Sparkel Reference Manual – Draft 0.7

S. Tucker Taft, AdaCore

October 23, 2023

Contents

Chapter 1

Introduction

Sparkel is a parallel version of the SPARK language, and is designed with the principle that if you want programmers to write parallel algorithms, you have to immerse them in parallelism, and force them to work harder to make things sequential. In Sparkel, parallelism is everywhere, and threads are treated as resources like virtual memory – a given computation can use 100s of threads in the same way it might use 100s of pages of virtual memory. Sparkel encourages the *divide-and-conquer* approach to parallel computation where threads are each given their own part of the problem to solve, but also supports concurrent access to *protected* objects, using both lock-based and lock-free synchronization.

Sparkel also supports annotations, such as preconditions and postconditions, and in fact requires them in some cases if they are needed to ensure that a given operation is safe. In particular, all checks that might normally be thought of as run-time checks (if checked by the language at all) are performed at compile time in Sparkel. This includes uninitialized variables, array index out of bounds, null pointers, race conditions, numeric overflow, etc. If an operation would overflow or go outside of an array given certain parameters, then a precondition is required to prevent such parameters from being passed to the operation. Sparkel is designed to support a *formal* approach to software design, with a relatively static model to simplify proving properties about the software, but with an explicit ability to specify run-time polymorphism where it is needed.

Sparkel has four basic concepts – (Generic) Packages, Types, Objects, and Operations. Every type is a private type, a record type, or an operation type. An object is an instance of some type. An operation operates on objects.

The only global variables allowed are those of a protected type. Any object to be updated by an operation must be an explicit `var` parameter to the operation, or be identified as a `global var` to the operation.

Sparkel has no pointers, though it has references, optional and expandable objects, and syntactic sugar for indexing (see below for a further discussion of Syntactic Sugar), which together provide a rich set of functionally equivalent capabilities without any hidden aliasing nor any hidden race conditions.

1.1 Language Design Principles

Below are some of the fundamental language design principles we tried to follow while designing Sparkel. Of course, at times we faced a conflict, so at those times tradeoffs had to be made. Although these are expressed as goals, by and large we believe they have been accomplished in the current design.

- The language should be easy to read, and look familiar to a broad swath of existing programmers, from the ranks of programmers in the Algol/Pascal/Ada/Eiffel family, to the programmers in the C/C++/Java/C# family, to the programmers in the ML/Haskell and Lisp/Scheme communities. Readability is to be emphasized over terseness, and where symbols are used, they should be familiar from existing languages, mathematics, or logic. Although extended character sets are more available

these days, most keyboards are still largely limited to the ASCII, or at best, the Latin-1, character set, so the language should not depend on the use of characters that are a chore to type.

Programs are often scanned backward, so ending indicators should be as informative as starting indicators for composite constructs. For example, "end loop" or "end class Stack" rather than simply "end" or "}".

- Parallelism should be built into the language to the extent that it is more natural to write parallel code than to write explicitly sequential code, and that the resulting programs can easily take advantage of as many cores as are available on the host computer.

- The language should have one primary way to do something rather than two or three nearly equivalent ones. Syntactic sugar (see section below) should be used to provide higher-level constructs, while keeping the core of the language minimal. Nonessential features should be eliminated from the core language, especially those that are error prone or complicate the testing or proof process. User-defined types and language-defined types should have the same capabilities.

- All code should be parameterizable to some extent, since it is arguable that most code would benefit from being parameterized over the precision of the numeric types, the character code of the strings involved, or the element types of the data structures being defined. In other words, any module can be a generic template or equivalent. But the semantics should be defined so that the parameterized modules can be fully compiled prior to being instantiated.

- The language should be inherently safe, in that the compiler should detect all potential race conditions, as well as all potential runtime errors such as the use of uninitialized data, out of bounds indices, overflowing numeric calculations, etc. Given the advances in static analysis, there is no reason that the compiler cannot detect all possible sources of run-time errors.

Programming is about human programmers clearly and correctly communicating with at least two audiences: 1) other human programmers, both current and future, and 2) a very literally-minded machine-based compiler or interpreter. What is needed is *human engineering*, which is the process of adapting a technology to be most useful to humans, by minimizing opportunities for errors, taking advantage of commonly understood principles, using terminology and symbols consistently and in ways that are familiar, and eliminating unnecessary complexity.

Here are some additional somewhat lower level principles followed during the Sparkel design:

- Full generality should be balanced against testability and provability. In particular, though passing functions and types as parameters is clearly useful, it is arguable whether full upward closures and types as true first-class objects (such as the *class* objects in *Smalltalk*), are useful enough to justify the significant testing and proof burdens associated with such constructs. The more disciplined packaging of type and function provided by statically-typed object-oriented programming can match essentially all of the capability provided by upward closures and types as first-class objects, while providing, through behavioral subtyping and other similar principles, a more tractable testing and proof problem.

- Avoid constructs that require fine-grained asynchronous garbage collection if possible. Garbage collectors are notoriously hard to test and prove formally, and are made even more complex when real-time and multi-processor requirements are added. Mark/release strategies, and more generally region-based storage management, as in the *Cyclone* language, suggest possible alternative approaches.

- Mutual exclusion and waiting for a condition to be true should be automatic as part of calling an operation for which it is relevant. This is as opposed to explicit lock/unlock, or explicit wait/signal. Automatic locking and/or waiting simplifies programming and eliminates numerous sources for errors in parallel programs with inter-thread synchronization. The result is also easier to understand and to prove correct.

1.2 Syntactic Sugar

Rather than building in many fundamentally different kinds of types and type constructors, such as enumeration types, array types, fixed-point types, etc., and many different constructs such as concatenation, indexing, and aggregates (e.g. for arrays or other containers), Sparkel uses the notion of *syntactic sugar* to transform these higher-level concepts into the core capabilities of the language. This syntactic-sugar approach allows great flexibility and extensibility, while keeping the core capabilities of the language very simple.

Within the reference manual, *Syntactic Equivalences* sections indicate where syntactic sugar is applied. These sections can be skipped in the initial reading of the reference manual.

1.3 Relationship between Sparkel and SPARK, Ada, and ParaSail

Describing Sparkel as a "parallel version of SPARK" is not the whole story. Sparkel is a language in its own right, starting from the SPARK subset of Ada as a set of basic capabilities, and then incorporating ideas from the ParaSail parallel programming language, a language designed from scratch to support safe and secure parallel programming.

Sparkel is designed to be a lean yet widely applicable parallel programming language, while preserving a formal approach to safety and security. Unlike SPARK, Sparkel does not limit its run-time semantics to be a subset of those of the Ada language. Sparkel preserves many of the restrictions of SPARK, while opening up the language in other ways to make the language easier to use and more appropriate for building entire applications, rather than simply the secure kernel of a system.

One way to describe Sparkel is with the following four somewhat ironic characteristics:

- *Mutable objects* with value semantics;
- *Stack-based* heap management;
- *Compile-time* exception handling;
- *Race-free* parallel programming.

These characteristics mean that Sparkel remains statically analyzable while providing a flexible, lean, easy to use parallel language.

1.3.1 Ada Features omitted from Sparkel

Relative to Ada, Sparkel leaves out the following features (many of these are also omitted from SPARK):

Discriminants – though Sparkel has `const` record components; syntactic sugar supports the syntax of a `discriminant_part`;

Tagged types – any record or private type can be extended – a run-time *type-id* appears only on polymorphic objects;

Tasks and task types – we are considering having a `"begin"` operator for protected types, which would be called by a pico-thread when the object is created, and would be automatically terminated when the scope exits;

Controlled types – we are considering having an `"end"` operator for protected types, which would be called immediately prior to the object being reclaimed;

Entry families – any parameter to a `queued` operation may be used in a dequeue condition;

Constrained vs. Unconstrained subtypes/objects – Sparkel objects can store any value of their subtype – predicates may be used to limit what values are permitted; in particular, arrays can change in length at run-time in Sparkel;

Interface types – any private type can be used as an interface to be implemented by another type;

Use (all) type clause – all operations of a type are implicitly visible for calls (and for using as values of an operation type);

Exceptions – though preconditions, null values, or multi-threaded exits/returns can substitute;

Access types, allocators, and aliased objects – short-lived references identified by `ref` can provide for some of this; optional objects and generalized indexing provide other capabilities to eliminate the need for (re)assignable pointers;

Generic formal subprograms – in Sparkel, objects of an operation type may be used instead – syntactic sugar is provided to mimic generic formal subprograms.

1.3.2 Features added to Sparkel

Relative to SPARK 2014, Sparkel has the following additional features:

Optional objects and null values – every type in Sparkel includes a *null* value, which may be stored in objects marked as `optional`; these allow objects to grow and shrink – see 3.2;

Instantiation of generic types – a type declared inside a generic package is termed a *generic type* and may be directly instantiated in Sparkel, rather than requiring that the enclosing package first be explicitly instantiated; – see 3.1;

Short-lived references – a short-lived `ref` to an existing object may be established in Sparkel; these are used to support user-defined indexing and slicing of container objects such as arrays and maps, and are also used to support iterating through a structure such as a tree or a graph; these are analogous to Ada's object `renames` but can be rebound as part of loop iteration; – see 3.3 and 5.6;

Implicitly parallel semantics – all expression evaluation in ParaSail has implicitly parallel semantics, in that expressions such as `F(X) + G(Y)` allow evaluation of `F(X)` and `G(Y)` in parallel with one another, and statement and loop semantics are designed to simplify automatic parallelization – see 10.2;

Explicitly parallel constructs – any `for` loop may be explicitly specified as `parallel` and statements may be separated by `||` rather than `";"` to indicate that parallel evaluation is expected, and the compiler should complain if the constructs cannot be safely executed in parallel – see 5.1 and 5.6;

Polymorphic types – the notation `T+` is used to refer to the type T and any type that extends or implements all of the operations expected for T; objects of a polymorphic type carry a run-time *type-id* which identifies their underlying non-polymorphic type; these are analogous to Ada's class-wide types – see 7.2.1;

Protected objects and locked and queued operations – within a *protected* type in Sparkel, operations may be identified as `locked` or `queued` with a parameter identified as the object which is being locked as a side-effect of a call on the operation; in the case of a `queued` operation, an arbitrary *dequeue* condition is specified that determines how long the caller will be suspended before executing the operation; these represent a generalization of Ada's protected types – see Chapter 10;

User-defined literals – any type may use literals, so long as it has a `"from_univ"` operator for the appropriate Univ_ type – see 2.4, 4.2, and 4.2.5;

Iterators in aggregates; Map-Reduce expressions – through the use of *syntactic sugar*, Sparkel supports a number of constructs for iterating over a data structure or sequence of values as part of constructing a new value – see 4.2.9 and 8.2.

Lambda expressions and Operation types – operations may be passed as parameters to other operations (analogous to Ada's access-to-subprogram types); values of an operation type may be specified by naming a declared operation which has a compatible parameter profile, or by giving a `lambda` expression which specifies the computation to be performed – see 6.5;

Generalized Case Statements – case statements are generalized to support selecting based on the underlying type of a polymorphic object, as well as more generally any sort of set membership – see 5.4;

Private types completed in body – private types are given their full definition in the body of a package rather than in a private part of the package spec.

1.4 History of Revisions

Draft 0.1 Initial revision.

Draft 0.2 Editorial fixes; Operation Types defined; Skip_List example fixed.

Draft 0.3 Incorporates the following changes:

- Add Revision History and To-Do-List Appendix.
- Define *structural equivalence* for operation types.
- Include a `value_filter` in a map-reduce expression.
- Allow for possibility of `record_aggregate` with no components.
- Fix use of `object_type` to use `subtype_indication` instead.
- Introduce `loop_variable_initializer` to fix bug in `initial_next_while_iterator` syntax.
- Provide Syntactic Equivalences for quantified and map-reduce expressions.
- Provide Syntactic Equivalences for the various iterator forms in terms of the `initial_value_iterator`.
- Provide Syntactic Equivalences for container aggregates in terms of individual operator calls.
- Provide explicit comparisons between Sparkel, Ada, and SPARK.
- Allow `'#'` notation for enumeration literals, and thereby simplify the Syntactic Equivalence required for enumeration-type definitions.
- Allow discriminant parts via Syntactic Equivalence.
- Allow 'when' condition on any exit statement.
- Attempt to explain the statement separators more clearly.
- Indicate where the Syntactic Equivalences are for compatibility with existing SPARK code.

Draft 0.4 Incorporates the following changes:

- Fix typographical error in syntactic expansion for an enum type with a representation clause.
- Fix typographical error in description of alternate location of `direction` within a `set_iterator`.
- Change use of "op" to be "func" or "proc" as appropriate.
- Define Enum_Types and Enum_Types_With_Rep in the Standard Library based on their new usage pattern.

- Trim the to-do list appropriately.
- Allow "end func F", "end proc P", and "end package Q" for consistency.
- Explain the expansion of a `type_name` into `type_name'[..]` possibly intersected with the Predicate.
- Mention lack of private part in a package (TBD: should we allow a private part, and private child units that can see it, or just require nested packages to support the equivalent?)
- Allow `container_specifier`s on formal parameters. Define the semantics in the Container Specifiers section. Move `container_specifier` to follow the `subtype_indication`, if any, so it can be used when the type name is omitted on an object decl, and when the parameter name is omitted on a parameter spec.

Draft 0.5 Incorporates the following changes:

- Minor rewording of syntactic equivalence for a discriminant part.

Draft 0.6 Incorporates the following changes:

- More complete description of container aggregate syntactic equivalences.

Draft 0.7 Incorporates the following changes:

- Describe notion of *primary nested type* and use it throughout so that a package may have the same name as its primary nested type, allowing simply "List" rather than "Lists.List" in most cases. Fix name of standard integer and float types to correspond to current implementation state, namely Integer and Float.

Chapter 2

Lexical Elements

2.1 Character Set

Sparkel programs are written using graphic characters from the ISO-10646 (Unicode) character set, as well as horizontal tab, form feed, carriage return, and line feed. A line feed terminates the line.

```
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

binary_digit ::= 0 | 1

hex_digit ::= digit | A..F | a..f

extended_digit ::= digit | A..Z | a..z
```

2.2 Delimiters

The following single graphic characters are delimiters in Sparkel:

```
( ) { } [ ] , ; . : & | = < > + - * / ’ ?
```

The following combinations of graphic characters are delimiters in Sparkel:

```
|| != =? <= >=
==> ** => [[ ]] << >>
:= <== <=> <|= += -= *= /= **= <<= >>= |= &=
.. <.. ..< <..<
```

The following combinations of graphic characters have special significance in Sparkel:

```
and= or= xor=
```

2.3 Identifiers

Identifiers start with a letter, and continue with letters or digits, optionally separated by underscores.

```
identifier ::= letter { [ _ ] ( letter | digit ) }
```

Upper and lower case is significant in identifiers, but two identifiers that differ only in case hide one another. Letters include any graphic character in the ISO-10646 character set that is considered a letter. An all-lower-case identifier must not be the same as a Sparkel reserved word (see 2.6).

Examples:

```
X, A_B, a123, A123, This_Is_An_Identifier, Xyz_1
```

NOTE: We are proposing to make upper/lower case significant in identifiers, but have hiding (or more generally overload resolution) ignore case distinctions, to avoid slightly incorrect upper/lower case resulting in different resolution. This also eliminates many compatibility issues with new reserved words, since reserved words only conflict with identifiers which are in all lower case, which is a rarity in existing Ada and SPARK code.

2.4 Literals

There are six kinds of literals in Sparkel: integer, real, character, string, enumeration, and `null`. The syntax for these literals is given below.

```
literal ::=
    integer_literal
  | real_literal
  | character_literal
  | string_literal
  | enumeration_literal
  | null_literal
```

2.4.1 Integer literals

Integer literals are by default decimal. Integers may also be written in binary, hexadecimal, or with an explicit base in the range 2 to 36.

Integer literals are of type Univ_Integer.

```
integer_literal ::=
    decimal_integer_literal
  | based_integer_literal
```

[*Possible addition:*

```
    | binary_integer_literal
    | hex_integer_literal
```

]

```
decimal_integer_literal ::= decimal_numeral

based_integer_literal ::= decimal_numeral # extended_numeral #
```

[*Possible addition:*

```
binary_integer_literal ::= 0 (b|B) binary_digit { [_] binary_digit}

hex_integer_literal ::= 0 (x|X) hex_numeral
```

]

```
decimal_numeral ::= digit { [_] digit }

hex_numeral ::= hex_digit { [_] hex_digit}

extended_numeral ::= extended_digit { [_] extended_digit }
```

Examples: `42, 1_000_000, 8#0177#` [, *Possible addition:* `0xDEAD_BEEF`]

2.4.2 Real Literals

Real literals are by default decimal, with an optional decimal exponent indicating the power of 10 by which the value is to be multiplied. Reals may also be written with an explicit base in the range 2 to 36, with a decimal exponent indicating the power of the base by which the value is to be multiplied.

Real literals are of type Univ_Real.

```
real_literal ::= decimal_real_literal | based_real_literal

decimal_real_literal ::= decimal_numeral . decimal_numeral [exponent]

based_real_literal ::=
  decimal_numeral # extended_numeral . extended_numeral # [exponent]


exponent ::= (e|E)[+|-] decimal_numeral
```

Examples:

```
3.14159, 0.15, 16#F.FFFF_FFFF_FFFF#e+16
```

2.4.3 Character Literals

Character literals are expressed as a pair of apostrophes bracketing a single `unescaped_character`, being any graphical character of the ISO-10646 character set [*Possible addition:* other than backslash, or a single `escaped_character`, being a backslash followed by an `escapable_character` or a hexadecimal character code].

Character literals are of type Univ_Character.

```
character_literal ::= ' character_specifier '

character_specifier ::=
     unescaped_character
```

[*Possible addition:*

```
   | escaped_character

  escaped_character ::= \ escapable_character | \ # hex_numeral #

  escapable_character ::= \ | ' | " | n | r | t | f | 0
```

The following escapable characters have the following interpretation when preceded by \:

```
\  -- backslash
'  -- apostrophe
"  -- double quote
n  -- line feed
r  -- carriage return
t  -- horizontal tab
f  -- form feed
0  -- Nul
```

A character literal of the form `'\#hex_numeral#'` specifies the character whose ISO-10646 code is equal to the value of the given `hex_numeral`. *end of Possible addition*]

Examples:

`'a', '0'` [, *Possible addition:* `'\'', '\r', '\#03_C0#'`]

2.4.4 String Literals

String literals are a sequence of graphical characters of the ISO-10646 character set enclosed in double quotes. Two double quotes in a row represent a single double quote within the string. [*Possible addition:* The backslash character may appear only as part of an escaped_character.]

String literals are of type Univ_String.

```
string_literal ::= " { character_specifier | "" } "
```

Example:

```
"A simple string literal"
```

[*Possible addition:*

```
"This is a multiline message\n and this is the second line."
```

]

2.4.5 Enumeration Literals

Enumeration literals are expressed with a # followed by an identifier or reserved word. In the case of an identifier, the # may be omitted in a context where a precondition identifies the subset of enumeration literals which are allowed (as when passed to a `"from_univ"` operator – see 4.2 and 4.2.5).

Enumeration literals are of type Univ_Enumeration.

```
enumeration_literal ::= [ # ] identifier | # reserved_word
```

Examples:

```
#case, True, Monday
```

2.4.6 Null Literals

Null literals are specified with the reserved word `null`.

```
null_literal ::= null
```

A null_literal represents the null value of the type that is determined by context.

2.5 Comments

Comments in Sparkel start with `--` and continue to the end of the line.

Examples:

```
-- According to the Algol 68 report,
-- comments are for the enlightenment of the human reader.
```

2.6 Reserved Words

The following words are reserved in Sparkel:

`abs`	`entry`	`mod`	`ref`
`abstract`	`exit`	`new`	`rem`
`all`	`for`	`not`	`return`
`and`	`forward`	`null`	`reverse`
`assert`	`func`	`of`	`separate`
`begin`	`function`	`optional`	`some`
`case`	`global`	`or`	`then`
`const`	`if`	`package`	`type`
`continue`	`in`	`parallel`	`until`
`declare`	`is`	`private`	`var`
`each`	`lambda`	`proc`	`when`
`else`	`limited`	`procedure`	`while`
`elsif`	`locked`	`protected`	`with`
`end`	`loop`	`queued`	`xor`

All reserved words in Sparkel are in lower case.

Chapter 3

Types and Objects

In Sparkel, every *object* is an instance of some *type*, and every type is either a *private type*, a *record type*, an *operation type*, or an *instantiation* or *derivation* of one of these. In addition, various qualifications and aspect specifications may be applied to the type to produce a particular *subtype* of the type.

3.1 Types and Subtypes

A type is declared as follows:

```
type_declaration ::=
  'type' identifier 'is' ['abstract'] type_definition [ aspect_specification ]

type_definition ::=
    record_type_definition
  | private_type_definition
  | operation_type_specifier
  | type_derivation

private_type_definition ::= ['limited' | 'protected']  'private'

type_derivation ::= 'new' type_specifier_list | type_extension

type_extension ::=
    'new' type_specifier_list 'with' record_type_definition
  | 'new' type_specifier_list 'with' private_type_definition

type_specifier_list ::= type_specifier [ { 'and' named_type_specifier } ]

type_specifier ::=
  named_type_specifier | anon_record_type_specifier | operation_type_specifier

named_type_specifier ::= type_name | type_instantiation

type_instantiation ::= type_name '<' [ generic_actuals ] '>'
```

See 3.1.1 for the syntax of a `record_type_definition` and `anon_record_type_specifier`. See 6.2 for the syntax of an `operation_type_specifier`. See 4.1 for the syntax of a `type_name`.

A subtype is declared as follows:

```
subtype_declaration  ::= 'subtype' identifier 'is' subtype_indication

subtype_indication    ::= ['optional'] type_specifier [ aspect_specification ]
```

See chapter 9 for the syntax of an aspect_specification.

A `type_declaration` introduces a new named type. If the declaration occurs within a generic package, the type is called a *generic* type. If the type has the same name as the enclosing package, it is the *primary nested type* of the package, and from outside the package, a name that denotes the package also denotes this type, providing a *short-hand* reference for the type. A `subtype_declaration` introduces a name for a renaming or subtype of an existing (sub)type. A `type_specifier` specifies a type using an `anon_record_type_specifier` (see 3.1.1 below), an `operation_type_specifier` (see 6.2), or by specifying a `type_name` that was declared by a `type_` or `subtype_declaration`, possibly providing generic actual parameters if the specified type is generic. A `subtype_indication` specifies a subtype of the type determined by the `type_specifier`, with possible additional qualifications and aspect specifications.

In a `type_derivation`, the first `type_specifier` identifies the *parent* of the new type, and the new type is *derived* from this parent, and *inherits* operations and components from this type (see Inheritance). Any additional `named_type_specifier`s identify *progenitors* of the new type, and the new type must *implement* the (non-optional) operations of these progenitors; it does not inherit the implementation of any operations nor any components from its progenitors (even if they have components). It inherits code and components only from its parent type.

Two `type_specifier`s identify the *same* type if and only if they are defined by structurally equivalent `anon_record_type_specifier`s (see 3.1.1 below) or `operation_type_specifier`s (see 6.2), or they refer to the same original `type_definition` and specify *equivalent* `generic_actuals`, if any.

Example:

Given a generic package List providing a List type defined as follows (see 7.1):

```
generic
    type Element_Type is private;
package List is
    type List is private;
    func Create return List;
    func Is_Empty(L : List) return Boolean;
    proc Append(var L : List; Elem : Element_Type);
    func Remove_First(var L : List) return optional Element_Type;
    func Nth_Element(ref L : List; N : Univ_Integer)
      return ref optional Element;
end List;
```

Note that in this example, type List is the *primary nested type* of the package List. A specific kind of list may be declared as follows:

```
type Bool_List is new List < Boolean >;
```

This declares a Bool_List type which represents a list of Booleans.

3.1.1 Record Types and Extensions

Most user-defined types in Sparkel are ultimately record types. A (named) record type or record extension can be defined using a `record_type_definition` within a `type_declaration`, according to the following syntax:

```
record_type_definition ::=
   [ 'limited' | 'protected' ] 'record'
     { component_specification ';' }
   'end' 'record' [ identifier ]
```

```
component_specification ::=
     component_mode [ identifier ':' ] subtype_indication [ ':=' expression ]

component_mode ::= [ 'ref' ] [ 'const' ]
```

An object of a *limited* type may be assigned a value only as part of its declaration; no subsequent assignments to the object as a whole are permitted (though assignments to individual non-limited components *are* permitted). A type is limited if it has a `ref` component, has a component of a limited type, or has the reserved word `limited` or `protected` in its definition.

A `record_type_definition` with a `ref` or limited component is required to be explicitly specified as either `limited` or `protected`.

If an identifier is omitted from a `component_specification`, the identifier of the (sub)type specified in the `subtype_indication` is presumed, if it is unique. If this identifier is not unique among the set of identifiers for the components, the component is anonymous.

Rather than declaring a named record type, an *anonymous* record type (also called a *tuple* type) may be specified using an `anon_record_type_specifier`:

```
anon_record_type_specifier ::=
  '(' component_specification { ';' component_specification ')'
```

An anonymous record type is *limited* if it contains a `ref` or limited component. Two anonymous record types are *structurally equivalent* if they have the same number of components with the same component modes and types. They need not have the same component names or default expressions, if any.

A `const` or `ref const` component is read only. Other components are writable if the enclosing record object is writable. Note that a `ref` component refers to a preexisting *target* object, which necessarily outlives the record object containing the ref.

Implementation note: wrapping a component in a record type should not increase the overall size of objects of the type. Although all types may be extended, normal types do not have *run-time tags*. Only *polymorphic* types have run-time tags (called *type-ids* in Sparkel). See 7.2.1.

3.1.2 Syntactic Equivalences

The declaration of a named record type is equivalent to a derivation from an anonymous record type:

```
   type R is record
      A : Integer;
      B : Float;
   end record R;
-- is equivalent to:
   type R is new (A : Integer; B : Float);
```

A type declaration for a private or record type may specify a `discriminant_part` which is equivalent to extending from an anonymous record type that has `const` components:

```
type_declaration ::=
  'type' identifier discriminant_part 'is'
     ['abstract'] type_definition [ aspect_specification ]

discriminant_part ::= anon_type_specifier
```

```
   type T(A : Integer; B : Boolean) is private;
-- expands into:
   type T is new (const A : Integer; const B : Boolean) with private;
```

There are several additional syntactic constructs provided for various language-provided generic types. These are defined in terms of the core syntax as follows:

```
type_definition ::=
    enumeration_type_definition
  | integer_type_definition | float_type_definition | fixed_type_definition
  | array_type_definition

type_declaration ::= protected_type_declaration

 enumeration_type_definition ::= '(' identifier {',' identifier } ')'

       type E is (A, B, C)
   -- expands into:
       type E is new Enum <[#A, #B, #C]>

       type E is (A, B, C); for E use (1, 2, 4)
   -- expands into:
       type E is new Enum_With_Rep
         <[#A => 1, #B => 2, #C => 4]>

 integer_type_definition ::= 'range' expression '..' expression

       type Int is range 1 .. 100
   -- expands into:
       type Int is new Integer <1 .. 100>;

 float_type_definition ::= 'digits' expression [ 'range' expression '..' expression ]

       type Flt is digits 5 range -10.0 .. 10.0
   -- expands into:
       type Flt is new Float <Digits => 5, Range => -10.0 .. 10.0>;

 fixed_type_definition ::= 'delta' expression [ 'range' expression '..' expression ]

       type Fix is delta 0.01 range -10.0 .. 10.0
   -- expands into:
       type Fix is new Fixed <Delta => 0.01, Range => -10.0 .. 10.0>;

 array_type_definition ::=
   'array' '(' index_subtype_definition{',' index_subtype_definition} ')' 'of' subtype_specifier

       type Enrollment is array (Course, Semester) of T;
   -- expands into:
       type Enrollment is new Array <T, Indexed_By => (Course; Semester)>>
                             -- note use of anon_record_type_specifier here

 protected_type_declaration ::=
   'protected' 'type' identifier 'is'
     { protected_operation_declaration }
   'private'
     { protected_element_declaration }
   'end' identifier
```

```
      protected type PT is
         func F return A;
         proc P (X : Integer);
         entry E (B : Boolean);
      private
         D : T;
      end PT;
   -- expands into:
        type PT is protected private;
        func F (locked PT) return A;
        proc P (locked var PT; X : Integer);
        proc E (queued var PT; B : Boolean);
      private
        type PT is protected record
           D : T;
        end record;
```

3.2 Objects

Objects contain data, and may either be variables (declared with 'var'), allowing their data to be changed after initialization, or constants (declared with 'const'), meaning the value of the data of the object cannot be changed between its initial and final reference.

An object is declared using the following syntax:

```
object_declaration ::=
    uninitialized_object_declaration
  | initialized_object_declaration

uninitialized_object_declaration ::=
  var_or_const identifier ':' subtype_indication [ container_specifier ] ';'

initialized_object_declaration ::=
    var_or_const identifier [ ':' subtype_indication ] [ container_specifier ] ':=' expression ';'
  | var_or_const identifier [ ':' subtype_indication ] '<==' object_name ';'

var_or_const ::= 'var' | 'const'
```

See section 8.4 for syntax of `container_specifier`.

If an object is declared with a `subtype_indication` specifying `'optional'`, its value may be any value of its type that satisfies the requirements of the subtype, as well as the *null* value of its type. An object not declared `optional` must be explicitly initialized before use to a non-null value satisfying the requirements of its subtype. The last use of a non-`optional` object (including a constant) may nevertheless be a *move* operation (see 5.2), leaving its final value null.

An uninitialized object has the null value initially, but this value can be read only if the object is declared `optional`. If the uninitialized object is an `optional` *constant*, this initial null value may be read, but the constant may not then be explicitly assigned a value on the path where the null value is read – the general rule is that two reads of the same constant always return the same value.

When an object is initialized using the `'<=='` *move* operation (as opposed to the ':=' *assign* operation – see 5.2 Assignment Statements), the initial value comes from an existing object (identified by an `object_name`). This value is *moved* into the new object, and the existing object is set to the `null` value as a side-effect. The

now null existing object denoted by `object_name` may be subsequently referenced only if the named object is declared `'optional'`.

Note: as a general rule, if an object is *not* declared `optional`, then its value might nevertheless be null before its first assignment and after its last use, but this null value can never be read.

Examples:

```
var BL : Bool_List := Create;
const T : Boolean := True;
var Result : optional T;
var Next <== Tree.Left;
```

These declare a variable boolean list, a constant with Boolean value True, a variable Result with implicit initial value of null, and a variable Next initialized by moving the value from Tree.Left, leaving Tree.Left null.

3.2.1 Syntactic Equivalences

For compatibility with existing SPARK code, the following syntactic equivalences are supported:

```
object_declaration ::=
      identifier ':' subtype_indication [ ':=' expression ]

         V : T := Expr
        -- expands into:
         var V : T := Expr

    | identifier ':' 'constant' subtype_indication [ ':=' expression ]

         C : constant T := Expr
        -- expands into:
         const C : T := Expr

    | identifier ':' 'constant' ':=' expression

         K : constant := Int_Expr
        -- expands into:tuple
         const K : Univ_Integer := Expr

         K : constant := Real_Expr
        -- expands into:
         const K : Univ_Real := Expr
```

NOTE: We are proposing to use `var X : T` *and* `const X : T` *for object declarations (instead of* `X : T` *and* `X : constant T`*) as part of our plan to eliminate the need for* `declare` ... `begin` ... `end` *and allow the interspersing of declarations and statements within a subprogram. In addition, we are supporting more type inference in object declarations, and the current syntax for variables doesn't easily allow for that (since if you leave out the type it looks like a regular assignment statement), and the current syntax for named numbers causes the inferred type to always be Univ_Integer or Univ_Real, which is not what we want in general.*

3.3 Object References

A reference to an existing (*target*) object is declared using the following syntax:

```
object_reference_declaration ::=
  'ref' [ var_or_const ] identifier [':' type_specifier ] '=>' object_name ';'
```

A variable reference is only permitted to a variable object. A constant reference provides read-only access to an object, whether or not the object itself is a constant. A reference not specified as `var` or `const` allows the same access as that provided by the object to which it refers.

Examples:

```
ref const Left => L.Left_Subtree;
ref var X => M[I];
ref Max => First_Element(A);
```

These create a read-only reference to the Left_Subtree component of L, a read-write reference to the Ith element of M (which must be a variable – see 8.1), and a reference to the first element of A, which is a read-write reference only if A is a variable. Note that in the third example, it is assumed that the First_Element function takes a `ref` parameter and returns a `ref` result (see 6.1);

3.3.1 Syntactic Equivalences

For compatibility with existing SPARK code, the following syntactic equivalence is supported:

```
object_reference_declaration ::=
  identifier ':' subtype_indication 'renames' object_name ';'
```

```
  B : Integer renames C.A
-- expands into:
  ref B : Integer => C.A
```

NOTE: We use `ref` consistently in Sparkel rather than `access` or `aliased` or `renames`, all of which are ways to create aliasing in Ada. The key differences between `ref` and anonymous access values in Ada is that they are implicitly dereferenced (no need for `.all`), they don't require `'Access` on creation, and they cannot be changed to designate a new target object (which is true in Ada for access parameters and access discriminants, but not true for stand-alone access objects in Ada 2012). A Sparkel `ref` is quite close to the notion of an `aliased` parameter in Ada 2012, and to a *reference* in C++.

To emphasize the difference relative to non-ref objects, the *target* of a ref is specified by using `=>` rather than `:=`. When a ref appears on the left hand side of an assignment (or in an expression), it is referring to the *content* of its target object, not to the ref itself, so it is syntactically impossible to change what is the target of a ref using an assignment statement. Note that within a `loop` statement, a new target for a ref can be established as part of starting a new iteration of the loop (see 5.6 and 5.6.1).

Chapter 4

Names and Expressions

4.1 Names

Names denote packages, types, objects, and operations.

```
name ::= package_name | type_name | object_name | operation_name

package_name ::= [ package_name '.' ] identifier

type_name ::= [ package_name '.' ] identifier  [ '+' ]

object_name ::=
    [package_name '.' ] identifier
  | object_indexing_or_slicing
  | operation_call
  | component_selection
```

See Operation Calls (Section 6.4) for the syntax of `operation_name` and `operation_call`. See Object Indexing and Slicing (Section 8.1) for the syntax of `object_indexing_or_slicing`.

4.1.1 Component Selection

If an object is of a record type, or of a type with one or more record extensions, then the components defined by the record type or the record extensions may be named using a `component_selection`. Components are named by naming the enclosing object, then a '.', and then the identifier of the component:

```
component_selection ::= object_name '.' identifier
```

Examples:

```
C.Real_Part, Point.X, List_Node.Next, T.Right_Subtree
```

4.2 Expressions

```
expression ::=
    literal
  | object_name
  | initial_value_specification
```

```
    | unary_operator expression
    | expression binary_operator expression
    | membership_test
    | null_test
    | quantified_expression
    | type_conversion
    | type_name
    | [ type_name ''' ] bracketed_expression

  bracketed_expression ::=
      aggregate
    | conditional_expression
    | map_reduce_expression
    | universal_conversion
    | '(' expression ')'
```

The `null_literal` (the reserved word `null`) evaluates to the *null* value, which is implicitly convertible to any type, and can be used to initialize any object declared to have an `'optional'` type.

Other literals evaluate to a value of a corresponding *universal* type, and are implicitly convertible to any type that has a corresponding `"from_univ"` operator, so long as the value satisfies the precondition of the operator (see 4.2.5).

A `type_name` followed by an apostrophe (`'`) may be used to specify explicitly the result type of a `bracketed_expression` – one of the forms of expression that is enclosed in `( )` or `[ ]`, where the type might not be resolvable without additional context.

A `type_name` by itself is permitted if the type has a `"[..]"` operator defined for itself, and is equivalent to the set of possible values of the type. If the `type_name` identifies a subtype with a non-trivial Predicate, then the set is reduced to those elements of the type that satisfy the Predicate (see Syntactic Equivalences below).

See Aspect Specifications (Chapter 9) for the syntax of `universal_conversion`. See 4.2.9 for the syntax of `initial_value_specification`.

Examples:

```
Y := "This is a string literal";  -- Y must be of a type with a "from_univ" operator
                                  -- from Univ_String
return null;  -- function must have a return type of the form "optional T"
              -- indicating it might return "null" rather than a value of type T

Display(Output, Complex'(Real => 1.0, Imaginary => 1.0));
              -- Explicitly specify the result type of an aggregate
```

4.2.1 Syntactic Equivalences

A `type_name` appearing by itself as an expression is expanded as follows:

```
-- Presuming the following declaration for Positive
  subtype Positive is Integer with Predicate => Positive > 0;

  for I in Positive ...
-- expands into
  for I in [for J in Integer'[..] when Positive'Predicate(J) => J] ...
-- which expands into
  for I in [for J in Integer'[..] when J > 0 => J] ...
-- which expands into
  for I in 1 .. Integer'Last ...
```

4.2.2 Unary and Binary Operators

The following are the unary operators in Sparkel:

```
"+", "-", "abs", "not"
```

The following are the binary operators in Sparkel:

```
"**"                    -- Exponentiation

"*", "/", "rem", "mod"  -- Multiply, Divide, Remainder, and Modulo operators

"+", "-"                -- Addition and subtraction
"&"                     -- Used to concatenate containers and elements

"..", "<..",            -- Interval operators; closed, open-closed,
"..<", "<..<"           -- closed-open, open-open

"|"                     -- Used to combine sets and elements

"<", "<=", "=",         -- The usual relational operators
"!=", ">=", ">"         -- ("/=" is a synonym for "!=" when used in an expression)
"=?"                    -- The "compare" operator; all relational
                        -- operators are defined in terms of "=?"

"<<", ">>"              -- left shift and right shift

"and", "or", "xor"      -- The basic boolean operators
"and then", "or else"   -- Short-circuit boolean operators
"==>"                   -- "implication" operator
```

The highest precedence operators are the unary operators and the exponentiation ("**") operator. The next lower precedence operators are the multiplication, division, and remainder operators. The next lower precedence operators are the addition, subtraction, and concatenation operators. Next are the interval operators. Next the *combine* operator (`"|"`). Next the relational, compare, and shift operators. Lowest are the boolean operators.

Addition, subtraction, multiplication, division, concatenation, and combination are left-associative. Exponentiation is right-associative. For other operators, parentheses are required to indicate associativity among operators at the same level of precedence, except that for the boolean operators, a string of uses of the same operator do not require parentheses, and are treated as left-associative.

The binary *compare* operator ("=?") returns an Ordering value indicating the relation between the two parameters, being `Less`, `Equal`, `Greater`, or `Unordered`. The value `Unordered` is used for types with only a partial ordering. For example, the "=?" operator for sets would typically return `Equal` if the sets have the same members, `Less` if the left operand is a proper subset of the right, `Greater` if the left operand is a proper superset of the right, and `Unordered` otherwise. All of the other relational operators are defined in terms of "=?" – only "=?" is user-definable for a given type.

The evaluation of an expression using a unary or binary operator is in general equivalent to a call on the corresponding operation, meaning that the operands are evaluated in parallel and then the operation is called (see 6.4). The short-circuit boolean operators `"and then"` and `"or else"` and the implication operator `"==>"` are implemented in terms of the corresponding `if_expression` (see 4.2.8):

A **and then** B — *equivalent to (if A then B else False)*
A **or else** B — *equivalent to (if A then True else B)*
A ⟹ B — *equivalent to (if A then B else True)*

Examples of unary and binary operators:

```
S1 =? S2          -- Compare S1 and S2,
                  -- return Less, Equal, Greater, or Unordered
X ** 3            -- X cubed
abs (X - Y)       -- absolute value of difference
0 ..< Length      -- The interval 0, 1, .. Length - 1
(A and B) or C    -- parentheses required
A or B or C       -- parentheses not required
X * Y + U * V     -- parentheses not required
```

4.2.3 Syntactic Equivalences

The operator `/=` is interpreted as "divide and assign" when used in an assignment context (see 5.2). However for compatibility with existing SPARK code, in an expression context `/=` is interpreted as "not equal" in addition to `!=`.

The relational operators are defined in terms of `"=?"` as follows:

```
A = B    ==>   A =? B in [ Equal ]
A != B   ==>   A =? B in [ Less, Greater, Unordered ]
A < B    ==>   A =? B in [ Less ]
A <= B   ==>   A =? B in [ Less, Equal ]
A > B    ==>   A =? B in [ Greater ]
A >= B   ==>   A =? B in [ Greater, Equal ]
```

4.2.4 Membership and Null Tests

A membership test is used to determine whether a value can be converted to a type, satisfies the predicates of a subtype, or is in a particular interval or set. A null test is used to determine whether a value is the null value. The result of a membership test or null test is of type Boolean.

```
membership_test ::=
    expression [ 'not' ] 'in' expression
  | expression [ 'not' ] 'in' type_name

null_test ::= expression 'is' 'null' | expression 'not' 'null'
```

Examples:

```
X in 3..5         -- True if X >= 3 and X <= 5
Y not in T+       -- True if Y is not convertible to T+
Red in Primary    -- True if Red satisfies the predicate for Primary
Z not null        -- True if Z does not have a null value
```

4.2.5 Other Sparkel Operators

```
"from_univ"  -- invoked implicitly to convert from a value of a universal type
"to_univ"    -- invoked using "[[ expression ]]" to convert to a universal type
             -- and used implicitly to convert to a universal type for operations
             -- that take universal-type parameters
"convert"    -- invoked using "type_name ( expression )" to convert between types
"indexing"   -- invoked by "object [ operation_actuals ]" to index into a container
"slicing"    -- invoked by "object [ operation_actuals ]" to select a slice of a container
"index_set"  -- invoked by an iterator to iterate over the elements of a container
```

```
"[]"         -- invoked by "[]" to create an empty container; invoked implicitly
                by "[ key1 => value1, key2 => value2, ... ]" followed by multiple calls
                on "|=" to build up a container given the key/value pairs
"[..]"       -- invoked by "[..]" to create a universal set;
                invoked implicitly to turn a type name into the set of its values
"magnitude"  -- invoked by "| val |", typically used for abs, size, discriminant, etc.
```

Examples:

```
X := 42;          -- Implicit conversion from Univ_Integer using "from_univ" operator
Print( [[X]] );   -- Convert back to Univ_Integer for printing using "to_univ" operator
C[Key]            -- The element of C associated with given Key using "indexing" operator
A[X..<Y]          -- The slice of A going from X to Y-1 using "slicing" operator
[]                -- An empty container using "[]" operator
|Str|             -- The length of the string Str using "magnitude" operator
```

4.2.6 Aggregates

Aggregates are used for constructing values out of their constituents. There are two kinds of aggregates: the `record_aggregate` for creating an object of a record type from its named components, and the `container_aggregate`, for creating an object of a container type (see 8.2) from a sequence of elements, optionally associated with one or more keys.

The `record_aggregate` is only available for a visible record type. A `container_aggregate` may be used with any type that defines the appropriate operators (see 8.2).

Aggregates have the following form:

```
aggregate ::= record_aggregate | container_aggregate

record_aggregate ::= '(' record_components ')'

record_components ::= [ record_component { ',' record_component } ]

record_component ::=
    [ identifier '=>' ] expression
  | identifier '<==' object_name
```

See 8.2 Container Aggregates for the syntax of a `container_aggregate`.

In a `record_aggregate`, named components (`record_component` with an `identifier` specified) must follow any positional components (those without an `identifier` specified). If the `'<=='` *move* operation is specified, then the value of the component is moved from the named existing object, leaving it null.

Examples:

```
(X => 3.5, Y => 6.2)       -- fully named record_aggregate

(Element, Next => null)    -- mixed positional and named record_aggregate

List := (Item <== Element, Next <== List);
                           -- move Element to front of linked list
```

4.2.7 Quantified Expressions

Quantified expressions are used to specify a boolean condition that depends on the properties of a set of values.

A quantified expression has the form:

```
quantified_expression ::=
  '(' 'for' all_or_some quantified_iterator '=>' condition ')'

all_or_some ::= 'all' | 'some'

quantified_iterator ::=
  set_iterator | element_iterator | initial_next_while_iterator
```

See Loop Statements (section 5.6) for the syntax of the various iterator forms.

A `quantified_expression` with the reserved word `all` is True if and only if the condition evaluates to True for all of the elements of the sequence produced by the `quantified_iterator`. A `quantified_expression` with the reserved word `some` is True if and only if the condition evaluates to True for at least one of the elements of the sequence produced by the `quantified_iterator`. It is not specified in what order the evaluations of the condition are performed, nor whether they are evaluated in parallel. The condition might not be evaluated for a given element of the sequence if the value for some other element already determines the final result.

Examples:

```
N_Is_Composite := (for some X in 2..N/2 => N rem X = 0);

Y_Is_Max := (for all I in Bounds(A) => A[I] <= Y);
```

Syntactic Equivalences

A quantified expression is equivalent to a loop that accumulates a boolean result summarizing the value of a condition evaluated over the iterated elements. A quantified expression with the reserved word `some` expands as follows:

```
   return (for some X in 2 .. N/2 => N rem X = 0);
-- expands into:
   var Result : Boolean := False;
   for X in 2 .. N/2 loop
      Result := Result or else (N rem X = 0);
   end loop;
   return Result;
```

A quantified expression with the reserved word `all` expands as follows:

```
   return (for all I in Bounds(A) => A[I] <= Y);
-- expands into:
   var Result : Boolean := True;
   for I in Bounds(A) loop
      Result := Result and then (A[I] <= Y);
   end loop;
   return Result;
```

4.2.8 Conditional Expressions

Conditional expressions are used to specify a value by conditionally selecting one expression to evaluate among several.

Conditional expressions are of one of the following forms:

```
conditional_expression ::= if_expression | case_expression
```

If Expression

An `if_expression` has the following syntax:

```
if_expression ::=
    '(' 'if' condition 'then' expression else_part_expression ')'

else_part_expresssion ::=
  { 'elsif' condition 'then' expression } 'else' expression
```

All `expression`s of an `if_expression` must be implicitly convertible to the same type.

To evaluate an `if_expression`, the initial `condition` and those within the associated `else_part_expression` are evaluated in sequence, and the first one that evaluates to True determines the expression to be evaluated (the one following the corresponding `'then'`). If all of the conditions evaluate to False, the last expression of the associated `else_part_expression` is evaluated to produce the value of the overall `if_expression`.

Examples:

```
Bigger := (if X > Y then X else Y);

return (if Y = 0 then null else X/Y);  -- return null if would divide by zero
```

Case Expression

Case expressions have the following form:

```
case_expression ::=
  '(' 'case' case_selector 'is'
      case_expression_alternative { ';'
      case_expression_alternative } [ ';'
      case_expression_default ]
  ')'

case_expression_alternative ::=
    'when' choice_list '=>' expression
  | 'when' identifier ':' type_name '=>' expression

case_expression_default ::=
  'when' 'others' '=>' expression
```

See Case Statements (section 5.4) for the syntax of `case_selector` and `choice_list`.

All expressions following `'=>'` of a `case_expression` must be implicitly convertible to the same type.

The `choice_list` or `type_name` of each `case_expression_alternative` determines a set of values. If there is not a `case_expression_default`, then the sets associated with the `case_expression_alternative`s must cover all possible values of the `case_selector`. The sets associated with the `case_expression_alternative`s must be disjoint with one another, except if there is a `type_name` that identifies a polymorphic type, in which case earlier alternatives take precedence over later polymorphic alternatives.

To evaluate a `case_expression`, the `case_selector` is evaluated. If the value of the `case_selector` is in a set associated with a given `case_expression_alternative`, the corresponding expression is evaluated. If the value is not a member of any set, then the expression of the `case_expression_default` is evaluated.

If a `case_expression_alternative` includes an identifier and a `type_name`, then within the expression, the identifier has the given type, with its value given by a conversion of the `case_selector` to the given type.

Example:

```
return (case Key =? Node.Key is
   when Less => Search(Node.Left, Key);
   when Equal => Node.Value;
   when Greater => Search(Node.Right, Key));
```

4.2.9 Map-Reduce Expressions

Map-reduce expressions are used to specify a value that is produced by combining a set of values, given an initial value and an operation to be performed with each value.

A map-reduce expression has the form:

```
map_reduce_expression ::=
  '(' 'for' map_reduce_iterator [ value_filter ] '=>'
                                    expression_with_initial_value ')'

map_reduce_iterator ::=
    set_iterator
  | 'each' element_iterator
  | initial_next_while_iterator

expression_with_initial_value ::= expression

initial_value_specification ::= '<' expression '>'
```

See Loop Statements (section 5.6) for the syntax of `value_filter` and the various iterator forms.

An `expression_with_initial_value` must have exactly one `initial_value_specification` within it (not including the contents of any nested `map_reduce_expressions`). The `expression` of the `initial_value_specification` must not refer to the loop variable of the iterator.

For the evaluation of a `map_reduce_expression`, first the `expession` of the `initial_value_specification` is evaluated and it becomes the *initial* result of the `map_reduce_expression`. Then for each element of the sequence of values produced by the `map_reduce_iterator` that satisfies the `value_filter`, if any, the `expression_with_initial_value` is evaluated, with the `initial_value_specification` taking on the value of the *current* result of the `map_reduce_expression`, and the result of the evaluation becoming the *next* result of the `map_reduce_expression`. After all of the elements of the sequence produced by the iterator have been combined, the last such evaluation determines the *final* result. If there are no elements in the sequence, then the *initial* result is used.

Examples:

```
Sum_Of_Squares := (for X in 1 .. N => <0> + X**2);

Largest_In_Absolute_Value :=
  (for each E of Arr => Max (<null>, abs E));
```

Note that the language-provided Max operations, when given a null operand, will return the other operand. The same applies to the Min operations.

Syntactic Equivalence

A map-reduce expression is equivalent to a loop that accumulates a result. For example, a map-reduce expression to return the sum of the squares of the odd integers <= N expands as follows:

```
   return (for X in 1 .. N when X mod 2 = 1 => <0> + X**2);
-- expands into:
```

```
var Result : Result_Type := 0;
for X in 1 .. N when X mod 2 = 1 loop
   Result := Result + X**2;
end loop;
return Result;
```

4.2.10 Type Conversion

A type conversion can be used to convert an expression from one type to another, by using a syntax like that of an operation call but with the operation identified by the name of the target type:

```
type_conversion ::= type_name '(' expression ')'
```

The expression of a `type_conversion` must be *convertible* to the target type. An expression of a type A is *convertible* to a type B if the type A is *convertible* to type B and the value of the expression after conversion satisfies any value-constraints on B.

Type A is *convertible* to type B if and only if:

- Types A and B are derived from the same original `type_definition` with equivalent actuals if any, but without any type extensions.
- Type B is a polymorphic type (see 7.2.1), and type A is derived from a type equivalent to the root type of B, in the case the root type of B is not a generic type, A implements all of the primitive operations of the root type of B.
- Type A is a polymorphic type, and the run-time type of the expression identifies a type that is convertible to B;
- Type A has a `"to_univ"` operator and type B has a `"from_univ"` operator such that the result type of the `"to_univ"` operator is the parameter type of the `"from_univ"` operator;
- Type A or type B has a `"convert"` operator that has a parameter type that matches type A and a result type that matches type B.

Chapter 5

Statements

Statements specify an action to be performed as part of a sequence of statements. A Sparkel statement can either be a simple statement, a compound statement containing other statements as constituents, or a local declaration:

```
statement ::= simple_statement | [ label ] compound_statement | local_declaration

simple_statement ::=
  assignment_statement
| exit_statement
| continue_statement
| return_statement
| operation_call

label ::= statement_identifier ':'

statement_identifier ::= identifier

compound_statement ::=
  if_statement | case_statement | loop_statement | block_statement

local_declaration ::= object_declaration | operation_declaration | operation_definition
```

If and only if a `compound_statement` is preceded by a label, then the `statement_identifier` must appear again at the end of the `compound_statement`.

If a `compound_statement` completes normally, as opposed to ending via an `exit_statement`, `continue_statement`, or `return_statement`, then the `with_values` clause, if any, at the end of the `compound_statement` is executed.

```
with_values ::=
    'with' identifier '=>' expression
  | 'with' '(' identifier '=>' expression { ',' identifier '=>' expression } ')'
```

5.1 Statement Separators

Statements are separated with `';'`, `'||'`, or `'then'`. The delimiter `';'` may also be used as a statement terminator. The separator used between statements determines whether the statements are executed in

strict sequence (’then’), in parallel (’||’), or according to data dependences (’;’). These three alternatives correspond to the semantics described below for statement_list, statement_thread_group, and statement_sequence/statement_thread:

```
statement_list ::=
  statement_group { [ ';' ] 'then' statement_group } ';'

statement_group ::= statement_sequence | statement_thread_group

statement_sequence ::= statement { ';' statement }

statement_thread_group ::=
      statement_thread [ ';' ]
 '||' statement_thread { [ ';' ]
 '||' statement_thread }

statement_thread ::= statement { ';' statement }
```

The scope of a local_declaration occurring immediately within a statement_sequence goes from the declaration to the end of the immediately enclosing statement_list. The scope of a local_declaration occurring immediately within a statement_thread goes from the declaration to the end of the statement_thread.

For the execution of a statement_list, each statement_group is executed to completion in sequence. For the execution of a statement_thread_group, each statement_thread is executed concurrently with other statement_threads of the same group. For the execution of a statement_sequence or statement_thread, expressions are evaluated and assignments and calls are performed in an order consistent with the order of references to unprotected objects (see chapter 10) occurring in the statements.

Examples:

```
A := C(B); D := F(E) || U := G(V); W := H(X);
```

The first two statements run as one thread, the latter two run as a separate thread.

```
block
   var A : Vector<Integer> := [X, Y];
 then
   Process(A[1]);
 ||
   Process(A[2]);
end block;
```

The declaration of A is completed before beginning the two separate threads invoking Process on the two elements of A.

5.2 Assignment Statements

An assignment_statement allows for replacing the value of one or more objects with new values.

```
assignment_statement ::=
    object_name ':=' expression
  | object_name '<==' object_name
  | object_name '<=>' object_name
  | record_aggregate ':=' expression
  | object_name operate_and_assign expression
```

There are builtin operations for simple assignment, for moving an object to a new location leaving a null behind, and for swapping the content of two objects:

```
":="   -- simple assignment of right-hand-side into left-hand-side
"<=="  -- move contents of right-hand-side to left-hand-side, leaving
          the right hand side "null"
"<=>"  -- swap left and right hand content
```

Multiple objects may be assigned in a single assignment by using a `record_aggregate` as the left hand side of an assignment.

In addition to the built-in assignment, move, and swap operations, several of the binary operators may be combined with "=" to produce operate-and-assign operations:

```
operate_and_assign ::=
    '+=' | '-=' | '&=' | '*=' | '/=' | '**=' | '<<=' | '>>='
  | 'and=' | 'or=' | 'xor=' | '|=' | '<|='
```

The last operator `'<|='` combines the right hand side into the left hand side, and then sets the right hand side to null. This is analogous to the move (`'<=='`) operation defined above, except that the left hand side is presumed to be a container into which the right hand side is combined.

Examples:

```
X := A + B;          -- Set X to sum of A and B
Tail <== List.Next;-- Remove the tail of List and assign to Tail.
Y <=> Z;             -- swap Y and Z
(Y, Z) := (Z, Y);    -- another way to swap Y and Z
X += 1;              -- Add one to X
Y *= 2;              -- Multiply Y by 2
C |= Elem;           -- Add Elem to the C container
C <|= Elem;          -- Move Elem into the C container
```

5.3 If Statements

If statements provide conditional execution based on the value of a boolean expression.

If statements are of the form:

```
if_statement ::=
  'if' condition 'then'
     statement_list
  [ else_part ]
  'end if' [ statement_identifier ] [ with_values ]

else_part ::=
    'elsif' condition 'then'
       statement_list
    [ else_part ]
  | 'else'
       statement_list

condition ::= expression  -- must be of a boolean type
```

For the execution of an `if_statement`, the condition is evaluated and if True, then the `statement_list` of the `if_statement` is executed. Otherwise, the `else_part`, if any, is executed.

For the execution of an `else_part`, if the `else_part` begins with `elsif`, then the condition is evaluated and if True, the `statement_list` following `then` is executed. Otherwise, the nested `else_part`, if any, is executed. If the `else_part` begins with `else`, then the `statement_list` following the `else` is executed.

Example:

```
if This_Were(A_Real_Emergency) then
   You_Would(Be_Instructed, Appropriately);
elsif This_Is(Only_A_Test) then
   Not_To_Worry;
end if;
```

5.4 Case Statements

Case statements allow for the selection of one of multiple statement lists based on the value of an expression.

Case statements are of the form:

```
case_statement ::=
  'case' case_selector 'is'
      case_alternative
    { case_alternative }
    [ case_default ]
  'end' 'case' [ statement_identifier ] [ with_values ]

case_selector ::= expression

case_alternative ::=
    when choice_list '=>' statement_list
  | when identifier ':' type_name '=>' statement_list

choice_list ::= choice { '|' choice }

choice ::= expression [ interval_operator expression ]

interval_operator ::= '..' | '..<' | '<..' | '<..<'

case_default ::=
  'when' 'others' '=>' statement_list
```

The `choice_list` or `type_name` of each `case_alternative` determines a set of values. If there is not a `case_default`, then the sets associated with the `case_alternative`s must cover all possible values of the `case_selector`. The sets associated with the `case_alternative`s must be disjoint with one another, except if there is a `type_name` that identifies a polymorphic type, in which case earlier alternatives take precedence over later polymorphic alternatives.

For the execution of a `case_statement`, the `case_selector` is evaluated. If the value of the `case_selector` is in a set associated with a given `case_alternative`, the corresponding `statement_list` is executed. If the value is not a member of any set, then the `statement_list` of the `case_default` is executed.

If a `case_alternative` includes an identifier and a `type_name`, then within the `statement_list`, the identifier has the given type, with its value given by a conversion of the `case_selector` to the given type.

Example:

```
case Lookahead(Input) is
  when 'a'..'z' | 'A'..'Z' =>
     Handle_Alphabetic(Input);
```

```
   when '0'..'9' =>
      Handle_Numeric(Input);
   when '\n' =>
      Handle_End_Of_Line(Input);
   when '\0' =>
      Handle_End_Of_Input(Input);
   when others =>
      Handle_Others(Input);
end case;
```

5.5 Block Statements

A block statement allows the grouping of a set of statements with local declarations and an optional set of assignments to perform if it completes normally.

A block statement has the following form:

```
block_statement ::=
  'block'
     statement_list
  'end' 'block' [ statement_identifier ] [ with_values ]
```

For the execution of a `block_statement`, the `statement_list` is executed. If the `statement_list` completes without being left due to an exit or return statement, the `with_values` clause at the end of the block, if any, is executed.

5.5.1 Syntactic Equivalences

For compatibility with existing SPARK code, the following syntactic equivalence is supported:

```
block_statement ::=
   [ 'declare' statement_list ] 'begin' statement_list 'end'
```

```
   declare
      var A : T;
   begin
      A := E;
   end;
-- expands into:
   block
      var A : T;
     then
      A := E;
   end block
```

5.6 Loop Statements

A loop statement allows for the iteration of a `statement_list` over a sequence of objects or values.

Loop statements have the following form:

```
loop_statement ::=
  while_until_loop | for_loop | indefinite_loop
```

```
while_until_loop ::= while_or_until condition loop_body

while_or_until ::= 'while' | 'until'
```

For the execution of a `while_until_loop` the condition is evaluated. If the condition is satisified, meaning it evaluates to True when `while` is specified or evaluates to False when `until` is specified, then the `statement_list` of the `loop_body` is executed, and if the `statement_list` reaches its end, the process repeats. If the condition is not satisfied, then the current iteration completes without executing the `statement_list`. If this is the last iteration active within the loop, the `while_until_loop` is completed, and the `with_values` clause at the end of the `loop_body`, if any, is executed.

```
indefinite_loop ::= loop_body
```

An `indefinite_loop` is equivalent to a `while_until_loop` that begins with `while` and has an expression of `True`.

```
for_loop ::=
    'for' iterator [ value_filter ] [ direction ] loop_body
  | 'for' '(' iterator_list ')' [ value_filter ] [ direction ] loop_body

value_filter ::= 'when' condition

direction ::= 'forward' | 'reverse' | 'parallel'

loop_body ::=
  'loop'
     statement_list
  'end' 'loop' [ statement_identifier ] [ with_values ]

iterator_list ::=
    iterator [ direction ] { ';' iterator [ direction ] }

iterator ::=
    set_iterator
  | 'each' element_iterator
  | value_iterator

set_iterator ::=
  identifier [ ':' type_name ] 'in' expression

value_iterator ::=
    initial_value_iterator
  | initial_next_while_iterator

initial_value_iterator ::=
    loop_variable_initializer [ while_or_until_condition]

initial_next_while_iterator ::=
    loop_variable_initializer next_values [ while_or_until condition ]

loop_variable_initializer ::=
    identifier [ ':' type_name ] ':=' expression
  | identifier '=>' object_name
```

```
next_values ::= 'then' expression { '||' expression }
```

See 8.3 for the syntax of an `element_iterator`.

A direction of `forward` or `reverse` is permitted only when at least one of the iterators of the `for_statement` is a `set_iterator` or an `element_iterator`. The direction determines the order of the sequence of values produced by such iterators. In the absence of a `forward` or `reverse` direction, such iterators may generate their sequence of values in any order.

The identifier of an iterator declares a *loop variable* which is bound to a particular object or value for each execution of the `statement_list` of the `loop_body`.

Each kind of iterator produces a sequence of values (or objects). If a `value_filter` is present, the sequence is reduced to those values (or objects) that satisfy the `value_filter condition`.

The values in the sequence produced by a `set_iterator` are all of the values of the set, less those that do not satisfy the `value_filter`, if any. The values in the sequence produced by a `value_iterator` are given explicitly by the initial value (or initial object when `'=>'` is used), and the next values, either specified in the `initial_next_while_iterator` itself after `then`, or in `continue` statements within the body of the loop, as long as the `while_or_until_condition` is satisfied. Again, if there is a `value_filter`, the values that do not satisfy the `value_filter` are skipped. See section 8.3 for a description of the sequence of objects, or key-value pairs, produced by an `element_iterator`.

If the `expression` of a `set_iterator` is a `type_name`, it is equivalent to invoking the `"[..]"` operator defined for that type, to produce the set of all values of the type (see section 4.2.5).

For the execution of a `for_loop` with a single iterator, the `statement_list` of the `loop_body` is executed once for each element in the sequence of values produced by the iterator (along with values specified by `continue_statements` that apply to the `for_loop` and have a `with_values` clause – see 5.6.1). For each execution of the `statement_list`, the loop variable is bound to the corresponding element of the sequence (or the value specified by the continue statement – see 5.6.1).

For the execution of a `for_loop` with multiple iterators, the `statement_list` of the `loop_body` is executed once for each set of elements determined by the set of iterators (and any applicable `continue_statements` having a `with_values` clause), with the iterator that produces the shortest sequence limiting the number of executions of the `statement_list`. That is, the `for_loop` terminates as soon as any one of the iterators has exhausted its sequence. If there is a `value_filter`, then the `loop_body` is skipped for any set of elements that does not satisfy the filter.

After a `for_loop` terminates normally, that is, without being exited by an exit or return statement, the `with_values` clause, if any, is executed.

Examples:

```
for I in 1..10 parallel loop
    X[I] := I ** 2;
end loop;
```

The above loop initializes a table of squares in parallel.

```
for each S of List_Of_Students(Classroom) when Is_Undergraduate(S) forward loop
    Print(Report, Name(S));
end loop;
```

The above loop prints the names of the undergraduate students (i.e. those satisfying the Is_Undergraduate filter) in the order returned by the List_Of_Students function for the given Classroom.

```
for X => Root then X.Left || X.Right while X not null loop
    Process(X.Data);
end loop;
```

The above loop calls Process on the Data component of the Root, and then initiates two new iterations in parallel, one on the Left subtree of X and one on the Right subtree. An iteration is not performed for cases

where X is null. The loop as a whole terminates when Process has been called on the Data component of each element of the binary tree.

5.6.1 Continue Statements

A continue statement may appear within a loop, and causes a new iteration of the loop to begin, optionally with new binding(s) for the loop variable(s).

A continue statement has the following form:

```
continue_statement ::= 'continue' 'loop' [ statement_identifier ] [ with_values ]
```

For the execution of a `continue_statement`, the current thread completes the current iteration of the immediately enclosing loop, and begins a new iteration of the specified loop (or in the absence of a `statement_identifier`, the immediately enclosing loop). If the identified loop is a `for_loop` without a specified sequence or next value, then there must be a `with_values` clause, which determines the new binding(s) for the loop variable(s).

Example:

```
for X => Root while X not null loop
    Process(X.Data);
 || continue loop with X => X.Left;
 || continue loop with X => X.Right;
end loop;
```

The above loop walks a binary tree in parallel, with the continue statements used to initiate additional iterations of the loop body to process the Left and Right subtrees of X. This is equivalent to the example given in 5.5, except that the subtrees of X are walked in parallel with calling Process on X.Data. (The example given in 5.5 could be made exactly equivalent by making that loop into a `parallel loop`, which means that while performing the `statement_list` of one iteration we proceed onto the next values.)

Note that the `loop_statement` whose iteration is terminated by a `continue_statement` may be nested within the `loop_statement` identified by the `statement_identifier`, and this outer `loop_statement` is the one that begins a new iteration.

Example:

```
   var Solutions : Protected_Vector<Solution> := [];
  Outer_Loop:
   for (C : Column := 1; Trial : Solution := Empty()) loop

      for R in Row parallel loop -- Iterate over the rows
         if Acceptable(Trial, R, C) then
            -- Found a Row/Column location that is acceptable
            if C < N then
               -- Keep going since haven't reached Nth column.
               continue loop Outer_Loop with (C => C+1,
                  Trial => Incorporate(Trial, R, C));
            else
               -- All done, remember trial result that works
               Solutions |= Incorporate(Trial, R, C);
            end if;
         end if;
      end loop;
   end loop Outer_Loop;
```

If an inner `loop_statement` has multiple iterations active concurrently, a `continue_statement` terminates only one of them. The other active iterations proceed independently. The inner `loop_statement` as a whole only completes when all of the active iterations within the loop are complete. If all of the iterations of the inner `loop_statement` end with a `continue_statement` to an outer `loop_statement`, then the

thread that initiated the inner `loop_statement` is terminated. If at least one of the iterations of the inner `loop_statement` completes normally, then the thread that initiated the inner `loop_statement` executes the `with_values` clause, if any, and proceeds with the statements following the inner `loop_statement`.

In this example, the above doubly nested loop iterates over the columns of a chessboard, and for each column iterates in parallel over the rows of the chessboard, trying to find a place to add a piece that satisfies the Acceptable function. When a place is found at a given row on the current column, the continue statement proceeds to the next column with the given Trial solution. Meanwhile, other rows are being checked, which may also result in additional continuations to subsequent columns. If a given row is not acceptable in a given column for the current Trial, it is ignored and the thread associated with that row completes rather than being used to begin another iteration of the outer loop.

5.6.2 Syntactic Equivalences

For compatibility with existing SPARK code, the `direction` may be specified immediately after the reserved word `'in'`, within a `set_iterator`:

```
set_iterator ::=
  identifier [ ':' type_name ] 'in' direction expression
```

```
    for X in reverse 1 .. 10 loop
 -- is equivalent to:
    for X in 1 .. 10 reverse loop
```

All of the iterator forms may be expressed in terms of the `initial_value_iterator` with an explicit `continue` statement.

```
    for X => Root then X.Left || X.Right while X not null loop
        Process(X.Data);
    end loop;
 -- expands into:
    for X => Root while X not null loop
        Process(X.Data);
      then
        continue loop with X => X.Left;
      ||
        continue loop with X => X.Right;
    end loop;
```

If a `parallel loop` is specified, then the `continue` statements occur in parallel with executing the loop body.

```
    for X => Root then X.Left || X.Right while X not null parallel loop
        Process(X.Data);
    end loop;
 -- expands into:
    for X => Root while X not null loop
        Process(X.Data);
      ||
        continue loop with X => X.Left;
      ||
        continue loop with X => X.Right;
    end loop;
```

The `set_iterator` expands as follows, depending on whether it is an unordered, forward, or reverse loop:

```
   for X in My_Set loop . . . -- unordered loop
-- expands into:
```

```
   var Copy_Of_My_Set := My_Set
   for X := Remove_Any(Copy_Of_My_Set)
         then Remove_Any(Copy_Of_My_Set) while X not null loop . . .

   for X in My_Set forward loop . . .
-- expands into:
   var Copy_Of_My_Set := My_Set
   for X := Remove_First(Copy_Of_My_Set)
         then Remove_First(Copy_Of_My_Set) while X not null loop . . .

   for X in My_Set reverse loop . . .
-- expands into:
   var Copy_Of_My_Set := My_Set
   for X := Remove_Last(Copy_Of_My_Set)
         then Remove_Last(Copy_Of_My_Set) while X not null loop . . .
```

A `set_iterator` specified with a `parallel loop` is like the unordered loop, but with the continuation to the next element happening in parallel with executing the body of the loop.

See 8.3 for the expansion of an `element_iterator`.

5.7 Exit statements

An exit statement may be used to exit a compound statement while terminating any other threads active within the compound statement.

An exit statement has the following form:

```
exit_statement ::=
  'exit' [ compound_kind ] [ statement_identifier ]
     [ 'when' condition ] [ with_values ]

compound_kind ::= 'if' | 'case' | 'block' | 'loop'
```

An exit statement exits the specified `compound_statement` (or in the absence of a `statement_identifier`, the immediately enclosing `compound_statement` of the specified `compound_kind`), terminating any other threads active within the identified statement. If neither a `compound_kind` nor a `statement_identifier` is specified, the `compound_kind` is taken to be `loop`.

If the `exit_statement` has a `'when' condition` clause, then the exit is only performed if the condition evaluates to True.

If the `exit_statement` has a `with_values` clause, then after terminating all other threads active within the compound statement, the assignments specified by the `with_values` clause are executed.

Example:

```
   const Result : Result_Type;
   block
      const Result1 := Compute_One_Way(X);
      exit block with Result => Result1;
    ||
      const Result2 := Compute_Other_Way(X);
      exit block with Result => Result2;
   end block;
```

The above block performs the same computation two different ways, and then exits the block with the Result object assigned to whichever answer is computed first.

5.7.1 Syntactic Equivalences

A `'when' condition` is equivalent to wrapping the `exit` statement in an `if` statement. As indicated above, `loop` is assumed if neither a `compound_kind` nor a `statement_identifier` is specified. Hence the following expansion occurs:

```
   exit when X = 0;
-- expands to:
   if X = 0 then
      exit loop;
   end if;
```

Chapter 6

Operations

Operations are used to specify an algorithm for computing a value or performing a sequence of actions. There are two kinds of operations – functions (`func`s) and procedures (`proc`s). In addition, a function or a procedure may be an *operator*, as indicated by its designator being an `operator_symbol` (which has the syntax of a `string_literal`). Operators have special meaning to the language, and are invoked using special syntax. Non-operator functions and procedures are invoked using a name followed by parameters (if any) in parentheses. Functions produce one or more results. Operations may update one or more of their variable parameters.

6.1 Operation Declarations

Operations are declared using the following forms:

```
operation_declaration ::=
  function_declaration | procedure_declaration

function_declaration ::=
  [ 'abstract' | 'optional' ] [ 'queued' ] 'func' designator [ parameter_profile ]
     'return' result_specification

procedure_declaration ::=
  [ 'abstract' | 'optional' ] [ 'queued' ] 'proc' designator [ parameter_profile ]

designator ::= identifier | operator_symbol

operator_symbol ::= string_literal

parameter_profile ::=
     '(' parameter_specification { ';' parameter_specification } ')'
   | '(' { parameter_specification ';' } global_specification { ';' global_specification } ')'

parameter_specification ::=
  [ parameter_mode ] [ identifier_list ':' ] subtype_indication
     [ container_specifier] [ ':=' expression ]

parameter_mode ::=
    'var'
```

```
  | 'ref' [ var_or_const ]
  | 'locked' [ 'var' ]
  | 'queued' [ 'var' ]

global_specification ::= global_mode [ identifier_list ]

global_mode ::= [ 'ref' ] 'global' [ 'var' ]

identifier_list ::= identifier { ',' identifier }

result_specification ::=
  [ result_mode ] [ identifier ':' ] subtype_indication

result_mode ::= 'ref' [ var_or_const ]
```

See 8.4 for the syntax of a `container_specifier`. Note that an `anon_record_type_specifier` (see 3.1.1) may be used as part of a `result_specification`, allowing multiple separately-named results to be returned.

If an identifier is omitted for a parameter, the identifier of the `type_name` (if any) used within the `subtype_indication` may be used within the operation to identify the parameter, if it is unique. Otherwise, the parameter is unnamed at the call point. In an `operation_definition` (see 6.3) all parameters must either have a specified identifier, or have a unique `type_name` identifier.

If there is no `parameter_mode`, or if `var` does not appear in the `parameter_mode`, then the formal is read-only within the body of the operation. If the `parameter_mode` is `ref` without being followed by `var` or `const`, then within the operation the formal is read-only; however, for any result that is also of mode simply `ref`, the result is (in the caller) a variable reference to the returned object if and only if all of the parameters with mode merely `ref` are variables in the caller. If any of the parameters with mode `ref` are constants, then all of the results with mode `ref` are constants.

A result of mode `ref` must be specified via a return statement as a reference to all or part of a parameter of mode `ref`. A result of mode `ref var` must be specified via a return statement as a reference to all or part of a parameter of mode `ref var`. A result of mode `ref const` must be specified via a return statement as a reference to all or part of some 'ref' parameter (`var`, `const`, or merely `ref`).

If a `global_specification` has no `identifier_list` then the operation is allowed to make the corresponding mode of reference to any visible global variable. This is most often used as part of an `operation_type` declaration (see 6.2).

See section 10.1.1 for the meaning of `'queued'` when applied to an operation as a whole.

See section 7.2 for the meaning of `'abstract'` and `'optional'` when applied to an operation.

Examples:

```
func Sin (X : Float) return Float;

func "=?" (Left, Right : Set) return Ordering;

func Divide ( Dividend : Integer; Divisor : Integer)
  return (Quotient : Integer; Remainder : Integer);

proc Update (var Obj : T; New_Info : Info_Type);

func "indexing"(ref C : Container; Index : Index_Type)
  return ref Element_Type;
```

6.1.1 Syntactic Equivalences

For compatibility with existing SPARK code, rather than `func` and `proc`, operations may be declared using the equivalents `function` and `procedure`.

6.2 Operation Types

An operation type may be used as a parameter type, to allow operations to be passed as parameters to other operations. All operation types are *limited* types, and hence do not permit assignment. Operation types are specified with the following syntax:

```
operation_type_specifier ::=
     [ 'queued' ] 'func' [ parameter_profile ] 'return' result_specification
   | [ 'queued' ] 'proc' [ parameter_profile ]
```

Two operation types are *structurally equivalent* if they specify the same number of parameters and results, with the same types and modes. Parameter names and defaults are not considered.

Examples:

```
type Trig_Func is func(Angle : Float) return Float;

type Action_Proc is proc(Obj : T; global var);
    -- Apparently has its effect by updating unspecified global variable(s)
```

6.3 Operation Definitions

An operation may be defined with a body, with an operation import, or by equivalence to an existing operation.

An operation definition has the following form:

```
operation_definition ::=
    function_definition
  | procedure_definition
  | operation_import
  | operation_equivalence

function_definition ::=
  function_declaration 'is'
    operation_body
  'end' [ 'func' ] designator

procedure_definition ::=
  procedure_declaration 'is'
    operation_body
  'end' [ 'proc' ] designator

operation_body ::= [ dequeue_condition ] statement_list

operation_import ::=
  operation_declaration 'with' 'Import' => string_literal

operation_equivalence ::=
```

```
    operation_declaration 'is' operation_name
  | operation_declaration 'is' [operation_designator] 'of' type_specifier
```

If an operation is declared with a separate `operation_declaration` (typically in a `package_specification`), then the `operation_declaration` part of the `operation_definition` must fully conform to it.

An `operation_import` indicates that the operation is defined externally to the current program, possibly in a different language.

An `operation_equivalence` indicates that the operation is merely a renaming of some existing operation, identified by the `operation_name`, or by a type and optional `operation_designator`. In this latter form, the existing operation must be an operation of the specified type, with the same designator as the new operation, or with the given `operation_designator` if specified. The existing operation must have the same number of parameters and results, of the same modes and with the same types.

Examples:

```
func Sin(X : Float) return Float with Import => "sinf";
  -- defined externally

proc "+="(var Left : Set; Right : Element) is "|=";
  -- defined by equivalence

func "in"(Left : Float; Right : Ordered_Set<Float>)
  is of Ordered_Set<Float>;  -- defined by equivalence

proc Update(var Obj : T; New_Info : Info_Type) is
    Obj.Info := New_Info;
end Update;

func Fib (N : Integer) return Integer is
   -- Recursive fibonacci but with linear time

   func Fib_Helper(M : Integer)
     return (Prev_Result : Integer; Result : Integer) is
     -- Recursive "helper" routine which
     -- returns the pair ( Fib(M-1), Fib(M) )
       if M <= 1 then
           -- Simple case
           return (Prev_Result => M-1, Result => M);
       else
           -- Recursive case
           const Prior_Pair := Fib_Helper(M-1);

           -- Compute next fibonacci pair in terms of prior pair
           return with
             (Prev_Result => Prior_Pair.Result,
              Result => Prior_Pair.Prev_Result + Prior_Pair.Result);
       end if;
   end Fib_Helper;

   -- Just pass the buck to the recursive helper function
   return Fib_Helper(N).Result;
end func Fib;
```

6.4 Operation Calls

Operation calls are used to invoke an operation, with parameters and/or results.

Operation calls are of the form:

```
operation_call ::= operation_name [ '(' operation_actuals ')' ]

operation_name ::=
    [ package_name '.' ] operation_designator
  | type_name ''' operation_designator
  | object_name '.' operation_designator

operation_designator ::= operator_symbol | identifier

operation_actuals ::= operation_actual { ',' operation_actual }

operation_actual ::=
    [ identifier '=>' ] actual_object
  | [ identifier '=>' ] actual_operation

actual_object ::= expression

actual_operation ::= operation_specification | 'null'
```

Unlike other names, an `operation_name` need not identify an operation that is directly visible. Operations declared within packages other than the current package are automatically considered, depending on the form of the `operation_name`:

- If the `operation_name` is of the form `package_name '.' operation_designator` then only operations in the named package are considered.
- If the `operation_name` is of the form `type_name ''' operation_designator` then only operations of the named type are considered.
- If the `operation_name` is of the form `object_name '.' operation_designator` then the call is equivalent to

  ```
  type_of_object_name ''' operation_designator
    '(' object_name ',' operation_actuals ')'
  ```

- Otherwise (the `operation_name` is a simple `operation_designator`), all operations with the given designator that are operations of any of the parameter types of the call, or of the expected result type of the call, are considered, along with locally declared (non-operator) operations with the given designator. (*Note that all operations of the parameter and results types are automatically visible. In Ada 2012 this would be as though there were a "use all type T" for each parameter or result type of the call. Sparkel does not have a "use [all] type" clause, as it would be redundant.*)

Any named `operation_actuals`, that is, those starting with `"identifier '=>'"`, must follow any positional `operation_actuals`, that is, those without `"identifier '=>'"`.

For the execution of an operation call, the `operation_actuals` are evaluated (in parallel – see 10.2), as are any default expressions associated with non-global parameters for which no actual is provided. For `global` parameters, a global protected object with the given identifier must be visible both to the caller and

the called operation, and if it is a `var` parameter, the caller must also have it as a `global var` parameter. After parallel evaluation of the `operation_actuals`, the body of the operation is executed, and then any results are available for use in the enclosing expression or statement.

If the type of one or more of the operation actuals is polymorphic (see 7.2.1), and the operation is an operation of the root type of the polymorphic type, then the actual body invoked depends on the run-time type-id of the actual if the corresponding formal parameter is *not* polymorphic. If multiple operation actuals have this same polymorphic type, and their corresponding formals are also *not* polymorphic, then their run-time type-ids must all be the same (with one exception – the `"=?"` operator always returns Unordered when given two polymorphic operands with different type-ids).

Examples:

```
Result := Fib (N => 3);

Graph.Display_Point(X, Y => Sin(X));

var A := Sparse_Array'Create(Bounds => 1..N);
```

6.5 Lambda Expressions

A lambda expression is used for defining an operation as part of passing it as an actual parameter to an operation call or a type_ or package_instantiation.

A lambda expression has the following form:

```
lambda_expression ::=
  [ 'lambda' parameters 'is' ] lambda_body

parameters ::= '(' identifier { ',' identifier } ')'

lambda_body ::= expression | '(' expression { ';' expression } ')'
```

Example:

```
Graph_Function(Window, lambda (X) is sin(X)**2);
```

The types of the parameters and the type of the result of the lambda_expression are determined by context (by the operation type specified for the corresponding parameter of the called routine or generic package). If the operation being passed is parameterless, then the lambda_body by itself is used. Such parameterless operations are passed, for example, as the right-hand side of the short-circuit operations (`and then`, `or else`, and `==>`).

6.6 Return Statements

A return statement is used to exit the nearest enclosing operation, optionally specifying one or more outputs.

A return statement has the following form:

```
return_statement ::=
    'return'
  | 'return' expression
  | 'return' with_values
```

If there is no output value specified, any outputs of the immediately enclosing operation must have already been assigned prior to the return statement. If there is only a single expression, the immediately enclosing operation must have only a single result.

Examples:

```
return Fib(N-1) + Fib(N-2);

return with (Quotient => Q, Remainder => R);
```

Chapter 7

Packages

Packages define a logically related group of types, operations, data, and, possibly, nested packages. Packages may be generic, parameterized by types, operations, or values.

Every package has a specification that declares external characteristics of its type and objects. If the specification of a package declares any private types or any non-abstract operations, the package must have a body that defines the internal representation of the private types and the algorithms for the operations

7.1 Package Specification

A package is declared by givings its package *specification*. The specification of a package uses the following syntax:

```
package_declaration ::= package_specification ';' | package_instantiation

package_specification ::=
 [ 'generic'
     { generic_formal_parameter ';' } ]
   'package' package_identifier 'is'
     { package_item }
   'end' [ 'package' ] package_identifier

package_identifier ::= { identifier  '.' } identifier

package_item ::=
    type_declaration
  | operation_declaration
  | object_declaration
  | package_declaration

package_instantiation ::=
   'package' package_identifier 'is' 'new' package_name '(' generic_actuals ')'
```

Generic formal parameters have the following form:

```
generic_formal_parameter ::= formal_type | formal_object

formal_type ::= type_derivation

formal_object ::= parameter_specification
```

An `object_declaration` that occurs immediately within a package specification or package body must be of a protected type (see 10.1).

Example (also used in section 3.1):

```
generic
    type Element_Type is new Assignable<>;
package List is
    type List is private;
    func Create return List;
    func Is_Empty(L : List) return Boolean;
    proc Append(var L : List; Elem : Element_Type);
    func Remove_First(var L : List) return optional Element_Type;
    func Nth_Element(ref L : List; N : Univ_Integer) return ref optional Element;
end List;
```

This defines the specification of the List package, which defines a type List and operations for creating a list, checking whether it is empty, appending to a list, removing the first element of the list, and getting a reference to the Nth element of the list.

7.1.1 Syntactic Equivalences

A `formal_object` whose type is an operation type is the way that a generic package may be parameterized by an operation. For compatibility with existing SPARK code, an alternative syntax is also provided:

```
generic_formal_parameter ::= formal_operation

formal_operation ::= 'with' operation_declaration [ 'is' operation_name ]
```

```
   with proc Action (Obj : T) is T'Print;
-- expands into:
   Action : proc(Obj : T) := T'Print
```

7.2 Type Inheritance and Extension

A type may be derived from an existing type, with or without extending the type, and may be defined to *implement* one or more other private types.

When a type T2 is *derived* from a type T1, it inherits *operations* from T1. As part of inheriting an operation from T1, the types of the non-polymorphic (see 7.2.1) parameters and results of the operation are altered by replacing each occurrence of the original type T1 with the new type T2. For example, an operation such as `func Invert(X : T1) return T1` becomes `func Invert(X : T2) return T2`.

An operation inherited from T1 is *abstract* only if the corresponding operation in T1 is abstract, or if the operation has a result which is of a type based on T1 (as does `Invert` in the above example). If the operation inherited from T1 is *not* abstract, then its implicit body is defined to call the operation of T1, with any parameter to this operation that is of the type T1 being passed the parent part of the corresponding parameter to the inherited operation.

An *inherited* operation may be *overridden* by providing a declaration for the operation in the package where the derived type is declared, with the same name and number and types of parameters and results as the inherited operation. An *abstract* inherited operation must be overridden unless the new type is itself specified as `'abstract'`.

Finally, if T1 has any *components*, then if T2 is *derived* from T1, T2 also inherits these components, with any visible components of T1 becoming visible components of T2.

If rather than being derived from T1, the type T2 *implements* T1 (directly or indirectly), and T2 is not itself declared as an abstract type, then T2 is required to declare a corresponding operation for each non-null

operation of type T1, but with the change in types of parameters and results from T1 to T2, as described above for inheritance.

If T1 has any *visible* components, then T1 cannot be *implemented* by other types, though T1 may still be *extended*.

Example:

```
generic
   type Skip_Elem_Type is private;
   Initial_Size : Univ_Integer := 8;
package Skip_List is
   type Skip_List is new Lists.List<Element_Type => Skip_Elem_Type>
      and Sets.Set<Elem_Type => Skip_Elem_Type> with private;

   -- The following operations are implicitly declared
   -- due to being inherited from List<Skip_Elem_Type>:

   -- abstract func Create return Skip_List;
   -- func Is_Empty(L : Skip_List) return Boolean;
   -- proc Append(var L : Skip_List; Elem : Skip_Elem_Type);
   -- func Remove_First(var L : Skip_List) return optional Skip_Elem_Type;
   -- func Nth_Element(ref L : Skip_List; N : Univ_Integer)
   --     return ref optional Skip_Elem_Type;

   func Create return Skip_List;
     -- This overrides the abstract inherited operation

   ... -- Here we may override other inherited operations
       -- or introduce new operations

   func Add(var L : Skip_List; Elem : Skip_Elem_Type) is Append;
      -- An operation required by the Set type, defined
      -- in terms of one inherited from List.

end Skip_List;
```

7.2.1 Polymorphic Types

If the name of a type is of the form `identifier '+'`, it denotes a *polymorphic* type. A polymorphic type represents the identified type plus any type that extends the type, or that implements all of the identified type's operations, with matching generic actuals. The identified type is called the *root* type for the corresponding polymorphic type.

For example, given the Skip_List generic type from the example in 7.2, and the Bool_List type from section 3.1:

```
type Bool_Skip_List is new Skip_Lists.Skip_List<Boolean>;

var BL : Bool_List+ := Bool_Skip_List'Create;
```

The variable BL can now hold values of any type that extends or implements the List type with Element_Type specified as Boolean. In this case it is initialized to hold an object of type Bool_Skip_List.

An object of a polymorphic type (a *polymorphic object*) includes a *type-id*, a run-time identification of the (non-polymorphic) type of the value it currently contains. The type-id of a polymorphic object may be tested with a membership test (see 4.2.4) or a case statement (see 5.4), and it controls which body is executed in certain operation calls (see 6.4). In the above example, the type-id of BL initially identifies the Bool_Skip_List type.

7.3 Package Body

A package body defines local types, operations, and objects for a package, as well as the full type for any private type not completed in the private part of the package specification, and a body for each operation declared in the package's specification that requires an implementation.

A package body has the following form:

```
package_body ::=
  'package' 'body' package_identifier 'is'
      { package_body_item }
  'end' [ 'package' ] package_identifier ';'

package_body_item ::=
    package_item
  | operation_body
  | package_body
```

Example:

```
package body List is
   type List_Node is record
       var Elem : Element_Type;
       var Next : optional List_Node;
   end record List_Node;

   type List is record
       var Head : optional List_Node;
   end record;

   func Create return List is
      return (Head => null);
   end Create;

   func Is_Empty(L : List) return Boolean is
      return L.Head is null;
   end Is_Empty;

   proc Append(var L : List; Elem : Element_Type) is
      for X => L.Head loop
         if X is null then
            -- Found the end, add new component here
            X := (Elem => Elem, Next => null);
         else
            -- Iterate with next node
            continue loop with X => X.Next;
         end if;
      end loop;
   end Append;

   func Remove_First(var L : List) return Result : optional Element_Type is
      if L.Head is null then
         -- List is empty, nothing to return
         return null;
      else
         -- Save first element and then delete node from list
```

```
            Result := L.Head.Elem;
            L.Head := L.Head.Next;
            return;    -- Result already assigned
         end if;
      end Remove_First;

      func Nth_Element(ref L : List; N : Univ_Integer)
        return ref optional Element is
         for (X => L.Head; I := 1) loop
            if X is null then
               -- reached end of list
               return null;
            elsif I = N then
               -- reached Nth element
               return X.Elem;
            else
               -- continue with next node of list
               continue loop with (X => X.Next, I => I+1);
            end if;
         end loop;
      end Nth_Element;

   end List;
```

The above defines the body for the package List whose specification is given in 7.1. Type List_Node is a type used for the implementation of the exported type List. The full type declaration is provided for the type List declared in the specification of package Lists. Following that are the bodies for the operations exported by the Lists package.

7.4 Package and Type Instantiation

Generic packages and types within them are instantiated by providing actuals to correspond to the generic formals. If an actual is not provided for a given formal, then the formal must have a default specified in its declaration, and that default is used.

The actual parameters used when instantiating a generic package or type to produce a (non-generic) package or type have the following form:

```
generic_actuals ::= [ generic_actual { ',' generic_actual } ]

generic_actual ::=
    [ identifier '=>' ] actual_type
  | [ identifier '=>' ] actual_operation
  | [ identifier '=>' ] actual_object

actual_type ::= subtype_indication
```

Any generic actuals with a specified identifier must follow any actuals without a specified identifier. The identifier given preceding `'=>'` in a generic_actual must correspond to the identifier of a formal parameter of the corresponding kind.

Chapter 8

Containers

A container is a type that defines an "indexing" operator, an "index_set" operator, a container aggregate operator "[]", a combining assignment operator `"|="`, and, optionally, a "slicing" operator. It will also typically define a Length or Count function, other operations for creating containers with particular capacities, for iterating over the containers, etc.

The *index type* of a container type is determined by the type of the second parameter of the "indexing" operator, and the *value type* of a container type is determined by the type of the result of the "indexing" operator.

The *index-set type* of a container type is the result type of the `"index_set"` operator, and must be either a set or interval over the index type.

Examples:

```
generic
    type Key_Type is new Hashable<>;
    type Element_Type is private;
package Map is
    type Map is private;
    func "[]" return Map;
    proc "|="(var M : Map; Key : Key_Type; Elem : Element_Type);
    func "indexing"(ref M : Map; Key : Key_Type)
      return ref optional Element_Type;
    func "index_set"(M : Map) return Set<Key_Type>;
end Map;
```

The Map package defines a container type Map with Key_Type as the index type and Element_Type as the value type. The package includes a parameterless container aggregate operator "[]" which produces an empty map, a combining operator `"|="` which adds a new `Key => Elem` pair to the map, "indexing" which returns a reference to the element of M identified by the Key (or null if none), and `"index_set"` which returns the set of Keys with non-null associated elements in the map.

```
generic
    type Element_Type is new Hashable<>;
package Set is
    type Set is private;
    func "[]"() return Set;      -- Empty set
    proc "|="(var S : Set; Elem : Element_Type);
    func Count(S : Set) return Univ_Integer;
    func "in"(Elem : Element_Type; S : Set) return Boolean;
    func "indexing"(ref S : Set;
      Index : Univ_Integer)
      return ref Elem
```

```
        with Pre => Index in 1 .. Count(S);
    func "index_set"(S : Set) return Interval<Univ_Integer>;
    func "=?"(Left, Right : Set) return Ordering;
end Set;
```

The Set package defines a container type Set with the Element_Type as the value type and Univ_Integer as the index type. The package includes a parameterless container aggregate operator "[]" which produces an empty set, a combining operator `"|="` which adds a new element to the set, an "in" operator which tests whether a given element is in the set, an "indexing" operator which returns the *n-th* element of the set, and an "index_set" operator which returns the interval of indices defined for the set (i.e. 1..Count(S)). The compare operator ("=?" – see 4.2.2) is provided for comparing sets for equality and subset/superset relationships.

```
generic
    type Component_Type is private;
    type Indexed_By is new Countable<>;
package Array is
    type Array is private;
    type Bounds_Type is Interval<Indexed_By>;
    func Bounds(A : Array) return Bounds_Type;

    func "[]"
      (Index_Set : Bounds_Type; Values : Map<Bounds_Type, Component_Type>)
      return Result : Array
      with Post => Bounds(Result) = Index_Set;

    func "indexing"(ref A : Array;
      Index : Indexed_By)
      return ref Component_Type;
    func "index_set"(A : Array)
      return Result : Bounds_Type
      with Pre => Index in Bounds(A),
           Post => Result = Bounds(A);

    func "slicing"(ref A : Array;
      Slice : Bounds_Type)
      return ref Result : Array
      with Pre => Slice <= Bounda(A),
           Post => Bounds(Result) = Slice;

    proc "|="(var A : Array;
      Index : Indexed_By;
      Value : Component_Type)
      with Pre => Index in Bounds(A);
    proc "|="(var A : Array;
      Slice : Bounds_Type;
      Value : Component_Type)
      with Pre => Slice <= Bounds(A);
end Array;
```

The Array package defines a container type Array with Component_Type as the value type and Indexed_By as the index type. The index-set type is Bounds_Type. The package includes a container aggregate operator "[]" which creates an array object with the given overall Index_Set and the given mapping of indices to values. It also defines an "indexing" operator which returns a reference to the component of A with the given Index, a "slicing" operator which returns a reference to a slice of A with the given subset of the Bounds, plus combining operators `"|="` which can be used to specify a new value for a single component or all components of a slice of the array A.

8.1 Object Indexing and Slicing

Object indexing is used to invoke the "indexing" operator to obtain a reference to an element of a container object. Object slicing is used to invoke the "slicing" operator to obtain a reference to a subset of the elements of a container object.

Object indexing and slicing use the following syntax. The form with `'[..]'` is only for slicing.

```
object_indexing_or_slicing ::=
    object_name '[' operation_actuals ']'
  | object_name '[..]'
```

If the form with `'[..]'` is used, or one or more of the `operation_actuals` are sets or intervals, then the construct is interpreted as an invocation of the "slicing" operator. Otherwise, it is interpreted as an invocation of the "indexing" operator. The `object_name` denotes the container object being indexed or sliced.

When interpreted as an invocation of the "slicing" operator, the construct is equivalent to:

```
"slicing" '(' object_name, operation_actuals ')'
```

or, for the form using `'[..]'`:

```
"slicing" '(' object_name ')'
```

When interpreted as an invocation of the "indexing" operator, the construct is equivalent to:

```
"indexing" '(' object_name, operation_actuals ')'
```

The implementation of an "indexing" operator must ensure that, given two invocations of the same "indexing" operator, if the actuals differ between the two invocations, then the results refer to different elements of the container object. Similarly, the implementation of a "slicing" operator must ensure that, given two invocations of the same "slicing" operator, if at least one of the actuals share no values between the two invocations, then the results share no elements.

If an implementation of the "indexing" operator and an implementation of the "slicing" operator for the same container type have types for corresponding parameters that are the same or differ only in that the one for the "slicing" operator is an interval or set of the one for the "indexing" operator, then the two operators are said to *correspond*. Given invocations of corresponding "indexing" and "slicing" operators, the implementation of the operators must ensure that if at least one pair of corresponding parameters share no values, then the results share no elements of the container object. A slice defined using `'[..]'` is presumed to refer to *all* elements of the container.

Examples:

```
Table[Key] += 1;        -- bump up Table entry associated with Key

A[1..3] <=> A[4..6];    -- swap halves of 6-element array
Qsort(V[..]);           -- Pass a slice representing all of V to Qsort
```

8.1.1 Syntactic Equivalences

For compatibility with existing SPARK code, parentheses `'(' ... ')'` may be used instead of brackets `'[' ... ']'` for indexing or slicing:

```
   M ( I, J )
-- is equivalent to
   M [ I, J ]

   A ( 1 .. 10 )
-- is equivalent to
   A [ 1 .. 10 ]
```

8.2 Container Aggregates

A container aggregate is used to create an object of a container type, with a specified set of elements, optionally associated with explicit indices.

```
container_aggregate ::=
    empty_container_aggregate
  | universal_container_aggregate
  | positional_container_aggregate
  | named_container_aggregate
  | iterator_container_aggregate

empty_container_aggregate ::= '[]'

universal_container_aggregate ::= '[..]'

positional_container_aggregate ::=
  '[' positional_container_element { ',' positional_container_element } ']'

positional_container_element ::= expression | default_container_element

default_container_element ::= 'others' '=>' expression

named_container_aggregate ::=
  '[' named_container_element { ',' named_container_element } ']'

named_container_element ::=
    choice_list '=>' expression
  | default_container_element

iterator_container_aggregate ::=
  '[' 'for' iterator [ value_filter ] [ direction ] [ ',' index_expr ]
      '=>' expression ']'

index_expr ::= expression
```

An `empty_container_aggregate` is only permitted if the container type has a parameterless container aggregate operator "[]".

A `universal_container_aggregate` is only permitted if the container type has a universal set operator "[..]".

The `choice_list` in a named_container_element must be a set of values of the index type of the container. The expression in a container_element must be of the value type of the container.

If present in a `container_aggregate`, a default_container_element must come last. A default_container_element is only permitted when the `container_aggregate` is being assigned to an existing container object, or the index-set type of the container has a universal set operator "[..]".

In an `iterator_container_aggregate`, the iterator must not be an initial_value_iterator, and if it is an initial_next_while_iterator, it must have a `while_or_until` condition.

The evaluation of a `container_aggregate` is defined in terms of a call on a container aggregate operator "[]" or "[..]", optionally followed by a series of calls on the combining move operation `"<|="` (for `positional_container_aggregate`s) or the `"var_indexing"` operator (for `named_container_aggregate`s).

For the evaluation of an `empty_container_aggregate`, the parameterless container aggregate operator "[]" is called. For the evaluation of a `universal_container_aggregate`, the parameterless universal container aggregate operator "[..]" is called.

For the evaluation of a `positional_container_aggregate` or a `named_container_aggregate`:

- if there is a container aggregate operator "[]" which takes an index set and a mapping of index subsets to values, this is called with the index set a union of the indices defined for the aggregate, and the mapping based on the container elements specified in the `container_aggregate`. The default_container_element is treated as equivalent to the set of indices it represents.
- if there is only a parameterless container aggregate operator "[]" then it is called to create an empty container; the combining operator `"<|="` is then called for each positional_container_element in the aggregate, while the `"var_indexing"` operator is called for each named_container_element, with a `choice_list` of more than one choice resulting in multiple calls.

If there is a default_container_element, it is equivalent to a container_element with a `choice_list` that covers all indices of the overall container not covered by earlier container_elements.

For the evaluation of an `iterator_container_aggregate`, the expression is evaluated once for each element of the sequence of values produced by the iterator, with the loop variable of the iterator bound to that element. If an explicit index_expr is present, it is provided as the index to the `"var_indexing"` operator. Otherwise, the loop variable is implicitly provided as the index, unless a direction or value_filter is specified or no `"var_indexing"` operator is available, in which case the `"<|="` operator is used to build up the container value without any explicit index. If there is a value_filter, the expression is evaluated only for elements of the sequence that satisfy the value_filter.

Examples:

```
[ 1, 2, 3, 4, 5 ]                -- positional container aggregate
[ 1..5 => 1, others => 0 ]          -- named with default
[ Red => 0x1, Green => 0x10, Blue => 0x100 ]
                                 -- all named
[ for I in 0..10 when I mod 2 = 0 => I ** 2 ]  -- table of even squares
[ for I in 1..N, Key[I] => Value[I]]  -- mapping given key/value vectors
```

8.2.1 Syntactic Equivalences

For compatibility with existing SPARK code, parentheses `'(' ... ')'` may be used instead of brackets `'[' ... ']'` for container aggregates.

A container aggregate is expanded into a series of calls on operators of the container type.

```
   [A, B, C]
-- expands into:
   var Agg : Container_Type := []
   Agg <|= A; Agg <|= B; Agg <|= C

   [K1 => V1, K2 => V2, K3 => V3]
-- expands into:
   var Agg : Container_Type := []
   "var_indexing"(Agg, K1) := V1
   "var_indexing"(Agg, K2) := V2
   "var_indexing"(Agg, K3) := V3
```

A use of `others =>` expands into the set of indices of the existing container object or the universal set operator `"[..]"` not already covered by earlier choices.

An `iterator_container_aggregate` is expanded into a loop using either the `"var_indexing"` or `"<|="` operator to add elements to the container.

```
   [for I in 1..10 => I * 2]
-- if there is an "var_indexing" operator available, expands into:
   var Agg : Container_Type := []
```

```
   for I in 1..10 loop
      Agg[I] := I * 2;
   end loop
-- if only a "<|=" operator is available, expands into:
   var Agg : Container_Type := []
   for I in 1..10 forward loop
     -- NOTE: "forward" is used by default (since the set "1..10" supports it)
      Agg <|= I * 2
   end loop

   [for each V of C when V > 0 => V]
-- expands into:
   var Agg : Container_Type := []
   for each V of C when V > 0 [forward] loop
     -- NOTE: "forward" is used only if container supports it.
      Agg <|= V
   end loop

   [for X => First then X.Next while X not null => X.Data]
-- expands into:
   var Agg : Container_Type := []
   for X => First then X.Next while X not null loop
      Agg <|= X.Data
   end loop
```

8.3 Container Element Iterator

An element iterator may be used to iterate over the elements of a container.

An element iterator has the following form:

```
element_iterator ::=
    identifier [ ':' type_name ] 'of' expression
  | '[' identifier '=>' identifier ']' 'of' expression
```

An `element_iterator` is equivalent to an iterator over the index set of the container identified by the `expression`. In the first form of the `element_iterator`, in each iteration the `identifier` denotes the element of the container with the given index. In the second form of the `element_iterator`, the first identifier has the value of the index itself, and the second identifier denotes the element at the given index in the container. The `identifier` denoting each element of the container is a variable if and only if the container identified by the `expression` is a variable.

Example:

```
for each [ Key => Value ] of Table loop
    -- Iterate over key/value pairs of table
    Display(Output, Key, Value);
end loop;
```

8.3.1 Syntactic Equivalences

An element iterator is equivalent to an iterator over the `"index_set"` of the container, as follows:

```
for each [ Key => Value ] of Table loop
    ...
end loop
```

```
-- expands into:
   for Key in "index_set"(Table) loop
       ref Value => "indexing"(Table, Key)  -- i.e. Table[Key]
       ...
   end loop
```

An element iterator with only a single identifier is equivalent to the `[Key => Value]` form with an anonymous Key.

```
    for each Elem of Container loop ...
-- expands into:
    for each [Anon_Key => Elem] of Container loop ...
```

8.4 Container Specifiers

There are various operations in Sparkel for moving rather than copying objects and components of objects, such as the `"<=="` and the `"<|="` operations (see section 5.2). These can be used to reduce the amount of copying that is performed, which can be important when dealing with containers whose elements are themselves large objects. In some cases, we may build up a large object, with the intent of moving it into a container. In this case, there is some advantage to indicating, when the object is declared, that it is specifically intended to be moved into a particular container or other object when complete. This will cause its storage to be allocated in the same region as that of the specified container or other existing object.

The container or object whose region is to be used may be indicated when declaring an object or a parameter, using a `container_specifier`, whose syntax is as follows:

```
container_specifier ::= 'for' object_name
```

If a formal parameter of an operation has a `container_specifier`, then the `_specifier`'s `object_name` must identify a `var` parameter or a result of the same operation.

A parameter with a `container_specifier` is set to null as a side-effect of the call. This occurs even if the formal parameter is not specified as a `var` parameter. The actual parameter is allowed to be a constant so long as it is the last use of the constant. If the actual parameter is an expression, then the region of the named object becomes a target for the evaluation of the expression. Note that the original value of the actual parameter may be preserved by parenthesizing the actual parameter, presuming the mode of the formal parameter is not `var`. This ensures that a copy of the actual parameter is made at the point of call.

Examples:

```
-- Compute the intersection of Left and Right
-- and put result back in Left.
var Result : Set for Left := [];  -- Result in same region as Left
for Elem in Right loop
    if Elem in Left then
        Result |= Elem;           -- Add Elem to intersection
    end if;
end loop;
Left <== Result;                  -- Move result to be new value for Left.

proc Move_Into_Set(var Left : Set; Right : Element_Type for Left);
                                  -- Value of Right moved into Left,
                                  -- leaving Right null.
func Destructive_Union(Left, Right : Set for Result) return Result : Set;
                                  -- Left and Right are union'ed to
                                  -- form the Result, with Left and Right
                                  -- ending up null after the call.
```

Chapter 9

Aspect Specifications

Declarations may be annotated using `aspect_specification`s to specify a precondition of an operation, a postcondition of an operation, a predicate for a subtype or an object, or an invariant for a type.

Aspect specifications have the following form:

```
aspect_specification ::=
  'with' aspect_mark [ '=>' expression ] {','
         aspect_mark [ '=>' expression ] }
```

Preconditions, postconditions, predicates, and invariants specified by `aspect_specification`s are checked by the Sparkel compiler, and it will complain if it cannot prove that the associated conditions evaluate to True on any possible execution of the program.

```
universal_conversion ::= '[[' expression ']]'
```

An expression of the form `'[[' expression ']]'` may be used to convert an expression to a universal type, generally for use in an `aspect_specification` for a precondition or a postcondition. The type of the expression must have a `"to_univ"` operator; the type of the `universal_conversion` is the result type of this operator.

Examples:

```
func Sqrt(X : Float) return Float
  with Pre => X >= 0.0,
       Post => Sqrt'Result >= 0.0;

type Age is new Integers.Range<0..200>;
subtype Minor is Age with Predicate => Minor < 18;
subtype Senior is Age with Predicate => Senior >= 50;
```

These `aspect_specification`s define predicates on two different subtypes of the Age type.

```
generic
   Modulus : Univ_Integer with Predicate => Modulus >= 2;
package Modular_Types is
   type Mod is private;
   func "from_univ"(Univ : Univ_Integer)
     return Modular
     with Pre => Univ in 0 ..< Modulus;

   func "to_univ"(Val : Modular) return Result : Univ_Integer
     with Post => Result in 0 ..< Modulus;
```

```
    func "+"(Left, Right : Modular) return Result : Modular
      with Post => [[Result]] = ( [[Left]] + [[Right]] ) mod Modulus;
    ...
end Modular_Types;
```

The precondition on `"from_univ"` indicates the range of integer literals that may be used with a modular type with the given modulus. The postcondition on `"to_univ"` indicates the range of values returned on conversion back to Univ_Integer. The postcondition on `"+"` expresses the semantics of the Modular `"+"` operator in terms of the language-defined operations on Univ_Integer.

Here is a longer example:

```
generic
    type Component is private;
    type Size_Type is new Countable<>;
package Stacks is
    type Stack is private;
    func Max_Stack_Size(S : Stack) return Size_Type;
    func Count(S : Stack) return Size_Type;

    func Create(Max : Size_Type) return Stack
      with Pre => Max > 0,
           Post =>  Max_Stack_Size(Create'Result) = Max
             and Count(Create'Result) = 0;

    proc Push
      (var S : Stack;
       X : Component)
       with Pre => Count(S) < Max_Stack_Size(S),
            Post => Count(S) = Count(S)'Old + 1;

    func Top(ref S : Stack) return ref Component
      with Pre => Count(S) > 0;

    proc Pop(var S : Stack)
      with Pre => Count(S) > 0,
           Post => Count(S) = Count(S)'Old - 1;
end Stack;

package body Stacks is
    type Stack is record
      const Max_Len : Size_Type;
      Cur_Len : Size_Type
        with Predicate => Cur_Len in 0..Max_Len;
      Data : Array<optional Component, Indexed_By => Size_Type>
        with Predicate => Length(Data) = Max_Len;
    end record
      with Type_Invariant => (for all I in 1..Cur_Len => Data[I] not null);
        -- invariant needed for Top()

    func Max_Stack_Size(S : Stack) return Size_Type is
       return S.Max_Len;
    end func Max_Stack_Size;

    func Count(S : Stack) return Size_Type is
      return S.Cur_Len;
    end func Count;
```

```
    func Create(Max : Size_Type) return Stack is
      return (Max_Len => Max, Cur_Len => 0, Data => [1.. Max_Len => null]);
    end Create;

    proc Push
      (var S : Stack;
       X : Component) is
      S.Cur_Len += 1;
      S.Data[S.Cur_Len] := X;
    end Push;

    func Top(ref S : Stack) return ref Component is
      return S.Data[S.Cur_Len];
    end func Top;

    func Pop(var S : Stack) is
        S.Cur_Len -= 1;
    end Pop;
end Stacks;
```

Chapter 10

Protected Objects and Parallel Execution

Expression evaluation in Sparkel proceeds in parallel (see 6.4), as do statements separated by '||' (see 5.1), and the iterations of a parallel loop (see 5.6 and 5.6.1). The Sparkel implementation ensures that this parallelism does not introduce *race conditions*, situations where a single object is manipulated concurrently by two distinct threads without sufficient synchronization. A program that the implementation determines might result in a race condition is illegal.

Objects in Sparkel are either *protected* or *unprotected*, according to whether their type is or is not a *protected* type. Protected objects allow concurrent operations by multiple threads by using appropriate hardware or software synchronization. An unprotected object allows concurrent operations only on non-overlapping parts of the object.

10.1 Protected Types

A type is *protected* if it has the reserved word `protected` in its definition, or if it is derived from a protected type.

Example:

```
generic
   type Item_Type is new Machine_Integer<>;
package  Lock_Free is
   type Atomic is protected private;
   func Create(Initial_Value : Item_Type) return Atomic;
   func Test_And_Set(var X : Atomic) return Item_Type;
       -- If X = 0 then set to 1; return old value of X
   func Compare_And_Swap(var X : Atomic;
      Old_Val, New_Val : Item_Type) return Item_Type;
       -- If X = Old_Val then set to New_Val; return old value of X
end Lock_Free;
...
var X : Lock_Free.Atomic<Int_32> := Create(0);
var TAS_Result : Int_32 := -1;
var CAS_Result : Int_32 := -1;
block
    TAS_Result := Test_And_Set(X);
  ||
    CAS_Result := Compare_And_Swap(X, 0, 2);
end block;
```

```
-- Now either TAS_Result = 0, CAS_Result = 1, and X is 1,
-- or TAS_Result = 2, CAS_Result = 0 and X is 2.
```

This is an example of a package which defines a protected atomic type whose objects can hold a single Machine_Integer, and can support concurrent invocations by multiple threads of Test_And_Set and Compare_And_Swap operations. The implementation of this type would presumably use hardware synchronization.

10.1.1 Locked and Queued Operations

The operations of a protected type may include the reserved word `locked` or `queued` for parameters of the type. If a protected type has any operations that have such parameters, then it is a *locking* type; otherwise it is *lock-free*. Any object of a locking type includes an implicit *lock* component.

If an operation has a parameter that is marked `locked`, then upon call, a lock is acquired on that parameter. If it is specified as a `var` parameter, then an exclusive read-write lock is acquired; if it is not specified as a `var` parameter then a sharable read-only lock is acquired. Once the lock is acquired, the operation is performed, and then the caller is allowed to proceed.

If an operation has a parameter that is marked `queued`, then the body of the operation must specify a *dequeue* condition. A `dequeue_condition` has the following form:

```
dequeue_condition ::= 'queued' while_or_until condition 'then'
```

A dequeue condition is *satisfied* if the condition evaluates to True and the reserved word `until` appears, or if the condition evaluates to False and the reserved word `while` appears.

Upon call of an operation with a `queued` parameter, a read-write lock is acquired, the dequeue condition of the operation is checked, and if satisfied, the operation is performed, and then the caller is allowed to proceed. If the dequeue condition is not satisfied, then the caller is added to a queue of callers waiting to perform a queued operation on the given parameter.

Within an operation of a protected type, given a parameter that is marked `locked` or `queued`, the components of that parameter may be manipulated knowing that an appropriate lock is held on that object. If there is a protected type parameter that is *not* marked `locked` or `queued`, then there is no lock on that parameter, and only protected components of such a parameter may be manipulated directly.

If upon completing a locked or queued operation on a given object, there are other callers waiting to perform queued operations, then before releasing the lock, these callers are checked to see whether the dequeue condition for one of them is now satisfied. If so, the lock is transferred to that caller and it performs its operation. If there are no callers whose dequeue conditions are satisfied, then the lock is released, allowing other callers not yet queued to contend for the lock.

If an operation declared in the visible part of a package performs a call on a queued operation internally, but does not have a queued parameter, then the operation as a whole must be marked with the reserved word `'queued'` prior to the reserved word `'func'` or `'proc'` (see 6.1). This indicates that an indefinite delay within the operation might occur, while waiting for the dequeue condition associated with some call to be satisfied. Such operations must not be called from within a locked operation, as they could cause a lock to be held indefinitely. On the other hand, operations with a parameter explicitly marked as `'queued'` may be called while already holding a lock on that parameter, but the dequeue condition must already be satisfied at the point of call.

Example:

```
generic
    type Element_Type is private;
package Queue is
    type Queue is protected private;
    func Create return Queue;
    proc Append(locked var Q : Queue; Elem : Element_Type);
    func First(locked Q : Queue) return optional Element_Type;
```

```
      -- Returns null if queue is empty.
    func Remove_First(queued var Q : Queue) return Element_Type;
      -- Queued until the queue has at least one element
end Queue;
...
var Q : Queue<Int32> := Create();
var A : Int32 := 0;
var B : Int32 := 0;
block
    Append(Q, 1); Append(Q, 2);
 ||
    A := Remove_First(Q);
 ||
    B := Remove_First(Q);
end block;
-- At this point, either A = 1 and B = 2
-- or A = 2 and B = 1.
```

In this example, we use a locking type Queue and use locked and queued operations from three separate threads to concurrently add elements to the queue and remove them, without danger of unsynchronized simultaneous access to the underlying queuing data structures.

Note: operations of a Sparkel protected type with a queued parameter are similar to Ada's protected entries, with the dequeue condition being analogous to the entry barrier. However, a dequeue condition in Sparkel may depend on the value of any parameter to the queued operation, whereas in an Ada protected entry, the barrier may not depend on any parameter, though it may depend on an entry family index, if any.

10.2 Parallel Evaluation

Two expressions that are parameters to an operation call (see 6.4) or a binary operator (see 4.2.2) are evaluated in parallel in Sparkel, as are the expressions that appear on the right hand side of an assignment and those within the `object_name` of the left hand side (see 5.2). In addition, the separate `statement_thread`s of a `statement_thread_group` (see 5.1) are performed in parallel. Finally, the iterations of a parallel loop (see 5.6) are performed in parallel.

Two `object_name`s that can be part of expressions or statements that are evaluated in parallel must not denote overlapping parts of a single unprotected object, if at least one of the names is the left-hand side of an assignment or the actual parameter for a `var` parameter of an operation call. Distinctly named components of an object are non-overlapping. Elements of a container associated with distinct indices are non-overlapping (see 8.1).

Examples:

```
func Bump(var A : Int) return Int;
X := 3 || X := 5          -- illegal
X := 3 || Y := X          -- illegal
A := X || B := X          -- legal
A[I] := 2 || A[J] := 3    -- illegal if I can equal J
Bump(X) + X               -- illegal
X := Bump(X)              -- legal
```

Chapter 11

Sparkel Source Files and Standard Library

11.1 Sparkel Source Files

Each Sparkel source file is made up of a sequence of standalone package or operation definitions. `With_clause`s can be used to specify which other packages or operations are visible when defining a given standalone package or operation.

```
source_file ::=
   { compilation_unit }

compilation_unit ::= context_clause standalone_program_unit

context_clause ::= { with_clause [ use_clause ] }

with_clause ::= 'with" program_unit_name { ',' program_unit_name } ';'

use_clause ::= 'use' package_name ';'

standalone_program_unit ::=
  package_declaration | package_body | operation_definition
```

A `with_clause` may be used to control which other standalone program units are visible within a given `standalone_program_unit`. A `use_clause` makes the declarations within a (non-generic) package specification directly visible. For a generic package, a `use_clause` makes the generic types within the package specification directly visible.

Note: There is no "use type" clause in Sparkel. See the description of "operation_name" resolution in 6.4.

11.2 Sparkel Syntax Shorthands

[*Possible addition:* Sparkel syntax is not as rigid as implied by the BNF given in this reference manual. In particular, semicolons at the end of a statement or declaration may be omitted, and "end XXX" at the end of a construct may be omitted, so long as proper indentation is used.

For example, the following is a legal use of these shorthands:

```
    func Fib (N : Integer) return Integer is
       -- Recursive fibonacci but with linear time

       func Fib_Helper(M : Integer)
         return (Prev_Result : Integer; Result : Integer) is
         -- Recursive "helper" routine which
         -- returns the pair ( Fib(M-1), Fib(M) )
           if M <= 1 then
               -- Simple case
               return (Prev_Result => M-1, Result => M)
           else
               -- Recursive case
               const Prior_Pair := Fib_Helper(M-1)

               -- Compute next fibonacci pair in terms of prior pair
               return with
                 (Prev_Result => Prior_Pair.Result,
                  Result => Prior_Pair.Prev_Result + Prior_Pair.Result)

       end Fib_Helper  -- This is optional

       -- Just pass the buck to the recursive helper function
       return Fib_Helper(N).Result

]
```

11.3 Sparkel Standard Library

Sparkel includes a number of language-provided packages in the Sparkel Standard Library, with names of the form `SSL.Core.*` and `SSL.Containers.*`.

SSL.Core itself is a non-generic package, and provides the predefined types of the language:

Univ_Integer arbitrary length integers

Univ_Real ratio of two Univ_Integers, with plus/minus zero and plus/minus infinity

Univ_Character 31-bit ISO-10646 (Unicode) characters

Univ_String vector of Univ_Characters

Boolean an enumeration type with two values False and True

Ordering an enumeration type with four values Less, Equal, Greater, and Unordered

The other language-provided types include:

type Any is abstract limited private all types implement Any implicitly

type Assignable is abstract private provides `":="`, `"<=="`, and `"<=>"` operations; all non-limited types implement Assignable implicitly

type Comparable is abstract limited private provides `"=?"` operator

type Hashable is abstract private implements Assignable and Comparable; provides `Hash` operation

type Countable is abstract private implements Hashable; provides `"+"` and `"-"` operators to add or subtract Univ_Integers to progress through the values of the type

type Imageable is abstract private implements Hashable; provides `To_String` and `From_String` functions to convert the value to and from a Univ_String.

type Vector<Element_Type is Assignable<>> an extensible array indexed by Univ_Integer starting at 1

type ZVector<Element_Type is Assignable<>> an extensible array indexed by Univ_Integer starting at 0

type ZString<> very similar to Univ_String, except that indexing starts at 0 rather than 1

type Interval<Bound_Type is Countable<>> intervals are constructed using the `".."`, `"..<"`, `"<.."`, and `"<..<"` operators with a low and high bound specified by values of the Bound_Type

type Set <Element_Type is Hashable<>> a set of Element_Type

type Map <Key_Type is Hashable<>; Element_Type is Assignable<>> a map from Key_Type to Element_Type

type Enum <Vector<Univ_Enumeration>> implements Countable and Imageable; used to define a new enumeration type given the vector of literals; see Syntactic Equivalences subsection of 3.1

type Enum_With_Rep <Map<Univ_Enumeration, Univ_Integer>> implements Imageable; used to define a new enumeration type given a map from each literal to its underlying value; see Syntactic Equivalences subsection of 3.1

type Integer <Interval<Univ_Integer>> implements Countable and Imageable; provides the usual operators

type Float<Univ_Integer> implements Hashable and Imageable; provides the usual operators

type Array<Element_Type is Assignable<>; Indexed_By is Countable<>> a fixed-size array of Element_Type, indexed by a specified countable type

Chapter 12

Appendix: A To-Do List for the Sparkel Reference Manual

- Decide whether we want to allow a private part, and private child units that can see it, or instead just rely on nested packages to provide that functionality.
- Need more general description of using T'X where T is a type and X denotes a declaration occurring in the enclosing (generic) package. The question is whether this is usable with any sort of declaration, including an object, type, or nested package declaration. If a type T1 extends T, then presumably T1'X should also be legal. We need to decide what it means. A simple rule is that if X denotes anything other than an operation of the type, T1'X and T'X are equivalent. That is, effectively all "attributes" are inherited, with only the operation attributes being overridable. But that doesn't really work if the properties of the entity are a function of T, and hence should be substituted with T1 on inheritance. At a minimum we need access to the generic actuals (of the package enclosing the type), if any.

  Anything that is overridable will necessarily need a "slot" in the type/instance descriptor. Actually, in ParaSail we include "nested types" in the type/instance descriptor, so that if you declare `"func "in"(Elem : T; Within : Set<T>)"` that you have a type/instance descriptor for `"Set<T1>"` ready-made at some known slot within the type/instance descriptor for "T1". Clearly you can't inherit code for an operation that involves one of these "nested" types, just as you can't inherit code for an operation that constructs an extended object. Such an operation becomes abstract on extension (derivation without extension could be more friendly as far as inheriting code), and needs to be defined in each extension. Operations that are defined by equivalence, however, should work, with the corresponding substitution. They presumably could be overridden, but wouldn't need to be.
- We need to think about the type/instance descriptor a bit, even though that should be strictly an implementation issue. It shows through into the language reference manual because clearly we need to be able to implement the language! In ParaSail, the type descriptor is effectively the "static link" when calling an operation, and provides access to global constants, module parameters, etc. In Sparkel, the type/instance descriptor would be for the whole package instance, presumably, and if there are multiple types defined in the package, they could all share the same instance descriptor if they have the same generic actuals.
- Generic subprograms (operations?) and instantiations thereof should be described. What about implicit instantiation, as in ParaSail? In ParaSail, operations can have generic formal types as their parameter types, and a parameter itself can be treated as a generic formal object, as in the example of exponentiation where the exponent needs to be a generic formal object to do units checking.
- We should describe the equivalence between `"new Any<>"` and "limited private" and `"new Assignable<>"` and "private".

- Sparkel should disallow using type-instantiation syntax with a type nested in a generic package with global variables, because anonymous/implicit instantiations of a package with global variables is problematic – you don't know how many copies of the global variables you will end up with. Explicit package instantiation is needed first to create a non-generic package. One consequence is that all types represented by a polymorphic type use the same set of global variables. This seems less important for types represented by a generic formal type. (General discussion of polymorphic types vs. generic formal types would be interesting.)

- We should probably disallow deriving from a generic formal type. The issue is that it might be polymorphic. In any case, it is kind of a weird thing to do. It makes more sense to use a generic formal type as a component of a type defined inside the generic.

- Probably should disallow global variables being declared in package bodies, to avoid the problems SPARK has with private state. We need to think about initialization of global variables (unfortunately). Hopefully we can do it all statically (presuming we can execute user-defined code statically). It might be simplified by disallowing package-body "begin" parts, meaning that global variables must be initialized at their point of (re)declaration. We might also require that the initial value for a global variable be computable at compile time, that is, not reference any other global variable, and certainly not *update* any other (global) variable as a side-effect.

- Should global_specification allow specifying a component of a global? If all globals are protected, then presumably they don't have visible components (though we could conceivably allow globals of a record type consisting only of protected (or const) components).

- Should mention optional private part of a package, and its visibility to child packages, and what can be postponed to package body (in particular, full type definitions and full object initializations). Do we need to have "private" and "limited" withs? We probably need to define child packages somewhere. We need to say what packages, if any, are implicitly "with"ed and or "use"ed.

- Should talk about pragmas somewhere. Should "assert" be its own kind of statement? And how do loop invariant/variants fit in? Those would seem to be better as aspects of a loop. Alternatively, could use aspects on an assert statement to qualify the kind of assertion. Assume statement is `"assert Blah with No_Proof;"`? Specify message as `"assert Blah with Message => "Disaster";"`?

- What about ambiguity between "with_values" and aspect specifications? If we start allowing aspect specifications on statement-ish things (like loops) this could be quite confusing. Alternatives for "with_values"? E.g. `"continue loop and then X => X.Left"`. But perhaps if we think of aspect specifications as defining values for attributes, which might be of an operation type (as with Pre and Post), there isn't such a disconnect.

- Think about the Array_Types.Array generic type, and how to handle multi-dimensional arrays. The current notion that a tuple type can be used is a bit naive. We need a way to compute the index into the underlying Basic_Array (or whatever we have), a way to iterate through all possible index values, etc. Your average tuple type doesn't have all of that. We probably need a type constructor to turn two countable types into a single countable type, or just provide separate 2-D, 3-D, etc. array type constructors. Conceivably we could allow resolving between two generic packages both with nested types called Array, based on the number of generic parameters.

  Alternatively, we say that if all of the components of a tuple type have a "[..]" operator and possibly a Predicate, then the overall tuple (or record) type has a "[..]" operator which produces a set of tuples, taking into account the "[..]" and Predicates of the components. We also may need the "+"/"-" operators associated with Countable types, to map down to integers for indexing into an underlying single-dimensional array. We also need to think about the constructor for a particular array object. That would presumably take a countable range for each index. An initialized constant constructor would want a mapping from index tuple to value, or a flat vector of values plus the bounds vector.

The other approach is the array-of-arrays approach. In some ways that would be the simplest, where syntactic sugar would be applied to break A[1,2,3] into A[1][2][3], and to the original array type definition to turn it into an array-of-array type definition. One down-side of an array-of-array is that almost certainly it would add a level of indirection for each dimension.

- No hiding of operations of a type with another matching operation, and no local declarations of operators seems like a good rule. Note that if our "operation" parameters are actually normal parameters of an operation type, then we don't need to specially disallow parameters with an operator-symbol as their designator, since these aren't permitted for "normal" parameters.

  More generally should we treat a global const as equivalent to a parameterless (global-free) func, and allow overloading, and have them inherited and be overridable? Yes, probably.

  Conceivably we could treat global variables as parameterless funcs which returned a ref, but they would have to have some kind of "ref global" as an input, so that would sort of defeat the purpose. However it would conceivably allow inheritance and overriding, but this doesn't seem like a feature.

- Do we want to allow operation types without any global references as being assignable, i.e. non-limited? Alternatively, we only allow this for named operation types (as in Ada), and perhaps with no globals. For operation types, it is unclear whether "global" means library level or simply up-level, and if it means up-level, whether it includes unprotected objects. Should this be determined by named vs anonymous, where for a named operation type, global means library-level, and necessarily protected, and disallows passing operations with non-library-level up level references? Seems like we should be explicit, rather than making subtle distinctions here... We could put "protected" explicitly in the global specification to limit it. One reason we care is whether we can evaluate a call via an object of an operation-type in parallel with itself, or with other expressions that use up-level variables.

- Do we want to require that generic actuals be computable at compile time? That eliminates the need to create type/instance descriptors at run-time, which seems like a good thing. It also simplifies proofs, I would think.