ParaSail Reference Manual – Draft

S. Tucker Taft

June 4, 2011

Contents

Chapter 1

Introduction

ParaSail stands for “Parallel Specification and Implementation 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 ParaSail, 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. ParaSail supports both lock-based and lock-free concurrent objects.

ParaSail also supports annotations, and in fact requires them in some cases if they are needed to prove 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 compile-time checks in ParaSail. 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 inputs, then a precondition is required to prevent such inputs from being passed to the operation. ParaSail 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.

ParaSail has only four basic concepts – Modules, Types, Objects, and Operations. Every type is an instantiation of a module. An object is an instance of some type. An operation operates on objects.

There are no global variables. Any object to be updated by an operation must be an explicit input or output to the operation.

ParaSail has user-defined indexing (analogous to arrays or tables), user-defined literals (integers, reals, strings, characters, and enumerations), user-defined “aggregates,” etc. Every type is the instantiation of some module, including those that might be considered the built-in types, and there are no “special” operators or constructs that only a built-in type can utilize.

ParaSail has no pointers, though it has references, optional and expandable objects, and user-defined indexing, which together provide a rich set of functionally equivalent capabilities without any hidden aliasing nor any hidden race conditions.

Chapter 2

Lexical Elements

2.1 Character Set

ParaSail 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 delimeters in ParaSail:

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

The following combinations of graphic characters are delimeters in ParaSail:

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

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

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

2.3 Identifiers

Identifiers start with a letter, and continue with letters, digits, and underscores.

```
identifier ::= letter { _ | letter | digit }
```

Upper and lower case is significant in identifiers. Letters include any graphic character in the ISO-10646 character set that is considered a letter. An identifier must not be the same as a ParaSail reserved word (see 2.6).

Examples:

```
X, A__B, a123, A123, This_Is_An_Identifier, Xyz_
```

2.4 Literals

There are five kinds of literals in ParaSail: integer, real, character, string, and enumeration.

```
literal ::=
    integer_literal
  | real_literal
  | character_literal
  | string_literal
  | enumeration_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
  | binary_integer_literal
  | hex_integer_literal
  | based_integer_literal

decimal_integer_literal ::= decimal_numeral

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

hex_integer_literal ::= 0 (x|X) hex_digit { [_] hex_digit}

based_integer_literal ::= decimal_numeral # extended_numeral #


decimal_numeral ::= digit { [_] digit }

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

Examples:

```
42, 1_000_000, 0xDEAD_BEEF, 8#0177#
```

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 other than apostrophe and backslash, or a single escaped_character, being a backslash followed by an escapable_character.

Character literals are of type Univ_Character.

```
character_literal ::= ' unescaped_character ' | ' escaped_character '

escaped_character ::= \ escapable_character

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
```

Examples:

```
 'a', '0', '\'', '\r'
```

2.4.4 String Literals

String literals are a sequence of graphical characters of the ISO-10646 character set enclosed in double quotes. The backslash and double-quote characters may appear only as part of an escaped_character.

String literals are of type Univ_String.

```
string_literal ::= " { unescaped_character | escaped_character } "
```

Example:

```
"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.
Enumeration literals are of type Univ_Enumeration.

```
enumeration_literal ::= # ( identifier | reserved_word )
```

Examples:

```
#red, #true, #Monday
```

2.5 Comments

Comments in ParaSail 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 ParaSail:

```
abs                   exit                  locked                ref
abstract              extends               loop                  rem
all                   exports               mod                   return
and                   for                   mutable               reverse
block                 forward               new                   select
case                  function              not                   some
class                 global                null                  then
concurrent            if                    of                    until
const                 implements            operator              var
continue              import                optional              while
each                  in                    or                    with
else                  interface             private               xor
elsif                 is                    procedure
end                   lambda                queued
```

All reserved words in ParaSail are in lower case.

Chapter 3

Types and Objects

In ParaSail, every object is an instance of some type, and every type is defined by instantiating a module and/or applying a constraint to an existing type.

3.1 Types

A type is declared by instantiating the interface of a module (see 7.1), or by constraining an existing type, using the following syntax:

```
type_declaration ::=
  'type' identifier 'is' ['new'] type_specifier [ constraint_annotation ] ';'

type_specifier ::= type_name | module_name '<' module_actuals '>'

constraint_annotation ::= annotation
```

See Chapter 9 for the syntax of an annotation.

The presence of 'new' in a type_declaration indicates that the type is not equivalent to any other type. If no 'new' is specified, then the type is *value-equivalent* to any other instantiation of the same module with *value-equivalent* actuals. The type is *constraint-equivalent* to any other instantiation of the same module with *constraint-equivalent* actuals, and with the same constraint annotation, if any. A constraint_annotation does not create a new type per se, but instead represents a constrained subtype of a type, with the constraint(s) determining which values belong to the subtype. In other words, *name equivalence* is used between two types if either was declared with the reserved word 'new.' Otherwise *structural equivalence* applies, where the basic structure is determined by the actuals supplied to the module instantiation, and the subset of values of the type is determined by the constraint annotations, if any.

Two types are considered *significantly* different if they result from instantiating different modules, or if they result from distinct instantiations of the same module at least one of which included the 'new' reserved word.

Example:

Given the interface of a List module defined as follows (see 7.1):

```
interface List <Element_Type is Assignable<>> is
    function Create() -> List;
    function Is_Empty(L : List) -> Boolean;
    procedure Append(L : ref var List; Elem : Element_Type);
    function Remove_First(L : ref var List) -> optional Element_Type;
    function Nth_Element(L : ref List; N : Univ_Integer)
      -> ref optional Element;
```

```
end interface List;
```

A specific kind of list may be declared as follows:

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

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

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 initial value of the data of the object cannot be changed during the life of the object.

An object is declared using the following syntax:

```
object_declaration ::=
    uninitialized_object_declaration
  | initialized_object_declaration

uninitialized_object_declaration ::=
  var_or_const identifier ':' object_type ';'

initialized_object_declaration ::=
  var_or_const identifier [ ':' object_type ] ':=' expression ';'

var_or_const ::= 'var' | 'const'

object_type ::= object_qualifier type_specifier [ constraint_annotation ]

object_qualifier ::= [ 'optional' ]  [ 'concurrent' ]
```

The value of an object may be null only if it is declared to have an `'optional'` type. An uninitialized object with an `'optional'` type has the null value initially.

An uninitialized object that has a non-optional type must be assigned a value prior to being referenced. An uninitialized constant object may be assigned a value at most once, and if it has an optional type, must not be assigned a value after its (null) value is referenced.

Examples:

```
var BL : Bool_List := Create();
const T : Boolean := #true;
var Result : optional T;
```

These declare a variable boolean list, a constant with Boolean value #true, and a variable Result with implicit initial value of null.

TBD: mutable objects.

3.3 Object References

A reference to an existing object is declared using the following syntax:

```
object_reference ::=
  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.

Examples:

```
const First => Nth_Element(L, 1);
var Elem => Nth_Element(M, I);
```

These create a read-only reference to the first element of L, and a read-write reference to the Ith element of M.

3.4 Declarations and Identifiers

The identifier introduced by the declaration of a type or object must not denote a currently visible declaration. However, when declaring a module formal, an operation input, or a function output, the identifier may be omitted, in which case it is taken, in the case of a type formal, from the module name, in the case of an input to an operation, from the type name, and in the case of a function output, from the function name. In addition, when inside a module, the module's simple name also identifies a type which is the current instance of the module.

The identifier introduced by the declaration of an operation must not denote a currently visible interface, type, or object, but may be the same as that of an existing operation, provided it differs *significantly* in the types of one or more inputs or outputs (see 3.1 for definition of *significantly* different types).

The full name of a module must be unique within a given program.

Chapter 4

Names and Expressions

4.1 Names

Names denote modules, types, objects, and operations.

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

module_name ::= [ module_name '::' ] identifier

type_name ::= type_identifier  [ '+' ]

type_identifier ::= [ type_identifier '::' ] identifier

object_name ::=
    identifier
  | object_indexing_or_slicing
  | operation_call
  | component_selection
```

See Operation Calls (Section 6.3) 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_declaration occurs immediately within the interface (see 7.1) or class (see 7.3) for a module, and the declaration is not for an initialized 'const' object, then it declares a *component* object. Components declared within a module comprise the data of each object of a type based on the module.

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
  | 'null'
  | object_name
  | postcondition_value
  | unary_operator expression
  | expression binary_operator expression
  | membership_test
  | null_test
  | quantified_expression
  | type_conversion
  | [ type_identifier '::' ] bracketed_expression

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

Literals evaluate to a value of a corresponding universal type, and are implicitly convertible to a type that has a corresponding `"from_univ"` operator, so long as the value satisfies the precondition of the operator.

The reserved word `'null'` evaluates to the null value, which can be used to initialize any object declared to have an `'optional'` type.

A type_identifier followed by `'::'` 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.

See Annotations (Chapter 9) for the syntax of postcondition_value and universal_conversion.

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 Unary and Binary Operators

The following are the unary operators in ParaSail:

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

The following are the binary operators in ParaSail:

```
"**"                    -- Exponentiation

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

"+", "-"                -- Addition and subtraction
```

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

"|"                       -- Used to combine elements into a container

"<", "<=", "==",          -- The usual relational operators
"!=", ">=", ">"
"=?"                      -- 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 and subtraction 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, and division 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 inputs, 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 6.3). The short-circuit boolean operators `"and then"` and `"or else"` and the implication operator `"==>"` are implemented in terms of the corresponding `if_expression` (see 4.2.6):

```
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
```

4.2.2 Membership and Null Tests

A membership test is used to determine whether a value can be converted to a type, satisfies the constraints of a type, 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 Color     // True if #red is convertible to Color
Z not null        // True if Z does not have a null value
```

4.2.3 Other ParaSail 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
"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 by "( operation_actuals )" to create an object from components
```

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
(A => 25, B => #true)
                    // An anonymous object with given values for its components
                    // using "()" operator
```

4.2.4 Aggregates

Aggregates are used for constructing values out of their constituents. There are two kinds of aggregates: the class_aggregate for creating an object of a 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 class_aggregate is generally only available when inside the class defining a module, or for a type based on a module that has only components declared in its interface. In addition, if the "()" operator is explicitly declared in the interface of a module, then the class_aggregate may be used.

Aggregates have the following form:

```
aggregate ::= class_aggregate | container_aggregate

class_aggregate ::= '(' operation_actuals ')'
```

See 6.3 Operation Calls for the syntax of operation_actuals. See 8.2 Container Aggregates for the syntax of a container_aggregate.

Examples:

```
(X => 3.5, Y => 6.2)        // fully named class_aggregate

(Element, Next => null)  // mixed positional and named class_aggregate
```

4.2.5 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 ::=
  index_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);
```

4.2.6 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
```

An if_expression has one of two alternative syntaxes:

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

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

All expressions of an if_expression must be null or implicitly convertible to the same type.

To evaluate an if_expression, the conditions are evaluated in sequence, and the first one that evaluates to true determines the expression to be evaluated (the one following the '?' or corresponding 'then'). If all of the conditions evaluate to false, the last expression of the if_statement is evaluated to produce the value of the if_expression.

Examples:

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

return Y == 0? null : X/Y;  // return null if would divide by zero
```

Case expressions have the following form:

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

case_expression_alternative ::=
    '[' choice_list ']' '=>' expression
  | '[' identifier ':' type_name ']' '=>' expression

case_expression_default ::=
  '[..]' '=>' 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 null or 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_alternatives must cover all possible values of the case_selector. The sets associated with the case_alternatives must be disjoint with one another.

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 of
   [#less] => Search(Node.Left, Key);
   [#equal] => Node.Value;
   [#greater] => Search(Node.Right, Key));
```

4.2.7 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 constraints on B.

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

- Types A and B are instances of the same module with value-equivalent (see 3.1) actuals (even if one of them is a 'new' type);
- Type B is a polymorphic type (see 7.2.1), and type A is an instance of a module that extends or implements the root interface of B, with value-equivalent actuals;
- Type A is a polymorphic type, and the type-id 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 input type of the `"from_univ"` operator;
- Type A or type B has a `"convert"` operator that has an input 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 ParaSail 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 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.

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

Examples:

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

The first two statements runs 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
    | class_aggregate ':=' expression
    | object_name operate_and_assign expression
```

There are builtin operations for simple assignment and for swapping the content of two objects:

```
  ":="   -- simple assignment of right-hand-side into left-hand-side
  ":=:"  -- swap left and right hand content
```

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

In addition, several of the binary operators may be combined with "=" to produce an operate-and-assign operation:

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

Examples:

```
X := A + B;           // Set X to sum of A and B
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
```

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 'of'
      case_alternative
    { case_alternative }
    [ case_default ]
  'end' 'case' [ statement_identifier ] [ with_values ]

case_selector ::= expression

case_alternative ::=
    '[' choice_list ']' '=>' statement_list
  | '[' identifier ':' type_name ']' '=>' statement_list

choice_list ::= choice { '|' choice }

choice ::= expression [ interval_operator expression ]

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

case_default ::=
  '[..]' '=>' 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_alternatives must cover all possible values of the case_selector. The sets associated with the case_alternatives must be disjoint with one another.

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) of
  ['a'..'z' | 'A'..'Z'] =>
     Handle_Alphabetic(Input);
  ['0'..'9'] =>
     Handle_Numeric(Input);
  ['\n'] =>
     Handle_End_Of_Line(Input);
  ['\0'] =>
     Handle_End_Of_Input(Input);
  [..] =>
     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.

5.6 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 [ direction ] loop_body
  | 'for' '(' iterator_list ')' [ direction ] loop_body

direction ::= 'forward' | 'reverse' | 'concurrent'

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

iterator_list ::= iterator { ';' iterator }

iterator ::=
    index_iterator
  | 'each' element_iterator
  | initial_next_while_iterator
  | initial_value_iterator

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

initial_next_while_iterator ::=
  initial_value_iterator [ next_values ] while_or_until condition

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

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

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 an index_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.

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.

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 concurrent loop
    X[I] := I ** 2;
end loop;
```

The above loop initializes a table of squares in parallel.

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

The above loop prints the names of the students in the given Classroom in the order returned by the List_Of_Students function.

```
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 concurrently, 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 concurrently with calling Process on X.Data. (The example given in 5.5 could be made exactly equivalent by making that loop into a `concurrent 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 : Vector<Solution> := [];
  *Outer_Loop*
   for (C : Column := 1; Trial : Solution := Empty()) loop

        for R in Row concurrent 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.

5.7 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 ] [ 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 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.

Chapter 6

Operations

Operations are used to specify an algorithm for computing a value or performing a sequence of actions. There are three kinds of operations – functions, procedures, and operators. Operators have special meaning to the language, and are invoked using special syntax. Functions produce one or more outputs. Most operators also produce an output. Procedures generally update one or more of their variable inputs.

6.1 Operation Declarations

Operations are declared using the following forms:

```
operation_declaration ::=
  function_declaration | procedure_declaration | operator_declaration

function_declaration ::=
  [ 'abstract' | 'optional' ] 'function' identifier inputs '->' outputs

procedure_declaration ::=
  [ 'abstract' | 'optional' ] 'procedure' identifier inputs

operator_declaration ::=
  [ 'abstract' | 'optional' ] 'operator' operator_symbol inputs [ '->' outputs ]

operator_symbol ::= string_literal

inputs ::= input | '(' [ input { ';' input } ] ')'

input ::= formal_object | formal_operation

formal_object ::=
  [ identifier ':' ] [ input_mode ] formal_object_type [ ':=' expression ]

input_mode ::=
    'ref' [ var_or_const ]
  | 'global' var_or_const
  | 'locked' var_or_const
  | 'queued' var_or_const
```

```
formal_object_type ::= object_type | identifier 'is' module_instantiation

formal_operation ::=
  operation_declaration [ 'is' operation_specification ]

operation_specification ::=
  operation_name | lambda_expression


outputs ::= output | '(' output { ';' output } ')'

output ::=
  [ identifier ':' ] [ output_mode ] formal_object_type

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

If an identifier is omitted for an input, the type_name 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 6.2) all inputs must either have an identifier, or have a unique type name.

If an identifier is omitted for an output of a function, and there is only one output, the function identifier may be used to identify the output. If there is more than one output, each one must have an identifier.

If a formal_object_type is of the form `identifier 'is' module_instantiation`, the actual parameter may be of any type that matches the module_instantiation (see 7.4). The specified identifier refers to the type of the actual parameter within the operation_declaration, and within the corresponding operation_definition.

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

An output of mode 'ref' must be specified via a return statement as a reference to all or part of an input of mode 'ref'. An output of mode 'ref' 'var' must be specified via a return statement as a reference to all or part of an input of mode 'ref' 'var'. An output of mode 'ref' 'const' must be specified via a return statement as a reference to all or part of some 'ref' input ('var', 'const', or merely 'ref').

The designator of a formal_operation may not be the same as that of an already defined operation with the same number of inputs and outputs, unless its parameter types are significantly different from the preexisting operation.

Examples:

```
function Sin (X : Float) -> Float;

operator "=?" (Left, Right : Set) -> Ordering;

function Divide ( Dividend : Integer; Divisor : Integer)
  -> (Quotient : Integer; Remainder : Integer);

procedure Update(Obj : ref var T; New_Info : Info_Type);

operator "*"(Left : Float_With_Units;
   Right : Right_Type is Float_With_Units<>)
  -> (Result : Result_Type is Float_With_Units
        <Unit_Dimensions => Unit_Dimensions + Right_Type.Unit_Dimensions>);

operator "indexing"(C : ref Container; Index : Index_Type)
```

```
    -> ref Element_Type;
```

6.2 Operation Definitions

An operation may be defined with a body, or with an import clause.

An operation definition has the following form:

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

function_definition ::=
  function_declaration 'is'
    operation_body
  'end' 'function' identifier

procedure_definition ::=
  procedure_declaration 'is'
    operation_body
  'end' 'procedure' identifier

operator_definition ::=
  operator_declaration 'is'
    operation_body
  'end' 'operator' operator_symbol

operation_body ::= [ dequeue_condition ] statement_list

operation_import ::=
  operation_declaration 'is' 'import' '(' operation_actuals ')'
```

If an operation is declared with a separate, stand-alone operation_declaration, then the operation_declaration in the operation_definition must fully conform to it. If any annotations appear prior to the 'is' of the operation_definition, then they must fully conform to the annotations on the separate operation_declaration. Similarly, if any comments appear prior to the 'is' of the operation_definition, then they must fully conform to comments on the separate operation_declaration.

Examples:

```
function Sin(X : Float) -> Float is import("sinf");

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

function Fib (N : Integer) -> Integer is
   // Recursive fibonacci but with linear time

   function Fib_Helper(M : Integer)
     -> (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 function Fib_Helper;

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

6.3 Operation Calls

Operation calls are used to invoke an operation, with inputs and/or outputs.

Operation calls are of the form:

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

operation_name ::=
    [ type_identifier '::' ] 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 modules other than the current module are automatically considered, depending on the form of the operation_name:

- If the operation_name is of the form `type_identifier '::' operation_designator` then only operations with the given designator declared within the module associated with 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 declared in the modules associated with the types of the operation inputs and outputs, if any, are considered, along with locally declared operations with the given designator.

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 operation inputs for which no actual is provided. For 'global' inputs, a global concurrent object with the given identifier must be visible both to the caller and the called operation, and if it is a 'var' input, the caller must also have it as a 'global' 'var' input. After parallel evaluation of the operation_actuals, the body of the operation is executed, and then any outputs 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 declared in the module that is associated with the root type of the polymorphic type, then the actual body invoked depends on the run-time type-id of the actual. If multiple operation actuals have this same polymorphic type, then their run-time type-ids must all be the same.

Examples:

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

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

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

6.3.1 Lambda expressions

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

A lambda expression has the following form:

```
lambda_expression ::=
  'lambda' inputs [ '->' outputs ] 'is' lambda_body

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

Example:

```
Graph_Function(Window, lambda (X : Float) -> Float is sin(X)**2);
```

6.4 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 output.

Examples:

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

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

Chapter 7

Modules

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

Every module has an interface that declares its external characteristics. If the interface of a module declares any non-abstract operations, the module must have a class that defines its internal representation and algorithms.

7.1 Interface Declaration for a Module

The interface of a module is declared using the following syntax:

```
interface_declaration ::=
  ['abstract'] ['concurrent'] 'interface' module_identifier
    < module_formals >
    [ module_ancestry ]
  'is'
     {interface_item}
  'end' 'interface' identifier ';'

module_identifier ::= identifier { '::' identifier }

interface_item ::=
  type_declaration | operation_declaration | object_declaration | interface_declaration
```

Module formal parameters have the following form:

```
module_formals ::= [ module_formal { ';' module_formal } ]

module_formal ::= formal_type | formal_operation | formal_object

formal_type ::= [ identifier 'is' ] interface_name '<' module_actuals '>'
```

Example (also used in section 3.1):

```
 interface List <Element_Type is Assignable<>> is
     function Create() -> List;
     function Is_Empty(L : List) -> Boolean;
     procedure Append(L : ref var List; Elem : Element_Type);
```

```
        function Remove_First(L : ref var List) -> optional Element_Type;
        function Nth_Element(L : ref List; N : Univ_Integer) -> ref optional Element;
    end interface List;
```

This defines the interface to a List module, which provides 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.

TBD: Import clause

7.2 Module Inheritance and Extension

A module may be defined as an *extension* of an existing module, and may be defined to *implement* the interface of one or more other modules.

```
  module_ancestry ::=
    ['extends' [ identifier ':' ] module_name [ '<' module_actuals '>' ] ]
    ['implements' module_list ]

  module_name ::= module_identifier

  module_list ::=
    module_name '<' module_actuals '>'
      { ',' module_name '<' module_actuals '>' }
```

If a module M2 *extends* a module M1, but does not specify the module_actuals for M1, then M2 inherits all of the module formals of M1. Otherwise, module M2 must have its own set of formals, which are then used to instantiate M1. The instance of M1 defined by the specified module_actuals, or by substituting the corresponding formals of M2 into M1, is called the *underlying type* for M2. A module has an *underlying type* only if it is defined to extend some other module. If a module M2 has an underlying type, then there is an *underlying* component of each object of any type produced by instantiating the module M2. This underlying component is of the underlying type, and is by default named by the identifier of the module being extended (e.g. M1), but may be given its own identifier by specifying it immediately after 'extends'.

A module *inherits* operations from the interfaces of the modules that it extends or implements. When an operation is inherited from the interface of a module M1 by a new module M2, the types of the inputs and outputs of any operation are altered by replacing each occurrence of the original module name M1 with the new module name M2, and by substituting in the formal parameter names of M2 for the corresponding formal parameter names of M1.

If the module M2 *extends* the module M1, then an operation inherited from M1 is abstract only if the corresponding operation in M1 is abstract, or if the operation has an output which is of a type based on M1. If the operation inherited from M1 is not abstract, then its implicit body is defined to call the operation of M1, with any input to this operation that is of the underlying type being passed the underlying component of the corresponding input to the inherited operation.

If the module M2 *implements* the module M1, then an operation inherited from M1 is optional if the corresponding operation in M1 is optional, and is otherwise abstract.

If two operations of the same name inherited from different modules end up with identical numbers and types of inputs and outputs, the non-abstract, non-optional one hides the others, and the optional one(s) hide the abstract one(s).

An inherited operation may be overridden by providing a declaration for the operation in the interface of the new module with the same name and number and types of inputs and outputs as the inherited operation. An abstract inherited operation must be overridden unless the new module is itself specified as 'abstract'.

Example:

```
interface Skip_List
  <Skip_Elem_Type is Assignable<>; Initial_Size : Univ_Integer := 8>
  implements List<Element_Type => Skip_Elem_Type> is

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

    // abstract function Create() -> Skip_List;
    // function Is_Empty(L : Skip_List) -> Boolean;
    // procedure Append(L : ref var Skip_List; Elem : Skip_Elem_Type);
    // function Remove_First(L : ref var Skip_List) -> optional Skip_Elem_Type;
    // function Nth_Element(L : ref Skip_List; N : Univ_Integer)
    //     -> ref optional Skip_Elem_Type;

    function Create() -> Skip_List;
      // This overrides the abstract inherited operation

    ... // Here we may override other inherited operations
        // or introduce new operations
end interface 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 or implements the identified type's interface, with matching module actuals. The identified type is called the *root* type for the corresponding polymorphic type.

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

```
type Bool_Skip_List is Skip_List<Boolean>;

var BL : Bool_List+ := Bool_Skip_List::Create();
```

The variable BL can now hold values of any type that is an instance of a module that implements the List interface, 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 4.2.2) or a case statement (see 5.4), and it controls which body is executed in certain operation calls (see 6.3). In the above example, the type-id of BL initially identifies the Bool_Skip_List type.

7.3 Class Definition for a Module

A class defines local types, operations, and data for a module, as well as a body for each operation declared in the module's interface.

A class has the following form:

```
class_definition ::=
  ['concurrent'] 'class' module_identifier
    [ < module_formals > ]
    [ module_ancestry ]
  'is'
     {local_class_item}
```

```
      'exports'
       {exported_class_item}
    'end' 'interface' identifier ';'

  local_class_item ::=
      type_declaration
    | operation_declaration
    | operation_definition
    | object_declaration
    | interface_declaration
    | class_definition

  exported_class_item ::=
      operation_definition
    | object_declaration
    | class_definition
```

An exported_class_item must correspond to an item declared in the module's interface.

Within an object_declaration in a class, a name may refer to any prior class_item using its simple identifier. Within a type_declaration in a class, a name within a constraint_annotation may refer to a local constant, but if the constant is not initialized at its declaration, the type has an *object-specific* constraint and may only be used within subsequent local (*object-specific*) type and object declarations (see the Index_Type of the Stack class in chapter 9 for an example of an *object-specific* constraint). Within other kinds of class_items, local interfaces, non-object-specific types, operations, and initialized constants may be referred to directly, but local variables and uninitialized constants are considered components of objects of a type associated with the enclosing class, and must be referred to using `component_selection` notation (see 4.1.1).

Example:

```
class List is
     interface List_Node<> is
         var Elem : Element_Type;
         var Next : optional List_Node;
     end interface List_Node;
     var Head : optional List_Node<>;
  exports
     function Create() -> List is
        return (Head => null);
     end function Create;

     function Is_Empty(L : List) -> Boolean is
        return L.Head is null;  // Must say "L.Head," not simply "Head"
     end function Is_Empty;

     procedure Append(L : ref var 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 procedure Append;
```

```
      function Remove_First(L : ref var List) -> 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
            Remove_First := L.Head.Elem;
            L.Head := L.Head.Next;
            return;   // Output already assigned
         end if;
      end function Remove_First;

      function Nth_Element(L : ref List; N : Univ_Integer)
        -> 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 function Nth_Element;

   end class List;
```

The above class defines the module List whose interface is given in 7.1. The items preceding `exports` are local to the module, and are used to implement the linked list structure. The items after `exports` correspond to the items declared in the List interface.

TBD: Private interfaces, module extensions, module specializations

7.4 Module Instantiation

Modules are instantiated by providing actuals to correspond to the module 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 module to produce a type have the following form:

```
module_actuals ::= [ module_actual { ',' module_actual } ]

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

actual_type ::= object_type
```

Any module actuals with a specified identifier must follow any actuals without a specified identifier. The identifier given preceding `'=>'` in a module_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:

```
interface Map<Key_Type is Hashable<>; Element_Type is Assignable<>> is
    operator "[]"() -> Map;
    operator "|="(M : ref var Map; Key : Key_Type; Elem : Element_Type);
    operator "indexing"(M : ref Map; Key : Key_Type)
      -> ref optional Element_Type;
    operator "index_set"(M : Map) -> Set<Key_Type>;
end interface Map;
```

The Map interface defines a container with Key_Type as the index type and Element_Type as the value type. The interface 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.

```
interface Set<Element_Type is Hashable<>> is
    operator "[]"() -> Set;        // Empty set
    operator "|="(S : ref var Set; Elem : Element_Type);
    function Count(S : Set) -> Univ_Integer;
    operator "in"(Elem : Element_Type; S : Set) -> Boolean;
    operator "indexing"(S : ref Set;
      Index : Univ_Integer {Index in 1..Count(S)})
      -> ref Elem;
    operator "index_set"(S : Set) -> Interval<Univ_Integer>;
    operator "=?"(Left, Right : Set) -> Ordering;
end interface Set;
```

The Set interface defines a container with the Element_Type as the value type and Univ_Integer as the index type. The interface 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 4.2.1) is provided for comparing sets for equality and subset/superset relationships.

```
interface Array<Component_Type is Assignable<>; Indexed_By is Countable<>> is
    type Bounds_Type is Interval<Indexed_By>;
    function Bounds(A : Array) -> Bounds_Type;

    operator "[]"
      (Index_Set : Bounds_Type; Values : Map<Bounds_Type, Component_Type>)
      -> Result : Array {Bounds(Result) == Index_Set} ;

    operator "indexing"(A : ref Array;
      Index : Indexed_By {Index in Bounds(A)} )
      -> ref Component_Type;
    operator "index_set"(A : Array)
      -> Result : Bounds_Type {Result == Bounds(A)} ;

    operator "slicing"(A : ref Array;
      Slice : Bounds_Type { Slice <= Bounds(A)} )
      -> Result : ref Array {Bounds(Result) == Slice} ;

    operator "|="(A : ref var Array;
      Index : Indexed_By {Index in Bounds(A)} ;
      Value : Component_Type);
    operator "|="(A : ref var Array;
      Slice : Bounds_Type {Slice <= Bounds(A)} ;
      Value : Component_Type);
end interface Array;
```

The Array interface defines a container with Component_Type as the value type and Indexed_By as the index type. The index-set type is Bounds_Type. The interface 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 share the following syntax:

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

If 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:

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

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

```
object_name."indexing" '(' 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 inputs 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 inputs share no values, then the results share no elements of the container object.

Examples:

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

A[1..3] :=: A[4..6];    // swap halves of 6-element array
```

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 ::= '..' => expression

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

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

iterator_container_aggregate ::=
  '[' 'for' iterator '=>' 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 next_values part.

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 assignment operation "|=".

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 container_element in the aggregate, 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.

Examples:

```
[ 1, 2, 3, 4, 5 ]                  // positional container aggregate
[ 1..5 => 1, .. => 0 ]             // named with default
[ #red => 0x1, #green => 0x10, #blue => 0x100 ]
                                   // all named
[ for I in 1..10 => I ** 2 ]       // table of squares
```

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;
```

Chapter 9

Annotations

Annotations may appear at various points within a program. Depending on their location, they can represent a precondition of an operation, a postcondition of an operation, a constraint on a type, an invariant of a class, or a simple assertion at a point in a sequence of statements.

Annotations have the following form:

```
annotation ::= '{' condition { ';' condition } '}'

postcondition_value ::= object_name '''

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

Annotations must evaluate to true under all situations where they are used. The ParaSail program is illegal if an annotation is violated, or if it cannot be proved true by the implementation.

Within an annotation that is used as a postcondition, a postcondition_value (e.g. `S'`) refers to the value of the specified object (e.g. `S`) after the operation is complete. The specified object must be a variable input to the operation.

An expression of the form `'[[' expression ']]'` may be used to convert an expression to a universal type, generally for use in an annotation. 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:

```
function Sqrt(X : Float { X >= 0.0 }) -> Float {Sqrt >= 0.0};
```

The first annotation is a precondition; the second is a postcondition.

```
type Age is new Integer<0..200>;
type Minor is Age { Minor < 18 };
type Senior is Age { Senior >= 50 };
```

These annotations define constraints on two different subtypes of the Age type.

```
interface Modular< Modulus : Univ_Integer {Modulus >= 2} > is
  operator "from_univ"(Univ : Univ_Integer {Univ in 0 ..< Modulus})
    -> Modular;

  operator "to_univ"(Val : Modular) -> Result : Univ_Integer
    { Result in 0 ..< Modulus };

  operator "+"(Left, Right : Modular) -> Result : Modular
    { [[Result]] == ( [[Left]] + [[Right]] ) mod Modulus };
  ...
end interface Modular;
```

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:

```
interface Stack
  <Component is Assignable<>;
   Size_Type is Integer<>> is
    function Max_Stack_Size(S : Stack) -> Size_Type;
    function Count(S : Stack) -> Size_Type;

    function Create(Max : Size_Type {Max > 0}) -> Stack
      {Max_Stack_Size(Create) == Max; Count(Create) == 0};

    procedure Push
      (S : ref var Stack {Count(S) < Max_Stack_Size(S)};
       X : Component) {Count(S') == Count(S) + 1};

    function Top(S : ref Stack {Count(S) > 0}) -> ref Component;

    procedure Pop(S : ref var Stack {Count(S) > 0})
      {Count(S') == Count(S) - 1};
end interface Stack;

class Stack is
    const Max_Len : Size_Type;
    var Cur_Len : Size_Type {Cur_Len in 0..Max_Len};
    type Index_Type is Size_Type {Index_Type in 1..Max_Len};
    var Data : Array<optional Component, Indexed_By => Index_Type>;
  exports
    {for all I in 1..Cur_Len => Data[I] not null}   // invariant for Top()

    function Max_Stack_Size(S : Stack) -> Size_Type is
        return S.Max_Len;
    end function Max_Stack_Size;

    function Count(S : Stack) -> Size_Type is
      return S.Cur_Len;
    end function Count;

    function Create(Max : Size_Type {Max > 0}) -> Stack
      {Max_Stack_Size(Create) == Max; Count(Create) == 0} is
      return (Max_Len => Max, Cur_Len => 0, Data => [.. => null]);
    end function Create;

    procedure Push
      (S : ref var Stack {Count(S) < Max_Stack_Size(S)};
       X : Component) {Count(S') == Count(S) + 1} is
      S.Cur_Len += 1;
      S.Data[S.Cur_Len] := X;
    end procedure Push;

    function Top(S : ref Stack {Count(S) > 0}) -> ref Component is
      return S.Data[S.Cur_Len];
    end function Top;
```

```
        procedure Pop(S : ref var Stack {Count(S) > 0})
          {Count(S') == Count(S) - 1} is
            S.Cur_Len -= 1;
        end procedure Pop;
    end class Stack;
```

This example illustrates annotations used as preconditions (`{Count(S) > 0}`), postconditions (`{Count(S') == Count(S) - 1}`), type constraints (`{Cur_Len in 0..Max_Len}`), and a class invariant (`{for all I in 1..Cur_Len => Data[I] not null}`).

Chapter 10

Concurrent Objects

Expression evaluation in ParaSail proceeds in parallel (see 6.3), as do statements separated by `'||'` (see 5.1), and the iterations of a concurrent loop (see 5.6 and 5.6.1). The ParaSail 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 ParaSail are either *concurrent* or *sequential*, according to whether their type is defined by instantiating a concurrent or non-concurrent module. Concurrent objects allow concurrent operations by multiple threads by using appropriate hardware or software synchronization. Sequential objects allow concurrent operations only on non-overlapping components.

10.1 Concurrent Modules

A module is *concurrent* if its interface is declared with the reserved word 'concurrent'. The class defining a concurrent module must also have the reserved word 'concurrent'. Types produced by instantiating a concurrent module are *concurrent* types.

Example:

```
concurrent interface Atomic<Item_Type is Machine_Integer<>> is
   function Create(Initial_Value : Item_Type) -> Atomic;
   function Test_And_Set(X : ref var Atomic) -> Item_Type;
     // If X == 0 then set to 1; return old value of X
   function Compare_And_Swap(X : ref var Atomic;
      Old_Val, New_Val : Item_Type) -> Item_Type;
        // If X == Old_Val then set to New_Val; return old value of X
end interface Atomic;
...
var X : 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 concurrent module which defines an atomic object which 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 module would presumably use hardware synchronization.

10.1.1 Locked and Queued Operations

The operations of a concurrent module M may include the reserved word 'locked' or 'queued' for inputs of a type based on M. If a concurrent module has any operations that have such inputs, then it is a *locking* module; otherwise it is *lock-free.* Any object of a type based on a locking module includes an implicit *lock* component.

If an operation has an input that is marked 'locked', then upon call, a lock is acquired on that input. If it is specified as a 'var' input, then an exclusive read-write lock is acquired; if it is specified as a 'const' input than 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 an input 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' input, 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 input.

Within an operation of a concurrent module, given an input that is marked 'locked' or 'queued', the components of that input may be manipulated knowing that an appropriate lock is held on that input object. If there is a concurrent input that is *not* marked 'locked' or 'queued', then there is no lock on that input, and only concurrent components of such an input 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.

Example:

```
concurrent interface Queue<Element_Type is Assignable<>> is
    function Create() -> Queue;
    procedure Append(Q : locked var Queue; Elem : Element_Type);
    function First(Q : locked const Queue) -> optional Element_Type;
    function Remove_First(Q : queued var Queue) -> Element_Type;
end interface 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 module 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.

10.2 Concurrent Evaluation

Two expressions that are inputs to an operation call (see 6.3) or a binary operator (see 4.2.1) are evaluated in parallel in ParaSail, 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_threads of a statement_thread_group (see 5.1) are performed in parallel. Finally, the iterations of a concurrent loop (see 5.6) are performed in parallel.

Two object_names that can be part of expressions or statements that are evaluated concurrently must not denote overlapping parts of a single sequential 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:

```
function Bump(A : ref var Int) -> 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

ParaSail Library

ParaSail includes a small number of predefined types and a number of language-provided modules.
The predefined types are:

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

Univ_Enumeration all values of the form # identifier

Boolean an enumeration type with two values #false and #true

Ordering an enumeration type with four values #less, #equal, #greater, and #unordered

The language-provided modules are:

```
abstract interface Any<> is
end interface Any;

interface Assignable<> is
    operator ":="(A : ref var Assignable; B : Assignable);
      // This is a pseudo-operation which is provided automatically
      // to all modules that extend Assignable.
      // By default, a module extends Assignable;
      // can be overridden by explicitly extending "Any<>".
end interface Assignable;

abstract interface Comparable<> is
    operator "=?"(Left, Right : Comparable) -> Ordering;
end interface Comparable;

abstract interface Hashable<> extends Assignable<>
  implements Comparable<> is
    function Hash(X : Hashable) -> Univ_Integer;
end interface Hashable;

abstract interface Countable<> extends Hashable<> is
    operator "+"(Left : Countable; Right : Univ_Integer) -> Countable;
    operator "+"(Left : Univ_Integer; Right : Countable) -> Countable;
    operator "-"(Left : Countable; Right : Univ_Integer) -> Countable;
    operator "-"(Left : Countable; Right : Countable) -> Univ_Integer;
end interface Countable;

interface Interval<Bound_Type is Countable<>> is
    ...
end interface Interval;

interface Integer
  <Range : Interval<Univ_Integer> := Default_Range>
  extends Countable<> is
    ...
end interface Integer;

interface Float<Digits : Univ_Integer := Default_Digits>
  extends Hashable<> is
    ...
end interface Float;

interface Character<Bits : Univ_Integer := 8; ...>
  extends Countable<> is
    ...
end interface Character;

interface String<Character_Type is Character<>>
  extends Hashable<> is
    ...
end interface String;

abstract interface Container
  <Value_Type is Assignable<>; Index_Type is Hashable<>>
  extends Hashable<> is
    ...
end interface Container;

interface Vector
  <Element_Type is Assignable<>>
  extends Container<Element_Type, Univ_Integer> is
    ...
end interface Vector;

interface Enum<Values is Vector<Univ_Enumeration>>
  extends Countable<> is
    ...
end interface Enum;

interface Array
  <Element_Type is Assignable<>; Indexed_By is Countable<>>
  extends Container<Element_Type, Indexed_By> is
    ...
end interface Array;

interface Set
  <Element_Type is Hashable<>>
  extends Container<Element_Type, Univ_Integer> is
    ...
```

```
end interface Set;

interface Map
  <Key_Type is Hashable<>; Element_Type is Assignable<>>
  extends Container<Element_Type, Key_Type> is
    ...
end interface Map;
```