King Basics for Ada Software Engineers

Jeffrey R. Carter
2024 November

Introduction

King is a high-level, strongly typed, general-purpose language for engineering software inspired by my many decades of software engineering and the many languages I have been exposed to. It has modules for encapsulation and information hiding, high-level concurrency, user-defined numeric types, fixed-point types, non-numeric enumeration types, maps, sequences, sets, and run-time checks. It does not have arrays, pointers, go-to statements, or programming by extension.

King is based on Ada 83 with its deficiencies corrected, ideas from later versions of Ada, and some ideas from other languages.

Contrary to popular belief, "King" is not an acronym, and does not stand for Knowledgeable, Intelligent, Nice, and Great. Nor is it intended to mean that King is the king of languages. Instead, it is named for Ada King (née Byron), the Countess of Lovelace, who should be familiar to all Ada software engineers.

This document is intended for people with good Ada and software engineering knowledge. Many features will be explained by reference to similar features in Ada or to software engineering concepts. As with Ada, understanding visibility is key to understanding King, and a good grasp of Ada's visibility concepts is essential.

Hello World

It is customary to begin such a document with the traditional "Hello, World" program. Here it is in King:

```
-- The traditional Hello-World program

use King.IO.Text;

Hello_World : procedure body is
   Message = "Hello, World!"; -- This is a named string
Hello_World : begin
   King.IO.Text.Put_Line (Line => Message); -- Implicitly converted to String
Hello_World : when E =>
   E.Raise;
Hello_World : end procedure;
```

This should be pretty clear. The main difference from Ada is the `Name :` syntax for most constructs and the exception handling. `Hello_World` is a library-level,

parameterless, main-program procedure body, which is the only kind of subprogram unit allowed at library-level. It is also the only kind of subprogram body that does not complete a separate specification.

The first line is a comment. King comments are identical to Ada comments.

The next (non-null) line is a `use` statement, which is equivalent to Ada's `with` statement. It makes the named module visible to the compilation unit. Unlike Ada's `with`, a `use` statement also makes the primitive operators of types declared in the module directly visible. The primitive operators of enumeration types include the enumeration literals. A `use` statement may only name a single module. King has no equivalent to Ada's `use` statement. As in Ada, all statements are terminated by a semicolon.

The next line introduces a `procedure body` named `Hello_World`. Unlike Ada, the name comes first, followed by a colon, as with object declarations. Procedures with parameter lists have them between `procedure` and the terminating semicolon for a procedure specification, and between `procedure` and `body` for a procedure body. Functions are similar, but use the reserved word `function` and have `return` *`subtype`* after the parameter list.

The next line declares a *named string*. Ada has named numbers; King has *named values*, which include Ada's named numbers, but also include named strings and *named characters*. There can only be one name in a named-value declaration. Character literals and named characters have type *`universal_character`* (any Unicode character), and string literals and named strings have type *`universal_string`* (sequences of *`universal_character`*). These are implicitly converted to any character or string type, much as universal numeric types are implicitly converted to any corresponding numeric type. King also implicitly converts values of *`universal_integer`* to *`universal_real`* when needed.

This declaration is nested directly in `Hello_World`, and so is indented one indentation level of 3 spaces (tabs are not allowed in King source text). Indentation in King directly reflects the nesting level of statements. This results in slightly different indentation in some cases than Ada, in which standard formatting sometimes indents things more than their nesting level.

The next line is a `begin`, which terminates the declarative region and separates it from the executable statements. The procedure name must precede `begin` as shown. While the repetitions of the procedure name are overkill for a short example like this, in real software it can significantly improve readability. Having separate declarative regions makes it easier to find declarations when reading code.

The **`when`** line terminates the executable statements and introduces the exception handler. Every subprogram must have an exception handler, to document that all possible exceptions have been considered. The procedure name must begin this line.

The reserved word **`when`** is followed by a name for the `Exception_Occurrence` being handled and an arrow. This name renames the actual exception being handled. `Exception_Occurrence` is a predefined module type described more

fully later; all exceptions are of this type. Procedure `Raise` is defined for the type, and raises the exception. The exception handler here is the form when no exception handling is needed. There is a special form of the case statement for dealing with different exceptions when exception handling is needed. It will be described later.

Finally, the `end` line terminates the procedure just as in Ada, except for the position of the name, which is mandatory, and repeating that this is a procedure.

Except for the spaces around the comment symbol and in comments, all of the spaces in this example are required.

Modules

No language is worth using if it lacks explicit support for modularity. King modules are similar to packages in Ada. Unlike packages, they do not have any equivalent to private parts and cannot contain declarations of variables in the specification (there is no difference between a constant and a variable for non-assignable types). Like most things in King, they come in two parts: the specification and the body. Modules implement the software-engineering concepts of encapsulation and information hiding.

Example

As a trivial example consider a very poor random-number generator:

```
-- A very poor random-number generator

Bad_Random : module is
   type State is hidden;
   -- Initial value is the result of calling Set_Seed with a
   -- Seed of Result'Last

   type Result is range 0 .. 2 ^ 16 - 1 with
      Signed_Representation => False,
      Overflow_Checking => False;

   Set_Seed : procedure (State : out State; Seed : in Result <- Result'Last);
   -- Sets State to produce the sequence of values defined by
   -- Seed

   Random : function (State : in out State) return Result;
   -- Returns the next value in the sequence defined by State
   -- and updates State accordingly
Bad_Random : end module;
```

This is a module specification. Here `State` is a *hidden type*. Its full declaration will be a record type in the module body. No operations are implicitly defined for a hidden type except assignment and the membership tests `in` and `not in`. Given an object `O` of a composite type, one can call a primitive operation `Op` with a first parameter of the type using `Object.Operation` notation: `O.Op`.

`Result` is an unsigned integer type with 2^{16} values and no overflow checking, similar to an equivalent modular type in Ada (King uses ^ for exponentiation). Note the aspect clauses for the declaration, which are similar to Ada's. By default, integer types have `Signed_Representation => True` and `Overflow_Checking => True`. Both of these aspects apply to the base type, `Result'Base`, which in this case should be the same as `Result`.

The combinations of these two aspects yields four kinds of integer types, compared to Ada's two. There is only one form of integer type declaration in King, compared to Ada's two.

Numeric literals can be given as here for base 10, or with Ada's based-literal notation for any base from 2 to 36, including 10. Letters used for digits greater than 9 must be capitals, as must the `E` that introduces an exponent.

The module contains two subprogram specifications.

Procedure `Set_Seed` has a parameter named `State` that has subtype `State`. This demonstrates that subtype names come from a different namespace than other identifiers; when the same identifier is used as a subtype name and another identifier, as here, the namespace that an identifier belongs to is clear from context. The parameter `Seed` has a default value of `Result'Last` as indicated by the assignment symbol `<-` followed by the default value, so one can call the procedure without supplying a value for that parameter.

Function `Random` uses and changes its parameter and returns the next random value.

Parameter modes (`in`, `in out`, and `out`) are required for all subprogram parameters. Each parameter in a parameter list must have its own mode and subtype.

A module specification that declares something that must be completed must be completed by a module body:

```
Bad_Random : module body is
   type State is record  -- Full type must be a record
      Seed : Result is Result'Last;
   end record State;

   Set_Seed : procedure (State : out State; Seed : in Result <- Result'Last) is
      (State.Seed <- Seed);

   Random : function (State : in out State) return Result body is
      null; -- Declarative region must have a declarative item
   Random : begin
      State.Seed <- State.Seed + 1;

      return State.Seed; -- I told you this is a bad RNG
   Random : when E =>
      E.Raise;
   Random : end function;
Bad_Random : begin
   null;
Bad_Random : when E =>
   E.Raise;
```

```
Bad_Random : end module;
```

The component Seed of type State has an explicit default initial value of Result'Last. Initial values are introduced by **is** rather than the assignment symbol as in Ada. As we've seen, named values are introduced by =, and defaults that are not initial values by <-. In the absence of explicit initial values, objects and components of most scalar types are initialized to the 'First of their subtypes unless they have the Default_Value aspect specified.

Components of record types are nested directly within the type, so they must be indented one indentation level. This imposes formatting of record types as shown, unlike the formatting used by the Ada Reference Manual (ARM).

Set_Seed is an example of a statement procedure, which is defined fully later.

As noted in the comment, a declarative region must have at least one declarative item. The null declaration, used here, must appear if there are no others.

Assignment statements are much as in Ada, except for the assignment symbol.

Incrementing Seed should overflow and wrap around to zero when its value is Result'Last.

Module bodies have a sequence of statements for elaboration and an exception handler for those statements, and these must be given for all modules.

Macro Modules

Modules can be parameterized to act on different types or with different operations, similar to generic packages in Ada. Such modules are called *macro modules*. ("Generic" might be a better term, but "macro" alliterates.)

```
-- A simple bounded-list module that can create lists of any definite type
-- with assignment and equality

Bounded_Lists : macro
   type Element is hidden with <-;

   "=" : function (Left : in Element; Right : in Element) return Boolean is <>;
Bounded_Lists : module is
   type List (Max_Length : Position_Value) is hidden with
      Default_Initial_Condition => List.Is_Empty;

   type Position is hidden with
      Default_Initial_Condition => Position = Invalid;

   Invalid : constant Position;

   Length : function (List : in List) return Count_Value;
   -- Returns the number of Elements in List

   Is_Empty : function (List : in List) return Boolean is
      (List.Length = 0);
```

```
Is_Full : function (List : in List) return Boolean is
   (List.Length = List.Max_Length);

Clear : procedure (List : in out List) with
   Postcondition => List.Is_Empty;

First : function (List : in List) return Position;
-- Returns the Position of the first Element in List
-- Returns Invalid if List.Is_Empty

Last : function (List : in List) return Position;
-- Returns the Position of the last Element in List
-- Returns Invalid if List.Is_Empty

Valid : function (List : in List; Position : in Position) return Boolean;
-- Returns True if Position is a valid Position for List; False otherwise
-- Returns False if List.Is_Empty

Next : function (List : in List; Position : in Position)
return Position with
   Precondition => List.Valid (Position);
-- Returns the Position in List after Position
-- Returns Invalid if Position = List.Last

Previous : function (List : in List; Position : in Position)
return Position with
   Precondition => List.Valid (Position);
-- Returns the Position in List before Position
-- Returns Invalid if Position = List.First

Value : function (List : in List; Position : in Position)
return Element with
   Precondition => List.Valid (Position);
-- Returns the Element stored in List at Position

Append : procedure (Onto : in out List; Item : in Element) with
   Precondition  => not Onto.Is_Full,
   Postcondition => not Onto.Is_Empty and Onto.Value (Onto.Last) = Item;

Insert : procedure
   (Into : in out List; Before : in Position; Item : in Element)
with
   Precondition  => not Into.Is_Full and Into.Valid (Before),
   Postcondition =>
      not Into.Is_Empty and Into.Value (Into.Previous (Before) ) = Item;

Delete : procedure (From : in out List; Position : in out Position) with
   Precondition  => From.Valid (Position),
   Postcondition => not From.Is_Full and Position = Invalid;
-- Deletes the Element stored in From at Position and makes Position invalid

type Location_Result (Found : Boolean <- False) is record
   case Found is
   when False =>
      null;
   when True =>
      Position : Position;
   end case;
end record Location_Result;
```

```
   Location : function (Within : in List; Item : in Element)
   return Location_Result with
      Postcondition => (if Location'Result.Found then
                           Within.Value (Location'Result.Position) = Item
                        else
                           True);
   -- If Within contains an Element that is "=" to Item, returns a value with
   -- Found of True and Position of the Position where Item is stored
   -- Otherwise, returns Location_Result'(Found => False)
Bounded_Lists : end module;
```

The body of a macro module is the same as for a module, so we won't look at it here. Also, lists are part of King's standard library, so it's unlikely anyone would ever implement a general-purpose one.

The name of a macro module is introduced on the **`macro`** line, in keeping with the general approach used so far. This also makes the name known if the **`module`** line is not visible because of positioning of the text or because the macro parameters are large.

The formal parameter `Element` will match any definite type with assignment. If **`with`** `<-` had been omitted, it would match any definite type. The macro function parameter "=" means such a function must also exist for the type.

`Position_Value` is a subtype defined in module `King_Predefined_Environment`, which is equivalent to Ada's package Standard. `Count_Value` is similar. The declarations are

```
type Unbounded_Integer is range <> with Default_Value => 0;
subtype Count_Value is Unbounded_Integer with Predicate => Count_Value >= 0;
subtype Position_Value is Unbounded_Integer with
   Predicate => Position_Value > 0,
   Default_Value => 1;
```

`Unbounded_Integer` is an unbounded integer type. This is the kind of integer used by the compiler to calculate static integer values exactly, sometimes referred to as a big integer. Unbounded integers do not have the `'First` and `'Last` attributes defined, so the aspect `Default_Value` must be specified. Unbounded integer types always have signed representations and cannot overflow.

Subtypes of unbounded integer types can have range constraints, but as we want all non-negative and all positive values for the constraints here, we can't use a range constraint and have to use a predicate instead.

There are also unbounded real types. Unbounded numeric types are usually implemented in software. Universal and static expressions are usually evaluated using unbounded numbers.

`Is_Empty` and `Is_Full` are expression functions, which are similar to Ada's. Since the bodies of the functions and their postconditions are identical, it avoids repetition to use them for these. Speaking of postconditions, here we see King's pre- and postconditions, which are similar to Ada's.

`Location_Result` is a variant record type, similar to Ada's. Note the indentation of the `case` part.

Child Units and Hidden Modules

For name-space control, library-level modules and hidden modules may have child units, which may be any of the possible library-level compilation units:

- Modules
- Macro modules
- Hidden modules
- Hidden macro modules
- Main-program procedure bodies

Thus, a module or hidden-module name can be the head of a hierarchy of unit names. Child units do not extend their parents and do not automatically have visibility into them. Any module may use any other, regardless of position in a hierarchy, so long as the dependencies are not circular. Using a child unit does not use its ancestors.

A hidden [macro] module must be a library-level child unit and is identified by having the reserved word **`hidden`** before **`module`** in its specification. Hidden modules allow for further partitioning of a logical piece of a system without making that partitioning externally visible. To this end, a hidden module can only be used by

- The body of its parent
- The body of a descendant module of its parent
- The specification of a hidden descendant module of its parent

[Hidden] macro modules and main-program procedure bodies cannot have children. Hidden modules can only have hidden children.

Macro Parameters

Macro Type Parameters

King's macro type parameters, from most general to most specific, are

```
type T is hidden (<>);
```

The actual parameter may be any type, definite or indefinite, with or without assignment. Within the module it is an indefinite type without assignment.

```
type T is hidden (<>) with <-;
```

The actual parameter may be any type with assignment, including a hidden type before the full type declaration. Within the module it is an indefinite type with assignment.

```
type T is hidden;
```

The actual parameter may be any definite type, with or without assignment. Within the module it is a definite type without assignment.

```
type T is hidden with <-;
```

The actual parameter may be any definite type with assignment. Within the module it is a definite type with assignment.

Macro hidden-type parameters have no operations defined for them by default. Operations may be defined through macro subprograms, such as "=" in the example above.

```
type T is map key_subtype => value_subtype;
```

The actual parameter may be any map type with the given key and value subtypes. Often either or both of the mapped subtypes will be other formal type parameters.

```
type T is sequence_set of element_subtype;
```

where *sequence_set* is one of **sequence** | **set**. The actual parameter may be any corresponding type with a matching element subtype; the element subtype will often be another formal type parameter.

```
type T is delta digits range <>;
```

The actual parameter may be any numeric type. Within the module, only integer literals and *universal_integer* values may be used with the type.

```
type T is delta digits <>;
```

The actual parameter may be any real type.

```
type T is range (<>);
```

The actual parameter may be any discrete type (enumeration or integer type).

```
type T is (<>);
```

The actual parameter may be any enumeration type.

```
type T is delta <>;
```

The actual parameter may be any fixed-point type.

```
type T is digits <>;
```

The actual parameter may be any floating-point type.

```
type T is range <>;
```

The actual parameter may be any integer type.

Macro Constant Parameters

Constant parameters allow passing values to the module. The value may be a literal, expression, variable, or constant. Within the module, the value is constant (may not be assigned to) and has the value of the actual at the point of the macro expansion (the formal is a copy of the actual if the subtype has assignment, or renames the actual if it does not).

```
Name : constant subtype_indication;
```

Macro Subprogram Parameters

Subprogram parameters are often used to provide operations for type parameters.

```
Name : subprogram_definition [is default];
```

Subprogram Defaults

`<>`
`Actual_Name`

These are the same as in Ada: if no actual is supplied, a visible matching subprogram with the same name as the formal (`<>`) or named `Actual_Name` is used.

For a function, an expression-function definition may used; the given expression function is used if no actual is supplied:

```
Less_Than : function (Left : in T; Right : in T) return Boolean is
   (Left < Right);
```

For a procedure, a [declare-]statement-procedure definition my be used; the given [declare-]statement procedure is used if no actual is supplied:

```
Do_Something : procedure (Value : in out T) is (null);
```

Tasks

Few programs these days do not need to deal with concurrency. GUIs are inherently parallel, and the prevalence of multi-core processors means that understanding concurrency is essential for software engineers, and high-level concurrency features are essential for languages.

Tasks are King's main feature for concurrency and come in two flavors: active tasks and passive tasks. Active tasks are like Ada's tasks in that they are self-scheduling, but there are no entries or rendezvous in King. Active tasks

communicate through passive tasks. Passive tasks are similar to Ada's protected objects. Assignment is not defined for task types.

Unlike Ada's shared-memory tasking model, the only variables declared outside a task that the task can access are passive tasks; it can also read named values and constants, and call subprograms of module types if they are task safe (described below). This is more restrictive than the shared-memory model, but makes it easy for King compilers to target any kind of platform, not just shared-memory symmetric multiprocessors.

A consequence of this restriction on what global variables a task may access is that a task cannot call a subprogram that might access such a variable. Subprograms by default may not do so. Subprograms may be marked with the aspect `Task_Safe => False` to indicate that they do access task-prohibited global variables. The compiler must ensure that subprograms adhere to the task-safety restrictions unless they are marked as task unsafe. An entire module or module type may be marked as task unsafe to indicate that all of its subprograms are task unsafe.

A task may call task-unsafe subprograms provided they are declared within the task. As everything is declared within the environment task, it may call such subprograms.

Active Tasks

An active task provides an independent thread of control. Conceptually, each active task is considered to run on its own processor, though in reality there may be more active tasks than actual processors.

The dining-philosophers problem should have an active task for each philosopher:

```
type Philosopher_ID is (Archimedes, Descartes, Hegel, Kant, Socrates);

type Philosopher (ID : Philosopher_ID <- Archimedes) is task;
```

The syntax for an active task lacks task subprograms, which are present for a passive task.

This task-type specification must be completed by a body, but as tasks usually need to access passive tasks, we'll put off looking at one until later.

Ending Task Execution

Normally, active tasks run until they determine that they should stop. Sometimes it is necessary for a task to end the execution of another task. This is done with an **end** statement

```
end Task_Name;
```

If the task is performing a call to a passive-task operation that is not queued or waiting for access to the task structure, the task continues until it returns

from that call before ending. This ensures that the internal state of the passive task remains consistent.

Dynamic Creation of Active Tasks

Sometimes it's necessary to create an arbitrary number of tasks of the same type, or with different values of the task's discriminants. These are done with a task aggregate or a **new** statement.

```
Philosopher'(ID => Socrates)      -- aggregate for a type with discriminants
Name'(others => <>)               -- aggregate for a type without discriminants
new Philosopher (ID => Socrates); -- new statement
```

The task resulting from the aggregate may be used to initialize a component (see below) or passed as an **in** mode parameter. The task resulting from the **new** statement exists but has no name, so it cannot be ended. It must either stop itself, or run forever.

Passive Tasks

Passive tasks have no associated thread of control, but do provide for mutual exclusion. Only one of a passive task's operations may be executed at a time. Unlike Ada's protected objects, which are special-purpose constructs for protecting access to shared data and so limit the actions they may perform, passive tasks are a general-purpose building block for anything that needs mutual exclusion without a thread of control.

In the dining-philosophers problem, the philosophers need to obtain two chopsticks to dine. As there are as many chopsticks as there are philosophers, this can lead to contention. Mutual exclusion is needed to prevent two philosophers from using the same chopstick at the same time:

```
type Chopstick_Control is task
   Has_Sticks : function (ID :  in Philosopher_ID) return Boolean;
   -- Returns True if Picked has been called with ID and returned True;
   -- False otherwise

   Picked : function (ID : in Philosopher_ID) return Boolean with
      Precondition  => not Has_Sticks (ID),
      Postcondition => Picked'Result = Has_Sticks (ID);
   -- Tries to get the chopsticks for philosopher ID
   -- Returns True if both chopsticks could be obtained and False otherwise

   Put : procedure (ID :  in Philosopher_ID) with
      Precondition  => Has_Sticks (ID),
      Postcondition => not Has_Sticks (ID);
   -- Puts down the chopsticks for philosopher ID
end task Chopstick_Control;
```

The body of a passive task has whatever declarations are needed, followed by a subprogram body with barrier for each of its subprograms:

```
use King.Algorithms.Wrapping;
...
```

```
Wrap : module is new King.Algorithms.Wrapping (Element => Philosopher_ID);
...
Chopstick_Control : task body is
   type State_Set is set of Philosopher_ID;

   Sticks : State_Set;
   -- ID in Sticks indicates that the chopstick ID is in use;
   -- otherwise, not in use
   Owners : State_Set;
   -- ID in Owners indicates that the philosopher ID owns (is holding) his
   -- chopsticks; otherwise, he does not

   Has_Sticks : function (ID :  in Philosopher_ID) return Boolean when True is
      (ID in Owners);

   Picked : function (ID : in Philosopher_ID) return Boolean when True body is
      Success : constant Boolean is
         ID not in Sticks and Wrap.Next (ID) not in Sticks;
   Picked : begin
      if Success then -- Both chopsticks are available; make them unavailable
         Sticks := Sticks + State_Set'{ID, Wrap.Next (ID)};
         Owners := Owners + State_Set'{ID};
      end if;
      -- else Sticks and Owners are unchanged

      return Success;
   Picked : when E =>
      E.Raise;
   Picked : end function;

   Put : procedure (ID :  in Philosopher_ID) when True body is
      null;
   Put : begin -- Make the chopsticks available
      Sticks := Sticks - State_Set'{ID, Wrap.Next (ID)};
      Owners := Owners - State_Set'{ID};
   Put : when E =>
      E.Raise;
   Put : end procedure;
Chopstick_Control : end task;
```

`Wrap` is an expansion of the macro module `King.Algorithms.Wrapping`, similar to a generic instantiation in Ada. It provides function `Wrap.Next`, which gives the next value after its argument, wrapping around from the last value to the first.

`State_Set` is a set type, implementing a mathematical set. Sets are initially empty. Set aggregates use braces (`{}`) (all aggregates must be qualified).

After declaring the state variables, we have the bodies for the subprograms. These are the same as regular subprograms, except for the barrier introduced by **`when`**. Procedures act much the same as entry bodies in Ada. Functions only differ in that they also return a value. In this case all the subprograms may always proceed, so the barriers are `True`. This is because barriers cannot reference the subprogram parameters.

Internal calls, calls to other subprograms of a passive task from within one of the task's subprograms, do not queue and raise the exception `Program_Error`

if the called subprogram's barrier is False. Preconditions are evaluated after the caller has obtained access to the task. Postconditions are evaluated before the caller relinquishes access to the task. Any calls to the task's functions in pre- and postconditions are internal calls.

Small functions like `Has_Sticks` that query the task's state, have True barriers, and are only used in pre- and postconditions are common.

Priorities

All tasks, active and passive, have an associated *priority*. When there are fewer actual processors than active tasks, the priority helps determine which task gets to run when an actual processor becomes available. An active task executing an operation of a passive task takes on the passive task's priority (which must be greater than or equal to the active task's priority) for the duration of the operation.

There exist the following declarations in module `King_Predefined_Environment`:

```
type Any_Priority is range 1 .. 64;

Max_Interrupt_Priority = 16;

subtype Interrupt_Priority is Any_Priority range 1 .. Max_Interrupt_Priority;
subtype Priority is Any_Priority range
   Max_Interrupt_Priority + 1 .. Any_Priority'Last;

Default_Priority = (Priority'First + Priority'Last) / 2;
```

Following normal usage, the smaller the priority's numeric value, the higher the priority. Priority 1 is the highest priority and 64, the lowest.

A task's priority may be specified by the `Priority` aspect. For an active task, the aspect takes a value of subtype `Priority`; for a passive task, `Any_Priority`. The default priority of an active task is `Default_Priority`; of a passive task, `Priority'First`.

Protected Queues

Queues with mutually exclusive access are a common mechanism for communication between tasks, so King provides them in its standard library. They can be either bounded or unbounded. King also includes normal, non-mutually exclusive queues of both kinds. Assignment is not defined for protected queues, but is for normal queues.

Continuing the dining-philosophers problem, the philosopher tasks need to report their state changes somehow, and should spend as little time as possible doing so. A queue of messages from the tasks, processed by a reporting task, does just that, so we would declare:

```
Message_Queues : module is new King.Data_Structures.Queues.Unbounded.Protected
   (Element => String);
```

```
subtype Message_Queue is Message_Queues.Handle;
```

with the predefined type `String` defined as

```
type String is sequence of Unicode;
```

`Unicode` is the Unicode character set. Queue component types have to be definite types with assignment.

Active Task Bodies

We can now look at active task bodies. These are similar to parameterless procedure bodies, except for the reserved word **`task`** instead of **`procedure`**. Returning to the `Philosopher` task type, and presuming a passive task type `Podium_Control`, its body might be

```
Chopsticks    : Chopstick_Control;
Message_Queue : Message_Queue;
Podium        : Podium_Control;

Philosopher : task body is
   Thinking = 10.0; -- Times in seconds
   Talking  = Thinking;
   Eating   = Thinking;
Philosopher : begin
   All_Rounds : for Round in Integer range 1 .. 10 loop
      Message_Queue.Put
         (Item => ID'Image & " thinking for " &
                  Thinking'Image (Before => 0, After => 1, Exponent => 0) );

      wait of Thinking;

      Message_Queue.Put (Item => ID'Image & " waiting for podium");
      Podium.Talk (ID => ID, Talking => Talking);
      Message_Queue.Put (Item => ID'Image & " waiting for chopsticks");

      Get_Sticks : loop
         exit Get_Sticks when Chopsticks.Picked (ID);

         wait of 1.0;
      Get_Sticks : end loop;

      Message_Queue.Put
         (Item => ID'Image & " eating for " &
                  Eating'Image (Before => 0, After => 1, Exponent => 0) );

      wait of Eating;

      Chopsticks.Put (ID => ID);
   All_Rounds : end loop;
Philosopher : when Error =>
   Message_Queue.Put (Item => ID'Image & " ended by " & Error.Information);
Philosopher : end task;
```

Not much here should need explanation. The passive task `Podium` controls access to the floor, putting a message on the queue when a philosopher is

talking and blocking the caller for `Talking` seconds. We also assume an active task that takes messages off the queue and outputs them.

String literals, which have type *`universal_string`*, are a form of aggregate for string types such as `String`. A string literal is `'"'` followed by zero or more graphic characters except `'"'` followed by `'"'`. Values of *`universal_string`* containing `'"'` or non-graphic characters may be created by concatenating string literals and the appropriate character literals or non-character literals of `Unicode`.

```
NUL_Terminated = "abcdef" & NUL;
```

Because this is a *`universal_string`*, `NUL` is the corresponding character from `Unicode`.

The attribute `'Image` for enumeration literals returns a `String` of the defining occurrence of the literal. For real values, including *`universal_real`*, the function also has the arguments `Before`, `After`, `Exponent`, and `Base`, with appropriate defaults for the type. `Before => 0` uses as many digits as necessary before the point to represent the value. `After => 0` has no digits after the point and no point, and `Exponent => 0` has no exponent part. The base defaults to 10. For integer values, there are `Width` and `Base` arguments.

King has two kinds of **`wait`** statements that block a task for a period of time. The **`wait of`** statement, used here, is for a relative amount of time, and blocks the task for the number of seconds given. It can be read as, "Perform a wait of this many seconds." The **wait for** statement, not yet seen, takes a `Duration` value and blocks the task until that time arrives, unless it is in the past. It can be read as, "Wait for this time to arrive." The duration is the number of seconds since the epoch, which is not defined, but meaningful time values can be constructed using the standard library.

All loops must be named, and the name must appear on **`exit`** statements.

All that remains now is to declare the philosophers and give them all their IDs:

```
type Philosophers is map Philosopher_ID => Philosopher;

Philosopher : constant Philosophers is
   Philosophers'(for ID in Philosopher_ID => Philosopher'(ID => ID) );
```

This declares a map from `Philosopher_ID` to `Philosopher` tasks and initializes it with a map aggregate. Since the map values are tasks, the component expression is a task aggregate. The resulting map object has the task with discriminant `ID` keyed by `ID`.

The reader may want to implement `Podium` and the reporter task, and put everything together in a main-program procedure as an exercise.

Identifiers

The intention is that King identifiers should be

```
English_Words_With_Initial_Capitals_Separated_By_Underlines
```

King uses English reserved words with the intention that King source text should be similar to English text, so non-English words should not be part of identifiers.

The characters allowed in identifiers are

- Capital letters `'A' .. 'Z'`
- Small letters `'a' .. 'z'`
- Digits `'0' .. '9'`
- The underline `'_'`

An identifier is made up of words separated by underlines. There is an initial word and zero or more subsequent words. The initial word must begin with a capital letter. Subsequent words can begin with a capital letter or a digit. Within a word, a capital letter cannot come after a small letter, and if two capital letters appear, the word cannot contain small letters. If an identifier has subsequent words, the initial word cannot be `A`, `An`, or `The`. Names may not repeat any information about the thing named that is provided by the language. Type names may not contain `Type`; records, `Record`; macro modules, `Macro`; and so on; and this applies to abbreviations such as `T[y[p]]`, `Rec`, and `M`.

King is case insensitive: `this`, `This`, and `THIS` are the same identifier. King is case aware, however: there is a defining occurrence of an identifier, and all other uses of that identifier must have the same casing as the defining occurrence. For things that have separate specifications and bodies, the defining occurrence is the first appearance in the specification. For things that are declared, the defining occurrence is the declaration. For names of loop and declare statements, the defining occurrence is the first line of the statement. For exception objects in exception handlers, the defining occurrence appears after **`when`**.

Reserved words must be all small letters, even when used as an attribute.

Types

Numeric Types

Integer Types

Integer types come in two kinds, bounded and unbounded. We have seen examples of both.

Unbounded Integer Types

The range of values that can be stored in unbounded integers is limited only by the available memory. They are usually implemented in software and may not be suitable for some algorithms.

Bounded Integer Types

Bounded integers have specified lower and upper bounds. The compiler must accept all bounded integer type declarations, regardless of the range of values covered. By default, bounded integers have a signed representation and overflow checking, but either or both can be changed as we have seen. To have an unsigned representation, the lower bound of the type must be non-negative.

Typically the target processor will have a number of hardware integer types, of 1, 2, 4, 8, ... bytes. If the range of a bounded integer type will fit in a single hardware integer type, then the smallest hardware integer type that will hold the range is the base type for the type. With a signed representation, the base type will be roughly symmetrical around zero, regardless of the given bounds. If overflow checking is not done for such a type, it applies to the base type; if an overflow occurs, it may result in a value that is not in the given range.

If the range is too big for the largest hardware integer, but will fit in two of them, then the base type will be two such hardware integers chained together. For larger ranges, more and more hardware integers may be chained together.

The compiler may set a limit N > 1 of the number of hardware integers it can chain together. If an integer type requires more than N hardware integers, the compiler may use an unbounded integer as the base type. In that case, the desired representation is ignored. If overflow checking is not done for the type, then subtracting one from the lower bound will give the upper bound, and adding one to the upper bound will give the lower bound.

There exist the predefined types

```
type Integer is range -(2 ^ 15) + 1 .. 2 ^ 15 - 1;
subtype Natural  is Integer range 0 .. Integer'Last;
subtype Positive is Integer range 1 .. Integer'Last;

type Byte_Value is range 0 .. 255 with Signed_Representation => False;
```

Integer Operations

King has the same unary and binary integer operators as Ada (King uses "^" for exponentiation, as seen above).

The module `King.Platform_Information` has the named integers

```
Max_Signed_Hardware_Integer   = implementation_defined_positive_integer;
Max_Unsigned_Hardware_Integer = implementation_defined_positive_integer;
```

A signed integer type with

```
range -Max_Signed_Hardware_Integer .. Max_Signed_Hardware_Integer
```

will fit in a single hardware integer, as will an unsigned integer type with

```
range 0 .. Max_Unsigned_Hardware_Integer
```

Integer types that fit in a single hardware integer have bit-wise logical operators defined for them, and attribute functions for rotation and shifting. The attribute `'Bit_Wise_Operators` is True for integer types with bit-wise logical operators and False for all other integer types.

Real Types

Real types come in two kinds, floating-point and fixed-point, both of which come in bounded and unbounded forms.

Floating-Point Types

Floating-point types are defined using the reserved word `digits`. Bounded versions have an explicit number of digits; the compiler must accept all bounded floating-point declarations. If a declaration does not fit in a hardware floating-point type, then the declaration's base type may be an unbounded floating-point type.

Unbounded versions use the box for the number of digits. Either kind may include an optional range constraint for the first-named subtype. Bounded types have a default range that increases with the number of digits; for types that fit in a hardware type, according to the requirements of the hardware type; for types that don't fit in a hardware type, as specified for floating-point model numbers in ARM G.

There exist the predefined types

```
type Float is digits 7;
subtype Natural_Float  is Float range 0              .. Float'Last;
subtype Positive_Float is Float range Float'Next (0) .. Float'Last;

type Rational is digits <> with
   Default_Value => 0;
subtype Natural_Rational is Rational with
   Predicate => Natural_Rational >= 0;
subtype Positive_Rational is Rational with
   Predicate => Positive_Rational > 0,
   Default_Value => 1;
```

`Rational` is the same as the type used by the compiler to evaluate static real expressions.

Fixed-Point Types

Fixed-point types use the reserved word `delta`. Bounded versions have an explicit range, and unbounded versions use the box for the range.

Conceptually, a fixed-point type is stored as an integer, with the value represented being the integer multiplied by the delta for the type. Unlike Ada, there is no *small* for a type that may differ from the delta. This eliminates surprises and the need for a separate form for currency values. The base type of a fixed-point type is a signed type roughly symmetrical around zero. Rounding for fixed-point types rounds to the nearest model number; if a value is exactly halfway between two model numbers, the value is rounded away from zero. The aspect `Truncates => True` may be used to have truncation rather than rounding, often needed for currency values.

The range of values represented by a fixed-point type with an explicit range must include both ends of the range, rounded to the nearest representable value if they are not multiples of the delta.

The compiler must accept all bounded fixed-point declarations. If there is no bounded representation for a bounded fixed-point type declaration, the base type may use an unbounded representation.

There exists the predefined type

```
type Duration is delta 10.0 ^ -9 range <> with
   Default_Value => 0;
subtype Natural_Duration is Duration with
   Predicate => Natural_Duration >= 0;
subtype Positive_Duration is Duration with
   Predicate => Positive_Duration > 0,
   Default_Value => Duration'delta;
```

Duration represents seconds with an accuracy of a nanosecond. It is used by **wait** statements and by time handling.

Unlike Ada, King allows a binary operator to be followed by a unary operator and its operand without requiring parentheses, as shown in the **delta** expression above.

Enumeration Types

King's enumeration types are declared and used much the same as Ada's, except for the use visibility of enumeration literals. Position numbers returned by `'Position` start with one, while the attributes `'Representation` and `'From_Representation` return and take the internal representation of the values, which by default start with zero.

There exist the predefined types

```
type Boolean is (False, True);

type Unicode is (NUL, SOH, STX, ETX, EOT, ENQ, ACK, BEL,
                 BS,  HT,  LF,  VT,  FF,  CR,  SO,  SI,
                 DLE, DC1, DC2, DC3, DC4, NAK, SYN, ETB,
                 CAN, EM,  SUB, ESC, FS,  GS,  RS,  US,
                 ' ', '!', '"', '#', '$', '%', '&', ''',
                 '(', ')', '*', '+', ',', '-', '.', '/',
                 '0', '1', '2', '3', '4', '5', '6', '7',
                 '8', '9', ':', ';', '<', '=', '>', '?',
```

```
                  '@', 'A', 'B', 'C', 'D', 'E', 'F', 'G',
                  'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
                  'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',
                  'X', 'Y', 'Z', '[', '\', ']', '^', '_',
                  '`', 'a', 'b', 'c', 'd', 'e', 'f', 'g',
                  'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
                  'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
                  'x', 'y', 'z', '{', '|', '}', '~', DEL,
                  R01, R02, BPH, NBH, R03, NEL, SSA, ESA,
                  HTS, HTJ, VTS, PLD, PLU, RI,  SS2, SS3,
                  DCS, PU1, PU2, STS, CCH, MW,  SPA, EPA,
                  SOS, R04, SCI, CSI, ST,  OSC, PM,  APC,
                  ' ', '¡', '¢', '£', '¤', '¥', '¦', '§', -- NBSP  ' ' (A0)
                  '¨', '©', 'ª', '«', '¬', SFH, '®', '¯', -- soft- SFH (AD)
                  '°', '±', '²', '³', '´', 'µ', '¶', '·',
                  '¸', '¹', 'º', '»', '¼', '½', '¾', '¿',
                  'À', 'Á', 'Â', 'Ã', 'Ä', 'Å', 'Æ', 'Ç',
                  'È', 'É', 'Ê', 'Ë', 'Ì', 'Í', 'Î', 'Ï',
                  'Ð', 'Ñ', 'Ò', 'Ó', 'Ô', 'Õ', 'Ö', '×',
                  'Ø', 'Ù', 'Ú', 'Û', 'Ü', 'Ý', 'Þ', 'ß',
                  'à', 'á', 'â', 'ã', 'ä', 'å', 'æ', 'ç',
                  'è', 'é', 'ê', 'ë', 'ì', 'í', 'î', 'ï',
                  'ð', 'ñ', 'ò', 'ó', 'ô', 'õ', 'ö', '÷',
                  'ø', 'ù', 'ú', 'û', 'ü', 'ý', 'þ', 'ÿ',
                  ...) with
   Bit_Size => 21;
```

Unicode has literals for all 1,112,064 Unicode code points. Character literals are used for graphic code points like 'A'. Meaningful identifiers are used for non-graphic code points with a meaningful name like NUL. Remaining positions have identifiers of the form U_hhhhhh, where hhhhhh is the hexadecimal representation of the code point (with capital letters for digits greater than 9).

Unicode and *universal_character* are effectively the same type, as any possible character literal is a value of Unicode.

The Bit_Size aspect is of little interest to most users, but specifies that values of Unicode will be represented using a minimum of 21 bits. There is also the aspect Byte_Size which specifies the minimum size in bytes; only one of these aspects may be specified. There are also attributes with the same names for querying these values.

If Bit_Size is specified for a subtype, then Byte_Size is defined as

```
Byte_Size = (Bit_Size + 7) / 8
```

If Byte_Size is specified, then Bit_Size is defined as

```
Bit_Size = 8 * Byte_Size
```

There are three predefined subtypes of Unicode:

```
subtype Basic_Multilingual_Plane is Unicode range
   NUL .. Unicode'From_Representation (16#FFFF#)
with
```

```
   Byte_Size => 2;
subtype Latin_1 is Unicode range NUL .. 'ÿ' with
   Byte_Size => 1;
subtype ASCII is Unicode range NUL .. DEL with
   Bit_Size => 7;
```

Unlike Ada's type `Character`, King's `Unicode` does not have any "magic" positions that can only be referenced by position number or representation. This eliminates the need for something like Ada's package `Latin_1`.

Composite Types

Map Types

Maps map from keys to associated values. In Ada, when all of the keys for a map have discrete subtypes, a constrained array type is often used to implement a map. King uses a notation similar to Ada's one-dimensional constrained array types for maps, but the key subtype may be any assignable type.

```
type Name is map key_subtype => value_subtype;
```

Map types have assignment if the value subtype has assignment; similarly for individual mappings. If the value subtype lacks assignment, the map must be initialized when declared and may only used to access those initial values, as in the map of `Philosopher` tasks above.

For an assignable map `M` with a key `K`, you can set the mapped value with an assignment statement:

```
M (K) <- Value;
```

You retrieve the value for a key with the same notation:

```
Value <- M (K);
```

If no value has been specified for a key, retrieving its value raises `Constraint_Violation`. Maps have the predefined function

```
M.Defined (K)
```

which returns `True` if `M (K)` returns a value and `False` if it raises `Constraint_Violation`.

A map aggregate is similar to an Ada qualified array aggregate using named notation.

Sequence Types

Sequence types define sequences, with items in a sequence defined by their position number.

```
type Name is sequence of Element;
```

The component subtype must have assignment. Individual positions in a sequence may be accessed by giving the position number in brackets:

```
S [I] := Value;
Value := S [I];
```

Position numbers have subtype `Position_Value` and must be in `1 .. S.Length`.

Slicing is also defined for sequences:

```
S [Low .. High] := S [Low - 1 .. High - 1];
```

(A named subtype of `Position_Value` may be used in place of an explicit range.) If the range is non-null, both values must be in `1 .. S.Length`.

Sequence aggregates are similar to Ada's qualified array aggregates, usually using positional notation, except sequence aggregates use brackets rather than parentheses.

```
type Demo is sequence of Boolean;

Demo : Demo is Demo'[True, True, True, False];
```

Named notation may be used, provided that each use of named notation identifies multiple positions with the same value or expression using the name, or follows named notation that does so. So the above could also be represented as

```
Demo'[1 .. 3 => True, 4 => False]
```

Iterative named notation may also be used, so this could also be written

```
Demo'[for I in 1 .. 4 => I < 4]
```

Sequences can be combined using "&" much like Ada's one-dimensional arrays.

Sequences can be extended

```
Demo <- Demo & False;
```

and items can be deleted

```
Demo <- Demo [1 .. 2] & Demo [4 .. Demo.Length];
```

For any sequence type ST with component type CT there are also procedures for these specific cases

```
Append : procedure (List : in out ST; Item : in CT) with
   Postcondition => List.Length = List'Old.Length + 1 and
                    List [List.Length] = Item;

Delete : procedure (List : in out ST; Position : in Position_Value) with
   Precondition  => Position in 1 .. List.Length,
   Postcondition => List.Length = List'Old.Length - 1;
```

Set Types

Mathematical sets are implemented in King with set types:

```
type Name is set of universe_subtype;
```

The universe subtype must have assignment. The following binary operators are defined for sets:

- "+" Union
- "*" Intersection
- "-" Difference
- "/" Symmetric difference
- "<=" Subset
- "<" Proper subset
- ">=" Superset
- ">" Proper superset
- "=" Equaltiy

A set may appear as the right operand of **[not] in**.

Set aggregates are the same as Ada's qualified array aggregates using positional notation, but with braces rather than parentheses:

```
if Color in Colors'{Green, Yellow, Red} then
```

Record Types

Except for the syntax differences, record types are declared and used much as in Ada.

Record aggregates are the same as Ada's qualified record aggregates using named notation.

Record types have assignment if all of their component subtypes have assignment, or if the `Assign` aspect is defined for the type. Record types that complete hidden types must always have assignment.

Derived Types

King has derived types very similar to those of Ada 83. However, King restricts the parent type to those declared in the current declarative part, including enclosing declarative parts up to and including `King_Predefined_Environment`.

Module Types

A module can implement an Abstract Data Type (ADT), with state held in an object and passed to the module's operations, as for `Bad_Random` above, or it can implement an Abstract State Machine (ASM), with state in the module body. ASM modules are inherently task unsafe. An ASM version of `Bad_Random` would look like

```
-- A  very poor random-number generator

Bad_Random : module with Task_Safe => False is
   type Result is range 0 .. 2 ** 16 - 1 with
      Signed_Representation => False,
      Overflow_Checking => False;

   Set_Seed : procedure (Seed : in Result <- Result'Last);
   -- Sets the state of the generator to produce the sequence of values
   -- defined by Seed

   Random : function return Result;
   -- Returns the next value
Bad_Random : end module;
```

Its body would be

```
Bad_Random : module body is
   State : Result is Result'Last;

   Set_Seed : procedure (Seed : in Result <- 0) is (State <- Seed);

   Random : function return Result is
      null;
   Random : begin
      State <- State + 1;

      return State;
   Random : when E =>
      E.Raise;
   Random : end function;
Bad_Random : begin
   null;
Bad_Random : when E =>
   E.Raise;
Bad_Random : end module;
```

`Set_Seed` is a statement procedure, as with the ADT version given earlier.

Module types are a way of defining ASM types. A module-type version of `Bad_Random` would be

```
type Result is range 0 .. 2 ** 16 - 1 with
   Signed_Representation => False,
   Overflow_Checking => False;

-- A  very poor random-number generator

type Bad_Random with Task_Safe => False is module
   Set_Seed : procedure (Seed : in Result <- Result'Last);
   -- Sets the state of the generator to produce the sequence of values
   -- defined by Seed

   Random : function return Result;
   -- Returns the next value
end module Bad_Random;
```

The specification of a module type can only contain subprogram specifications, so the declaration of `Result` must be pulled out.

The body of this type is identical to the body of the ASM module version.

An object of a module type is effectively a module, and its operations are called the same way. Unlike modules, module types can be the parameters of subprograms. Assignment is not defined for module types, so they serve as the equivalent of Ada's limited types.

There exists the predefined type

```
type Exception_Occurrence is module
   Name : function return String;
   -- Returns the full name of the occurrence

   Message : function return String;
   -- Returns the message associated with the occurrence

   Information : function return String;
   -- Returns Name & ' ' & Message & implementation_defined_String

   Set : procedure (Message : in String) with
      Postcondition => Message = Exception_Occurrence.Message;

   Raise : procedure (Message : in String <- "");
   -- Raises the occurrence.
   -- If Message is not null, uses it instead of the currently associated
   -- message
end module Exception_Occurrence;
```

This is the type of exception objects. Note that `Exception_Occurrence` operations are task safe, so tasks may call these operations for external constants of the type. Most exceptions are declared constant for this reason.

Boolean Types

A boolean type is the predefined type `Boolean` or any type derived from it. A value of any boolean type may be used where a *`boolean`*`_expression` is required (**`if`**, **`else_if`**, or after **`when`** in barriers and **`exit`** and **`return`** statements).

Character Types

A character type is any enumeration type with a character literal as an enumeration literal, such as type `Unicode`. A *universal_character* value is implicitly converted to any character type if it is valid for that type.

String Types

A string type is any sequence type with a character type as the component type, such as type `String`. A *universal_string* value is implicitly converted to any string type if it is valid for that type.

There are several restricted subtypes of `String`:

```
subtype BMP_String is String with
   Predicate => (for all C of BMP_String => C in Basic_Multilingual_Plane);
subtype Latin_1_String is String with
   Predicate => (for all C of Latin_1_String => C in Latin_1);
subtype ASCII_String is String with
   Predicate => (for all C of ASCII_String => C in ASCII);
```

Discriminants

We have seen discriminants for task types. Discriminants may also be specified for record types, hidden types, and module types.

Record discriminants may be used to define variant records or define the discriminant of a discriminated component. Such a component may have a size that depends on its discriminant (such as bounded data structures). If the record discriminant has a default value, then unconstrained, varying-size objects of the type may be declared.

Hidden types may be declared to be indefinite by including unknown discriminants [(<>)], the same notation used for macro type parameters that accept indefinite actual parameters. Objects of such types must be constrained by their initialization, usually provided through a function that returns the type.

Exceptions

Exceptions are somewhat different from Ada. Rather than exceptions and corresponding exception occurrences in handlers, all exceptions are exception occurrences. Rather than multiple handlers, King has a single handler and a special form of the case statement for handling multiple exceptions. The predefined exceptions are

```
Constraint_Violation : constant Exception_Occurrence;
Memory_Exhausted     : constant Exception_Occurrence;
Program_Error        : constant Exception_Occurrence;
```

Expressions

Expressions are much as they are in Ada. There is also the declare expression, which is

```
(declare DE_declaration {DE_Declaration} begin expression)
```

where *DE*_Declaration is a constant or subtype declaration.

Static Expressions and Subexpressions

Static expressions only involve values known by the compiler during compilation. Static expressions are evaluated exactly by the compiler. Static numeric expressions are evaluated using universal operations, which always have their usual mathematical meaning. Static string expressions use the universal "&" operator, which always performs simple concatenation.

A static subexpression is a part of a non-static expression that only uses static values and may be evaluated during compilation without changing the entire expression's value. Static subexpressions are evaluated the same way as static expressions, by the compiler, exactly, and using universal operations.

```
Pi = 3.14159265358979323846;
X : Angle;
...
X <- X + Pi / 2;
```

The compiler must calculate `Pi / 2` exactly (1.5707963267948966192) and use that value at run time, even if "/" has been redefined for subtype `Angle`. The redefined operator will be used if one of the operands is qualified as being of the subtype: `Angle'(Pi) / 2`. We don't expect the compiler to evaluate the redefined "/" during compilation, so the subexpression will be evaluated at run time.

Subprograms

King has function and procedure subprograms similar to those in Ada 83. There are also special forms known as expression functions and statement procedures, which do not allow for separate specifications and bodies. Normal subprograms, except for main-program procedures, which may not have a specification, have two separate parts, the specification and the body, as we have seen.

We have encountered expression functions and statement procedures already. Statement procedures come in two flavors, simple statement procedures and declare-statement procedures. A simple statement procedure has the form

```
Name : procedure [(parameters)] is (simple_statement);
```

where *Simple*_Statement is an assignment, null, or procedure-call statement. The simple statement is terminated by the right parenthesis, not by a semicolon. Since we can write

```
Nothing : procedure is (null);
```

King has no need of a special case for null procedures.

Declare-statement procedures are similar but have a declarative part much like declare expressions:

```
Name : procedure [(parameters)] is
   (declare DE_declaration {DE_Declaration} begin simple_statement);
```

Expression functions and statement procedures are useful for simple subprograms where the postcondition and implementation are very similar. A main-program procedure cannot be a statement procedure.

Compound Statements

Conditional Statements

Case Statements

Ordinary case statements are, except for formatting, the same as in Ada.

There is a special form of the case statement for exception handling:

```
Name : when Occurrence_Name =>
   case Occurrence_Name is
   when exception_occurrence {| exception_occurrence}=>
      <statements>
  {when exception_occurrence {| exception_occurrence}=>
      <statements>}
   when others =>
      <statements>
   end case;
```

This form can only appear in an exception handler and only the exception handler's occurrence name may appear after **case**. The exception occurrences appearing in the case branches are declared exceptions, such as Constraint_Violation or King.IO.Invalid_Name. There must be an **others** case, as it is possible to handle exceptions that are not visible.

If Statements

The **if** statement has two forms in King. The first has an **if** part and an optional **else** part.

```
   if boolean_expression then
      <statements>
  [else
```

```
      <statements>]
   end if;
```

The other has **else_if** parts and a mandatory **else** part.

```
   if boolean_expression then
      <statements>
   else_if boolean_expression then
      <statements>
  {else_if boolean_expression then
      <statements>}
   else
      <statements>
   end if;
```

Loop Statements

There are two kinds of loops in King, **for** loops and general loops. All loops must be named, and the loop name must appear on **exit** statements. The syntax is

```
Name : [for prefix] loop
   <statements>
Name : end loop;
```

"**for** prefix" can be one of

```
for Loop_Variable in [reverse] discrete_subtype_name
```

or

```
for Loop_Variable in [reverse] discrete_subtype_definition
```

where *discrete_subtype*_definition is

```
[discrete_subtype_name] range Low .. High
```

(the subtype name is needed if Low and High are both universal),

or

```
for Loop_Variable in [reverse] map_or_sequence'range
```

where map_or_sequence is a map or sequence object name. For a map, Loop_Variable takes on all the values of the key for which values are defined, in an implementation-defined order; **reverse** is not allowed for maps. For a sequence, Loop_Variable takes on all of the position numbers for the object. Finally,

```
for Element of map_sequence_or_set
```

where `map_sequence_or_set` is a map, sequence, or set object name. `Element` refers to each value stored in the object (for a set, each value that is a member of the set). For a set, `Element` is a constant; for the others, a variable.

The **`exit`** statement is

```
exit Loop_Name [when boolean_expression];
```

Declare Statements

The declare statement, similar to Ada's block statement, provides a local declarative region, sequence of statements, and exception handler.

```
Name : declare
   <declarations>
Name : begin
   <statements>
Name : when Occurrence_Name =>
   <exception handling>
Name : end declare;
```

Select Statements

Select statements allow *blocking calls* to be abandoned if they don't start soon enough.

```
   Name : select
      <blocking call>

      [<statements>]
  {Name : or
      <blocking call>

      [<statements>]}
   Name : or
      <wait statement>

      [<statements>]
   Name : end select;
```

A blocking call is a call to a subprogram of a passive task. Where the blocking call is a function call, it must be part of an expression that is part of a valid statement. There can only be one blocking call controlling a branch of a select statement.

If the **`wait`** statement expires before any of the blocking calls start execution, the blocking calls are abandoned and any statements after the **`wait`** are executed.

If one of the blocking calls starts first, the other blocking calls and the **`wait`** are abandoned; after the blocking call finishes, any statements after the blocking call are executed.

If the **wait** is a relative wait with a value of zero or less, or an absolute wait with a time that is not in the future, and none of the blocking calls can start immediately, then the blocking calls are all abandoned and any statements after the **wait** are executed. This is similar to Ada's conditional entry call.

Parallel Statements

Parallel statements provide the opportunity for light-weight threading for parallel execution of code. Whether the opportunity is taken, and how many threads are used if it is, is up to the compiler and run-time system.

Task Declare Statements

Task declare statements provide for parallel execution of arbitrary blocks of code.

```
   Name : task declare
      <declarations>
   Name : begin
      <statements>
   Name : and
      <statements>
  {Name : and
      <statements>}
   Name : end task declare;
```

If there are fewer threads than statement blocks, the statement blocks are executed by threads in textual order. None of the statement blocks can contain a transfer-of-control statement out of the block. If an exception is raised within a statement-block thread, all of the threads end their execution and control transfers to the appropriate exception handler. Execution continues after the task declare statement when all of the statement blocks have completed.

Task Loop Statements

A task loop is a **for** loop with the reserved word **task** before **loop**. A task loop cannot contain the reserved word **reverse**. A task loop indicates that all of the iterations of the loop may proceed in parallel. If there are fewer threads than iterations, each thread will execute the loop body for a single subset of the iterations. A task loop cannot contain a transfer-of-control statement out of the loop. If an exception is raised within a task-loop thread, all of the threads end their execution and control transfers to the appropriate exception handler. Execution continues after the task loop statement when all of the threads have completed.

Restrictions

Anything that can be done by a parallel statement can be done by ordinary sequential code. They may, therefore, access any variables that sequential code may access, with some restrictions. In the following, *object* refers to a simple variable, a component of a map, sequence, or set variable, or to an

object of a module type. Different branches of a task declare statement and different iterations of a task loop statement may not:

- Modify the same object
- Read an object that is modified by a different branch or iteration

The compiler must detect and reject violations of these. Operations of an object of a module type are assumed to modify the object. Usually one wraps such an object in a passive task to operate on it from a parallel statement.

Transfer-of-Control Statements

King's transfer-of-control statements are the **exit** and **return** statements. The **exit** statement was described above. The **return** statement is

```
return [expression] [when boolean_expression];
```

A **return** statement without an expression may appear in an active task body to end the execution of the task.

Assignment and Assignable Types

Assignment is defined for some types, and not for others. Types for which assignment is defined are said to *have assignment* or *be assignable*; types for which assignment is not defined are said to *lack assignment* or *be non-assignable*.

- Elementary (enumeration and numeric types) are assignable
- Hidden types are assignable
- Record types with the Assign aspect defined (see below) are assignable; without that aspect, they are non-assignable if they have a non-assignable component and assignable otherwise
- Record components of an assignable type may be assigned to individually even if the the record type is non-assignable
- Task types are non-assignable
- Module types are non-assignable
- Set and sequence types are assignable
- Map types are assignable if their value types are assignable and non-assignable otherwise

In addition, assignment is not defined to

- Constants of any type
- Parameters of mode **in** of any type

User-Defined Assignment, Initialization, and Finalization

Conceptually, each record type R has three attribute procedures

```
R'Assign : procedure (To : in out R; From : in R);
R'Initialize : procedure (Item : in out R);
R'Finalize   : procedure (Item : in out R);
```

which are called to perform assignment, initialization, and finalization of objects of the type (no such calls need actually be made for the default cases). These can be defined by supplying matching procedures via the `Assign`, `Initialize`, and `Finalize` aspects for the type. Since the full type for a hidden type must be a record type, these are defined for the full types of hidden types as well.

```
type R is record
   F : Integer;
end record R with
   Assign => Aardvark, Initialize => Impala, Finalize => Frog;

Aardvark : procedure (To : in out R; From : in R);
Impala : procedure (Item : in out R);
Frog   : procedure (Item : in out R);
```

The procedure `R'Default_Assign` represents how R is assigned if it is assignable and no procedure is supplied for `Assign`. It may be called within an `Assign` procedure to obtain that assignment (using an assignment statement would result in recursion).

The compiler may not optimize away objects of a type with the Finalize aspect specified.

Separate Subunits

Nested modules, subprograms, module types, and task types can have their bodies deferred to a separate compilation unit with the reserved word **`separate`**:

```
A.B.C : module is
   type T1 is (X, Y, Z);

   P : procedure (V : in T1);

   type T2 (D : T1) is task;
A.B.C : end module;

A.B.C : module body is
   P : procedure (V : in T1) is separate;

   T2 : task body is separate;
A.B.C : end module;

separate A.B.C.P : procedure (V : in T1) body is
   ...
```

```
A.B.C.P : end procedure;


separate A.B.C.T2 : task body is
   ...
A.B.C.T2 : end task;
```

The Standard Library

The Predefined Environment

We have encountered many of the types and exceptions declared in the module `King_Predefined_Environment` already. There is also

```
type Byte_List is sequence of Byte_Value;
```

Platform Information

We have encountered the maximum hardware integer values declared in the module `King.Platform_Information` already. There is also the named integer

```
Max_Hardware_Digits = implementation_defined_natural_integer;
```

This will be zero if the platform does not have hardware floating-point.

There are also types for all of the hardware integer types provided by the platform, both signed and unsigned. These have names of the form `Signed_`*n* and `Unsigned_`*n*, where *n* is the number of bytes used by the type. Typically these will be 1, 2, 4, and 8.

If the platform provides hardware floating-point, there will also be types for all of the hardware floating-point types provided. These have names of the form `Float_`*n*, where *n* is the number of digits provided by the type.

Input-Output

Binary I/O

For binary I/O, there exists the module

```
King.IO : module is
   type File_Mode is (In_File, In_Out_File, Out_File, Append_File);
   subtype Input_Mode  is File_Mode range In_File     .. In_Out_File;
   subtype Output_Mode is File_Mode range In_Out_File .. Append_File;

   Invalid_Name : constant Exception_Occurrence;
   -- Name supplied to Open or Create is invalid
   Invalid_File    : constant Exception_Occurrence; -- File not suitable for I/O
   EOF_Encountered : constant Exception_Occurrence;
   -- Attempt to read past the end of file

   type File_Handle is task
      Is_Open : function return Boolean; -- Initially returns False

      Mode : function return File_Mode with
```

```
   Precondition => Is_Open;

Name : function return String with
   Precondition => Is_Open;
-- Returns the Name used to open or create the file

Size : function return Count_Value with
   Precondition => Is_Open;
-- Returns the size of the file in bytes

Open : procedure (Name : in String; Mode : in File_Mode) with
   Precondition  => not Is_Open,
   Postcondition => Is_Open;
-- If a valid file named Name exists, opens it in Mode
-- Raises Invalid_Name if no file named Name exists
-- Raises Invalid_File if Name exists but is not valid for I/O

Create : procedure (Name : in String; Mode : in Output_Mode) with
   Precondition  => not Is_Open,
   Postcondition => Is_Open;
-- Raises Invalid_Name if Name is not a valid name for a file
-- Raises Invalid_File if Name is valid but cannot be created
-- If Name does not exist, creates an empty file with that name
-- and opens it in Mode. Mode = Append_File is the same as Out_File
-- If Name exists and Mode /= Append_File, deletes the existing file
-- and then proceeds as if the file had not existed
-- If Name exists and Mode = Append_File, opens the file in Mode

Close : procedure with
   Precondition  => Is_Open,
   Postcondition => not Is_Open;

End_Of_File : function return Boolean with
   Precondition => Is_Open and then Mode in Input_Mode;

Seek : procedure (Position : in Position_Value) with
   Precondition => Is_Open and then Position in 1 .. Size + 1;
-- Sets the current position of the file to Position
-- Size + 1 is the end of the file; attempting to read will raise
-- EOF_Encountered
-- Writing will append to the file

Value : function (Position : in Count_Value <- 0) return Byte_Value with
   Precondition => Is_Open and then
                   (Mode in Input_Mode and Position in 0 .. Size);
-- If Position > 0, effectively calls Seek with Position
-- Returns the byte at the current position in the file
-- Advances the current position of the file to the next position

Put : procedure (Item : in Byte_Value; Position : in Count_Value <- 0)
with
   Precondition => Is_Open and then
                   (Mode in Output_Mode and Position in 0 .. Size + 1);
-- If Position > 0, effectively calls Seek with Position
-- Writes Item at the current position in the file
-- Advances the current position of the file to the next position

Get : procedure (Item : out Byte_List; Position : in Count_Value <- 0)
with
   Precondition => Is_Open and then
```

```
                        (Mode in Input_Mode and Position in 0 .. Size);
      -- If Position > 0, effectively calls Seek with Position
      -- Does the equivalent of
      --    Read : for I in Item'range loop
      --       Item [I] <- Value;
      --    Read : end loop;

      Put : procedure (Item : in Byte_List; Position : in Count_Value <- 0)
      with
         Precondition => Is_Open and then
                        (Mode in Output_Mode and Position in 0 .. Size + 1);
      -- If Position > 0, effectively calls Seek with Position
      -- Calls Put for each byte in Item with Position => 0
   end task File_Handle;

   Standard_Input  : constant File_Handle;
   Standard_Output : constant File_Handle;
   Standard_Error  : constant File_Handle;
King.IO : end module;
```

Some operations are not appropriate for all files. If they are applied to an inappropriate file, `Invalid_File` is raised. Specifically, `Standard_Input`, `Standard_Output`, or `Standard_Error` cannot have `Close`, `Create`, `Open`, or `Seek` applied to them.

For all types except module and task types, there is the attribute pseudo-function

```
T'As_Byte_List : function (Item : in T) return Byte_List;
```

As with all type-based attribute subprograms with a first parameter of the type, it can be applied to a value of the type with

```
V'As_Byte_List
```

The length of the result is `T'Byte_Size`. Usually this does not require any execution; it simply allows the compiler to treat the bytes of `V` as a `Byte_List`. This is called a pseudo-function because it's really a view conversion; as such, the result is not a constant as for real functions.

For every fixed-sized subtype and every value that `'As_Byte_List` can be applied to, there is the attribute `'Byte_Size` that is the minimum number of bytes needed to hold a value of the subtype, or to hold the value.

A type is fixed-sized if it is an enumeration type, a bounded numeric type represented by a fixed number of bytes, or a record type with all components of fixed-sized types.

For all types except module and task types there is the attribute pseudo-function

```
T'From_Byte_List : function (List : in Byte_List) return T with
   Precondition => List.Length >= T'Byte_Size;
```

The bytes in `List [1 .. T'Byte_Size]` are interpreted as a value of the subtype. As with `'As_Byte_List`, this usually does not require any execution, and is really a view conversion. If it is possible for `List` to contain a sequence of bytes that is not the representation of a valid value of `T`, then a check is performed that `List` is valid. `Constraint_Violation` is raised if it is not.

Given

```
X : A;
```

the calls

```
B'From_Byte_List (X'As_Byte_List)
```

are equivalent to Ada's `Unchecked_Conversion`.

With these attributes and this module, one can perform arbitrary, heterogeneous binary I/O.

Text I/O

For Latin-1 text I/O, there exists the module

```
use King.IO;

King.IO.Text : module is
   subtype File_Handle is King.IO.File_Handle;

   subtype Input_Mode  is King.IO.Input_Mode;
   subtype Output_Mode is King.IO.Output_Mode;

   Standard_Input  : constant File_Handle renames King.IO.Standard_Input;
   Standard_Output : constant File_Handle renames King.IO.Standard_Output;

   type EOL_ID is (DOS_Windows_EOL, Mac_EOL, Unix_EOL, Native_EOL);
   -- Indicates the kind of system that output EOLs should be written for
   -- DOS_Windows_EOL: CR-LF
   -- Mac_EOL:         CR
   -- Unix_EOL:        LF
   -- Native_EOL:      The native EOL for the system the program is running on

   -- For input operations, the Boolean parameter Any_EOL, defaulted to True,
   -- indicates if the operation should recognize EOLs from any system (True)
   -- or only those native to the system the program is running on

   End_Of_Line : function
      (File : in File_Handle <- Standard_Input; Any_EOL : in Boolean <- True)
   return Boolean with
      Precondition => File.Is_Open and then File.Mode in Input_Mode;
   -- Returns True if the next thing in File is a line terminator;
   -- False otherwise

   Skip_Line : procedure (Number  : in Position_Value <- 1;
                          File    : in File_Handle    <- Standard_Input;
                          Any_EOL : in Boolean        <- True)
   with
```

```
      Precondition => File.Is_Open and then File.Mode in Input_Mode;
   -- Skips Number line terminators in File

   New_Line : procedure (Number : in Position_Value <- 1;
                         File   : in File_Handle    <- Standard_Output;
                         EOL    : in EOL_ID         <- Native_EOL)
   with
      Precondition => File.Is_Open and then File.Mode in Output_Mode;
   -- Writes Number line terminators to File

   Next : function
      (File : in File_Handle <- Standard_Input; Any_EOL : in Boolean <- True)
   return Latin_1 with
      Precondition => File.Is_Open and then File.Mode in Input_Mode;
   -- Skips any line terminators in File, then returns the next character

   Put : procedure
      (Item : in Latin_1; File : in File_Handle <- Standard_Output)
   with
      Precondition => File.Is_Open and then File.Mode in Output_Mode;
   -- Writes Item to File

   Next_Line : function
      (File : in File_Handle <- Standard_Input; Any_EOL : in Boolean <- True)
   return Latin_1_String with
      Precondition => File.Is_Open and then File.Mode in Input_Mode;
   -- Returns all the characters in File until the next line terminator
   -- Skips the line terminator

   Put : procedure
      (Item : in Latin_1_String; File : in File_Handle <- Standard_Output)
   with
      Precondition => File.Is_Open and then File.Mode in Output_Mode;
   -- Calls Put to File for each character in Item

   Put_Line : procedure (Item : in Latin_1_String;
                         File : in File_Handle <- Standard_Output
                         EOL  : in EOL_ID      <- Native_EOL)
   with
      Precondition => File.Is_Open and then File.Mode in Output_Mode;
   -- Has the effect of doing
   --    Put (Item => Item, File => File);
   --    New_Line (Number => 1, File => File, EOL => EOL);
King.IO.Text : end module;
```

Text I/O is very similar to Ada's. There are no pages, and no line or column counting. You can read and write files with line terminators for any of the major operating systems.

You can, of course, mix binary I/O operations with text operations for the same file. `End_Of_File`, being a binary operation, works correctly, unlike Ada's equivalent for text files.

There are equivalent modules `King.IO.BMP_Text` and `King.IO.Unicode_Text` for little-endian I/O of 2- and 3-byte values.

[TBD: module(s) for I/O of encoded (UTF-8, -16, -32) text]

User-defined character and string types may be written and read using the `'As_Universal` attribute function, which converts any character value to the equivalent *universal_character* value. Consider the case of Roman numbers:

```
type Roman_Numeral is ('I', 'V', 'X', 'L', 'C', 'D', 'M') with Bit_Size => 3;
type Roman_Number is sequence of Roman_Numeral;

Value : Roman_Number;
...
Put : for C of Value loop
   King.IO.Text.Put (Item => C'As_Universal);
Put : end loop;

Value.Clear;

Get_Line : loop
   exit Get_Line when King.IO.Text.End_Of_Line;

   Value <- Value & King.IO.Text.Next'As_Universal;
Get_Line : end loop;

King.IO.Text.Skip_Line;
```

There is also module `King.IO.Directories` for directory operations.

Command Line

For querying the command line, there is the module

```
King.Command_Line : module is
   Command : function return Latin_1_String;
   -- Returns the command name used to run the program

   Count : function return Count_Value;
   -- Returns the number of command-line arguments provided

   Value : function (Position : in Position_Value) return Latin_1_String with
      Precondition => Position in Position_Value range 1 .. Count;
   -- Returns the argument at Position
King.Command_Line : end module;
```

Time and Date

For obtaining and manipulating times and dates, King has the modules

```
King.Calendar : module is
   Update_Interval = implementation_defined_real_number;
   -- The value returned by Clock changes at this interval (in seconds)

   Clock : function return Natural_Duration;
   -- A monotonic clock returning the number of seconds since the epoch
   -- (which is implementation defined)
King.Calendar : end module;
```

`Calendar` is King's equivalent to Ada's Real_Time. `Update_Interval` must be one millisecond or less, ideally one microsecond or less. The **`wait for`**

statement takes a Duration that is interpreted the same as a value obtained from Clock, and blocks the task until Clock would return a value >= that given.

```
King.Calendar.UTC : module is
   To_UTC : function (Monotonic : in Duration) return Duration;
   -- Converts Monotonic, a monotonic time (an offset from the monotonic epoch),
   -- to a UTC time [an offset from the CE epoch (0001-01-01 00:00:00.00)]

   From_UTC : function (UTC : in Duration) return Duration;
   -- Converts UTC, a UTC time, to a  monotonic time

   Local_Offset : function return Duration;
   -- Returns the offset added to UTC to obtain the local system time

   subtype Year_Number is Unbounded_Integer with
      Predicate => Year_Number /= 0, Default_Value => 1;
   -- Negative values represent years BCE
   -- Zero is excluded because the calendar has no year zero

   subtype CE_Year is Year_Number with
      Predicate => CE_Year > 0;

   subtype Month_Number   is Integer  range 1 .. 12;
   subtype Day_Number     is Integer  range 1 .. 31;
   subtype Day_Seconds    is Duration range 0 .. 86_400 - Duration'delta;
   subtype Hour_Number    is Integer  range 0 .. 23;
   subtype Minute_Number  is Integer  range 0 .. 59;
   subtype Minute_Seconds is Duration range 0 .. 60 - Duration'delta;

   type Time_YMDS is record
      Year    : Year_Number;
      Month   : Month_Number;
      Day     : Day_Number;
      Seconds : Day_Seconds;
   end record Time_YMDS;

   Days_In_Month : function (Time : in Natural_Duration) return Day_Number;
   Days_In_Month : function (Year : in CE_Year; Month : in Month_Number)
   return Day_Number;

   "\" : function (Time : in Natural_Duration)  return Time_YMDS;
   "\" : function (Time : in Time_YMDS) return Duration with
      Precondition => Time.Year > 0 and then
                      Time.Day in Day_Number range
                         1 .. Days_In_Month (Time.Year, Time.Month);
   -- Conversions between CE time values and Time_YMDS

   type Time_YMDHMS is record
      Year    : Year_Number;
      Month   : Month_Number;
      Day     : Day_Number;
      Hour    : Hour_Number;
      Minute  : Minute_Number;
      Seconds : Minute_Seconds;
   end record Time_YMDHMS;

   Image : function (Time : in Time_YMDHMS) return String is
      (Time.Year'Image   (Width => 4, Zero_Filled => True) & '-' &
```

```
         Time.Month'Image  (Width => 2, Zero_Filled => True) & '-' &
         Time.Day'Image    (Width => 2, Zero_Filled => True) & ' ' &
         Time.Hour'Image   (Width => 2, Zero_Filled => True) & ':' &
         Time.Minute'Image (Width => 2, Zero_Filled => True) & ':' &
         Time.Seconds'Image
            (Before => 2, After => 2, Exponent => 0, Zero_Filled => True) ) with
      Precondition => Time.Year < 0 or else Time.Day in Day_Number range
                         1 .. Days_In_Month (Time.Year, Time.Month);

   "\" : function (Time : in Natural_Duration) return Time_YMDHMS;
   "\" : function (Time : in Time_YMDHMS) return Natural_Duration with
      Precondition => Time.Year > 0 and then
                      Time.Day in Day_Number range
                         1 .. Days_In_Month (Time.Year, Time.Month);
   -- Conversions between CE time values and Time_YMDHMS

   type Time_HMS is record
      Hour    : Hour_Number;
      Minute  : Minute_Number;
      Seconds : Minute_Seconds;
   end record Time_HMS;

   "\" : function (Time : in Day_Seconds) return Time_HMS;
   "\" : function (HMS  : in Time_HMS)    return Day_Seconds;
   -- Conversions between Day_Seconds and Time_HMS

   type Day_Name is
      (Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday);

   Day_Of_Week : function (Time : in Natural_Duration) return Day_Name;
   Day_Of_Week : function
      (Year : in CE_Year; Month : in Month_Number; Day : in Day_Number)
   return Day_Name with
      Precondition => Day in Day_Number range 1 .. Days_In_Month (Year, Month);
   -- Only defined for years CE

   Leap_Year : function (Time : in Natural_Duration) return Boolean;
   Leap_Year : function (Year : in CE_Year) return Boolean;
King.Calendar.UTC : end module;
```

Other Modules

There are also extensive module hierarchies for data structures (`King.Data_Structures`), algorithms (`King.Algorithms`), and a portable GUI (`King.GUI`), similar in concept to Ada GUI (https://github.com/jrcarter/Ada_GUI).

One of the algorithms is the macro module `King.Algorithms.Wrapping` that we encountered earlier:

```
King.Algorithms.Wrapping : macro
   type Element is (<>);
King.Algorithms.Wrapping : module
   Previous : function (Item : in Element) return Element is
      (if Item = Element'First then Element'Last else Item'Previous);

   Next : function (Item : in Element) return Element is
      (if Item = Element'Last then Element'First else Item'Next);
```

```
King.Algorithms.Wrapping : end module;
```

Reserved Words

King has the following reserved words:

```
abs         all         and
begin       body
case        constant
declare     delta       digits
else        else_if     end         exit
for         function
hidden
if          in          is
loop
macro       map         mod         module
new         not         null
of          or          others      out
procedure
range       record      rem         renames     return      reverse
select      separate    sequence    set         some        subtype
task        then        type
use
wait        when        with
xor
```

Run-Time Checks

Of course King has run-time checks similar to Ada's, including pre- and postcondition checks. Unlike Ada, King has no way to suppress them.

Unchecked Pre- and Postconditions

Normal pre- and postconditions document the software and also provide checks for correctness. However, in some cases, actually checking such conditions may lead to software not meeting its timing requirements. Consider

```
subtype Potential_Prime is Unbounded_Integer with
   Predicate => Potential_Prime > 1,
   Default_Value => 2;

P : procedure (Prime : in Potential_Prime) with
   Precondition =>
      (for all I in Potential_Prime range 2 .. Prime / 2 => Prime rem I /= 0);
```

This documents and checks that `Prime` is a prime number. `Prime` may be arbitrarily large, and performing the exhaustive check that it is prime could take an arbitrarily long time. If you can't wait 100 years for a subprogram call to proceed, you might be inclined to leave off the precondition, or state it as a comment.

For such cases, King provides the aspects `Unchecked_Precondition` and `Unchecked_Postcondition`. These provide documentation but do not generate checks. Unless your case is as egregious as this example, you should always

start with normal conditions, only making them unchecked if measurement shows it is necessary.

Parameter Association

We have seen many examples of parameter associations, but certain requirements of them may not be clear. Procedure calls, macro expansions, and discriminant associations must use named notation. Function calls must use positional notation for the first parameter. Subsequent function parameters may use either positional or named notation, whichever is clearer. When a function call uses `Object.Operation` notation, the `Object` is the first parameter, so all other parameters may use either positional or named notation.

Singleness

A general rule in King is "singleness". A use clause may only name a single module. A declaration may only declare a single thing. Each parameter declaration in a subprogram parameter list declares a single parameter. Each macro parameter declaration declares a single macro parameter. Each discriminant declaration in a discriminant list declares a single discriminant. A parameter or discriminant association only associates a single parameter or discriminant. Each line may contain at most a single statement.

An exception to this rule is in aggregates using named notation, where multiple names or index values separated by .. and | may appear.

Required Whitespace

In the remarks about the Hello-World program, it was noted that most of the spaces in the program are required. Here we'll define when whitespace is required in more detail. Some definitions

- Spaces: One or more space characters
- Line end: A line terminator and any immediately preceding comment
- Whitespace: Spaces or one or more line ends

An indentation level is three space characters.

Binary operators, the assignment symbol, stand alone colons, and arrow symbols must be preceded by spaces and followed by whitespace.

Unary operators that are symbols must be preceded by either a left parenthesis or whitespace, but not by a left parenthesis followed by whitespace, and there must be no whitespace between them and their operands.

Unary operators that are reserved words must be preceded by either a left parenthesis or whitespace, but not by a left parenthesis followed by whitespace, and separated from their operands by a single space.

Terminator semicolons must be followed by whitespace including at least one line end.

There must be no whitespace between a semicolon and what precedes it.

Sequences of parentheses/brackets/braces of the same type must be separated by spaces.

Left parentheses/brackets/braces not preceded by the same thing must be preceded by whitespace. Left parentheses/brackets/braces not followed by the same thing must be not be followed by whitespace.

Right parentheses/brackets/braces not followed by the same thing must be followed by punctuation or whitespace. Right parentheses/brackets/braces not preceded by the same thing must not be preceded by whitespace.

There must be one blank line between use clauses and the rest of the compilation unit.

There must be one blank line between compound statements and surrounding statements at the same indentation level.

There must be one blank line between transfer of control statements (return, exit, raise) and surrounding statements at the same indentation level.

There must be one blank line between wait statements and surrounding statements at the same indentation level.

There must be no blank lines between statements at different indentation levels.

There may not be more than two consecutive blank lines.

Distribution

A program may be divided into partitions, and partitions may run on physically separate machines (a distributed system). A named active task is designated the master task of a partition, and the partition name is the task name. By default, there is a single partition with the environment task as its master task. Passive tasks and other active tasks may be associated with a partition, indicating that their data are stored on and their execution is performed by the same machine as the master task of the partition. Active tasks may make calls to passive tasks in other partitions; partitioning does not change the visibility or scope of objects in the program.

A master task is indicated by the `Master_Task` aspect; things are associated with a master task by the `Partition` aspect naming the master task.

[Mechanism for multiple, dynamic instances of a partition?]

Source Code, Text Files, and File Names

A King compilation unit is a sequence of lines, each of which is a sequence of Unicode characters. King does not define how this source code is stored or

how it is provided to the compiler. However, the kinds of systems that King is expected to be used on have file systems and native text-file formats, and most compilers read their source code from such text files. It is likely that King compilers on such systems will do the same, using text input similar to `King.IO.Text` that handles line terminators in non-native files.

A King compiler that reads its source code from text files may put no restrictions on the number of compilation units in a file, or on the names of the files it reads. The operation of the compiler must be the same regardless of how compilation units are stored in files and what the file names are. The compiler must accept and process module specifications, even if the compilation does not produce any IR or object code, and regardless of whether the module body is present. It must accept and process library-level bodies, provided their specifications are known to the compiler, even if separate subunits are not present.

Suggested Organization and Naming

That said, it is usually a good idea to develop software with one compilation unit per file, with a consistent naming scheme that results in the system's tools presenting the files in a meaningful order. It is suggested that a King compilation unit be in a file named by the identifying occurrence of the full unit name, followed by an appropriate extension. Recommended extensions are .kg1 for specifications, .kg2 for library-level bodies, and .kg3 for separate subunits. So the specification of module `King.Data_Structures.Queues.Unbounded.Protected` would be in a file named King.Data_Structures.Queues.Unbounded.Protected.kg1 with its body in King.Data_Structures.Queues.Unbounded.Protected.kg2. If it has a separate subunit named `Insert`, that would be in King.Data_Structures.Queues.Unbounded.Protected.Insert.kg3.

KILLAR

The standard library has a number of modules that are difficult or impossible to implement in King. For implementing them, there is a King-like language, called the King Implementation Language for Libraries with Arrays and References (KILLAR), that can be used to implement such modules.

KILLAR adds arrays to King through array types. Array types are the same as in Ada, except that aggregates must be qualified. KILLAR also adds the concept of a half-constrained array:

```
type Something is array (Positive range 1 .. <>, Positive range 1 .. <>)
of Component;
```

KILLAR adds pointers to King through reference types.

```
type A is reference of B;
P : A; -- Initial value is null
P <- new reference of B; -- Default initialized
P <- new reference of B'(Value); -- Initialized to Value
P.all -- What P points to; raises Constraint_Violation if P = null
```

```
P.all.Field -- Record component; all is required
P.all (42) -- Indexed component; all is required
P'Free; -- Unchecked deallocation; sets P to null
```

All unreleased allocated memory for a reference type is freed when the type goes out of scope. Reference types are elementary types.

KILLAR can only compile modules (including hidden and macro modules), and those modules must declare array or reference types in their bodies. Array and reference types cannot be declared in normal module specifications, but may be declared in hidden module specifications. Normal modules compiled with KILLAR can be used from King as if they were King modules.

A normal KILLAR module declares the `KILLAR` aspect in its body, between **`body`** and **`is`**:

```
Name : module body with Killar is
```

Hidden KILLAR modules have the aspect in the specification, between **`module`** and **`is`**.

`array` and **`reference`** are reserved words in KILLAR modules, though not in King.

Technical Considerations

Compiling Hidden Types

King has separate compilation similar to Ada's, and it is possible to compile code that uses a module specification before the module body is available. Some may wonder how a compiler can deal with a unit that declares an object of type `State` if the module body is not available. How much space should be allocated for the object? How should it be initialized? One option is to introduce a level of indirection. However, the expected way it should be handled is for the compiler to do incremental compilation, and the final increment and code generation to happen when all the bodies in the system are available, but retaining the information from previous increments to minimize the effort. Incremental compilers existed in 1984, so we should be able to do this now.

In this way the need for indirection can be avoided, because the increment when the full type becomes known allows for enough space to be allocated and initialization to be arranged. The compiler would also be expected to expand calls to `Set_Seed` in line, if that would be an optimization, after its body becomes available.

Calls to Passive Task Operations

A call to a passive task operation, including to a function, proceeds much like a call to a protected entry in Ada. Since there is no restriction on what a

passive task operation may do, implementations in which a task executes an operation on behalf of another task are not suitable for King's passive tasks.

Unconstrained Record Objects

Ada's unconstrained, varying-size record types have an undefined implementation issue that causes apparent undefined semantics. For an Ada record type such as

```
type V_String (Length : Natural := 0) is record
   Value : String (1 .. Length) := (others => ' ');
end record;

S : V_String;
```

`S` may be allocated in one of two ways. One way is to allocate enough space for the largest variant. In many cases such an object will not fit on the stack, and the elaboration of `S` will raise `Storage_Error`. This is the approach used by the GNAT compiler.

The other way is to allocate `Length` and a pointer on the stack, and allocate `Value` on the heap, using only enough heap to hold the current size. This is similar to the approach used by the Janus/Ada compiler.

Thus, identical code compiled with two compilers for the same platform might work fine with one compiler but fail with another. King considers this amount of implementation variability unacceptable.

King's versions of such objects are discussed above. They can also happen directly in KILLAR (described below) with an array component.

King's requirements that objects be allocated on the heap if there is not enough room for them on the stack (described below) mean both approaches will usually work, but there will still be cases in which there will not be enough memory for the largest variant, even though the default variant will fit.

For the kind of systems King and KILLAR are currently intended for, the side effects of the first approach are simply a nuisance. In the interest of predictability and portability, King mandates the second approach.

For non-assignable discriminated types, including non-assignable record types, the discriminant cannot be changed once the object is declared, so the object is constrained by the default value of the discriminant if declared without an explicit discriminant association.

Stack and Heap Allocation

How a compiler implements subprogram calls (stack frames) is an implementation issue that should not concern the developer. In Ada, a large object that does not fit on the stack often results in `Storage_Error`, and the developer then explicitly puts the object on the heap (via an access type) to avoid this.

King requires that an object declaration succeed if there is room in memory for the object; no distinction is made between stack and heap memory. King therefore requires that an object declaration that does not fit on the stack must be tried on the heap before raising `Memory_Exhausted`.

Most large objects in King are dynamically sized, and so are probably allocated on the heap anyway, and this is not an issue. But large constant objects may be declared that need not be dynamic; consider a map constant if there are a large number of keys, and the associated values are large.

Elaboration

The `use` clauses and elaboration-time subprogram calls in a program establish a directed, acyclic graph of elaboration order. Modules must be elaborated in this order. When the graph allows a set of modules to be elaborated at the same time, the modules in the set must be elaborated in `String "<"` order of their full names.

Alternatives

Unreserved Keywords

Some of the reserved words are only used in limited circumstances (**`map`**, **`reverse`**) and might be better considered unreserved keywords that only have a special meaning in those circumstances.

Explicitly Empty Parameter Lists

In keeping with the philosophy that things that are deliberately empty should be explicitly marked as such with **`null`** (declarative regions, sequences of statements), perhaps subprogram parameter lists should, too.

```
P : procedure (null);
F : function (null) return T;
V : T is F (null);
P (null);
```

Array and Reference Types

Have them in King with the same restrictions?

Type Names

Perhaps add a prefix to all type names: `` `Type_Name `` ?