Modula-2 Revision 2010

Language Report for the

Bootstrap Kernel Subset (BSK)

Benjamin Kowarsch and Rick Sutcliffe

July 2020

Font Test Page

This page is **not** part of the (final) document.

def This is a definition.

- This is a bullet list item.
- This is another bullet list item.

Roman Font

roman (ec-qagr)

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789 !"#$%&’ ()*+,-. /:;<=>? @[\]^_` {|}~

Alternate Roman Font

alt roman (ec-qcsr)

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789 !"#$%&’ ()*+,-. /:;<=>? @[\]^_‘ {|}~

Sans Serif Font

sans serif (phvr8t at 8.50006pt)

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789 !"#$%&’ ()*+,-. /:;<=>? @[\]^_‘ {|}~

Monospace Font

monospaced (FiraMono-Regular-tlf-t1 at 6.99997pt)

```
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789 !"#$%&’ ()*+,-.  /:;<=>?  @[\]^_‘ {|}~
```

Typography

The following typographic conventions are used throughout this document:

Body Text

Conventions used within body text

Sample	Typeface	Usage
All imports are qualified	sans serif	represents body text
Pragmas may **not** be nested	sans serif bold	represents emphasis in body text
$\boldsymbol{cardinality}$ or value count	roman bold italic	represents defined reference terms
Type `CHAR` is a character type	monospaced	represents inline source code
def $\boldsymbol{frac}\, x = x - \boldsymbol{int}\, x$	framed box “def”	represents a definition
where $\tau_1, \tau_2 \in \tau_{cardinal}$	math font	represents inline math formulas

Grammar

Conventions used within EBNF grammar listings

Sample	Typeface	Usage
`statementSeq`	monospaced	symbols, operators, punctuation, literals
`identList` `:= ···`	monospaced bold	definition of a terminal or non-terminal
`MODULE` `ident ';'`	monospaced bold grey	reserved words of Modula-2
`/* whitespace */`	monospaced italic grey	comments

Source Code

Conventions used within Modula-2 source code listings

Sample	Typeface	Usage
`a := b + c;`	monospaced	symbols, operators, punctuation, literals
`VAR` `ch : CHAR;`	monospaced bold	reserved words of Modula-2
`(* whitespace *)`	monospaced italic grey	comments
`<*FFI="C"*>`	monospaced bold grey	pragmas

Syntax Notation

An Extended Backus-Naur Formalism (EBNF) variant is used throughout this document to specify syntax. The specifics of this EBNF are specified below.

Syntax

```
/* non-terminals */
grammar := ( definition ';' )+ ;
definition := symbolDef | aliasDef ;
symbolDef := Ident ':=' expression | literal ;
expression := term+ ( '|' term+ )* ;
term := factor ( '*' | '+' | '?' )? ;
factor := Ident | literal | '(' expression ')' ;
literal := charOrRange | quotedCharOrRange | QuotedCharSeq ;
charOrRange := 'ASCII(' Number ( '..' Number)? ')';
quotedCharOrRange := QuotedChar ( '..' QuotedChar )? ;
aliasDef := 'alias' Ident '=' Ident ;
/* terminals */
Ident := TerminalIdent | NonTerminalIdent ;
TerminalIdent := ( 'A' .. 'Z' ) Letter* ;
NonTerminalIdent := ( 'a' .. 'z' ) Letter* ;
Letter := ( 'a' .. 'z' ) | ( 'A' .. 'Z' ) ;
QuotedLetter := "'" Letter "'" | "'" Letter "'" ;
QuotedChar := "'" PrintableChar "'" | "'" PrintableChar "'" ;
QuotedCharSeq := "'" PrintableChar+ "'" | "'" PrintableChar+ "'" ;
PrintableChar := ASCII(0x20..0x7E) ;
Number := Digit+ | '0x' ( Digit | 'A' .. 'F' )+ ;
Digit := '0' .. '9' ;
/* non-semantic symbols */
Whitespace := ASCII(0x20) ;
Comment := '/*' PrintableChar* '*/' ;
```

Semantics

Syntax	Position	Sample	Meaning	Precedence
`|`	infix	`t1 | t2`	alternative terms `t1` and `t2`	(lowest) 1
`?`	postfix	`t?`	optional term `t`	2
`+`	postfix	`t+`	arbitrary repetition of a mandatory term `t`	2
`*`	postfix	`t*`	arbitrary repetition of an optional term `t`	2
`(` `)`	enclosure	`( t1 t2 t3 )`	precedence grouping of enclosed terms	(highest) 3

Math Notation

The following math symbols are used throughout this document:

General

Symbol	Meaning	Pronounced as
$\overset{def}{\longmapsto}$	Definition	defines
$\in:$	Type Membership	is of type
$\Gamma_t \vdash s$	Type regime assertion	s is true under type regime t

Type Relations

Symbol	Meaning	Pronounced as
$\equiv$	Identity	is identical to
$\doteq$	Canonical Identity	is canonically identical to
$\widehat{=}$	Structural Equivalence	is structurally equivalent to
$<:$	Subtype	is subtype of
$>:$	Extenstion	is extension of

Type Compatibility

Symbol	Meaning	Pronounced as
$\overset{:=}{\rightarrow}$	Assignment Compatibility	is assignment compatible to
$\overset{copy}{\longrightarrow}$	Copy Compatibility	is copy compatible to
$\overset{val}{\longrightarrow}$	By-Value Passing Compatibility	is by-value passing compatible to
$\overset{ref}{\longrightarrow}$	By-Reference Passing Compatibility	is by-reference passing compatible to
$\overset{expr}{\longleftrightarrow}$	Expression Compatibility	is expression compatible with

Chapter 1

Lexical Entities

1.1 Reserved Words

1.1.1 Classical Tokens

`ALIAS`, `AND`, `ARGLIST`, `ARRAY`, `BEGIN`, `CASE`, `CONST`, `COPY`, `DEFINITION`, `DIV`, `DO`, `ELSE`, `ELSIF`, `END`, `EXIT`, `FOR`, `IF`, `IMPLEMENTATION`, `IMPORT`, `IN`, `LOOP`, `MOD`, `MODULE`, `NEW`, `NOP`, `NOT`, `OCTETSEQ`, `OF`, `OPAQUE`, `OR`, `POINTER`, `PROCEDURE`, `READ`, `RECORD`, `RELEASE`, `REPEAT`, `RETAIN`, `RETURN`, `SET`, `THEN`, `TO`, `TYPE`, `UNTIL`, `VAR`, `WHILE`, `WRITE`.

1.1.2 Schroedinger's Tokens

`ADDRESS`, `CAPACITY`, `CAST`, `NIL`.

1.2 Special Symbols

1.2.1 Delimiters

`(` `)` `[` `]` `{` `}` `'` `"`

1.2.2 Punctuation

`.` `,` `:` `;` `=` `+` `*` `@` `|` `..` `:=` `++` `--` `.*`

1.2.3 Operators

`=` `#` `<` `<=` `>` `>=` `+` `-` `&` `\` `*` `/` `::` `^` `.`

1.2.4 Comment Delimiters

`!` `(*` `*)`

1.2.5 Pragma Delimiters

`<*` `*>`

1.3 Identifiers

Identifiers denote predefined or user defined names for syntactic entities. A standard Modula-2 identifier starts with a letter and may be followed by letters and digits.

```
StdIdent := Letter ( Letter | Digit )* ;
```

1.3.1 Foreign Identifiers

A foreign identifier may also contain leading, non-trailing, non-consecutive `$` characters and non-leading, non-trailing, non-consecutive `_` characters. Foreign identifiers are used in connection with foreign library APIs. They are disabled by default. Their use must be enabled by compiler option or by pragma.

```
ForeignIdent :=
  '$' ( Letter | Digit ) ForeignIdentTail* | StdIdent ForeignIdentTail+ ;
ForeignIdentTail :=
  ( '$' | '_' ) ( Letter | Digit )+ ;
```

1.4 Numeric Literals

Numeric literals denote numeric values, either real number values, whole number values or character code values.

1.4.1 Real Numbers

Real number values are always given in decimal notation, start with an ***integral part***, followed by a ***fractional part***, followed by an optional exponent.

```
RealNumber := integralPart fractionalPart exponentialPart? ;
integralPart := '0' | ( '1' .. '9' ) ( DigitSeparator? DigitSequence )? ;
fractionalPart := '.' DigitSequence ;
exponent := 'e' ( '+' | '-' )? DigitSequence ;
DigitSequence := DecimalNumber ;
```

1.4.2 Whole Numbers

Whole number values may be given in decimal, radix-2 or radix-16 notation. C-style prefixes are used to indicate the radix of non-decimal literals. Prefix `0b` indicates radix-2 and prefix `0x` indicates radix-16. Digits may be grouped using `'` as a digit separator. A digit separator must always be preceded and followed by a digit.

```
WholeNumber := DecimalNumber | Base2Number | Base16Number ;
DecimalNumber := Digit+ ( DigitSeparator Digit+ )* ;
Base2Number := '0b' Base2Digit+ ( DigitSeparator Base2Digit+ )* ;
Base16Number := '0x' Base16Digit+ ( DigitSeparator Base16Digit+ )* ;
Digit := '0' .. '9' ;
Base2Digit := '0' | '1' ;
Base16Digit := Digit | 'A' .. 'F' ;
alias DigitSeparator = "'" ;
```

1.4.3 Character Codes

Character code values are always given in radix-16 notation with prefix `0u`.

```
CharacterCode := '0u' Base16Digit+ ( DigitSeparator Base16Digit+ )* ;
```

1.5 Quoted Literals

Quoted literals denote text. They are delimited by single quotes `'` or double quotes `"`.

```
QuotedLiteral := SingleQuotedLiteral | DoubleQuotedLiteral ;
```

A single quoted literal may contain double quotes but not single quotes. A double quoted literal may contain single quotes but not double quotes. A quoted literal may contain whitespace but **no** tabulator, **no** newline, nor any other control codes. As a result, it may **not** span multiple lines.

```
SingleQuotedLiteral := "'" ( AnyPrintableExceptSingleQuote | EscSeq )* "'" ;
DoubleQuotedLiteral := '"' ( AnyPrintableExceptDoubleQuote | EscSeq )* '"' ;
```

Backslash escape sequences may be used to denote newline and tabulator within a quoted literal. Escape sequence `\n` denotes newline and `\t` denotes tabulator. A verbatim backslash must be escaped as `\\`. No other escape sequences shall be supported.

```
EscSeq := '\' ( 'n' | 't' | '\' ) ;
```

1.6 Non-Semantic Entities

1.6.1 Whitespace

Whitespace `ASCII(0x20)` terminates a symbol, except within quoted literals, pragmas and comments.

1.6.2 Tabulator

Tabulator `ASCII(0x09)` terminates a symbol, except within pragmas and comments.

1.6.3 Newline

Newline terminates a symbol, except pragmas and block comments. A newline increments the line counter and resets the column counter used in compile-time messages.

```
Newline := ASCII(0x0A) | ASCII(0x0D) ( ASCII(0x0A) )? ;
```

1.7 Comments

Comments are non-semantic symbols ignored by the language processor. They may occur anywhere before or after semantic symbols and are used for documentation and annotation. There are line and block comments.

1.7.1 Line Comments

Line comments start with a `!` and are terminated by newline.

```
LineComment := '!' ( AnyPrintable | Tabulator )* Newline ;
```

1.7.2 Block Comments

Block comments are delimited by block comment delimiters `(*` and `*)`. Block comments may span multiple lines and may be nested up to a maximum of nine levels, not counting the outermost comment.

```
BlockComment := '(*' ( BlockComment | AnyPrintable | Tabulator | Newline )* '*)' ;
```

1.8 Pragmas

Pragmas are directives to the language processor to control or influence the compilation process. They are delimited by delimiters `<*` and `*>`, may span multiple lines but may **not** be nested.

```
Pragma := '<*' PragmaBody '*>' ;
```

Chapter 2

Compilation Units and Scopes

2.1 Compilation Units

Modula-2 programs and libraries consist of one or more compilation units called modules. A program consists of exactly one program `MODULE` and any number of library modules. A library module consists of a definition part, called a `DEFINITION` `MODULE`, and an optional implementation part, called an `IMPLEMENTATION` `MODULE`. The definition part defines the client interface of the library. The implementation part implements the library according to the interface. The implementation part is strictly accessible through its definition part only.

2.2 Identifier Visibility

Identifiers that are predefined by the language are pervasive, that is, they are visible in every scope without import. Identifiers provided by a special built-in module are visible only within a scope into which they are explicitly imported. Identifiers defined within a definition module are visible within the definition module in which they are defined, within the corresponding implementation module and any scope into which they are explicitly imported. Identifiers declared within an implementation or program module are visible only within the scope in which they are declared.

Constants and aliases are visible in the scope in which they are defined or declared from the point forward where they are defined or declared. That is, constants and aliases must be defined or declared before they are referenced.

Types, procedures and variables are visible in the entire scope in which they are defined or declared. That is, they may be referenced before they are defined or declared.

2.3 Scope

Scope is a classification of the visibility of identifiers. There are three levels of scope. Global scope, module scope and local scope. Module scope is a sub-scope of global scope and local scope is a sub-scope of module scope.

2.3.1 Global Scope

Global scope encompasses an entire program. Only predefined identifiers have global scope.

2.3.2 Module Scope

Module scope encompasses a module. Identifiers that are defined or declared in, or imported into the top level of a module have module scope.

2.3.3 Local Scope

Local scope encompasses a `PROCEDURE` declaration or a `FOR` statement.

Local scopes may be nested. The surrounding scope in which a local scope is nested is called the outer scope. The nested scope is called the inner scope.

Identifiers defined or declared in an outer scope are also visible in an inner scope and may be redefined within the inner scope. Redefining an identifier of an outer scope within an inner scope causes the entity denoted by the identifier to be shadowed. A shadowed entity is no longer addressable through its identifier within the scope in which it has been shadowed and within any of its sub-scopes.

Identifiers that belong only to an inner scope are visible only in the inner scope and any of its sub-scopes but not in any outer scope.

2.3.3.1 Procedure Scope

Procedure scope is created by a `PROCEDURE` declaration.

Procedure scope overlaps with the outer scope of the procedure. The outer scope extends to the end of the procedure header and continues after the end of the procedure body. The inner scope extends from the beginning of the procedure header to the end of the procedure body. The procedure header thus belongs to both the outer and the inner scope.

The procedure's identifier is visible in both the outer and the inner scope. It may be referenced in either scope and may not be redefined in either scope.

The identifiers of a procedure's formal parameters exist in both the outer and the inner scope but are visible only within the inner scope and may not be redefined therein.

2.3.3.2 FOR Statement Scope

`FOR` statement scope is created by a `FOR` statement.

A `FOR` statement scope does not overlap with its outer scope. It extends from the beginning of the `FOR` statement's header to the end of its body. The loop variants declared within the loop header have local scope, they are only visible within the loop.

2.4 Import and Export

Identifiers defined within a definition module are automatically exported for import and use by client modules. By contrast, program and implementation modules do not export any identifiers. An `IMPORT` directive may be used within any kind of module to explicitly import the identifiers exported by definition modules. A definition module does not implicitly re-export imported identifiers.

```
IMPORT FooLib, BarLib, BazLib;
```

2.4.1 Duplicate Import

A duplicate import is an import of a library that has already been imported into the same scope. Any such import is ignored and shall cause a compile time warning.

2.4.2 Qualified Import

All imports are qualified. That is, an imported identifier is referenced in the importing module by a qualified identifier. A qualified identifier consists of the module identifier, followed by a period, followed by the actual identifier. Qualification prevents name conflicts between identical identifiers in different libraries.

```
IMPORT Flintstones, Rubbles;
...
person := Flintstones.Fred;
person.spouse := Flintstones.Wilma;
person.friend := Rubbles.Barney;
person.friend.spouse := Rubbles.Betty;
```

2.4.3 Import Aggregation

Imported modules are not implicitly re-exported but they may be **explicitly** re-exported. Module identifiers to be imported for automatic re-export are marked with a re-export suffix `+`.

```
import := IMPORT libIdent reExport? ( ',' libIdent reExport? )* ';' ;
alias libIdent = StdIdent ;
alias reExport = '+' ;
```

A library module that imports other modules for the sole purpose of re-export is called an import aggregator. The facility is called import aggregation. It is useful for importing multiple libraries with a single import directive.

```
DEFINITION MODULE CharIO;
IMPORT SoleCharIO+, ArrayOfCharIO+;
END CardinalIO.
```

When a client module `M` imports an import aggregator `A` which imports and re-exports modules `B` and C, client module `M` then automatically imports modules `B` and `C`.

```
IMPORT CharIO; VAR char : CHAR; str : ARRAY 10 OF CHAR;
...
WRITE "char: ", char; (* use of syntax bindings in SoleCharIO *)
WRITE "string: ", str; (* use of syntax bindings in ArrayOfCharIO *)
```

2.5 Abstract Data Type Libraries

An abstract data type library is a library that provides an abstract data type (ADT) whose identifier matches the identifier of its library module.

```
DEFINITION MODULE String;
TYPE String = OPAQUE;
```

The identifier of such an ADT is **implicitly** aliased when it is imported and may therefore be used unqualified.

```
IMPORT String;
VAR s : String; (* alias for String.String *)
```

An ADT library may provide syntax bindings (see chapter 10). Except for IO bindings, other libraries may not.

2.6 Institution

The introduction of a name along with an associated specification of an entity is called institution. There are two kinds:

- definition
- declaration

A definition is an institution of interface of an entity, a declaration is an institution of implementation. However, in the case of constants, variables and types other than ***opaque types***, interface and implementation are inseparable. For this reason, by convention, any institution within the definition part of a library module is called a definition and any institution within the implementation part of a library module is called a declaration.

Chapter 3

Aliases, Constants and Variables

3.1 Aliases

An alias represents the unqualified name of a qualified identifier. The unqualified name is called alias and the qualified identifier it represents is called its translation. An alias may be instituted within implementation and program modules only. It is visible only within the scope in which it is declared and its translation must be visible within that scope. Aliases may be declared explicitly by name or implicitly by wildcard.*

```
aliasDeclaration := namedAliasDecl | wildcardAliasDecl ;
```

3.1.1 Named Alias Declaration

A named alias declaration specifies one or more aliases to be declared explicitly by their name. An alias declaration for a single alias specifies its translation explicitly. The alias must match the unqualified identifier of its translation. An alias declaration for multiple aliases specifies their translations implicitly by qualified wildcard.

```
namedAliasDecl :=
  aliasName ( '=' qualifiedName | ( ',' aliasName )* '=' qualifiedWildcard ) ;
alias aliasName = StdIdent ;
alias qualifiedName = qualident ;
qualifiedWildcard := qualident '.*' ;
```

A declaration of the form

```
ALIAS Fred = Flintstone.Fred;
```

declares a single alias `Fred` to represent qualified identifier `Flintstone.Fred`.

A declaration of the form

```
ALIAS Fred, Wilma = Flintstone.*;
```

declares two aliases `Fred` and `Wilma` to represent `Flintstone.Fred` and `Flinstone.Wilma` respectively.

3.1.2 Wildcard Alias Declaration

A wildcard alias declaration clause declares all aliases matching a qualified wildcard. The qualifying part of the qualified wildcard must reference an enumeration type.

```
wildcardAliasDecl := '*' '=' qualifiedWildcard ;
```

A wildcard alias declaration within the context and of the form

```
TYPE Colour = ( Red, Green, Blue );
ALIAS * = Colour.*;
```

*This replaces unqualified import and the `WITH` statement of earlier versions of Modula-2.

declares aliases for every value in enumeration type `Colour`. The resulting aliases are `Red`, `Green` and `Blue` representing `Colour.Red`, `Colour.Green` and `Colour.Blue` respectively.

3.1.3 Visibility of Aliases

Unlike unqualified names brought into scope by unqualified import in earlier Modula-2 versions, unqualified names declared by an alias declaration have local scope. They are only visible within the scope in which they are declared, reducing the likelihood of name collision when using unqualified names.

Given the type declaration

```
TYPE MsgType = ( Info, Warning, Error, Fatal );
```

the values of type `MsgType` are only visible by their qualified names.

An alias declaration within a local scope of the form

```
PROCEDURE EmitMsg ( type : MsgType; CONST text : ARRAY OF CHAR );
ALIAS * = MsgType.*;
BEGIN
  (* unqualified aliases for MsgType values are in scope here *)
END EmitMsg;
(* unqualified aliases for MsgType values are no longer in scope here *)
```

declares aliases `Info`, `Warning`, `Error` and `Fatal` within the local scope of procedure `EmitMsg`. The aliases are visible only within the scope of the procedure.

3.2 Constants

A constant is an identifier that represents an immutable value determined at compile time. Its value is specified by a ***constant expression*** in its institution clause. A constant may be instituted within definition, implementation and program modules.

```
constDefinition :=
  CONST ident '=' constExpression ;
alias constDeclaration = constDefinition ;
```

The expression that specifies the value of the constant must be a ***constant expression*** and it may not be a module or type identifier.

3.3 Variables

A variable is a named reference to a mutable value. It is always associated with a type and may only hold values of its type. The type is specified in its institution clause and is immutable. A variable may be instituted within definition, implementation and program modules. The syntax of variable definition and declaration clauses differs slightly.

3.3.1 Variable Definition

A variable definition may define variables of named types only.

```
varDefinition :=
  VAR ( identList ':' typeIdent ';' )+ ;
```

Variables defined in a definition module are always exported immutable. They are mutable within the corresponding implementation module but immutable in any other scope.

3.3.2 Variable Declaration

A variable declaration may declare variables of named or anonymous types.

```
varOrFieldDeclaration :=
  identList ':' ( typeIdent | anonType ) ;
anonType :=
  ARRAY valueCount OF typeIdent | subrangeType | procedureType ;
```

3.3.3 Variable Initialisation

Variables of all ***pointer types*** are automatically initialised to hold the invalid pointer value `NIL`. Variables of other types are not initialised. Their values are undetermined after allocation and must be explicitly initialised by assignment at runtime.

Chapter 4

Types

A type is a classification of units of data that determines storage representation, value range and semantics. A type may be instituted within definition, implementation and program modules. The syntax of type definition and declaration clauses differs slightly.

4.1 Type Institution

4.1.1 Type Definition

A type definition may specify any type constructor except the indeterminate type constructor.

```
typeDefClause :=
  TYPE ( typeDefinition ';' )+ ;
typeDefinition :=
  ident '=' ( OPAQUE | type ) ;
type :=
  aliasType | derivedType | subrangeType | enumType | setType |
  arrayType | recordType | pointerType | procedureType ;
```

4.1.2 Type Declaration

A type declaration may specify any type constructor except the `OPAQUE` type constructor.

```
typeDeclClause :=
  TYPE ( typeDeclaration ';' )+ ;
typeDeclaration :=
  ident '=' ( indeterminateType | type ) ;
```

4.2 Opaque Types

An ***opaque type*** is a ***pointer type*** whose name is public but whose target type is hidden. Its definition is placed in a definition module and has the following syntax

```
opaqueTypeDefinition :=
  TYPE ident '=' OPAQUE ;
```

A matching declaration specifying the target type must be placed in the corresponding implementation module. It has the following syntax

```
opaqueTypeCompletion :=
  TYPE ident '=' POINTER TO targetTypeIdent ;
```

where the type identifier in the definition matches that in the declaration.

4.3 Alias Types

An alias type is a synonym for another type. The type for which it is a synonym is called its base type. The syntax for its institution is

```
aliasTypeInstitution :=
  TYPE ident '=' ALIAS OF typeIdent ;
```

An alias type and its base type are identical and thus interchangeable.

4.4 Derived Types

A derived type is a derivative of another type. The type from which it is derived is called its base type. The syntax for its institution is

```
derivedTypeInstitution :=
  TYPE ident '=' ALIAS OF typeIdent ;
```

The type obtains its properties from its base type, except for its identifier. It has the same semantics as its base type. However, a derived type and its base type are different types and the two are **not** interchangeable.

4.5 Subrange Types

A subrange type is a subtype of an ordinal type. The type of which it is a subtype is called its base type. A subrange type obtains its properties from its base type, except for its identifier and its value range. It has the same semantics as its base type. The value range of a subrange type is a subset, but not necessarily a strict subset of the value range of its base type. The syntax for its institution is

```
subrangeTypeInstitution :=
  TYPE ident '=' '[' lowerBound '..' upperBound ']' OF ordinalType ;
alias lowerBound, upperBound = constExpression ;
alias ordinalType = typeIdent ;
```

A subrange type is always upwards ***compatible*** with its base type, but a base type is not downwards ***compatible*** with its subtypes. This restriction exists because any value of a subrange type is always a legal value of its base type but a value of a base type is not necessarily a legal value of its subrange type.

4.6 Enumeration Types

An enumeration type is an ordinal type whose legal values are determined by an identifier list given in its institution clause.

```
enumTypeInstitution :=
  TYPE ident '=' '(' ( '+' enumTypeToExtend ',' )? identList ')' ;
alias enumTypeToExtend = typeIdent ;
```

The identifier list is automatically enumerated, that is, each identifier is assigned an enumeration value. The enumeration value assigned to the leftmost identifier is always zero. The enumeration value assigned to any following identifier is the enumeration value of its predecessor incremented by one.

A type institution clause of the form

```
TYPE Colour = ( Red, Green, Blue );
```

institutes an enumeration type `Colour` with three enumerated values `Red`, `Green` and `Blue` whose enumeration values are 0, 1 and 2 respectively. The value identifiers must be qualified by the enumeration type identifier.

The maximum number of values in an enumeration type is defined by constant `Impl.MaxEnumValues`. Its value is implementation defined but must not be less than 4096.

4.6.1 Enumeration Type Extension

An enumeration type may be instituted as an extension of another enumeration type. Such a type is called a type extension and the type it extends is called its base type. To institute a type extension of an enumeration type, the identifier list within the institution clause is preceded with the identifier of the base type preceded by the `+` symbol. An extension type inherits all the enumerated values of its base type along with their enumeration values.

A type institution clause of the form

```
TYPE MoreColour = ( +Colour, Orange, Magenta, Cyan );
```

institutes an enumeration type `MoreColour` as an extension of enumeration type `Colour` with six enumerated values `Red`, `Green`, `Blue`, `Orange`, `Magenta` and `Cyan` whose enumeration values are 0 to 5 respectively.

An enumeration type is always downwards ***compatible*** with any of its type extensions, but an enumeration type extension is not upwards ***compatible*** with its base type. This restriction exists because any value of a base type is always a legal value of its extension type but a value of an extension type is not necessarily a legal value of its base type.

4.7 Set Types

A set type is a ***collection type*** that represents a mathematical set of unique finite values and whose component type is an enumeration type. The syntax for its institution is

```
setTypeInstitution :=
  TYPE ident '=' SET OF enumTypeIdent ;
```

The component type determines the finite values of the set type. Each value may be stored in the set at most once. The maximum number of elements in a set type is defined by constant `Impl.MaxSetElements`. Its value is implementation defined but must not be less than 128.

4.8 Array Types

An ***array type*** is an indexed ***collection type*** with a ***capacity limit*** and a component type. ***Capacity limit*** and component type are specified in its institution clause.

```
arrayTypeInstitution :=
  TYPE ident '=' ARRAY capacityLimit OF typeIdent ;
```

The ***capacity limit*** determines the maximum number of components that can be stored in an instance of the ***array type***. The component type determines the type of the values that can be stored. The index type is always `LONGCARD`. The index of the first component of an instance of an ***array type*** is always zero.

Semantics

Unlike earlier versions of Modula-2, the number of values stored in an array may change at runtime within the ***capacity limit*** of the ***array type***. To this end, the ***array type*** is internally represented by a ***record type*** that consists of a ***value count*** field and a payload field. The ***value count*** field holds the number of values stored in the array and the payload field holds the values stored in the array.

Values are stored contiguously from the lowest index which is always zero, up to the highest index which is one less than the array's value count and bounded by the ***capacity limit*** of the ***array type***. Values are addressed by their indexed position within the array using subscript notation.

Array types support insert, append, concatenation, slicing and removal operations. A value is insertable and appendable if it is ***compatible*** with the array's component type; an array or array slice is insertable, appendable and concatenable if its component type is ***compatible*** with the target array's component type, provided that the result of the operation will not exceed the ***capacity*** of the target array.

For arrays whose component type is not a character type, the ***value count*** may be obtained using predefined function `COUNT`. For arrays whose component type is a character type, the character count may be obtained using predefined function `LENGTH` which returns the number of characters excluding the `ASCII-NUL` terminator.

4.9 Record Types

A ***record type*** is a compound type whose components are of arbitrary types. Components are called fields. The number of fields is arbitrary. Its institution syntax is

```
recordTypeInstitution :=
  TYPE ident '=' RECORD ( '(' recTypeToExtend ')' )?
    fieldList ( ';' fieldList )* END ;
alias recTypeToExtend = recordTypeIdent ;
alias fieldList = varOrFieldDeclaration ;
```

A ***record type*** may be extensible or non-extensible.

An ***extensible record type*** is instituted by specifying a base type enclosed in parentheses following reserved word `RECORD` in the institution clause. A ***record type*** whose definition lacks a base type specification is non-extensible. An ***extensible record type*** that specifies `NIL` as its base type is called a root type.

Any ***extensible record type*** inherits all the fields from its base type, including any fields the base type may have inherited. A type so defined is also called a type extension of the base type. A type extension is upwards ***compatible*** with its base type, but the base type is not downwards ***compatible*** with its type extensions.

4.9.1 Indeterminate Types

An indeterminate type is a ***pointer type*** whose target type is a ***record type*** whose last field is indeterminate at compile time. An indeterminate field is an open array field whose ***capacity limit*** cannot be determined from its declaration. Instead it is determined by an allocation statement at runtime. The declaration syntax is

```
indeterminateType :=
  POINTER TO RECORD ( fieldList ';' )* indeterminateField END ;
indeterminateField :=
  '+' ident ':' ARRAY capacityField OF typeIdent ;
alias capacityField = ident ;
```

An indeterminate type contains an embedded field of type `LONGCARD` to hold the ***capacity limit*** of the indeterminate field. The capacity field must be explicitly initialised when an instance of an indeterminate type is allocated at runtime.

An instance of an indeterminate type is ***assignment*** and ***copy compatible*** to another if both are of the same type, regardless of their allocated ***capacity***. However, when copying an instance of indeterminate type to another instance of the same type, the copy operation may overflow the target instance's ***capacity***. In case of overflow, a runtime fault will occur.

4.10 Pointer Types

A ***pointer type*** is a reference type. Its legal values are references to instances of another type. The type it references is called its target type. The target type is specified within its institution clause

```
pointerTypeInstitution :=
  TYPE ident '=' POINTER TO typeIdent ;
```

An instance of a ***pointer type*** may store any reference to a value of the target type or the invalid pointer value `NIL`.

4.11 Procedure Types

A procedure type is a reference type whose legal values are references to procedures or functions that match the signature associated with the type. The signature is specified within its institution clause

```
procedureTypeInstitution :=
  TYPE ident '=' PROCEDURE
    ( '(' formalType ( ',' formalType )* ')' )? ( ':' returnedType )? ;
```

The instance of a procedure type may store any reference to a procedure or function whose signature matches the procedure signature associated with the procedure type or the invalid pointer value `NIL`.

The signatures of a procedure and a procedure type match if their respective formal types exactly match, that is, including any formal type attributes and including any return type.

4.12 Formal Types

A type specification for a procedure parameter is called a formal type. It may be associated with a formal type attribute. A formal type attribute determines the parameter passing convention of parameters of the formal type.

```
formalType := ( CONST | VAR )? nonAttrFormalType ;
```

A formal type may be a type identifier, an ***open array type***, a casting formal type or a variadic formal type.

```
nonAttrFormalType :=
  simpleFormalType | castingFormalType | variadicFormalType ;
simpleFormalType :=
  ( ARRAY OF )? typeIdent ;
```

4.12.1 Formal Open Array Types

An ***open array type*** is an ***array type*** whose ***capacity limit*** is indeterminate at compile time. The ***capacity*** and ***value count*** of a parameter of a formal ***open array type*** are determined at runtime when an argument is passed to it.

```
formalOpenArrayType := ARRAY OF typeIdent ;
```

4.12.2 Casting Formal Types

A casting formal type is a formal type with implicit cast semantics. An argument that is passed to a parameter of a casting formal type is implicitly cast to the formal type. The use of casting formal type syntax must be enabled by import of special module `UNSAFE`.

There are two casting formal types: `CAST` `ADDRESS` and `CAST` `OCTETSEQ`.

```
castingFormalType := CAST ( ADDRESS | OCTETSEQ ) ;
```

4.12.2.1 Formal Type CAST ADDRESS

Formal type `CAST` `ADDRESS` imposes an implicit cast to type `UNSAFE.ADDRESS`.

For semantics see section 11.2.5.1 CAST ADDRESS in chapter Low-Level Facilities.

4.12.2.2 Formal Type CAST OCTETSEQ

Formal type `CAST` `OCTETSEQ` imposes an implicit cast to a sequence of octets.

For semantics see section 11.2.5.2 CAST OCTETSEQ in chapter Low-Level Facilities.

4.12.3 Variadic Formal Types

A variadic formal type is a formal list type whose arity is indeterminate. Its argument count cannot be determined from its declaration and may vary. A parameter of a formal variadic type may be passed a variable number of arguments. The argument count is determined when a procedure with a variadic formal parameter is invoked.

```
variadicFormalType := ARGLIST OF typeIdent ;
```

Chapter 5

Procedures

A procedure is a sequence of statements associated with an identifier and it may be invoked from any scope in which its identifier is visible. A procedure has its own local scope in which local entities may be declared. It may have zero or more associated parameters and it may return a result in its own name. A procedure that returns a result in its own name is called a function procedure, or simply a function. A procedure that does not is called a regular procedure. Collectively both are referred to as procedures.

5.1 Procedure Institution

5.1.1 Procedure Definition

A procedure definition specifies a procedure header only.

```
alias procedureDefinition = procedureHeader ;
procedureHeader :=
  PROCEDURE procedureSignature ;
procedureSignature :=
  ident ( '(' formalParams ( ';' formalParams )* ')' )? ( ':' returnedType )? ;
```

5.1.2 Procedure Declaration

A procedure declaration specifies a procedure header followed by a procedure body.

```
procedureDeclaration :=
  procedureHeader ';' block ident ;
```

5.1.2.1 The Procedure Header

A procedure header consists of a procedure's identifier, its formal parameters if any, and its return type if any. The procedure header represents the interface of the procedure.

5.1.2.2 The Procedure Body

A procedure body consists of a local block followed by the procedure's identifier. The procedure body represents the implementation of the procedure.

5.1.2.3 Formal Parameters

Parameter declarations within the parameter list of a procedure header are called the procedure's formal parameters. Formal parameters determine the number, order, type and passing convention of arguments that may be passed to a procedure when it is invoked.

```
formalParams :=
  ( CONST | VAR )? identList ':' nonAttrFormalType ;
```

The identifier of a formal parameter is the identifier by which the value of the passed in argument or argument list can be referenced within the body of the procedure.

5.1.2.4 Parameter Passing Conventions

There are three parameter passing conventions.

- pass by value
- pass by reference, mutable
- pass by reference, immutable

5.1.2.5 Pass By Value

The default parameter passing convention is pass by value. It is used when no attribute is specified for a formal parameter or formal parameter list. Such a parameter is called a value parameter. When an argument is passed to a value parameter, a copy is passed to the procedure. The scope of the copy is the procedure's local scope.

5.1.2.6 Pass By Reference – Mutable

The pass by mutable reference convention is used when the `VAR` attribute is specified for a formal parameter or formal parameter list. Such a parameter is called a `VAR` parameter. When an argument is passed to a `VAR` parameter, a mutable reference to the argument is passed to the procedure. The procedure may or may not modify the argument. Immutable entities may therefore not be passed to `VAR` parameters.

5.1.2.7 Pass By Reference – Immutable

The pass by immutable reference convention is used when the `CONST` attribute is specified for a formal parameter or formal parameter list. Such a parameter is called a `CONST` parameter. When an argument is passed to a `CONST` parameter, an immutable reference to the argument is passed to the procedure. The procedure may not modify the argument. That is, within the scope of the procedure the `CONST` parameter is treated as if it was a constant. Both mutable and immutable entities may be passed to `CONST` parameters.

5.1.2.8 Formal Open Array Parameters

A formal open array parameter is a formal parameter of a formal ***open array type***. Its ***capacity*** is indeterminate at compile time. ***Capacity*** and ***value count*** are determined at runtime when an argument is passed to it in a procedure call.

An argument passed to an open array parameter must be an array whose value type is ***compatible*** with the value type of the parameter to which it is passed.

Given the declarations

```
TYPE Chars = ARRAY 100 OF CHAR;
TYPE Buffer = ARRAY 100 OF OCTET;
VAR chars : Chars; buffer : Buffer;
PROCEDURE PadWithZeroes ( VAR array : ARRAY OF OCTET );
```

variable `buffer` may be passed to parameter `array` in a call to procedure `PadWithZeroes` because their value types match, but variable `chars` may not be passed to `array` because their value types are incompatible.

```
chars := "abc"; buffer := { 1, 2, 3 };
PadWithZeroes(buffer); (* OK *)
PadWithZeroes(chars); (* compile time error : incompatible type *)
```

The actual ***capacity*** is passed as a hidden parameter immediately preceding the open array parameter.

```
PROCEDURE PadWithZeroes
  ( (*hidden capacity : LONGCARD;*) VAR array : ARRAY OF OCTET );
```

Its value may be obtained within the procedure body by calling function `CAPACITY` with the identifier of the open array parameter as argument.

```
PROCEDURE PadWithZeroes ( VAR array : ARRAY OF OCTET );
BEGIN
  WHILE COUNT(array) < CAPACITY(array) DO
    APPEND(array, 0)
  END (* WHILE *)
END PadWithZeroes;
```

5.1.2.9 Casting Formal Parameters

A casting formal parameter is a formal parameter of a casting formal type. It causes an argument passed to it to be cast to its formal type. There are two kinds.

5.1.2.9.1 Casting Formal Address Parameter

A casting formal address parameter is a parameter of formal type `CAST` `ADDRESS`.

For semantics see section 11.2.5.1 CAST ADDRESS in chapter Low-Level Facilities.

5.1.2.9.2 Casting Formal Octet Sequence Parameter

A casting formal ***octet sequence*** parameter is a parameter of formal type `CAST` `OCTETSEQ`.

For semantics see section 11.2.5.2 CAST OCTETSEQ in chapter Low-Level Facilities.

5.1.2.10 Variadic Formal Parameters

A variadic formal parameter is a formal parameter of a variadic formal type to which a variable number of arguments may be passed. All arguments must be ***passing compatible*** to the value type of the variadic formal parameter to which they are passed.

```
PROCEDURE average ( args : ARGLIST OF REAL ) : REAL;
...
v := average(1.2, 3.4, 5.6); (* OK *)
v := average(1.2, 3); (* compile time error: incompatible type *)
```

The actual argument count is passed as a hidden parameter immediately preceding the argument list.

```
PROCEDURE average ( (*hidden argc : LONGCARD;*) args : ARGLIST OF REAL ) : REAL;
```

Its value may be obtained within the procedure body by calling function `COUNT` with the identifier of the formal variadic parameter as argument.

```
PROCEDURE Variadic ( args : ARGLIST OF T );
BEGIN
  WRITE "number of arguments: ", COUNT(args);
```

A variadic parameter is an indexed collection. It may be empty. A `FOR` statement may be used to iterate over its values.

```
PROCEDURE average ( args : ARGLIST OF REAL ) : REAL;
VAR sum : REAL;
BEGIN
  IF COUNT(args) = 0 THEN
    RETURN 0.0
  END; (* IF *)
  sum := 0.0;
  FOR value IN args DO
    sum := sum + value
  END; (* FOR *)
  RETURN sum / COUNT(args)
END average;
```

As with other arrays, values may also be addressed by subscript. The index of the first value is always zero.

```
FOR index, value IN args DO
  sum := sum + args[index] (* args[index] is equivalent to value *)
END; (* FOR *)
```

An argument list passed to a variadic formal parameter may be enclosed in curly braces to delineate it from other arguments. Such delineation is required whenever the arguments passed to a variadic formal parameter are not distinguishable by their type from the argument that follows the list.

Given a formal parameter list of the form

```
PROCEDURE P ( list1, list2 : ARGLIST OF REAL; x : LONGREAL );
```

arguments to be passed to parameters list1 and list2 in a procedure call to `P` are not distinguishable by type from their following arguments and must be delineated by curly braces.

```
P({0.1, 2.3}, {3.4, 5.6, 7.8}, 9.0);
```

Chapter 6

Statements

A statement represents an action that can be executed to cause a transformation of the computational state of a program. Statements are used for their effects only, they do not return values and they may not occur within expressions.

6.1 The Assignment Statement

The assignment statement is a value-mutator statement. It assigns a given value to a mutable value.

```
assignment :=
  designator ':=' expression ;
```

The designator is called the L-value, the expression is called the R-value. The L-value must be mutable and the R-value must be ***assignment compatible*** to the L-value. If these conditions are not met, a compile time error will occur.

6.2 The INC/DEC Statement

The INC/DEC statement is a value-mutator statement. It consists of a designator followed by a mutator symbol, where `++` increments and `--` decrements the designator by one. The designator is an L-value. It must be mutable and of a ***whole number type***. If these conditions are not met, a compile time error will occur.

```
incOrDecStatement :=
  designator ( '++' | '--' );
```

6.3 The COPY Statement

The `COPY` statement is a value-mutator statement. It copies values across type boundaries where types have a common or ***compatible*** value type. Its syntax is

```
copyStatement :=
  COPY designator ':=' expression ;
```

The designator is called the L-value, the expression is called the R-value. The L-value must be mutable. The R-value must be ***copy compatible*** to the L-value. Both must be non-elementary types. If these conditions are not met, a compile time fault will occur.

6.3.1 Copying Sets

Variables of different set types are not ***assignment compatible*** but they are ***copy compatible*** if their value types match or one is an extension or subtype of the other.

Given the declarations

```
TYPE BaseColour = ( Red, Green, Blue );
TYPE ExtColour = ( +BaseColour, Orange, Magenta, Cyan );
TYPE BaseColours = SET OF BaseColour; ExtColours = SET OF ExtColour;
VAR baseColours : BaseColours; extColours : ExtColours;
```

a statement of the form

```
COPY extColours := baseColours;
```

copies set `baseColours` to set `extColours`.

6.3.2 Copying Arrays

Variables of different ***array types*** are not ***assignment compatible*** but they are ***copy compatible*** if their value types match or one is an extension or subtype of the other.

Given the declarations

```
TYPE PosInt = [0..TMAX(INTEGER)] OF INTEGER;
TYPE ArrayA = ARRAY 20 OF INTEGER; ArrayB = ARRAY 10 OF PosInt;
VAR arrayA : ArrayA; arrayB : ArrayB;
```

a statement of the form

```
COPY arrayA := arrayB;
```

copies array `arrayB` to array `arrayA`.

6.3.3 Copying Between Sets and Arrays

Variables of array and set types are not ***assignment compatible*** with each other but they are ***copy compatible*** if their value types are ***copy compatible***.

Given the declarations

```
TYPE Token = ( Unknown, Alias, And, Arglist, Array, Begin ... );
TYPE TokenSet = SET OF Token; TokenArray = ARRAY 100 OF Token;
VAR set : TokenSet; array : TokenArray;
```

a statement of the form

```
COPY set := array;
```

copies the `Token` values stored in `set` to `array`.

6.4 The Procedure Call Statement

A procedure call statement is used to invoke a procedure. It consists of a designator that designates the called procedure, optionally followed by a list of parameters enclosed in parentheses to be passed to the procedure. Parameters passed are called arguments or actual parameters, those defined in the procedure's header are called formal parameters.

In every procedure call, the types of actual and formal parameters must match. If they don't, a compile time error shall occur. Procedure calls may be recursive, that is, a procedure may call itself within its body. Recursive calls shall be optimised by tail call elimination (TCE), except when generating source code in a target language that does not support TCE.

6.5 The RETURN Statement

The `RETURN` statement is a flow-control statement used within a procedure body to return control to its caller and in the main body of the program to return control to the operating environment that activated the program.

Depending on the type of procedure in which a `RETURN` statement is used, it may or may not return a value. When returning from a regular procedure, no value may be returned. When returning from a function procedure a value of the procedure's return type must be returned. Non-compliance will cause a compile time error.

6.6 The NEW Statement

The `NEW` statement is a memory management statement. It allocates storage for a new instance of the target type referenced by the type of its argument. Its syntax is

```
newStmt :=
  NEW designator ( ':=' initValue | CAPACITY expression )? ;
```

where `designator` must be an L-value and `initValue` must be an R-value.

A statement of the form

```
NEW p;
```

allocates a new instance of the target type of pointer `p` and passes a reference for the new instance in `p`.

6.6.1 Allocation with Initialisation

By default, a newly allocated instance is not initialised. To initialise a new instance during its allocation, an initialisation value may be specified within a `NEW` statement.

A statement of the form

```
NEW array := { a, b, c };
```

allocates a new instance, initialises it with values `a`, `b` and `c`, and passes a reference to it in `array`.

6.6.2 Allocation of Instances of Indeterminate Type

A capacity value may be specified to initialise a new instance of an indeterminate type.

A statement of the form

```
NEW buffer CAPACITY 1000;
```

allocates a new instance of an indeterminate type with a ***capacity limit*** of 1000 components of the type's indeterminate field and passes a reference for the new instance in `buffer`.

6.7 The RETAIN Statement

The `RETAIN` statement is a memory management statement. It prevents premature deallocation of the target referenced by its argument.

6.8 The RELEASE Statement

The `RELEASE` statement is a memory management statement. It cancels a prior invocation of `RETAIN` for the target referenced by its argument and deallocates the target if all prior invocations of `RETAIN` for the target have been cancelled.

6.9 The IF Statement

The IF statement is a flow-control statement that passes control to one of a number of blocks within its body depending on the value of a boolean expression. It evaluates a condition in form of a boolean expression. If the condition is `TRUE` then program control passes to its `THEN` block. If the condition is `FALSE` and an `ELSIF` branch follows, then program control passes to the `ELSIF` branch to evaluate that branch's condition.

Again, if the condition is `TRUE` then program control passes to the `THEN` block of the `ELSIF` branch. If there are no `ELSIF` branches, or if the conditions of all `ELSIF` branches are `FALSE`, and if an `ELSE` branch follows, then program control passes to the `ELSE` block. At most one block in the statement is executed. `IF`-statements must always be terminated with an `END`.

6.10 The CASE Statement

The `CASE` statement is a flow-control statement that passes control to one of a number of labeled statements or statement sequences depending on the value of an ordinal expression.

```
caseStatement :=
  CASE expression OF ( '|' case )+ ( ELSE statementSequence )? END ;
case :=
  caseLabels ( ',' caseLabels )* : StatementSequence ;
caseLabels :=
  constExpression ( .. constExpression )? ;
```

Control is passed to the first statement following the case label that matches the ordinal expression. Case labels must be unique. There is no *"fall through"*. At the end of a label, control is passed to the first statement after the `CASE` statement.

If no case label matches, control is passed to the `ELSE` block, or, if there is no `ELSE` block, to the first statement after the `CASE` statement.

6.11 The LOOP Statement

The `LOOP` statement is a flow-control statement used to repeat a statement or statement sequence indefinitely unless explicitly terminated by an `EXIT` statement within its body.

6.12 The WHILE Statement

The `WHILE` statement is a flow-control statement used to repeat a statement or statement sequence depending on a condition in form of a boolean expression. The expression is evaluated each time before the `DO` block is executed. The `DO` block is repeated as long as the expression evaluates to `TRUE` unless the statement is explicitly terminated by an `EXIT` statement within its body.

6.13 The REPEAT Statement

The `REPEAT` statement is is a flow-control statement used to repeat a statement or statement sequence depending on a condition in form of a boolean expression. The expression is evaluated each time after the `REPEAT` block has executed. The `REPEAT` block is repeated as long as the expression evaluates to `TRUE` unless the statement is explicitly terminated by an `EXIT` statement within its body.

6.14 The FOR Statement

The `FOR` statement is a flow-control statement that iterates over an iterable entity, executing a statement or statement sequence during each iteration cycle. It consists of a loop header and a loop body. The header consists of one or two loop variants, an optional iteration order, and an iterable expression. The body consists of a statement or statement sequence.

```
forStatement :=
  FOR forLoopVariants IN iterableExpr DO statementSequence END ;
```

6.14.1 The Loop Variants

The loop variant section contains one or two identifiers through which indices and values of the iterable expression are referenced within the loop.

```
forLoopVariants :=
  indexOrSoleValue ascOrDesc? ( ',' value )? ;
alias indexOrSoleValue, value = ident ;
```

The loop variant identifiers are declared by the loop header and they are only in scope within the header and body. Once a `FOR` loop has terminated, its loop variants are no longer in scope. The number of possible loop variants depends on the loop's iterable expression.

If the iterable expression is an ordinal type, a subrange of an ordinal type or the designator of a set, the header contains a sole loop variant, immutable within the loop body.

If the iterable expression is the designator of an array `a`, the header contains a loop variant representing the iteration index `i` and an optional second loop variant `v` representing the array component `a[i]`. Index `i` is immutable within the loop body. Value `v` is mutable if array `a` is mutable, otherwise `v` is immutable.

During the first iteration cycle the loop variant section references that index, value or index/value pair which is first for the prevailing iteration order. Before each subsequent iteration cycle the loop variant section is advanced to its successor for the prevailing iteration order. Iteration continues until all indices, values or pairs have been visited unless the `FOR` statement is explicitly terminated by an `EXIT` statement within its body.

6.14.2 The Iteration Order

The prevailing iteration order may be specified by an ascender or descender following the first loop variant. An ascender imposes ascending order and is denoted by the `++` symbol. A descender imposes descending order and is denoted by the `--` symbol. When omitted, the iteration order is ascending.

```
ascOrDesc := '++' | '--' ;
```

6.14.3 The Iterable Expression

The iterable expression — or iterable in short — is denoted by (1) the identifier of an ordinal or subrange type, (2) an anonymous subrange of an ordinal type or (3) the designator of a set, array or array slice.

```
iterableExpr :=
  ordinalRange OF ordinalType | designator ;
```

6.14.4 Iterating over Ordinal Types

If the iterable is an ordinal type or a subrange thereof, only one loop variant may be given. The loop variant is immutable. Its type is the ordinal type or subrange given and the loop iterates over all values of the iterable.

A statement of the form

```
FOR char IN CHAR DO
  WRITE char
END; (* IF *)
```

iterates over all values of type `CHAR`.

Given the declaration

```
TYPE Colour = ( Red, Green, Blue );
```

a statement of the form

```
FOR colour IN Colour DO
  WRITE nameOfColour(colour), "\n"
END; (* IF *)
```

iterates over all enumerated values of type `Colour`.

A statement of the form

```
FOR value-- IN [1..99] OF CARDINAL DO
  WRITE value, " bottles of beer, take one down and pass it around.\n"
END; (* FOR *)
```

iterates over subrange `[1..99]` of type `CARDINAL` in descending order.

6.14.5 Iterating over Collections

If the iterable is the designator of a set, only one loop variant may be given. The loop variant is immutable. Its type is the element type of the set and the loop iterates over all values in the set.

A statement of the form

```
FOR elem IN set DO
  WRITE nameOfElem(elem), "\n"
END; (* IF *)
```

iterates over all elements stored in `set`.

If the iterable is the designator of an array or array slice, one or two loop variants may be given. The first loop variant is immutable and of type `CARDINAL`. The optional second loop variant is mutable if the array is mutable, otherwise immutable, its type is the element type of the array. The loop iterates over all components stored in the array.

A statement of the form

```
FOR index, value IN source DO
  target[index] := value
END; (* IF *)
```

iterates over all components of `array`.

A statement of the form

```
FOR index, value IN source[n..m] DO
  WRITE "source[", index, "] = ", value, "\n"
END; (* IF *)
```

iterates over all components in array slice `source[n..m]`.

6.15 The EXIT Statement

The `EXIT` statement is a control-flow statement used within the body of a `LOOP`, `WHILE`, `REPEAT` or `FOR` statement to terminate execution of the loop and transfer control to the first statement after the loop body. The `EXIT` statement may only occur within the body of loop statements. Non-compliance shall cause a compile time fault.

6.16 The READ Statement

The `READ` statement reads one or more values from a communications channel and transfers the values in a non-empty variadic list of designators. Its syntax is

```
readStmt :=
  READ ( '@' chan ':' )?
  NEW? designator ( ',' designator )*
```

where `chan` is the designator of a communications channel.

A statement of the form

```
READ @file : foo, bar, baz;
```

reads three values from channel `file` and passes them in variables `foo`, `bar` and `baz`.

The communications channel may be omitted in which case a default input channel is used.

A statement of the form

```
READ foo;
```

reads a value from the default input channel and passes it in variable `foo`.

The list of designators may be prefixed by reserved word `NEW` to allocate new memory for each value read. In this case every designator must designate a value of a ***pointer type*** and its value must be `NIL`.

A statement of the form

```
READ @file : NEW ptr;
```

reads a value from channel `file`, allocates a new instance of `ptr` and passes the read value in `ptr^`.

6.17 The WRITE Statement

The `WRITE` statement writes one or more values to a communications channel. Its syntax is

```
writeStmt :=
  WRITE ( '@' chan ':' )?
  outputArgs ( ',' outputArgs )* ;
outputArgs :=
  formattedArgs | unformattedArgs ;
formattedArgs :=
  '#' '(' fmtStr, expressionList ')' ;
alias unformattedArgs = expressionList ;
```

where `chan` is the designator of a communications channel and `fmtStr` is a format specifier string.

A statement of the form

```
WRITE @file : foo, bar, baz;
```

writes the values `foo`, `bar` and `baz` to channel `file`.

The communications channel may be omitted in which case a default output channel is used.

A statement of the form

```
WRITE foo;
```

writes the value of `foo` to the default output channel.

The list of output values may include formatted and unformatted values. A formatted value or value list is preceded by a format specifier, enclosed in parentheses and preceded by `#`.

A statement of the form

```
WRITE @file : "Price: ", #("5:3;2", price), "incl. VAT\n";
```

writes three values to channel `file` applying format specifier `"5:3;2"` to value `price`.

Format specifiers are library defined but language specified for all built-in types, their interpretation takes place within the IO library. The syntax of format specifiers for built-in types is described in Appendix F.

6.18 The NOP Statement

The `NOP` statement represents an explicit empty statement.*

```
emptyStmt := NOP ;
```

Loop bodies without any statements do not show author intent and may well indicate incomplete code. The grammar therefore forbids any such empty constructs altogether. Whenever an empty statement is intended, the `NOP` statement must be used.

```
(* skip all whitespace in stream *)
WHILE stream.nextChar() = ' ' DO
  NOP
END; (* WHILE *)
```

*An implementation may provide a compiler switch to control whether the `NOP` statement is ignored or translated into an empty statement or no-operation opcode if supported by the target language or instruction set of the target architecture.

Chapter 7

Operators and Expressions

7.1 Operator Precedence and Associativity

Expressions are evaluated according to the precedence level and associativity of the operators within the expression. Operators of higher precedence are evaluated before operators of lower precedence. Where operators have the same precedence, their associativity determines evaluation order. For left-associative operators of the same precedence the evaluation order is left to right.

The properties of operators are given in the table below:

Operator	Math Symbols	Meaning	Associativity	Arity	Position	Precedence
`=`	$=$	equality	none	binary	infix	1 (lowest)
`#`	$\neq$	inequality				
`<`	$< \subset \prec$	less, strict subset, ranks before				
`<=`	$\leq \subseteq \preceq$	less/equal, subset, ranks before/equal				
`>`	$> \supset \succ$	greater, strict superset, ranks after				
`>=`	$\geq \supseteq \succeq$	greater/equal, superset, ranks after/equal				
`==`	$\equiv$	identity				
`IN`	$\in$	set membership				
`+`	$+ \cup$	addition, set union	left	binary	infix	2
`-`x	$-$	arithmetic negation	none	unary	prefix	
x`-`y	$-$	subtraction	left	binary	infix	2
`&`	$\|\|$	concatenation				
`\`	$\setminus$	set difference				
`OR`	$\vee$	logical disjunction				
`*`	$\times \cap$	multiplication, set intersection	left	binary	infix	3
`/`	$\div \triangle$	division, symmetric set difference				
`DIV`	$\div$	integer division				
`MOD`	mod	modulus				
`AND`	$\wedge$	logical conjunction				
`NOT`	$\neg$	logical negation	none	unary	prefix	4
`::`		type conversion	none	binary	infix	5
`( )`		sub-expression precedence	none	unary	bifix	6
`[ ]`	a_i	subscript	left	unary	postfix	6
`[ ..]`		insertion				
`^`		dereference				
`.`		field selection	left	binary	infix	6 (highest)

Table 7.1: Operators and their properties

7.2 Relational Operations

7.2.1 Equality

The binary infix operator `=` denotes an equality test. An expression of the form `a = b` returns `TRUE` if `a` and `b` are equal, otherwise `FALSE`. If both operands are arrays, or both operands are sets, their component types must be the same. Otherwise, both operands must be of the same type.

7.2.2 Inequality

The binary infix operator `#` denotes an inequality test. An expression of the form `a # b` returns `TRUE` if `a` and `b` are not equal, otherwise `FALSE`. If both operands are arrays, or both operands are sets, their component types must be the same. Otherwise, both operands must be of the same type.

7.2.3 Less-Than

The binary infix operator `<` denotes a less-than comparison when the operands are of a ***scalar type***. An expression of the form `a < b` returns `TRUE` if the value of `a` is less than the value of `b`, otherwise `FALSE`. Both operands must be of the same type.

7.2.4 Precedes

The binary infix operator `<` denotes a collation order comparison when the operands represent character strings. An expression of the form `a < b` returns `TRUE` if `a` ranks before `b` using ASCII collation order, otherwise `FALSE`.

7.2.5 Strict Subset

The binary infix operator `<` denotes a strict subset comparison when the operands are of a set type. An expression of the form `a < b` returns `TRUE` if $a \subset b$ otherwise `FALSE`. The component types of the operands must be of the same type.

7.2.6 Less-Than-Or-Equal

The binary infix operator `<=` denotes a less-than-or-equal comparison when the operands are of a ***scalar type***. An expression of the form `a <= b` returns `TRUE` if the value of `a` is less than or equal to the value of `b`, otherwise `FALSE`. Both operands must be of the same type.

7.2.7 Precedes-Or-Equal

The binary infix operator `<=` denotes a collation order comparison when the operands represent character strings. An expression of the form `a <= b` returns `TRUE` if `a` ranks before `b` or if both rank equally using ASCII collation order, otherwise `FALSE`.

7.2.8 Subset

The binary infix operator `<=` denotes a subset comparison when the operands are of a set type. An expression of the form `a <= b` returns `TRUE` if $a \subseteq b$, otherwise `FALSE`. The component types of the operands must be of the same type.

7.2.9 Greater-Than

The binary infix operator `>` denotes a greater-than comparison when the operands are of a ***scalar type***. An expression of the form `a > b` returns `TRUE` if the value of `a` is greater than the value of `b`, otherwise `FALSE`. Both operands must be of the same type.

7.2.10 Succeeds

The binary infix operator `>` denotes a collation order comparison when the operands represent character strings. An expression of the form `a > b` returns `TRUE` if `a` ranks after `b` using ASCII collation order, otherwise `FALSE`.

7.2.11 Strict Superset

The binary infix operator `>` denotes a strict superset comparison when the operands are of a set type. An expression of the form `a > b` returns `TRUE` if $a \supset b$, otherwise `FALSE`. The component types of the operands must be of the same type.

7.2.12 Greater-Than-Or-Equal

The binary infix operator `>=` denotes a greater-than-or-equal comparison when the operands are of a ***scalar type***. An expression of the form `a >= b` returns `TRUE` if the value of `a` is greater than or equal to the value of `b`, otherwise `FALSE`. Both operands must be of the same type.

7.2.13 Succeeds-Or-Equal

The binary infix operator `>=` denotes a collation order comparison when the operands represent character strings. An expression of the form `a >= b` returns `TRUE` if `a` ranks after `b` or if both rank equally using ASCII collation order, otherwise `FALSE`.

7.2.14 Superset

The binary infix operator `>=` denotes a superset comparison when the operands are of a set type. An expression of the form `a > b` returns `TRUE` if $a \supseteq b$, otherwise `FALSE`. The component types of the operands must be of the same type.

7.2.15 Identity

The binary infix operator `==` denotes an identity test. An expression of the form `a == b` returns `TRUE` if `UNSAFE.ADR(a)` equals `UNSAFE.ADR(b)`, otherwise `FALSE`.

7.2.16 Set Membership

The binary infix operator `IN` denotes a set membership test. An expression of the form `a IN set` returns `TRUE` if $a \in set$, otherwise `FALSE`. The right operand must be of a set type. The left operand must be of the component type of the set type.

7.3 Logical Operations

7.3.1 Disjunction

The binary infix operator `OR` denotes logical disjunction. An expression of the form `a OR b` returns the logical disjunction $a \vee b$. Both operands must be of type `BOOLEAN`.

7.3.2 Conjunction

The binary infix operator `AND` denotes logical conjunction. An expression of the form `a AND b` returns the logical conjunction $a \wedge b$. Both operands must be of type `BOOLEAN`.

7.3.3 Logical Negation

The unary prefix operator `NOT` denotes logical negation. An expression of the form `NOT a` returns the logical inverse $\neg a$. The operand must be of type `BOOLEAN`.

7.4 Arithmetic Operations

7.4.1 Addition

The binary infix operator `+` denotes addition when its operands are of numeric type. An expression of the form `a + b` returns the sum of `a` and `b`. Both operands must be of ***compatible*** numeric types.

7.4.2 Arithmetic Negation

The unary prefix operator `-` denotes arithmetic negation. An expression of the form `-a` returns the complement of `a`. The operand must be of a signed numeric type.

A unary `-` operator may only appear before a multi-term expression if the expression is enclosed in parentheses. This resolves an ambiguity in earlier versions of Modula-2.

7.4.3 Subtraction

The binary infix operator `-` denotes subtraction. An expression of the form `a - b` returns the difference of `a` and `b`. Both operands must be of ***compatible*** numeric types.

7.4.4 Multiplication

The binary infix operator `*` denotes multiplication when its operands are of numeric type. An expression of the form `a * b` returns the product of `a` and `b`. Both operands must be of ***compatible*** numeric types.

7.4.5 Real Number Division

The binary infix operator `/` denotes real number division when its operands are of a ***real number type***. An expression of the form `a / b` returns the quotient of `a` and `b`. Both operands must be of ***compatible*** numeric types.

7.4.6 Integer Division

The binary infix operator `DIV` denotes Euclidean integer division. An expression of the form `a DIV b` returns the quotient of `a` and `b`. Both operands must be of a ***compatible*** numeric types.

7.4.7 Modulus

The binary infix operator `MOD` denotes the modulus of Euclidean integer division. An expression of the form `a MOD b` returns the modulus of `a` and `b`. Both operands must be of a ***compatible*** numeric types.

7.5 Set Operations

7.5.1 Set Union

The binary infix operator `+` denotes set union when its operands are of a set type. An expression of the form `a + b` returns the set union $a \cup b$. Both operands must be of a set type. The component types of the operands must be of the same type.

7.5.2 Set Difference

The binary infix operator `\` denotes set difference. An expression of the form `a \b` returns the set difference $a \setminus b$. Both operands must be of a set type. The component types of the operands must be of the same type.

7.5.3 Set Intersection

The binary infix operator `*` denotes set intersection when its operands are of a set type. An expression of the form `a * b` returns the set intersection $a \cap b$. Both operands must be of a set type. The component types of the operands must be of the same type.

7.5.4 Symmetric Set Difference

The binary infix operator `/` denotes symmetric set difference when its operands are of a set type. An expression of the form `a / b` returns the symmetric set difference $a \triangle b$. Both operands must be of a set type. The component types of the operands must be of the same type.

7.6 Collection Operations

7.6.1 Insertion

The subscript operator `[ ]` with a subscript followed by suffix `..` denotes insertion. An assignment of the form `a[i..]:=v` inserts value `v` into array `a` at subscript `i`. The operand must be of an ***array type***. An insertion designator may only appear as an L-value on the left side of an assignment.

7.6.2 Concatenation

The binary infix operator `&` denotes concatenation. An expression of the form `a & b` returns the concatenation product of `a` and `b`. Both operands must be of a ***collection type***. The component types of the operands must be copy compatible to the L-Value to which the expression is copied or passed to.

7.7 Miscellaneous Operations

7.7.1 Type Conversion

The binary infix operator `::` denotes type conversion. An expression of the form `T::v` converts value `v` to type `T` and returns the converted value. The left operand must be a type identifier. The right operand may be a value or expression of any convertible type.

7.7.2 Sub-Expression Precedence

The bifix operator `(` ... `)` denotes sub-expression precedence. In an expression of the form `a □ (b ○ c)`, the evaluation of sub-expression `(b ○ c)` takes precedence for any arbitrary operators □ and ○ regardless of their precedence levels.

7.8 Designation Operations

7.8.1 Subscript Addressing

The postfix operator `[` ... `]` denotes subscript addressing. The operator encloses a subscript or a range of subscripts. The operand must be the designator of an array. Subscripts must be of a ***whole number type***.

A sole subscript addresses a single value. A paired subscript addresses a range of values.

- An expression of the form `a[i]` designates the value of `a` at subscript `i`
- An expression of the form `a[i..j]` designates the value range of `a` from subscript `i` to `j`

Non-negative subscripts represent positions relative to the first component.

- $\boldsymbol{index}\ i =$ `i::LONGCARD` $\forall\ i \in \{\tau_{whole} \mid 0 \leq i \leq \boldsymbol{count}\ a \wedge i < \boldsymbol{capacity}\ a\}$

Negative subscripts represent positions relative to the position after the last component.

- $\boldsymbol{index}\ i =$ `COUNT(a)-ABS(i)::LONGCARD` $\forall\ i \in \{\tau_{whole} \mid i < 0 \wedge \boldsymbol{abs}\ i < \boldsymbol{count}\ a\}$

where `a` is the array operand and `i` a subscript value.

7.8.2 Pointer Dereference

The unary postfix operator `^` denotes pointer dereference. An expression of the form `p^` designates the entity referenced by pointer `p` where `p` must be of a ***pointer type***.

7.8.3 Field Selection

The binary infix operator `.` denotes field selection when the left operand is an instance of a ***record type***. An expression of the form `a.b` designates field `b` of record `a`.

7.8.4 Name Selection

The binary infix operator `.` denotes name selection when the left operand is a qualifier of a qualified identifier. An expression of the form `a.b` designates `b` qualified by `a`.

7.9 Operations By Type

7.9.1 Elementary Types

	=	#	<	<=	>	>=	+	-x	x-y	*	/	DIV	MOD	AND	OR	NOT	::
BOOLEAN	•	•	•	•	•	•	–	–	–	–	–	–	–	•	•	•	–
Other Enumerations	•	•	•	•	•	•	–	–	–	–	–	–	–	–	–	–	–
Character Types	•	•	•	•	•	•	–	–	–	–	–	–	–	–	–	–	•
Cardinal Types	•	•	•	•	•	•	•	–	•	•	–	•	•	–	–	–	•
Integer Types	•	•	•	•	•	•	•	•	•	•	–	•	•	–	–	–	•
Real Number Types	•	•	•	•	•	•	•	•	•	•	•	–	–	–	–	–	•

Table 7.2: Elementary types and their supported operators

	ORD()	PRED() SUCC()	ABS()	ODD()	SGN()	POW2() LOG2()	ENTIER()	TMIN() TMAX()	TSIZE()
BOOLEAN	•	•	–	–	–	–	–	•	•
Other Enumerations	•	•	–	–	–	–	–	•	•
Character Types	•	•	–	–	–	–	–	•	•
Cardinal Types	•	•	–	•	–	•	–	•	•
Integer Types	•	•	–	•	•	–	–	•	•
Real Number Types	–	–	•	–	•	–	•	•	•

Table 7.3: Elementary types and their supported pervasive functions and macros

	ADD()	SUB()	SHL()	SHR()	BIT()	SETBIT()	BWNOT()	BWAND()	BWOR()
BOOLEAN	–	–	–	–	–	–	–	–	–
Other Enumerations	–	–	–	–	–	–	–	–	–
Character Types	•	•	•	•	•	•	•	•	•
Cardinal Types	•	•	•	•	•	•	•	•	•
Integer Types	•	•	•	•	•	•	•	•	•
Real Number Types	•	•	•	•	•	•	•	•	•

Table 7.4: Elementary types and their supported UNSAFE procedures and functions

7.9.2 Compound Types

	=	#	<	<=	>	=>	==	IN	+	-	&	\	*	/	a[...]	^	.
SET	•	•	•	•	•	•	•	•	•	–	–	•	•	•	–	–	–
ARRAY	•	•	–	–	–	–	•	–	–	–	•	–	–	–	•	–	–
ARRAY OF CHAR	•	•	•	•	•	•	•	–	–	–	•	–	–	–	•	–	–
ARRAY OF UNICHAR	•	•	•	•	•	•	•	–	–	–	•	–	–	–	•	–	–
RECORD	•	•	–	–	–	–	•	–	–	–	–	–	–	–	–	–	•

Table 7.5: Compound types and their supported operators

	APPEND()	INSERT() REMOVE()	UNIVSET()	CAPACITY()	COUNT()	LENGTH()	TSIZE()	TLIMIT()
SET	–	•	•	•	•	–	•	•
ARRAY	•	–	–	•	•	–	•	•
ARRAY OF CHAR	•	–	–	•	–	•	•	•
ARRAY OF UNICHAR	•	–	–	•	–	•	•	•
RECORD	–	–	–	–	–	–	•	–

Table 7.6: Compound types and their supported pervasive procedures, functions and macros

7.9.3 Pointer Types

	=	#	^	NEW	RELEASE	TSIZE()
OPAQUE	•	•	–	•	•	•
POINTER	•	•	•	•	•	•

Table 7.7: Pointer types and their supported operations

7.9.4 Procedure Types

	=	#	p(...)	TSIZE()
PROCEDURE	•	•	•	•

Table 7.8: Procedure types and their supported operations

7.9.5 Abstract Data Types

	RETAIN	READ WRITE	APPEND	@VALUE() @STORE()	@ATINSERT() @ATREMOVE()	COUNT() LENGTH()	TLIMIT()
ADT	o	o	o	o	o	o	o

Table 7.9: Abstract data types and their library supportable operations

7.9.6 UNSAFE Types

	=	#	<	<=	>	>=	a[i]	a[i..j]	a[i..]	^
BYTE	•	•	•	•	•	•	–	–	–	–
WORD	•	•	•	•	•	•	–	–	–	–
LONGWORD	•	•	•	•	•	•	–	–	–	–
ADDRESS	•	•	•	•	•	•	•	–	–	•
OCTETSEQ	–	–	–	–	–	–	•	–	–	–

Table 7.10: UNSAFE types and their supported operators

	ODD()	LENGTH()	ADD() SUB()	SHL() SHR()	BIT() SETBIT()	BWNOT()	BWAND() BWOR()	TMIN() TMAX()	TSIZE()	TLIMIT()
BYTE	•	–	•	•	•	•	•	•	•	–
WORD	•	–	•	•	•	•	•	•	•	–
LONGWORD	•	–	•	•	•	•	•	•	•	–
ADDRESS	•	–	•	•	•	•	•	•	•	–
OCTETSEQ	–	•	–	–	–	–	–	–	always 0	always 0

Table 7.11: UNSAFE types and their supported built-in procedures, functions and macros

Chapter 8

Type Compatibility

Type ***compatibility*** is a relation between two types τ_1 and τ_2 that determines whether an entity ε_1 of type τ_1 may be assigned to, copied to, passed to or used together in an expression with an entity ε_2 of type τ_2. Type compatibility thus determines the compatibility of two entities by their associated types.

8.1 Compatibility Relations

The compatibility relation between any two entities is congruent to the compatibility relation of their types. Any compatibility relation is always transitive, but not necessarily commutative. There are five compatibility relations.

8.1.1 Assignment Compatibility

Assignment compatibility is a relation between two types τ_1 and τ_2 that determines whether an entity ε_1 of type τ_1 may be assigned to an entity ε_2 of type τ_2. The relation is denoted by the $\overset{:=}{\rightarrow}$ symbol.

$$\tau_1 \overset{:=}{\rightarrow} \tau_2 \land \varepsilon_1 \in: \tau_1 \land \varepsilon_2 \in: \tau_2 \Rightarrow \varepsilon_1 \overset{:=}{\rightarrow} \varepsilon_2 \tag{8.1}$$

8.1.2 Copy Compatibility

Copy compatibility is a relation between two types τ_1 and τ_2 that determines whether an entity ε_1 of type τ_1 may be copied to an entity ε_2 of type τ_2. The relation is denoted by the $\xrightarrow{copy}$ symbol.

$$\tau_1 \xrightarrow{copy} \tau_2 \land \varepsilon_1 \in: \tau_1 \land \varepsilon_2 \in: \tau_2 \Rightarrow \varepsilon_1 \xrightarrow{copy} \varepsilon_2 \tag{8.2}$$

8.1.3 By-Value Passing Compatibility

By-value passing compatibility is a relation between two types τ_1 and τ_2 that determines whether an entity ε_1 of type τ_1 may be passed to a formal by-value parameter ε_2 of type τ_2. The relation is denoted by the $\xrightarrow{val}$ symbol.

$$\tau_1 \xrightarrow{val} \tau_2 \land \varepsilon_1 \in: \tau_1 \land \varepsilon_2 \in: \tau_2 \Rightarrow \varepsilon_1 \xrightarrow{val} \varepsilon_2 \tag{8.3}$$

8.1.4 By-Reference Passing Compatibility

By-reference passing compatibility is a relation between two types τ_1 and τ_2 that determines whether an entity ε_1 of type τ_1 may be passed to a formal by-reference parameter ε_2 of type τ_2. The relation is denoted by the $\xrightarrow{ref}$ symbol.

$$\tau_1 \xrightarrow{ref} \tau_2 \land \varepsilon_1 \in: \tau_1 \land \varepsilon_2 \in: \tau_2 \Rightarrow \varepsilon_1 \xrightarrow{ref} \varepsilon_2 \tag{8.4}$$

8.1.5 Expression Compatibility

Expression compatibility is a relation between two types τ_1 and τ_2 that determines whether two entities ε_1 of type τ_1 and ε_2 of type τ_2 may be used together in a binary expression. The relation is commutative. It is denoted by the $\xleftrightarrow{expr}$ symbol.

$$\tau_1 \xleftrightarrow{expr} \tau_2 \land \varepsilon_1 \in: \tau_1 \land \varepsilon_2 \in: \tau_2 \Rightarrow \varepsilon_1 \xleftrightarrow{expr} \varepsilon_2 \tag{8.5}$$

8.1.6 Hierarchy of Compatibility Relations

Assignment compatibility implies all other compatibility relations. Copy compatibility implies by-value passing compatibility. Passing compatibility implies expression compatibility.

$$\tau_1 \overset{:=}{\Rightarrow} \tau_2 \Rightarrow \begin{cases} \tau_1 \xrightarrow{copy} \tau_2 \Rightarrow \tau_1 \xrightarrow{val} \tau_2 \\ \qquad\qquad\quad \tau_1 \xrightarrow{ref} \tau_2 \end{cases} \Rightarrow \tau_1 \xleftrightarrow{expr} \tau_2 \tag{8.6}$$

8.1.7 Full and Limited Compatibility

Full or *unlimited* compatibility encompasses all available compatibility relations. Conversely, *limited* compatibility encompasses one or more but not all relations. Whenever entities are called ***compatible*** without qualifier, *full* compatibility is implied.

8.2 Type Regimes

A type regime is a set of rules that govern type compatibility. A type regime is called limited if only a subset of its rule set applies. The following type regimes are in use

8.2.1 Strict Name Equivalence

Under strict name equivalence, types are equivalent if their identifiers are identical. Equivalent types are fully ***compatible*** and the relationship is commutative.

$$\Gamma_{name\text{-}equiv} \vdash \boldsymbol{id}\,\tau_1 = \boldsymbol{id}\,\tau_2 \Rightarrow \tau_1 \equiv \tau_2 \Rightarrow \tau_1 \overset{:=}{\Rightarrow} \tau_2 \wedge \tau_2 \overset{:=}{\Rightarrow} \tau_1 \tag{8.7}$$

8.2.2 Super-Type Equivalence

Under super-type equivalence, a type is upwards ***compatible*** with its super-type, but a super-type is not downwards ***compatible*** with its subtype.

$$\Gamma_{supertype\text{-}equiv} \vdash \tau_1 <: \tau_2 \Rightarrow \tau_1 \overset{:=}{\Rightarrow} \tau_2 \wedge \neg\, \tau_2 \overset{:=}{\Rightarrow} \tau_1 \tag{8.8}$$

8.2.3 Component Type Equivalence

Under component type equivalence, compound types are ***copy compatible*** if their component types are ***compatible***.

$$\Gamma_{comp\text{-}equiv} \vdash \boldsymbol{comp\text{-}type}\ \tau_1 \overset{:=}{\Rightarrow} \boldsymbol{comp\text{-}type}\ \tau_2 \Rightarrow \tau_1 \xrightarrow{copy} \tau_2 \wedge \tau_2 \xrightarrow{copy} \tau_1 \tag{8.9}$$

8.2.4 Target Type Equivalence

Under target type equivalence, ***pointer types*** are ***compatible*** if their target types are.

$$\Gamma_{target\text{-}equiv} \vdash \boldsymbol{target\text{-}type}\ \tau_1 \overset{:=}{\Rightarrow} \boldsymbol{target\text{-}type}\ \tau_2 \Rightarrow \tau_1 \overset{:=}{\Rightarrow} \tau_2 \tag{8.10}$$

8.2.5 Structural Equivalence

Under structural equivalence, types are ***compatible*** if they have identical structure. Types have identical structure if their canonical definitions are identical. Structural Equivalence is commutative. It is denoted by the $\widehat{=}$ symbol, pronounced *"is structurally equivalent to"*.

$$\Gamma_{struct\text{-}equiv} \vdash \tau_1 \widehat{=} \tau_2 \Rightarrow \tau_1 \overset{:=}{\Rightarrow} \tau_2 \wedge \tau_2 \overset{:=}{\Rightarrow} \tau_1 \tag{8.11}$$

8.3 Compatibility by Type Classification

All types follow strict name equivalence by default, except for type `OCTETSEQ`.

$\boxed{\text{def}}\ \forall\, \tau_1, \tau_2 \in \{\tau_{any} \setminus \tau_{octetseq} \mid \boldsymbol{id}\,\tau_1 = \boldsymbol{id}\,\tau_2\} : \tau_1 \overset{:=}{\Rightarrow} \tau_2 \wedge \tau_2 \overset{:=}{\Rightarrow} \tau_1$

The type regime is then further relaxed depending on the classification of the type.

8.3.1 Compatibility of Alias Types

An alias type is always identical to the type of which it is an alias.

def $\forall\, \tau_1, \tau_2 \in \{\tau_{any} \mid \tau_1\ \boldsymbol{is\text{-}alias\text{-}of}\ \tau_2\} : \tau_1 \equiv \tau_2$

8.3.2 Compatibility of Derived Types

A derived type is by definition incompatible with the type from which it is derived.

def $\forall\, \tau_1, \tau_2 \in \{\tau_{any} \mid \tau_1\ \boldsymbol{is\text{-}derivative\text{-}of}\ \tau_2\} : \neg\, \tau_1 \overset{:=}{\rightarrow} \tau_2 \wedge \neg\, \tau_2 \overset{:=}{\rightarrow} \tau_1$

8.3.3 Compatibility of Anonymous Types

Anonymous types with the same canonical type declaration are identical.

def $\forall\, \tau_1, \tau_2 \in \{\tau_{anon} \mid \tau_1\ \boldsymbol{is\text{-}canonically\text{-}equal\text{-}to}\ \tau_2\} : \tau_1 \equiv \tau_2$

8.3.4 Compatibility of Subrange Types

Subrange types follow super-type equivalence.

def $\forall\, \tau_1, \tau_2 \in \{\tau_{ordinal} \mid \tau_1\ \boldsymbol{is\text{-}subrange\text{-}of}\ \tau_2\} : \tau_1 \overset{:=}{\rightarrow} \tau_2 \wedge \neg\, \tau_2 \overset{:=}{\rightarrow} \tau_1$
where $\tau_{ordinal} = \tau_{char} \cap \tau_{whole} \cap \tau_{enum}$.

8.3.5 Compatibility of Enumeration Types

Enumeration types follow super-type equivalence.

def $\forall\, \tau_1, \tau_2 \in \{\tau_{enum} \mid \tau_2\ \boldsymbol{is\text{-}extension\text{-}of}\ \tau_1\} : \tau_1 \overset{:=}{\rightarrow} \tau_2 \wedge \neg\, \tau_2 \overset{:=}{\rightarrow} T_1$

8.3.6 Compatibility of Set Types

Set types follow value-type equivalence.

def $\forall\, \tau_1, \tau_2 \in \{\tau_{set} \mid \boldsymbol{value\text{-}type}\ \tau_1 = \boldsymbol{value\text{-}type}\ \tau_2\} : \tau_1 \xrightarrow{copy} \tau_2 \wedge \tau_2 \xrightarrow{copy} \tau_1$

def $\forall\, \tau_1, \tau_2 \in \{\tau_{set} \mid \boldsymbol{value\text{-}type}\ \tau_1 \subset \boldsymbol{value\text{-}type}\ \tau_2\} : \tau_1 \xrightarrow{copy} \tau_2 \wedge \neg\, \tau_2 \xrightarrow{copy} \tau_1$

8.3.7 Compatibility of Array Types

Array types follow value-type equivalence.

def $\forall\, \tau_1, \tau_2 \in \{\tau_{array} \mid \boldsymbol{value\text{-}type}\ \tau_1 = \boldsymbol{value\text{-}type}\ \tau_2\} : T_1 \xrightarrow{copy} T_2 \wedge T_2 \xrightarrow{copy} T_1$

def $\forall\, \tau_1, \tau_2 \in \{\tau_{array} \mid \boldsymbol{value\text{-}type}\ \tau_1 \subset \boldsymbol{value\text{-}type}\ \tau_2\} : \tau_1 \xrightarrow{copy} \tau_2 \wedge \neg\, \tau_2 \xrightarrow{copy} \tau_1$

8.3.8 Compatibility of Record Types

Non-Extensible ***record types*** only follow the default strict name equivalence whereas ***extensible record types*** further follow limited super-type equivalence. Their compatibility is limited to by-reference passing compatibility and below.

def $\forall\, \tau_1, \tau_2 \in \{\tau_{extrec} \mid \tau_2\ \boldsymbol{is\text{-}extension\text{-}of}\ \tau_1\} : \tau_1 \xrightarrow{ref} \tau_2 \wedge \neg\, \tau_2 \xrightarrow{ref} \tau_1$

8.3.9 Compatibility of Pointer Types

Opaque pointer types only follow the default strict name equivalence while transparent ***pointer types*** further follow target-type equivalence.

def $\forall\, \tau_1, \tau_2 \in \{\tau_{transptr} \mid \boldsymbol{target\text{-}type}\ \tau_1 = \boldsymbol{target\text{-}type}\ \tau_2\} : \tau_1 \overset{:=}{\rightarrow} \tau_2 \wedge \tau_2 \overset{:=}{\rightarrow} \tau_1$
where $\tau_{transptr} = \tau_{pointer} \setminus \tau_{opaque}$.

Extensible record pointer types follow unlimited super-type equivalence.

def $\forall\, \tau_1, \tau_2 \in \{\tau_{extrec} \mid \tau_2\ \boldsymbol{is\text{-}extension\text{-}of}\ \tau_1\} : \tau_1 \overset{:=}{\rightarrow} \tau_2 \wedge \neg\, \tau_2 \overset{:=}{\rightarrow} \tau_1$

8.3.10 Compatibility of Procedure Types

Procedure types follow structural equivalence.

$\boxed{\text{def}}\ \forall\, \tau_1, \tau_2 \in \{\tau_{proc} \mid \tau_1 \widehat{=} \tau_2\} : \tau_1 \overset{:=}{\rightarrow} \tau_2 \wedge \tau_2 \overset{:=}{\rightarrow} \tau_1$

Any two procedure types are structurally equivalent if their respective signatures match.

$\boxed{\text{def}}\ \forall\, \tau_1, \tau_2 \in \{\tau_{proc} \mid \boldsymbol{signature\text{-}of}\ \tau_1 \doteqdot \boldsymbol{signature\text{-}of}\ \tau_2\} : \tau_1 \widehat{=} \tau_2$

The type of a procedure p is the anonymous procedure type τ defined by p's institution.

$\boxed{\text{def}}\ \forall\, p \in \{\boldsymbol{institution\text{-}of}\ p \overset{def}{\longmapsto} \tau \mid \tau \in \tau_{anon} \wedge \tau \in \tau_{proc}\} :$
$\boldsymbol{signature\text{-}of}\ p \equiv \boldsymbol{signature\text{-}of}\ \tau \wedge \boldsymbol{type\text{-}of}\ p = \tau$

8.3.11 Compatibility of Numeric Types

Numeric types follow strict name equivalence (default).

$\boxed{\text{def}}\ \forall\, \tau_1, \tau_2 \in \{\tau_{numeric} \mid \boldsymbol{id}\,\tau_1 = \boldsymbol{id}\,\tau_2\} : \tau_1 \overset{:=}{\rightarrow} \tau_2 \wedge \tau_2 \overset{:=}{\rightarrow} \tau_1$

Numeric subrange types further follow super-type equivalence.

$\boxed{\text{def}}\ \forall\, \sigma, \tau \in \{\tau_{scalar} \mid \sigma <: \tau\} : \sigma \overset{:=}{\rightarrow} \tau \wedge \neg\, \tau \overset{:=}{\rightarrow} \sigma$

8.3.12 Compatibility of Machine Types

machine types follow strict name equivalence only.

$\boxed{\text{def}}\ \forall\, \tau_1, \tau_2 \in \{\tau_{machine} \mid \boldsymbol{id}\,\tau_1 = \boldsymbol{id}\,\tau_2\} : \tau_1 \overset{:=}{\rightarrow} \tau_2 \wedge \tau_2 \overset{:=}{\rightarrow} \tau_1$

Parameters of formal type `OCTETSEQ` are ***copy*** and ***passing compatible*** only.

$\boxed{\text{def}}\ \forall\, \varepsilon_1, \varepsilon_2 \in \{\boldsymbol{type\text{-}of}\ \varepsilon_1, \varepsilon_2 = \tau_{octetseq}\} :$
$\neg\, \varepsilon_1 \overset{:=}{\rightarrow} \varepsilon_2 \wedge \varepsilon_1 \xrightarrow{copy} \varepsilon_2 \wedge \varepsilon_1 \xrightarrow{val} \varepsilon_2 \wedge \varepsilon_1 \xrightarrow{ref} \varepsilon_2 \wedge \neg\, \varepsilon_1 \xleftrightarrow{expr} \varepsilon_2$

8.4 Compatibility of Literals

The compatibility of a literal value cannot be determined by type because literals are not strictly associated with a specific type. Instead, compatibility is defined as follows:

A literal value υ is ***compatible*** with type τ if v does not overflow τ and if

- υ is a character code or quoted character literal and τ is a character type, or
- υ is a whole number literal and τ is a ***whole number type*** or ***machine type***, or
- υ is a real number literal and τ is a ***real number type***, or
- υ is a quoted character string literal and τ is a ***string type***, or
- υ is a non-empty structured literal and structurally equivalent to τ

An empty structured literal υ is ***compatible*** with type τ if

- υ is the empty string `""` and τ is a character type or ***string type***, or
- υ is the empty collection `{}` and τ is a set type or ***array type*** but not a ***string type***

Chapter 9

Predefined Identifiers

Predefined identifiers are language defined and built-in. They are **pervasive**, that is, they are visible in every scope without import. Pervasive identifiers do not have qualified names and are always referenced unqualified. Unlike in earlier versions of Modula-2, pervasive identifiers may not be redefined.

The following identifiers are predefined:

`NIL`, `TRUE`, `FALSE`, `BOOLEAN`, `OCTET`, `CHAR`, `UNICHAR`, `CARDINAL`, `LONGCARD`, `INTEGER`, `LONGINT`, `REAL`, `LONGREAL`, `APPEND`, `INSERT`, `REMOVE`, `CHR`, `UCHR`, `ORD`, `ODD`, `ABS`, `SGN`, `MIN`, `MAX`, `LOG2`, `POW2`, `ENTIER`, `PRED`, `SUCC`, `PTR`, `UNIVSET`, `CAPACITY`, `COUNT`, `LENGTH`, `TMIN`, `TMAX`, `TSIZE`, `TLIMIT`.

9.1 Predefined Constants

9.1.1 Constant `NIL`

Identifier `NIL` represents an invalid pointer or address value. It is per definition ***compatible*** with any ***pointer type***, including all ***opaque pointer types*** and all procedure types.

9.1.2 Constants `TRUE` and `FALSE`

Identifiers `TRUE` and `FALSE` represent the values of type `BOOLEAN`.

```
ALIAS TRUE, FALSE = BOOLEAN.*;
```

9.2 Predefined Types

9.2.1 Type `BOOLEAN`

Type `BOOLEAN` is an ***enumerated type***.

It represents logical truth values used in boolean expressions.

```
TYPE BOOLEAN = ( FALSE, TRUE );
```

The order is significant. `ORD(FALSE)` is always zero, `ORD(TRUE)` is always one.

9.2.2 Type `OCTET`

Type `OCTET` is a ***cardinal type***.

It represents the smallest addressable unit which is per definition 8 bits.

Its value range is always `[0..255]`.

Whole number literals are ***compatible*** with type `OCTET`.

9.2.3 Type `CHAR`

Type `CHAR` is a ***countable type***.

It represents the 7-bit character codes of ISO 646 (aka ASCII).

Its value range is always `[0u0..0u7F]`.

Quoted literals of exactly one ISO 646 code point are ***compatible*** with type `CHAR`.

9.2.4 Type `UNICHAR`

Type `UNICHAR` is a ***countable type***.

It represents the 32-bit character codes of ISO 10646 in UCS-4 encoding.

Its value range is always `[0u0..0u10FFFF]`.

Quoted literals of exactly one ISO 10646 code point are ***compatible*** with type `UNICHAR`.

9.2.5 Type `CARDINAL`

Type `CARDINAL` is a ***cardinal type***. It represents unsigned ***whole numbers***.

Its value range is `[0..TMAX(CARDINAL)]` where `TMAX(CARDINAL) = POW2(8*TSIZE(CARDINAL)) - 1`.

The value of `TSIZE(CARDINAL)` is implementation defined, where `TSIZE(CARDINAL) <= TSIZE(LONGCARD)`.

Whole number literals are ***compatible*** with type `CARDINAL`.

9.2.6 Type `LONGCARD`

Type `LONGCARD` is a ***cardinal type***. It represents unsigned ***whole numbers*** with extended range.

Its value range is `[0..TMAX(LONGCARD)]` where `TMAX(LONGCARD) = POW2(8*TSIZE(LONGCARD)) - 1`.

The value of `TSIZE(LONGCARD)` is implementation defined, where `TSIZE(LONGCARD) >= TSIZE(ADDRESS)`.

Whole number literals are ***compatible*** with type `LONGCARD`.

9.2.7 Type `INTEGER`

Type `INTEGER` is an ***integer type***. It represents signed ***whole numbers***.

Its value range is `[TMIN(INTEGER)..TMAX(INTEGER)]`.

The internal representation of type `INTEGER` is always in two's complement. Thus

- `TMIN(INTEGER) = (-1) * POW2(8*TSIZE(INTEGER) - 1)`
- `TMAX(INTEGER) = POW2(8*TSIZE(INTEGER) - 1) - 1`

The value of `TSIZE(INTEGER)` is always equal to `TSIZE(CARDINAL)`.

Whole number literals are ***compatible*** with type `INTEGER`.

9.2.8 Type `LONGINT`

Type `LONGINT` is an ***integer type***. It represents signed ***whole numbers*** with extended range.

Its value range is `[TMIN(LONGINT)..TMAX(LONGINT)]`.

The internal representation of type `LONGINT` is always in two's complement. Thus

- `TMIN(LONGINT) = (-1) * POW2(8*TSIZE(LONGINT) - 1)`
- `TMAX(LONGINT) = POW2(8*TSIZE(LONGINT) - 1) - 1`

The value of `TSIZE(LONGINT)` is always equal to `TSIZE(LONGCARD)`.

Whole number literals are ***compatible*** with type `LONGINT`.

9.2.9 Type REAL

Type `REAL` is a ***real number type***. It represents real numbers with standard range and precision.

Its value range is `[TMIN(REAL)..TMAX(REAL)]`.

The internal representation of type `REAL` is implementation defined, where

- `TMIN(REAL) <= TMIN(LONGINT)`
- `TMAX(REAL) >= TMAX(LONGCARD)`

IEEE 754 single or double precision format is recommended for type `REAL`.

Real number literals are ***compatible*** with type `REAL`.

9.2.10 Type LONGREAL

Type `LONGREAL` is a ***real number type***. It represents real numbers with extended range and/or precision.

Its value range is `[TMIN(LONGREAL)..TMAX(LONGREAL)]`.

The internal representation of type `LONGREAL` is implementation defined, where

- `TMIN(LONGREAL) <= TMIN(REAL)`
- `TMAX(LONGREAL) >= TMAX(REAL)`
- the precision of type `REAL` must not exceed that of type `LONGREAL`

IEEE 754 double or quadruple precision format is recommended for type `LONGREAL`.

Real number literals are ***compatible*** with type `LONGREAL`.

9.3 Predefined Procedures

9.3.1 Procedure APPEND

Procedure `APPEND` appends the values of a given list to an array.

```
PROCEDURE APPEND ( c : ArrayType; values : ARGLIST OF ComponentType );
```

Values to be appended must be of the component type of the array.

9.3.2 Procedure INSERT

Procedure `INSERT` inserts the values of a given list into a set.

```
PROCEDURE INSERT ( VAR set : SetType; values : ARGLIST OF ComponentType );
```

Values passed in the list must be of the value type of the set.

9.3.3 Procedure REMOVE

Procedure `REMOVE` has two signatures, depending on the type of its first argument.

If its first argument is a set, it removes the values of a given list from the set.

```
PROCEDURE REMOVE ( VAR set : SetType; values : ARGLIST OF ComponentType );
```

Values passed in the list must be of the value type of the set.

If its first argument is an array, it removes the values addressed by a given index range from the array.

```
PROCEDURE REMOVE ( VAR array : ArrayType; startIndex, endIndex : LONGCARD );
```

Any relative (negative) indices passed are translated into absolute (positive) indices by the language processor.

9.4 Predefined Functions

9.4.1 Function CHR

Function `CHR` returns the character whose character code is its argument.

```
PROCEDURE CHR ( n : T ) : CHAR;
```

where $0 \leq$ `n` $\leq 127 \wedge$ `T` $\in \tau_{cardinal}$

9.4.2 Function UCHR

Function `UCHR` returns the unicode character whose character code is its argument.

```
PROCEDURE UCHR ( n : T ) : UNICHAR;
```

where $0 \leq$ `n` $\leq$ `0x10FFFF` $\wedge$ `T` $\in \tau_{cardinal}$

9.4.3 Function ORD

Function `ORD` returns the ordinal value of its countable argument.

```
PROCEDURE ORD ( value : T ) : LONGCARD;
```

where `T` $\in \tau_{countable}$

9.4.4 Function ODD

Function `ODD` returns `TRUE` if its argument is odd, otherwise `FALSE`.

```
PROCEDURE ODD ( value : T ) : BOOLEAN;
```

where `T` $\in \tau_{whole} \cup \tau_{machine}$

9.4.5 Function ABS

Function `ABS` returns the absolute value of its scalar argument.

```
PROCEDURE ABS ( value : T1 ) : T2;
```

where `T1` $\in \tau_{signed} \cap \tau_{scalar} \wedge$ `T2` $=$ `T1`

9.4.6 Function SGN

Function `SGN` returns the sign or ***signum value*** of its argument.

```
PROCEDURE SGN ( value : T1 ) : T2;
```

where `T1` $\in \tau_{signed} \cap \tau_{scalar} \wedge$ `T2` $=$ `T1`

9.4.7 Function MIN

Function `MIN` returns the smallest value in its variadic argument list.

```
PROCEDURE MIN ( values : ARGLIST OF T1 ) : T2;
```

where `T1` $\in \tau_{countable} \cup \tau_{scalar} \wedge$ `T2` $=$ `T1`

9.4.8 Function MAX

Function `MAX` returns the largest value in its variadic argument list.

```
PROCEDURE MAX ( values : ARGLIST OF T1 ) : T2;
```

where `T1` $\in \tau_{countable} \cup \tau_{scalar} \wedge$ `T2` $=$ `T1`

9.4.9 Function LOG2

Function `LOG2` returns the truncated binary logarithm of its argument.

```
PROCEDURE LOG2 ( n : T1 ) : T2;
```

where `n` $\neq 0 \wedge$ `T1` $\in \tau_{cardinal} \wedge$ `T2` $=$ `T1`

9.4.10 Function POW2

Function `POW2` returns the value of its argument raised to the power of 2.

```
PROCEDURE POW2 ( n : T1 ) : T2;
```

where `T1` $\in \tau_{cardinal} \wedge$ `T2` $=$ `T1`

9.4.11 Function ENTIER

Function `ENTIER` returns the largest integer or ***entier*** of its argument.

```
PROCEDURE ENTIER ( r : T1 ) : T2;
```

where `T1` $\in \tau_{real} \wedge$ `T2` $=$ `T1`

9.4.12 Function PRED

Function `PRED` returns the predecessor of its argument.

```
PROCEDURE PRED ( value : T1 ) : T2;
```

where `T1` $\in \tau_{countable} \wedge$ `T2` $=$ `T1`

9.4.13 Function SUCC

Function `SUCC` returns the successor of its argument.

```
PROCEDURE SUCC ( value : T1 ) : T2;
```

where `T1` $\in \tau_{countable} \wedge$ `T2` $=$ `T1`

9.4.14 Function PTR

Function `PTR` returns a pointer to its argument.

```
PROCEDURE PTR ( value : T1 ) : POINTER TO T2;
```

where `T1` $\in \tau_{any} \wedge$ `T2` $=$ `T1`

9.4.15 Function UNIVSET

Function `UNIVSET` returns the universal set for a given set.

```
PROCEDURE UNIVSET ( set : T1 ) : T2;
```

where `T1` $\in \tau_{set} \wedge$ `T2` $=$ `T1` and literals may not be passed to `set`.

9.4.16 Function CAPACITY

Function `CAPACITY` returns the ***capacity*** of its argument.

```
PROCEDURE CAPACITY ( entity : T ) : LONGCARD;
```

where `T` $\in \tau_{collection}$

9.4.17 Function COUNT

Function `COUNT` returns the value-count or ***cardinality*** of its argument.

```
PROCEDURE COUNT ( entity : T ) : LONGCARD;
```

where `T` $\in \tau_{collection} \setminus \tau_{string} \setminus \tau_{octetseq}$

The argument must be either (1) a value of a ***collection type*** that is **not** a ***string type***, or (2) the identifier of an open array parameter whose value type is not a character type, or (3) the identifier of a variadic argument list.

9.4.18 Function LENGTH

Function `LENGTH` returns the ***length*** of the character string or ***octet sequence*** denoted by its argument.

```
PROCEDURE LENGTH ( entity : T ) : LONGCARD;
```

where `T` $\in \tau_{string} \cup \tau_{octetseq}$

The argument must be either (1) a value of a ***string type***, or (2) the identifier of an open array parameter whose value type is a character type, or (3) the identifier of a casting formal parameter of type `OCTETSEQ`.

9.5 Built-in Macros

9.5.1 Macro TMIN

An invocation of macro `TMIN` is replaced by the smallest value of the type denoted by its argument.

```
(*MACRO*) PROCEDURE TMIN ( typeIdent ) : T;
```

where `typeIdent` $= id$ `T` $\wedge$ `T` $\in \tau_{countable} \cup \tau_{scalar}$

9.5.2 Macro TMAX

An invocation of macro `TMAX` is replaced by the largest value of the type denoted by its argument.

```
(*MACRO*) PROCEDURE TMAX ( typeIdent ) : T;
```

where `typeIdent` $= id$ `T` $\wedge$ `T` $\in \tau_{countable} \cup \tau_{scalar}$

9.5.3 Macro TSIZE

An invocation of macro `TSIZE` is replaced by the ***allocation size*** of the type denoted by its argument.

If the (primary) argument does **not** denote an indeterminate type, no second argument may be passed.

```
(*MACRO*) PROCEDURE TSIZE ( typeIdent ) : LONGCARD;
```

If the primary argument denotes a ***pointer type*** with indeterminate target, a second argument must be passed,

```
(*MACRO*) PROCEDURE TSIZE ( typeIdent; capacity : LONGCARD ) : LONGCARD;
```

and the result is the ***allocation size*** for the target with the ***capacity*** given by the second argument.

In any event the (primary) argument must be a type identifier.

9.5.4 Macro TLIMIT

An invocation of macro `TLIMIT` is replaced by the ***capacity limit*** of the type denoted by its argument.

```
(*MACRO*) PROCEDURE TLIMIT ( typeIdent ) : LONGCARD;
```

The argument must be an identifier of a ***collection type***.

9.6 Primitives

Primitives are low-level procedures or macros that implement statements or designators within statements and expressions. When a statement or designator implemented by a primitive is found, it is replaced with an invocation of the associated primitive. Primitives are not callable directly in user code. However, primitives may be the target of syntax bindings in ADT library modules.

9.6.1 Primitive @EQUALS

Primitive `@EQUALS` implements expressions of the form `a = b` and `a # b` for compound types.

It compares its arguments and returns `TRUE` if they are equal, otherwise `FALSE`.

```
PROCEDURE @EQUALS ( CONST c1, c2 : CompoundType ) : BOOLEAN;
```

For values `a` and `b` of the same compound type

- any expression of the form `a = b` is replaced by a function call `@EQUALS(a, b)`
- any expression of the form `a # b` is replaced by an expression `NOT @EQUALS(a, b)`

9.6.2 Primitive @PRECEDES

Primitive `@PRECEDES` implements expressions of the form `s1 < s2` and `s1 >= s2` for character arrays.

It compares its arguments and returns `TRUE` if the first argument precedes the second, otherwise `FALSE`.

```
PROCEDURE @PRECEDES ( CONST s1, s2 : CompoundType ) : BOOLEAN;
```

For values `s1` and `s2` of a ***string type***

- any expression of the form `s1 < s2` is replaced by a function call `@PRECEDES(s1, s2)`
- any expression of the form `s1 >= s2` is replaced by an expression `NOT @PRECEDES(s1, s2)`

The default collation order is determined by

- ordinality of type `CHAR` for `ARRAY OF CHAR` types
- ordinality of type `UNICHAR` for `ARRAY OF UNICHAR` types

It may be changed by binding library supplied comparison functions to primitives `@PRECEDES` and `@SUCCEEDS`.

9.6.3 Primitive @SUCCEEDS

Primitive `@SUCCEEDS` implements expressions of the form `s1 > s2` and `s1 <= s2` for character arrays.

It compares its arguments and returns `TRUE` if the first argument succeeds the second, otherwise `FALSE`.

```
PROCEDURE @SUCCEEDS ( CONST s1, s2 : CompoundType ) : BOOLEAN;
```

For values `s1` and `s2` of a ***string type***

- any expression of the form `s1 > s2` is replaced by a function call `@SUCCEEDS(s1, s2)`
- any expression of the form `s1 <= s2` is replaced by an expression `NOT @SUCCEEDS(s1, s2)`

The default collation order is the same as for primitive `@PRECEDES`.

9.6.4 Primitive @VALUE

Primitive `@VALUE` implements R-value subscript designators.

It returns the value stored at a given index in an indexed collection.

```
PROCEDURE @VALUE ( CONST c : IndexedCollection; atIndex : LONGCARD ) : ComponentType;
```

Any R-value subscript designator `a[i]` is replaced by a function call `@VALUE(a, i)`.

9.6.5 Primitive @STORE

Primitive `@STORE` implements L-value subscript designators.

It stores a given value in an indexed collection at a given index.

```
PROCEDURE @STORE ( VAR c : IndexedCollection; atIndex : LONGCARD; value : ComponentType );
```

A statement of the form `a[i] := v` is replaced by a procedure call `@STORE(a, i, v)`.

9.6.6 Primitive @ATINSERT

Primitive `@ATINSERT` implements L-value subscript insertion designators.

It inserts one or more values into an indexed collection starting at a given index.

```
PROCEDURE @ATINSERT
  ( VAR c : IndexedCollection; atIndex : LONGCARD; values : ARGLIST OF ComponentType );
```

A statement of the form `a[i..] := v` is replaced by a procedure call `@ATINSERT(a, i, v)`.

9.6.7 Primitive @ATREMOVE

Primitive `@ATREMOVE` implements syntax for indexed value removal.

It removes a range of values from an indexed collection within a given index range.

```
PROCEDURE @ATREMOVE ( VAR c : IndexedCollection; startIndex, endIndex : LONGCARD );
```

For values `a` of an ***array type***, `s` of a ***string type*** and `c` of any indexed collection type

- a statement of the form `a := {}` is replaced by a procedure call `@ATREMOVE(a, 0, COUNT(a)-1)`
- a statement of the form `s := ""` is replaced by a procedure call `@ATREMOVE(s, 0, LENGTH(s)-1)`
- a statement of the form `REMOVE(c, i, j)` is replaced by a procedure call `@ATREMOVE(c, i, j)`

9.6.8 Primitive @STDIN

Primitive `@STDIN` inserts the default value for an omitted input channel in `READ` statements.

```
PROCEDURE @STDIN : LibraryDefinedChannelType;
```

9.6.9 Primitive @STDOUT

Primitive `@STDOUT` inserts the default value for an omitted output channel in `WRITE` statements.

```
PROCEDURE @STDOUT : LibraryDefinedChannelType;
```

Chapter 10

Syntax Binding

Syntax Binding is the process of mapping library defined procedures to built-in syntax. A procedure is bound to a given syntax form by including a binding specifier within the procedure's header. The specifier determines to which syntax form the procedure is bound. It must be included in both the procedure's definition and declaration.

```
procedureHeader :=
  PROCEDURE ( '[' bindingSpecifier ']')? procedureSignature ;
bindingSpecifier :=
  NEW ( '+' | '#' )? | RETAIN | RELEASE |
  READ ( '*' )? | WRITE ( '#' )? | ( '@' )? StdIdent ;
```

A syntax form to which library defined procedures may be bound is said to be bindable. Procedures bound to it are called its bindings. While bindable syntax forms are type agnostic, their bindings are type specific, that is to say, each binding supports arguments of a specific type. For each type to be supported by a bindable syntax form, a binding specific to arguments of the type is required.

A bindable syntax form `S` is said to have a binding for type `T` when a library procedure `p` that supports arguments of `T` has been bound to `S`. Procedure `p` is said to be bound to `S` in respect of `T`.

10.1 Syntax Transformation

Bindable syntax forms act as ***Wirthian macros***. A syntax form `S` with binding `p` for type `T`, an occurrence of `S` with a primary argument or argument list `a` of type `T` is resolved into a procedure call to `p`, passing `a`.

`S a ...` → `T.p(a...)`

10.2 Memory Management Bindings

10.2.1 Binding to NEW

Statement `NEW` has three bindable syntax forms.

10.2.2 NEW without Arguments

An ADT module `Foo` with a binding

```
PROCEDURE [NEW] New ( VAR p : Foo );
```

binds ADT library procedure `Foo.New()` to the `NEW` statement in respect of type `Foo`.

Any use of the `NEW` statement with a designator of type `Foo` and without initialiser nor capacity specifier is then replaced by a call to procedure `Foo.New()`.

Given the following import and declaration within a client module

```
IMPORT foo; VAR foo : Foo;
```

a statement of the form `NEW foo` is resolved into a library call `Foo.New(foo)`.

10.2.3 NEW with Intialiser

An ADT module `Foo` with a binding

```
PROCEDURE [NEW+] NewWithArgs ( VAR p : Foo; initVal : ARGLIST OF Value );
```

binds ADT library procedure `Foo.NewWithArgs()` to the `NEW` statement in respect of type `Foo`.

Any use of the `NEW` statement with a designator of type `Foo` and an initialiser is then resolved into a call to procedure `Foo.NewWithArgs()`.

Given the following import and declaration within a client module

```
IMPORT Foo; VAR foo : Foo;
```

a statement of the form `NEW foo := { a, b, c }` is resolved into a library call `Foo.NewWithArgs(foo, a, b, c)`.

10.2.4 NEW with Capacity

An ADT module `Foo` with a binding

```
PROCEDURE [NEW#] NewWithCapacity ( VAR p : Foo; capacity : LONGCARD );
```

binds ADT library procedure `NewWithCapacity()` to the `NEW` statement in respect of type `Foo`.

Any use of the `NEW` statement with a designator of type `Foo` and a capacity specifier is then resolved into a call to procedure `Foo.NewWithCapacity()`.

Given the following import and declaration within a client module

```
IMPORT Foo; VAR foo : Foo;
```

a statement of the form `NEW foo CAPACITY n` is resolved into a library call `Foo.NewWithCapacity(foo, n)`.

10.2.5 Binding to RETAIN

An ADT module `Foo` with a binding

```
PROCEDURE [RETAIN] Retain ( p : Foo );
```

binds ADT library procedure `Retain()` to the `RETAIN` statement in respect of type `Foo`.

Any use of `RETAIN` with a designator of type `Foo` is then resolved into a call to procedure `Foo.Retain()`.

Given the following import and declaration within a client module

```
IMPORT Foo; VAR foo : Foo;
```

a statement of the form `RETAIN foo` is resolved into a library call `Foo.Retain(foo)`.

10.2.6 Binding to RELEASE

An ADT module `Foo` with a binding

```
PROCEDURE [RELEASE] Release ( VAR p : Foo );
```

binds ADT library procedure `Release()` to the `RELEASE` statement in respect of type `Foo`.

Any use of `RELEASE` with a designator of type `Foo` is then resolved into a call to procedure `Foo.Release()`.

Given the following import and declaration within a client module

```
IMPORT Foo; VAR foo : Foo;
```

a statement of the form `RELEASE foo` is resolved into a library call `Foo.Release(foo)`.

10.3 IO Bindings

10.3.1 Bindings to READ

Statement `READ` has two associated bindings.

- `[READ]` bindings are used to generate calls for **plain** designators in a `READ` statement
- `[READ*]` bindings are used to generate calls for `NEW`-**prefixed** designators in a `READ` statement

An IO module `FooIO` with a binding

```
PROCEDURE [READ] Read ( chan : ChanIO.Channel; VAR value : Foo );
```

binds IO library procedure `Read()` to the `READ` statement in respect of type `Foo`.

Any plain designator of type `Foo` within a `READ` statement will then generate a call to procedure `FooIO.Read()`.

An IO module `FooIO` with a binding

```
PROCEDURE [READ*] ReadNew ( chan : ChanIO.Channel; VAR ptr : Foo ); (* Foo is a pointer type *)
```

binds IO library procedure `ReadNew()` to the `READ` statement in respect of pointer type `Foo`.

Any `NEW`-prefixed designator of pointer type `Foo` within a `READ` statement will then generate a call to procedure `FooIO.ReadNew()`.

10.3.1.1 READ with Explicit IO Channel

Given the following imports and declaration within a client module

```
IMPORT Foo, FooIO, ChanIO; VAR foo : Foo; file : ChanIO.Channel;
```

- a statement of the form `READ @file : foo` is resolved into a library call `FooIO.Read(file, foo)`
- a statement of the form `READ @file : NEW foo` is resolved into a library call `FooIO.ReadNew(file, foo)`

10.3.1.2 READ with Default IO Channel

If the IO channel argument is omitted, default input channel primitive `@STDIN` is automatically inserted. In order to resolve `@STDIN` an appropriate IO library must be imported to bind an input channel to `@STDIN` and make it visible within the current module scope. Import of module `StdIO` binds `StdIO.stdIn` to `@STDIN`.

Given the following imports and declaration within a client module

```
IMPORT Foo, FooIO, StdIO; VAR foo : Foo;
```

- a statement of the form `READ foo` is resolved into a library call `FooIO.Read(StdIO.stdIn, foo)`
- a statement of the form `READ NEW foo` is resolved into a library call `FooIO.ReadNew(StdIO.stdIn, foo)`

10.3.1.3 READ with Multiple Arguments

Given modules `Foo`, `Bar`, `Baz` and `Bam`, their respective IO modules `FooIO`, `BarIO`, `BazIO` and `BamIO` with bindings to `READ` for their respective types and the following imports and declaration within a client module

```
IMPORT Foo, FooIO, Bar, BarIO, Baz, BazIO, Bam, BamIO, StdIO;
VAR foo : Foo; bar : Bar; baz : Baz; bam : Bam;
```

a statement of the form

```
READ foo, NEW bar, baz, NEW bam;
```

is resolved into a statement sequence

```
FooIO.Read(StdIO.stdIn, foo); BarIO.ReadNew(StdIO.stdIn, bar);
BazIO.Read(StdIO.stdIn, baz); BamIO.ReadNew(StdIO.stdIn, bam);
```

10.3.2 Bindings to WRITE

Statement `WRITE` has two associated bindings.

- `[WRITE]` bindings are used to generate calls for **unformatted** output values in a `WRITE` statement
- `[WRITE#]` bindings are used to generate calls for **formatted** output values in a `WRITE` statement

An IO module `FooIO` with a binding

```
PROCEDURE [WRITE] Write ( chan : ChanIO.Channel; value : Foo );
```

binds IO library procedure `Write()` to the `WRITE` statement in respect of type `Foo`.

Any unformatted output value of type `Foo` within a `WRITE` statement will then generate a call to procedure `FooIO.Write()`.

An IO module `FooIO` with a binding

```
PROCEDURE [WRITE#] WriteF ( chan : ChanIO.Channel; CONST fmtStr : ARRAY OF CHAR; value : Foo );
```

binds IO library procedure `WriteF()` to the `WRITE` statement in respect of type `Foo`.

Any formatted output value of type `Foo` within a `WRITE` statement will then generate a call to procedure `FooIO.WriteF()`.

10.3.2.1 WRITE with Explicit IO Channel

Given the following imports and declaration within a client module

```
IMPORT Foo, FooIO, ChanIO; VAR foo : Foo; file : ChanIO.Channel;
```

- a statement of the form `WRITE @file : foo` is resolved into a library call `FooIO.Write(file, foo)`
- a statement of the form `WRITE @file : #(fmt, foo)` is resolved into a call `FooIO.WriteF(file, fmt, foo)`

10.3.2.2 WRITE with Default IO Channel

If the IO channel argument is omitted, default output channel primitive `@STDOUT` is automatically inserted. In order to resolve `@STDOUT` an appropriate IO library must be imported to bind an output channel to `@STDOUT` and make it visible within the current module scope. Import of module `StdIO` binds `StdIO.stdOut` to `@STDOUT`.

Given the following imports and declaration within a client module

```
IMPORT Foo, FooIO, StdIO; VAR foo : Foo;
```

- a statement of the form `WRITE foo` is resolved into a library call `FooIO.Write(StdIO.stdOut, foo)`
- a statement of the form `WRITE #(fmt, foo)` is resolved into a call `FooIO.WriteF(StdIO.stdOut, fmt, foo)`

10.3.2.3 WRITE with Multiple Arguments

Given modules `Foo`, `Bar`, `Baz` and `Bam`, their respective IO modules `FooIO`, `BarIO`, `BazIO` and `BamIO` with bindings to `WRITE` for their respective types and the following imports and declaration within a client module

```
IMPORT Foo, FooIO, Bar, BarIO, Baz, BazIO, Bam, BamIO, StdIO;
VAR foo : Foo; bar : Bar; baz : Baz; bam : Bam;
```

a statement of the form

```
WRITE foo, #(fmt1, bar), baz, #(fmt2, bam);
```

is resolved into a statement sequence

```
FooIO.Write(StdIO.stdOut, foo); BarIO.WriteF(StdIO.stdOut, fmt1, bar);
BazIO.Write(StdIO.stdOut, baz); BarIO.WriteF(StdIO.stdOut, fmt2, bam);
```

10.4 Bindings to Pervasives

10.4.1 Binding to Procedure APPEND

An ADT module `Foo` with a binding

```
PROCEDURE [APPEND] Append ( VAR a : Foo; values : ARGLIST OF Value );
```

binds ADT library procedure `Foo.Append()` to pervasive procedure `APPEND()` in respect of type `Foo`.

Any call to `APPEND()` with a first argument of type `Foo` is then resolved into a call to procedure `Foo.Append()`.

Given the following import and declaration within a client module

```
IMPORT Foo; VAR foo : Foo;
```

an invocation of the form `APPEND(foo, v1, v2, v3)` is resolved into a library call `Foo.Append(foo, v1, v2, v3)`.

10.4.2 Binding to Function COUNT

An ADT module `Foo` with a binding

```
PROCEDURE [COUNT] count ( VAR a : Foo ) : LONGCARD;
```

binds ADT library function `Foo.count()` to pervasive function `COUNT()` in respect of type `Foo`.

Any invocation of `COUNT()` with an argument of type `Foo` is then resolved into a call to function `Foo.count()`.

Given the following import and declaration within a client module

```
IMPORT Foo; VAR foo : Foo;
```

an invocation of the form `COUNT(foo)` is resolved into a library call `Foo.count(foo)`.

Within any given ADT module, bindings to `COUNT` and `LENGTH` are mutually exclusive.

10.4.3 Binding to Function LENGTH

An ADT module `Foo` with a binding

```
PROCEDURE [LENGTH] length ( VAR a : Foo ) : LONGCARD;
```

binds ADT library function `Foo.length()` to pervasive function `LENGTH()` in respect of type `Foo`.

Any invocation of `LENGTH()` with an argument of type `Foo` is then resolved into a call to function `Foo.length()`.

Given the following import and declaration within a client module

```
IMPORT Foo; VAR foo : Foo;
```

an invocation of the form `LENGTH(foo)` is resolved into a library call `Foo.length(foo)`.

Within any given ADT module, bindings to `COUNT` and `LENGTH` are mutually exclusive.

10.4.4 Binding to Macro TLIMIT

An ADT module `Foo` with a binding

```
CONST [TLIMIT] Capacity = 1000;
```

binds ADT library constant `Foo.Capacity` to pervasive macro `TLIMIT()` in respect of type `Foo`.

Given the following import within a client module

```
IMPORT Foo;
```

an expression of the form `TLIMIT(Foo)` is replaced by library constant `Foo.Capacity`.

10.5 Bindings to Primitives

10.5.1 Binding to Primitive @EQUALS

An ADT module `Foo` with a binding

```
PROCEDURE [@EQUALS] equals ( CONST a, b : Foo ) : BOOLEAN;
```

binds function `Foo.equals()` to primitive `@EQUALS` in respect of type `Foo`.

Any (internal) use of primitive `@EQUALS` with arguments of type `Foo` is then resolved into a library call to function `Foo.equals()` during translation of syntax forms that are synthesised using primitive `@EQUALS`.

10.5.2 Binding to Primitive @PRECEDES

An ADT module `Foo` with a binding

```
PROCEDURE [@PRECEDES] precedes ( CONST a, b : Foo ) : BOOLEAN;
```

binds function `Foo.precedes()` to primitive `@PRECEDES` in respect of type `Foo`.

Any (internal) use of primitive `@PRECEDES` with arguments of type `Foo` is then resolved into a library call to function `Foo.precedes()` during translation of syntax forms that are synthesised using primitive `@PRECEDES`.

10.5.3 Binding to Primitive @SUCCEEDS

An ADT module `Foo` with a binding

```
PROCEDURE [@SUCCEEDS] succeeds ( CONST a, b : Foo ) : BOOLEAN;
```

binds function `Foo.succeeds()` to primitive `@SUCCEEDS` in respect of type `Foo`.

Any (internal) use of primitive `@SUCCEEDS` with arguments of type `Foo` is then resolved into a library call to function `Foo.succeeds()` during translation of syntax forms that are synthesised using primitive `@SUCCEEDS`.

10.5.4 Binding to Primitive @STORE

An ADT module `Foo` with a binding

```
PROCEDURE [@STORE] StoreAtIndex ( VAR a : Foo; atIndex : LONGCARD; value : Value );
```

binds procedure `Foo.StoreAtIndex` to primitive `@STORE` in respect of type `Foo`.

Any (internal) use of primitive `@STORE` with a first argument of type `Foo` is then resolved into a library call to procedure `Foo.StoreAtIndex` during translation of syntax forms that are synthesised using primitive `@STORE`.

10.5.5 Binding to Primitive @VALUE

An ADT module `Foo` with a binding

```
PROCEDURE [@VALUE] valueAtIndex ( VAR a : Foo; atIndex : LONGCARD ) : Value;
```

binds function `Foo.valueAtIndex()` to primitive `@VALUE` in respect of type `Foo`.

Any (internal) use of primitive `@VALUE` with a first argument of type `Foo` is then resolved into a library call to function `Foo.valueAtIndex()` during translation of syntax forms that are synthesised using primitive `@VALUE`.

10.5.6 Binding to Primitive @ATINSERT

An ADT module `Foo` with a binding

```
PROCEDURE [@ATINSERT] AtInsert ( VAR a : Foo; atIndex : LONGCARD; values : ARGLIST OF Value );
```

binds procedure `Foo.AtInsert` to primitive `@ATINSERT` in respect of type `Foo`.

Any (internal) use of primitive `@ATINSERT` with a first argument of type `Foo` is then resolved into a library call to procedure `Foo.AtInsert` during translation of syntax forms that are synthesised using primitive `@ATINSERT`.

10.5.7 Binding to Primitive @ATREMOVE

An ADT module `Foo` with a binding

```
PROCEDURE [@ATREMOVE] AtRemove ( VAR a : Foo; startIndex, endIndex : LONGCARD );
```

binds procedure `Foo.AtRemove` to primitive `@ATREMOVE` in respect of type `Foo`.

Any (internal) use of primitive `@ATREMOVE` with a first argument of type `Foo` is then resolved into a library call to procedure `Foo.AtRemove` during translation of syntax forms that are synthesised using primitive `@ATREMOVE`.

10.5.8 Binding to Primitive @STDIN

An IO module `MyIO` with a binding

```
PROCEDURE [@STDIN] stdIn : ArbitraryIOChannelType;
```

binds function `MyIO.stdIn` to primitive `@STDIN`. (Internal) use of `@STDIN` is then resolved to `MyIO.stdIn`.

10.5.9 Binding to Primitive @STDOUT

An IO module `MyIO` with a binding

```
PROCEDURE [@STDOUT] stdOut : ArbitraryIOChannelType;
```

binds function `MyIO.stdOut` to primitive `@STDOUT`. (Internal) use of `@STDOUT` is then resolved to `MyIO.stdOut`.

Chapter 11

Low-Level Facilities

Low-level facilities are provided by special module `UNSAFE`. Although presented as a library it is built into the language. Its facilities are potentially unsafe, their use may bypass the type safety of the language and they must therefore be used with caution. The module must be imported before its facilities can be used.

Module `UNSAFE` provides the following identifiers:

`BitsPerByte`, `BytesPerWord`, `BytesPerLongWord`, `BitsPerAddress`, `LMTBits`, `BOMaxBits`, `BYTE`, `WORD`, `LONGWORD`, `ADDRESS`, `ADD`, `SUB`, `SETBIT`, `HALT`, `ADR`, `CAST`, `BIT`, `SHL`, `SHR`, `BWNOT`, `BWAND`, `BWOR`.

11.1 Low-Level Constants

11.1.1 `BitsPerByte`

Constant `BitsPerByte` represents the bit width of the smallest addressable machine unit of the target architecture measured in bits. Its value is therefore target architecture dependent. For the overwhelming majority of platforms in use this value will be eight.

Memory allocation in Modula-2 is strictly in units of eight bits and multiples thereof. On (legacy) targets where the smallest addressable machine unit is not eight bits, the mapping of type `OCTET` to ***machine types*** is implementation defined.

11.1.2 `BytesPerWord`

Constant `BytesPerWord` represents the size of a machine register of the target architecture smaller than the largest machine register. Its value is target architecture dependent.

11.1.3 `BytesPerLongWord`

Constant `BytesPerLongWord` represents the size of the largest machine register of the target architecture. Its value is target architecture dependent. If the target architecture does not support machine registers of different sizes, `BytesPerLongWord` is equal to `BytesPerWord`.

11.1.4 `BitsPerAddress`

Constant `BitsPerAddress` represents the bit width of type `ADDRESS`.

11.1.5 `LMTBits`

Constant `LMTBits` represents the bit width of the largest machine type.

11.1.6 `BOMaxBits`

Constant `BOMaxBits` represents the bit width of the largest operand supported by the `UNSAFE` bit operations `ADD()`, `SUB()`, `SETBIT()`, `BIT()`, `SHL()`, `SHR()`, `BWNOT()`, `BWAND()` and `BWOR()`.

11.2 Low-Level Types

11.2.1 BYTE

Type `BYTE` is a ***machine type***.

It represents the smallest addressable storage unit of the target architecture.

Its bit width is defined by constant `BitsPerByte`.

Its value range is `[0..POW2(BitsPerByte)-1]`.

Whole number literals are ***compatible*** with type `BYTE`.

11.2.2 WORD

Type `WORD` is a ***machine type***.

It represents a storage unit equivalent to the size of a machine register of the target architecture.

Its bit width is defined by constants `BitsPerByte` and `BytesPerWord`.

Its value range is `[0..POW2(BitsPerByte*BytesPerWord)-1]`.

Whole number literals are ***compatible*** with type `WORD`.

11.2.3 LONGWORD

Type `LONGWORD` is a ***machine type***.

It represents a storage unit equivalent to the size of the largest machine register of the target architecture.

Its bit width is defined by constants `BitsPerByte` and `BytesPerLongWord`.

Its value range is `[0..POW2(BitsPerByte*BytesPerLongWord)-1]`.

Whole number literals are ***compatible*** with type `LONGWORD`.

11.2.4 ADDRESS

Type `ADDRESS` is a (semi-abstract) reference type.

It represents the addressable memory space available to the Modula-2 runtime environment.

Its bit width is defined by constant `BitsPerAddress`.

Its value range is `[0..POW2(BitsPerAddress)-1]`.

Whole number literals and `NIL` are ***compatible*** with type `ADDRESS`.

11.2.4.1 Address Mapping

On platforms where the smallest addressable storage unit is eight bits, type `ADDRESS` represents machine addresses of the target architecture. On (legacy) platforms where the smallest addressable unit is not eight bits, the type represents abstract addresses that need to be mapped to the platform's actual address space. On such platforms, address mapping is implementation defined.

Regardless of the underlying architecture, type `ADDRESS` is classified as a ***machine type***.

11.2.4.2 Dual Representation

Type `ADDRESS` exhibits type duality. It is presented both as a pointer type and as an ***array type***. It may thus be thought of as having two overlapping definitions

```
TYPE ADDRESS =
| POINTER TO OCTET
| SEQUENCE OF OCTET (* conceptual pseudo—code, not valid Modula—2 syntax *)
END; (* ADDRESS *)
```

11.2.4.2.1 Presentation as a Pointer

An entity `a` of type `ADDRESS` may be dereferenced using the `^` operator. An expression of the form `a^` is of type `OCTET` and represents the octet value stored at address `a`.

```
adr := UNSAFE.ADR(x);
WRITE "octet value at address ", adr, " is ", adr^;
```

11.2.4.2.2 Presentation as a Stream

An entity `a` of type `ADDRESS` may also be dereferenced by subscript. An expression of the form `a[n]` is of type `OCTET` and represents the octet value stored at address `a + n`, where `n` is an offset of type `LONGCARD`. Negative indices are not supported.

```
adr := UNSAFE.ADR(x);
WRITE "octet value at address ", adr, "[", offset, "] is ", adr[offset];
```

11.2.5 Casting Formal Types

Import of module `UNSAFE` enables the use of casting formal type syntax `CAST ADDRESS` and `CAST OCTETSEQ` within the importing module scope. Without such import the syntax is unavailable and any attempt to use it will cause a syntax error at compile time.

11.2.5.1 CAST ADDRESS

An argument passed to a parameter of formal type `CAST ADDRESS` is cast to type `ADDRESS`. That is, within the body of the procedure, the argument is interpreted as a value of type `ADDRESS`. It must be of a ***pointer type***.

```
PROCEDURE Allocate ( VAR ptr : CAST ADDRESS; size : LONGCARD );
...
TYPE FooPtr = POINTER TO Foo; VAR foo : FooPtr;
Allocate(foo, TSIZE(Foo));
```

All ***pointer type***s are ***passing compatible*** to formal type `CAST ADDRESS`.

11.2.5.2 CAST OCTETSEQ

An argument passed to a parameter of formal type `CAST OCTETSEQ` is cast to an ***octet sequence****. That is, within the body of the procedure, the parameter is interpreted as a stream of values of type `OCTET`, individually addressable by subscript. The argument may be of any type.

```
PROCEDURE HexDump ( dataSeq : CAST OCTETSEQ );
...
VAR foo : Foo; bar : Bar; baz : Baz;
HexDump(foo), HexDump(bar); HexDump(baz);
```

Any entity regardless of its type is ***passing compatible*** to formal parameters of type `OCTETSEQ`, but a parameter of type `OCTETSEQ` is ***passing compatible*** only to formal parameters of type `OCTETSEQ`.

When an argument is passed to a parameter of formal type `CAST OCTETSEQ`, the ***allocation size*** of the argument is passed as a hidden parameter of type `LONGCARD` immediately before the casting parameter.

```
PROCEDURE HexDump ( (*hidden size : LONGCARD;*) dataSeq : CAST OCTETSEQ );
```

The value of the hidden parameter may be obtained within the procedure body by calling function `LENGTH` with the identifier of the casting parameter as its argument.

```
LENGTH(dataSeq) (* number of octets passed to dataSeq *)
```

*This replaces the implicit casting semantics of `ARRAY OF WORD` and `ARRAY OF LOC` in earlier versions of Modula-2.

Operations on parameters of type `OCTETSEQ` are strictly limited to `[` `]` subscript, `FOR` iteration, `ADR()`, `CAST()` and `LENGTH()`. The limitation does not apply to the octet values of an ***octet sequence*** parameter.

11.2.5.2.1 Addressing by Subscript

The values of an ***octet sequence*** parameter are addressable by subscript. Negative indices are not supported.

```
WRITE "octet value at index ", index, " is ", dataSeq[index];
```

11.2.5.2.2 Iteration by FOR Loop

An ***octet sequence*** parameter is iterable by `FOR` loop.

```
PROCEDURE HexDump ( VAR dataSeq : CAST OCTETSEQ );
BEGIN
  FOR value IN dataSeq DO
    WRITE value, " "
  END (* FOR *)
  WRITE "\n", LENGTH(dataSeq), " values\n"
END HexDump;
```

11.3 Low-Level Procedures

11.3.1 ADD

Procedure `ADD` interprets the bit patterns of its operands as unsigned numbers and adds the value of the second to the first operand, ignoring overflow.

```
PROCEDURE ADD ( VAR x : T1; y : T2 );
```

where `T1`, `T2` $\in \tau_{builtin} \setminus \tau_{real} \wedge$ $tsize$ `T1` $\geq tsize$ `T2`

11.3.2 SUB

Procedure `SUB` interprets the bit patterns of its operands as unsigned numbers and subtracts the value of the second from the first operand, ignoring underflow.

```
PROCEDURE SUB ( VAR x : T1; y : T2 );
```

where `T1`, `T2` $\in \tau_{builtin} \setminus \tau_{real} \wedge$ $tsize$ `T1` $\geq tsize$ `T2`

11.3.3 SETBIT

Procedure `SETBIT` sets the bit at the zero-based index given by its second argument of the value given by its first argument to the bit value given by its third argument.

```
PROCEDURE SETBIT ( VAR target : T1; bitIndex : T2; bit : BOOLEAN );
```

where `T1`, `T2` $\in \tau_{builtin} \wedge$ `T2` $\in \tau_{cardinal} \wedge 0 \leq$ `bitIndex` $< 8 \cdot tsize$ `T1`

The bit index of the least significant bit is always zero.

11.3.4 HALT

Procedure `HALT` aborts the running program and passes an exit value to its host environment. On host environments that do not support an exit value, the exit value is discarded. On host environments that support an exit value, the range and meaning of exit values is dependent on the host environment.

```
PROCEDURE HALT ( exitValue : T );
```

where `T` $\in \tau_{whole}$

11.4 Low-Level Functions

11.4.1 ADR

Function `ADR` returns the address of its argument.

```
PROCEDURE ADR ( value : T ) : ADDRESS;
```

where `T` $\in \tau_{any}$

11.4.2 CAST

Function `CAST` represents an unsafe type transfer operation. It returns the value of its second argument, interpreted as if it was of the type given by its first argument, regardless of whether the value makes sense for the target type or not.

```
PROCEDURE CAST ( typeIdent; value : T1 ) : T2; (* pseudo-code *)
```

where `T1` $\in \tau_{any} \wedge$ `typeIdent` $= \boldsymbol{id}$ `T2` $\wedge \boldsymbol{tsize}$ `T1` $\leq \boldsymbol{tsize}$ `T2`

If the second argument is a compile time expression, then it must be any of `NIL`, `TRUE` or `FALSE`, or it must be an explicitly typed expression of the form `expr::T`.

If the ***allocation size*** of the second argument is smaller than the ***allocation size*** of the target type, the higher bits will be filled with zeroes up to the bit width of the target type.

11.4.3 BIT

Function `BIT` returns `TRUE` if the bit at the zero-based index given by its second argument of the value given by its first argument is set, otherwise `FALSE`.

```
PROCEDURE BIT ( value : T1; bitIndex : T2 ) : BOOLEAN;
```

where `T1` $\in \tau_{builtin} \wedge$ `T2` $\in \tau_{cardinal} \wedge 0 \leq$ `bitIndex` $< 8 \cdot \boldsymbol{tsize}$ `T1`

The bit index of the least significant bit is always zero.

11.4.4 SHL

Function `SHL` returns the value of its first argument bit-shifted to the left by the number of positions passed in its second argument.

```
PROCEDURE SHL ( value : T1; shiftFactor : T2 ) : T3;
```

where `T1` $\in \tau_{builtin} \wedge$ `T2` $\in \tau_{cardinal} \wedge$ `T3` $=$ `T1`

Bits that are shifted out to the left are discarded and will not cause underflow.

11.4.5 SHR

Function `SHR` returns the value of its first argument bit-shifted logically to the right by the number of positions passed in its second argument.

```
PROCEDURE SHR ( value : T1; shiftFactor : T2 ) : T3;
```

where `T1` $\in \tau_{builtin} \wedge$ `T2` $\in \tau_{cardinal} \wedge$ `T3` $=$ `T1`

Bits that are shifted out to the right are discarded and will not cause overflow.

11.4.6 BWNOT

Function `BWNOT` returns the bitwise logical negation of its argument.

```
PROCEDURE BWNOT ( value : T1 ) : T2;
```

where `T1` $\in \tau_{builtin} \wedge$ `T2` $=$ `T1`

11.4.7 BWAND

Function `BWAND` returns the bitwise logical conjunction of its first argument and a bit mask given by its second argument.

```
PROCEDURE BWAND ( value : T1; mask : T2 ) : T3;
```

where `T1` $\in \tau_{builtin} \wedge$ `T2` $\in \tau_{cardinal} \wedge \boldsymbol{tsize}$ `T2` $= \boldsymbol{tsize}$ `T1` $\wedge$ `T3` $=$ `T1`

11.4.8 BWOR

Function `BWOR` returns the bitwise logical disjunction of its first argument and a bit mask given by its second argument.

```
PROCEDURE BWOR ( value : T1; mask : T2 ) : T3;
```

where `T1` $\in \tau_{builtin} \wedge$ `T2` $\in \tau_{cardinal} \wedge \boldsymbol{tsize}$ `T2` $= \boldsymbol{tsize}$ `T1` $\wedge$ `T3` $=$ `T1`

Chapter 12

Pragma Facilities

Pragmas are non-semantic directives – instructions to the language processor to control or influence the compilation process but do not alter the semantics of the compiling source code. There are two kinds:

- language defined pragmas
- implementation defined pragmas

12.1 Pragma Symbol Naming

- Pragma symbols of language defined pragmas are always in all-uppercase
- Pragma symbols of implementation defined pragmas may **not** be in all-uppercase
- The implementation prefix matches constant `Impl.ShortName`
- Implementation prefix and implementation defined pragma symbols follow the syntax of `StdIdent`

12.2 Language Defined Pragmas

The revised Modula-2 language defines 18 language defined pragmas but the Bootstrap Kernel Subset only includes the two pragmas for foreign function interfacing.

12.2.1 FFI Pragma

The `FFI` pragma follows the module identifier in the header of a definition module and establishes the calling convention of the specified foreign API, language and its execution environment for the module.

```
FFIPragma := '<*' FFI '=' ForeignAPISpecifier '>' ;
ForeignAPISpecifier := '"C"' | '"CLR"' | '"JVM"' ;
```

```
(* Interface to the C stdio library *)
DEFINITION MODULE CStdIO <*FFI="C"*>;
```

12.2.2 FFIDENT Pragma

The `FFIDENT` pragma follows the identifier of a constant, type, variable or function definition or declaration and maps it to the foreign API identifier specified in the pragma.

```
FFIdentPramga := '<*' FFIDENT '=' ForeignIdentifier '*>' ;
alias ForeignIdentifier = QuotedLiteral ;
```

```
(* Interface to an OpenVMS SMG library routine *)
PROCEDURE PasteVirtualDisplay <*FFIDENT="SMG$PASTE_VIRTUAL_DISPLAY"*>
  ( displayID, pasteboardID : INTEGER; row, column : INTEGER ) : CARDINAL;
```

12.3 Implementation Defined Pragmas

Implementation defined pragmas are specific to a language processor and should therefore be ignored by other language processors. An implementation defined pragma may be preceded with an implementation specific prefix in order to distinguish it from other implementation defined pragmas for other processors that may use the same pragma symbol. When a processor does not understand an implementation defined pragma it will emit an informational, warning, error or fatal error message depending on the pragma's message mode suffix.

```
ImplDefinedPragma :=
  '<*' ( implPrefix '.' )? pragmaSymbol ( '=' constExpr )? '|' ctMsgMode '*>' ;
alias implPrefix, pragmaSymbol = <implementation-defined> ; /* follows StdIdent syntax */
ctMsgMode := INFO | WARN | ERROR | FATAL ;
```

Use of an implementation defined pragma of a fictional compiler is shown below:

```
<*JollyM2.UnrollLoops=TRUE|WARN*>
FOR plane IN array DO
  FOR row IN plane DO
    FOR col IN row DO
      ...
    END (* FOR *)
  END (* FOR *)
END; (* FOR *)
<*JollyM2.UnrollLoops=FALSE|WARN*>
```

Terms of Reference

Allocation Size

The ***allocation size*** of a type T, denoted as $tsize\ T$ is the size of the memory required to allocate an instance of T. It is measured in units of 8 bits called octets.

Capacity

The ***capacity*** of a collection c is the maximal number of values that can be stored in c. Its value is determined by the type of c.

def $capacity\ c = tlimit\ T$ where $c \in: T$

The ***capacity*** of a *pass-by-value* open array parameter p_{val} is equivalent to the ***cardinality*** or ***value count*** of the collection passed to p_{val}.

def $capacity\ p_{val} = count\ c$ where c is the collection passed to p_{val}

The ***capacity*** of a *pass-by-reference* open array parameter p_{ref} is equivalent to the ***capacity*** of the collection passed to p_{ref}.

def $capacity\ p_{ref} = capacity\ c$ where c is the collection passed to p_{ref}

Capacity Limit

The ***capacity limit*** of a ***collection type*** T, denoted as $tlimit\ T$ is the maximum number of items that an instance of T can store. A ***capacity limit*** of zero indicates that the ***collection type*** does not impose any hard limit.

Cardinality

The ***cardinality*** of a set A, denoted as $|A|$ or $card\ A$ is a measure of the number of elements in A. In this specification, the definition is extended to include collections, open array parameters and variadic argument lists.

def the ***cardinality*** of a collection c is the number of components stored in c

def the ***cardinality*** of an open array parameter p is that of the collection passed in p

def the ***cardinality*** of a variadic argument list v is the number of arguments passed in v

Compatibility

Compatibility is a property that determines whether one entity may be assigned to, copied to, passed to or used in an expression with another entity.

def a literal is fully ***compatible*** with type T if it is ***assignment***, ***copy***, ***by-value*** and ***by-reference passing compatible*** to, and ***expression compatible*** with type T

def type T_1 is fully ***compatible*** with type T_2 if it is ***assignment***, ***copy***, ***by-value*** and ***by-reference passing compatible*** to, and ***expression compatible*** with type T_2

Compatibility, Assignment

Assignment compatibility is a property that determines whether one entity may be assigned to another entity.

def a literal is ***assignment compatible*** to type T if it may be assigned to an L-value of type T

def type T_1 is ***assignment compatible*** to type T_2
if an R-value of type T_1 may be assigned to an L-value of type T_2

Compatibility, Copy

Copy compatibility is a property that determines whether one entity may be copied to another entity.

def a literal is ***copy compatible*** to type T if it may be copied to a value of type T

def type T_1 is ***copy compatible*** to type T_2
if an R-value of type T_1 may be copied to an L-value of type T_2

Compatibility, Passing

Parameter ***passing compatibility*** is a property that determines whether an entity may be passed to a formal parameter.

def a literal is ***by-value passing compatible*** to type T
if it may be passed to a formal by-value parameter of type T

def a literal is ***by-reference passing compatible*** to type T
if it may be passed to a formal `CONST` parameter of type T

def type T_1 is ***by-value passing compatible*** to type T_2
if a value of type T_1 may be passed to a formal by-value parameter of type T_2

def type T_1 is ***by-reference passing compatible*** to type T_2
if a value of type T_1 may be passed to a formal by-reference parameter of type T_2

Compatibility, Expression

Expression compatibility is a property that determines whether an entity may be used together with another entity as operands in a binary expression.

def a literal is ***expression compatible*** to type T in respect of operation M if it may be used together with a value of type T as the operands in a binary expression with any operator representing operation M

def types T_1 and T_2 are ***expression compatible*** in respect of operation M if a value of type T_1 and a value of type T_2 may be used together as the operands in a binary expression with any operator representing operation M

Constant Expression

A ***constant expression*** is an expression that is computable at compile time. It does not reference any variables nor invoke any library functions.

Fractional Part

The ***integral part*** of a real number x is the value that represents the digits of x before the decimal point if x was to be written out in decimal notation without an exponent.

def $\boldsymbol{frac}\ x = x - \boldsymbol{int}\ x$

Integral Part

The ***integral*** part of a real number x is the value that represents the digits of x before the decimal point if x was to be written out in decimal notation without an exponent.

def $\boldsymbol{int}\ x = \boldsymbol{sgn}\ x \cdot \boldsymbol{entier}\ \boldsymbol{abs}\ x$

Largest Integer or Entier

The ***entier*** value of a real number x is the largest integer less than x.

def $\boldsymbol{entier}\, x = \boldsymbol{max}\, i$ where $i \in \{\mathbb{Z} \mid i < x\}$

Length

The ***length*** of a character string s is the number of characters stored in s excluding its NUL terminator.

def $\forall\ s \in: \tau_{string} : \boldsymbol{length}\, s = \boldsymbol{min}\, n$ where $n \in \{\mathbb{N} \mid s_n =$ ASCII.NUL $\}$

The ***length*** of an ***octet sequence*** s is the ***allocation size*** of the actual parameter p passed in to a formal ***octet sequence*** parameter.

def $\forall\ s \in: \tau_{octetseq} : \boldsymbol{length}\, s = \boldsymbol{capacity}\, p$ where p is passed to s

Number Sets

In this specification abstract number sets are defined as follows:

def the set of the abstract ***natural numbers*** $\mathbb{N} = \{0, 1, 2, 3 \cdots \infty\}$

def the set of the abstract ***ordinal numbers*** $\mathbb{O} = \{0, 1, 2, 3 \cdots \omega, \omega + 1, \omega + 2 \cdots\}$

def the set of the abstract ***whole numbers*** $\mathbb{Z} = \{-\infty \cdots -3, -2, -1, 0, +1, +2, +3 \cdots +\infty\}$

Odd Value

The ***odd value*** of a whole number i indicates whether i is odd.

def $\boldsymbol{odd}\, i = \boldsymbol{true}$ for $i \in \{\mathbb{Z} \mid \boldsymbol{abs}\, i\ \boldsymbol{mod}\, 2 = 1\}$

def $\boldsymbol{odd}\, i = \boldsymbol{true}$ for $i \in \{\mathbb{Z} \mid \boldsymbol{abs}\, i\ \boldsymbol{mod}\, 2 = 0\}$

Signum Value

The ***signum*** or $\boldsymbol{sgn}$ value of a signed number x indicates the sign of x.

def $\boldsymbol{sgn}\, x = \quad 1$ for $x \in \{\mathbb{R} \mid x > 0\}$

def $\boldsymbol{sgn}\, x = \quad 0$ for $x \in \{\mathbb{R} \mid x = 0\}$

def $\boldsymbol{sgn}\, x = -1$ for $x \in \{\mathbb{R} \mid x < 0\}$

Type, Any

A type is a classification of storage units defined by a set of three properties: storage representation, value range and semantics.

The set of all types is denoted τ_{any}.

Type, Builtin

A ***builtin type*** is a type that is built into the language core, that is, either predefined or provided by builtin module `UNSAFE`.

The set of all ***builtin types*** is denoted $\tau_{builtin}$.

def $\forall\ \tau \in \{\tau_{any} \mid \boldsymbol{builtin}\, \tau\} \Rightarrow \tau \in \tau_{builtin}$

Type, Countable

A ***countable type*** is a type that represents a set of values with a 1:1 correspondence to the first m ***ordinal numbers*** $\mathbb{O}$ where $m \in \mathbb{O}$.

def $\boldsymbol{countable}\ T$ if $\exists\ f : T \to \mathbb{O}$

The set of all ***countable types*** is denoted $\tau_{countable}$.

def $\forall\ \tau \in \{\tau_{any} \mid \boldsymbol{countable}\ \tau\} \Rightarrow \tau \in \tau_{countable}$

Type, Enumerated

An ***enumerated type*** is a type that represents a finite ordered set of identifiers.

def $\boldsymbol{enumerated}\ \tau$ if $\forall\ e \in: \{\tau \mid \boldsymbol{is\text{-}identifier}\ e\} \wedge \exists\ f : \tau \to \mathbb{N}$

The set of all ***enumerated types*** is denoted τ_{enum}.

def $\forall\ \tau \in \{\tau_{any} \mid \boldsymbol{enumerated}\ \tau\} \Rightarrow \tau \in \tau_{enum}$

Type, Machine

A ***machine type*** is a type that represents storage units that match the storage units of the target architecture and have modular arithmetic semantics.

The set of all ***machine types*** is denoted $\tau_{machine}$.

def $\forall\ \tau \in \{\tau_{any} \mid \boldsymbol{is\text{-}machine\text{-}type}\ \tau\} \Rightarrow \tau \in \tau_{machine}$

Type, Signed

A ***signed type*** is a numeric type that represents both positive and negative scalar numeric values.

def $\boldsymbol{signed}\ \tau$ if $\exists\ x, y \in: \{\tau \mid x < 0 \wedge y > 0\}$

The set of all ***signed types*** is denoted τ_{signed}.

def $\forall\ \tau \in \{\tau_{any} \mid \boldsymbol{signed}\ \tau\} \Rightarrow \tau \in \tau_{signed}$

Type, Unsigned

An ***unsigned type*** is a numeric type that represents non-negative scalar numeric values.

def $\boldsymbol{unsigned}\ \tau$ if $\forall\ x \in: \{\tau \mid x \geq 0\}$

The set of all ***unsigned types*** is denoted $\tau_{unsigned}$.

def $\forall\ \tau \in \{\tau_{any} \mid \boldsymbol{unsigned}\ \tau\} \Rightarrow \tau \in \tau_{unsigned}$

Type, Whole Number

A ***whole number type*** is a numeric type that represents a subset of the set of integers $\mathbb{Z}$.

def $\boldsymbol{whole}\ \tau$ if $\forall\ n \in: \{\tau \mid n \in \mathbb{Z}\}$

The set of all ***whole number types*** is denoted τ_{whole}.

def $\forall\ \tau \in \{\tau_{any} \mid \boldsymbol{whole}\ \tau\} \Rightarrow \tau \in \tau_{whole}$

Type, Real Number

A ***real number type*** is a numeric type that represents a subset of the set of real numbers $\mathbb{R}$.

def $\boldsymbol{real}\ \tau$ if $\forall\ r \in: \{\tau \mid r \in \mathbb{R}\}$

The set of all ***real number types*** is denoted τ_{real}.

def $\forall\ \tau \in \{\tau_{any} \mid \boldsymbol{real}\ \tau\} \Rightarrow \tau \in \tau_{real}$

Type, Cardinal

A ***cardinal type*** is a numeric type that represents a subset of the unsigned ***whole numbers*** $\mathbb{Z}^+$.

def $\boldsymbol{cardinal}\ \tau$ if $\boldsymbol{unsigned}\ \tau\ \wedge\ \boldsymbol{whole}\ \tau$

The set of all ***cardinal number types*** is denoted $\tau_{cardinal}$.

def $\forall\ \tau \in \{\tau_{any} \mid \boldsymbol{cardinal}\ \tau\} \Rightarrow \tau \in \tau_{cardinal}$

Type, Integer

An ***integer type*** is a numeric type that represents a subset of the signed ***whole numbers*** $\mathbb{Z}$.

def $\boldsymbol{integer}\ \tau$ if $\boldsymbol{signed}\ \tau\ \wedge\ \boldsymbol{whole}\ \tau$

The set of all ***integer types*** is denoted $\tau_{integer}$.

def $\forall\ \tau \in \{\tau_{any} \mid \boldsymbol{integer}\ \tau\} \Rightarrow \tau \in \tau_{integer}$

Type, Scalar

A ***scalar type*** is a numeric type that represents scalar values.

The set of all ***scalar types*** is denoted τ_{scalar}.

def $\forall\ \tau \in \{\tau_{any} \mid \boldsymbol{scalar}\ \tau\} \Rightarrow \tau \in \tau_{scalar}$

Type, Collection

A ***collection type*** represents n-tuples of an arbitrary component type, where n is bounded by its ***capacity***.

The set of all ***collection types*** is denoted $\tau_{collection}$.

def $\forall\ \tau \in \{\tau_{any} \mid \boldsymbol{collection}\ \tau\} \Rightarrow \tau \in \tau_{collection}$

Type, Array

An ***array type*** is an ordered ***collection type*** whose components are mapped to the set of the first m ***ordinal numbers*** $\mathbb{O}$ where $m - 1$ is the ***capacity limit*** of the ***array type***.

The set of all ***array types*** is denoted τ_{array}.

def $\forall\ \tau \in \{\tau_{collection} \mid \boldsymbol{array}\ \tau\} \Rightarrow \tau \in \tau_{array}$

Type, String

A ***string type*** is an ***array type*** whose component type is a character type.

The set of all ***string types*** is denoted τ_{string}.

def $\forall\ \tau \in \{\tau_{array} \mid \boldsymbol{string}\ \tau\} \Rightarrow \tau \in \tau_{string}$

Type, Record

The set of all ***record types*** is denoted τ_{record}.

def $\forall\ \tau \in \{\tau_{any} \mid \boldsymbol{record}\ \tau\} \Rightarrow \tau \in \tau_{record}$

Type, Extensible Record

The set of all ***extensible record types*** is denoted τ_{extrec}.

def $\forall\ \tau \in \{\tau_{record} \mid \boldsymbol{extensible}\ \tau\} \Rightarrow \tau \in \tau_{extrec}$

Type, Extensible Record Pointer

An ***extensible record pointer type*** is a non-opaque ***pointer type*** whose target type is an ***extensible record type***.

def $\tau_{recptr} = \{\tau \in \tau_{pointer} \setminus \tau_{opaque} \mid \boldsymbol{target\text{-}of}\ \tau \in \tau_{extrec}\}$

The set of all ***extensible record pointer types*** is denoted τ_{recptr}.

def $\forall\ \tau \in \{\tau_{pointer} \mid \boldsymbol{is\text{-}recptr}\ \tau\} \Rightarrow \tau \in \tau_{recptr}$

Type, Octet Sequence

Casting formal type `OCTETSEQ` is formally called the ***octet sequence type*** and denoted $\tau_{octetseq}$.

Type, Opaque

An ***opaque type*** is a ***pointer type*** whose identifier is visible upon import but whose target type is hidden within the implementation part of the library module that institutes it.

The set of all ***opaque types*** is denoted τ_{opaque}.

def $\forall\ \tau \in \{\tau_{pointer} \mid \boldsymbol{is\text{-}opaque}\ \tau\} \Rightarrow \tau \in \tau_{opaque}$

Type, Open Array

An ***open array type*** is a formal ***array type*** whose ***capacity*** is indeterminate at compile time.

The set of all ***open array types*** is denoted $\tau_{openarray}$.

def $\forall\ \tau \in \{\tau_{formal} \mid \boldsymbol{is\text{-}open\text{-}array}\ \tau\} \Rightarrow \tau \in \tau_{openarray}$

Type, Variadic Parameter List

A ***variadic parameter list*** is a formal ***collection type*** whose ***cardinality*** is indeterminate at compile time.

The set of all ***variadic parameter lists*** is denoted $\tau_{arglist}$.

def $\forall\ \tau \in \{\tau_{formal} \mid \boldsymbol{is\text{-}variadic\text{-}list}\ \tau\} \Rightarrow \tau \in \tau_{arglist}$

Value Count

The ***value count*** of an entity e is synonym with the ***cardinality*** of e.

Wirthian Macro

A ***Wirthian macro*** is a built-in syntactic entity whose occurrence is replaced with an appropriate library call, possibly filling in omitted arguments. The macro may be a statement or a built-in procedure or function. The mapped-to library must be explicitly imported.*

*Classic examples of Wirthian macros are procedures `NEW` and `DISPOSE` in earlier versions of Modula-2.

Appendix A

Managing Technical Debt

Technical debt is a metaphor for the sum total consequences of improper (software) design and implementation. Such deficiencies may be unintentional, or they may well be intentional in an attempt to shorten time to delivery. In either case, resulting deficiencies tend to accumulate over time until they reach a critical point where further development and maintenance on the software is severely impaired.

The metaphor suggests – analogous to financial debt – that (1) technical debt has to be paid down over time and with interest, and (2) interest will grow exponentially if the debt is not paid down. If technical debt is left unchecked for too long, the software may become unmaintainable. Eventually, the time and cost to reduce the deficiencies to a manageable level may vastly outpace the time and cost to rewrite the software from scratch.

In many organisations it is next to impossible to obtain the resources required to do proper maintenance on larger software projects without quantifying the debt, its financial impact and the cost required to reduce it. But without documenting the technical debt first, it is impossible to obtain that data and present figures with any confidence. Documenting and reporting of technical debt is thus a necessary prerequisite to managing the debt.

A.1 Documenting Debt

Modula-2 provides a facility to document technical debt directly within the source code. The facility is incorporated into the language syntax so that documented issues become machine readable for automated report generation. A reporting tool may then generate reports on the weight, quantity and location of technical debt within a code base directly from its source code. A simple report writer may be built into the language processor. For more complex analysis an external report writer may be used.

A.2 Syntax

A `TO` `DO` list is a machine readable directive to document technical debt directly within the source code. It may be placed wherever a definition, a declaration or a statement can occur. It may be empty, or contain a tracking reference of an external issue tracking system, and a list of tasks with descriptions and effort estimations.

```
toDoList := TO DO ( trackingRef? taskToDo ( ';' taskToDo )* END )? ;
trackingRef := '(' issueId ( ',' severity ',' description )? ')' ;
taskToDo := description ( ',' estimatedTime timeUnit )? ;
alias issueId, severity, estimatedTime = NumberLiteral ;
alias description = StringLiteral ;
alias timeUnit = StdIdent ; /* must be any of w, d, or h */
```

A.3 Reporting Debt

The language processor must report any `TO` `DO` list with line number as an informational compile time message and any syntax violation within as a compile time error. It must further report the total number of `TO` `DO` lists found in a compilation unit. Any further analysis and reporting is implementation defined. It is recommended to implement more complex analysis and reporting as a separate reporting tool outside of the language processor.

A.4 Examples

A.4.1 TO DO without Arguments

The arguments may be omitted from a `TO` `DO` list altogether.

```
PROCEDURE TitleCaseTransform ( VAR s : ARRAY OF CHAR );
BEGIN
  FOR index, char IN s DO
    TO DO
  END (* FOR *)
END TitleCaseTransform;
```

A.4.2 TO DO with Issue Tracking

A tracking reference of an external issue tracking system may be included in a `TO` `DO` list.

```
PROCEDURE skipBlockComment ( infile : Infile ) : Token;
  TO DO (1337, 2 (*minor*), "empty comment not recognised")
    "verify loop exit conditions", 1h;
    "fix and test", 1h
  END; (* TO DO *)
BEGIN
```

A tracking reference may refer to another at a different location while omitting its severity and description.

```
  TO DO (1337)
    "Fix: move increment to the end of the loop"
  END; (* TO DO *)
```

A.4.3 TO DO without Issue Tracking

The tracking reference may be omitted in a `TO` `DO` list.

```
TYPE DynString = POINTER TO StringDescriptor;
TO DO
  "bindings to NEW and RETAIN", 2d;
  "bindings to @VALUE, @STORE, @LENGTH", 2d;
  "bindings to @EQUALS, @PRECEDES, @SUCCEEDS", 4d;
  "bindings to APPEND, @ATINSERT, @ATREMOVE", 3d;
  "bindings to READ and WRITE, using template", 2d;
  "unit testing", 2d
END; (* TO DO *)
```

A.5 Sample Report

```
$ m2tdr --summary --proj m2bsk
Modula-2 technical debt summary report for project m2bsk
found 14 issues with 39 tasks in 9 modules, thereof
   5 issues of weight 3 (major) with 23 tasks, est. 338 hours,
   7 issues of weight 2 (minor) with 12 tasks, est. 121 hours,
   2 issues of weight 1 (cosmetic) with 4 tasks, est. 19 hours;
found further 26 untracked tasks in 21 modules, est. 312 hours.
Total estimated effort for 65 tasks: 790 hours (avg. 12 hours/task).
```

Appendix B

Implementation Parameters

Module `Impl` provides constants with implementation specific parameters. Its provision is mandatory. Where any of the required constants are dependent on user selectable compiler options, implementors may implement module `Impl` as a built-in module.

B.1 Identity

Constant	Type	Meaning
`ShortName`	String Literal	short name of the language processor
`LongName`	String Literal	long name of the language processor
`MajorVersion`	Whole Number	major version of the language processor
`MinorVersion`	Whole Number	minor version of the language processor
`SubMinorVersion`	Whole Number	sub-minor version of the language processor
`BuildNumber`	Whole Number	build number of the language processor
`HostEnvironment`	String Literal	name of the host environment or operating system

Table B.1: Constants indicating the implementation identity

B.2 Capability Flags

Boolean Constant	indicates whether language processor supports ...
`SupportsM2BSK`	Modula-2 R10 Bootstrap Kernel (M2BSK) subset
`SupportsM2R10`	Modula-2 R10 (M2R10) full specification
`SupportsUtf8Source`	source files encoded in UTF-8
`SupportsCFFI`	C foreign function interface and `<*FFI="C"*>` pragma
`SupportsCLRFFI`	Microsoft CLR FFI and `<*FFI="CLR"*>` pragma
`SupportsJVMFFI`	Java Virtual Machine FFI and `<*FFI="JVM"*>` pragma

Table B.2: Constants indicating implementation capabilities

B.3 Implementation Limits

Whole Number Constant	indicates ...	required
`MaxIdentLength`	maximum significant length of identifiers	≥ 31
`MaxNumLiteralLength`	maximum length of number literals	≥ 40
`MaxStrLiteralLength`	maximum length of string literals	≥ 120
`MaxLineCounter`	maximum value of line counter for error messages	≥ 32000
`MaxColumnCounter`	maximum value of column counter for error messages	≥ 160
`MaxEnumValues`	maximum number of values of an enumeration type	≥ 4096
`MaxSetElements`	maximum number of elements in an enumerated set	≥ 128
`MaxProcNestLevel`	maximum nesting level of local procedures	≥ 2

Table B.3: Constants indicating implementation limits

Appendix C

Target Parameters

Module `Trgt` provides constants with target specific parameters. Its provision is mandatory. Where any of the required constants are dependent on user selectable compiler options, implementors may implement module `Trgt` as a built-in module.

C.1 Identity

Constant	Type	Meaning
`ShortName`	String Literal	short name of the target
`LongName`	String Literal	long name of the target
`MajorVersion`	Whole Number	major version of the code generator for this target
`MinorVersion`	Whole Number	minor version of the code generator for this target
`SubMinorVersion`	Whole Number	sub-minor version of the code generator for this target

Table C.1: Constants indicating the target identity

C.2 Classification

Boolean Constant	indicates whether target ...
`IsProcessor`	is a hardware architecture
`isVirtualProcessor`	Is a processor independent assembler backend
`isVirtualMachineHost`	Is a virtual machine host and runtime environment
`isHigherLevelLanguage`	Is a language processor for a higher level programming language

Table C.2: Constants indicating target classification

C.3 Characteristics

Boolean Constant	indicates whether ...
`IsBigEndian`	target uses big-endian byte order
`IsLittleEndian`	target uses little-endian byte order
`SupportsTCE`	code generator supports tail call elimination for this target

Table C.3: Constants indicating target characteristics

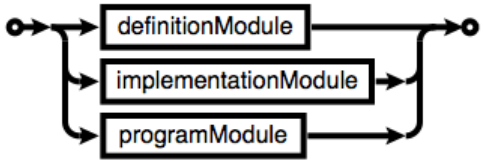
Appendix D

Syntax Diagrams

Non-Terminals

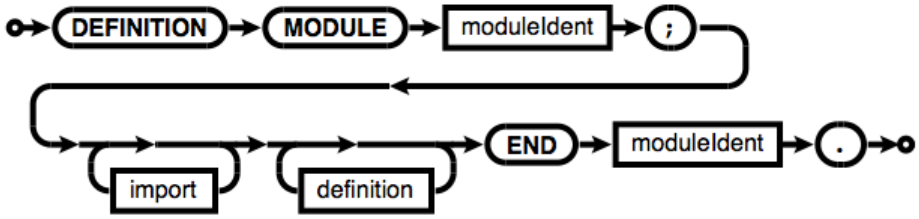
Start Symbol

compilationUnit

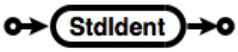


Definition Module Syntax

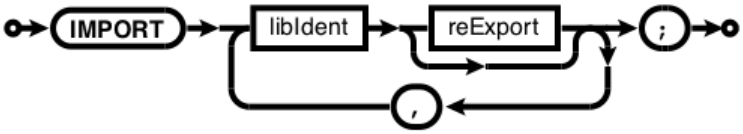
definitionModule



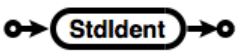
moduleIdent



import

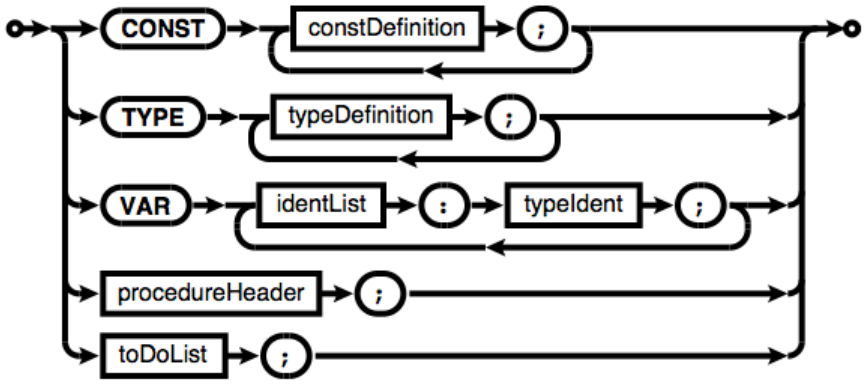


libIdent



reExport

definition

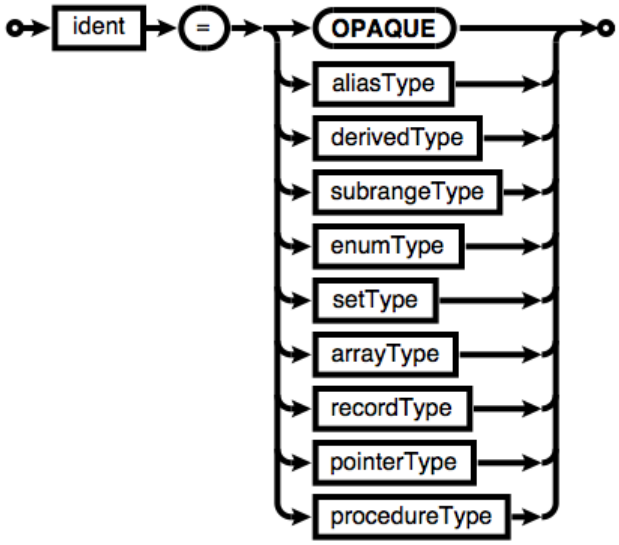


constDefinition

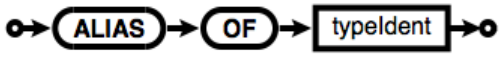
ident

constExpression

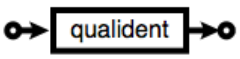
typeDefinition



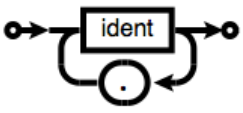
aliasType



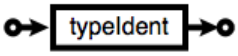
typeIdent



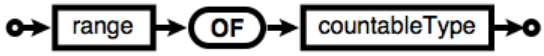
qualident



derivedType

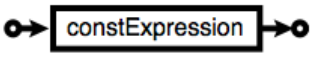


subrangeType

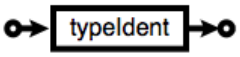


range

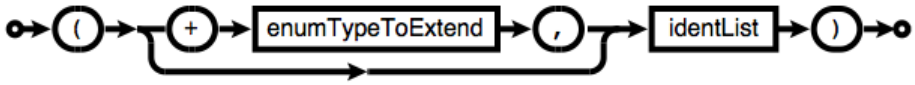
lowerBound



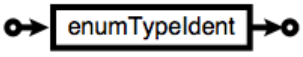
countableType



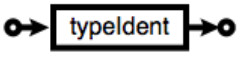
enumType



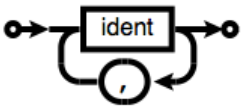
enumTypeToExtend



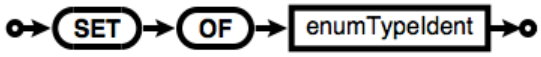
enumTypeIdent



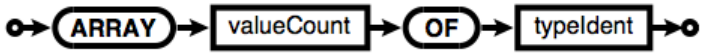
identList



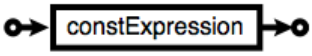
setType



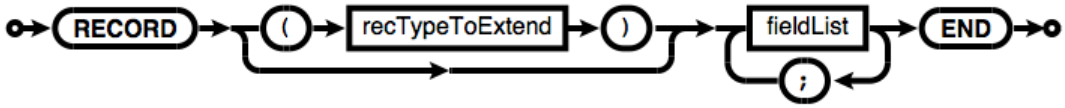
arrayType



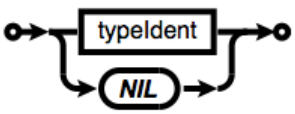
valueCount



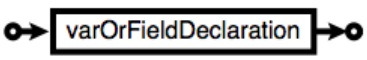
recordType



recTypeToExtend

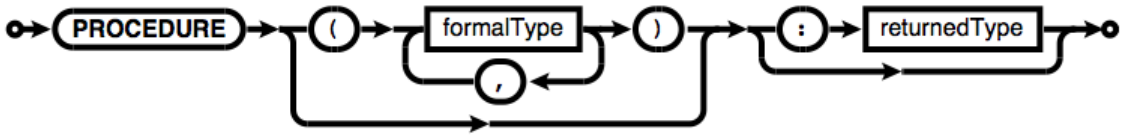


fieldList

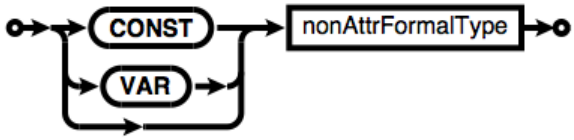


pointerType

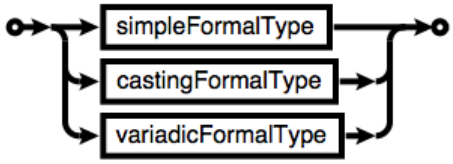
procedureType



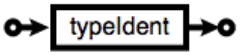
formalType



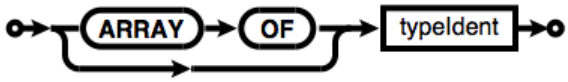
nonAttrFormalType



returnedType



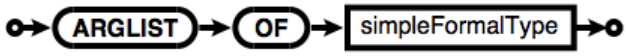
simpleFormalType



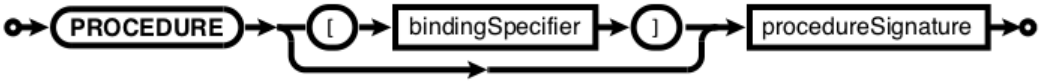
castingFormalType

△ Syntax enabled by import of module `UNSAFE`

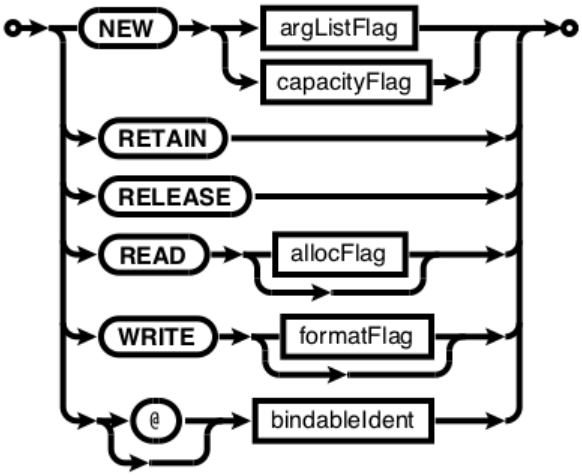
variadicFormalType



procedureHeader



bindingSpecifier



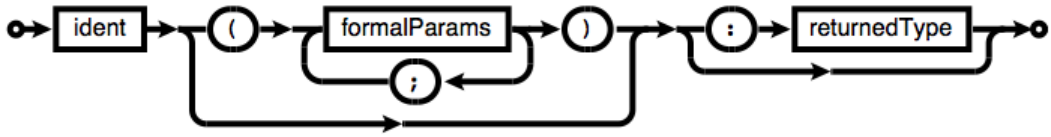
argListFlag

capacityFlag

formatFlag

bindableIdent

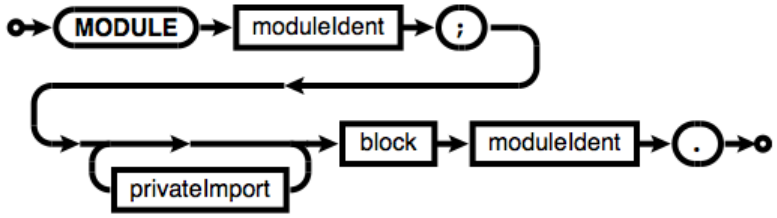
procedureSignature



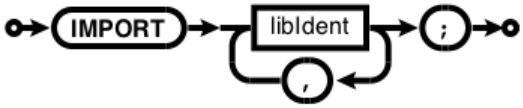
formalParams

Implementation and Program Module Syntax

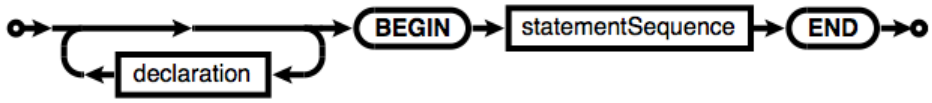
programModule



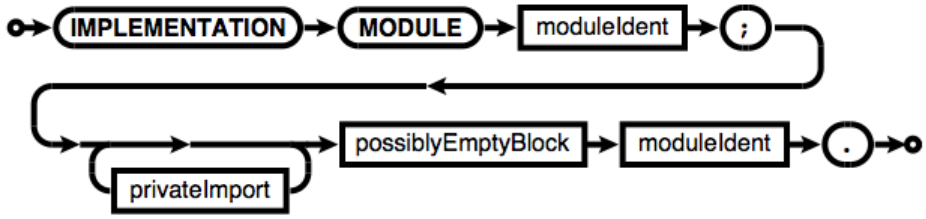
privateImport



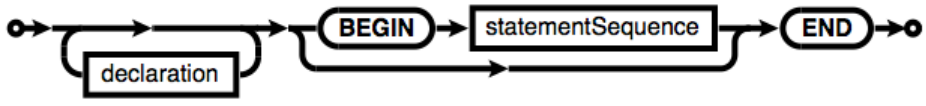
block



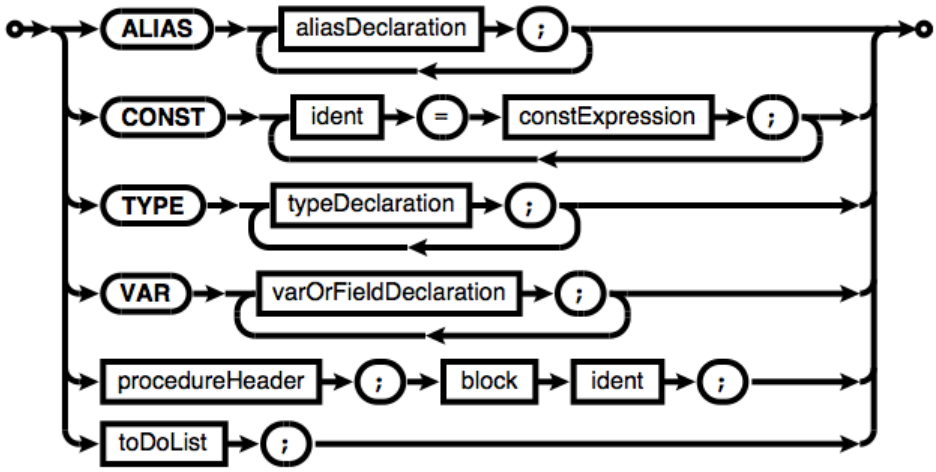
implementationModule



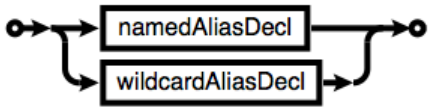
possiblyEmptyBlock



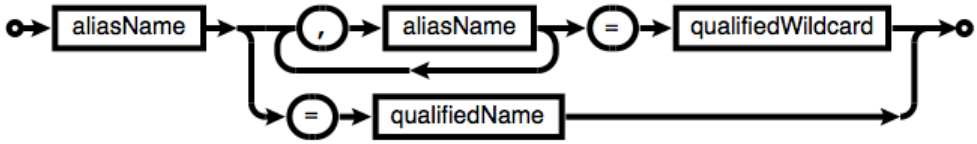
declaration



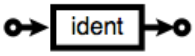
aliasDeclaration



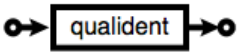
namedAliasDecl



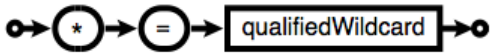
aliasName



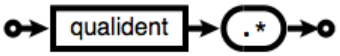
qualifiedName



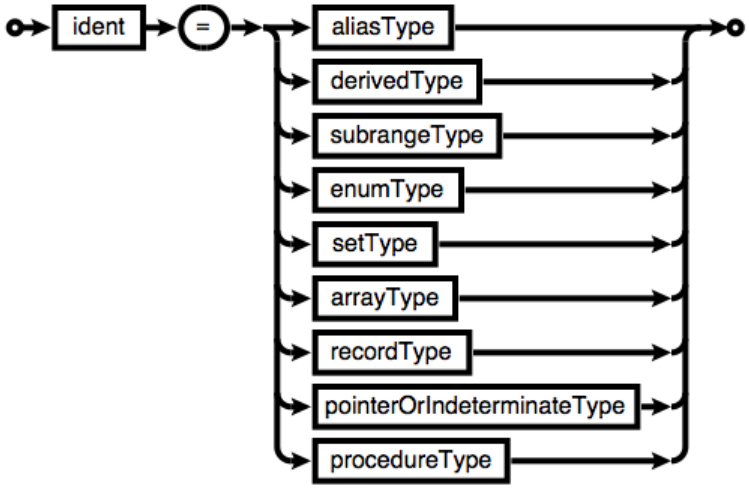
wildcardAliasDecl



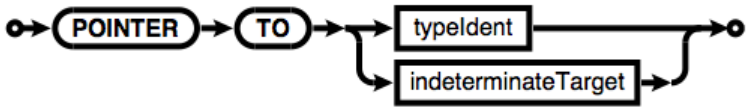
qualifiedWildcard



typeDeclaration

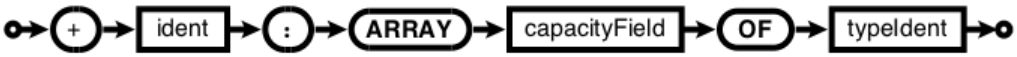


pointerOrIndeterminateType

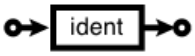


indeterminateTarget

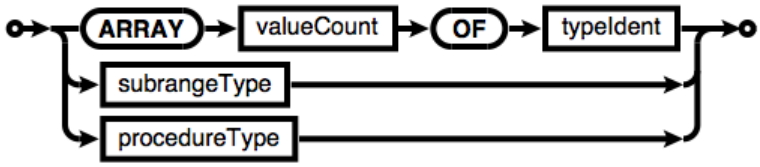
indeterminateField



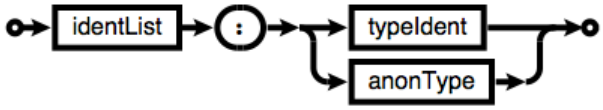
capacityField



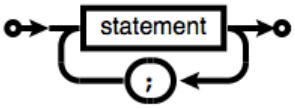
anonType



varOrFieldDeclaration

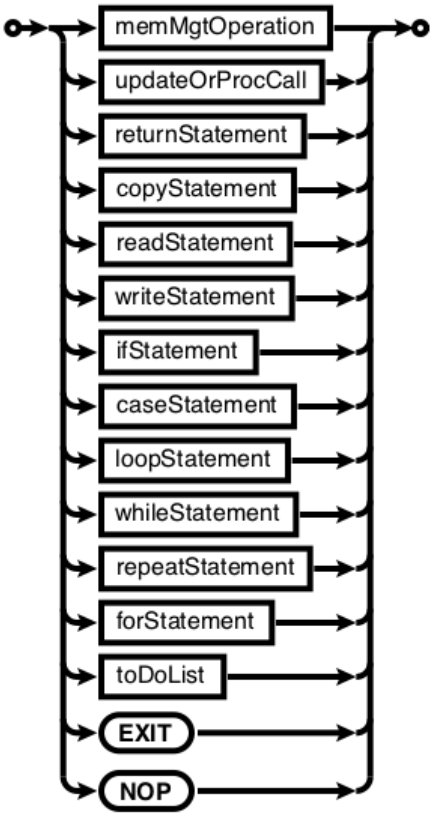


statementSequence

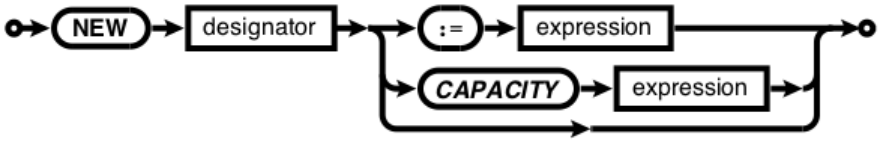


memMgtOperation

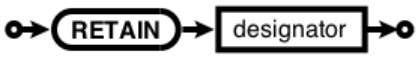
statement



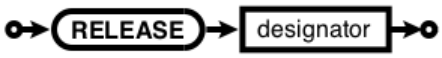
newStatement



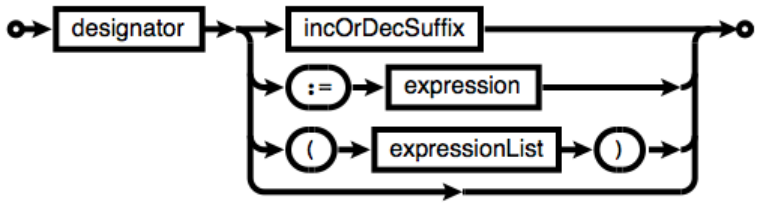
retainStatement



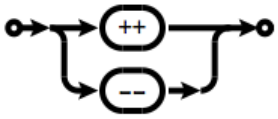
releaseStatement



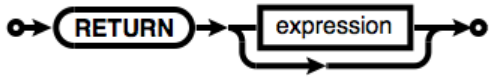
updateOrProcCall



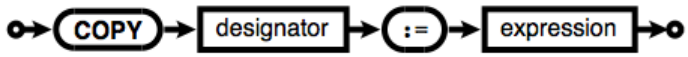
incOrDecSuffix



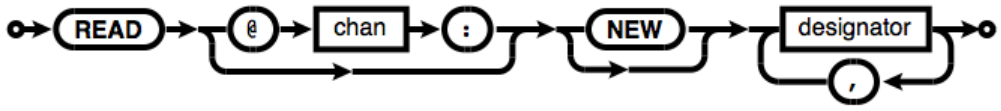
returnStatement



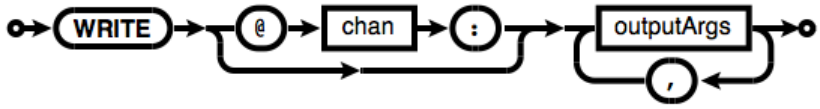
copyStatement



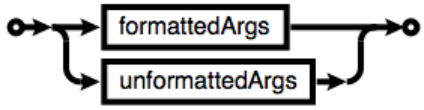
readStatement



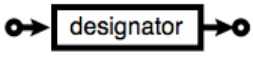
writeStatement



outputArgs

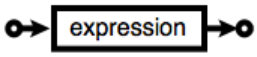


chan

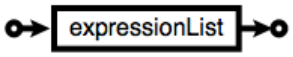


formattedArgs

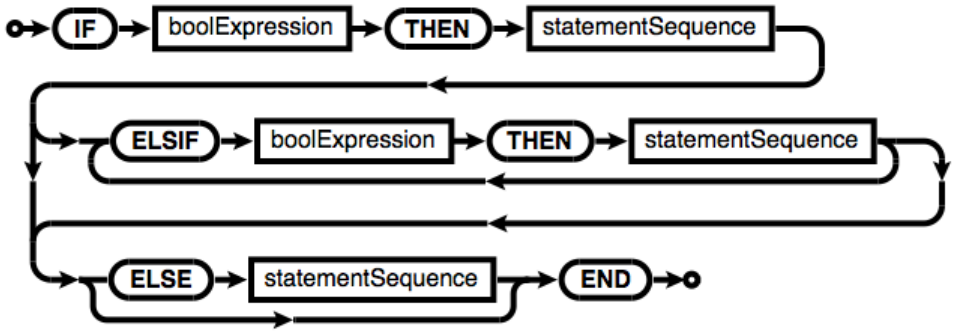
fmtStr



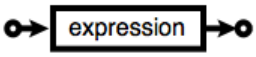
unformattedArgs



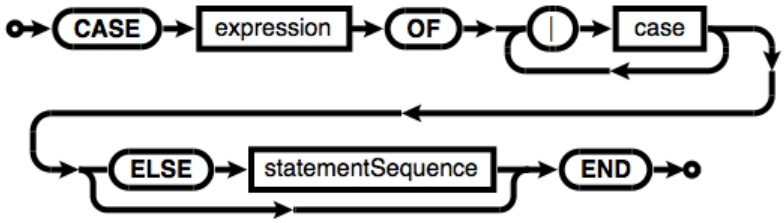
ifStatement



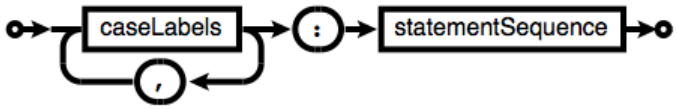
boolExpression



caseStatement

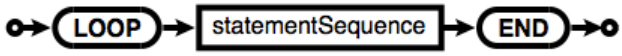


case



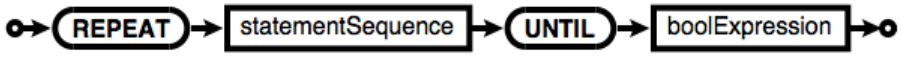
caseLabels

loopStatement

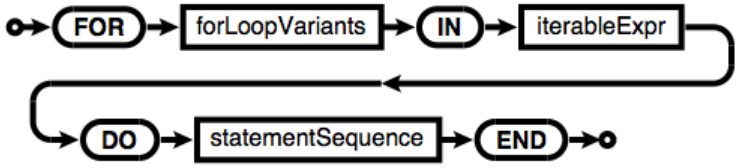


whileStatement

repeatStatement



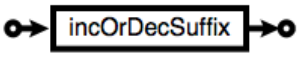
forStatement



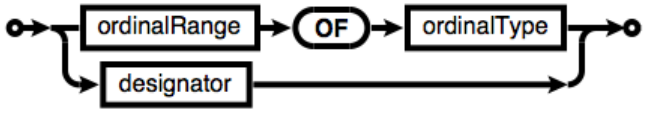
forLoopVariants

indexOrSoleValue

ascOrDesc

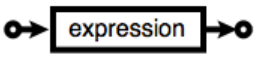


iterableExpr

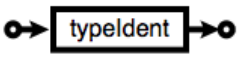


ordinalRange

firstValue

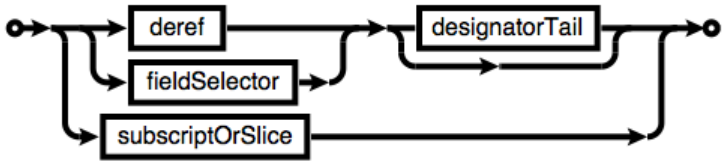


ordinalType



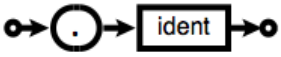
designator

designatorTail

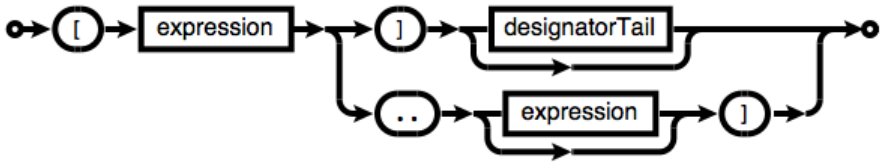


deref

fieldSelector

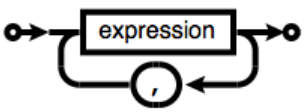


subscriptOrSlice

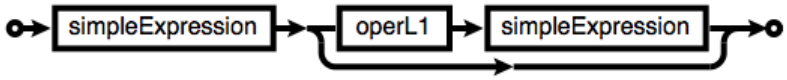


Expressions

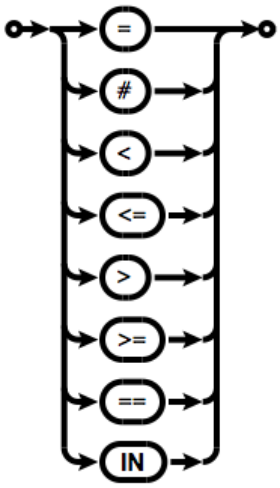
expressionList



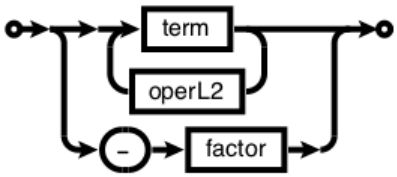
expression



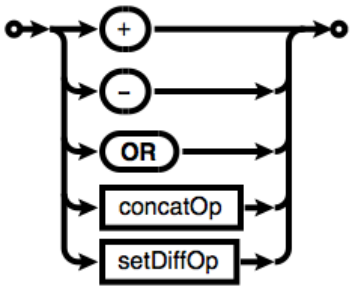
operL1



simpleExpression



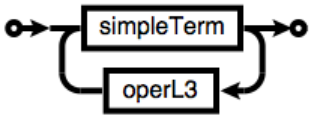
operL2



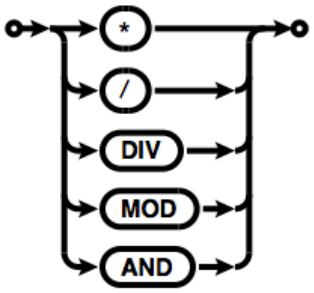
concatOp

setDiffOp

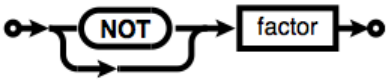
term



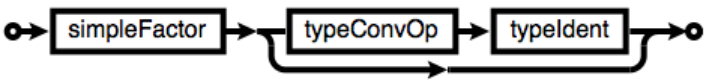
operL3



simpleTerm

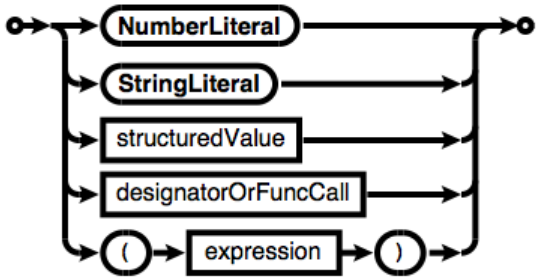


factor

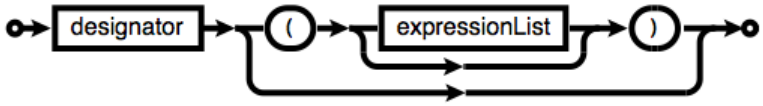


typeConvOp

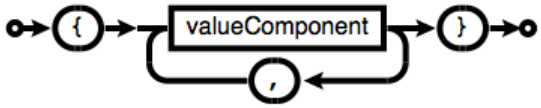
simpleFactor



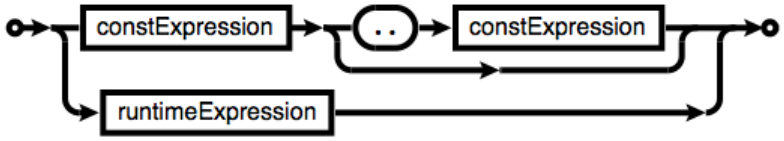
designatorOrFuncCall



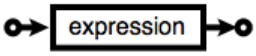
structuredValue



valueComponent

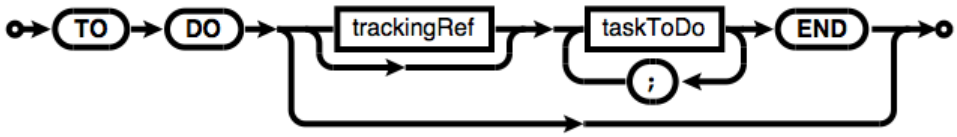


runtimeExpression

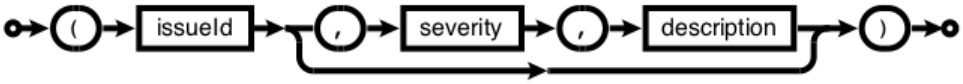


To Do Lists

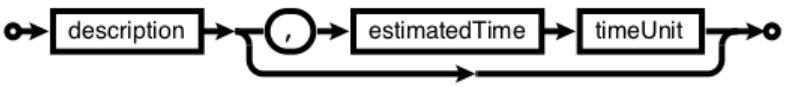
toDoList



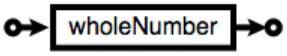
trackingRef



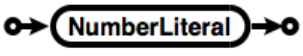
taskToDo



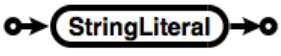
issueId, severity, estimatedTime



wholeNumber



description

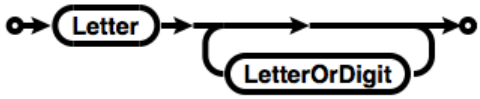


timeUnit

△ By convention `timeUnit` identifiers should be `w` for weeks, `d` for days, and `h` for hours

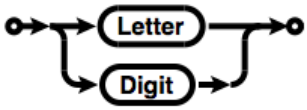
Terminals

StdIdent

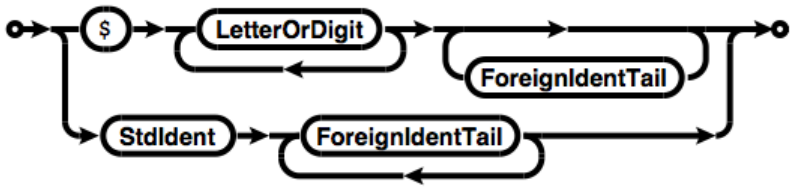


△ Default identifier syntax

LetterOrDigit

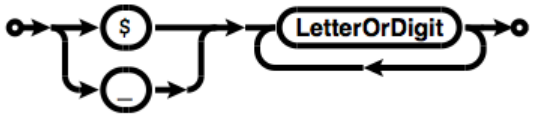


ForeignIdent

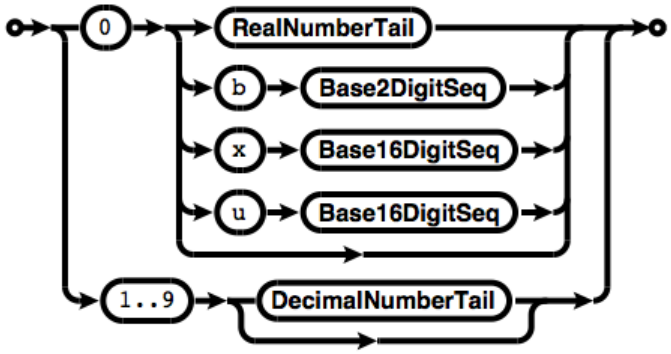


△ Foreign identifier syntax for interfacing to foreign APIs, enabled by FFI pragma or compiler switch

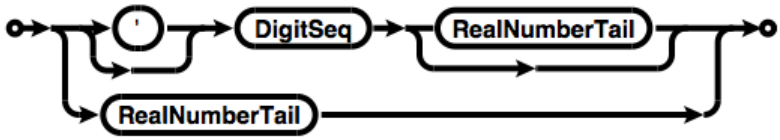
ForeignIdentTail



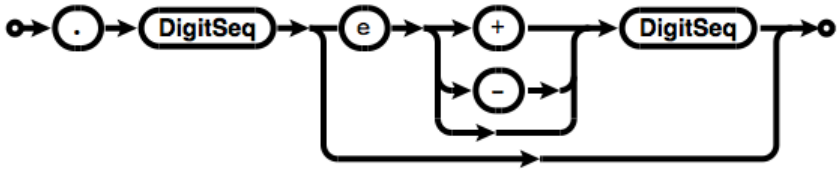
NumberLiteral



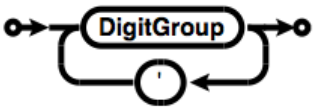
DecimalNumberTail



RealNumberTail



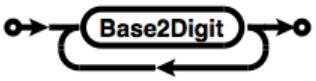
DigitSeq



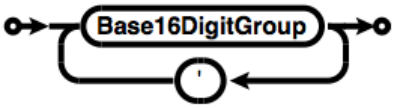
DigitGroup

Base2DigitSeq

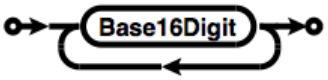
Base2DigitGroup



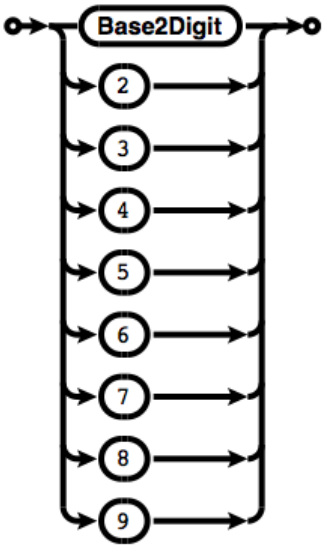
Base16DigitSeq



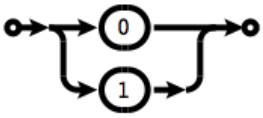
Base16DigitGroup



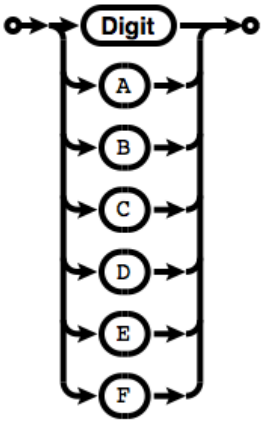
Digit



Base2Digit

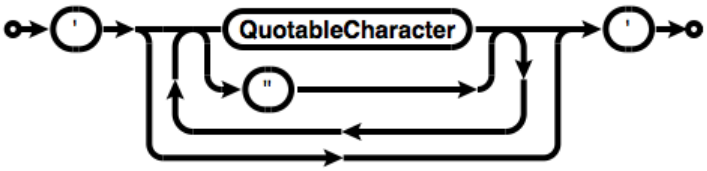


Base16Digit

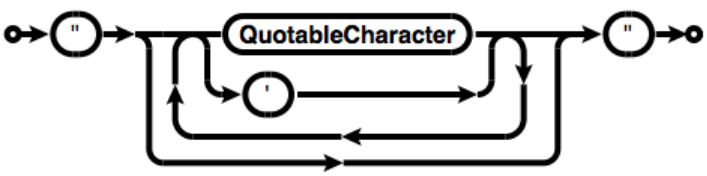


StringLiteral

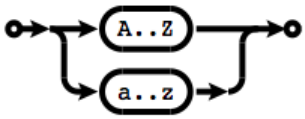
SingleQuotedString



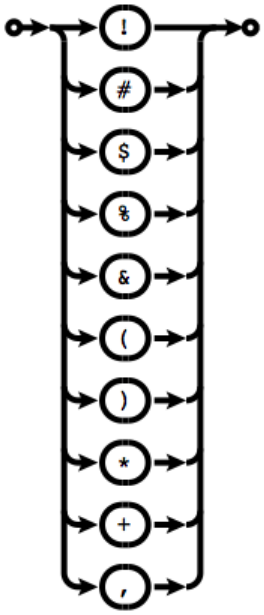
DoubleQuotedString

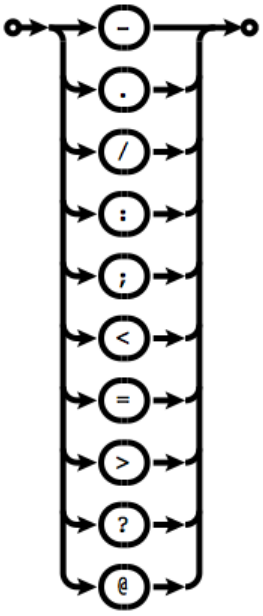


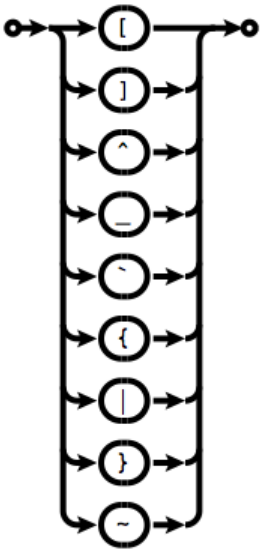
Letter



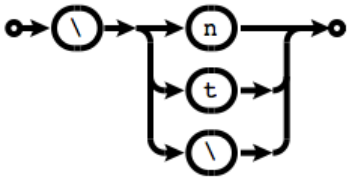
NonAlphaNumQuotable





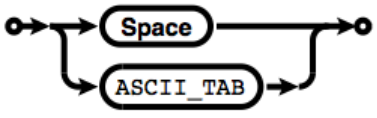


EscapedCharacter

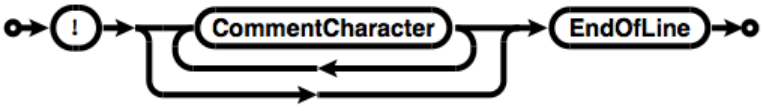


Ignore Symbols

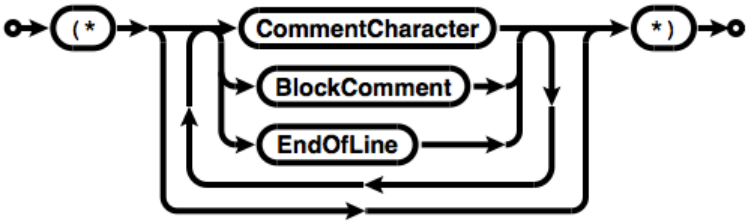
Whitespace



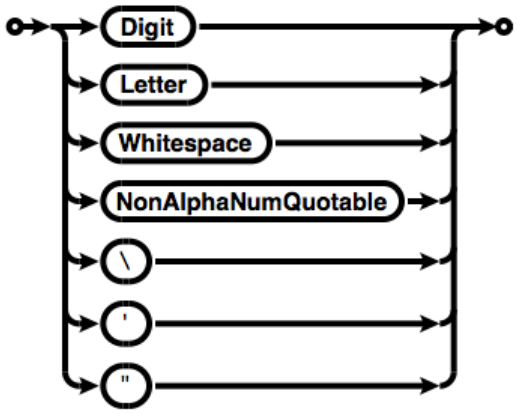
LineComment



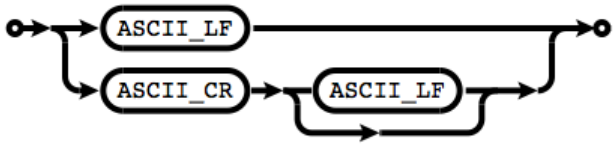
BlockComment



CommentCharacter

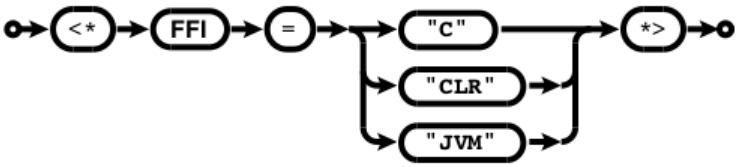


EndOfLine

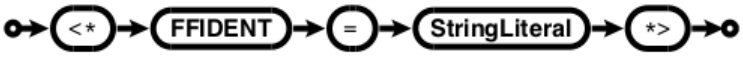


Pragmas

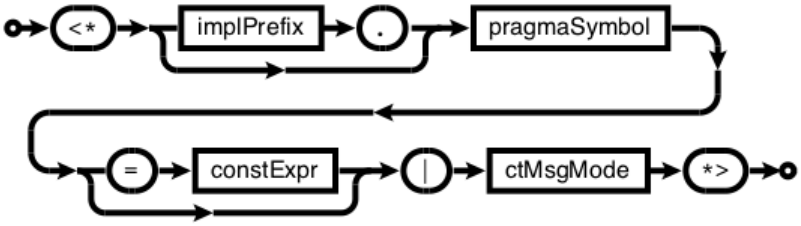
ffiPragma



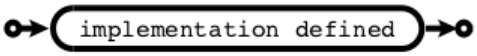
ffidentPragma



implDefPragma



implPrefix, pragmaSymbol



ctMsgMode

Appendix E

IO Library Interfaces

Group IO Aggregators

Group IO aggregators are modules that import and re-export the IO library modules for a given group of types. They represent the top user-level of the type specific IO library. The standard library defines two group IO aggregator modules:

`PervasiveIO`

IO aggregator `PervasiveIO` imports and re-exports the IO library modules for the pervasive types.

```
DEFINITION MODULE PervasiveIO;
IMPORT BooleanIO+, CharIO+, UnicharIO+,
  OctetIO+, CardinalIO+, LongCardIO+, IntegerIO+, LongIntIO+, RealIO+, LongRealIO+;
END PervasiveIO.
```

`MachineTypeIO`

IO aggregator `MachineTypeIO` imports and re-exports the IO library modules for the ***machine types***.

```
DEFINITION MODULE MachineTypeIO;
IMPORT ByteIO+, WordIO+, LongWordIO+, AddressIO+;
END MachineTypeIO.
```

Type Specific IO Aggregators

The standard library defines 14 type specific IO aggregator modules, one for each pervasive type and one for each ***machine type***. Each type specific IO aggregator imports and re-exports the IO modules for values and arrays of a given type.

The module identifier of a type specific IO aggregator for a given type `T` is composed from the titlecase transformed identifier of type `T`, followed by suffix `IO`. Type identifiers with prefix `LONG` are considered two words, others are considered one word. Thus, the titlecase transformation of `UNICHAR` is `Unichar`, but that of `LONGCARD` is `LongCard`.

The IO aggregator for a given type `T` may be generated from the template* below by replacing

- all occurrences of `<#TypeIdent#>` with the identifier of type `T`, and
- all occurrences of `<#TitleCasedTypeIdent#>` with the titlecase transformation of the identifier of type `T`.

Common template for pervasive types and ***machine types***

```
DEFINITION MODULE <#TitleCasedTypeIdent#>IO;
IMPORT Sole<#TypeIdent#>IO+, Array<#TypeIdent#>IO+;
END <#TitleCasedTypeIdent#>IO.
```

*Template expansion may easily be automated using the Unix `awk` or `sed` utilities.

Type Specific Value IO Modules

The standard library defines 14 type specific value IO modules, one for each pervasive type and one for each ***machine type***. Each type specific value IO module provides IO bindings for reading and writing sole values of a given type.

The module identifier of a type specific value IO module for a given type `T` is composed from prefix `Sole`, followed by the titlecase transformed identifier of type `T`, followed by suffix `IO`.

The interface of a value IO module for a given type `T` may be generated from one of the two templates below by replacing

- all occurrences of `<#TypeIdent#>` with the identifier of type `T`, and
- all occurrences of `<#TitleCasedTypeIdent#>` with the titlecase transformation of the identifier of type `T`.

Template for pervasive types

```
DEFINITION MODULE Sole<#TitleCasedTypeIdent#>IO;
IMPORT ChanIO;
PROCEDURE [READ] Read ( chan : ChanIO.Channel; VAR value : <#TypeIdent#> );
PROCEDURE [WRITE] Write ( chan : ChanIO.Channel; value : <#TypeIdent#> );
PROCEDURE [WRITE#] WriteF ( chan : ChanIO.Channel; CONST fmtStr : ARRAY OF CHAR; value : <#TypeIdent#> );
END Sole<#TitleCasedTypeIdent#>IO.
```

Template for ***machine types***

```
DEFINITION MODULE Sole<#TitleCasedTypeIdent#>IO;
IMPORT UNSAFE, ChanIO;
PROCEDURE [READ] Read ( chan : ChanIO.Channel; VAR value : UNSAFE.<#TypeIdent#> );
PROCEDURE [WRITE] Write ( chan : ChanIO.Channel; value : UNSAFE.<#TypeIdent#> );
PROCEDURE [WRITE#] WriteF
  ( chan : ChanIO.Channel; CONST fmtStr : ARRAY OF CHAR; value : UNSAFE.<#TypeIdent#> );
END Sole<#TitleCasedTypeIdent#>IO.
```

Type Specific Array IO Modules

The standard library defines 14 type specific array IO modules, one for each pervasive type and one for each ***machine type***. Each type specific array IO module provides IO bindings for reading and writing arrays of a given type.

The module identifier of a type specific array IO module for a given type `T` is composed from prefix `ArrayOf`, followed by the titlecase transformed identifier of type `T`, followed by suffix `IO`.

The interface of an array IO module for a given type `T` may be generated from one of the two templates below by replacing

- all occurrences of `<#TypeIdent#>` with the identifier of type `T`, and
- all occurrences of `<#TitleCasedTypeIdent#>` with the titlecase transformation of the identifier of type `T`.

Template for pervasive types

```
DEFINITION MODULE ArrayOf<#TitleCasedTypeIdent#>IO;
IMPORT ChanIO;
PROCEDURE [READ] Read ( chan : ChanIO.Channel; VAR array : ARRAY OF <#TypeIdent#> );
PROCEDURE [WRITE] Write ( chan : ChanIO.Channel; CONST array : ARRAY OF <#TypeIdent#> );
PROCEDURE [WRITE#] WriteF
  ( chan : ChanIO.Channel; CONST fmtStr : ARRAY OF CHAR; CONST array : ARRAY OF <#TypeIdent#> );
END ArrayOf<#ModuleIdent#>IO.
```

Template for ***machine types***

```
DEFINITION MODULE ArrayOf<#TitleCasedTypeIdent#>IO;
IMPORT UNSAFE, ChanIO;
PROCEDURE [READ] Read ( chan : ChanIO.Channel; VAR array : ARRAY OF UNSAFE.<#TypeIdent#> );
PROCEDURE [WRITE] Write ( chan : ChanIO.Channel; CONST array : ARRAY OF UNSAFE.<#TypeIdent#> );
PROCEDURE [WRITE#] WriteF
  ( chan : ChanIO.Channel; CONST fmtStr : ARRAY OF CHAR; CONST array : ARRAY OF UNSAFE.<#TypeIdent#> );
END ArrayOf<#TitleCasedTypeIdent#>IO.
```

Standard IO Channels

```
DEFINITION MODULE StdIO;
IMPORT ChanIO, IOStatus;
PROCEDURE [@STDIN] stdIn : ChanIO.Channel;
PROCEDURE [@STDOUT] stdOut : ChanIO.Channel;
PROCEDURE stdErr : ChanIO.Channel; (* for use by runtime system *)
PROCEDURE SetStdIn ( chan : ChanIO.Channel; VAR status : IOStatus );
PROCEDURE SetStdOut ( chan : ChanIO.Channel; VAR status : IOStatus );
PROCEDURE SetStdErr ( chan : ChanIO.Channel; VAR status : IOStatus );
END StdIO.
```

Channel Based IO

```
DEFINITION MODULE ChanIO;
IMPORT IOStatus, IOSIZE;
TYPE Channel = POINTER TO Descriptor;
TYPE Descriptor = RECORD ( NIL )
  op : Operations
END;(*Descriptor*)
TYPE Operations = POINTER TO OpDescriptor;
TYPE OpDescriptor = RECORD ( NIL )
  (* method slots for status handling *)
  statusOf    : PROCEDURE ( Channel ) : IOStatus;
  StatusMsg   : PROCEDURE ( Channel, IOStatus );
  SetStatus   : PROCEDURE ( Channel, IOStatus, BOOLEAN );
  (* method slots for parameter query *)
  modeOf      : PROCEDURE ( Channel ) : FileMode;
  nameLen     : PROCEDURE ( Channel ) : LONGCARD;
  GetName     : PROCEDURE ( Channel, VAR ARRAY OF CHAR );
  (* method slots for generic input *)
  dataReady   : PROCEDURE ( Channel, IOSIZE ) : BOOLEAN;
  ReadOctet   : PROCEDURE ( Channel, VAR OCTET );
  ReadBlock   : PROCEDURE ( Channel, VAR ARRAY OF OCTET );
  insertReady : PROCEDURE ( Channel ) : BOOLEAN;
  Insert      : PROCEDURE ( Channel, OCTET );
  (* method slots for generic output *)
  WriteOctet  : PROCEDURE ( Channel, OCTET );
  WriteBlock  : PROCEDURE ( Channel, ARRAY OF OCTET, VAR IOSIZE );
  isFlushable : PROCEDURE ( Channel ) : BOOLEAN;
  Flush       : PROCEDURE ( Channel );
END; (*OpDescriptor*)
END IOChannel.
```

Status

```
DEFINITION MODULE Status;
(* Status Root Type *)
TYPE Status = RECORD ( NIL )
  failed : BOOLEAN
END; (*Status*)
END Status.
```

```
DEFINITION MODULE StatusCode;
(* Status Code Base Type *)
TYPE StatusCode = ( Success, Failure );
END StatusCode.
```

IO Status

```
DEFINITION MODULE IOStatus;
IMPORT Status, StatusCode;
TYPE IOStatus = RECORD ( Status )
  (* inherits failed : BOOLEAN *)
  code : IOStatusCode
END; (*IOStatus*)
END IOStatus.
```

```
DEFINITION MODULE IOStatusCode;
TYPE IOStatusCode =
  ( +StatusCode,             (* inherits Success and Failure       *)
    FileNotFound,            (* no file found with this filename   *)
    NameTooLong,             (* the passed filename is too long    *)
    IllegalCharsInName,      (* illegal chars in passed filename   *)
    InvalidMode,             (* the passed mode is invalid         *)
    AlreadyOpen,             (* the passed file is already open    *)
    MayNotOpenDirectory,     (* attempt to open a directory        *)
    AccessDenied,            (* the filesystem denied file access  *)
    AccessBeyondEOF,         (* attempt to read past end of file   *)
    NotOpenForReading,       (* file is not in Read mode           *)
    NotOpenForWriting,       (* file is not in Write/Append mode   *)
    OutOfRange,              (* input data is out of target range  *)
    WrongFormat,             (* input data is in unexpected format *)
    UnexpectedEndOfLine,     (* unexpected end-of-line in input    *)
    UnexpectedEndOfInput,    (* unexpected end of input data       *)
    ConnectionClosed,        (* remote end closed the connection   *)
    InvalidFilePos,          (* attempt to set a invalid position  *)
    InsertBufferFull,        (* capactiy of insert buffer exceeded *)
    FileSizeLimitExceeded,   (* attempt to write past size limit   *)
    OpenFileLimitExceeded,   (* attempt to open too many files     *)
    OperationNotSupported,   (* unsupported operation attempted    *)
    DeviceFull,              (* the device capacity is exceeded    *)
    DeviceError );           (* the device reported a failure      *)
END IOStatusCode.
```

IO Size

```
DEFINITION MODULE IOSIZE;
(* assuming systems where TSIZE(LONGCARD)<64 don't have 64-bit file systems *)
TYPE IOSIZE = ALIAS OF LONGCARD;
(* on systems where file size exceeds LONGCARD, however unlikely,
   IOSIZE may be redefined accordingly and the library rebuilt. *)
END IOSIZE.
```

Appendix F

IO Library Format Specifiers

Format Specifier Syntax for Type BOOLEAN

```
boolFmtStr := shortBool | longBool ;
shortBool := '[' ( trueSym | falseSym ) ']' ;
alias trueSym, falseSym = PrintableChar ;
longBool := 'TF' | 'Tf' | 'tf' | 'YN' | 'Yn' | 'yn' ;
```

```
WRITE #("TF", bool); (*TRUE or FALSE*)  WRITE #("[0|1]", bool); (*0 or 1*)
```

Format Specifier Syntax for Character Types

```
charFmt := quoted? glyph | codePoint ;
quoted := 'Q' | 'QQ' ;
glyph = 'A' | 'UTF8' ;
codePoint := 'U+' digitCount | '0u' digitFmt? ;
digitFmt := digitCount digitGrouping? ;
digitCount := leastNumOfDigits | fixedNumOfDigits ;
alias leastNumOfDigits = '*' ;
fixedNumOfDigits := Digit+ ;
digitGrouping := digitSep digitsPerGroup ;
digitSep := localeDependent | apostrophe | space ;
alias digitsPerGroup = fixedNumOfDigits ;
alias localeDependent = ':' ;
alias apostrophe = "'"
alias space = '_' ;
```

```
WRITE #("Q", char); (*'$'*)  WRITE #("QQ", char); (*"$"*)  WRITE #("0u8'2", char); (*0u00'00'00'24*)
```

Format Specifier Syntax for Type OCTET

```
octetFmt := decimalOctet | octetBase16 | base2 ;
decimalOctet := leadSpace? compact | leadZeroes ;
alias leadSpace = '_' ;
alias leadZeroes = 'Z' ;
alias compact = '*' ;
octetBase16 := '0x' | 'X' ;
base2 := ( '0b' | 'B' ) digitFmt ;
```

```
WRITE #("0x", n); (*0x2A*)  WRITE #("Z", n); (*042*)  WRITE #("0b8'4", n); (*0b0010'1010*)
```

Format Specifier Syntax for Cardinal Types

```
cardinalFmt := decimalCardinal | base16 | base2 ;
decimalCardinal := leadZeroes? ( digitFmt justification? )? ;
alias leadZeroes = 'Z';
digitFmt := intDigits digitGrouping? ;
alias intDigits, digitsPerGroup = digitCount ;
digitCount := Digit+ ;
digitGrouping := digitSep digitsPerGroup ;
digitSep := ':' | '_' | "'" ;
justification := '|' position '|' ;
position := '<' | '>' ;
base16 := ( '0x' | 'X' ) digitFmt ;
base2 := ( '0b' | 'B' ) digitFmt ;
```

Format Specifier Syntax for Integer Types

```
integerFmt := sign? decimalCardinal | base16 | base2 ;
sign := '+' | '-' ;
```

```
WRITE #("0x", i); (*0x2A*)  WRITE #("Z", i); (*042*)  WRITE #("0b8'4", i); (*0b0010'1010*)
```

Format Specifier Syntax for Real Number Types

```
realFmt := engOrSciFmt | acctFmt ;
engOrSciFmt := ( 'E' )? sign? significand exponent? justification? ;
sign := '+' | '-' ;
significand := intDigits digitGrouping? ';' fracDigits digitGrouping? ;
digitGrouping := ':' digitsPerGroup ;
exponent := 'e' sign? expDigits digitGrouping? ;
alias intDigits, fracDigits, expDigits, digitsPerGroup = digitCount ;
digitCount := Digit+ ;
acctFmt := 'A' ( standard | ledger ) ;
standard := ( sign spacer? | spacer sign ) significand ;
alias spacer = '_' ;
ledger := filler? significand acctSign ;
filler := '*' | '=' ;
acctSign := 'CR' | '()' ;
```

```
WRITE #("E-2;9e3", r); (* 12.345,678,900e006*)  WRITE #("A7:3;2_CR", r); (*1,234,567.89 CR*)
```

Format Specifier Syntax for Character Arrays

```
strFmt := quoted? glyphs ;
quoted := 'Q' | 'QQ' ;
glyphs = 'A' | 'UTF8' ;
```

Format Specifier Syntax for Non-Character Arrays

```
arrayFmt := '{' itemSep '}' ( '|' valueFmt )? ;
itemSep := ( ',' | ';' )? '_' ;
valueFmt := boolFmt | octetFmt | cardinalFmt | integerFmt | realFmt ;
```

To Do:

This page is **not** part of the (final) document.

Date of this draft: 18 July 2020 (#3)

Content

Main Part

- More detail on static versus dynamic allocation
- More detail on allocation of indeterminate records
- More detail on “under-the-hood” transformations

Appendices

- Add interfaces for lower-level IO libraries

Formatting

- Use LaTeX titling package to define fonts/styles for headings
- Change `\rmfamily` back to proper roman font when done

Decision

- Should we include `FIRST()`, `LAST()`, `PREV()` and `NEXT()` and their bindings in the BSK in order to support native syntax for library implemented key/value stores? While not essential for bootstrapping the compiler, it should probably be included in the new edition of PIM and Springer’s ideas about page count will likely force us to limit the book to a subset. Of course, we can always make PIM5 a superset of BSK with these four functions added.