Programming Language Flex
Language Reference Manual

Version: 4.0 (Draft 2.7)

Programming Language Flex

Language Reference Manual

Programming Language Flex
Language Reference Manual

Authors: Ondřej Tučný, Aleš Procháska

Version: 4.0 (Draft 2.7)
Print date: 25.5.2004
File name:.pdf

Contents

History of changes

Changes since version draft 2.6:

- — Floating-point and fixed-point type definition syntax rules clarified.
- — New keywords **abstract** and **expose** and several accidentally missign keywords added to the keyword table in chapter 2.2.
- — The syntax of classes, modules and subprograms extended with **abstract** and **expose**.
- — The metastatement **#define** changed to **#template**.
- — In chapter 5.1, class types prohibited in Type specification.
- — The `target_machine_direct_byteorder` and `host_machine_direct_byteorder` environment variables added to chapter 13.8.
- — Chapter 6.1.1 describing expression classes added.
- — Started describing static and dynamic attributes in chapters 12.2 and 12.3

1. Introduction

Flex is a general purpose programming language. Intended range of usage encloses virtually any type of application, starting from systems programming, through embedded systems' development, small single-purpose applications, information systems, distributed systems, mission-critical applications, up to large demanding multi-platform applications.

Flex is a classic programming language supporting different programming paradigms, including object-oriented programming, formal development methods, and parallel computing. No experimental or unproven language constructs and features were included into this language.

Programming language Flex is the result of research in programming languages and programming techniques held for more than five years at A && L soft. Its features were tested and proven to be practically usable and helpful in real software development of large-scale multi-platform complex applications.

1.1 Design Goals and Fundamental Features

Easy to use programming language with sufficient expressiveness. Flex is designed to allow different programmers with different programming experience level and preferred programming style (in concern with preferred methods how to solve certain problems) to cooperate on a single project. This enables less-experienced programmers to use only simpler language features and build programs from predefined components, while giving advanced programmers the freedom to use all language features.

Flex is based on the Pascal family of languages and was greatly influenced by the Ada programming language. While it is a feature-rich language that offers approximately the same functionality as Ada, it is still easy to learn and can be taught in more levels of difficulty.

Certain language features that were not straightforwardly usable, or implementable, or led to difficult semantic rules with obscure behavior in non-standard situations, were omitted from the language. All language constructs are designed to be straightforward with clear semantics.

The syntax of the language is constructed to improve readability of the program and support formal description of algorithms and coding style rather than ease of writing. All syntax rules are based on few fundamental principles that lead to syntactically consistent source code.

Support for all levels of abstraction. Flex can be used for system level programming as well as for formal description of algorithms on the highest level of abstraction. This is enabled by sufficient support for both data abstraction and precise machine dependent data type specifications.

Various language constructs, which need a strong run-time support system, exist in the language. These constructs are intended to ease software development on a high level of formalization. In Flex, there is no need to manipulate with objects in memory as if they had no type. Unlike in other languages, where various library routines are needed to perform certain actions (e.g. memcpy or memset in C/C++), there is always a language defined construct that is able to do such action without the need of a library routine in a straightforward manner (e.g. assignment of structured variables, aggregates, **nil** as a universal concept).

Flex covers multitasking, inter-task communication, synchronization, and concurrent processing as native qualities of the language. This enables programmers to use most advanced features of the underlaying operating system and hardware platform in a consistent and natural way.

Preventing errors, maintenance cost, and reusability. Today's main goals in software development are reusability and low maintenance cost. Flex offers certain mechanisms to reach such goals with low effort.

Lot of errors in current programs arise from careless of developers, misuse of variables, types and subprograms, and lack of support for advanced features in programming languages. Flex allows the programmer to declare a certain entity in a way, that prevents its accidental misuse in an originally unintended manner. Native support for advanced features like multi-tasking or synchronization gives the programmer the opportunity to concentrate of software engineering instead of solving trivial problems again and again.

Properties of the type system (especially inheritance, encapsulation of data structures and dynamic type information) allow creation of general modules, that can be reused in future projects with low requirements for changes.

Attributes give the programmer the ability to reference certain properties of every entity (like the size in bytes of a type) in a consistent way without the need to declare additional constants or use complex programming rules. Algorithms can be specified without the knowledge of actual values of such attributes leading to better reusability and maintainability of the source code.

Strong type system. The type system covers all simple and compound data types as well as special types supporting other features of the language instead of introducing new types of entities.

Strong type checking, both static and dynamic, is a fundamental concept. Flex emphasizes a type as a key entity, that forms a base for formal description of algorithms. Static type checking rules can be constrained by the programmer enabling the programmer to declare special-purpose types with restricted compatibility — this mechanism can virtually disallow compatibility with any other type.

Inheritance is a general concept, that allows the programmer to build type hierarchies of any kind. Inheritance mechanisms are defined for all data types, even simple types, arrays, and pointers. Together with dynamic type checking, inheritance allows run-time processing of polymorphic data with a common ancestor. Dynamic type information (type tags) is also the base concept utilized in dispatching of virtual methods — any type hierarchy based on any data type can be used for dispatching.

Encapsulation of types enables the programmer to define abstract data types together with related operations, thus leading to better control over a certain data structure, even in situations, where object-oriented programming cannot be used with sufficient effectivity.

The attributes of every type (size of the type, precision of a number, alignment of components etc.) can be either chosen by the compiler to best-fit the target machine, or specified by the programmer. This gives the programmer required control over the generated code and layout of data structures, where appropriate.

Unification of language constructs. Great attention has been given to possible unification of apparently disconnected language constructs. This endeavor led to a merger of exception handling, inter-task communication and synchronization, leading to a powerful mechanism of synchronous and asynchronous messages.

Modules and classes share most of their functionality. Modules can be inherited one from another as well as classes. Virtual methods can be declared both in classes and modules, and were generalized to allow simultaneous dispatching on multiple arguments, not only the class's current instance.

Other example is the inheritance paradigm in the type system.

Future extensibility. All language constructs are designed to be simply extensible in the future, when new ideas and programming techniques evolve. This extensibility is based on an open syntax scheme, like standardized way of building declarations of new types of entities, and overall fundamental semantic concepts, like declarative regions, fundamental concepts of the type system and unification of apparently disconnected language constructs (like inter-task communication and exception handling).

No language defined entities. Flex contains no built-in features such as predefined types, subprograms, constants or other entities. There is a predefined environment, that is intended to enable quick start-up and easy programming of very small single-purpose applications, but it is fully specified in source code and can be modified and even completely eliminated from a certain project — it is not a compiler-built-in feature.

Similarly there is a standardized container module — module `Flex` — for the run-time environment support routines. As well as the predefined environment, it is not a compiler-built-in feature, and it is up to the programmer if he uses this run-time system, redefines it, or completely removes (and so effectively disabling certain language features

— but it is the purpose of the run-time system to provide support for advanced features of the language, like tasking, synchronization, inter-task communication, exception handling etc.).

This enables the programmer to take complete control over the application and disallow any third-party components. This is extremely important in security related and mission-critical applications that need to have certification of all components.

1.2 Structure of This Document

Every chapter usually describes a single language construct (syntax and semantic rules) or a group of related constructs. All text shall be interpreted as normative, if not stated otherwise.

Syntax

Names of the syntax categories are in the left column; definitions are in the right column. Alternatives of a definition are separated by a vertical bar | or a line break. If the definition doesn't fit to a single line, it continues on the next line right indented.

Syntax categories are named by one or more words, first starting with a capital letter and the rest written in lowercase. Words printed in *italics* describe additional semantic limitations of the syntax category and syntax category with such words is equal to that without italicized words (but is its semantic restriction).

Reserved words are printed like **this**.

Optional items are enclosed in square brackets [ ]. Items enclosed in curly brackets { } can be repeated zero or more times.

Square and curly brackets and quotation marks (and sometimes also other symbols) which are part of the syntax are given in quotation marks like this: "[".

Example rule	**keyword** [*Optional* item] {. *Repeated* item }
Item	Variant 1
	Variant 2 | Variant 3
	Variant 4
	"[" Still variant 4 "]"

Summary of all syntax rules is given in appendix C.

Semantics

Both static (compile-time) and dynamic (run-time) semantic rules are described here. Some simple semantic rules (like restrictions of the type of an expression) can be given as part of the syntax rules (see the description of the Syntax section).

In the text we refer to the syntax categories both on the left side and on the right side of the syntax rules Like this. When referring to a syntax category given on the right side of any syntax rule, the occurrences in the corresponding *Syntax* section (not its definition) are usually meant.

Definitions of terms are printed like this. A more readable form of definitions is preferred instead of a formal definition. In such cases the reader is expected to have a general knowledge of the common meaning of such term from other programming languages. Summary of all definitions is the glossary in appendix B.

Note: Notes give comments on the language rules and point out characteristics that are not clearly obvious. Notes are informative.

Language design: Comments on the decisions made during the language design and other language design topics. Language design notes are informative.

Implementation advice: Recommendations given to the implementation to successfully implement a given language construct or rule. Implementation advices are informative.

Implementation Defined

Permissions of any implementation defined characteristics, implementation requirements, advices etc., if any.

Predefined Environment

Parts of the predefined environment.

Examples

Shows usage of described constructs. This section is informative.

1.3 References

Normative References

Following standards are normative for purposes of this Language Reference Manual:

— [IEC 60559] IEC 60559:1989, Binary Floating-point Arithmetic for Microprocessor Systems,
— [ISO 8601] ISO 8601:1988, Data elements and interchange formats — Information interchange — Representation of dates and times,
— [ISO 10646] ISO/IEC 10646-1:1993, Information technology — Universal Multiple-Octet Coded Character Set (UCS),
— [ISO 11578] ISO/IEC 11578:1996 (E), Information technology — Open Systems Interconnection — Remote Procedure Call (RPC), Annex A: Universal Unique Identifier,
— [UNICODE] The Unicode Consortium, The Unicode Standard, Version 3.2.0,
— [UNICODE TR15] The Unicode Consortium, Unicode Technical Report #15: Unicode Normalization Forms,
— [RFC 2119] Network Working Group, RFC 2119: Key words for use in RFCs to Indicate Requirements Levels, March 1997,
— [ECMA 335] ECMA–335, Common Language Infrastructure (CLI) Partitions I to V, December 2001.

Following documents are related to this Language Reference Manual:

— [SMPL] Programming Language Flex — Standard Multiplatform Library Reference Manual, 2002, A && L soft,
— [ENVREC] Recommended Values of Environment Variables, 2002, A && L soft.
— [NCI] Programming Language Flex — Native Compiler Interface, 2002, A && L soft.

Informative References

Following references are given for informative purposes only:

— [AARM] ISO/IEC 8652:1995(E), Information technology — Programming languages — Ada, Annotated Ada Reference Manual, Version 6.0, 21 December 1994,
— [AMPL] Advanced Multiplatform Library Reference Manual, 2002, A && L soft.

1.4 Wording Conventions

The words must, must not, required, shall, shall not, should, should not, recommended, may and optional in this Language Reference Manual are to be interpreted as described in [RFC 2119].

2. Lexical Elements

The text of a program (source code) consists of one or more files. A file is a sequence of lexical elements. Lexical elements are composed of characters. Lexical elements are keywords, identifiers, metaidentifiers, numeric literals, character literals, character aggregates and comments. In some cases adjacent lexical elements have to be separated by Spaces. Source code is divided into lines separated by Line ends. A Line end is treated as a lexical element separator but in the Multiline comment.

2.1 The Alphabet

Syntax

The following alphabet is used for composition of lexical elements:

Letter	a | b | c | d | e | f | g | h | i | j | k | l | m
	n | o | p | q | r | s | t | u | v | w | x | y | z
	A | B | C | D | E | F | G | H | I | J | K | L | M
	N | O | P | Q | R | S | T | U | V | W | X | Y | Z
Digit	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
Special character	' | """" | < | = | > |. |, |; |: | + | - | / | * | _ | (|) | \ | ^ | $ | & | #
	"{" | "}" | "[" | "]"
Space	
Line end	
Displayable character	Letter
	Digit
	Space
	Special character
	Implementation defined

Semantics

The Flex alphabet is case-insensitive. The encoding and graphic representation of Letter, Digit and Special character shall be according to the [ISO 10646] standard.

Special characters are used to either compose Symbols, which are separate lexical elements, or are part of other lexical elements.

Implementation Defined

The encoding and graphic representation of a Space, Line end and implementation defined Displayable character is implementation defined.

Implementation can support more encodings for Space and Line end, e.g. tabulator for Space and page break for Line end.

Structure of source files is implementation defined. The implementation should accept at least plain text files (as defined by the host operating system) in the UTF–8 encoding.

2.2 Keywords

Syntax

Keyword	**abs** | **abstract** | **accept** | **adjust** | **aliased** | **and** | **array** | **attribute** **begin** | **break** **case** | **catch** | **character** | **class** | **commit** | **concurrent** | **const** | **current** **declare** | **delay** | **discard** | **div** | **do** **else** | **elsif** | **end** | **entry** | **enum** | **exit** | **expose** | **extend** **fixed** | **float** | **for** **generic** | **goto** **if** | **in** **label** | **leave** | **logical** | **loop** **macro** | **message** | **mod** | **module** **new** | **nil** | **not** **of** | **or** | **others** | **out** | **overload** | **override** **pred** | **private** | **procedure** | **program** | **protected** | **public** **queue** **raise** | **range** | **record** | **ref** | **return** | **reverse** | **rollback** **scalesend** | **sequential** | **set** | **shl** | **shr** | **signed** | **static** | **string** | **succ** | **supervised** **tag** | **task** | **then** | **this** | **to** | **type** **unchecked** | **unsigned** | **until** | **use** **var** | **vector** | **virtual** **when** | **while** | **with** **xor**

Semantics

Keywords are reserved words that cannot be used as identifiers.

2.3 Symbols

Syntax

Symbol	< | <= | = | >= | > | <> |. |, |; |: := | + | | / | * | (|) | ^ | & | "[" | "]"

Symbols are composed of one or two Special characters and are used to represent operators, separators and form other constructs.

2.4 Identifiers and Metaidentifiers

Syntax

Identifier first letter	Letter _
Identifier next letter	Letter

	Digit
	_
Identifier	Identifier first letter { Identifier next letter }
Metaidentifier	#Identifier

Semantics

The length of an identifier is unlimited.

2.5 Numeric Literals

Syntax

Extended digit	Digit
	Letter
Decimal number	Digit { Digit | _ }
Hexadecimal number	$ Extended digit { Extended digit | _ }
Based number	Decimal number $ Extended digit { Extended digit | _ }
Exponent	E [+] Decimal number
	E - Decimal number
Real number	Decimal number. Decimal number [Exponent]
	Decimal number [. Decimal number] Exponent
Integer number	Decimal number
	Hexadecimal number
	Based number
Numeric literal	Integer number
	Real number

Semantics

An underline character in a Numeric literal does not affect its value.

A Decimal number denotes a number in the conventional decimal notation (that is, the base is ten). A Hexadecimal number denotes a number in the hexadecimal notation (that is, the base is sixteen). A Based number denotes a number with a given base (the value of the Decimal number preceding the "$"). The base shall be at least two and at most thirty-six.

The extended digits A through Z represent the digits ten through thirty-five. The value of each Extended digit in a Hexadecimal number or a Based number shall be less than the base.

2.6 Character Literals and Aggregates

Syntax

Character literal	""" Displayable character """
	Character encoding
Character aggregate	Character encoding Character aggregate element
	{ Character aggregate element }
	String of characters { Character aggregate element }

Character aggregate element	Character encoding
	String of characters
String of characters	' { Character } '
Character	Displayable character *except '*
	' '
Character encoding	\ Integer number

Semantics

The Character encoding denotes [ISO 10646] encoding of a single character.

The length of a Character aggregate is the number of Character and Character encoding elements. The double "''" (apostrophee) character in a Character represents a single apostrophee character in the resulting value.

Language design: A Character aggregate cannot consist only of a single Character encoding, because it would be ambiguous. We decided not to introduce two different syntax constructs for Character encodings that represent a Character literal and that are part of a Character aggregate.

2.7 Comments

Syntax

Multiline comment	"{" { Multiline comment element } "}"
Multiline comment element	Displayable character *except { and }*
	Multiline comment
	Line end
End of line comment	- - { End of line comment element } Line end
End of line comment element	Displayable character

Semantics

Comments have no semantic meaning and have no influence to other language constructs. When an End of line comment without the trailing Line end appears as a last lexical element in a file, then the end of file shall be substituted to the Line end.

2.8 Lines of Text

The length of a source line and the total length of the source code are unlimited.

3. Declarations and Program Structure

3.1 Declarations

The language defines several kinds of entities that are declared by declarations.

In this chapter we describe the general concepts of declarations.

Syntax

Declaration	Module declaration
	Class declaration
	type Type declaration { Type declaration }
	var Variable declaration { Variable declaration }
	const Constant declaration { Constant declaration }
	label Label declaration { Label declaration }
	Subprogram declaration
	Task declaration
	Override declaration
	Overload declaration
	Operator declaration
	Message declaration
	Queue declaration

Semantics

A declaration is a language construct that associates a name with an entity. A declaration may appear explicitly in the program text (an explicit declaration), or may be supposed to occur at a given place in the text as a consequence of the semantics of another construct (an implicit declaration).

Glosasry[Declaration]: A declaration is a language construct that associates a name with an entity. A declaration may appear explicitly in the program text (an explicit declaration), or may be supposed to occur at a given place in the text as a consequence of the semantics of another construct (an implicit declaration).

Note: An implicit declaration contains no specification (see later) — only entities with trivial specifications can be implicitly declared.

For an explicit declaration the name associated with the entity is usually an identifier given in the declaration. The name associated with an implicitly declared entity is to be thought of in terms of the syntactic category Name rather than an Identifier.

Following are declarations:

— Attribute declaration,
— Compilation unit,
— Constant declaration,
— Class declaration,
— Enumeration item declaration,
— Formal parameter declaration,
— Label declaration,
— Message declaration,
— Module declaration,
— Operator declaration,

- Overload declaration,
- Override declaration,
- Program declaration,
- Queue declaration,
- Record item declaration,
- Return parameter declaration,
- Special method declaration,
- Subprogram declaration, that is:
    - Procedure declaration,
    - Static subprogram declaration,
    - Virtual subprogram declaration,
    - Macro declaration,
- Task declaration,
- Type declaration,
- Variable declaration.

Note: Note that Attribute declaration, Enumeration item declaration, Formal parameter declaration, Program declaration, Record item declaration, Return parameter declaration and Special method declaration are not included in the syntax category Declaration.

A declarative region is a part of a language construct that can contain declarations. This region needs not to be continuous and can be divided into several parts. Declarative regions may be nested. All entities declared within a single declarative region must have distinct names. Every declarative region is associated with an entity. An entity can have zero or more (separate) declarative regions, that can be nested one in another.

Declarative regions are classified as primary regions, attribute regions, component regions and enumeration regions. These classifications affect visibility rules and are used for simplicity.

All of the following represent (separate) declarative regions for specific kinds of entities:

Glosasry[Declarative region]: A declarative region is a part of a language construct that can contain declarations. Every declarative region is associated with an eclosing entity declaration (either implicit or explicit) and forms a logical namespace for entities declared immediately within this declarative region. More declarative regions can be associated with a single entity declaration.

- an Attribute specification (attribute region).
- a Compilation unit (primary region),
- a Declare statement (primary region),
- an Enumeration type definition (enumeration region),
- a Full body (primary region),
- a List of formal parameters (component region),
- a List of message parameters (component region),
- a Public body together with the corresponding Private body (primary region),
- a Record type definition (component region),
- a Subprogram body (primary region),

Each of the above except an Attribute specification and Declare statement forms a main declarative region of the associated entity.

No two entities declared within a single declarative region shall have the same name (identifier). The names (identifiers) of all entities declared within an enumeration region shall be treated also as if present in the innermost enclosing primary region (that is to be directly visible in that region), but the declaration shall stay only as part of the corresponding enumeration region.

Note: This means that enumeration items belong to two declarative regions — (1) to the declarative region of the corresponding Enumeration type definition and (2) to the declarative region to which the innermost enclosing explicitly declared entity other than a record item belongs.

Every (explicit) declaration contains a specification, which defines all properties of an entity (such as its name, type, formal parameters etc.), that are needed for proper usage of that entity. A specification containing only the entity's name (Identifier) is a trivial specification.

Glosasry[Specification]: A specification is a part of every (explicit) declaration which defines basic properties of an entity (such as its name, type, formal parameters etc.) that are needed for proper usage of that entity. A specification containing only the entity's name (Identifier) is a trivial specification. Syntactically a specification forms the first part of a declaration up to a "=" or end of declaration.

Note: Note that if there are only optional constructs (except the name, however) in a specification of some kind of entity and a particular declaration of such an entity contains none of these optional parts, its specification is not trivial.

A definition of an entity defines all other properties (such as subcomponents, statement part, etc.). Syntactically a definition follows the corresponding specification and is separated by a "=".

Glosasry[definition]: A definition of an entity defines the all other properties — internals of that entity — (such as subcomponents, statement part, etc.) that are not defined by the entity's specification. Syntactically a definition follows (if present in a declaration) the corresponding specification and is separated by a "=".

Note: We use this vague definition of a specification and a definition, because of its high dependence on the entity's kind. For example a type is needed to properly declare a variable, while a type definition is not necessary to declare a type.

The syntax rule for recognition between an entity's specification and definition immediately resolves any confusions for each kind of entity.

A definition of an entity can be separated from its specification, and other entities can be declared in between. In this case a declaration is split in an incomplete declaration and a completion of the declaration, if both parts are presented in a single declaration it is a full declaration. If an entity is declared using an incomplete declaration then corresponding completion must follow anywhere in the same declarative region.

For entities with trivial specifications the declaration completion and a full declaration are syntactically the same. In this case no special syntax rule for the completion is defined and the full declaration is used instead.

A declaration of an entity, possibly incomplete, must always precede the first usage of that entity.

Examples

Explicit and implicit declarations:

```
-- variables i and j are explicitly declared
var i: unsigned 32;          -- a variable declaration with an implicit type
                              -- declaration
var j: i:type;               -- i:type is a name of an implicitly declared type
```

Specification and definition of a declaration:

```
procedure p (i: integer) = -- <-- specification
var j: integer;            -- <-- definition (with subcomponent declaration)
begin                       -- <-- definition
  end p;                    -- <-- definition

module m =                  -- an entity with a trivial specification
end m;
```

Incomplete declarations and their completions:

```
procedure p (i: integer);  -- incomplete declaration
-- possible declarations of other entities here
procedure p =               -- completion of the declaration (different from
begin                       -- a full declaration)
  end p;

module m;
module m =                  -- completion is the same as a full declaration
end m;
```

Declarative regions:

```
var i: signed 32;
var i: unsigned 16;         -- illegal, i is alredy declared in the same region

type
  r = record
      -- enumeration items a,b,c will be included in the same declarative
      -- region as r
      x: array 1..10,1..10 of enum a;b;c; end enum;

      -- legal, a is not contained within the declarative region of this
      -- record definition
      a: integer;
      end record

      -- legal, r.x and r:x are not in the same declarative region
      attribute x:= 10;

var a: r;                   -- illegal, a is already declared (as an
                             -- enumeration item) in this declarative region
```

3.2 Visibility Rules

Semantics

Visibility rules for attributes (entities declared within attribute regions) are discussed in chapter 6.2.6. For all other declarative regions the visibility rules are as follows.

A visibility level defines the basic rules of visibility of every entity declared within a declarative region. Three visibility levels exist:

— public,
— protected,
— private.

Entities declared as public inside an enclosing entity are visible from all other entities referencing the enclosing entity. Protected entities are visible in the context of the enclosing entity and all derived entities (either directly or transitively). Private entities are visible only in the context of the enclosing entity.

All entities declared inside a Subprogram body or a Private body are treated as private, in all other declarative regions the default visibility is public.

An entity is directly visible at a given place (place of reference), if and only if:

— it is declared within the current declarative region (where the place of reference occurs), from its beginning up to the place of reference, or in any of the enclosing regions up to the main declarative region of the innermost enclosing entity,
— if the innermost enclosing entity has an ancestor, it is declared as public within its main declarative region or recursively within its ancestor,
— if the innermost enclosing entity contains a With clause, and it is declared as public within one of the **with**ed modules and classes (in order of appearance in the With clause) or within their ancestors (recursively),

Note: If the innermost enclosing entity contains more With clauses (e.g. one within a Public body and one within a Private body), they are searched in order of appearance in the declaration.

However, if a name within a With clause denotes an incomplete declaration of a module or a class, then no entity is visible.

— otherwise if it is directly visible from a place of reference (and declarative region) corresponding to the placement of the current declarative construct, that represents the innermost enclosing entity.
  Note: E.g. either an incomplete declaration, declaration completion (or a Public body or a Private body) or a full declaration.

Implementation advice: An application of these rules is a recursive search algorithm for a Direct name resolution. To resolve an identifier, this algorithm first searches through nested declarative regions up to the main declarative region of the current entity, then tries to recursively find it in its ancestor (if any), then in all **with**ed modules and classed and finally, if still not found, applies this algorithm to the entity superior to the current entity.

Examples

```
module a1 =
  var i: integer;             -- I has visibility level public
protected
  var j: integer;             -- J has visibility level protected
private
  var k: integer;             -- K has visibility level private
  end a1;

module a2 =
  extend a1;                   -- A2 is derived from A1
  procedure p1 =
  var x: integer;             -- X has visibility level private
  begin
    i:=j;                      -- both I and J are visible here
    k:=i;                      -- illegal, K is invisible
    end p1;
  procedure p2 =
  begin
    p1.x:=0;                   -- illegal: 1) X is not visible here, 2) X does
                               -- not exist in this context

    end p2;
  end a2;

module b =
  with a1;
  procedure p =
  begin
    i:=0;                      -- I is visible, because A1 was with'ed
    j:=0;                      -- illegal, J is not visible here
    end p;
  end b;
```

3.3 Program Structure and Execution

Syntax

Compilation unit	{ Declaration } [Program declaration]
Program declaration	**program** Identifier [Attribute specification] = Subprogram body;

Semantics

A compilation unit is a set of related entities, that form an executable program. The declaration of a compilation unit can optinally contain a Program declaration, that declares the main entity of the program simply called also a program. The words program entity are used to refer to an actual Program declaration when needed.

Compilation of a compilation unit that contains a Program declaration produces an executable program. The compilation of a compilation unit without a Program declaration produces a component library. In most cases, both of these refered to as programs in this Language Reference Manual.

This Language Reference Manual especially does not specify the following regarding executable programs and component libraries:

— the form of the resulting program,
— the way how a program is produced from a source text,
— the actions that need to be taken to prepare the program for execution (e.g. loading an executable image, JIT compiler invokation etc.)
— the interfaces a program offers to other programs and other means of interoperability,
— the way how these interfaces are declared or marked within the source text,
— the way how a component library can be used.

The execution of a program is the process of applying the run-time effects declared in the source text of the program. The combination of software and hardware elements that executes programs is called an execution environment (e.g. an operating system, component manager, virtual machine etc.).

3.3.1 Executable Program Execution

Semantics

The execution of an executable program consists of the following consecutive actions:

— the main task is created by the execution environment and control is transfered to the compilation unit,
— all static entities are initialized in order of appearance in the compilation unit and recursively through all entities, which involves:
  — calling **entry** special methods of all modules,
  — initializing all static and task instances of classes,
  — setting initial values to static and task variables,
— the program entity is called,
— all static entities are finalized in reverse order appearance in the compilation unit and recursively through all entities, which involves:
  — calling **exit** special methods of all modules,
  — finalizing all static and task instances of classes,
— the control is transfered back to the execution environment.

The way how task variables are handled (time of initialization and finalization for each task that uses the component library) in component libraries is implementation defined. The implementation may limit or even prohibit declaration and usage of task variables and optionally also tasks in component libraries.

Implementation advice: It is recommended not to prohibit declarations of task variables, but rather turn them into static variables if needed. This increases compatibility and reuse of source text originally used in executable programs.

3.3.2 Component Library Execution

Semantics

The execution of a component library consists of the following consecutive actions:

- control is transfered to the compilation unit in context of an arbitrary task,
- all static entities are initialized in order of appearance in the compilation unit and recursively through all entities, which involves:
    - calling **entry** special methods of all modules,
    - initializing all static instances of classes,
    - setting initial values to static variables,
- control is transfered back to the execution environment to allow execution of the component library in an implementation defined way; after this execution is finished, control is transfered to the compilation unit in context of an arbitrary task (not necessarily the same task as for initialization) and finalization begins,
- all static entities are finalized in reverse order appearance in the compilation unit and recursively through all entities, which involves:
    - calling **exit** special methods of all modules,
    - finalizing all static instances of classes,
- the control is transfered back to the execution environment.

3.3.3 Classification of Errors

Semantics

The language defines three categories of errors:

- compile-time errors,
- execution errors,
- run-time errors.

Compile-time errors are caused by any violation of the language rules, that can be enforced by the compiler during the compilation.

Execution errors are detected during the execution of the program and correspond to those language rules, that cannot be enforced during the compilation (run-time checks). When such an error is detected, one of the language defined messages shall be raised, unless the appropriate message is not declared or otherwise not understood by the compiler (e.g. there was no binding created using the pragma `Assign` — see chapter ???) in which case an execution error shall be treated as a run-time error. An execution error may result in partially erroneous execution of the program (e.g. when a check fails during the execution of a statement, some parts of the statement may not be executed and so leading to erroneous execution).

Run-time errors are caused by the execution environment and its effect is in general not predictable and leads to erroneous execution.

3.4 Modules and Classes

Syntax

Module declaration	Incomplete module declaration
	Module declaration completion
	Full module declaration
	Public module declaration
	Private module declaration

Incomplete module declaration	**module** Identifier [Attribute specification];
Full module declaration	**module** Identifier [Attribute specification] = Full body;
Module declaration completion	**module** Identifier = Full body;
	module public Identifier = Public body;
Public module declaration	**modulepublic** Identifier [Attribute specification] = Public body;
Private module declaration	**module private** Identifier = Private body;
Class declaration	Incomplete class declaration
	Class declaration completion
	Full class declaration
	Public class declaration
	Private class declaration
Incomplete class declaration	**class** [**abstract**] Identifier [Attribute specification];
Class declaration completion	**class** [**abstract**] Identifier = Full body;
	class [**abstract**] **public** Identifier = Public body;
Full class declaration	**class** [**abstract**] Identifier [Attribute specification] = Full body;
Public class declaration	**class** [**abstract**] **public** Identifier [Attribute specification] = Public body;
Private class declaration	**class** [**abstract**] **private** Identifier = Private body;
Full body	[Ancestor specification] [Interface specification] [With clause] [Use clause]
	{ Declaration }
	[**protected** { Declaration }]
	[**supervised** { Declaration }]
	[**private** { Declaration }]
	{ Special method declaration }
	end Identifier
Public body	[Ancestor specification] [Interface specification] [With clause] [Use clause]
	{ Declaration }
	[**protected** { Declaration }]
	[**supervised** { Declaration }]
	end Identifier
Private body	[With clause] [Use clause]
	{ Declaration }
	{ Special method declaration }
	end Identifier
Module ancestor specification	**extend** *Module* name;
Class ancestor specification	**extend** *Class* name;
Class interface specification	**expose** *Abstract class* name {, *Abstract class* name };
With clause	**with** Used entity reference {, Used entity reference };
Use clause	**use** Used entity reference {, Used entity reference };
Used entity reference	[**supervised**] *Module or class* name

Semantics

All declarations after the keyword **protected** inside a Full body or a Public body have the visibility level protected. All declarations after the keyword **supervised** inside a Full body or a Public body have the visibility level supervised. All declarations after the keyword **private** inside a Full body and all declarations inside a Private body have the visibility level private.

```
module flex =
private
  Implementation Defined
  end flex;

module system =
  Implementation Defined
  end system;
```

The module `Flex` is intended as a location of the run-time environment support routines. The module `System` shall define the interface to the target operating system.

The module `Flex` shall not **extend**, **with** or **use** any other modules or classes.

Implementation advice: If any part of the interface to the operating system is needed, it shall be redeclared rather than **with**ing the module `System`.

3.4.1 Special Methods

Syntax

Special method declaration	Special method identifier = Subprogram body **end** Special method identifier;
Special method identifier	**entry** **exit** **adjust** **commit** **rollback**

4. Types

Semantics

A regular type is an abstract entity that defines a set of values and a set of operations on these values. A singular type is an abstract entity that defines a group of entities.

Three basic kinds of regular types exist: elementary types, composite types and classes. Elementary types represent values that are logically indivisible while composite types are composed of components that can be referenced separately. Classes share properties of composite types and modules and thus are declared by a special declaration different from all other types (see chapter 3.4).

Elementary types, composite types with definite number of components and classes are constrained, composite types with indefinite number of components (unconstrained array and unconstrained string) are unconstrained.

Types can be derived from each other to form a type hierarchy. This hierarchy is based on simple inheritance — a single type can have at most one ancestor. A derived type is a descendant of its ancestor. A root type has no ancestor.

A universal type is a formal type that is used to describe the type of certain expressions. Following universal types exist: *universal integer*, *universal real*, *universal logical*, *universal character*, *universal array* with a given base type (here the base type is the type of the array's components) and *universal NIL*.

A universal type cannot be explicitly declared and cannot be an ancestor of another type.

4.1 Type Declarations

Syntax

Type declaration	Incomplete type declaration
	Full type declaration
	Private extension declaration
Incomplete type declaration	Identifier;
Full type declaration	Identifier = [Type compatibility determination] Type
	[Attribute specification] [:= *Constant* expression];
Type compatibility determination	**protected**
	private
Type	Type definition
	Type name [Derivation specification]
Type definition	Integer type definition
	Enumeration type definition
	Logical type definition
	Character type definition
	Implicit discrete type definition
	Real type definition
	Pointer type definition
	Array type definition
	Unconstrained array type definition
	String type definition
	Unconstrained string type definition
	Record definition

	Set type definition Subprogram type definition Message type definition Queue type definition Tag type definition Generic type definition

Semantics

A type declared by a Type declaration or a Class declaration is a named type. Named types are tagged, while unnamed (implicitly declared) types are untagged.

Note: An untagged type is always a root type. A tagged type may have an untagged ancestor.

If a type is declared with an Incomplete type declaration, its declaration shall be completed with a Type declaration completion of a regular type, if it was used as a:

— type of a constant in an Incomplete constant declaration,
— type of a formal parameter in an Incomplete subprogram declaration,
— base type in a Pointer to object type definition.

Language design: For an Incomplete constant declaration, the Incomplete type declaration shall be completed with a Type declaration completion of a constrained type, but this rule would be unnecessarily strong – this particular check can be performed within the Constant declaration completion.

4.2 Views of a Type

Semantics

A view of a type specifies properties of a type in the given location. A view of a type can be:

— an incomplete view if only an Incomplete type declaration exists so far in the type's declarative region,
— a partial view if only the Incomplete type declaration is visible in the given location and the corresponding Type declaration completion exists in an invisible part of the type's declarative region,
— a full view if the whole declaration of the given type is visible in the given location,
— an unchecked view is used, where a partial or full view would be normally used by default, but where the structure of the type is explicitly forced to be unknown; also the view of an untyped object (e.g. a view of a type of a type-less — **unchecked** — parameter) is an unchecked view.

Structure of a type is visible in the given location, only if the view of the type in that location is a full view. Otherwise the structure of the type is hidden. In an incomplete view no properties of a type can be determined. When an incomplete view of a type is referenced in an entity declaration, that type is assumed to be regular and its declaration must be completed with a regular Type declaration completion.

An Incomplete type declaration cannot be finished with a singular or unconstrained Type declaration completion, if a partial view of the type can exist.

Note: This means, that if a singular or unconstrained type is to be declared, its Incomplete type declaration and Type declaration completion cannot be placed in different visibility areas of the enclosing declarative region.

4.3 Type Derivation

Syntax

Derivation specification	Type constraint

	Type extension

Semantics

A type can be defined by specifying the name of a parent type and optionally a Derivation specification. A Derivation specification can be either a Type constraint, constraining the bounds (or other properties) of the parent type, or a Type extension, adding properties to the parent type. If no Derivation specification is specified, the properties of the defined type are as of the parent type.

4.3.1 Type Constraints

Syntax

Type constraint	Discrete range constraint
	Array range constraint
	String length constraint
Discrete range constraint	**range** *Constant* range
Array range constraint	**range** *Constant* range
String length constraint	**range** *Constant* expression

Semantics

For a Discrete range constraint, the parent type shall be any discrete type other than a modular integer. The type of the Constant range shall be compatible with the parent type. The Constant range shall not denote a null range and its bounds shall be within the range of the parent type.

A type defined with a Discrete range constraint is derived from the parent type and its range is that of the Constant range.

The size of the defined type remains unchanged and is that of the parent type.

For an Array range constraint, the parent type shall be any (constrained or unconstrained) array type. The type of the Constant range shall be compatible with the parent type's range type. The Constant range shall not denote a null range and its bounds shall be within the range of the parent type's range type.

A type defined with an Array range constraint is derived from the parent type and its range type is derived from the parent type's range type having a range of the Constant range.

The size of the defined type is computed from the given range.

For a String length constraint, the parent type shall be any (constrained or unconstrained) string type. The type of the Constant expression shall be compatible with the parent type's range type. The value of the Constant expression shall be within the range of the parent type's range type.

A type defined with a String length constraint is derived from the parent type and its range type is derived from the parent type's range type having the upper bound of the Constant expression.

The size of the defined type is computed from the given upper bound.

4.3.2 Type Extensions

Syntax

Type extension	**with** *Regular* type definition
	with private
Private extension declaration	Identifier = *Regular* type definition [:= *Constant* expression];

Semantics

A type can be extended only by a regular type definition. The type extension declared with the keywords **with private** is a private extension. The type extension declared with the single keyword **with** is a public extension.

A declaration of a type with a private extension is an incomplete declaration and needs to be completed with a Private extension declaration. The incomplete Full type declaration and the corresponding Private extension declaration shall be placed in different visibility areas of the enclosing declarative region.

Note: This implies that a partial view of the type can exist, which is exactly why private extensions exist.

Language design: Here we have a little confusion in incomplete declarations. An incomplete declaration is said to be just the specification part of a declaration, while here it is a full declaration with the exception of one part of its definition, which is split into a separate syntax construct that plays role of the completion. This is due to the nature of private extensions — we need the parent type to be specified early, while the extension part to be specified in a place, where it can be hidden. The natural way of making things work here is to break the rule. Nevertheless we believe that the syntax and semantics are clear.

For a parent type whose view is partial, the extension can be either a public extension or a private extension. For a parent type whose view is full and that is a record, the extension must be a public extension by a Record type definition. This specific public extension is a record extension. No other types can be parent types of a type extension.

The Constant expression within the Full type declaration for a public extension and within the Private extension declaration for a private extension specifies the initial value of the extension part of the type (even if it is a record extension). For a private extension no initial value shall be specified within the Full type declaration.

The size of the resulting type is based on the size of the parent type and the size of the extension part.

4.4 Elementary Types

4.4.1 Scalar Types

4.4.1.1 Discrete Types

Semantics

A discrete type defines a discrete range of values. Every discrete value has its distinct ordinal value, which is of type *universal integer*. Values of a discrete type are ordered in the notion of their ordinal values' natural ordering.

4.4.1.1.1 Integer Types

Syntax

Integer type definition	[**mod**] **signed** *Constant* expression
	[**mod**] **unsigned** *Constant* expression

Semantics

An Integer type definition defines either signed or unsigned integers. The Constant expression of an Integer type definition shall be of any integer type and its value shall be greater than zero; it specifies the size, in bits, of the defined integer type. If the **mod** keyword is specified, a modular integer type is created.

Values of an integer type shall be encoded in the 2's complement code. The range of a signed integer is $-2^{bits\text{-}1}$.. $2^{bits\text{-}1}$-1. The range of an unsigned integer is 0.. 2^{bits}-1.

For a modular integer type, the result of a predefined arithmetic operator is computed in a modulo 2^{bits} arithmetic. For a non-modular integer type, `Constraint_Error` is raised if the result of a predefined operator is outside the range of the type.

The ordinal value of an integer value is the value itself.

Implementation Defined

The set of allowed bit sizes is implementation defined. If the implementation does not support the requested bit size, it shall raise a compile-time error.

Note: This means that no rounding to the nearest greater allowed size or any other substitution for the requested bit size shall be performed. An Integer type definition allows the specification of a precise machine number type. However, the implementation is permitted to support any other integer types not directly supported by the target machine.

Predefined Environment

The predefined environment declares basic integer types defined as follows:

```
type
  integer = signed Implementation Defined;
  natural = unsigned Implementation Defined;
```

Implementation Defined

The size of the predefined integer types `Integer` and `Natural` are implementation defined. `Integer` and `Natural` should have the same size of at least 16 bits.

4.4.1.1.2 Enumeration Types

Syntax

Enumeration type definition	**enum** { Enumeration item declaration } **end enum**
Enumeration item declaration	Identifier [Attribute specification];

Semantics

An Enumeration type definition defines an enumeration type, having the values of the enumeration items. The size, in bits, of the defined type shall be the least allowed sufficient size to represent ordinal values of all enumeration items.

The ordinal value of the first declared enumeration item is zero. The ordinal value of any subsequent enumeration item is one more than that of its predecessor.

4.4.1.1.3 Logical Types

Syntax

Logical type definition	**logical** *Constant* expression

Semantics

A Logical type definition defines a logical type, having the values `True` and `False`. The Constant expression of a Logical type definition shall be of any integer type and its value shall be greater than zero; it specifies the size, in bits, of the defined logical type.

The ordinal value of `False` is zero and the ordinal value of `True` is one.

Language design: The language does not define any names for the values of a given logical type. The attributes `:True` and `:False` shall be used instead.

Predefined Environment

The predefined environment declares the logical type `Boolean` and constants `True` and `False` (of type *universal logical*).

```
type
  boolean = logical Implementation Defined;
```

```
    const
      true     = 0=0;
      false    = 0<>0;
```

Note: The expression 0=0 yields a value of type *universal logical*.

4.4.1.1.4 Character Types

Syntax

Character type definition	**character** *Constant* expression

Semantics

A Character type definition defines a character type. The Constant expression of a Character type definition shall be of any integer type and its value shall be greater than zero; it specifies the size, in bits, of the defined character type.

The values of a character type are Character literals with ordinal values within the range 0.. $2^{bits\text{-}1}$. The ordinal value of a Character is the encoding corresponding to the given graphical representation according to [ISO 10646]. The ordinal value of a Character encoding is the encoding itself.

Predefined Environment

The predefined environment declares a character type `Char`.

```
    type
      char = character Implementation Defined;
```

Implementation Defined

The size of the predefined type Char is implementation defined. It should be 32 in order to enclose all [ISO 10646] characters.

4.4.1.1.5 Implicit Discrete Type Definition

Syntax

Implicit discrete type definition	**range** *Constant* range

Semantics

An Implicit discrete type definition defines a discrete type by specifying its range. The defined type is either signed, unsigned, logical or character. The Constant range shall not denote a null range.

The values of the type include all values within the Constant range. The size, in bits, of the defined type shall be the least allowed sufficient size to represent all values within the Constant range.

4.4.1.2 Real Types

4.4.1.2.1 Floating-point Types

Syntax

Floating-point type definition	**float** *Constant integer* expression

Predefined Environment

The predefined environment declares a floating-point type `Real`.

```
type
  real = digits Implementation Defined;
```

4.4.1.2.2 Fixed-point Types

Syntax

Floating-point type definition	**fixed** *Constant integer* expression **scale** Precision specifications
Precision specifications	*Constant real* expression
	mod *Constant integer* expression

4.4.2 Pointer Types

Syntax

Pointer base class	**class**
	unchecked

Semantics

A pointer represents an indirect handle to a run-time representation of an entity (either staticaly declared or dynamicaly created). The value of an object of a pointer type holds information needed to provide such indirect access to other entity. There are several kinds of pointers distinguished by the type of the target object. Pointers can denote:

- objects,
- subprograms,
- tasks,
- messages,
- queues.

Language design: In Flex we have generalized pointers as means of indirect access to various kinds of language defined run-time entities. We do not feel that pointers — when thought of abstractly enough — are something low-level or insecure, that has no place in a modern programming language and should be hidden among other implementation details. A pointer in Flex is a serious instrument designed to provide explicitly indirect access to run-time entities, that the user cannot simply access in any other way.

The Pointer base class determines compatibility of different pointer types and other properties of a pointer and is one of:

- strict,
- **class**,
- **unchecked**.

The base class of any other pointer than a pointer-to-object is always strict. When not specified, the default base class is strict.

4.4.2.1 Pointer to Object

Syntax

Pointer *to object* type definition	^ [Pointer base class] [Object access determination] Type
	^ **unchecked** [Object access determination]

Semantics

A Pointer *to object* type definition defines a pointer type that provides a handle to an object.

The Type shall denote either a regular type or an incomplete type. When an incomplete type is specified, its declaration shall be completed with a Type declaration completion of a regular type. When the **class** Pointer base class is specified, the Type shall be a tagged type.

Note: Note that an incomplete type is always a tagged type.

4.4.2.2 Pointer to Subprogram

Syntax

Pointer *to subprogram* type definition	^ *Subprogram* type

4.4.2.3 Pointer to Task

Syntax

Pointer *to task* type definition	^ *Task* type

4.4.2.4 Pointer to Message

Syntax

Pointer *to message* type definition	^ *Message* type

4.4.2.5 Pointer to Queue

Syntax

Pointer *to queue* type definition	^*Queue* type

4.4.3 Special Types

4.4.3.1 Subprogram Types

Syntax

Subprogram type definition	**procedure** [List of formal parameters] [Return parameter declaration]

4.4.3.2 Task Types

See chapter 9.1

4.4.3.3 Message Types

See chapter 10.1

4.4.3.4 Queue Types

See chapter 10.2.

4.4.3.5 Tag Types

Syntax

Tag type definition	**tag**

4.4.3.6 Generic Types

See chapter 14.1

4.5 Composite Types

4.5.1 Array Types

Syntax

Array type definition	[**vector**] **array** *Constant* range {, *Constant* range } **of** *Constrained* type
Unconstrained array type definition	[**vector**] **array of** *Constrained* type

4.5.2 String Types

Syntax

String type definition	**string** *Constant* expression **of** *Constrained* type
Unconstrained string type definition	**string of** *Constrained* type

4.5.3 Record Types

Syntax

Record type definition	[**vector**] **record** { Record item } **end record**
Record item	Record item declaration
	with *Record type* name {, *Record type* name };
Record item declaration	Identifier: [Object access determination] *Constrained* type [Attribute specification]

4.5.4 Set Types

Syntax

Set type definition	**set of** *Discrete* type

4.6 Classes

See chapter 3.4.

4.7 Compatibility of Types

Semantics

There are several compatibility classes, that define compatibility requirements used in different language constructs to determine whether two types are compatible or not in the context of a certain operation. These classes correspond to kinds of operations performed upon entities of these types. Following subchapters describe these compatibility classes, that are then referenced from other places of this Language Reference Manual.

When applicable, compatibility classes also define the type of the resulting value that is yielded by evaluation of the corresponding construct.

Following information is required to perform a type check:

- types T1 and T2,
- views of T1 and T2,
- kinds of T1 and T2 (e.g. signed, enumeration, record, etc.).

The result of a type check is:

- the type of the resulting value yielded by the corresponding construct (if applicable),
- the level of compatibility which can be:
  - incompatible, when the types are incompatible,
  - convertible, when the value of T1 or T2 (depending on the semantics of the operation) can be converted to the value of T2 or T1 respectively,
  - dynamically compatible, when the types are compatible and a dynamic check is required to ensure compatibility on run-time,
  - statically compatible, when the types are compatible and no dynamic check is needed to ensure this,
  - identical, when T1 and T2 are identical types (denote the same type entity).

Elementary compatibility checks perform the actual compatibility check for types T1 and T2, whose compatibility was predetermined based on its views and/or kinds of types. Compatibility classes define which elementary compatibility checks shall be used for specific combinations of views and/or kinds of types. For unspecified combinations the compatibility level shall be *incompatible*.

4.7.1 Elementary Compatibility Checks

Semantics

Following elementary compatibility checks exist. These checks shall be used only under the conditions defined by compatibility classes (e.g. only for particular combinations of views and/or kinds of types).

- **Arithmetic Check.** For two identical types the resulting compatibility level is *identical*. For T1 and T2 that are both either modular or non-modular, the resulting compatibility level is *convertible*. Otherwise the type are *incompatible*.
- **Component Concatenation Check.** For two identical types the resulting compatibility level is *identical*. Otherwise the resulting compatibility level is *incompatible*.
- **Constrained Array Check.** For two identical types the resulting compatibility level is *identical*. Otherwise the resulting compatibility level is *incompatible*.
- **Convertibility Check.** For two identical types the resulting compatibility level is *identical*. Otherwise the resulting compatibility level is *convertible*.
- **General Base Check.** For two identical types the resulting compatibility level is *identical*. For two types with identical base types the resulting compatibility level is *statically compatible*. Otherwise the resulting compatibility level is *incompatible*.

- **Inheritance Check.** For two identical types the resulting compatibility level is *identical*. When T1 is an ancestor T2 or vice-versa, the resulting compatibility level is *statically compatible*. Otherwise the resulting compatibility level is *incompatible*.
- **Left-side Component Concatenation Check.**
- **NIL check.** The resulting compatibility level is *statically compatible*.
- **Pointer Assignment Check.** For two identical types the resulting compatibility level is *identical*. When T1 and T2 denote different kinds of entities (e.g. T1 is a pointer-to-object and T2 is a pointer-to-task), the resulting compatibility level is *incompatible*. Otherwise one of the following applies:

#	Base class of T1	Base class of T2	Condition
PA1	strict	strict	When the base types are identical, the resulting compatibility level is *statically compatible*, otherwise it is *incompatible*.
PA2	strict	**class**	When the base type of T1 is identical or an ancestor of the base type of T2, the resulting compatibility level is *statically compatible*, otherwise it is *incompatible*.
PA3	strict	**unchecked**	The resulting compatibility level is *statically compatible*.
PA4	**class**	strict	When the base type of T1 is identical or an ancestor of the base type of T2, the resulting compatibility level is *statically compatible*, otherwise it is *incompatible*.
PA5	**class**	**class**	When the base types of T1 and T2 are identical, the resulting compatibility level is *statically compatible*. When the base type of T1 is an acestor of the base type of T2 or vice-versa, the resulting compatibility level *dynamicall compatible*. Otherwise the resulting compatibility level is *incompatible*.
PA6	**class**	**unchecked**	The resulting compatibility level is *incompatible*.
PA7	**unchecked**	strict	The resulting compatibility level is *statically compatible*.
PA8	**unchecked**	**class**	The resulting compatibility level is *convertible*.
PA9	**unchecked**	**unchecked**	The resulting compatibility level is *statically compatible*.

- **Pointer Comparison Check.**

#	Base class of T1	Base class of T2	Condition
PA1	strict	strict	
PA2	strict	**class**	
PA3	strict	**unchecked**	
PA4	**class**	strict	
PA5	**class**	**class**	
PA6	**class**	**unchecked**	
PA7	**unchecked**	strict	
PA8	**unchecked**	**class**	
PA9	**unchecked**	**unchecked**	

- **Pointer Reference Check.**

#	Base class of T1	Base class of T2	Condition
PA1	strict	strict	
PA2	strict	**class**	
PA3	strict	**unchecked**	
PA4	**class**	strict	
PA5	**class**	**class**	
PA6	**class**	**unchecked**	
PA7	**unchecked**	strict	
PA8	**unchecked**	**class**	

#	Base class of T1	Base class of T2	Condition
PA9	**unchecked**	**unchecked**	

- **Right-side Component Concatenation Check.**
- **Simple Check.** For two identical types the resulting compatibility level is *identical*. For T1 that is an ancestor of T2, the resulting compatibility level is *statically compatible*. For T1 that is a descendant of T2, the resulting compatibility level is *dynamicall compatible*. Otherwise the types are *convertible*.
- **Strict Inheritance Check.** For two identical types the resulting compatibility level is *identical*. For T1 that is an ancestor of T2, the resulting compatibility level is *statically compatible*. Otherwise the resulting compatibility level is *incompatible*.
- **String Ordering Check.**
- **Unconstrained Array Check.** For two types with identical base types the resulting compatibility level is *statically compatible*. Otherwise the resulting compatibility level is *incompatible*.
- **Universal Array Check.** One of the following applies:
  - When at least one of T1 or T2 has a universal base type, these base types have to be compatible in the Symmetric Conversion Compatibility Class for the resulting compatibility level to be *convertible*, otherwise it is *incompatible*.
  - Whe both T1 and T2 have non-universal base types (that is the base types are regular), these base types have to be identical for the resulting compatibility level to be *convertible*, otherwise it is *incompatible*.

4.7.2 Assignment Compatibility Class

Semantics

For partial views of T1 and T2, Partial Check shall be performed. For partial view of T1 and unchecked view of T2,.... For unchecked view if T1, the size of T2 shall be as of T1's. For full views of T1 and T2, the checks are as follows:

#	T1	T2	Check
AC1	any signed type	any signed type	Arithmetic Check
AC2	any unsigned type	any unsigned type	Arithmetic Check
AC3	any integer type	*universal integer*	Convertibility Check
AC4	any enumeration type	any enumeration type	Simple Check
AC5	any logical type	any logical type	Simple Check
		universal logical	Convertibility Check
AC6	any character type	any character type	Simple Check
		universal character	Convertibility Check
AC7	any real type	any real type	Simple Check
		universal real	Convertibility Check
AC8	any pointer type	any pointer type	Pointer Assignment Check
AC9	any tag type	any tag type	Simple Check
AC10	any constrained array type	any constrained array type	Constrained Array Check
		any unconstrained array type	General Array Check
		any *universal array* type	Universal Array Check
AC11	any unconstrained array type	any array type	Unconstrained Array Check
		any *universal array* type	Universal Array Check
AC12	any string type	any string type	General Base Check
		any *universal array* type	Universal Array Check
AC13	any record type	any record type	???
AC14	any set type	any set type	General Base Check
AC15	any class type	any class type	???
AC16	any regular type	*universal NIL*	NIL check

4.7.3 Value Transformation Compatibility Class

Semantics

For full views of T1 and T2, the checks are the same as in the Assignment Compatibility Class with following additions:

#	T1	T2	Check
VT1	*universal integer*	*universal integer*	Convertibility Check
VT2	*universal real*	*universal real*	Convertibility Check
VT3	*universal logical*	*universal logical*	Convertibility Check
VT4	*universal character*	*universal character*	Convertibility Check
VT5	*universal array*	*universal array*	Universal Array Check

4.7.4 Typecast Compatibility Class

Semantics

For full views of T1 and T2, the checks are the same as in the Value Transformation Compatibility Class with following additions (TC3, TC4 and TC5) and changes (TC1 and TC2):

#	T1	T2	Check
TC1	any signed type	any signed type	Convertibility Check
TC2	any unsigned type	any unsigned type	Convertibility Check
TC3	any signed type	any unsigned type	Convertibility Check
TC4	any unsigned type	any signed type	Convertibility Check
TC5	any real type	any integer type	Convertibility Check
		universal integer	Convertibility Check

4.7.5 Symmetric Conversion Compatibility Class

Semantics

For full views of T1 and T2, the checks are as follows:

#	T1	T2	Check
SC1	*universal integer*	*universal integer*	Convertibility Check
SC2	*universal integer*	any integer type	Convertibility Check
SC3	any integer type	*universal integer*	Convertibility Check
SC4	*universal character*	*universal character*	Convertibility Check
SC5	*universal character*	any character type	Convertibility Check
SC6	any character type	*universal character*	Convertibility Check
SC7	*universal logical*	*universal logical*	Convertibility Check
SC8	*universal logical*	any logical type	Convertibility Check
SC9	any logical type	*universal logical*	Convertibility Check
SC10	*universal real*	*universal real*	Convertibility Check
SC11	*universal real*	any real type	Convertibility Check
SC12	any real type	*universal real*	Convertibility Check

4.7.6 Object Reference Compatibility Class

Semantics

For full views of T1 and T2, the checks are as follows:

#	T1	T2	Check
OR1	any signed type	any signed type	Strict Inheritance Check
OR2	any unsigned type	any unsigned type	Strict Inheritance Check
OR3	any enumeration type	any enumeration type	Strict Inheritance Check
OR4	any logical type	any logical type	Strict Inheritance Check
OR5	any character type	any character type	Strict Inheritance Check
OR6	any real type	any real type	Strict Inheritance Check
OR7	any pointer type	any pointer type	Pointer Reference Check
OR8	any tag type	any tag type	Strict Inheritance Check
OR9	any constrained array type	any constrained array type	Constrained Array Check???
OR10	any unconstrained array type	any array type	???
OR11	any constrained string type	any constrained string type	???
OR12	any unconstrained string type	any string type	???
OR13	any record type	any record type	Strict Inheritance Check
OR14	any set type	any set type	Strict Inheritance Check
OR15	any class type	any class type	Strict Inheritance Check

4.7.7 Range Constructor Compatibility Class

Semantics

For full views of T1 and T2, the checks are as follows:

#	T1	T2	Check
RC1	any signed type	any signed type	Arithmetic Check
RC2	any unsigned type	any unsigned type	Arithmetic Check
RC3	any integer type	*universal integer*	Convertibility Check
RC4	*universal integer*	any signed type	Convertibility Check
RC5	*universal integer*	*universal integer*	Convertibility Check
RC6	any enumeration type	any enumeration type	Strict Inheritance Check
RC7	any logical type	any logical type	Convertibility Check
RC8	any character type	any character type	Convertibility Check

4.7.8 General Arithmetic Compatibility Class

Semantics

For full views of T1 and T2, the checks are joined Integer Arithmetic Compatibility Class and Real Arithmetic Compatibility Class checks.

4.7.9 Integer Arithmetic Compatibility Class

Semantics

For full views of T1 and T2, the checks are as follows:

#	T1	T2	Check
IA1	any signed type	any signed type	Arithmetic Check
IA2	any unsigned type	any unsigned type	Arithmetic Check
IA3	any integer type	*universal integer*	Convertibility Check
IA4	*universal integer*	any integer type	Convertibility Check
IA5	*universal integer*	*universal integer*	

4.7.10 Real Arithmetic Compatibility Class

Semantics

For full views of T1 and T2, the checks are as follows:

#	T1	T2	Check
RA1	any real type	any real type	Convertibility Check
RA2	any real type	*universal real*	Convertibility Check
RA3	*universal real*	any real type	Convertibility Check
RA4	*universal real*	*universal real*	Convertibility Check

4.7.11 Set Operations Compatibility Class

Semantics

For full views of T1 and T2, the checks are as follows:

#	T1	T2	Check
SO1	any set type	any set type	Set Check

4.7.12 Concatenation Compatibility Class

Semantics

For full views of T1 and T2, the checks are as follows:

#	T1	T2	Check
C1	any constrained string type	any constrained string type	
C2	any constrained string type	any unconstrained string type	
C3	any unconstrained string type	any string type	
C4	any string type	any *universal array* type	
C5	any *universal array* type	any string type	
C6	any *universal array* type	any *universal array* type	
C7	any *universal array* type	any unindexed or universal unindexed type	
C8	any unindexed or universal unindexed type	any *universal array* type	
C9	any constrained string type	any unindexed or universal unindexed type	
C10	any unconstrained string type	any unindexed or universal unindexed type	
C11	any unindexed or universal unindexed type	any constrained string type	
C12	any unindexed or universal unindexed type	any unconstrained string type	
C13	any unindexed type	any unindexed type	

4.7.13 Logical Operations Compatibility Class

Semantics

For full views of T1 and T2, the checks are as follows:

#	T1	T2	Check
LO1	any logical or *universal logical* type	any logical or *universal logical* type	Convertibility Check

4.7.14 Bitwise Operations Compatibility Class

Semantics

For full views of T1 and T2, the checks are as follows:

#	T1	T2	Check
BO1	any signed type	any signed type	Arithmetic Check
BO2	any unsigned type	any unsigned type	Arithmetic Check
BO3	any integer type	*universal integer*	Convertibility Check
BO4	*universal integer*	any integer type	Convertibility Check
BO5	*universal integer*	*universal integer*	Convertibility Check

4.7.15 Comparison Operations Compatibility Class

Semantics

For full views of T1 and T2, the checks are as follows:

#	T1	T2	Check
CO1	any signed type	any signed type	Arithmetic Check
CO2	any unsigned type	any unsigned type	Arithmetic Check
CO3	any integer type	*universal integer*	Convertibility Check
CO4	*universal integer*	any integer type	Convertibility Check
CO5	*universal integer*	*universal integer*	
CO6	any enumeration type	any enumeration type	
CO7	any logical or *universal logical* type	any logical or *universal logical* type	Convertibility Check
CO8	any character type	any character type	Convertibility Check
CO9	any real type	any real type	Convertibility Check
CO10	any real type	*universal real*	Convertibility Check
CO11	*universal real*	any real type	Convertibility Check
CO12	*universal real*	*universal real*	Convertibility Check
CO13	any pointer type	any pointer type	Pointer Comparison Check
	any string type		
	any array type		
	any record type		
	any set type		
	any class type		

4.7.16 Ordering Operations Compatibility Class

Semantics

For full views of T1 and T2, the checks are as follows:

#	T1	T2	Check
CO1	any signed type	any signed type	Arithmetic Check
CO2	any unsigned type	any unsigned type	Arithmetic Check
CO3	any integer type	*universal integer*	Convertibility Check
CO4	*universal integer*	any integer type	Convertibility Check

#	T1	T2	Check
CO5	*universal integer*	*universal integer*	
CO6	any enumeration type	any enumeration type	
CO7	any logical or *universal logical* type	any logical or *universal logical* type	Convertibility Check
CO8	any character type	any character type	Convertibility Check
CO9	any real type	any real type	Convertibility Check
CO10	any real type	*universal real*	Convertibility Check
CO11	*universal real*	any real type	Convertibility Check
CO12	*universal real*	*universal real*	Convertibility Check
	any string type		
	any array type		
	any set type		

4.7.17 Membership Tests Compatibility Class

Semantics

For full views of T1 and T2, the checks are as follows:

#	T1	T2	Check
MT1	any ordinal type	any set type	

5. Objects and Constants

Semantics

A constant is a compile-time entity that holds a value of a given regular or universal type. An object is a run-time entity that holds a value of a given regular type. An object can hold only values of its type and the value **nil**.

A constant is declared by a Constant declaration.

An object is:

— a variable declared by a Variable declaration,
— a subprogram formal parameter declared by a Formal parameter declaration or a Return parameter declaration,
— a dynamic object created at run-time by a New statement,
— a subcomponent of an object.

An object can be accessed indirectly through another object of a pointer-to-object type. For an object created by a New statement this is the only access posible.

Note: Static attributes are generally not constants — they do not always contain a value, they can reference other entities. Dynamic attributes are generally not objects (`:Length` of a string object is an object itself) — they can be synthesized at run-time from other properties of a given object.

A single declaration can denote more than one run-time object (either when the scope of a variable is entered more than once during the execution of a program or when a type — specially a class — is used to denote an object). In this case the word instance is used to denote a particular object.

5.1 Constant Declaration

Syntax

Constant declaration	Incomplete constant declaration
	Constant declaration completion
	Full constant declaration
Incomplete constant declaration	Identifier: Type
Constant declaration completion	Identifier = *Constant* expression [Attribute specification]
Full constant declaration	Identifier [: Type] = *Constant* expression [Attribute specification]

Semantics

The Type shall denote a constrained type that is not a class. For an Incomplete constant declaration, the Type can denote an incomplete type, and denote a constrained type in the corresponding Constant declaration completion.

If the type of a constant is not explicitly specified, it is the type of the Constant expression. If the type was explicitly specified, the Constant expression shall be of any compatible type.

5.2 Variable Declaration

Syntax

Variable declaration	Identifier: [Object access determination] [Memory class] [**ref**] Type [Attribute specification] [:= *Constant* expression]
Memory class	**static**
	class
	task

Semantics

A Variable declaration declares a varible and specifies its Identifier, Object access determination, Memory class and allocation, Type, optional attributes and optional initialization expression. The Type of a variable shall be a constrained type, unless the **ref** modifier is specified, which allows the type to be unconstrained.

Language design: We do not permit a declaration of multiple variables by specifying multiple identifiers in a single Variable declaration. This concept, as can be found in other languages, requires special handling by the implementation. We definitely do not want the implementation to evaluate a single construct (its source representation) and produce multiple equivalent objects — this scheme would be necessary to keep consistent relations between the included Attribute declarations and the variables being declared — which would make the implementation much harder.

The run-time scope of a variable is determined by its memory class. A memory class is specified either implicitly in consequence to the semantics of the enclosing entities or explicitly by a Memory class. Following memory classes exist:

- automatic,
- **static**,
- **class**,
- **task**.

A variable's default memory class is automatic, if it was declared within a Subprogram declaration, an Override declaration, a Special method declaration or a Task declaration, and it is **class**, if it was declared within a Class declaration. In all other cases the default memory class is **static**.

An explicit Memory class specification is not allowed within a macro. A **class** memory class can be specified only within a Class declaration and a Subprogram declaration, that is enclosed within a Class declaration.

The memory class of a variable determines its creation and destruction:

- An automatic variable is created when the execution enters the enclosing subprogram and destroyed when it is left. If the subprogram is entered more than once, a new instance of the variable is created.

  Note: This implies, that automatic variables are created on a stack (or equivalent data structure), and that there is one stack per task.

- A **static** variable is created when the execution of the compilation unit starts and destroyed when it ends.
- A **class** variable is created when an instance of the enclosing Class declaration is created and is destroyed when the instance is destroyed.
- For each task, a new instance of a **task** variable is created when the task starts and destroyed when it ends.

Furthermore for a variable declared with the **ref** modifier there is an automatic memory management (allocation a deallocation of memory) provided. For variables of a constrained type memory is allocated upon creation, for variables of an unconstrained type it is allocated upon first change (which may include initialization) and reallocated whenever needed. This does not apply for indirect access to the variable through a pointer or a parameter. All occupied memory is released upon variable destruction. `Memory_Error` can be raised during changes that can affect the size of a **ref** variable (that is also during initialization).

Note: Note that automatic reallocation of memory for variables of unconstrained types means that corresponding `:Last` attributes can change during the execution of the appropriate subprogram. On the other hand the value of `:Last` (and thus the amount of memory allocated) is preserved during calls to other subprograms.

Language design: The **ref** variables are not intended as a substitute to a garbage collector. Its intent is to provide automatic memory management for easy manipulation with unconstrained arrays and strings without the need to handle memory management manually. We believe that garbage collectors shall not be applied universaly and shall be provided by other means than the language itself (at least as oposed to the current state of research in this area) and do only tasks that the programmer wants them to do. The main reason against including a general purpose garbage collector into Flex is indeterministic finalization of objects which shall not be allowed in high-integrity applications. The **ref** variables offer basic automatic memory management for simple memory allocation tasks while not disallowing perfect control over complex abstract dynamic data structures.

An implicit value of a variable can be specified by a Constant expression, that is of a compatible type. When no implicit value is specified, the default implicit value is **nil**. Every variable is upon its creation automatically initialized to its implicit value.

Note: Initializing a **ref** variable of an unconstrained string type to even only the default implicit value **nil** may lead to allocation of at least as much memory needed to hold the run-time `:Length` attribute. On the other hand initializing a **ref** variable of an unconstrained array type to **nil** may not require any memory allocation.

5.3 Object Access Determination

Syntax

Object access determination	**protected**
	const

Semantics

The Object access determination limits the set of operations allowed on an object.

If the keyword **const** is specified in an object declaration (that is either a Variable declaration or Record item declaration) or if an object is accessed through a pointer declared with the **const** modifier within its type's Pointer *to object* type definition or if it is a subcomponent of such an object, neither of the following is allowed:

- changing the object's value using any means of the language (e.g. assigning to it, changing any of its dynamic attributes),
- passing the object as an **out** or **in out** parameter to a subprogram,
- passing the object as a destination parameter in an Accept statement,
- passing the reference to the object to a pointer whose type was not defined with the **const** Object access determination.

If the object is of a class type, calling any of its methods is allowed even if it changes its value.

If an objects contains subcomponents declared as **const** (e.g. a record with a **const** in a Record item declaration), than only following rules apply to that object:

- performing any changes to the object that could affect the **const** subcomponents (e.g. assigning to it),
- passing the object as a destination parameter in an Accept statement.

The only modifications allowed to a **const** object are during its initialization and finalization.

The **protected** modifier can be specified only when the object or pointer-to-object type is declared within a public or protected part of an enclosing module or class. If **protected** is specified in an object declaration (that is either a Variable declaration or Record item declaration) or if an object is accessed through a pointer declared with the **protected** modifier within its type's Pointer *to object* type definition or if it is a subcomponent of such an object, neither of the following is allowed outside the module or class enclosing the corresponding object or pointer-to-object type declaration:

- changing the object's value using any means of the language (e.g. assigning to it, changing any of its dynamic attributes),
- passing the object as an **out** or **in out** parameter to a subprogram,
- passing the object as a destination parameter in an Accept statement,
- passing the reference to the object to a pointer whose type was not defined with the **const** modifier or was defined with a **protected** modifier, but its enclosing module or class does not enclose (or is not equal to) that of the object.

  Note: The last condition ensures that the protection can be only widened and never narrowed.

If an objects contains subcomponents declared as **protected** (e.g. a record with a **protected** in a Record item declaration), some or all of that are being accessed outside the enclosing module or class corresponding to their places of declaration, than following rules apply to that object:

- performing any changes to the object that could affect those **protected** subcomponents (e.g. assigning to it),
- passing the object as a destination parameter in an Accept statement,

Examples

```
type
  tstring = string 100 of char;
  pstring = ^tstring;
  cstring = ^const tstring;

procedure display_tstring (p: in tstring) =...;
procedure display_pstring (p: in pstring) =...;
procedure display_cstring (p: in cstring) =...;

var
  text: const tstring:= 'Hello World!';

begin
  text:='Hello All!';        -- illegal
  display_tstring(text);     -- legal
  display_pstring(^text);    -- illegal
  display_cstring(^text);    -- legal
```

6. Names and Expressions

6.1 General Expression Evaluation Rules

Semantics

The evaluation of an expression (or of its subexpressions) can be either compile-time or run-time. If all parts of an expression can be evaluated in compile-time it is said to be a constant expression. All possible compile-time evaluations shall be performed in compile-time and shall not be deferred until run-time.

Note: This requirement of all possible compile-time evaluations does not include optimizations of expressions whose constant subexprepressions determine a constant result of the whole expression. On the other hand such optimizations are not denied to be used in evaluations of *Constant* expressions.

A context of an expression is the enclosing construct (which can also be an expression), that sets constraints on the expression. These constraints (both static and dynamic) specify the required expression class, allowed types, dynamic attributes and other context-dependent characteristics.

In expression evaluation dynamic attributes are utilized to ensure run-time checks a dynamic semantics. An expression is said to provide a dynamic attribute if it denotes an object, that contains the value of that dynamic attribute or has any other characteristics that can be used to compute this value (e.g. the `:Size` attribute can be computed from the `:Last` attribute and vice-versa). An expression must have a specific dynamic attribute, if it is required so by its context.

The quality of provisioning a dynamic attribute does not imply any requirements on physical availability or actual computation of the attribute when this is not required by the context.

6.1.1 Classes of Expressions

Each expression is classified to be of a specific expression class, which is further used in other language rules:

- the expression of the module class denotes a module,
- the expression of the variable class denotes an object,
- the expression of the value class denotes a value that is a result of a computation,
- the expression of the subprogram class denotes a subprogram,
- the expression of the message class denotes a message,
- the expression of the type class denotes a type,
- the expression of the tag class denotes a tag,
- the expression of the composite class denotes a comoposite value (e.g. a range),
- all other expression are unclassified.

6.2 Names

Syntax

Name	Direct name
	Selected component
	Indexed component or slice
	Dereference
	Call
	Attribute
	Type conversion
	Array aggregate

	String aggregate
	Record aggregate
	Set aggregate
	Literal
	(Expression)

Semantics

A Name denotes an entity — and its value if it is an object — or a value.

Several dynamic attributes can be associated with a Name that denotes an object.

6.2.1 Direct Names

Syntax

Direct name	Extended identifier
	this
	current Entity selector
Extended identifier	Identifier
	Special method identifier
Entity selector	Special method identifier
	procedure
	static
	virtual
	override
	task
	macro
	class
	module
	program

Semantics

For a Direct name that is an Identifier, it denotes an entity that is directly visible at the given place and whose identifier is the same as the given one. If there are more directly visible entities with the same identifier, the Direct name denotes the first one (assuming the order implied by the application of the rules for direct visibility, see chapter 3.2).

Note: If the Direct name denotes an overload, it still denotes only one entity even that all directly visible overloads with the same identifier apply during the overload resolution.

For a Direct name that is a Special method identifier, it denotes the corresponding special method of the enclosing module or class (even if it was not declared, see chapter 3.4.1).

For a Direct name that is the **this** keyword, it denotes the enclosing module or the current instance of an enclosing class.

For a Direct name that is the **current** keyword, it denotes the most enclosing entity of kind given by the Entity selector.

The type of a Direct name is as follows:

- for a type or class it is the type or class itself,
- for an object or constant it is the type of the object or constant,
- for an enumeration item it is the corresponding enumeration type,
- for a record item it is the type of the record item,
- for an array component it is the base type of the array,
- for a subprogram it is the type of the subprogram,
- for other kinds of entities no type is associated with the given Direct name.

6.2.2 Selected Components

Syntax

Selected component	Name. Extended identifier

Semantics

The Name shall denote an entity that can contain named subcomponents, that is one of the following:

— module, class, subprogram, task, program or a special method,

— an expression of a record or class type (whose current view is a full view) (or a record type itself).

The Extended identifier denotes a visible subcomponent of the base entity.

The type of a Selected component depends on the kind of the subcomponent and is selected according to the rules presented for a Direct name.

6.2.3 Indexed Components and Slices

Syntax

Indexed component or slice	Name "[" Choice {, Choice } "]"
Choice	Expression
	Range
Range	Simple expression.. Simple expression
	Discrete type name

6.2.4 Dereferences

Syntax

Dereference	Name ^

6.2.5 Calls

Syntax

Call	Subprogram name [List of actual parameters]
Subprogram name	*Procedure* name
	Macro name
	Overload name
	Static subprogram name
	Virtual subprogram name
	Special method name
	Task name
List of actual parameters	([Parameter association] {, [Parameter association] })
Parameter association	Expression
	for *Parameter* identifier **use** Expression

Semantics

6.2.6 Attributes

Syntax

Attribute	Name: *Attribute* identifier

6.2.7 Type Conversions

Syntax

Type conversion	*Type* name (Expression)

6.2.8 Aggregates

Syntax

Array aggregate	[*Array type* name:]
	"[" [Array item association {, Array item association }] "]"
	Character aggregate
String aggregate	[*String type* name:]
	"[" [Array item association {, Array item association }] "]"
	Character aggregate
Array item association	Expression
	for Choice **use** Expression
	for others use Expression
Record aggregate	[*Record type* name:] "[" Record item association
	{, Record item association } "]"
Record item association	Expression
	for *Record item* identifier **use** Expression
Set aggregate	[*Set type* name:] "[" Choice {, Choice } "]"

6.2.9 Literals

Syntax

Literal	Numeric literal
	Character literal
	nil

Semantics

A Literal represents a direct value given in its textual form as a single lexical element. The type of an Integer number is *universal integer*, the type of a Real number is *universal real*, the type of a Character literal is *universal character* and the type of a **nil** is *universal NIL*.

6.2.10 Parenthesized Expressions

Semantics

A parenthesized expression is an Expression enclosed in parenthesis.

6.3 Expressions

Syntax

Expression	Relation { **and** Relation }
	Relation { **or** Relation }
	Relation { **xor** Relation }
	Relation { **and then** Relation }
	Relation { **or else** Relation }
Relation	Simple expression [Relational operator Simple expression]
	Simple expression [**not**] **in** Range
Relational operator	= | <> | < | <= | > | >=
Simple expression	[Unary adding operator] Term { Adding operator Term }
Unary adding operator	+ | -
Adding operator	+ | - | &
Term	Factor { Multiplying operator Factor }
Multiplying operator	* | / | **div** | **mod** | **shl** | **shr**
Factor	[High precedence unary operator] { ^ } Name
High precedence unary operator	**not** | **succ** | **pred** | **abs**

6.3.1 Resolution of Types, Overloaded Subprograms and Operators

Semantics

6.3.2 Operators and Expression Evaluation

6.3.2.1 Logical Operators

Semantics

6.3.2.2 Relational Operators

Semantics

The type of a Relation is *universal logical*.

The equality operators = and <> are defined for all compatible types.

For discrete types the relational operators are defined in terms of corresponding mathematical operations on the ordinal values of the values of the operands.

For real types the relational operators are defined in terms of corresponding mathematical operations on the values of the operands.

Two pointer-to-object values are equal if they designate the same object.

Two pointer-to-subprogram values are equal if they designate the same subprogram and the same instance of a class (if any operand designates a pointer-to-method value).

Two pointer-to-task values are equal if they designate the same task.

6.3.2.2.1 Membership Tests

Semantics

6.3.2.3 Binary Adding Operators

Semantics

6.3.2.4 Unary Adding Operators

Semantics

6.3.2.5 Multiplying Operators

Semantics

6.3.2.6 High Precedence Unary Operators

Semantics

6.3.3 Reference

Semantics

The symbol "^" in a Factor represents a reference to a runtime representation of an entity or object. It can be applied recursively to cancel the effects of both explicit and implicit dereferences.

7. Statements

Syntax

Sequence of statements	{ Label } [Statement] {; { Label } [Statement] }
Label	*Label* name:
Statement	Simple statement
	Compound statement
Simple statement	Assignment statement
	Call statement
	Goto statement
	Break statement
	Return statement
	New statement
	Discard statement
	Delay statement
	Send statement
	Raise statement
	In statement
	Out statement
Compound statement	Accept statement
	If statement
	Case statement
	Loop statement
	Block
	Concurrent block
	Sequential block
	Declare statement

Semantics

A Statement defines an action to be performed upon its execution.

The normal order of execution of Statements inside a Sequence of statements is the same as their order in the source code. A transfer of control can lead to execution of another statement that would be normally expected.

The term completion of execution of a Compound statement refers to a set of actions taken when the execution of the Compound statement is completed normally or when a transfer of control is performed from inside of the Compound statement to another Statement not enclosed by the Compound statement.

When a transfer of control occurs from a source Statement to a target Statement, execution of all Compound statements, enclosing the source but not the target Statement, is completed in order from the innermost to the outermost Compound statement.

Note: For simplicity we use the term Statement also to denote such targets of a transfer of control where is no statement and rather an end of a Sequence of statements is present. In this case we mean the execution continues at the innermost enclosing Compound statement or Statement part (its particular section respectively).

7.1 Assignment Statement

Syntax

Assignment statement	Name:= Expression
	[Unary operator] Name **and** Relation { **and** Relation }
	[Unary operator] Name **or** Relation { **or** Relation }
	[Unary operator] Name **xor** Relation { **xor** Relation }
	[Unary operator] Name **and then** Relation { **and then** Relation }
	[Unary operator] Name **or else** Relation { **or** else Relation }
	[Unary operator] Name Relational operator Simple expression
	[Unary operator] Name Adding operator Term { Adding operator Term }
	[Unary operator] Name Multiplying operator Factor { Multiplying operator Factor }
	Unary operator Name
Unary operator	Unary adding operator
	High precedence unary operator

Semantics

The left operand of the assignment operator is required to have the `:Size` attribute if it is an unchecked view of an object. It is required to have the `:Last` attribute if it denotes an object of an unconstrained type. The right side of the assignment operator is requi

7.2 Call Statement

See chapter 8.5

7.3 Goto Statement

Syntax

Goto statement	**goto** *Label* name

Semantics

The *Label* name shall denote a label declared in the innermost enclosing Subprogram body.

The execution of a Goto statement transfers control to the target label — next statement executed is the one immediately following the target label — while completing the execution of all Compound statements.

Note: According to the rules given in chapter 7 execution of all Compound statements, enclosing the Goto statement but not the target label, is completed.

If the Goto statement and the target label are both enclosed in the same Statement part, then they shall be both enclosed in the same section.

A Goto statement is not permitted in the Concurrent sequence of statements. If a Goto statement is (recursively) enclosed in a Compound statement, that is part of a Concurrent sequence of statements, then the target label shall be also enclosed in the same Compound statement.

7.4 Break Statement

Syntax

Break statement	**break**

7.5 Return Statement

Syntax

Return statement	**return**

7.6 New Statement

Syntax

New statement	**new** *Object* name [**tag** Expression] [**range** *Integer* expression]

Semantics

A New statement allocates memory for the given object.

The Object name shall denote an object of a pointer-to-object type.

7.7 Discard Statement

Syntax

Memory deallocation	**discard** *Variable* name

7.8 Delay Statement

Syntax

Delay statement	**delay** *Numeric* expression

Semantics

A Delay statement causes the task to stop its execution and wait for the given time in seconds. Waiting can be interrupted by an asynchronous message.

The Expression shall be of any real or integer type.

Execution of a Delay statement consumes no or only least necessary CPU time.

Implementation Defined

The timer resolution is implementation defined.

7.9 Send Statement

See chapter 10.3.

7.10 Raise Statement

See chapter 10.5.

7.11 Accept Statement

See chapter 10.4.

7.12 In Statement

See chapter 14.2.

7.13 Out Statement

See chapter 14.3.

7.14 If Statement

Syntax

If statement	**if** *Logical* expression **then** Sequence of statements
	{ **elsif** *Logical* expression **then** Sequence of statements }
	[**else** Sequence of statements]
	end if

Semantics

An If statement conditionally executes at most one of its Sequences of statements. The Logical expression shall be of any logical type.

The Logical expressions after **if** and **elsif** are evaluated in succession, until one of them evaluates to `True` or all evaluate to `False`. When a Logical expression evaluates to `True`, the corresponding Sequence of statements is executed. If all Logical expressions evaluate to `False`, the Sequence of statements after **else** is executed if present, otherwise the execution of an If statement has no effect except evaluation of the Logical expressions.

7.15 Case Statement

Syntax

Case statement	**case** Expression { Alternative } **end case**
Alternative	**when** *Constant* choice {, *Constant* choice } **do**
	Sequence of statements
	when others do Sequence of statements

7.16 Loop Statement

Syntax

Loop statement	[Iteration scheme] **loop** Sequence of statements **end loop**
	loop Sequence of statements **until** *Logical* expression
Iteration scheme	**for** *Variable* identifier **in** [Order determination] Range
	for *Variable* identifier [Order determination]
	while *Logical* expression
Order determination	**reverse**
	concurrent

7.17 Block

Syntax

Block	**begin** Statement part **end**
Statement part	[Sequence of statements]
	[Catch section]
	[Leave section]
Leave section	**leave** [Sequence of statements]

7.18 Concurrent Block

See chapter 9.3.

7.19 Sequential Block

See chapter 9.4.

7.20 Declare Statement

Syntax

Declare statement	**declare** { Declaration } **begin** Statement part **end declare**

8. Subprograms

Syntax

Subprogram declaration	Procedure declaration
	Static subprogram declaration
	Virtual subprogram declaration
	Macro declaration
Subprogram heading	[List of formal parameters] [Return parameter declaration]
	: *Subprogram type* name
List of formal parameters	(Formal parameter declaration {; Formal parameter declaration })
Formal parameter declaration	Identifier: [Mode] [Control] *Regular type* name
	[Attribute specification] [:= *Constant* expression]
	Identifier: [Mode] **unchecked** [Attribute specification]
Return parameter declaration	**return** [Return control] *Regular type* name
Mode	**in**
	out
	in out
Control	**static**
	ref
	unchecked
	class
	virtual
Return control	**class**
	virtual
Subprogram body	[With clause] [Use clause]
	{ Declaration }
	begin Statement part **end** Identifier

Semantics

A subprogram represents a run-time entity that can be called by a Call to execute its Statement part, that is part of its Subprogram body. Several kinds of subprograms exist, that differ in how and when a call can be performed:

- procedures (see chapter 8.1),
- static subprograms (see chapter 8.2),
- virtual subprograms (see chapter 8.3),
- macros (see chapter 14).

Note: Note that special methods and tasks are also called using Calls, but they are separated from subprograms because of substantial differences in their semantics.

The Subprogram heading specifies the subprogram's type either by its direct implicit declaration or by a reference to an existing subprogram type with a *Subprogram type* name.

The Formal parameter declaration declares a formal parameter and specifies its Identifier, Mode, Control, type, optional attributes and initial value. Formal parameters of a subprogram represent values that are passed to and/or returned from the subprogram.

Glosasry[Formal Parameter]: A formal parameter is an entity representing within a subprogram a value passed to and/or returned from that subprogram. One formal parameter can be declared as a return parameter (see definition).

The Return parameter declaration declares a formal parameter of mode **out** and a language defined identifier `Result`. Subprograms that return a value (its type has a Return parameter declaration) are function subprograms, all other are non-function subprograms.

Glosasry[Return Parameter]: A return parameter is a selected formal parameter that is used to return value from a subprogram. This value is interpreted as a result of a Call.

The common word argument (more precisely a formal argument) is used for short instead of a formal parametr. The word argument doesn't include the return parametr. The combination of formal parameters (and thus also a return parameter) and their modes and controls is refered to as a parameter profile.

Glosasry[Parameter Profile]: A parameter profile of a subprogram is the actual list of properties of formal parameters and the optional return parameter regardless of any type of the subprogram or identifiers of arguments — that is the list of combinations of <mode, control, type> of all formal parameters and the optional return parameter. This term is used rather informaly especially when speaking about counts of parameters and its properties.

Glosasry[Argument]: The word argument is used as a synonym to Formal Parameter and Actual Parameter with respect to the context of the actual usage.

The mode of a parameter specifies the way in which the parameter's value is transfered between the caller and the called subprogram. Following modes exist:

— The mode **in** specifies that the value is passed from the caller to the subprogram and cannot be modified within the subprogram. The corresponding object is neither initialized nor finalized within the subprogram (unless specified otherwise by the Control modifier or a special calling convention).

Note: Specifying a parameter of a pointer-to-object type as **in** does not prevent modifications of the object the pointer is pointing to. When such behavior is required, a pointer-to-constant-object shall be used instead.

— The mode **out** specifies that the value is passed from the subprogram back to the caller. The corresponding object is first initialized upon the execution enters the subprogram. The value of such formal parameter can be freely modified within the subprogram.

— The mode **in out** specifies that the value is first passed from the caller to the subprogram and after the subprogram's execution completes, it is returned back to the caller. The corresponding object is neither initialized nor finalized within the subprogram. The value of such formal parameter can be freely modified within the subprogram.

When no mode is specified, **in** is assumed by default.

Language design: The modes of parameters are designed as the primary way of describing data-flow. We believe that the traditional view of passing parameters by value or by reference is rather obsolete and that specification of way how values are transfered between callers and callees is much more important for program design issues, source readability and clarity. There are way how to precisely control the passing mechanisms and calling conventions (see later paragraphs), but these are provided for system programing and low-level tasks rather than every day application use.

The technological way of how and where the values of parameters are passed and the actual mechanism of a subprogram call is specified by a calling convention. The detailed specification of any calling convention is out of scope of this Language Reference Manual. The only definition provided here are the abstract ways how parameters shall be passed within a *default calling convention*, which interconnects the modes of parameters with *by value* and *by reference* parameter passing mechanisms:

— an **in** parameter of a simple type shall be passed by value unless modified by the Control modifier,

— an **in** parameter of a composite or class type shall be passed by reference unless modified by the Control modifier,

— an **out** or **in out** parameter shall be always passed by reference regardless of the Control modifier.

The implementation shall assume this *default calling convention* by default for all subprograms.

The control of a parameter allows control of specific features:

— the abstract technological way how a parameter shall be passed (either by value or by reference),

— type checking rules,

— behavior of virtual calls.

The five provided Control modifiers affect these features as follows:

- The **static** modifier specifies that the parameter shall be passed by value. In the *default calling convention* this modifier is not permitted for parameters with modes **out** and **in out**.
- The **ref** modifier specifies that the parameter shall be passed by reference. In the *default calling convention* this modifier has no effect for parameters with modes **out** and **in out**.
- The **unchecked** modifier specifies that the parameter shall be passed by reference and that no type checking shall be performed. When a parameter is marked as **unchecked**, its type need not be specified and thus resulting in a type-less parameter. The default view of such parameter's type is the unchecked view (see chapter 4.2).
- The **class** modifier specifies that the parameter shall be passed by reference together with its dynamic type information (its dynamic **:**`Tag` attribute).
- The **virtual** modifier has the same properties as the **class** modifier and furthermore marks the corresponding parameter as a dispatching parameter of a virtual call.

Language design: As prescribed in a Language Design note above, the modifiers **static** and **ref** are provided primarily for enhancing interoperability with other languages, systems and third-party APIs (typically in conjunction with an implementation defined calling convention). The modifier **unchecked** is intended for low-level routines that need to operate on a variety of types careless of the inheritance hierarchy (e.g. low-level memory manipulation routines). On the other hand, the modifiers **class** and **virtual** support object-oriented programming on a high level of abstraction.

However these modifiers provide so different functionality, we decided to put them all under a single syntax category because they cannot be separated into orthogonal groups of properties with no significant impact one to the another. For example, the combination **static unchecked** might be reasonable under some conditions, but it would be so rarely used and confusing, that it is worth disallowing — and so do other combinations like **unchecked virtual**, which is highly confusing, or **static virtual**, which is unimplementable.

If an **in** parameter of a class type were to be passed by value due to a calling convention or a **static** modifier, its value shall be adjusted upon entry of the subprogram by calling the **adjust** special method and finalized before exit by calling the **exit** special method. The implementation should report a warning in this case and a separate warning in all other cases when a parameter of a class type is declared as to be passed by value regardless of the mode of the parameter.

Note: Passing a parameter of a class type by value unintentionally could cause unexpected behavior due to class instance duplication that is not easy to track down.

Implementation Defined

The implementation may provide a pragma `Convention` to override the *default calling convention* and thus enhance interoperability with other languages and systems. The pragma shall take two arguments — first specifying the Name of the subprogram and the second specifying the Identifier of the desired calling convention. The identifier `Default` shall be reserved to represent the *default calling convention*.

8.1 Procedures

Syntax

Procedure declaration	Incomplete procedure declaration Abstract procedure declaration Procedure declaration completion Full procedure declaration
Incomplete procedure declaration	**procedure** Identifier [Subprogram heading] [Attribute specification];
Abstract procedure declaration	**procedure abstract** Identifier [Subprogram heading] [Attribute specification];
Procedure declaration completion	**procedure** Identifier = Subprogram body;

Full procedure declaration	**procedure** Identifier [Subprogram heading] [Attribute specification] = Subprogram body;

Semantics

A procedure is a subprogram that can be called by specifying its name and actual parameters.

8.2 Static Subprograms

Syntax

Static subprogram declaration	Incomplete static subprogram declaration Abstract static subprogram declaration Static subprogram declaration completion Full static subprogram declaration
Incomplete static subprogram declaration	**static** Identifier [Subprogram heading] [Attribute specification];
Abstract static subprogram declaration	**static abstract** Identifier [Subprogram heading] [Attribute specification];
Static subprogram declaration completion	**static** Identifier = Subprogram body;
Full static subprogram declaration	**static** Identifier [Subprogram heading] [Attribute specification] = Subprogram body;

Semantics

A Static subprogram declaration may appear only within a Class declaration.

8.3 Virtual Subprograms

Syntax

Virtual subprogram declaration	Incomplete virtual subprogram declaration Abstract virtual subprogram declaration Virtual subprogram declaration completion Full virtual subprogram declaration
Incomplete virtual subprogram declaration	**virtual** Identifier [Subprogram heading] [Attribute specification];
Abstract virtual subprogram declaration	**virtual abstract** Identifier [Subprogram heading] [Attribute specification];
Virtual subprogram declaration completion	**virtual** Identifier = Subprogram body;
Full virtual subprogram declaration	**virtual** Identifier [Subprogram heading] [Attribute specification] = Subprogram body;

Semantics

A Virtual subprogram declaration may appear only within a Class declaration, Module declaration, Program declaration and a Compilation unit.

8.4 Overrides of Abstract or Virtual Subprograms

Syntax

Override declaration	**override** *Virtual or abstract subprogram* name [Override heading] = Subprogram body;
Override heading	**with** (Controlling parameter specification {, Controlling parameter specification })
Controlling parameter specification	*Type* name **for** *Parameter* identifier **use** *Type* name

Semantics

An Override declaration may appear only within a Class declaration, Module declaration, Program declaration and a Compilation unit.

8.5 Call Statement

Syntax

Call statement	Call *of a non-function subprogram*

9. Tasks and Concurrency

9.1 Task Type Definition

Syntax

Task type definition	**task** [List of formal parameters]

9.2 Task Declaration

Syntax

Task declaration	Incomplete task declaration Task declaration completion Full task declaration
Incomplete task declaration	**task** Identifier [Subprogram heading] [Attribute specification];
Task declaration completion	**task** Identifier = Subprogram body;
Full task declaration	**task** Identifier [Subprogram heading] [Attribute specification] = Subprogram body;

9.3 Concurrent Block

Syntax

Concurrent block	**concurrent** Concurrent sequence of statements **end concurrent**
Concurrent sequence of statements	[Statement] {; [Statement] }

Note: Note that a Label is not permitted in a Concurrent sequence of statements. See also rules for the Goto statement in chapter 7.3.

9.4 Sequential Block

Syntax

Sequential block	**sequential** Sequence of statements **end sequential**

10. Messages

Messages provide a generalized mechanism for synchronous and asynchronous inter-task communication and task synchronization.

Note: Asynchronous messages provide similar functionality as the exception mechanism found in other languages.

Language design: We considered making a single mechanism for exceptions and inter-task communication and so reducing the amount of language rules, improving future extensibility, and increasing the value of the language construct. The notion of synchronous and asynchronous communication also fits well to this scheme.

This approach does not increase the implementation complexity; on the contrary, it is rather simplifying.

Syntax

Message declaration	**message** Identifier [Message heading] [Attribute specification];
Message heading	[List of message parameters]
	extend Name *of a message*
	: *Message type* name
List of message parameters	(Message parameter {; Message parameter })
Message parameter	Identifier: *Type* name [:= *Constant* expression] [Attribute specification]

Predefined Environment

```
type
  exception = message;

message program_error: exception;
message constraint_error     extend program_error;
message memory_error         extend program_error;
message tasking_error        extend program_error;
message implementation_error extend program_error;
message generic_error        extend program_error;
message task_abort: exception;
```

10.1 Message Type Definition

Syntax

Message type definition	**message** [List of message parameters] [Attribute specification]
List of message parameters	(Message parameter {; Message parameter })

10.2 Queue Type Definition

Syntax

Queue type definition	**queue** [**of** *Message or message type* name {, *Message or message type* name }]

10.3 Send Statement

Syntax

Send statement	**send** [Name *of message with actual parameters*] [**to** *Task or queue* name] [Lifetime determination] [Send accepted alternative] [Send time-out alternative] [**end send**]
Lifetime determination	**delay** *Real* expression
	do Sequence of statements
Send accepted alternative	**when accept do** Sequence of statements
Send time-out alternative	**when delay do** Sequence of statements

Semantics

A synchronous message is sent to a specified task or queue by a Send statement. The receiving task or queue is specified in the **to** clause. If no task or queue is specified, the message is sent to the current task. The message can be accepted by the receiving task or fetched from the receiving queue with an Accept statement.

Every message must be accepted by the receiving task or fetched from a queue with an Accept statement within a period of time, that is called the lifetime of the message, otherwise a time-out occurs and the message is discarded. The lifetime of a message can be explicitly specified with the Lifetime determination, which is either a **delay** clause or a **do** clause.

A **delay** clause specifies the lifetime in seconds. If the lifetime is zero, the message must be accepted immediately by the receiving task — it must be executing an Accept statement that can handle the message being sent. If the lifetime is less than zero, `Constraint_Error` is raised in the current task and the message is not sent.

A **do** clause specifies a Sequence of statements that are being executed while the message is being sent. If the message is not accepted until the **do** clause is executed, it is discarded.

If no Lifetime determination is specified, the lifetime of the message is infinite.

When a Send accepted alternative or a Send time-out alternative is present in a Send statement, it is said to be blocking and the sending task is blocked until the message is accepted or times-out (unless a **do** clause is specified). Otherwise the Send statement is non-blocking and the sending task continues in normal execution (possibly executing a **do** clause within the Send statement).

A Send accepted alternative is executed when the message was accepted by the receiving task. A Send time-out alternative is executed when the message timed-out.

A message cannot be delivered when........ Tasking_Error is raised when a message cannot be delivered to the receiving task during the execution of a blocking Send statement.

Examples

Reporting status:

```
message
  Starting;
  Aborting;
  Request_Work;
  Requested_Work (Job_ID: natural);
  Working (Job_ID: natural);
  End_of_Job;

var
  -- some global variables
  Client_Task: array 1..10 of ^task;
  Monitoring_task: ^task;
  i: Client_Task:range:= i:first;

task Working_Task =
var
  Job_ID: natural;

begin
  -- inform monitoring task that we started work
  send Starting to Monitoring_Task^ delay 10.0 end send;

  -- main loop
  loop
    -- request some work, if the client is ready
    send Request_Work to Client_Task[i]^ delay 0.0
      when accept do
        accept delay 1.0

          -- wait for requested work (at most 1.0 second)
          when Requested_Work(Job_ID) do
            -- Inform the monitoring task, that we are going to do some work
            send Working(Job_ID) to Monitoring_Task^
              do
                -- do the requested job
                Perform_Job(Job_ID);

              -- message accepted during the calculation
              when accept do
                -- notify the monitor, that we are finished
                send End_of_Job to Monitoring_Task end send;

              -- message timed-out
              when delay do
                -- do not send any notifications
              end send;

          -- answer not arrived in time
          when delay do
            -- something went wrong
```

```
              Report_Error('Answer to request not arrived');
            end accept;
        end send;

      -- serve next task
      if i=i:last
        then i:=i:first
        else succ i
        end if;
      end loop;

  catch
    when Task_Aborted do
        send Aborting to Monitor_Task delay 10.0 end send;
    end Working_task;
```

10.4 Accept Statement

Syntax

Accept statement	**accept** [Accept constraint] { Accept alternative } [Accept others alternative] [Accept timeout alternative] **end accept**
Accept constraint	**delay** [*Real* expression]
Accept alternative	**when** Name *of message with destination parameters* {, Name *of message with destination parameters* } **do** Sequence of statements
Accept others alternative	**when others do** Sequence of statements
Accept timeout alternative	**when delay do** Sequence of statements

10.5 Raise Statement

Syntax

Raise statement	**raise** [Name *of message with actual parameters*] [**in** *Task* name]

10.6 Catch Section

Syntax

Catch section	**catch** { Accept alternative } [Accept others alternative]

11. Overloading

11.1 Overloading of Subprograms

Syntax

Overload declaration	**overload** Identifier: *Subprogram* name {, *Subprogram* name };

11.2 Operator Declarations

Syntax

Operator declaration	**overload** Overloadable operator: *Procedure* name {, *Procedure* name };
Overloadable operator	:= | + | - | * | / | & | **div** | **mod** | **shl** | **shr** < | > | <= | >= | = | <> **and** | **or** | **xor** | **not** | **succ** | **pred** | **abs**

Semantics

The Operator declaration associates a predefined operator with a set of procedures. These procedures overload the predefined semantic of the given Overloadable operator. An Operator declaration for a given operator can appear at most once within a single declarative region and is subject to the common visibility rules.

Implementation advice: The implementation should treat the textual representation of the operator (e.g. a symbol or keyword) as an identifier (possibly enclosed in quotes) to ease the application of visibility rules and other appropriate language rules.

Unary and binary adding operators "+" and "-" are represented by same symbols and thus can be overload only in same declarations. The resolution of whether a unary or binary form is to be overloaded is made according to the count of arguments of the given procedure.

There is also no distinction between the standard and assignment forms of the operators. The form is determined according to the presence or absence of the overloading procedure's return parameter.

Language design: Membership test operators **in** and **not in** allow choices to appear as the right operand. Logical operators **and then** and **or else** assume a special evaluation scheme. In both cases, such behavior could not be reached by overloading, so overloading of these operators is not permitted.

Following restrictions apply to the types of arguments and return values of the defining procedures:

— For a procedure that has one argument and a return parametr, the Overloadable operator shall be any overloadable unary operator. The mode the argument shall be **in** and it shall not be declared as **unchecked**. Such procedure represents an overloaded standard form of a unary operator.

— For a procedure that has one argument and no return parametr, the Overloadable operator shall be any overloadable unary operator. The mode of the argument shall be **in out** and it shall not be declared as **unchecked**. Such procedure represents an overloaded assignment form of a unary operator.

— For a procedure that has two arguments and a return parametr, the Overloadable operator shall be any overloadable binary operator except ":=". The mode of both arguments shall be **in** and no argument shall be declared as **unchecked**. Such procedure represents an overloaded standard form of a binary operator.

- For a procedure that has two arguments and no return parametr, the Overloadable operator shall be any overloadable binary operator except ":=". The mode of the first argument shall be **in out**, the mode of the second argument shall be **in** and no argument shall be declared as **unchecked**. Such procedure represents an overloaded assignment form of a binary operator.
- For the assignment operator ":=", the procedure shall have two arguments and no return parametr. The mode of the first argument shall be **out**, the mode of the second argument shall be **in** and no argument shall be declared as **unchecked**.
- No other procedures can be used to overload operators.

11.3 Overload Resolution Rules

12. Attributes

12.1 Attribute Specification

Syntax

Attribute specification	{ Attribute specification item }
Attribute specification item	Attribute determination
	Attribute declaration
Attribute determination	**for** Identifier *of a language defined attribute*
	use *Constant* expression
Attribute declaration	**attribute** Identifier:= Attribute value
Attribute value *Constant* expression	*Entity* name
	Constant range

12.2 Static Attributes

For the purpose of description of attributes, following conventions apply:

- `T` denotes a Prefix that is a non-universal type,
- `V` denotes a Prefix that is either a value or a variable,
- `C` denotes a Prefix that is a constant value,
- `X` denotes a Prefix of any kind.

12.3 Dynamic Attributes

Semantics

Dynamic attributes represent run–time properties of objects. There is neither special syntax nor naming to choose a dynamic attribute in the Attribute construct. Static or dynamic form of an attribute is chosen according to the nature of the given Prefix. Following attributes can be dynamic:

- `:Component`,
- `:Last`,
- `:Length`,
- `:Ord`,
- `:Range`,
- `:Size`,
- `:Tag`,
- `:Unchecked`.

13. Metastatements

Syntax

Metastatement	#case metastatement
	#declared metastatement
	#template metastatement
	#display metastatement
	#environment metastatement
	#error metastatement
	#expand metastatement
	#if metastatement
	#include metastatement
	#option metastatement
	#pragma metastatement
	#representation metastatement
	#separate metastatement
	#syntax metastatement
	#warning metastatement

13.1 Conditional Compilation

Syntax

#if metastatement	**#if** Metacondition **then**; Text
	{ **#elsif** Metacondition **then**; Text }
	[**#else**; Text]
	#end if;
Metacondition	*Constant logical* expression { **#and then**; *Constant logical* expression }
	Constant logical expression { **#or else**; *Constant logical* expression }
#case metastatement	**#case** *Constant* expression;
	{ **#when** *Constant* choice **do**; Text }
	[**#when others do**; Text]
	#end case;
Text	{ Text element }
Text element	Identifier
	Numeric literal
	Character literal
	Character aggregate
	Keyword
	Symbol
	Metaidentifier
	Space
	Line end

	End of line comment
	Multiline comment
#declared metastatement	**#declared** Identifier;

13.2 Messages

Syntax

#display metastatement	**#display** *Constant* expression;
#warning metastatement	**#warning** *Constant* expression;
#error metastatement	**#error** *Constant* expression;

Semantics

The *Constant* expression denotes a message that shall be reported by the implementation after processing of the corresponding metastatement together with an apropriate position information. The type of the *Constant* expression shall be either string or unconstrained string of a character type or a *universal array of universal character*.

A #display metastatement causes the message to be just reported. The message of a #warning metastatement shall be reported as a warning and the message of a #error metastatement shall be reported as an error. Warnings and errors generated in this way shall be treated by the implementation as any other warnings and errors by default.

13.3 File Inclusion

Syntax

#include metastatement	**#include** Expression;

13.4 Separate Modules and Classes

Syntax

#separate metastatement	**#separate** [Part specification] Identifier;
Part specification	"public"
	private

13.5 Text Substitutions

Syntax

#template metastatement	**#template** Identifier [(Identifier {, Identifier })];
	Text
	#end Identifier;
#expand metastatement	**#expand** Identifier;
	{ **#for** Identifier **use**; Text }
	[**#for others use**; Text]
	#end Identifier;

13.6 Syntax Extension

Syntax

#syntax metastatement	**#syntax** Identifier; Text **#end** Identifier;

Semantics

The #syntax metastatement encloses program text written in a different programming language or any other textual notation specified by the Identifier. Processing of such program text is implementation defined. Normally this foreign program text should be integrated into the resulting program in an appropriate way.

Note: This can be a sequence of assembler instructions, an SQL statement, a binded resource, as well as a fragment of an XML document (e.g. XHTML).

Implementation Defined

The implementation may restrict usage of the #syntax metastatement for a particular language Identifier only to appropriate places within the source code (e.g. only to a place where a Statement can occur or within a #template metastatement.

13.7 Pragmas and Compilation Options

Syntax

#pragma metastatement	**#pragma** Identifier [(Pragma parameter {, Pragma parameter })];
#option metastatement	**#option** Identifier [(Pragma parameter {, Pragma parameter })];
Pragma parameter	*Constant* expression
	Entity name
	Identifier

Semantics

The #pragma metastatement is provided as an instrument to specify implementation defined characteristics of the program, that are typically associated with an entity (e.g. calling convention, import from a dynamic library etc.), and other actions not defined by the language itseft. The #option metastatement is provided as an instrument to set implementation defined options as part of the source text (e.g. verbosity of error messages or warning can be controlled through these options).

The Identifier represents the name of the pragma or option. For each pragma and option the implementation shall specify corresponding parameter profile (number, kinds and meanings of parameters).

Pragmas and options unknown to an implementation shall be ignored and should be reported as warnings to the user. No restrictions shall be applied to the parameter profile in this case.

13.8 Compilation Environment

Syntax

#environment metastatement	**#environment** Identifier;

Semantics

The #environment metastatement allows retrieval of value of certain property into the source text. These properties are called environment variables. The Identifier represents the name of an environment variable. The value of a given environment variable shall be substituted for the corresponding #environment metastatement.

The following table summarizes all language defined environment variables, their meanings and types (in alphabetical order):

Environment variable	Type	Meaning
`language_version`	*universal integer*	The version of the language to which the implementation conforms.
`compiler_name`	*universal array of universal character*	Trade name of the compiler or product.
`compiler_type`	*universal array of universal character*	A canonical name of the compiler.
`compiler_subtype`	*universal array of universal character*	A canonical name of a modification of the compiler or product (e.g. command-line compiler or IDE).
`compiler_version`	*universal array of universal character*	Version string of the compiler or product.
`compile_date`	*universal array of universal character*	Date in UTC time zone of the compilation start. The date shall be formated according to [ISO 8601], paragraph 5.2.1.1, Extended format (without time zone specification).
`compile_time`	*universal array of universal character*	Time in UTC time zone of the compilation start. The time shall be formated according to [ISO 8601], paragraph 5.3.1.1, Extended format (without time zone specification).
`compile_local_date`	*universal array of universal character*	Same as `compile_date`, but in local time zone.
`compile_local_time`	*universal array of universal character*	Same as `compile_time`, but in local time zone.
`target_os_name`	*universal array of universal character*	Trade name of the target operating system.
`target_os_family`	*universal array of universal character*	Target operating system's family's canonical name (e.g. Windows, Unix).
`target_os_type`	*universal array of universal character*	Type of the operating system within the family (e.g. NT, 9x or Linux, FreeBSD).
`target_os_version`	*universal array of universal character*	Version string of the target operating system.
`target_os_subsystem_type`	*universal array of universal character*	Type of the target subsystem (e.g. console or GUI).

Environment variable	Type	Meaning
`target_os_subsystem_version`	*universal array of universal character*	Version string of the target subsystem.
`target_machine_architecture`	*universal array of universal character*	Canonical name of the target machine's architecture (e.g. IA-32, IA-64, PPC).
`target_machine_bits`	*universal integer*	Length in bits of the target machine's native word.
`target_machine_direct_byteorder`	*universal logical*	True, if the target machine stores number in direct byte-order (e.g. LSB-first), False if in reverse byte-order (e.g. MSB-first)
`target_machine_version`	*universal array of universal character*	Version string of the target machine.
`host_os_name`	*universal array of universal character*	Same as corresponding `target_*` environment variable, but for the host system.
`host_os_family`	*universal array of universal character*	
`host_os_type`	*universal array of universal character*	
`host_os_version`	*universal array of universal character*	
`host_os_subsystem_type`	*universal array of universal character*	
`host_os_subsystem_version`	*universal array of universal character*	
`host_machine_architecture`	*universal array of universal character*	
`host_machine_bits`	*universal integer*	
`host_machine_direct_byteorder`	*universal logical*	
`host_machine_version`	*universal array of universal character*	

Recommended values and canonical names yielded by certain environment values can be found in [ENVREC].

13.9 Representation of Characters

Syntax

#representation metastatement	**#representation** *Source* character literal *Destination* character literal;

14. Macros

Syntax

Macro declaration	Incomplete macro declaration Macro declaration completion Full macro declaration
Incomplete macro declaration	**macro** Identifier [Subprogram heading] [Attribute specification];
Macro declaration completion	**macro** Identifier = Subprogram body;
Full macro declaration	**macro** Identifier [Subprogram heading] [Attribute specification] = Subprogram body;

14.1 Generic Type Definition

Syntax

Generic type definition	**generic**

14.2 In Statement

Syntax

In statement	**in** *Variable* name

14.3 Out Statement

Syntax

Out statement	**out** Expression

15. Optimization

16. Predefined Environment

Predefined Environment

The Language Reference Manual defines a set of declarations referenced to as a predefined environment. Parts of the predefined environment are defined throughout this Language Reference Manual. The purpose of the predefined environment is to give the user a specific set of entities that are guarantied to be accesible to the user's program.

Language design: The predefined environment is not language defined. A language defined entity is thought to be built-in to the implementation (defined by the compiler).

All parts of the predefined environment shall be enclosed within a single source file and shall be compiled before the first declaration or metastatement of the user program (or any its part).

Implementation Defined

The implementation should provide the full predefined environment as defined in this Language Reference Manual, but is not required so. There may be conditions under which it is eligible to omit parts of the predefined environment. The module `Flex` shall be provided even if its private part is empty.

The implementation shall provide the source code of the predefined environment to allow user modifications. The user may disable parts of the predefined environment except the module `Flex`, which must not harm compilation of the module `Flex` or cause any run–time malfunctions.

Appendices

Appendix A. Specialized Application Areas

A.1 System Programming

A.1.1 Machine Code Insertion

Implementation Defined

For a #syntax metastatement, the implementation may define the language identifier `Asm` for purposes of symbolic machine code insertion (assembler).

A.1.2 The `:Machine_Pointer` Attribute

Implementation Defined

For a pointer to object and pointer to subprogram *universal integer* types, the implementation should provide the attribute `:Machine_Pointer` of type *universal logical*. The default value of this attribute shall be `False`. Setting this attribute to `True` by the user shall omit all dynamic attributes from the pointer type being constructed leaving only the raw machine address of the target object or subprogram.

A.1.3 The `:Mapped_to_IO` and `:IO_Address` Attributes

Implementation Defined

For a static variable, the implementation may provide the `:Mapped_to_IO` and `:IO_Address` attributes. The attribute `:Mapped_to_IO` shall be of type *universal logical* with a default value `False`, the attribute `:IO_Address` shall be of type *universal integer* with a default value of zero. The implementation may introduce further restrictions on the variable's type and other properties.

Setting the attribute `:Mapped_to_IO` to `True` maps the corresponding variable to an external hardware IO bus'es data port addressed by the value of `:IO_Address`.

A.2 Secure Systems

A.2.1 The `:Secure` Attribute

Implementation Defined

For a variable or a type, the implementation may define the attribute `:Secure` of type *universal logical*. The default value of this attribute shall be `False`. When this attribute is set to `True` by the user, **nil** shall be assigned to the corresponding variable when the execution leaves its scope.

The implementation should report a warning when such variable or type is initialized by an initialization expression to any other value than **nil**.

Note: This attribute is provided to support automatic destruction of sensitive data in security related algorithms (e.g. passwords, encryption keys, personal data, etc.).

A.3 Conformance to The Common Language Infrastructure

Appendix B. Glossary

Regular type

Singular type

Unconstrained type is either an unconstrained array or an unconstrained string. Size of an object of an unconstrained type is not known at compile-time — the:size attribute is always dynamic.

Constrained type is every regular type that is not unconstrained. Size of an object of a constrained type is known at compile-time.

Tagged type is every explicitly declared type.

Argument	The word argument is used as a synonym to Formal Parameter and Actual Parameter with respect to the context of the actual usage.
Declaration	A declaration is a language construct that associates a name with an entity. A declaration may appear explicitly in the program text (an explicit declaration), or may be supposed to occur at a given place in the text as a consequence of the semantics of another construct (an implicit declaration).
Declarative region	A declarative region is a part of a language construct that can contain declarations. Every declarative region is associated with an eclosing entity declaration (either implicit or explicit) and forms a logical namespace for entities declared immediately within this declarative region. More declarative regions can be associated with a single entity declaration.
Formal Parameter	A formal parameter is an entity representing within a subprogram a value passed to and/or returned from that subprogram. One formal parameter can be declared as a return parameter (see definition).
Parameter Profile	A parameter profile of a subprogram is the actual list of properties of formal parameters and the optional return parameter regardless of any type of the subprogram or identifiers of arguments — that is the list of combinations of <mode, control, type> of all formal parameters and the optional return parameter. This term is used rather informaly especially when speaking about counts of parameters and its properties.
Return Parameter	A return parameter is a selected formal parameter that is used to return value from a subprogram. This value is interpreted as a result of a Call.
Specification	A specification is a part of every (explicit) declaration which defines basic properties of an entity (such as its name, type, formal parameters etc.) that are needed for proper usage of that entity. A specification containing only the entity's name (Identifier) is a trivial specification. Syntactically a specification forms the first part of a declaration up to a "=" or end of declaration.
definition	A definition of an entity defines the all other properties — internals of that entity — (such as subcomponents, statement part, etc.) that are not defined by the entity's specification. Syntactically a definition follows (if present in a declaration) the corresponding specification and is separated by a "=".

Appendix C. Syntax Summary