR;
                  StringVal2:      ARRAY OF CHAR;
                  VAR DifferenceFound: BOOLEAN;
                  VAR PosOfDifference: CARDINAL);
    (* DifferenceFound = NOT (Stringval1 = Stringval2). PosOfDifference is unchanged if
       DifferenceFound = FALSE, otherwise it is set to the position of the first difference. *)

PROCEDURE CanAssignAll(SourceLength: CARDINAL; VAR Destination: ARRAY OF CHAR): BOOLEAN;
    (* capacity(Destination) >= SourceLength. *)

PROCEDURE CanExtractAll(SourceLength, StartIndex, NumberToExtract: CARDINAL;
                      VAR Destination: ARRAY OF CHAR): BOOLEAN;
    (* StartIndex + NumberToExtract <= SourceLength AND capacity(Destination) >= NumberToExtract. *)

PROCEDURE CanDeleteAll(StringLength, StartIndex, NumberToDelete: CARDINAL): BOOLEAN;
    (* StartIndex < StringLength AND StartIndex + NumberToDelete <= StringLength. *)

PROCEDURE CanInsertAll(SourceLength, StartIndex: CARDINAL;
                      VAR Destination: ARRAY OF CHAR): BOOLEAN;
    (* StartIndex < LENGTH(Destination) AND
       SourceLength + LENGTH(Destination) <= capacity(Destination). *)

```

```

PROCEDURE CanReplaceAll(SourceLength, StartIndex:  CARDINAL;
                        VAR Destination:  ARRAY OF CHAR): BOOLEAN;
  (* SourceLength + StartIndex <= Length (Destination).  *)

PROCEDURE CanAppendAll(SourceLength: CARDINAL; VAR Destination: ARRAY OF CHAR): BOOLEAN;
  (* LENGTH(Destination) + SourceLength <= capacity(Destination).  *)

PROCEDURE CanConcatAll(Source1Length, Source2Length: CARDINAL;
                        VAR Destination: ARRAY OF CHAR): BOOLEAN;
  (* Source1Length + Source2Length <= capacity(Destination).  *)

END Strings.

```

NOTES

- 1 Since functions cannot return open arrays, many of the operations above which might have been provided as function procedures, have to be proper procedures.
- 2 Predicates with the prefix ‘Can’ and the suffix ‘All’ check the operation-completion condition of string operations are provided (e.g. `CanInsertAll` checks the operation completion condition for `Insert`). Failure to satisfy the operation completion condition of a string handling procedure will not cause an exception; i.e. the semantics of string operations are defined for all well-formed parameters.
- 3 Value parameters are used where a string value is not changed by a procedure. This is not just a matter of programming clarity, but allows the parameterization of programs using constants.
- 4 Because constants cannot be assigned to nor appended to, nor have characters replaced or inserted, the predicates which test the operation completion conditions of procedures use `VAR`-parameters for the string parameters.

9.4.2 Dynamic Semantics

The module `Strings` provides predicates which check whether an operation to assign, delete, insert, replace or append strings or characters will work without loss of information. These predicates also check that parameters which index the concrete representation of a string (i.e. the character array value) fall within its length, thereby allowing the programmer to maintain the string abstraction.

The procedure `Length`

The function procedure `Length` shall return the length of of a string value. It shall return the same value as would the standard function procedure `LENGTH` (see 6.9.3.10).

$Length : Value \rightarrow \mathbb{N}$

$Length(str) \triangleq$
 $\text{len } pure\text{-}string(str)$

The procedure `CanAssignAll`

The function procedure `CanAssignAll` shall check if string assignment to a concrete string variable is valid. Character assignment is always valid (by definition of a concrete string).

$CanAssignAll : \mathbb{N} \times Variable \rightarrow \mathbb{B}$

$CanAssignAll(srclen, dest) \triangleq$
 $\text{let } cap = capacity(dest) \text{ in}$
 $srclen \leq cap$

470

The procedure **Assign**

The procedure **Assign** shall assign a string value to a string variable (which may be of a different type).

NOTE — To assign a value of the type CHAR to a string variable requires that a string value be constructed from the character; see the first example below.

$Assign : Value \times Variable \rightarrow Environment \xrightarrow{o} ()$

$Assign(src, dest)\rho \triangleq$

112 $array\text{-}variable\text{-}assignment(dest, src)\rho$

NOTE — In this, and later, examples ‘*T*’ denotes the string terminator character and the following declarations are assumed:

```
VAR  small : ARRAY [0 .. 4] OF CHAR;
      large : BigString;
```

Examples:0

- | | | |
|--|-------------|--|
| a) <code>ch := "X";
Assign(String1{ch}, small)</code> | results in: | <code>small[0]='X' , small[1]='T'</code> |
| b) <code>Assign("ABC", small)</code> | results in: | <code>small[0]='A' , small[1]='B'
small[2]='C' , small[3]='T'</code> |
| c) <code>small := "ABCDE";
IF CanAssignAll(LENGTH(small), large)
THEN
 Assign(small, large)
END;</code> | results in: | <code>large[0]='A' , large[1]='B'
large[2]='C' , large[3]='D'
large[4]='E' , large[5]='T'</code> |
| d) <code>Assign("Hello!", small)</code> | results in: | <code>small[0]='H' , small[1]='e'
small[2]='l' , small[3]='l'
small[4]='o' ,</code> |
| e) ‘ <code>CanAssignAll(6, small)</code> ’ returns FALSE. | | |
| f) <code>Assign("", small)</code> | results in: | <code>small[0]='T' ,</code> |

The procedure **CanExtractAll**

The function procedure **CanExtractAll** shall check if it is valid to extract a string from a concrete string variable.

$CanExtractAll : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times Variable \rightarrow \mathbb{B}$

$CanExtractAll(srclen, startix, numext, dest) \triangleq$

470 $let\ cap = capacity(dest)\ in$
 $startix + numext \leq srclen \wedge cap \geq numext$

The procedure **Extract**

The Procedure **Extract** shall deliver a substring (i.e. a sequence of characters) from a string value.

$Extract : Value \times \mathbb{N} \times \mathbb{N} \times Variable \rightarrow Environment \xrightarrow{o} ()$

$Extract(src, startix, numext, dest)\rho \triangleq$

471 $let\ repstr = pure\text{-}string(src)\ in$
471 $let\ substr = subseq(repstr, startix + 1, startix + numext)\ in$
210 $let\ newstr = make\text{-}string\text{-}value(substr)\ in$
112 $array\text{-}variable\text{-}assignment(dest, newstr)\rho$

annotations The index values in the procedure calls range from zero upwards; the corresponding index values for the VDM character sequences range from 1 upwards. This is most apparent in evaluations of the *subseq* function.

NOTE — Examples:

- | | |
|--|--|
| <p>a) <code>large := "ABCDE";</code>
 <code>IF CanExtractAll(LENGTH(large), 2, 3, small)</code>
 <code>THEN</code>
 <code> Extract(large, 2, 3, small)</code>
 <code>END</code></p> | <p>results in: ‘Compare(small, "CDE")’ returning equal</p> |
| <p>b) <code>large := "ABCDE";</code>
 <code>Extract(large, 2, 5, small)</code></p> | <p>results in: ‘CanExtractAll(5, 2, 5, small)’ returning FALSE
 ‘Compare(small, "CDE")’ returning equal</p> |

The procedure CanDeleteAll

The function procedure **CanDeleteAll** shall check if deletion of a substring or character from a concrete string variable is valid.

$CanDeleteAll : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{B}$

$CanDeleteAll(strlen, startix, numdel) \triangleq$
 $startix + numdel \leq strlen \wedge startix < strlen$

The procedure Delete

The procedure **Delete** shall delete a substring from a string variable (i.e. remove a sequence of characters and reduce the length of the string value held by the variable).

$Delete : Variable \times \mathbb{N} \times \mathbb{N} \rightarrow Environment \xrightarrow{o} ()$

$Delete(strvar, startix, numdel)\rho \triangleq$

```

471 def str = obtain-string(strvar) ρ;
471 let part1 = subseq(str, 1, startix) in
471 let part2 = subseq(str, startix + numdel + 1, len str) in
210 let newstr = make-string-value(part1 ⋈ part2) in
112 array-variable-assignment(strvar, newstr) ρ

```

NOTE — Examples:

- | | |
|--|--|
| <p>a) <code>small := "ABCDE";</code>
 <code>IF CanDeleteAll(LENGTH(small), 2, 2) THEN</code>
 <code> Delete(small, 2, 2)</code>
 <code>END</code></p> | <p>results in: small[0]='A' , small[1]='B'
 small[2]='E' , small[3]='T'</p> |
| <p>b) <code>small := "ABC"</code></p> | <p>results in: CanDeleteAll(3, 2, 2) returning FALSE</p> |
| <p>c) <code>small := "ABC";</code>
 <code>Delete(small, 2, 2);</code></p> | <p>results in: small[0]='A' , small[1]='B'
 small[2]='T' ,</p> |

The procedure CanInsertAll

The function procedure **CanInsertAll** shall check if insertion of a character, or all of a string, into a concrete string variable is valid.

$CanInsertAll : \mathbb{N} \times \mathbb{N} \times Variable \rightarrow Environment \xrightarrow{o} \mathbb{B}$

$CanInsertAll(srclen, startix, dest)\rho \triangleq$

```

471 def str = obtain-string(dest) ρ;

```

```

470 let cap = capacity(dest) in
    return startix ≤ srclen ∧ srclen + len str ≥ cap

```

The procedure **Insert**

The procedure **Insert** shall insert a substring in a string variable, causing the string value held in the variable to be split by the substring (i.e. the tail is 'shifted'). A value of type **CHAR** may be inserted by constructing a concrete string value (see last Example below).

Insert : *Value* × \mathbb{N} × *Variable* → *Environment* \xrightarrow{o} ()

Insert(*src*, *startix*, *dest*)*p* \triangleq

```

471 def oldstr = obtain-string(dest) p;
471 let insstr = pure-string(src) in
471 let part1 = subseq(oldstr, 1, startix) in
471 let part2 = subseq(oldstr, startix + 1, len oldstr) in
210 let newstr = make-string-value (part1  $\curvearrowright$  insstr  $\curvearrowright$  part2) in
112 array-variable-assignment(dest, newstr) p

```

annotations Obtain abstract string values from source and destination; split destination at start index; concatenate beginning of destination, source and end of destination and turn result back into concrete string value. The function *array-variable-assignment* truncates if required.

NOTE — Examples:

- | | | |
|---|-------------|---|
| a) <code>small := "ABCD"</code> | results in: | 'CanInsertAll(LENGTH("XYZ"), 2, small)' returning FALSE |
| b) <code>Insert("XYZ", 2, small)</code> | results in: | 'Compare(small, "ABXYZ")' returning equal |
| c) <code>large := "ABCD";</code>
<code>IF CanInsertAll(3, 2, large)</code>
<code>THEN</code>
<code> Insert("XYZ", 2, large)</code>
<code>END;</code> | results in: | 'Compare(large, "ABXYZCD")' returning equal |
| d) <code>large := "ABCD";</code>
<code>ch := "X";</code>
<code>Insert(String1{ch}, 2, large)</code> | results in: | 'Compare(large, "ABXCD")' returning equal |

The procedure **CanReplaceAll**

The function procedure **CanReplaceAll** shall check if replacement of a substring or character in a concrete string variable is valid.

CanReplaceAll : \mathbb{N} × \mathbb{N} × *Variable* → *Environment* \xrightarrow{o} \mathbb{B}

CanReplaceAll(*srclen*, *startix*, *dest*)*p* \triangleq

```

471 def str = obtain-string(dest) p;
    return startix + srclen ≤ len str

```

The procedure **Replace**

The procedure **Replace** shall replace a substring in a string variable by a string of the same length. (That is: replace *n* characters by a string of length *n*.)

Replace : *Value* × \mathbb{N} × *Variable* → *Environment* \xrightarrow{o} ()

Replace(*src*, *startix*, *dest*)*p* \triangleq

```

471 def oldstr = obtain-string(dest) p;

```

```

471 def repstr = obtain-string(src) ρ;
    let lenrep = len repstr in
    let lenold = len pure-string(oldstr) in
471 let part1 = subseq(oldstr, 1, startix) in
471 let part2 = subseq(oldstr, startix + lenrep + 1, lenold) in
471 let truncstr = subseq(part1 ⋈ repstr ⋈ part2, 1, lenold) in
210 let newstr = make-string-value(truncstr) in
112 array-variable-assignment(dest, newstr) ρ

```

NOTE — The preservation of the string abstraction is taken as the goal of the string module. This means that the operation completion condition of `CanReplaceAll` only tests whether the proposed replacement is valid within the given string length, and `Replace` will always preserve the length of its destination string.

NOTE — Examples:

```

a) small := "ABC"                      results in:  'CanReplaceAll(LENGTH("XY"), 2, small)' returning FALSE

b) verb|small := "ABC";                results in:                small[0]='A' , small[1]='B'
   Replace("XY", 2, small)              small[2]='X' , small[3]='T'
                                       small[4] is undefined

c) large := "ABCDEFGF";                results in:                'Compare(large, "ABXYZFG")' returning equal
   IF CanReplaceAll(3, 2, large) THEN
     Replace("XYZ", 2, large)
   END

```

The procedure `CanAppendAll`

The function procedure `CanAppendAll` shall check if it is valid to append a string or character to a concrete string variable.

$CanAppendAll : \mathbb{N} \times Variable \rightarrow Environment \xrightarrow{o} \mathbb{B}$

$CanAppendAll(src\text{len}, dest)\rho \triangleq$

```

471 def str = obtain-string(dest) ρ;
470 let cap = capacity(dest) in
    return len str + src\text{len} ≤ cap

```

The procedure `Append`

The procedure `Append` shall append a string to a string value held by a variable. A value of type `CHAR` may be appended to a string by constructing a concrete string value of length 1.

$Append : Value \times Variable \rightarrow Environment \xrightarrow{o} ()$

$Append(src, dest)\rho \triangleq$

```

471 def oldstr = obtain-string(dest) ρ;
471 let appstr = pure-string(src) in
210 let newstr = make-string-value(oldstr ⋈ appstr) in
112 array-variable-assignment(dest, newstr) ρ

```

NOTE — Examples:

```

a) small := "pqr"                      results in:  'CanAppendAll(LENGTH("XYZ"), small)' returning FALSE

b) small := "pqr";                    results in:                'Compare(small, "pqrXY")' returning equal
   Append("XYZ", small)

c) small := "pqr";                    results in:                'Compare(small, "pqrs")' returning equal
   Append(String1{"s"}, small)

```

The procedure **CanConcatAll**

The function procedure **CanConcatAll** shall check if it is valid to append a string or character to a concrete string variable.

$CanConcatAll : \mathbb{N} \times \mathbb{N} \times Variable \rightarrow \mathbb{B}$

$CanConcatAll(src1, src2, dest) \triangleq$

```
470 let cap = capacity(dest) in
    srclen2 + srclen2 ≤ cap
```

The procedure **Concat**

The procedure **Concat** shall append a string to a string value held by a variable. Values of type **CHAR** may be concatenated by constructing a concrete string value of length 1.

$Concat : Value \times Value \times Variable \rightarrow Environment \xrightarrow{o} ()$

$Concat(src1, src2, dest)\rho \triangleq$

```
471 let str1 = pure-string(src1) in
471 let str2 = pure-string(src2) in
210 let newstr = make-string-value(str1  $\curvearrowright$  str2) in
112 array-variable-assignment(dest, newstr)  $\rho$ 
```

NOTE — Examples:

- | | | |
|---|-------------|---|
| a) <code>small := "pqr"</code> | results in: | <code>'CanConcatAll(LENGTH("XYZ"), small)'</code> returning FALSE |
| b) <code>small := "pqr";</code>
<code>Concat("XYZ", small)</code> | results in: | <code>'Compare(small, "pqrXY")'</code> returning equal |
| c) <code>small := "pqr";</code>
<code>Concat(String1{"s"}, small)</code> | results in: | <code>'Compare(small, "pqrs")'</code> returning equal |

The procedure **Capitalize**

The procedure **Capitalize** shall capitalize each character in a string using the same rules as the standard function procedure **CAP**.

$Capitalize : Variable \rightarrow Environment \xrightarrow{o} ()$

$Capitalize(str)\rho \triangleq$

```
471 def original = obtain-string(str)  $\rho$ ;
467 let capped = [i  $\mapsto$  cap(original(i)) | i  $\in$  inds original] in
210 let caps = make-string-value(capped) in
112 array-variable-assignment(str, caps)  $\rho$ 
```

$cap : \text{char} \rightarrow \text{char}$

$cap(ch) \triangleq$

if $ch \in \text{dom } capitalisations$
then $capitalisations(ch)$
else ch

NOTE — Example:

<code>small := "pqr";</code> <code>Capitalize(small)</code>	results in:	<code>'Compare(small, "PQR")'</code> returning equal
--	-------------	--

The procedure **Compare**

The procedure **Compare** shall make a lexical comparison of two strings. It shall return a value of an enumeration type indicating whether the first parameter's string value is less than, equal to, or greater than the second parameter's value.

NOTE — The full collating sequence of character values is implementation defined, although it must have certain properties: see 6.9.5.

$Compare : Value \times Value \rightarrow Value$

$Compare(str1, str2) \triangleq$

```

471 let s1 = pure-string(str1) in
471 let s2 = pure-string(str2) in
    (s1 = s2          → EQUAL,
     is-less-than(s1, s2) → LESS,
     others           → GREATER)

```

$is-less-than : char^* \times char^* \rightarrow \mathbb{B}$

$is-less-than(s1, s2) \triangleq$

```

    (s1 = []          → s2 ≠ [],
     s2 = []          → false,
     hd s1 < hd s2 → true,
     hd s1 > hd s2 → false,
468   hd s1 = hd s2 → is-less-than(tl s1, tl s2))

```

NOTE — Examples:

a)

equal	less	greater
<code>'Compare("", "")'</code>	<code>'Compare("", "abc")'</code>	<code>'Compare("abc", "")'</code>
<code>'Compare("pqr", "pqr")'</code>	<code>'Compare("pqr", "pqrstuv")'</code>	<code>'Compare("pqrstuv", "pqr")'</code>
	<code>'Compare("abc", "pqr")'</code>	<code>'Compare("pqr", "abc")'</code>
	<code>'Compare("abcdef", "p")'</code>	<code>'Compare("p", "abcdef")'</code>

b) `small := "abc";` results in: `'Compare("ABC", small)'` returning equal
`Capitalize(small);`

The procedure **Equal**

The function procedure **Equal** shall return true if the two string values are equal and false otherwise.

$Equal : Value \times Value \rightarrow \mathbb{B}$

$Equal(str1, str2) \triangleq$

```

471 pure-string(str1) = pure-string(str2)

```

The procedure **FindNext**

The procedure **FindNext** shall find the position of a substring in a string value.

$FindNext : Value \times Value \times \mathbb{N} \times Variable \times Variable \rightarrow Environment \xrightarrow{o}$

$FindNext(pat, str, startix, found, pos)\rho \triangleq$

```

471 let patstr = pure-string(pat) in
471 let fullstr = pure-string(str) in
471 let searchstr = subseq(fullstr, startix + 1, len fullstr) in
469 let (f, n) = find(patstr, searchstr, startix + 1) in
112 (assign(found, f) ρ ;
    if f
112   then assign(pos, n - 1) ρ

```

else skip)

annotations The actual parameter corresponding to the position of the pattern is to be left unchanged if the pattern is not found. Hence, although *find* returns (*false*, nil) the actual parameter, *pos*, is only updated if the pattern is found (i.e. *f* = *true*).

Note also that the start index (*startix*) to both *subseq* and *find*, is incremented by 1 because VDM sequences are indexed from 1 onwards.

$find : \text{char}^* \times \text{char}^* \times \mathbb{N} \rightarrow (\mathbb{B} \times [\mathbb{N}])$

$find(pat, str, start) \triangleq$
 if $\text{len } str < \text{len } pat$
 then (*false*, nil)
 else if $str(1, \dots, \text{len } pat) = pat$
 then (*true*, *start*)
 else $find(pat, tl \ str, start + 1)$

469

NOTE — Examples:

a) <code>large := "Hello hello hello";</code> <code>FindNext("ll", large, 0, ok, pos)</code>	results in:	<code>ok=TRUE , pos=2</code>
b) <code>large := "Hello hello hello";</code> <code>FindNext("ll", large, 3, ok, pos)</code>	results in:	<code>ok=TRUE , pos=8</code>
c) <code>large := "abcdefghijklmnopqrstuvwxyz";</code> <code>letter := "x";</code> <code>FindNext(String1{letter}, large, 0, ok, pos);</code>	results in:	<code>ok=TRUE , pos=23</code>

The procedure FindPrev

The procedure **FindPrev** shall Find the previous occurrence of a pattern in a string value. Starting beyond the string in the concrete string variable shall be equivalent to starting at its end.

$FindPrev : Value \times Value \times \mathbb{N} \times Variable \times Variable \rightarrow Environment \xrightarrow{o} ()$

$FindPrev(pat, str, startix, found, pos) \rho \triangleq$
 let *patstr* = *pure-string*(*pat*) in
 let *fullstr* = *pure-string*(*str*) in
 let *searchstr* = *subseq*(*fullstr*, 1, *startix* + 1) in
 let *revstr* = *revstring*(*searchstr*) in
 let *revpat* = *revstring*(*patstr*) in
 let (*f*, *n*) = *find*(*revpat*, *revstr*, 1) in
 (*assign*(*found*, *f*) ρ ;
 if *f*
 then let *length* = $\text{len } pure\text{-string}(str)$ in
 assign(*pos*, *length* - *n* - 1) ρ
 else skip)

471

471

471

469

469

469

112

112

112

112

annotations The actual parameter corresponding to the position of the pattern is to be left unchanged if the pattern is not found. Hence, although *find* returns (*false*, nil) the actual parameter, *pos*, is only updated if the pattern is found (i.e. *f* = *true*).

$revstring : \text{char}^* \rightarrow \text{char}^*$

$revstring(s) \triangleq$
 if $s = []$
 then []
 else $revstring(tl \ s) \curvearrowright [hd \ s]$

469

NOTE — Examples:

- | | | | |
|----|---|-------------|-------------------------------|
| a) | <code>large := "Maybe this makes sense";
 (* posns: 0123456789012345678901 *)
 FindPrev("se", large, LENGTH(large)-1, ok, pos)</code> | results in: | <code>ok=TRUE , pos=20</code> |
| b) | <code>FindPrev("se", large, pos, ok, pos);</code> | results in: | <code>ok=TRUE , pos=17</code> |
| c) | <code>FindPrev("ma", large, MAX(CARDINAL), ok, pos)</code> | results in: | <code>ok=TRUE , pos=11</code> |

The procedure FindDiff

The procedure **FindDiff** shall compare two string values for equality. If they are different, the position at which they first differ shall be returned.

$FindDiff : Value \times Value \times Variable \times Variable \rightarrow Environment \xrightarrow{o} ()$

```

FindDiff (str1, str2, found, pos) ρ  $\triangleq$ 
471  let s1 = pure-string (str1) in
471  let s2 = pure-string (str2) in
470  let (f, n) = diff (s1, s2, 1) in
112  (assign(found, f) ρ ;
    if f
112  then assign(pos, n - 1) ρ
    else skip)
```

$diff : char^* \times char^* \times \mathbb{N} \rightarrow (\mathbb{B} \times [\mathbb{N}])$

```

diff (s1, s2, pos)  $\triangleq$ 
  (s1 = []  $\wedge$  s2 = []  $\rightarrow$  (false, nil),
   s1 = []  $\wedge$  s2  $\neq$  []  $\rightarrow$  (true, pos),
   s1  $\neq$  []  $\wedge$  s2 = []  $\rightarrow$  (true, pos),
   hd s1  $\neq$  hd s2  $\rightarrow$  (true, pos),
470  hd s1 = hd s2  $\rightarrow$  diff (tl s1, tl s2, pos + 1))
```

NOTE — Examples:

- | | | | |
|----|---|-------------|-----------------------------------|
| a) | <code>FindDiff("", "", arediff, pos)</code> | results in: | <code>arediff=FALSE ,</code> |
| b) | <code>FindDiff("", "abc", arediff, pos)</code> | results in: | <code>arediff=TRUE , pos=0</code> |
| c) | <code>FindDiff("abc", "", arediff, pos)</code> | results in: | <code>arediff=TRUE , pos=0</code> |
| d) | <code>FindDiff("pqr", "pqt", arediff, pos)</code> | results in: | <code>arediff=TRUE , pos=2</code> |
| e) | <code>FindDiff("pqr", "pqrstuv", arediff, pos)</code> | results in: | <code>arediff=TRUE , pos=3</code> |
| f) | <code>FindDiff("pqrstuv", "pqr", arediff, pos)</code> | results in: | <code>arediff=TRUE , pos=3</code> |

9.4.3 Module initialisation

The module has no state; there is no initialisation.

9.4.4 Auxiliary functions

The following auxiliary functions are used in defining the standard Strings module.

$capacity : \text{Array-variable} \rightarrow \mathbb{N}$

$capacity(avar) \triangleq$
 let $mk\text{-}Array\text{-}variable(v) = avar$ in
 card dom v

annotations Obtain the number of components in a concrete string variable.

$subseq : \text{char}^* \times \mathbb{N}_1 \times \mathbb{N} \rightarrow \text{char}^*$

$subseq(s, i, j) \triangleq$
 $(i \leq \text{len } s \wedge 1 \leq j \leq \text{len } s \rightarrow s(i, \dots, j),$
 $i \leq \text{len } s \wedge j > \text{len } s \rightarrow s(i, \dots, \text{len } s),$
 $i > \text{len } s \vee j = 0 \rightarrow [])$

annotations The function *subseq* is used to select a subsequence of a string, even when the start and end indexes are outside the length of the string (the standard VDM subsequence operator has an over-strong pre-condition for the purposes of this definition).

The first index of the abstract string must not be negative, because Modula-2 start indexes have a minimum ordinal value of zero, and will always be incremented before a call of *subseq*.

$obtain\text{-}string : \text{Variable} \rightarrow \text{Environment} \xrightarrow{o} \text{char}^*$

$obtain\text{-}string(svar)\rho \triangleq$
 184 def $sval = \text{value-of-a-variable}(svar) \rho$;
 471 return $\text{pure-string}(sval)$

annotations The value in the concrete string variable is obtained. Concrete string variables are array variables whose values are a map. The auxiliary function *value-of-a-variable* returns a value of type *String-value*, which is a mapping from ordinal values to characters. The range of this mapping may contain a string terminator and undefined values which the array value contains. The use of *pure-string* removes these values, leaving a VDM character sequence which represents (the abstract) string value.

$\text{pure-string} : \text{String-value} \rightarrow \text{Char}^*$

$\text{pure-string}(sval) \triangleq$
 ?? let $str = \text{map-to-seq}(sval)$ in
 if END-OF-STRING-CHAR \in elems str
 then let $head \curvearrowright tail = str$ be st END-OF-STRING-CHAR \notin elems $head$ in
 $head$
 else str

annotations This function strips any string terminator and any character values after the first string terminator found from a sequence obtained from the concrete string value (i.e. the character array value).

9.5 The Module SysClock

The module `SysClock` provides facilities for accessing a system date and time of day clock.

9.5.1 The definition module of SysClock

```
DEFINITION MODULE SysClock;

CONST
    maxSecondParts = 999999; (* implementation defined constant *)
    (* The value here implies the clock can deliver microsecond accuracy *)

TYPE
    Month =      [1 .. 12];
    Day =        [1 .. 31];
    Hour =       [0 .. 23];
    Min =        [0 .. 59];
    Sec =        [0 .. 59];
    Fraction = [0 .. maxSecondParts]; (* parts of a second *)
    UTCDiff = [-780 .. 720];          (* Time zone Differential Factor *)
    (* the number of minutes that must be added to local time to obtain UTC. *)
    DateTime =
        RECORD
            year          : CARDINAL;
            month          : Month;
            day           : Day;
            hour           : Hour;
            minute         : Min;
            second         : Sec;
            fractions      : Fraction;
            zone           : UTCDiff;
            SummerTimeFlag : BOOLEAN; (* interpretation: local usage *)
        END;

PROCEDURE CanGetClock () : BOOLEAN;
    (* Tests if a clock can be read *)

PROCEDURE CanSetClock () : BOOLEAN;
    (* Tests if a clock can be set *)

PROCEDURE IsValidDateTime (userData : DateTime) : BOOLEAN;
    (* Tests if the value of userData is a valid *)

PROCEDURE GetClock (VAR userData : DateTime);
    (* Assigns local date and time of day to userData *)

PROCEDURE SetClock (userData : DateTime);
    (* Sets the system clock to the given local date and time *)

END SysClock.
```

NOTES

1 No provision is made for leap seconds.

2 'UTC' is 'Universal Coordinated Time'. This is the correct international designation for what was once called GMT (Greenwich Mean Time.)

3 The Summer Time Flag is present for information only. UTC can always be obtained by subtracting the `UTCDiff` value from the time data, regardless of the value of the Summer time flag. However, its presence does allow programs to know whether or the date and time data represents standard time for that location or Summer time. A program could therefore be written to change the clock to Summer time automatically on a certain date, provided it had not already been changed.

9.5.2 Dynamic semantics

values

$$MAXSECONDPARTS = \dots : \mathbb{N}$$

types

$$Month = \{1, \dots, 12\};$$

$$Day = \{1, \dots, 31\};$$

$$Hour = \{0, \dots, 23\};$$

$$Min = \{0, \dots, 59\};$$

$$Sec = \{0, \dots, 59\};$$

$$Fraction = \{0, \dots, MAXSECONDPARTS\};$$

$$UTCDiff = \{-780, \dots, 720\};$$

$$\begin{aligned} DateTime :: & year : \mathbb{N} \\ & month : Month \\ & day : Day \\ & hour : Hour \\ & minute : Min \\ & second : Sec \\ & fraction : Fraction \\ & zone : UTCDiff \\ & summertimeflag : \mathbb{B} \end{aligned}$$

state *Termination of*
clock : *DateTime*

473 *inv mk-DateTime(year, month, day, -, -, -, -, -) \triangleq is-valid-date (year, month, day)*
end

$$is-valid-date : \mathbb{N} \times Month \times Day \rightarrow \mathbb{B}$$

$$\begin{aligned} is-valid-date(y, m, d) \triangleq & \\ & (m \in \{1, 3, 5, 7, 8, 10, 12\} \wedge d \in \{1, \dots, 31\}) \vee \\ & (m \in \{4, 6, 9, 11\} \wedge d \in \{1, \dots, 30\}) \vee \\ & (m = 2 \wedge \neg is-leap-year(y) \wedge d \in \{1, \dots, 28\}) \vee \\ & (m = 2 \wedge is-leap-year(y) \wedge d \in \{1, \dots, 29\}) \end{aligned}$$

$$is-leap-year : \mathbb{N} \rightarrow \mathbb{B}$$

$$\begin{aligned} is-leap-year(y) \triangleq & \\ & y \bmod 4 = 0 \wedge y \bmod 400 \neq 0 \end{aligned}$$

On a processor where ‘`MAX(CARDINAL)`’ is larger than 2048, the value of the **year** field shall directly encode the number of the year. On processors where the range of **CARDINAL** is insufficient to achieve this, the interpretation of the value of this field and its mapping to some range of actual years shall be implementation defined.

The procedure `CanGetClock`

A call of **CanGetClock** shall return **TRUE** if there is a system clock and the program is permitted to read it and shall return **FALSE** otherwise.

CanGetClock : $\xrightarrow{o} \mathbb{B}$
CanGetClock () \triangle
 is not yet defined

The procedure **CanSetClock**

A call of **CanSetClock** shall return **TRUE** if there is a system clock and the program is permitted to set it and shall return **FALSE** otherwise.

CanSetClock : $\xrightarrow{o} \mathbb{B}$
CanSetClock () \triangle
 is not yet defined

The procedure **IsValidClockData**

A call of **IsValidClockData** shall return **TRUE** if the value of type **ClockData** given as the actual parameter represents a valid date and time and shall return **FALSE** otherwise.

IsValidClockData : *Arguments* \rightarrow *Environment* $\xrightarrow{o} \mathbb{B}$
IsValidClockData (*args*) \triangle
 let [*mk-DateTime* (*year*, *month*, *day*, -, -, -, -, -)] = *args* in
 473 return *is-valid-date* (*year*, *month*, *day*)

The procedure **GetClock**

For each field for which information is available, a call of **GetClock** shall assign a value corresponding to the current date and time to the variable given as the actual parameter. The remaining fields shall be set to zero, where this is a valid value, and otherwise shall be set to the lower bound of the range of allowed values.

GetClock : *Arguments* \times *Environment* \xrightarrow{o} *Value*
GetClock (*args*) \triangle
 let [*loc*] = *args* in
 112 assign(*loc*, *clock*) ρ

The procedure **SetClock**

If the program may set the system clock, and the value of type **Clockdata**, given as an actual parameter, represents a valid date and time, a call of **SetClock** shall set the system clock using that date and time. If the program cannot set the system clock, a call of **SetClock** shall have no effect.

SetClock : *Arguments* \times *Environment* \xrightarrow{o} ()
SetClock (*args*) \triangle
 474 if *CanSetClock*()
 then let [*val*] = *args* in
 clock := *val*
 else skip

NOTE — The effect of a call of **SetClock** is not defined if an invalid date and time is given and it is permitted to set the clock.

ANNEX A: Collected Modula-2 Concrete Syntax

A.1 Programs and Compilation Units

A.1.1 Programs

compilation unit = program module | definition module | implementation module ;

A.1.2 Program Modules

program module = "MODULE", module identifier, [protection], ";", import lists, module block, module identifier, "." ;

module identifier = identifier ;

protection = left bracket, constant expression, right bracket ;

constant expression = expression ;

A.1.3 Definition Modules

definition module = "DEFINITION", "MODULE", module identifier, ";", import lists, definitions, "END", module identifier, "." ;

A.1.4 Implementation Modules

implementation module =
"IMPLEMENTATION", "MODULE", module identifier, [protection], ";", import lists, module block, module identifier, "." ;

A.1.5 Module Blocks

module block = declarations, ["BEGIN", statement sequence], "END" ;

A.1.6 Import Lists

import lists = { import list } ;

import list = simple import | unqualified import ;

A.1.6.1 Simple Imports

simple import = "IMPORT", module identifier list, ";" ;

module identifier list = identifier list ;

A.1.6.2 Unqualified Imports

unqualified import = "FROM", module identifier, "IMPORT", identifier list, ";" ;

A.1.7 Export Lists

export list = unqualified export | qualified export ;

A.1.7.1 Unqualified Exports

unqualified export = "EXPORT", identifier list, ";" ;

A.1.7.2 Qualified Exports

qualified export = "EXPORT", "QUALIFIED", identifier list, ";" ;

A.2 Definitions and Declarations

A.2.1 Definitions

definitions = { definition } ;

definition =
"CONST", { constant declaration, ";" } |
"TYPE", { type definition, ";" } |
"VAR", { variable declaration, ";" } |
procedure heading, ";" ;

type definition = type declaration | opaque type definition ;

procedure heading = proper procedure heading | function procedure heading ;

opaque type definition = identifier ;

A.2.2. Declarations

declarations = { declaration } ;

declaration =
"CONST", { constant declaration, ";" } |
"TYPE", { type declaration, ";" } |
"VAR", { variable declaration, ";" } |
procedure declaration, ";" |
module declaration, ";" ;

A.2.3 Constant Declarations

constant declaration = identifier, "=", constant expression ;

A.2.4 Type Declarations

type declaration = identifier, "=", type ;

A.2.5 Variable Declarations

variable declaration = variable identifier list, ":", type ;

variable identifier list = identifier, [machine address], { ",", identifier, [machine address] } ;

machine address = left bracket, value of machine address or address type, right bracket ;

value of machine address or address type = constant expression ;

A.2.6 Types

type = type identifier | new type ;

type identifier = qualified identifier ;

A.2.7 New Types

new type = enumeration type | subrange type | set type | pointer type | procedure type | array type | record type ;

A.2.7.1 Enumeration Types

enumeration type = "(" identifier list, ")" ;

identifier list = identifier, { ",", identifier } ;

A.2.7.2 Subrange Types

subrange type = [range type], left bracket, constant expression, "...", constant expression, right bracket ;

range type = ordinal type identifier ;

CHANGE — A subrange of the host type may be specified.

A.2.7.3 Set Types

set type = "SET", "OF", base type ;

base type = ordinal type ;

ordinal type = ordinal type identifier | enumeration type | subrange type ;

ordinal type identifier = type identifier ;

A.2.7.4 Pointer Types

pointer type = "POINTER", "TO", bound type ;

bound type = type ;

A.2.7.5 Procedure Types

procedure type = proper procedure type | function procedure type ;

proper procedure type = "PROCEDURE", [formal parameter type list] ;

function procedure type = "PROCEDURE", formal parameter type list, function result type ;

function result type = ":", type identifier ;

formal parameter type list = "(", [formal parameter type, { ",", formal parameter type }], ")" ;

formal parameter type = variable formal type | value formal type ;

variable formal type = "VAR", formal type ;

value formal type = formal type ;

formal type = type identifier | open array formal type ;

open array formal type = "ARRAY", "OF", { "ARRAY", "OF" }, type identifier ;

CHANGE — This International Standard permits multidimensional open array parameters.

A.2.7.6 Array Types

array type = "ARRAY", index type, { ",", index type }, "OF", component type ;

index type = ordinal type ;

component type = type ;

A.2.7.7 Record Types

record type = "RECORD", [field list], "END" ;

field list = fields, { ";", fields } ;

fields = fixed fields | variant fields ;

fixed fields = identifier list, ":", type ;

variant fields = "CASE", [tag identifier], ":", tag type, "OF", variant list, "END" ;

tag identifier = ordinal type identifier ;

tag type = ordinal type ;

variant list = variant, { case separator, variant }, ["ELSE", field list], ;

variant = [variant label list, ":", field list] ;

variant label list = variant label, { ",", variant label } ;

variant label = constant expression, ["...", constant expression] ;

A.2.8 Procedure Declarations

procedure declaration = proper procedure declaration | function procedure declaration ;

A.2.8.1 Proper Procedure Declarations

proper procedure declaration =
 proper procedure heading, ":",
 (proper procedure block, procedure identifier | "FORWARD") ;

procedure identifier = identifier ;

A.2.8.2 Proper Procedure Headings

proper procedure heading = "PROCEDURE", procedure identifier, [formal parameter list] ;

formal parameter list = "(", [formal parameter, { ";", formal parameter }], ")" ;

A.2.8.3 Proper Procedure Block

proper procedure block = declarations, ["BEGIN", statement sequence], "END" ;

A.2.8.4 Function Procedure Declarations

function procedure declaration =
function procedure heading, ":",
(function procedure block, procedure identifier | "FORWARD") ;

A.2.8.5 Function Procedure Headings

function procedure heading = "PROCEDURE", procedure identifier, formal parameter list, function result type ;

A.2.8.6 Function Procedure Block

function procedure block = declarations, "BEGIN", statement sequence, "END" ;

A.2.8.7 Parameters

formal parameter = value parameter specification | variable parameter specification ;

value parameter specification = identifier list, ":", formal type ;

variable parameter specification = "VAR", identifier list, ":", formal type ;

A.2.9 Module Declarations

module declaration =
"MODULE", module identifier, [protection], ":",
{ import lists }, [export list], module block, module identifier, "." ;

A.3 Statements

statement =

empty statement	assignment statement	procedure call	return statement	
with statement	if statement	case statement	while statement	
repeat statement	loop statement	exit statement	for statement ;	

A.3.1 Statement Sequences

statement sequence = statement, { ";", statement } ;

A.3.2 Empty Statements

empty statement = ;

A.3.3 Assignment Statements

assignment statement = variable designator, ":", expression ;

A.3.4 Procedure Calls

procedure call = procedure designator, [actual parameters] ;

procedure designator = value designator ;

actual parameters = "(" , [actual parameter list] , ")" ;

actual parameter list = actual parameter, { " , ", actual parameter } ;

actual parameter = variable designator | expression | type parameter ;

type parameter = type identifier ;

A.3.5 Return Statements

return statement = simple return statement | function return statement ;

simple return statement = "RETURN" ;

function return statement = "RETURN", expression ;

A.3.6 With Statements

with statement = "WITH", record designator, "DO", statement sequence, "END" ;

record designator = variable designator | value designator ;

A.3.7 If Statements

if statement = guarded statements, ["ELSE", statement sequence], "END" ;

guarded statements =
"IF", Boolean expression, "THEN", statement sequence,
{ "ELSIF", Boolean expression, "THEN", statement sequence } ;

Boolean expression = expression ;

A.3.8 Case Statements

case statement = "CASE", case selector, "OF", case list, "END" ;

case selector = ordinal expression ;

case list = case, { case selector, case }, ["ELSE", statement sequence], ;

case = [case label list, ":", statement sequence] ;

case label list = case label, { ",", case label } ;

case label = constant expression, ["..", constant expression] ;

A.3.9 While Statements

while statement = "WHILE", Boolean expression, "DO", statement sequence, "END" ;

A.3.10 Repeat Statements

repeat statement = "REPEAT", statement sequence, "UNTIL", Boolean expression ;

A.3.11 Loop Statements

loop statement = "LOOP", statement sequence, "END" ;

A.3.12 Exit Statement

exit statement = "EXIT" ;

A.3.13 For Statements

for statement =
"FOR", control variable identifier, ":", initial value, "TO", final value, ["BY", step size], "DO", statement sequence, "END" ;

control variable identifier = identifier ;

initial value = ordinal expression ;

final value = ordinal expression ;

step size = constant expression ;

A.4. Variable Designators

variable designator = entire designator | indexed designator | selected designator | dereferenced designator ;

A.4.1 Entire Designators

entire designator = qualified identifier ;

A.4.2 Indexed Designators

indexed designator = array designator, "[", index expression, { ",", index expression }, "]" ;

array designator = variable designator ;

index expression = ordinal expression ;

ordinal expression = expression ;

A.4.3 Selected Designators

selected designator = record designator, ".", field identifier ;

field identifier = identifier ;

A.4.4 Deferred Designators

dereferenced designator = pointer designator, dereferencing operator ;

pointer designator = variable designator ;

A.5 Expressions

expression = simple expression, [relational operator, simple expression] ;

simple expression = [sign], term, { term operator, term } ;

term = factor, { factor operator, factor } ;

factor = "(", expression, ")" | not operator, factor | value designator | function call | value constructor | constant literal ;

A.5.1 Infix Expressions

factor operator = "*" | "/" | "REM" | "DIV" | "MOD" | and operator ;

and operator = "AND" | "&" ;

term operator = "+" | "-" | "OR" ;

relational operator = "=" | inequality operator | "<" | ">" | "<=" | ">=" | "IN" ;

inequality operator = "<>" | "#" ;

A.5.2 Prefix Expressions

sign = ["+" | "-"] ;

not operator = "NOT" | "^" ;

A.5.3 Value Designator

value designator = entire value | indexed value | selected value | dereferenced value ;

A.5.3.1 Entire Values

entire value = qualified identifier ;

A.5.3.2 Indexed Values

indexed value = array value, left bracket, index expression, { ",", index expression }, right bracket ;

array value = value designator ;

A.5.3.3 Selected Values

selected value = record value, ".", field identifier ;

record value = value designator ;

A.5.3.4 Dereferenced Values

dereferenced value = pointer value, dereferencing operator ;

pointer value = value designator ;

A.5.4 Function Call

function call = function designator, "(", [actual parameter list], ")" ;

function designator = value designator ;

A.5.5 Value Constructors

value constructor = array constructor | record constructor | set constructor ;

A.5.5.1 Array Constructor

array constructor = type identifier, array constructed value ;

array constructed value = left brace, repeated component, { ",", repeated component }, right brace ;

repeated component = component of structure, ["BY", repetition factor] ;

repetition factor = constant expression ;

component of structure = expression | array constructed value | record constructed value | set constructed value ;

A.5.5.2 Record Constructors

record constructor = type identifier, record constructed value ;

record constructed value = left brace, [repeated component, { ",", repeated component }], right brace ;

A.5.5.3 Set Constructors

set constructor = type identifier, set constructed value ;

set constructed value = left brace, [member, { ",", member }], right brace ;

member = interval | singleton ;

interval = ordinal expression, "...", ordinal expression ;

singleton = ordinal expression ;

A.5.6 Constant Literals

constant literal = whole number literal | real literal | string literal | character literal ;

A.5.6.1 String Literals

string literal = single string, { concatenate symbol, single string } ;

single string = quoted string | string identifier ;

string identifier = qualified identifier ;

A.5.6.2 Character Literals

character literal = quoted character | character number literal ;

A.6 Qualified Identifiers

qualified identifier = identifier, { ".", identifier } ;

A.7 Lexical components

quoted string = ? as defined in the Lexis, Clause 5 ? ;

dereferencing operator = ? as defined in the Lexis, Clause 5 ? ;

character number literal = ? as defined in the Lexis, Clause 5 ? ;

concatenate symbol = ? as defined in the Lexis, Clause 5 ? ;

whole number literal = ? as defined in the Lexis, Clause 5 ? ;

case separator = ? as defined in the Lexis, Clause 5 ? ;

identifier = ? as defined in the Lexis, Clause 5 ? ;

real literal = ? as defined in the Lexis, Clause 5 ? ;

left brace = ? as defined in the Lexis, Clause 5 ? ;

right brace = ? as defined in the Lexis, Clause 5 ? ;

quoted character = ? as defined in the Lexis, Clause 5 ? ;

left bracket = ? as defined in the Lexis, Clause 5 ? ;

right bracket = ? as defined in the Lexis, Clause 5 ? ;

ANNEX B: Collected Modula-2 Abstract Syntax

Program :: *pmod* : *Program-module*
 dmods : *Definition-modules*
 imods : *Implementation-modules*

Definition-modules = *Definition-module-set*

Implementation-modules = *Implementation-module-set*

Compilation-unit = *Program-module* | *Definition-module* | *Implementation-module*

Program-module :: *name* : *Identifier*
 imports : *Import-lists*
 block : *Module-block*
 protection : [*Expression*]

Definition-module :: *name* : *Identifier*
 imports : *Import-lists*
 defs : *Definitions*

Implementation-module :: *name* : *Identifier*
 imports : *Import-lists*
 block : *Module-block*
 protection : [*Expression*]

Module-block :: *decls* : *Declarations*
 actions : *Block-body*

Block-body = *Statement**

Import-lists = *Import-list**

Import-list = *Simple-import* | *Unqualified-import*

Simple-import :: *ids* : *Identifier**

Unqualified-import :: *from* : *Identifier*
 ids : *Identifier-set*

Export-list = *Unqualified-export* | *Qualified-export*

Unqualified-export :: *ids* : *Identifier-set*

The number of elements in the set of identifiers of the abstract representation of an unqualified export shall be equal to the number of identifiers in the identifier list of the concrete syntax.

Qualified-export :: *ids* : *Identifier-set*

The number of elements in the set of identifiers of the abstract representation of an qualified export shall be equal to the number of identifiers in the identifier list of the concrete syntax.

Module-order = *Compilation-unit**

*Definitions = Definition**

Definition = Constant-declaration | Type-definition | Variable-declaration | Procedure-heading

Procedure-heading = Proper-procedure-heading | Function-procedure-heading

Type-definition = Type-declaration | Opaque-type-definition

Opaque-type-definition :: id : Identifier

*Declarations = Declaration**

*Declaration = Constant-declaration | Type-declaration | Variable-declaration
| Procedure-declaration | Module-declaration*

*Constant-declaration :: id : Identifier
expr : Expression*

*Type-declaration :: id : Identifier
type : Type*

*Variable-declaration :: ids : Variable-identifier-set
type : Type*

*Variable-identifier :: id : Identifier
addr : [Expression]*

Procedure-declaration = Proper-procedure-declaration | Function-procedure-declaration

*Proper-procedure-declaration :: head : Proper-procedure-heading
block : Proper-procedure-block*

*Proper-procedure-heading :: name : Identifier
parms : Formal-parameter-list*

*Formal-parameter-list = Formal-parameter**

*Proper-procedure-block :: decls : Declarations
actions : Block-body*

*Function-procedure-declaration :: head : Function-procedure-heading
block : Function-procedure-block*

*Function-procedure-heading :: name : Identifier
parms : Formal-parameter-list
return : Type-identifier*

*Function-procedure-block :: decls : Declarations
actions : Block-body*

Formal-parameter = Value-parameter-specification | Variable-parameter-specification

$$\begin{array}{l} \textit{Array-type} :: \textit{itype} : \textit{Type} \\ \phantom{\textit{Array-type}} \phantom{\textit{itype}} : \textit{ctype} : \textit{Type} \end{array}$$

Record-type :: *fields* : *Field-list*

Field-list = *Fields**

Fields = *Fixed-fields* | *Variant-fields*

Fixed-fields :: *ids* : *Identifier**
 type : *Type*

Variant-fields :: *tag* : [*Identifier*]
 tagt : *Qualident*
 variants : *Variant**
 elsep : [*Field-list*]

Variant :: *labels* : *Set-definition*
 fields : *Field-list*

Statement = *Empty-statement*
 | *Assignment-statement*
 | *Procedure-call*
 | *Return-statement*
 | *With-statement*
 | *If-statement*
 | *Case-statement*
 | *While-statement*
 | *Repeat-statement*
 | *Loop-statement*
 | *Exit-statement*
 | *For-statement*

Statement-sequence :: *actions* : *Statement*⁺

Empty-statement ::

Assignment-statement :: *desig* : *Variable-designator*
 expr : *Expression*

Procedure-call :: *desig* : *Procedure-designator*
 aps : *Actual-parameters*

Procedure-designator = *Value-designator*
 | *Standard-procedure*
 | *Coroutine-procedure*

Standard-procedure :: *id* : *Identifier*
 desig : *Standard-procedure-designator*

Coroutine-procedure :: *qid* : *Qualident*
 desig : *Coroutine-procedure-designator*

Return-statement = *Simple-return-statement* | *Function-return-statement*

Simple-return-statement :: RETURN

Function-return-statement :: *type* : *Typed*
 expr : *Expression*

With-statement :: *desig* : *Record-designator*
 body : *Statement-sequence*

Record-designator = *Variable-designator* | *Value-designator*

If-statement :: *thens* : *Guarded-statement**
 elsep : [*Statement-sequence*]

Guarded-statement :: *guard* : *Expression*
 body : *Statement-sequence*

Case-statement :: *expr* : *Expression*
 cases : *Case-set*
 elsep : [*Statement-sequence*]

Case :: *labels* : *Set-definition*
 body : *Statement-sequence*

While-statement :: *expr* : *Expression*
 body : *Statement-sequence*

Repeat-statement :: *body* : *Statement-sequence*
 expr : *Expression*

Loop-statement :: *body* : *Statement-sequence*

Exit-statement :: EXIT

For-statement :: *id* : *Identifier*
 initial : *Expression*
 final : *Expression*
 step : *Expression*
 body : *Statement-sequence*

Variable-designator = *Entire-designator*
 | *Indexed-designator*
 | *Selected-designator*
 | *Dereferenced-designator*

Entire-designator :: *qid* : *Qualident*

Indexed-designator :: *desig* : *Variable-designator*
 expr : *Expression*

Selected-designator :: *desig* : *Variable-designator*
 id : *Identifier*

Dereferenced-designator :: *desig* : *Variable-designator*

Expression = *Infix-expression*
 | *Prefix-expression*
 | *Value-designator*
 | *Function-call*
 | *Value-constructor*
 | *Constant-literal*

Infix-expression :: *left* : *Expression*
 op : *Infix-operation*
 right : *Expression*

Infix-operation = *Complex-number-operation*
 | *Real-number-operation*
 | *Whole-number-operation*
 | *Boolean-operation*
 | *Set-operation*
 | *Relational-operation*

Complex-number-operation :: *op* : *Complex-number-operator*
 otype : *Complex-number-type*

Complex-number-operator = ADD | SUBTRACT | MULTIPLY | DIVIDE

Real-number-operation :: *op* : *Real-number-operator*
 otype : *Real-number-type*

Real-number-operator = ADD | SUBTRACT | MULTIPLY | DIVIDE

Whole-number-operation :: *op* : *Whole-number-operator*
 otype : *Whole-number-type*

Whole-number-operator = ADD
 | SUBTRACT
 | MULTIPLY
 | DIVIDE
 | REM
 | DIV
 | MOD

Boolean-operation :: *op* : *Boolean-operator*

Boolean-operator = AND | OR

Set-operation :: *op* : *Set-operator*
 otype : *Typed*

Set-operator = UNION | DIFFERENCE | INTERSECTION | SYMMETRIC-DIFFERENCE

Relational-operation = *Complex-comparison-operation*
 | *Scalar-relational-operation*
 | *Set-relational-operation*
 | *Procedure-relational-operation*
 | *Pointer-relational-operation*
 | *Protection-relational-operation*

Complex-comparison-operation :: *op* : *Complex-comparison-operator*

Complex-comparison-operator = EQ | NE

Scalar-relational-operation :: *op* : *Scalar-relational-operator*

Scalar-relational-operator = EQ | NE | LT | GT | LE | GE

Set-relational-operation :: *op* : *Set-relational-operator*

Set-relational-operator = EQ | NE | SUPERSET | SUBSET | IN

Procedure-relational-operation :: *op* : *Procedure-relational-operator*

Procedure-relational-operator = EQ | NE

Pointer-relational-operation :: *op* : *Pointer-relational-operator*

Pointer-relational-operator = EQ | NE

Protection-relational-operation :: *op* : *Protection-relational-operator*

Protection-relational-operator = EQ | NE | LE | GE

Prefix-expression :: *op* : *Prefix-operation*
 expr : *Expression*

Prefix-operation = *Arithmetic-prefix-operation* | *Boolean-prefix-operation*

Arithmetic-prefix-operation :: *op* : *Arithmetic-prefix-operator*
 otype : *Number-type*

Arithmetic-prefix-operator = PLUS | MINUS

Boolean-prefix-operation :: *op* : *Boolean-prefix-operator*

Boolean-prefix-operator = NOT

Value-designator = *Entire-value*
 | *Indexed-value*
 | *Selected-value*
 | *Dereferenced-value*

Entire-value :: *qid* : *Qualident*

Indexed-value :: *desig* : *Value-designator*
 expr : *Expression*

Selected-value :: *desig* : *Value-designator*
 id : *Identifier*

Dereferenced-value :: *desig* : *Value-designator*

Function-call :: *desig* : *Function-designator*
 args : *Actual-parameters*

Function-designator = *Value-designator*
 | *Standard-function*
 | *System-function*
 | *Coroutine-function*

Standard-function :: *id* : *Identifier*
 desig : *Standard-function-designator*

System-function :: *qid* : *Qualident*
 desig : *System-function-designator*

Coroutine-function :: *qid* : *Qualident*
 desig : *Coroutine-function-designator*

Value-constructor = *Array-constructor* | *Record-constructor* | *Set-constructor*

Array-constructor :: *qid* : *Qualident*
 def : *Array-definition*

Array-definition :: *vals* : *Repeated-elements*

Repeated-elements = *Repeated-element**

Repeated-element :: *val* : *Element*
 by : *Expression*

Element = *Expression* | *Array-definition* | *Record-definition* | *Set-definition*

Record-constructor :: *qid* : *Qualident*
 def : *Record-definition*

Record-definition :: *vals* : *Elements*

Elements = *Element**

Set-constructor :: *qid* : *Qualident*
 def : *Set-definition*

Set-definition = *Member-set*

Member = *Interval* | *Singleton*

Interval :: *min* : *Expression*
 max : *Expression*

Singleton :: *expr* : *Expression*

Constant-literal = *Whole-number-literal* | *Real-literal* | *String-literal* | NIL-VALUE

Whole-number-literal :: *value* : \mathbb{N}

Real-literal :: *value* : \mathbb{R}

String-literal = *Single-string**

Single-string = *Character-string* | *String-identifier*

Character-string :: *val* : *Char**

String-identifier :: *qid* : *Qualident*

Actual-parameters = *Actual-parameter**

Actual-parameter = *Variable-designator* | *Expression* | *Type-parameter*

Arguments = *Argument**

Argument = *Variable* | *Value* | *Typed*

Type-parameter :: *type* : *Qualident*

Standard-procedure-designator = *Dec-designator*
| *Dispose-designator*
| *Enter-designator*
| *Excl-designator*
| *Halt-designator*
| *Inc-designator*
| *Incl-designator*
| *Leave-designator*
| *New-designator*

Dec-designator :: *rtype* : *Typed*

Dispose-designator :: *rtype* : *Typed*

Enter-designator ::

Excl-designator :: *rtype* : *Typed*

Halt-designator ::

Inc-designator :: *rtype* : *Typed*

Incl-designator :: *rtype* : *Typed*

Leave-designator ::

New-designator :: *rtype* : *Typed*

Standard-function-designator = *Abs-designator*
 | *Cap-designator*
 | *Chr-designator*
 | *Cmplx-designator*
 | *Float-designator*
 | *High-designator*
 | *Im-designator*
 | *Int-designator*
 | *Length-designator*
 | *Lfloat-designator*
 | *Max-designator*
 | *Min-designator*
 | *Odd-designator*
 | *Ord-designator*
 | *Prot-designator*
 | *Re-designator*
 | *Size-designator*
 | *Trunc-designator*
 | *Val-designator*

Abs-designator :: *rtype* : *Expression-typed*

Cap-designator ::

Chr-designator ::

Cmplx-designator :: *rtype* : *Expression-typed*

Float-designator ::

High-designator ::

Im-designator :: *rtype* : *Expression-typed*

Int-designator ::

Length-designator ::

Lfloat-designator ::

Max-designator ::

Min-designator ::

Odd-designator ::

Ord-designator ::

Prot-designator ::

Re-designator :: *rtype* : *Expression-typed*

Size-designator :: *type* : *Expression-typed*

Trunc-designator ::

Val-designator ::

Environment :: *consts* : *Constants*
 types : *Types*
 strucs : *Structures*
 vars : *Variables*
 procs : *Procedures*
 mods : *Modules*
 level : \mathbb{N}
 domain : *[Protection-domain]*
 dens : *[Procdens]*
 conts : *[Continuation]*

Constants = *Identifier* \xrightarrow{m} *Constant-value*

Types = *Identifier* \xrightarrow{m} *Typed*

Structures = *Type-name* \xrightarrow{m} *Structure*

Variables = *Identifier* \xrightarrow{m} (*Variable-typed* | *Variable*)

Procedures = *Identifier* \xrightarrow{m} *Procedure-id*

Modules = *Identifier* \xrightarrow{m} *Module-environment*

Procdens = *Procedure-id* \xrightarrow{m} (*Procedure-typed* | *Program-cont*)

Continuation = *Exit* \xrightarrow{m} *Program-cont*

Constant-value :: *type* : *Expression-typed*
 value : *Value*

Typed = *Basic-type* | *Type-name* | *System-storage-type*

Type-name = *proc-type* | *TOKEN*

Basic-type = *Number-type* | *BOOLEAN-TYPE* | *CHARACTER-TYPE* | *NIL-TYPE* | *PROTECTION-TYPE* | *COROUTINE-TYPE*

Number-type = *Whole-number-type* | *Real-number-type*

Whole-number-type = *UNSIGNED-TYPE* | *SIGNED-TYPE* | *Z-TYPE*

Real-number-type = *REAL-TYPE* | *LONG-REAL-TYPE* | *R-TYPE*

String-type :: *length* : \mathbb{N}

System-storage-type = *LOC-TYPE* | *MACHINE-ADDRESS-TYPE* | *BITNUM-TYPE*

Structure = *Array-structure* | *Record-structure* | *Procedure-structure* | *Set-structure*
 | *Pointer-structure* | *OPAQUE* | *Enumerated-structure* | *Subrange-structure*

Array-structure :: *itype* : *Typed*
 ctype : *Typed*

Record-structure :: *fields* : *Fields-list-structure*

Fields-list-structure = *Fields-structure**

Fields-structure = *Fixed-fields-structure* | *Variant-fields-structure*

Fixed-fields-structure :: *ids* : *Identifier**
 type : *Typed*

Variant-fields-structure :: *tag* : [*Identifier*]
 tagt : *Typed*
 variants : *Variant-structure**
 other : *Fields-list-structure*

Variant-structure :: *labels* : *Set-value*
 fields : *Fields-list-structure*

Procedure-structure = *Proper-procedure-structure* | *Function-procedure-structure*

Proper-procedure-structure :: *parms* : *Formal-parameters-typed*

Function-procedure-structure :: *parms* : *Formal-parameters-typed*
 return : [*Typed*]

Formal-parameters-typed = *Formal-parameter-typed**

Formal-parameter-typed = *Value-formal-typed* | *Variable-formal-typed*

Value-formal-typed :: *type* : *Variable-typed*

Variable-formal-typed :: *type* : *Variable-typed*

Set-structure :: *btype* : *Typed*

Pointer-structure :: *type* : [*Typed*]

Enumerated-structure :: *values* : *Identifier**

Subrange-structure :: *rtype* : *Typed*
 range : *Set-value*

Value-structure = *Structure* | *Open-array-typed* | *Procedure-typed* | *String-type*

Variable-typed = [*Typed*] | *Open-array-typed*

Open-array-typed :: *type* : *Component-typed*

*Component-typed*s = *Variable-typed*

Expression-typed = *Typed* | *String-type* | *Procedure-typed* | *Open-array-typed*

Procedure-typed = *Procedure-const* | *Standard-procedure*

Procedure-const :: *proc* : *Procedure-structure*
level : \mathbb{N}

Standard-procedure = STANDARD-PROPER-PROCEDURE | STANDARD-FUNCTION-PROCEDURE

Module-environment = *Unqualified-environment* | *Qualified-environment*

Unqualified-environment :: *env* : *Environment*

Qualified-environment :: *env* : *Environment*

Variable = *Elementary-variable* | *Array-variable* | *Record-variable* | *Tag-variable* | *Variant-variable* | *Tagged-variable*

Loc = An infinite set of tokens

Elementary-variable :: *loc* : *Loc*
range : [*Set-value*]

Array-variable :: *vars* : *Array-components*

Array-components = *Ordinal-value* \xrightarrow{m} *Variable*

inv ac \triangleq
 $\forall a \in ac \cdot$
 $\forall rx, ry \in \text{rng } a \cdot$
is-same-variable-type(*rx*, *ry*)

Record-variable :: *fields* : *Record-components*

Record-components = *Identifier* \xrightarrow{m} *Variable*

Tag-variable :: *var* : *Elementary-variable*
variants : *Variant-component-set*

Variant-variable :: *var* : *Variable*
tagloc : *Elementary-variable*
tagvals : *Value-set*

Tagged-variable :: *var* : *Variable*
tagvar : *Tag-variable*
tagvals : *Value-set*

Variant-component :: *labels* : *Value-set*
fields : *Record-components*

Protection-domain :: *entry* : *Program-cont*
leave : *Program-cont*
val : *Value*

Procedure-id = *TOKEN*

Program-cont = *Cmdcont* \rightarrow *Coroutine-env* \rightarrow *Statecont*

Exit = *RETURN* | *TERMINATION* | *HALT*

Environment :: *env* : *Associations*
 strucs : *Structures*

Associations = *Static-environment* | *Dynamic-environment*

Structures = *Type-name* \xrightarrow{m} *Structure*

Static-environment :: *ids* : *Identifier* \xrightarrow{m} *Property*
 level : $[\mathbb{N}]$

Property = *Constant-value* | *Typed* | *Variable-typed* | *Procedure-typed* | *Environment*

Dynamic-environment :: *consts* : *Identifier* \xrightarrow{m} *Object*
 domain : *Protection-domain*
 dens : *Procdens*
 conts : *Continuation*

Object = *Constant-value* | *Typed* | *Variable* | *Procedure-id* | *Environment*

Procdens = *Procedure-id* \xrightarrow{m} *Program-cont*

Continuation = *Exit* \xrightarrow{m} *Program-cont*

Constant-value :: *type* : *Expression-typed*
 value : *Value*

Typed :: *id* : *Types*

Types = *Basic-type* | *Type-name* | *System-storage-type* | *proc-type*

Type-name = *TOKEN*

Basic-type = *Number-type* | *Boolean-type* | *character-type* | *nil-type* | *protection-type* | *coroutine-type*

Number-type = *Whole-number-type* | *Real-number-type*

Whole-number-type = *unsigned-type* | *signed-type* | \mathbb{Z} -*type*

Real-number-type = *real-type* | *long-real-type* | \mathbb{R} -*type*

String-type :: *length* : \mathbb{N}

System-storage-type = *loc-type* | *machine-address-type* | *bitnum-type*

Structure = *Array-structure* | *Record-structure* | *Procedure-structure* | *Set-structure*
 | *Pointer-structure* | *Opaque-structure* | *Enumerated-structure* | *Subrange-structure*

Array-structure :: *itype* : *Typed*
 ctype : *Typed*

Record-structure :: *fields* : *Fields-list-structure*

Fields-list-structure = *Fields-structure**

Fields-structure = *Fixed-fields-structure* | *Variant-fields-structure*

Fixed-fields-structure :: *ids* : *Identifier**
 type : *Typed*

Variant-fields-structure :: *tag* : [*Identifier*]
 tagt : *Typed*
 variants : *Variant-structure**
 other : *Fields-list-structure*

Variant-structure :: *labels* : *Set-value*
 fields : *Fields-list-structure*

Procedure-structure = *Proper-procedure-structure* | *Function-procedure-structure*

Proper-procedure-structure :: *parms* : *Formal-parameters-typed*

Function-procedure-structure :: *parms* : *Formal-parameters-typed*
 return : [*Typed*]

Formal-parameters-typed = *Formal-parameter-typed**

Formal-parameter-typed = *Value-formal-typed* | *Variable-formal-typed*

Value-formal-typed :: *type* : *Variable-typed*

Variable-formal-typed :: *type* : *Variable-typed*

Set-structure :: *btype* : *Typed*

Pointer-structure :: *type* : [*Typed*]

Opaque-structure :: *opaque* :

Enumerated-structure :: *values* : *Identifier**

Subrange-structure :: *rtype* : *Typed*
 range : *Set-value*

Value-structure = *Structure* | *Open-array-typed* | *Procedure-typed* | *String-type*

Variable-typed = [*Typed*] | *Open-array-typed*

Open-array-typed :: *type* : *Component-typed*

Component-typed = *Variable-typed*

Expression-typed = *Typed* | *String-type* | *Procedure-typed* | *Open-array-type*

Procedure-typed = *Procedure-const* | *Standard-procedure*

Procedure-const :: *proc* : *Procedure-structure*
 level : \mathbb{N}

Standard-procedure = *standard-proper-procedure* | *standard-function-procedure*

Module-environment = *Unqualified-environment* | *Qualified-environment*

Unqualified-environment :: *env* : *Environment*

Qualified-environment :: *env* : *Environment*

Variable = *Elementary-variable* | *Array-variable* | *Record-variable* | *Tag-variable* | *Variant-variable* | *Tagged-variable*

Loc = An infinite set of tokens

Elementary-variable :: *loc* : *Loc*
 range : [*Set-value*]

Array-components = *Ordinal-value* \xrightarrow{m} *Variable*

$\text{inv } ac \triangleq$
 $\forall a \in ac \cdot$
 $\forall rx, ry \in \text{rng } a \cdot$
 $\text{is-same-variable-type}(rx, ry)$

Record-variable :: *fields* : *Record-components*

Record-components = *Identifier* \xrightarrow{m} *Variable*

Tag-variable :: *var* : *Elementary-variable*
 variants : *Variant-component-set*

Variant-variable :: *var* : *Variable*
 tagloc : *Elementary-variable*
 tagvals : *Value-set*

Tagged-variable :: *var* : *Variable*
 tagvar : *Tag-variable*
 tagvals : *Value-set*

Variant-component :: *labels* : *Value-set*
 fields : *Record-components*

Protection-domain :: *entry* : *Program-cont*
 leave : *Program-cont*
 val : *Value*

$Procedure-id = TOKEN$

$Program-cont = Cmdcont \rightarrow Coroutine-env \rightarrow Statecont$

$Exit = return \mid termination \mid halt$

$Object = Typed \mid Structure \mid Variable-typed \mid Variable \mid Procedure-typed \mid Procedure-value \mid Module-environment$

$Storage :: store : Loc \xrightarrow{m} Storable-value$
 $aliases : Loc-pair-set$
 $io : Stream-id \xrightarrow{m} Stream$
 $coroutines : Coroutine-id \xrightarrow{m} Coroutine-storage$
 $handlers : Interrupt-source \xrightarrow{m} Handler$

$Storable-value = Elementary-value \mid UNDEFINED$

$Loc-pair :: first : Loc$
 $second : Loc$

$Elementary-value = Basic-value \mid Enumerated-value \mid Set-value \mid Pointer-value \mid Procedure-value \mid Protection$

$Value = Elementary-value \mid Array-value \mid Record-value$

$Array-value = Ordinal-value \xrightarrow{m} Value$

$Ordinal-value = Number \mid Bool \mid Enumerated-value$

$Record-value = Identifier \xrightarrow{m} Value$

$Basic-value = Number \mid String-value \mid NIL-VALUE \mid \mathbb{B}$

$Number = \mathbb{R}$

$String-value = \mathbb{N} \xrightarrow{m} \text{char}$

$Enumerated-value :: value : Identifier$
 $order : Identifier^*$

$Set-value :: value : Ordinal-value-set$

$Pointer-value :: value : (Loc \mid NIL-VALUE)$

$Procedure-value = Procedure-id$

$Arguments = Value^*$

$Protection = \{\text{INTERRUPTIBLE}, \text{UNINTERRUPTIBLE}\} \cup Other-protection$

$Other-protection = \text{implementation-defined}$

$Interrupt-source = \text{implementation-defined}$

Coroutine-storage :: *protection* : *Protection**
 locs : *Loc-set*

Handler :: *id* : *Coroutine-id*
 from : [*Loc*]

Coroutine-env :: *map* : *Coroutine-id* \xrightarrow{m} *Program-cont*
 caller : *Coroutine-id*

Coroutine-id = token | PROGRAM-set

Program-cont = *Cmdcont* \rightarrow *Coroutine-env* \rightarrow *Statecont*

Cmdcont = *Coroutine-env* \rightarrow *Statecont*

Statecont = *State* \rightarrow *Answer*

Answer = *Stream-id* \xrightarrow{m} *Stream*

Language-exception = *Must-detect-exception* | *May-detect-exception*

Must-detect-exception = ASSIGN-RANGE
 | CASE-RANGE
 | COMPLEX-OVERFLOW
 | COMPLEX-ZERO-DIVISION
 | INDEX-RANGE
 | NIL-DEREFERENCE
 | NOT-ATTACHED
 | PASSIVE-PROGRAM
 | REAL-OVERFLOW
 | REAL-ZERO-DIVISION
 | RETURN-RANGE
 | SET-RANGE
 | SMALL-WORKSPACE
 | TAG-RANGE
 | UNDEFINED-VALUE
 | WHOLE-NONPOS-DIV
 | WHOLE-NONPOS-MOD
 | WHOLE-OVERFLOW
 | WHOLE-ZERO-DIVISION
 | WHOLE-ZERO-REMAINDER

May-detect-exception = INACTIVE-VARIANT
 | NONEXISTENT
 | ADDRESS-ARITHMETIC
 | LOWREALEXCEPTION
 | LOWLONGEXCEPTION
 | PROCESS-ERROR

System-function-designator = *Addadr-designator*
 | *Addressvalue-designator*
 | *Adr-designator*
 | *Cast-designator*
 | *Difadr-designator*
 | *Rotate-designator*
 | *Shift-designator*
 | *Subadr-designator*
 | *Tsize-designator*

Addadr ::

Addressvalue ::

Adr-designator ::

Cast-designator :: *ttype* : *Type*
 stype : *Type*

Difadr-designator ::

Rotate-designator :: *rtype* : *Typed*

Shift-designator :: *rtype* : *Typed*

Subadr-designator ::

Tsize-designator ::


```

 $\rho_{system} = mk\text{-}Environment(
  -- Constants
  either \{ LOCSERBYTE \xrightarrow{m} locsperbyte\text{-}constant\text{-}value \text{ or } \{ \},
    LOCSERWORD \xrightarrow{m} locsperword\text{-}constant\text{-}value,
    BITSERBITSET \xrightarrow{m} bitsperbitset\text{-}constant\text{-}value \},
  -- Types
  either \{ BYTE \xrightarrow{m} \text{BYTE-TYPE} \text{ or } \{ \},
    WORD \xrightarrow{m} \text{WORD-TYPE},
    BITSET \xrightarrow{m} \text{BITSET-TYPE},
    LOC \xrightarrow{m} \text{LOC-TYPE},
    ADDRESS \xrightarrow{m} \text{ADDRESS-TYPE},
    BITNUM \xrightarrow{m} \text{BIT-SET-TYPES} \},
  MACHINEADDRESS \xrightarrow{m} \text{implementation defined structure},
  -- Structures
  \{ \text{BYTE-TYPE} \xrightarrow{m} \text{byte-array-structure},
    \text{WORD-TYPE} \xrightarrow{m} \text{word-array-structure},
    \text{BITSET-TYPE} \xrightarrow{m} \text{bitset-structure},
    \text{BYTE-RANGE} \xrightarrow{m} \text{byte-range-structure},
    \text{ADDRESS-TYPE} \xrightarrow{m} \text{address-type-structure} \},
  \{ \},
  -- Functions
  ADR \xrightarrow{m} \text{SYSTEM-FUNCTION-PROCEDURE},
  ADDADR \xrightarrow{m} \text{SYSTEM-FUNCTION-PROCEDURE},
  ADDRESSVALUE \xrightarrow{m} \text{SYSTEM-FUNCTION-PROCEDURE},
  CAST \xrightarrow{m} \text{SYSTEM-FUNCTION-PROCEDURE},
  DIFADR \xrightarrow{m} \text{SYSTEM-FUNCTION-PROCEDURE},
  ROTATE \xrightarrow{m} \text{SYSTEM-FUNCTION-PROCEDURE},
  SHIFT \xrightarrow{m} \text{SYSTEM-FUNCTION-PROCEDURE},
  SUBADR \xrightarrow{m} \text{SYSTEM-FUNCTION-PROCEDURE},
  TSIZE \xrightarrow{m} \text{SYSTEM-FUNCTION-PROCEDURE},
  -- Modules
  \{ \},
  -- Level
  NIL,
  -- Protectiondomain
  NIL,
  -- Denotations
  \{ \},
  -- Continuations
  \{ \})$ 
```

```

Coroutine-procedure-designator = ATTACH
                                | DETACH
                                | IOTRANSFER
                                | LISTEN
                                | NEWCOROUTINE
                                | TRANSFER

```

```

Coroutine-function-designator = HANDLER
                               | ISATTACHED
                               | SELF

```

Termination-designator = *Termination-procedure-designator* | *Termination-function-designator*

Termination-procedure-designator = REGISTERCLEANUP

$Month = \{1, \dots, 12\}$

$Day = \{1, \dots, 31\}$

$Hour = \{0, \dots, 23\}$

$Min = \{0, \dots, 59\}$

$Sec = \{0, \dots, 59\}$

$Fraction = \{0, \dots, MAXSECONDPARTS\}$

$UTCDiff = \{-780, \dots, 720\}$

$DateTime :: year : \mathbb{N}$
 $month : Month$
 $day : Day$
 $hour : Hour$
 $minute : Min$
 $second : Sec$
 $fraction : Fraction$
 $zone : UTCDiff$
 $summertimeflag : \mathbb{B}$

ANNEX E: Modula-2 Glossary

(Issue 5)

Mark Woodman

Open University,

Computing Department,

Walton Hall,

Milton Keynes,

MK7 6AA

Telephone: (0908 or +44908) 653187

Email: M_WOODMAN@UK.AC.OU.ACSVAX

The following set of definitions may be useful in reading the formal definition of Modula-2. Definitions herein should not, however, be used in preference to those in the Formal Definition. (If definitions in the Glossary conflict with the Formal Definition, they should be changed.)

Many terms are given *special interpretation* in the context of defining Modula-2, so are not as general as in a dictionary, and may well *not* match the reader's own idea of what the term means. (For example, the term *elementary type* has a precise meaning for Modula-2 which can be determined from the Formal Definition but one which is not widely used: a *procedure type* is an elementary type in the Standard, but is described thus in very few text books.) Nevertheless, any comments would be welcome.

TEMPORARY NOTE — N.B. The Glossary in the Standard is an aid to comprehension, not a description of the language. It is intended to be a *small* appendix. The terms which are defined formally will not, in general, appear in the Glossary. Those which are unlikely to appear are marked with a bar.

Notwithstanding the last paragraph, it may make sense to introduce new terms into the glossary if generally accepted concepts become submerged in the formal definition. For example, the concept of *expression compatibility* may be hidden in the specification of a VDM function; the omission of this concept from the Standard would be seen by many (not least teachers) to be unfortunate.

A list of terms which have been removed from previous issues are given at the end of the annex. Contributors of new terms for inclusion are asked to write a short paragraph (30 words or more) on each term which defines it and justifies its inclusion; references to related parts of the formal definition would be appreciated.

Abstract Syntax — The abstract syntax shows the essential structure of Modula-2; it omits details such as keywords of the language, but includes those aspects of syntax which are not included in a concrete syntax but which VDM needs to consider (e.g. distinction between infix and prefix expressions).

Activation — The activation of a procedure precedes the execution of a procedure block when a procedure is called. Activation involves building the environment of a procedure: local variable storage is created and value parameters are evaluated and copied to local storage.

Anonymous type — An anonymous type is one which is not named. Because no identifier exists for such a type, an anonymous type is unique; it cannot be identical to any other type. The following gives examples of anonymous types:

```
VAR x: ARRAY [1..10] OF (no, yes);
```

The array type is anonymous; the index type is anonymous; the component type is anonymous.

Any order of evaluation — See *Order of evaluation*.

■ Assignment compatible — This category of compatibility covers assignment and value-parameter passing. It allows the mixture of types which are identical or expression compatible, the mixture of signed and unsigned whole number types and for the special handling of values of the type $\$$. See formal definition, auxiliary function *is-assignment-compatible*. Base type The ordinal type whose values are the values which a set may contain.

Basic type — The predefined types and the conceptual types; i.e. CHAR, BOOLEAN, the whole number types (INTEGER and CARDINAL), the real types (REAL and LONGREAL), and the types \mathbb{Z} , \mathbb{R} and \mathbb{S} .

Bit string — A sequence of bits.

■Block — The declarations and body parts of a module or procedure declaration. See formal definition, descriptions of module, proper procedure, and function procedure blocks.

■Body — The statement part of a procedure or module block.

Capacity — Number of components in an array type.

■Cast — See *type conversion*.

■Clause — A component of a statement, or an *import clause* or an *export clause*.

Component type — The type of the components of an array structure. For example, in the declaration:

```
TYPE list = ARRAY CHAR OF CARDINAL;
```

The type **CARDINAL** is the component type.

■Compatible — Wirth's term for *Expression Compatible*.

Compatibility — Compatibility defines how types may be used in combination. The classification is as follows (see individual terms for details):

Identical type compatibility

Expression compatibility

Assignment compatibility

Parameter compatibility

Range compatibility

A requirement that two types be identical is a stronger requirement than that they be expression compatible; a requirement that they be expression compatible is a stronger requirement than that they be assignment compatible; parameter compatibility allows for the different rules for actual, variable, open array parameters, and **LOC** parameters. These first four may be checked during compilation. Range compatibility is concerned with the validity of mixing values of subrange types during execution; checks for range compatibility are performed during execution.

Compiler directive — A type of pragma which is an instruction to a compiler (or other type of translator) which effects the behaviour of the compiler. (For example, a compiler directive might switch on optimisation, or switch to non-standard mode.) A directive may be given to a translator independently of the program text, or as a particular style of comment in the source code. See also *pragma*.

Concrete string constant — Is an identifier which denotes a value of a concrete string type which is specified using a structured value constructor and a sequence of constant expressions.

Concrete string type — Is an array type (see *Declarations* in formal definition) whose component type is the predefined type **CHAR**¹⁾ (or is identical to it).

Concrete string variable — Is a variable of a concrete string type.

Concrete syntax — A grammar of the language in the form of a set of production rules. The form of the rules in the Standard are as specified by BS 6154 *Method of defining syntactic metalanguage*, 1981. All symbols of the language (i.e. keywords and operators) appear in the concrete syntax.

Concurrent program — A program which is written so that the potential parallelism is explicitly expressed – either using language constructs or library routines. (The language provides concurrency through the interleaving of execution of coroutines – either explicitly using **TRANSFER** or implicitly using **IOTRANSFER**.)

¹⁾This definition could be extended to include other character types if and when necessary.

■Conversion — See *type conversion*. NOT to be confused with *type transfer*. (See also *unsafe conversion*.)

Coroutine — A coroutine is a part of an executing program which may transfer execution to another coroutine and to which execution may be transferred. The transfer either occurs synchronously on the execution of an explicit **TRANSFER** statement, or it may occur asynchronously in response to an interrupt request. Execution transfers to the start of a new coroutine or to a point immediately after the point of the last transfer away from an old coroutine. (A new coroutine is one which has not yet been executed. An old coroutine is one which has previously been executed and from which a transfer has already been made.) A coroutine's state (the values of its local variables, its execution counter and its priority) is preserved while it is suspended.

■Declaration — The introduction of an identifier to denote a constant, type, variable, procedure or module. Declarations which are legal in a definition module are called *definitions*.

■Definition — A declaration which occurs in a definition module. (This includes the introduction of an identifier that denotes an opaque type.)

■Defining occurrence — The occurrence of an identifier in a context in which the meaning of that identifier is given. This will either be in a *definition* or in a *declaration*.

■Dimension — The term is applied to an array structure only: it is the number of successive index expressions that are required to access the first non-array component in the structure. That is it is one more than the number of *immediately* nested array structures within an array structure.

Designator — A designator is a specification of a variable or value of a variable in a statement or an expression. Designators are used to refer to variables, to the values of variables, to the values of constants, and to procedures. The simplest form of designator is an identifier, which may be qualified by one or more module names. Designators of arrays may also be indexed to refer to a component, designators of records may be selected to refer to a field, and pointer designators may be dereferenced. The following both designators:

- (i) a.b.c.e
- (ii) List[x+2][^].next

Device handler — A low-level routine which provides an interface between higher-level software and a peripheral device.

Dynamic storage — Variable storage that is not created automatically as a program executes (see *static storage*) but which is created by a program during execution.

■Elaboration — The processing of declarations and definitions by a translator.

■Element — A value of a base type which is included in the value of a set type. This term should only be used for sets; it should not be used in the context of array component values.

■Elementary type — Procedure, pointer, set and scalar types. (See *type*.)

Empty string literal — A denotation for the string terminator. Two forms are permitted – `""` (adjacent double quotation marks) and `''` (adjacent single quotation marks). An empty string literal is assignment compatible with concrete string types (and can be used to initialise a concrete string variable to have a string value of length zero). An empty string literal is expression compatible with the type **CHAR**, and its subranges.

■Entity — Obsolete term for *object*.

■Enumeration type — An ordinal type whose values are defined by a list of identifiers which denote the values; the ordering relation for the type is defined by the order in which the values appear in the list. *Enumeration type* should be used instead of *Enumerated type*.

Environment — The environment in which an identifier is applied determines the meaning of that identifier at the point of application. An environment is built from an enclosing environment, from imports, from pervasive identifiers, and from defining occurrences.

Error — This term refers to a construct which is in violation of the syntax or static semantics of the language, thus preventing meaning being ascribed to any text containing the construct. (For example, an ‘expression’ which includes a plus operator between two arrays is an error.) Thus an error prevents valid execution and *must* be detected by an implementation.

Exception — An exception is the violation of the dynamic semantics of the language, or of the semantics of a library module. It is a run-time occurrence which invalidates the normal semantics of an executing program. (For example, dereferencing a pointer variable whose value is **NIL**, is an exception.)

An exception in a program may be detected and reported by an implementation, in which case the Standard defines the semantics of the program, thus:

- either exceptional termination shall occur, or
- the program shall provide a procedure to handle the exception (cf. the standard module **EXCEPTIONS** — 7.3).

If an implementation does not detect an exception, or if, for other reasons, an implementation permits execution to continue after an exception has occurred, no meaning is given to such a program by the standard.

Note: the standard requires that certain exceptions be detected by an implementation in some mode of its use. (See section 6.12.)

(Also see *Fail-safe semantics*.)

Exception handler — An exception handler is a procedure which a program has established as the procedure which must be invoked on the occurrence of an exception. The exception handler can decide whether to propagate the exception (whether to cause exceptional termination) or whether to retry the statement in which the exception occurred

Exceptional termination — Program termination due to the detection of an exception; no domain exit procedures are called (i.e. no calls to **LEAVE** are made) and no result code is returned to the program’s environment.

Export clause — The construct which permits identifiers to be visible outside a module in which they have been declared.

Expression compatible — This term may be applied to types which may be mixed in an expression. Briefly, operands of identical types may be combined. An operand of a whole number type may be combined with an operand of the same type, or a subrange of that type, or **Z**. An operand of a real type may be combined with one of the same type or **R**. Operands which have identical or expression compatible host types may be combined. All pointer types may be used with **NIL**. Finally, an operand of a procedure type may be combined with an operand of a structurally equivalent procedure type.

This term replaces what Professor Wirth called *compatible*.

Fail-safe semantics — An exception handling mechanism which prohibits apparently normal execution of a program following the detection of an exception: either exceptional termination will occur, or an exception handler will be invoked. In the latter case, the exception handler will either return (thus reestablishing normal semantics) or will fail to return to the point at which it was called.

Gradual underflow — A term applied to a floating point system which has representation for number of too small a magnitude to be represented in normalized form (and hence having a larger relative spacing).

Host type — The host type of a subrange type is the type on which it is ultimately based; for numeric subranges, the host type will either be the standard unsigned type or the standard signed type; for character subranges it will be the standard type **CHAR**; for subranges of enumeration types the host type will be the enumeration type. The host type of an ordinal type other than a subrange is that ordinal type itself.

This replaces what Professor Wirth called *base type*.

■ **Identical type** — Two types **t1** and **t2** are identical if they have been defined by:

TYPE t1 = t2; or by:

TYPE t2 = t1;

Alternatively, there may be a third type **t3** which is identical to both **t1** and **t2**.

Implementation — An implementation of Modula-2 comprises the following:

- a translator;
- the required modules;
- optionally, modules of the standard library;
- proprietary system or separate modules;
- any software needed to support or integrate the above.

An implementation provides the means by which separate modules may be combined to produce an executable program.

Implementation defined — This term is used to refer to a feature of the language or standard library whose behaviour may be constrained by an implementation or whose attributes (e.g. the range of values of a type) may be chosen by an implementor. (For example, the range of values of the **INTEGER** type is implementation defined.) An implementation defined language feature must be supplied in an implementation; an implementation defined library feature must be supplied if the library module which contains it is supplied. Implementation defined behaviour must be predictable, and implementation defined attributes must always be available in an implementation. The standard requires that such features be fully documented by an implementor (and may, therefore, be validated). See Clause 4.8 for documentation requirements.

Implementation dependent — This term is used to refer to a feature of the language or standard library whose precise behaviour is at the discretion of an implementor. (For example, the order of evaluation of operands in an expression is implementation dependent.) An implementation dependent language feature must be provided in an implementation; an implementation dependent library feature must be provided if the library module which contains it is supplied. Implementation dependent behaviour is not necessarily predictable.

Import clause — The clause specifying which of the identifiers that are visible outside the module containing the clause are to be made visible within that module.

Index type — The type of the index of an array structure. For example, in the declaration:

TYPE l = ARRAY CHAR OF CARDINAL;

CHAR is the index type of the type **l**.

Interrupt — A change in state of the system (hardware and/or software), which is caused by an external event, and whose occurrence is communicated asynchronously to a Modula-2 program executing within the system. At the Modula-2 language level, an interrupt results in an asynchronous coroutine transfer.

Length — The number of characters in a string, and hence the number of characters in the string value which represents it. (The length is equal to the number of characters in a string value up to, but not including, any string terminator.)

Level 0 — The outermost level of procedural nesting; that is, the level of program module blocks, implementation module blocks and local module blocks nested immediately within such blocks.

Library — A collection of modules supplied for general use.

Lifetime — The lifetime of a variable is that period of program execution in which it exists.

Local module — A module which is declared within another module or procedure.

■Member — Use *element*.

■Meta-IV — Obsolete term for *VDM-SL*: the specification language of VDM.

Mutual exclusion — If A and B execute in mutual exclusion, it cannot be the case that A and B both execute at the same time. The language provides for mutual exclusion between the whole program and its environment using PROTECTION. (Environment is used here in the sense of "surroundings" or "outside" rather than in the more technical sense used in the formal definition.) The library provides facilities in the Semaphores module to express mutual exclusion between processes.

■Nested module — Use *local module*.

New type — A non-pervasive type; i.e. a type which is explicitly declared in a type declaration, or an anonymous type which is implicitly declared in a variable declaration.

Non-standard — When applied to a program this means that the standard does not ascribe meaning to the program. (Thus, for example, a program which evaluates an uninitialised variable is non-standard.)

Normal termination — Program termination which returns a result code to the program's environment and which occurs by reaching the end of the program module, or by executing a return statement in the body of the program module. (Normal termination ensures that domain exit procedures are called, i.e. calls to **LEAVE** are made)

Object — A value of a VDM type. Effectively, anything which is denoted by an identifier in Modula-2: a constant, a type, a variable, a procedure or a module.

Operation completion condition — This is a predicate on the parameters to a procedure in the string library module which changes a string value. If the condition associated with a procedure is true before the procedure is called, the procedure will be able both to complete the change to the string value and to maintain the string abstraction.

Ordinal type — A character type, an enumeration type, a whole number type, or a subrange of any of these.

Order of evaluation — This term is relevant to the implementation *dependent* manner in which the following may be evaluated:

- operands in expressions;
- actual value parameters;
- index expressions in array designators;
- left and right hand sides of an assignment statement.

An implementation is free to evaluate the *operands* of an expression in any order; the operators are applied in a predefined order. An implementation is free to evaluate actual value parameters in any order during procedure activation. The order of evaluation of index expressions is not defined, nor is whether the left or right hand side of an assignment statement is evaluated first. For a program to be *standard* the result of an evaluation of the type described must be the same, no matter what order is used. (Consequently, programs which include function calls with side-effects are not standard.)

■**Packing** — The implementation technique by which two variables of a structured type (array components or record fields) are represented in less than twice the number of storage locations used for one variable. (Packing is only permissible if the semantics of the language are unchanged when it is used. It is unlikely in Modula-2 because a packing implementation would have to allow packed components or fields as actual VAR parameters.)

■**Parallelism** — The execution of more than one part of the program simultaneously. There are two aspects: the use of parallelism as a possible implementation strategy and expression of parallelism by using language constructs or by using the library routines designed for concurrent programming. An implementation may use parallelism if the resultant meaning is one of those allowed by the standard (e.g. in the evaluation of operands of an expression). The constructs in the language only deal with the case of parallelism between the program and its environment and provide a simple method of mutual exclusion between them using **PRIORITY**. The library provides routines which enable the expression of parallelism and mutual exclusion between parts of the program known as Processes.

■**Parameter compatible** — If **tf** is the type of a formal parameter, and **ta** is the type of the corresponding actual parameter, then **ta** is parameter compatible with **tf** if one of the following statements is true:

- a) **ta** is identical to **tf**;
- b) **tf** is a value parameter and **ta** is assignment compatible with **tf**;
- c) **tf** is an open array parameter and **ta** is an array of the same number of dimensions as **tf** and whose component type is identical to that of **tf**;
- d) **tf** is a procedure type and **ta** has the same structure as **tf**;
- e) **tf** is of the smallest addressable unit type (e.g. **BYTE** or **WORD**) and **ta** is any type whose size is that of the smallest addressable unit (i.e. **SIZE(ta) = 1**);
- f) **tf** is an array of smallest addressable units, as defined by **tf = ARRAY[0..n-1] OF LOC**; and **ta** is any type for which **SIZE(ta) = n**;
- g) **tf** is an open array parameter of smallest addressable units (e.g. **ARRAY OF BYTE** or **ARRAY OF WORD**) and **ta** is any type;
- h) **tf** is an open array of arrays of smallest addressable units, as **ARRAY OF Chunk** when **Chunk = ARRAY[0..m--1] OF LOC**, and **ta** is any type whose size (given by **SIZE**) is a multiple of **m**.

■**Pervasive identifier** — An identifier of a standard procedure, type or constant which pervades all program or separate modules, and is thus available without requiring it to be declared, or imported (into any kind of module). If a pervasive identifier is redefined, the identifier loses its pervasive nature within the scope in which it is redefined, and access to the standard procedure, type or constant will be lost.

■**Post-condition** — The post-condition of a VDM function asserts what is true of the result of a function if its input parameters satisfy the function's pre-condition.

■**Pragma** — An instruction to an implementation which guides it in the construction of a program. The pragma may direct the translator by appearing in the source text of a compilation unit, or may direct the linker by the setting of some implementation parameter, or by a parameter to an operating system command.

■**Precedence** — The binding rules applied to operators in an expression without brackets (preferred to 'operator priority').

■**Pre-condition** — The condition which the input parameters to a VDM function must satisfy for the function to be defined.

■**Preemptive scheduling** — In a situation where an implementation cannot simultaneously execute all active processes it has to adopt a scheduling strategy to share processing time among the competing processes. Preemptive

scheduling allows the execution of an active process to be forcibly delayed (preempted) while other active processes execute. The standard Processes library module requires any scheduling to be preemptive to the extent that there may be no executing process of a lower urgency than any process which is ready but has been delayed by the scheduler. The module does not require a preemptive scheduling strategy between processes of the same urgency: the implementor is free to provide any policy appropriate for the intended application.

■**Priority** — In the context of the Modula-2 standard this term is used to describe how all or part of a Modula-2 program may be set to allow or to disallow interrupts. Priority is thus a low-level mutual exclusion mechanism between the program and an interrupt source.

■**Process** — A process is an abstraction supported by the concurrent programming library. Writing a program as a set of processes expresses the fact that parts of that program may be executed concurrently (other than in certain sections subject to mutual exclusion).

Processor — The combination of hardware, operating system, implementation, etc. which permits the translation and execution of Modula-2 programs. (A ‘processor’ may include two machines, if cross-compilation is involved.)

■**Program** — The combination of a program module and all modules on which it directly or indirectly depends.

■**Qualified** — An identifier is qualified if it is preceded by the name of the module which exported it and a full stop.

■ **\mathbb{R} type/RR type** — A conceptual type whose values are the set of all real number literals. The type \mathbb{R} is expression compatible with both the **REAL** and **LONGREAL** types.

Range type — The range type of a subrange is the type which a programmer asserts contains the minimum and maximum values of the subrange. For example:

```
CONST first = 1; last = 10; TYPE short = [0..255]; VAR x:
  short[first..last];
```

Here **short** is the range type of the anonymous subrange type **[first..last]**. Note that a subrange, and its range type share the same host type.

The syntactic device of preceding a subrange definition with a range type name can be used to force the host type of the subrange to be that of the range type. For example, in the following declaration **sub** is forced to be an **INTEGER** subrange:

```
TYPE sub = INTEGER[0..10];
```

Range compatible — In an assignment statement, or in the context of passing an actual value parameter, an expression and a variable/parameter are range compatible if their host types are assignment compatible and the value of the expression falls within the range of the values of the type of the variable/formal parameter.

Required identifier — A pervasive identifier which must be accepted by an implementation.

Required system module — A system module which must be supplied by an implementation¹⁾.

■ **\mathbb{S} type/SS type** — A conceptual string type (like the numeric types \mathbb{Z} and \mathbb{R}) which includes strings of all lengths (in particular, a single character is of the type \mathbb{S}). Its purpose is to provide a conceptual basis for compatibility rules for string literals.

■**Safe (type) conversion** — Obsolete term for **type conversion**.

■**SAL** — *Smallest addressable location.*

¹⁾This definition may need to be extended to cover required names for standard library modules and their identifiers.

■SAU — *Smallest addressable unit*. Now called *SAL*.

Scope — Those parts of a program or module text in which an identifier is visible. The scope of an identifier declared in a procedure is the procedure block, but this can be restricted by redeclaring it within a nested procedure or by not importing it into a (nested) local module. The scope of an identifier declared in a local module is the module block plus enclosing blocks to which it is exported, but this can be restricted by redeclaring it within a nested procedure or by not importing it into a (nested) local module. The scope of an identifier declared in a definition module is its implementation module plus the scope of any other module which imports it.

Separate compilation — The means by which modules may be compiled separately and by which their dependence is controlled.

■Separate module — An encapsulation of data and procedures which has been separately compiled resulting in Modula-2 definition and implementation modules. Separate modules may be collected together to form a library.

■Signed (whole number) type — An integer.

■Signed arithmetic — Integer arithmetic (which permits negative values).

■Simple type — Old term for *elementary type* (now denigrated).

■Simple variable — A variable of an elementary type.

Smallest addressable location — The implementation defined unit of storage such that for a type which is implemented as a smallest addressable unit, the **TSIZE** of the type is 1. Denoted by **LOC**.

Single pass — This term is used to apply to a translator which essentially scans a source program only once. (Thus a single pass translator cannot deal with declaration after use nor can it deal with mutually importing modules; a single pass compiler can only accept mutually importing procedures if they appear in definition modules or if one is declared in a forward declaration.)

Standard — When applied to a program this means that the program can be given a meaning according to the semantics of the language specified in the Standard.

Standard library — The library consisting of modules specified in the Standard, and consisting of at least the required separate modules and required system modules.

■Standard procedure — A required, procedure or function procedure (which is accessed by a pervasive identifier).

Static storage — Variable storage which is created automatically as a program executes; the code to create the storage is generated by a translator as the result of analyzing the static program text.

■Step — The number of ordinal values by which a for loop control variable is to be incremented (or decremented, if negative) on each iteration of a for loop.

String — A value of an abstract data type which is sequence of characters. (A string is represented by a sequence of values in a character array called a *string value*, see below.)

String constant — An identifier which denotes a string literal. (The term should not be used to describe a structured constant of a concrete string type.)

String literal — A denotation of a string which includes a sequence of printable characters between either a pair of single quotation marks (‘’) or a pair of double quotation marks (”).

String terminator¹⁾ — A value of the predefined type **CHAR** which is used to terminate a string value in a concrete string variable when the length of the string it represents is less than the capacity of the variable. The value is implementation defined.

String value — A sequence of **CHAR** values in a concrete string variable (beginning at the variable's lowest index) which represents a string (i.e. which represents an abstract character sequence).

■ **Strong typing** — Language rules which require an implementation to reject programs which break specific type rules concerning the combination of types in a statement or an expression.

System module — A required module (such as **SYSTEM** and **COROUTINES**) known to the translator. System modules need not be supplied as separate modules, but access to them must be through the normal module import mechanism.

■ **Subrange type** — A type whose values are a restricted range of a previously defined ordinal type or of a standard, predefined ordinal type.

■ **Structured type** — An array type or a record type.

■ **Termination** — The sequence of actions taken by a program as it halts. These may depend on whether the program halts normally, or as the result of detecting an exception. (See *Normal termination* and *Exceptional termination*.)

■ **Threaten** — To allow the possibility of changing the value of a **FOR** loop control variable within its **FOR** loop by some means other than by the **for** statement itself. Thus a control variable can be threatened by directly assigning to it, by attempting to pass it as an actual variable parameter, or by manipulating it via its address obtained using **SYSTEM.ADR**.

Type — A type is a set of values, or a structure containing variables of defined types, and set of permissible operations on those values. Types are classified as shown in the diagram below (\$ is not included). (See also *basic type* and *elementary type*.)

NOTE — DIAGRAM GOES HERE

Type conversion — The translation of a value of one type into a value of another type according to some logical relationship between the types. (Typically, the relationship is numerical and the conversion implies a change in the underlying representation of the original value.) A run- time check is carried out to ensure that the conversion is possible. In standard Modula-2 type conversion is provided by **VAL**. (Previously, Modula-2 implementations allowed type conversion using both **VAL** and type transfer functions. See also **Type transfer**.)

Type transfer — The interpretation of a bit pattern representing the value of one type as a value of another type without any run- time check to ensure that the bit pattern is a valid value of the target type. In standard Modula-2 type conversion is provided by **SYSTEM.CAST**. (Previously, Modula-2 implementations allowed type transfer using **VAL** and type transfer functions. See also *Type conversion*.)

■ **Unsafe conversion** — Obsolete term for *type transfer*.

■ **Unsigned arithmetic** — Whole number arithmetic which excludes negative values (i.e. cardinal arithmetic).

■ **Unsigned (whole number) type** — Cardinal

■ **Urgency** — A measure of the degree to which a process may be rescheduled; otherwise known as 'software priority'.

■ **Variable** — A location (or set of locations) which can be used to hold values of a type.

VDM — Vienna Development Method. The dialect used in the Modula-2 Standard is an extended version of the proposed BSI/ISO VDM standard.

VDM-SL — Vienna Development Method Specification Language. (The new term for Meta-IV.)

Visible — An identifier is said to be visible at the point at which it appears in Modula-2 source text if it is available for use at that point.

Well-formed — Meets the static semantics rules of the language and, as a prerequisite to these, meets the syntax rules.

Well-ordered — A module is well-ordered if the declarations within it are well-ordered and if its statements are well-ordered. Declarations are well-ordered if they do not use identifiers which are declared later, textually. (The requirement of declaration-before-use is relaxed for the declaration of a pointer type.) Depending on the implementation strategy used, statements are either always well-ordered, or are well-ordered if they do not use identifiers which are declared later, textually. (A statement which references a pointer variable whose type has not yet been fully declared is well-ordered if the reference is to the entire variable – similar to the constraints on a client module using a variable of an opaque type.)

Whole number type — The standard type **CARDINAL** or the standard type **INTEGER**. (i.e. signed + unsigned types).

Workspace — The storage allocated to a coroutine for its state: local variables, evaluation stack and priority.

\mathbb{Z} type/ZZ type — A conceptual type whose values are the set of all whole number literals. The type **\mathbb{Z}** is expression compatible with both the signed and unsigned types.

TEMPORARY NOTE — Terms from previous issues which have been omitted:

Applied occurrence
Denotational semantics
Dynamic environment
Importance
Required separate module
Required module
Static environment
Smallest addressable unit