CMU-CS-80-149

# Programming In Ada: Examples

P. Hibbard, A. Hisgen, J. Rosenberg, M. Sherman

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA. 15213

October 1980



This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-78-C-1551.

# Table of Contents

# 1. Introduction

Ada [8] is a language containing many advanced features not available previously in any widely-used programming language. These features include: data abstraction mechanisms (packages, private types, derived types, overloading, user redefinition of operators), explicit parallelism and synchronization (tasks, entries, accept statements), a rich separate compilation facility and a powerful strong-typing mechanism. The use of these facilities allows for highly readable, efficient and maintainable programs. However, techniques and paradigms acquired in previous experience with other programming languages do not necessarily carry over directly into good Ada programming techniques. Indeed, our experience has been that a considerable re-learning effort is required to program comfortably in Ada.

This report is the beginning of an exploration aimed at evolving a philosophy of programming in Ada. The culmination of this effort will be a book covering a large spectrum of techniques desirable for efficient programming in Ada.

This report is *not* an introduction either to Ada or to programming. We have assumed a rather high level of knowledge of revised Ada on the part of the reader. In addition, familiarity with data structures, structured programming practices and parallelism is a prerequisite to understanding this report.

In choosing examples for inclusion, we considered the following criteria to be essential:

- The examples should be "realistic". The feature provided by an example should be of more than "academic interest".[1]
- The examples must be self-contained, complete Ada *compilations*. Portions or excerpts of programs are not acceptable.
- The examples must be small enough to be comprehensible with a reasonable amount of effort.

We did *not* choose the examples in an attempt to cover any predetermined number of language features. The aim of our research is to evolve appropriate methods of programming in Ada, and not to explore language design. We decided to allow the programs themselves to dictate the language features to be displayed.

Five examples have been included. The first example is of a generic package providing two

[1] For example, programs like the "sieve of Eratosthenes", for determining prime numbers [10], were ruled out for their lack of extra-academic utility.

abstractions for queues (FIFO lists). The example shows some aspects involved in attempting to implement a convenient, efficient and transportable library package. Both of the queue types are used in later examples in this report.

A simple directed graph package is displayed in Chapter 3. The primary purpose of this generic library package is the provision of an iterator facility. The use of the iterator, and alternative graph traversal capabilities are discussed. Some interesting trade-offs involved in information hiding versus ease of use and readability are also described.

The next example is highly machine dependent. It is a console teletype driver for a PDP-11[2]. Representation specifications, interrupt handling and machine-dependent programming are illustrated.

The fourth example, contained in chapter 5, is a package providing a string table creation and search mechanism. Interesting uses of generic packages and parameters are shown. Problems encountered in providing a "protected" mechanism are also described.

Chapter 6 contains the last example. A procedure is given that implements the relaxation method for determining the temperature distribution on a rectangular plate. The algorithm is implemented by a user-specified number of tasks. This is an interesting problem in parallelism and synchronization.

All of the examples have been checked for (compile-time) semantic correctness by a semantic analyzer for revised Ada provided to us by Intermetrics, Inc. [16].

We welcome any and all comments on this effort. Comments and suggestions may be sent by U. S. mail to:

Professor Peter Hibbard
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pa. 15213

or, via the ARPANET to:

PETER.HIBBARD@CMU-10A

---

[2] PDP is a registered trademark of the Digital Equipment Corporation.

# 2. An Implementation of Queues

## 2.1 Description

One of the most common data structures in programs are queues, which are frequently used as buffers between processing elements.

The generic package below provides two kinds of queues: a finite queue for use with sequential programs, and a finite queue for use with multi-tasking programs. Each instantiation of the package requires a type parameter that specifies the type of the queued elements. After instantiation, any number of queues may be declared by using the types `Queue` and `Blocking_Queue`.

In addition to the package specifications, the following information applies to the use of the package:

- For a queue variable, `Q`, the invocation `Init_Queue` must be made once. This invocation must proceed any use of the `Append`, `Remove`, `Is_Full`, `Is_Empty` or `Destroy_Queue` subprograms with queue `Q`.
- When any queue variable `Q` is no longer needed, the invocation `Destroy_Queue(Q)` *should* be made.
- The `MaxQueuedElts` discriminant to the queue types represents a minimum performance specification. All implementations of `Queue_Package` will guarantee that at least `MaxQueuedElts` number of items can be held in a queue.

The two types of queues have different semantics for the `Append` and `Remove` operations. The semantics for type `Queue` are:

- If the specified queue is empty and a call to `Remove` is made, the exception `Empty_Queue` will be raised.
- If the specified queue is full and a call to `Append` is made, the exception `Full_Queue` will be raised.

The semantics for type `Blocking_Queue` are:

- If the specified queue is empty and a call to `Remove` is made, the calling task will be blocked until a corresponding call to `Append` is made.
- If the specified queue is full and a call to `Append` is made, the calling task will be blocked until a corresponding call to `Remove` has been made.
- All operations provided for type `Blocking_Queue` are indivisible.

## 2.2 Implementation

The technique used for the static queue is a simple circular array. It is fully analyzed elsewhere [9].

The technique used for the blocking queue is the same except there is an **accept** statement surrounding each operation to provide mutual exclusion and blocking.

## 2.3 Program Text

```
generic
   type EltType is private;

package Queue_Package is

   type Queue(MaxQueuedElts : Natural) is limited private;

   procedure Append(Q : in out Queue; E : in EltType);
   procedure Remove(Q : in out Queue; E : out EltType);
   function Is_Empty(Q : in Queue) return Boolean;
   function Is_Full(Q : in Queue) return Boolean;
   procedure Init_Queue(Q : in out Queue);
   procedure Destroy_Queue(Q : in out Queue);

   Full_Queue, Empty_Queue : exception;

   type Blocking_Queue(MaxQueuedElts : Natural) is limited private;

   procedure Append(Q : in out Blocking_Queue; E : in EltType);
   procedure Remove(Q : in out Blocking_Queue; E : out EltType);
   function Is_Empty(Q : in Blocking_Queue) return Boolean;
   function Is_Full(Q : in Blocking_Queue) return Boolean;
   procedure Init_Queue(Q : in out Blocking_Queue);
   procedure Destroy_Queue(Q : in out Blocking_Queue);

   pragma Inline(Is_Empty,Is_Full,Init_Queue,Destroy_Queue);

private

   subtype Non_Negative is Integer range 0..Integer'LAST;

   type Queue(MaxQueuedElts : Natural) is
      record
         FirstElt, LastElt : Non_Negative := 0;
         CurSize : Non_Negative := 0;
         Elements: array(0..MaxQueuedElts) of EltType;
      end record;
```

```
   task type Blocking_Queue_Task is
      entry Pass_Discriminants(Queue_Size : in Natural);
      entry Put_Element(E : in EltType);
      entry Get_Element(E : out EltType);
      entry Check_Full(B : out Boolean);
      entry Check_Empty(B : out Boolean);
      entry ShutDown;
   end Blocking_Queue_Task;

   type Blocking_Queue(MaxQueuedElts : Natural) is
      record
         Monitor : Blocking_Queue_Task;
      end record;

end Queue_Package;

-------------------------------------------------------------------------

package body Queue_Package is

   pragma Inline(Is_Empty,Is_Full,Init_Queue,Destroy_Queue);

   procedure Append(Q : in out Queue; E : in EltType) is
   begin
      if Q.CurSize = Q.MaxQueuedElts then
         raise Full_Queue;
      else
         Q.CurSize := Q.CurSize + 1;
         Q.LastElt := (Q.LastElt + 1) mod Q.MaxQueuedElts;
         Q.Elements(Q.LastElt) := E;
      end if;
   end Append;

   procedure Remove(Q : in out Queue; E : out EltType) is
   begin
      if Q.CurSize = 0 then
         raise Empty_Queue;
      else
         Q.CurSize := Q.CurSize - 1;
         Q.FirstElt := (Q.FirstElt + 1) mod Q.MaxQueuedElts;
         E := Q.Elements(Q.FirstElt);
      end if;
   end Remove;

   function Is_Full(Q : in Queue) return Boolean is
   begin
      return Q.CurSize = Q.MaxQueuedElts;
   end Is_Full;

   function Is_Empty(Q : in Queue) return Boolean is
   begin
      return Q.CurSize = 0;
   end Is_Empty;
```

```
procedure Init_Queue(Q : in out Queue) is
begin
   null;
end Init_Queue;

procedure Destroy_Queue(Q : in out Queue) is
begin
   null;
end Destroy_Queue;

task body Blocking_Queue_Task is
   MaxSize: Natural;
begin
   accept Pass_Discriminants(Queue_Size : in Natural) do
      MaxSize := Queue_Size;
   end Pass_Discriminants;

   declare
      Queued_Elements: Queue(MaxSize);
   begin
      Init_Queue(Queued_Elements);

   Monitor_Operations:
      loop
         select
            when not Is_Full(Queued_Elements) =>
               accept Put_Element(E : in EltType) do
                  Append(Queued_Elements, E);
               end Put_Element;
         or
            when not Is_Empty(Queued_Elements) =>
               accept Get_Element(E : out EltType) do
                  Remove(Queued_Elements, E);
               end Get_Element;
         or
            accept Check_Full(B : out Boolean) do
               B := Is_Full(Queued_Elements);
            end Check_Full;
         or
            accept Check_Empty(B : out Boolean) do
               B := Is_Empty(Queued_Elements);
            end Check_Empty;
         or
            accept ShutDown;
            exit Monitor_Operations;
         or
            terminate;
         end select;
      end loop Monitor_Operations;
      Destroy_Queue(Queued_Elements);
   end;
end Blocking_Queue_Task;
```

```
      procedure Append(Q : in out Blocking_Queue; E : in EltType) is
      begin
         Q.Monitor.Put_Element(E);
      end Append;

      procedure Remove(Q : in out Blocking_Queue; E : out EltType) is
      begin
         Q.Monitor.Get_Element(E);
      end Remove;

      function Is_Full(Q : in Blocking_Queue) return Boolean is
         Temp : Boolean;
      begin
         Q.Monitor.Check_Full(Temp);
         return Temp;
      end Is_Full;

      function Is_Empty(Q : in Blocking_Queue) return Boolean is
         Temp: Boolean;
      begin
         Q.Monitor.Check_Empty(Temp);
         return Temp;
      end Is_Empty;

      procedure Init_Queue(Q : in out Blocking_Queue) is
      begin
         Q.Monitor.Pass_Discriminants(Q.MaxQueuedElts);
      end Init_Queue;

      procedure Destroy_Queue(Q: in out Blocking_Queue) is
      begin
         Q.Monitor.ShutDown;
      end Destroy_Queue;

   end Queue_Package;
```

## 2.4 Discussion

### 2.4.1 Use of Limited Private Types

The ability to declare queues is provided via the limited private types `Queue` and `Blocking_Queue`. The need for the private specification should be clear. The only way the package can guarantee correct operation of the `Remove` and `Append` procedures is to prevent the package user from having access to the internal representation. To allow the changing of the queue representation, the package must also guarantee that no part of the user's program depends on the current implementation.

The need to have the type limited is a bit more subtle. Suppose assignment were permitted between

two variables of type Queue. After assigning one queue variable to another queue variable, we need to know whether the two variables represent two different queues with the same elements being queued and in the same order, or whether the assignment means that the two variables denote the same queue. In the current implementation, the former semantics would be supported by the assignment statement. If a typical implementation using dynamic storage were provided, the latter interpretation would probably prevail. There are two different "meanings" for the assignment statement. The equality operator would be even more fuzzy. For example, what if two different queues contained the same elements but they happened to be in different array locations in the particular implementation above? For this case, the predefined equality operator would unexpectedly return False. To avoid having the user program depend on one of these interpretations, which could be changed easily, the visible types are both limited and private.

### 2.4.2 Initialization and Finalization

In the description of this package, it is asserted that all queues must be initialized by the Init_Queue subprogram. Examination of the package body reveals that the Init_Queue subprogram does nothing for nonblocking queues. It would therefore appear to be unnecessary. However, consider the case where the representation changed from an array to a pointer, and where the implementation used linked list elements with a dummy block at the front of the queue.[3] This representation requires some preprocessing to initialize the data structure before it can be used. To ensure that changing representations is possible, all queue packages provide Init_Queue in their specifications.

It should be noted that other solutions exist for the initialization problem. It is possible to enclose the data needed for the abstraction in a record along with a boolean variable that indicates if the record has been initialized. Default values for record types allow one to guarantee that this boolean variable has the initial value False. Every routine in the package would validate the data structure (by checking the boolean field in the record) before using it. We believe that such an approach is generally wasteful. Many of the clever algorithms for data structure manipulation attempt to avoid extra tests by special features, such as the dummy blocks mentioned above. Requiring a special test before any use of the variable nullifies the added value of many of these algorithms.

Finalization is not as easy to deal with as initialization. The current implementation of nonblocking queues, as arrays, needs no final processsing when the scope containing the queue variable is exited. Exiting from blocks and procedures reclaims the storage automatically. But, suppose the queue's elements were stored in explicitly allocated storage, via new. When the scope containing the queue variable is exited, the storage for the queue's elements will remain. By providing a user-level

[3] This technique is discussed in Wulf, et. al. [19].

mechanism, such as the `Destroy_Queue` subprogram, the package assumes the burden of providing a means for releasing unneeded objects.

### 2.4.3 Passing Tasks as In Out Parameters

It would seem that the limited private type for `Blocking_Queue` should be a task type and not a record type. If the type had been a task, the only permissible parameter mode for a `Blocking_Queue` parameter would have been **in**. Thus, the specification of the `Append` procedure would declare the blocking queue parameter as being passed with the **in** mode. This looks very confusing to a programmer working from the package specifications. It is usually assumed that objects that are changed by a subprogram are passed with the **in out** mode. The philosophy of data abstraction in Ada requires that such knowledge be hidden from the user of the package to insure that such knowledge is not exploited in some way. A record may be passed with the **in out** mode, even if contains a task type as a component. This technique helps hide the actual implementation.

Furthermore, if the limited private type for `Blocking_Queues` were a task, user programs could not write subprograms that passed `Blocking_Queues` as **in out** parameters. This would be a natural way to program as many systems would build subprograms where a queue would be passed along with other data.

There is a third advantage of using a record type rather than a task type. With a record type, the Ada specifications for blocking queues and nonblocking queues are identical. The user could therefore switch from one to the other without having to change other parts of his program (except as it might rely on other parts of the semantics of the queue package).

### 2.4.4 Passing Discriminants to Tasks

As this example illustrates, it is useful to give parameters to an abstract data type, in this case a queue size specification and an element type. Two methods can be used to accomplish this: a generic parameter can be passed to the package or a discriminant can be provided for the declared type.

Instantiating the package once for each queue size clutters the program text. Furthermore, it means that each instantiation generates a new queue type. The user would have to duplicate subprograms to handle each size.[4]

Allowing the size of an abstract data type to be part of a subtype is more convenient and familiar to a programmer. This is done by using a discriminant constraint in a record type. Both varieties of queues are therefore implemented with records.

---

[4]This is similar to a well known problem in Pascal: arrays with different sizes are different types. This means that one cannot easily write a procedure to sort an arbitrary integer array.

For the blocking queue, there must be a way to pass these discriminant values to the task. This is done by means of the `Init_Queue` procedure and the `Pass_Discriminants` entry.

### 2.4.5 The Elements Array

A close reading of the program reveals that the last element of the `Elements` array is never used. This results from an interaction between the restrictions on discriminants in Ada and the use of **mod** in the algorithm for calculating index values. The formula for calculating the `FirstElt` and `LastElt` values relies on the **mod** function to wrap the index around, from `MaxQueuedElts-1` to 0. This avoids a special test when a value is at its maximum, which would be necessary if the algorithm used array elements 1 through `MaxQueuedElts`. The rules for discriminants allow only the discriminant to appear as an array bound in a record, not a expression. So, `MaxQueuedElts` must be used instead of `MaxQueuedElts-1`.

### 2.4.6 Remove as a Procedure

It might seem natural to have specified `Remove` as a function instead of a procedure. However, Ada restricts the modes of parameters to functions to **in**. Since we made the decision to pass a queue **in out** if it is to be modified, we were forced to make `Remove` a procedure.

# 3. A Simple Graph Package Providing an Iterator

## 3.1 Description

Graphs, of one form or another, are an important data structure throughout most of computer science. This example displays an implementation of a simple package providing an abstraction of directed graphs. The specification includes some type definitions, culminating in the `Obj` record type, which serves to define the structure of directed graphs. The user is responsible for allocation, initialization and manipulation of the nodes which form a graph.

The primary function of the package is to provide an "iterator". Iterators provide a means for enabling a user-definable looping abstraction mechanism. This particular iterator provides a facility which may be used to breadth-first traverse a graph.

One might expect that a "complete" library graph package would provide a much healthier range of functionality than our example does. In addition, it would seem advisable to hide much of the currently-exposed detail. The package could provide a uniform, implementation-independent manner of creating, manipulating and destroying nodes.

We avoided such an implementation so as to allow the reader an unobstructed view of the iterator.

The algorithm for the breadth-first traversal is a modified version of that in Horwitz and Sahni, page 264 [7] (see section 3.4.1 of this paper). Other traversal methods, expressed as iterators, could easily be provided. For example, a depth-first traversal would be a primary candidate.

## 3.2 Specifications

To use the iterator, one must declare a variable of type `Breadth_First`. The iterator is initialized by invoking the procedure `Start`. `Start` takes three parameters:

| | |
|---|---|
| `B` | The iterator to initialize. |
| `N` | The node at which the breadth-first traversal is to begin. If `N=null`, the exception `Null_Node` is raised and initialization is not completed. |
| `Max_Nodes` | The maximum number of nodes which might be reached during this search. A rather generous estimate of this value may be provided with little cost in unnecessary overhead. If this number proves to be insufficient, the exception `Too_Many_Nodes` will be raised at some point by the `Next` function (see below). |

Due to the algorithm chosen, there is an upper limit on the total number of graph iterations which may

be started (whether used to completion or not). If this limit is exceeded, the exception `Too_Many_Traversals` will be raised (see section 3.4.1).

Once an iterator has been properly initialized, three routines are available:

- The `More` function may be called to determine if there are reachable nodes which have not yet been generated.
- The next node in the search is obtained via the `Next` function. For any iterator, `B`, if `More(B)=False`, then the invocation `Next(B)` will raise the exception `End_Of_Graph`.
- To inform the iterator of the end of its usefulness, the `Stop` procedure is provided. After invocation, the iterator is available for re-use if desired.

An attempt to invoke `More`, `Next` or `Stop` with an uninitialized iterator will cause the `Start_Error` exception to be raised. Attempting to `Start` an iterator more than once, without invoking `Stop`, will also raise this exception. In addition, only one iterator per package instantiation may be active at any time. `Start_Error` will be raised if this is violated.

A client of this package should not rely on any semantics which depends on actions taken if edges of a graph are modified during a traversal.

## 3.3 Program Text

```
-------------------------------------------------------------------
--
-- Directed Graph Package with Iterator
--
-------------------------------------------------------------------

generic
   type Item is limited private;

package Graph_Package is

   type Obj;
   type Node is access Obj;
--
-- The following two type definitions allow a node to have an arbitrary
-- number of descendants.
--
   type Sons_Array is array(Natural range <>) of Node;
   type Sons is access Sons_Array;

   type Hidden_Type is limited private;
```

```
      type Obj is
         record
            Contents : Item;
            Descendants : Sons;
            Hidden : Hidden_Type; -- "Hidden" field
         end record;

   --
   -- The following specifies the breadth first iterator
   --

      type Breadth_First is limited private;

      procedure Start(B : in Breadth_First;
                      N : in Node;
                      Max_Nodes : in Natural);
      function More(B : in Breadth_First) return Boolean;
      function Next(B : in Breadth_First) return Node;
      procedure Stop(B : in Breadth_First);

      Start_Error, Null_Node, End_Of_Graph : exception;
      Too_Many_Traversals, Too_Many_Nodes : exception;

   private

      type Hidden_Type is
         record          -- To obtain default initialization
            Counter : Integer := Integer'FIRST;
         end record;

      task type Breadth_First is
         entry Start(N : in Node; Max_Nodes : in Natural);
         entry More(B : out Boolean);
         entry Next(N : out Node);
         entry Stop;
      end Breadth_First;

   end Graph_Package;

   ------------------------------------------------------------------

   with Queue_Package;          -- From Chapter 2 of this report

   package body Graph_Package is

      Start_Flag : Boolean := False;

      Counter : Integer := Integer'FIRST;

   -- Counter is incremented by 1 at the start of each traversal.  Each
   -- node's Counter field (within Hidden) contains a copy of the value
   -- of Counter the last time the node was generated.  Upon reaching a
   -- node its Counter field is compared with the variable Counter to
   -- determine if it has already been seen during this traversal.
```

```
procedure Start(B : in Breadth_First;
                N : in Node;
                Max_Nodes : in Natural) is
begin
   if Start_Flag then raise Start_Error; end if;
   if N = null then raise Null_Node; end if;
   B.Start(N, Max_Nodes);
   Start_Flag := True;
end Start;

function More(B : in Breadth_First) return Boolean is
   Flag : Boolean;
begin
   if not Start_Flag then raise Start_Error; end if;
   B.More(Flag);
   return Flag;
end More;

function Next(B : in Breadth_First) return Node is
   N : Node;
begin
   if not Start_Flag then raise Start_Error; end if;
   B.Next(N);
   return N;
end Next;

procedure Stop(B : in Breadth_First) is
begin
   if not Start_Flag then raise Start_Error; end if;
   B.Stop;
   Start_Flag := False;
end Stop;

task body Breadth_First is

   Current : Node;              -- Node to expand next

   Size : Natural;              -- This holds a copy of the Max_Nodes
                                -- parameter to the Start entry.
                                -- It is necessary because scope of
                                -- entry parameter is limited to
                                -- accept body.


 begin

    accept Start(N : in Node; Max_Nodes : in Natural) do
       if Counter = Integer'LAST then
          raise Too_Many_Traversals;
       end if;
       Counter := Counter + 1;
       Current := N;
       N.Hidden.Counter := Counter;
       Size := Max_Nodes;
    end Start;
```

```
            declare
               package Q is new Queue_Package(Node);
               Queue : Q.Queue(Size);
               procedure Next_Body(N : out Node) is separate;
                 -- Separately compile to aid readability
            begin
               Q.Init_Queue(Queue);

            Iterator_Operations:
               loop
                  select

                     accept More(B : out Boolean) do
                        B := Current /= null;
                     end More;

                  or
                     accept Next(N : out Node) do
                        Next_Body(N);
                     end Next;

                  or
                     accept Stop do
                        Q.Destroy_Queue(Queue);
                     end Stop;
                     exit Iterator_Operations;

                  or
                     terminate;          -- Simply die, when scope exited

                  end select;
               end loop Iterator_Operations;
            end;

         end Breadth_First;

      end Graph_Package;
```

```
-----------------------------------------------------------------
--
-- Body of Next_Body
--
-----------------------------------------------------------------

separate (Graph_Package)

procedure Next_Body(N : out Node) is

begin

   if Current = null then
      raise End_Of_Graph;
   end if;

   N := Current;
   if N.Descendants /= null then
      for I in N.Descendants.all'RANGE loop
         declare
            Desc : Node renames N.Descendants(I);
         begin
            if Desc /= null and then Desc.Hidden.Counter < Counter then
               -- This node has not been seen this traversal
               Q.Append(Queue, Desc);
               Desc.Hidden.Counter := Counter;
            end if;
         end;
      end loop;
   end if;

   if Q.Is_Empty(Queue) then
      Current := null;
   else
      Q.Remove(Queue, Current);
   end if;

exception
   when Q.Full_Queue =>
      Start_Flag := True;
      Q.Destroy_Queue(Queue);
      raise Too_Many_Nodes;

end Next_Body;
```

## 3.4 Discussion

### 3.4.1 The Algorithm

We chose the particular algorithm for graph traversal because it obviated the need for a "visited" flag in each node. This enabled us to eliminate the associated overhead required to maintain access to all of the flags (to clear them before each traversal).

The algorithm works by maintaining a single global counter in each iterator and a local counter in each node. The global counter is incremented by one at the beginning of each traversal (initiated by a call to `Start`). The local counter in a node indicates the value contained in the global counter the last time the node was reached. During a traversal, the local counter of a node may be compared with the global counter to determine if the node was already reached (during this traversal). This node was seen earlier if and only if the local counter is equal to the global counter.

To maintain correctness, when the global counter overflows, we must disallow further traversals. (Only traversals for the particular package instantiation are forbidden, since the global counter is allocated on a per-instantiation basis.) This could be considered a serious restriction on the usefulness of this facility for some compiler implementations. Consider, that on a PDP-11, standard integers are only 16 bits. This allows only about 64,000 traversals.

A reasonable machanism for alleviating this problem would be to add a second generic parameter to `Graph_Package`:

```
generic
   type Item is private;
   type Counter_Type is range <>;
```

The user would supply any `Integer` type as `Counter_Type`. The type would be used to declare the global and local counters. This would allow users of implementations which support larger `Integer` types, such as `Long_Integer` or `Long_Long_Integer`, to effect an enormous upper limit on the number of traversals.[5]

### 3.4.2 Information Hiding

Frequently, when designing a package to provide an abstract type, one desires to provide the type with two kinds of structure: a part which is visible to (and, usually, manipulable by) the client, and, a part which is hidden from him. This hidden part generally contains data which is specific to the particular implementation. Good programming practice dictates that the user be protected from himself, by not allowing him access to this information. There seems to be no prescribed manner for implementing this in Ada. However, there are several choices available.

---

[5] A 36 bit `Integer` type, as on the PDP-10, would allow for approximately 64 billion traversals. If traversals were initiated at the average rate of 100/second, it would take 20 years to exhaust the capacity of an iterator.

The obvious method is to hide *all* of the information from the customer. This is easily accomplished by specifying a **(limited) private** type definition for the abstract type. The disadvantage of this approach is that the package must provide a means for accessing and modifying the "visible" parts of the type structure. This is most easily done by specifying procedures (and, possibly, functions) which perform the actions within the body of the package. Our limited experience indicates that this adds a significant amount of complexity to the specification and use of the abstraction.

For example, the statement

```
X.Contents.Flag := True;
```

might need to be expressed as

```
Temp_Contents := Get_Contents(X);
Temp_Contents.Flag := True;
Set_Contents(X, Temp_Contents);
```

This method does, however, allow a fine degree of control over the abstraction since all access to the objects is strictly controlled.

We believe the scheme we have chosen provides less control but a greater degree of readability than the former scheme.

The idea is to force a form of hiding by placing the desired parts within the public type, but contained within a field whose type is **limited private**. This prevents the user from doing anything "significant" with the data. Unfortunately, the user can still select the hidden field, as well as declare variables, record fields and formal parameters of this type. Ultimately, however, nothing may be "done" with these objects.

A more serious problem is that the user is not able to perform assignment between objects of the abstract type. This is because it contains a **limited private** field. In contrast, equality for the type may be provided by the package explicitly, if desired.

Admittedly, the solution is imperfect, yet, it provides a form of information hiding intermediate between the previous method and no hiding at all.

### 3.4.3 In/In Out Parameters

In section 2.4.6, the point was made that parameters that will be modified should probably be passed **in out**. In this specification of `Start`, `Next` and `Stop` we have not followed this policy. We felt that in this case, the readability gained by allowing `Next` to be a function, outweighed the other considerations. (Recall that functions are not allowed to have **in out** parameters.)

### 3.4.4 Using the Iterator

We demonstrate a use of the iterator facility in the following example.

```
function Reach(From, To : in Node) return Boolean is

--
-- Determine if node To is reachable from node From
--

   B : Breadth_First;

begin

   Start(B, From, Size);          -- Size is a global
   while More(B) loop
      if Next(B) = To then
         Stop(B);
         return True;
      end if;
   end loop;
   Stop(B);
   return False;

end Reach;
```

Using an alternative loop termination technique, we might replace the body of Reach by

```
   Start(B, From, Size);          -- Size is a global
   loop

      begin
         if Next(B) = To then
            Stop(B);
            return True;
         end if;

      exception
         when End_Of_Graph =>
            Stop(B);
            return False;
      end;

   end loop;
```

### 3.4.5 Iterators Versus Generic Procedures

Our decision to provide an iterator is based upon our belief that iterators provide a natural and familiar mechanism for looping. The Ada **for** loop is a simple form of iterator. Several languages, notably IPL-V [14], CLU [12] and Alphard [6, 15], have included user-definable iterator facilities directly in the language definitions.

An alternative to an iterator is to define a generic procedure to enable graph traversal. Consider the specification

```
generic
   with procedure Visit(N : in Node; Continue : out Boolean);
procedure Breadth_First(N : in Node; Max_Nodes : in Natural);
```

To use this facility, the client instantiates `Breadth_First` with a procedure that performs the desired actions on its `Node` parameter. The `Continue` parameter is used to inform the `Breadth_First` procedure of when to stop. `Breadth_First` would operate by invoking `Visit` (in reality, the actual procedure parameter instantiated for `Visit`) once for each node reached. `Breadth_First` terminates its actions when `Continue` becomes `False`.

The procedure `Reach`, shown previously, might be written

```
function Reach(From, To : in Node) return Boolean is
   Result : Boolean := False;
   procedure Visit(N : in Node; Continue : out Boolean) is
   begin
      if N = To then
         Result := True;
         Continue := False;
      else
         Continue := True;
      end if;
   end Visit;
   procedure Walk is new Breadth_First(Visit);
begin
   Walk(From, Size);                -- Size is a global variable
   return Result;
end Reach;
```

Our primary argument against the generic procedure is that its use obscures the fact that looping is being performed. On the other hand, the iterator provides a facility whose use displays the looping.

# 4. A Console Driver for a PDP-11

## 4.1 Description

A typical function in embedded systems is performed by a device driver which provides a convenient interface between a system and the particular hardware requirements of an input/output device. Some of the functions performed by this program are buffering of requests for the device, ensuring the integrity and validity of these requests, and fielding interrupts from the hardware.

This example illustrates the general organization which may be used, with some specific details for a PDP-11 console terminal [5, 4].

The device driver program makes the following assumptions about the underlying run-time system and implementation:

- The type that describes the available hardware for the `Low_Level_IO` package includes the constants `Console_Keyboard_Control`, `Console_Keyboard_Data`, `Console_Printer_Control`, and `Console_Printer_Data`. It is assumed that the system will treat calls of `Send_Control` and `Receive_Control` as read and write operations with the correct size and at the correct location (e.g., operations on console devices are mapped into reading and writing locations 777560-777566 octal).

- Specifying a location for an entry means that interrupts which use that location will be translated into a call on the entry.

- No data are explicitly passed by an interrupt. Therefore, all entries that have representation specifications must have no parameters.

In addition to the task specifications, the following information is necessary to use this package:

- The package makes no guarantees about servicing all interrupts. If the underlying run-time system can guarantee that all interrupts will be translated into entry calls, then the package will not lose any interrupts. This says nothing about the proper servicing of those interrupts. On the PDP-11, the datum indicated by an interrupt is lost if the data register of a device is not read before the next interrupt is processed.

- If requests for a device occur faster than the device can process them, the driver will cause the requesting process to block until the request can be processed properly.

- A one-half second delay while waiting for the output device is sufficient time to allow completion of a request. If the output device does not respond within one-half second, the program will initiate the transmission of the next character. Retransmission of characters is not attempted.

- 264 characters of data buffering are provided. The characters passed to the `Write_Character` procedure will be output without modification.

- The ShutDown entry is used as a way to terminate the device driver cleanly. When a call to ShutDown is made, the currently buffered data will be destroyed. No further processing of these data or servicing of outstanding interrupts will be done by the device driver.

- The Reset entry is functionally equivalent to a ShutDown entry call followed by a re-elaboration of the task declaration. The Reset procedure will also try to send the necessary control signals to reset the hardware device.

Knowledge of the following is also necessary to understand the functioning of this program:

- The device interrupt vectors start at 60 octal for the input device, 64 octal for the output device.

- Device interrupts are enabled by sending the value 100 octal to the device's control register.

## 4.2 Implementation

A central problem in this example is implementing asynchronous processes with the synchronous mechanism of rendezvous provided by Ada. This is done using three explicit tasks that monitor requests from the program, interrupts from the input device, and interrupts from the output device. All three tasks communicate via shared queues. As long as queues allow fast access, no part of the system will be blocked while waiting for another task to complete a rendezvous. However, if the program produces requests faster than the device can process them, a queue can become full and the requesting task will be blocked.

## 4.3 Program Text

```
package Terminal_Driver_Package is

   task Terminal_Driver is
      entry Read_Character(C : out Character);
      entry Write_Character(C : in Character);
      entry Reset;
      entry ShutDown;
   end Terminal_Driver;

end Terminal_Driver_Package;
```

```
with Queue_Package, Low_Level_IO;
use Low_Level_IO;

package body Terminal_Driver_Package is

   task body Terminal_Driver is

      -- Group all of the machine dependent constants together

      Console_Input_Vector : constant := 8#60#;
      Console_Output_Vector : constant := 8#64#;
      Enable_Interrupts : Integer := 8#100#;
      Write_Time_Out : constant Duration := 0.5;
      Number_Of_Lines: constant := 2;
      LineLength: constant := 132;

      task type Device_Reader is
         entry Interrupt;
         entry StartUpDone;
         for Interrupt use at Console_Input_Vector;
      end Device_Reader;

      task type Device_Writer is
         entry Interrupt;
         entry StartUpDone;
         for Interrupt use at Console_Output_Vector;
      end Device_Writer;

      package Char_Queue_Package is new Queue_Package(Character);
      use Char_Queue_Package;

      type DriverStateBlock is
         record
            InputCharBuffer, OutputCharBuffer :
               Blocking_Queue(Number_Of_Lines*LineLength);
            CurReader : Device_Reader;
            CurWriter : Device_Writer;
         end record;

      type RefToBlock is access DriverStateBlock;
      CurState: RefToBlock;

      task body Device_Reader is
         TempInput : Character;
      begin
         accept StartUpDone;
         Send_Control(Console_Keyboard_Control, Enable_Interrupts);
         loop
            accept Interrupt do
               Receive_Control(Console_Keyboard_Data, TempInput);
            end Interrupt;
            Append(CurState.InputCharBuffer, TempInput);
         end loop;
      end Device_Reader;
```

```
task body Device_Writer is
   TempOutput : Character;
begin
   accept StartUpDone;
   Send_Control(Console_Printer_Control, Enable_Interrupts);
   accept Interrupt; -- spurious interrupt caused by Send_Control
   loop
      Remove(CurState.OutputCharBuffer, TempOutput);
      Send_Control(Console_Printer_Data, TempOutput);
      select
         accept Interrupt;
      or
         delay Write_Time_Out;
      end select;
   end loop;
end Device_Writer;

procedure ShutDownOld is
begin
   raise CurState.CurReader'FAILURE;
   raise CurState.CurWriter'FAILURE;
   Destroy_Queue(CurState.InputCharBuffer);
   Destroy_Queue(CurState.OutputCharBuffer);
end ShutDownOld;

procedure StartUp is
begin
   CurState := new DriverStateBlock;
   Init_Queue(CurState.InputCharBuffer);
   Init_Queue(CurState.OutputCharBuffer);
   CurState.CurReader.StartUpDone;
   CurState.CurWriter.StartUpDone;
end StartUp;
```

```
    begin
       StartUp;

    Console_Operations:
       loop
          select
             accept Read_Character(C : out Character) do
                Remove(CurState.InputCharBuffer, C);
             end Read_Character;
          or
             accept Write_Character(C : in Character) do
                Append(CurState.OutputCharBuffer, C);
             end Write_Character;
          or
             accept Reset do
                ShutDownOld;
                StartUp;
             end Reset;
          or
             accept ShutDown;
             ShutDownOld;
             exit Console_Operations;
          or
             terminate;
          end select;
       end loop Console_Operations;
    exception
       when Terminal_Driver'FAILURE =>
          ShutDownOld;

    end Terminal_Driver;

end Terminal_Driver_Package;
```

## 4.4 Discussion

### 4.4.1 Use of a Package to Surround the Task

Ada does not allow tasks to be in a library. To allow the inclusion of the `Terminal_Driver` task into a library, we have elected to enclose it in a package.

### 4.4.2 Distinction Between Task Types and Tasks

Unlike most examples of tasks in this report, this task does not explictly define a type. Task types are usually used when many objects of a particular class are desired. It is probable that only one driver per physical device is needed. Although it can be argued that a type would allow the use of the same task type for different terminals (or even devices), the restrictions on address specifications weakens this argument. The device addresses must be static expressions. They cannot be passed as

discriminants to a type (or via an initializing entry call). The only reasonable way to pass these parameters at compile time would be to use generic parameters to the package. The object instantiated would still be a task, not a task type.

This package is not generic because of the number of parameters which would be necessary: four device names, buffering sizes, and time-out values. The use of the device information is also tightly wired into the task. For example, it is known that immediately after enabling interrupts on the output device, a spurious interrupt will be generated that provides no data. This peculiarity may not be true of all devices. Declaring a task type might tempt users to use this driver improperly.

The user may wish to create some higher level operations and capture the device driver as part of the representation for the abstraction. Unfortunately, the decision to declare a task instead of a task type has precluded this possibility.

### 4.4.3 Resetting and Terminating the Terminal Driver

Unfortunately, devices malfunction. A device driver must be able to deal effectively with this problem. The Ada language intended the `FAILURE` exception and **abort** statement to be used for these purposes, but use of these techniques causes an irreversible exit from the scope of the device driver. By providing the user with additional entries to manipulate the operation of the driver, as well as the operations within the driver, the driver task circumvents this restriction.

### 4.4.4 Interfacing to Devices

The example shows one possible interface between the hardware and the run-time system for Ada. Two important decisions concern the lack of parameters to entries that are mapped to interrupts, and the use of the `Low_Level_IO` procedures to access device registers rather than the direct reading or writing of the (virtual memory) device registers.

When a device interrupts the PDP-11, it usually means that a datum can be read from the device's data register. It might seem natural to pass this datum to the entry call mapped to the interrupt. This choice was rejected for three reasons. First, it would require an elaborate explanation of the implementation of entry calls in appendix F of the Language Reference Manual [8]. Entry calls are supposed to be a machine-independent part of the language. The proper place to describe machine dependencies is within the `Low_Level_IO` package, which is intended to be machine dependent.

Second, attaching these details to the entry call semantics makes addition of devices more difficult. Each time a new device is added, the compiler must be changed to provide a semantics for it, in particular, the new kind of calling sequence. With the interpretation used in this example, the compiler can translate all entry calls with representation specifications into the same style of code.

Third, it would move the buffering decisions for interrupts from the device driver task to the run-time system. The number of pending interrupts that have not been accepted by a program depends on the amount of buffering the run-time system provides. If input data must also be stored, the number of pending entry calls (from interrupts) is more limited than if no such data were kept. Further, entry call blocks could not be shared among different entries. The decision for this amount of storage would be entirely determined by the run-time system. If only interrupts are handled, the run-time system can optimize its storage to hold only relevant interrupt information. The program can then provide its own buffering for as much or as little data as it desires.

There are cases, such as in radar systems, where the existence of new data immediately invalidates any need to keep old data. Such requirements could not be added easily by the user programmer to the run-time system design. In the case of the console terminal driver, 264 characters of data are buffered. As long as the run-time system will pass on the interrupts faithfully, the package can make claims about the amount of buffering available.

PDP-11 device registers are referenced by reading and writing into designated memory locations. It might seem reasonable to obtain the data in these registers by declaring a variable with a representation specification that maps it to the specified location. The decision to use the `Low_Level_IO` package is motivated by the desire to keep machine dependencies as local as possible. If directly-mapped variables were used, the semantics of the assignment statement and type system would have to define explicitly what is meant when a variable of a certain type is written or read. It is inappropriate to make these decisions just to provide input/output capabilities. For each device that a system supports, it is reasonable and practical to specify exactly the subprograms of `Low_Level_IO`. Furthermore, as long as these semantics can be guaranteed among different implementations of a machine's architecture, the program can be moved from machine to machine.

# 5. Table Creation and Table Searching

## 5.1 Description

In this section we present an example that uses generic packages to provide routines to search a table and to retrieve the items stored there. The table, and auxiliary data objects created to help the search, are encapsulated within a package created by instantiating a generic package. They are not accessible except by the routines exported from that package. A program may have several tables which it can search; a package must be instantiated for each table, and the routines which are exported act specifically upon the table within the corresponding package.

The package has been designed so that the table size is fixed once it has been instantiated. Table look-up returns all items whose keys "match" the given key. A match occurs if the given key is a leading substring of a key associated with an entry. A facility such as this might be used, for example, to store the reserved words for a compiler, or the list of commands available to a command interpreter.

Each entry in the table comprises two parts: a key, implemented as a value of type `Text`, provided by the library package `Text_Handler` ( [8], section 7.6), with which the entry will be retrieved, and an item whose type is defined by the user. The generic package `Symbol_Table_Package_Generator` is instantiated with the item type and the maximum length of a key as parameters, and it yields, among other things, the generic package `Symbol_Table`. Instantiation of this latter generic package with the table as a parameter causes an auxiliary object to be created to allow fast look-up in the table, and yields functions to search and retrieve entries in the table.

The types which are provided are:

`Entry_Type` The type of each entry in the table.

`Table_Type` The type of the table — an array of `Entry_Type` values.

`Table_Pointer` This type is private to the package. Values of this type are returned when the table is searched, and they are used to retrieve the entries.

`Entry_Collection`
An array of `Table_Pointer` values.

The functions which are yielded are:

`Make_Entry` This takes a `String` and an `Item_Type` and returns a record of type `Entry_Type`.

| | |
|---|---|
| `Search` | This takes a `String` as a parameter. It searches the table and returns an `Entry_Collection` whose elements indicate those entries with keys matching the given key. If no keys match, the `Entry_Collection` returned is a null array. If several keys match, the `Entry_Collection` value has more than one element, and the `Table_Pointers` are sorted by ascending key values. |
| `GetKey` | This takes a `Table_Pointer` value and retrieves the key of the element which is indicated. |
| `GetItem` | This takes a `Table_Pointer` value and retrieves the item of the element which is indicated. |

The exception `Invalid_Table_Pointer` will be raised if `GetItem` or `GetKey` is passed a `Table_Pointer` value which is out of bounds. This might occur if an uninitialized `Table_Pointer` variable were used as a parameter. The exception `MaxSize_Error` will be raised if the size specified for the keys, via the generic parameter `MaxSize`, is too large.

## 5.2 Implementation

The table of entries passed in as a parameter is not modified by the program. Instead an auxiliary array of pointers is created and this array is sorted during the elaboration of the package, by using a heapsort [11]. Heapsort is preferred over quicksort because it has better behavior if the input table is nearly sorted.

The `Entry_Collection` for the entries with matching keys is found by the following strategy. First a binary search is performed using routine `Find`. This operates by probing the midpoint of a segment of the table, delimited by `L` and `U`, in order to find a match. One of three conditions holds after returning from the call. If `L` is greater than `U` then no element matches the given key. If `L` is equal to `U` then only one element matches. If `L` is less than `U` then the element pointed to by `P` matches, but there may be other keys between `L` and `U` which might also match. To check this, further probes are performed in the lower region delimited by `L` and `P-1`, and in the upper region delimited by `P+1` and `U`, using repeated binary searches. The searches are repeated until a call on `Find` fails to yield a `P` pointing to a matching key. The value of `P` yielded by the previous call on `Find` is the smallest or largest entry in the table that matches the given key.

## 5.3 Program Text

```
------------------------------------------------------------------
--
-- Generic Package Specifications
--
------------------------------------------------------------------


with Text_Handler;    -- [8], section 7.6

generic
   type Item_Type is limited private;
   MaxSize : in Natural;

package Symbol_Table_Package_Generator is

   type Entry_Type(Length : Natural) is private;

   function Make_Entry(S : in String;
                       I : in Item_Type) return Entry_Type(MaxSize);

   type Table_Type is array (Integer range <>) of Entry_Type(MaxSize);

   MaxSize_Error : exception;

   generic
      Table : in Table_Type;

   package Symbol_Table is

      type Table_Pointer is private;
      type Entry_Collection is array (Natural range <>) of Table_Pointer;

      function GetKey(Index : in Table_Pointer) return String;
      function GetItem(Index : in Table_Pointer) return Item_Type;
      function Search(Key : in String) return Entry_Collection;

      pragma Inline(GetKey, GetItem);

      Invalid_Table_Pointer : exception;

   private

      type Table_Pointer is new Integer range Table'FIRST..Table'LAST;

   end Symbol_Table;
```

```
private

   type Entry_Type(Length : Natural) is
      record
         Key : Text_Handler.Text(Length);
         Item : Item_Type;
      end record;

end Symbol_Table_Package_Generator;


------------------------------------------------------------------
--
-- Package Bodies
--
------------------------------------------------------------------

with Text_Handler;

package body Symbol_Table_Package_Generator is

   pragma Inline(GetKey, GetItem);

   function Make_Entry(S : in String;
                       I : in Item_Type) return Entry_Type(MaxSize) is
   begin
      return Entry_Type'(Length => MaxSize,
                         Key => Text_Handler.To_Text(S, MaxSize),
                         Item => I);
   end Make_Entry;

   package body Symbol_Table is

      Ptr : Entry_Collection(1..Table'LENGTH);

      L, R, I, J, Reg : Table_Pointer;

      function GetKey(Index : in Table_Pointer) return String is
         -- Returns the key of an entry
      begin
         return Text_Handler.Value(Table(Integer(Index)).Key);
      exception
         when Constraint_Error => raise Invalid_Table_Pointer;
      end;

      function GetItem(Index : in Table_Pointer) return Item_Type is
         -- Returns the item of an entry
      begin
         return Table(Integer(Index)).Item;
      exception
         when Constraint_Error => raise Invalid_Table_Pointer;
      end GetItem;
```

```
function Search(Key : in String) return Entry_Collection is

   -- Search for entries with matching keys

   L, U, Posn, P, Psave, Garbage : Integer;
   Found : Boolean;

   procedure Find(Key : in String;
                  Lower_Bound, Upper_Bound : in Integer;
                  L, U, I : out Integer;
                  Found : out Boolean) is separate;

begin

-- First probe

   Find(Key, Ptr'FIRST, Ptr'LAST, L, U, Posn, Found);

   if U > L then      -- Not examined all entries between L and U

      P := Posn;              -- Probe lower region
      loop
         Psave := P;
         Find(Key, L, P-1, L, Garbage, P, Found);
         exit when not Found;
      end loop;
      L := Psave;

      P := Posn;              -- Probe upper region
      loop
         Psave := P;
         Find(Key, P+1, U, Garbage, U, P, Found);
         exit when not Found;
      end loop;
      U := Psave;

   end if;

   return Ptr(L..U);

end Search;

begin

-- Initialize ptr

   for Cntr in Ptr'RANGE loop
      Ptr(Cntr) := Table_Pointer(Table'FIRST + Cntr - Ptr'FIRST);
   end loop;
```

```
      -- Heapsort the table indirectly through Ptr

         L := Ptr'LAST / 2 + 1;
         R := Ptr'LAST;

      Outer_Loop:
         loop

            if L > 1 then
               L := L - 1;
               Reg := Ptr(L);
            else
               Reg := Ptr(R);
               Ptr(R) := Ptr(1);
               R := R - 1;
               exit Outer_Loop when R = 1;
            end if;

            J := L;

         Inner_Loop:
            loop
               I := J;
               exit Inner_Loop when J > R/2;
               J := 2 * J;
               if J < R and then GetKey(Ptr(J)) < GetKey(Ptr(J+1)) then
                  J := J + 1;
               end if;
               exit Inner_Loop when GetKey(Reg) >= GetKey(Ptr(J));
               Ptr(I) := Ptr(J);
            end loop Inner_Loop;

            Ptr(I) := Reg;

         end loop Outer_Loop;

         Ptr(1) := Reg;

      end Symbol_Table;

   begin       -- Body of Symbol_Table_Package_Generator

      if MaxSize not in Text_Handler.Index then
         raise MaxSize_Error;
      end if;

   end Symbol_Table_Package_Generator;
```

```
------------------------------------------------------------------
--
-- Body of Find Procedure
--
------------------------------------------------------------------

separate (Symbol_Table_Package_Generator)

procedure Find(Key : in String;
               Lower_Bound, Upper_Bound : in Integer;
               L, U, I : out Integer;
               Found : out Boolean) is
   -- Binary search
begin
   L := Lower_Bound;
   U := Upper_Bound;

Find_Loop:
   while U >= L loop
      I := (L + U) / 2;
      declare
         Key_Of_Probe : constant String := GetKey(Ptr(I));
      begin
         if Key_Of_Probe'LENGTH > Key'LENGTH then
            exit Find_Loop when
                 Key =
                 Key_Of_Probe(Key_Of_Probe'FIRST..
                              Key_Of_Probe'FIRST+Key'LENGTH-1);
         else
            exit Find_Loop when Key = Key_Of_Probe;
         end if;
         if Key < Key_Of_Probe then
            U := I - 1;
         else
            L := I + 1;
         end if;
      end;
   end loop Find_Loop;
   Found := U >= L;
end Find;
```

## 5.4 Discussion

### 5.4.1 Use of The Package

Following is an example of this package used as part of a command scanner.

```
------------------------------------------------------------------
--
-- Instantiation of the generic packages
--
------------------------------------------------------------------

type Actions is (Find, Delete, Insert, Alter, Execute, Quit);

package Actions_Symbol_Table is
   new Symbol_Table_Package_Generator(Actions, 8);

package My_Symbol_Table is
   new Actions_Symbol_Table.Symbol_Table(
      ( Make_Entry("find", Find),
        Make_Entry("delete", Delete),
        Make_Entry("insert", Insert),
        Make_Entry("enter", Insert),
        Make_Entry("alter", Alter),
        Make_Entry("amend", Alter),
        Make_Entry("execute", Execute),
        Make_Entry("quit", Quit),
        Make_Entry("leave", Quit)
   )
);
```

### 5.4.2 Use of Packages

This example illustrates several features of the use of generic packages to effect data encapsulation.

- The table is passed into the generic package `Symbol_Table` as an **in** parameter; thereafter it cannot be manipulated or altered from outside the package. This assures us that the search function, which relies upon the proper ordering of the keys as established by the initial sort, cannot fail no matter how erroneous is the client program using the package.

- The functions `Search`, `GetItem` and `GetKey` which are exported from a package are specific to the package, and hence to the data encapsulated within it. Thus, it is not necessary to supply further parameters to these functions to indicate which of several tables they are to access. In addition, the type of `Table_Pointer` is specific to the instance of the package, and therefore it is not possible to access a table in one package with the pointers from some other package.

### 5.4.3 The Types of the Entries in the Table

In the sorting and searching no use is made of the items in each entry, thus they can be made private to the package; furthermore, by sorting on an auxiliary array of pointers, rather than on the array of entries itself, the package does not require to take copies, and the items can be **limited private**. This has two advantages. First, the item type can be of any size and complexity, without slowing down the sorting, and second. the items can be of any type, including task types.

Choosing an appropriate type for the keys causes some problems. There appear to be three basic choices, none of them entirely satisfactory. In addition to the choice illustrated in the example, we have the following.

- We can implement the key as a `String`. The type of the entry then becomes:

```
type Entry_Type(Length : Natural) is
   record
      Key : String(1..Length);
      Item : Item_Type;
   end record;
```

  and we must provide a parameter in the first generic instantiation to specify the maximum size of the strings, as in the current example. The aggregate which is then passed as a parameter to the second generic instantiation must contain string literals padded out with the appropriate number of fill characters. This is tedious.

- We could implement the key as an **access** `String`. The declaration for the `Entry_Type` becomes:

```
type Pointer_To_String is access String;

type Entry_Type is
   record
      Key : Pointer_To_String;
      Item : Item_Type;
   end record;

type Table_Type is array (Integer range <>) of Entry_Type;
```

  This has the virtue that strings of any size may be used as keys, and no parameter is needed to specify the length (thus the outer generic package will only require the type parameter). Unfortunately, there is a serious flaw in this implementation. Since it is possible for the client to keep an access path to the entries in the table externally to the package, it will be possible to alter the keys. Thus an erroneous client program could alter the behaviour of the package in unpredictable ways.

One additional aspect of these three versions should be mentioned. Only in the case where the key is a `String` is it possible to create the aggregate at compile time. In the other two cases it will be necessary to create the aggregate at run time, and thus it is likely that two copies of the strings and the items will exist in the memory of the computer. This could preclude the use of the `Text_Handler` and the **access** `String` versions if the table is large.

### 5.4.4 Use of a Private Type for the Pointers to the Table

In order to prevent the user of the package from performing operations on the pointers into the table, and possibly converting a legal pointer into an illegal one, `Table_Pointer` is made private. There is still the possibility that uninitialized `Table_Pointer` values will be passed by the client to the routines `GetKey` and `GetItem`. The exception `Invalid_Table_Pointer` is raised if the undefined value

happens to violate the range constraints. We could alter the implementation to ensure that all `Table_Pointer` values are initialized, by making the type be a record type, and providing initialization. We chose not to do it in this example, to avoid further complexity.

### 5.4.5 Nesting a Generic Package Within a Generic Package

If a package is to have the features we desire, we need at least two generic parameters: the type of the items, and the table which is to be searched. However, the language does not allow us to pass both these parameters at the same time to the same generic package, since the type of the table must be available before we can pass the table as a parameter. Thus it is necessary to have two nested generic packages: the outer takes the type parameter, and produces among other things the type of the table and the inner generic package. The inner package is then instantiated with the table as parameter.

### 5.4.6 String Comparisons

Several interesting points arise as a result of the need to perform string comparisons within the body of `Find`. Several operations need to be performed on the key which is retrieved from the table when it is being probed. Rather than retrieve the key repeatedly, we have decided to take a copy of it and perform the operations on the copy. Since we do not know the size of the string before we retrieve it, we must either take a copy in a new access value, or in a declaration in an inner block. We choose the latter to avoid the need for garbage collection.

### 5.4.7 Use of Integers in Find

The variables `Lower_Bound`, `Upper_Bound`, `L`, `U` and `I` are all declared as `Integer`, rather than as integers constrained by the bounds of `Ptr` even though they are used as indexers for `Ptr`, because the algorithm allows them to range from `Ptr'FIRST-1` to `Ptr'LAST+1`. More bounds checking will occur in the body of `Find` and `Search` than is necessary, but there is no simple way to avoid this. We considered placing range constraints on the parameters to `Find` (for example, `Lower_Bound`, `Upper_Bound` and `P` can be constrained to the range `1..Table_Length`, and `L` and `U` to the range `0..Table_Length+1`). While this could help locate errors more accurately, it might also add sufficiently to the complexity of the source that additional errors would be introduced by the programmer.

# 6. Solution of Laplace's Equation with Several Ada Tasks

## 6.1 Description

In this section we present the solution of a numerical problem using several Ada tasks. The solution is designed for a multiprocessor computer system such as C.mmp [18] or Cm* [17] in which the processors have access to a shared memory.

Our problem involves the Laplace partial differential equation:[6]

$$\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} = 0$$

on a rectangular region D. We are given as boundary conditions the values of U on the edges of D. We are to approximate a solution to the Laplace equation by finding the values of U at each point of a mesh of points in the interior of D. The mesh is an m-by-n rectangular array of points, (i,j), i = 1,...,m and j = 1,...,n.

The Laplace equation can be approximated by the difference equations:

$$U_{i+1,j} + U_{i,j+1} + U_{i-1,j} + U_{i,j-1} - 4U_{i,j} = 0$$

The value of U at a point (i,j) is the average of the values of U at the four neighboring points. We have one such equation for each interior point (i,j). Taken together, the equations give us a linear system of simultaneous equations to solve.

Let us number the points (i,j) in row major order, which gives the numbering 1,...,mn. Let the vector **u** be:

$$\mathbf{u} = (U_1, U_2, \ldots, U_{mn})$$

Our linear system may then be written as Au = 0. The coefficient matrix, A, is a symmetric band matrix with

- All diagonal elements equal to -4; and
- For non-diagonal elements, $a_{p,q} = 1$ if point p is a neighbor of point q, and 0 otherwise.

The method we will use to solve this system is a parallel version of Gauss-Seidel iteration with N Ada

[6] In this presentation, we follow closely the treatment of Dahlquist and Björck [3], sections 5.6 and 8.6.3

tasks. Each task is assigned to work on a portion of the system. Within each portion, the method reduces to the normal single-process Gauss-Seidel iteration. The parallel Gauss-Seidel method was studied by G. Baudet, who implemented it on C.mmp [1, 2].

The Gauss-Seidel iteration formula for deriving the $(n+1)^{st}$ approximation of $U_{i,j}$ from the $n^{th}$ is:

$$U_{i,j}^{(n+1)} = (U_{i-1,j}^{(n+1)} + U_{i,j-1}^{(n+1)} + U_{i+1,j}^{(n)} + U_{i,j+1}^{(n)}) / 4$$

A modification to Gauss-Seidel which accelerates its convergence is the *successive overrelaxation* method. Its iteration formula is:

$$U_{i,j}^{(n+1)} = U_{i,j}^{(n)} + Omega*(U_{i-1,j}^{(n+1)} + U_{i,j-1}^{(n+1)} + U_{i+1,j}^{(n)} + U_{i,j+1}^{(n)} - 4U_{i,j}^{(n)}) / 4$$

Here, Omega is the parameter to the successive overrelaxation method.[8] When Omega is equal to 1, the formula reduces to the original Gauss-Seidel formula.

## 6.2 Implementation

The procedure we provide, `Parallel_Relaxation`, takes the matrix `U` as a parameter. It is assumed that the outside edge of `U` has been initialized by the caller with the desired values. The procedure computes the values of `U` for the interior (i.e., non-edge) points.

In our implementation, the coefficient matrix A is not present explicitly, rather, the coefficients are simply reflected in the iteration formula for successive overrelaxation.

The procedure `Parallel_Relaxation` also takes a parameter, `NumberOfRegions`, which indicates the number of Ada tasks to create.

Each Ada task works on a region of the matrix `U`. Each region is itself a rectangular matrix. A rectangular grid of such regions is laid out over the interior of the matrix `U`. One Ada task is dedicated to each region.

Testing for the convergence of the system as a whole must be done with care. To have convergence, each task must believe that its own region has converged, and furthermore, all tasks must hold this belief "simultaneously"; that is, they must reach a unanimous consensus on whether or not they all have converged. Observe that a task may decide prematurely that its region has converged, even though a change may soon occur in its region because of a change in one of its neighbors.

---

[7] Besides the treatment of Gauss-Seidel iteration in Dahlquist and Björck [3] and other texts on numerical methods, the reader may wish to refer to McCracken [13], who gives a single-processor treatment of our problem as part of Case Study 11B.

[8] Choosing an appropriate value for Omega is discussed in Dahlquist and Björck [3], sections 5.6 and 8.6.3.

For each region task, there is a coordinator task which is used to assist in the communication between the region tasks. The communication consists of one region task, A, advising a neighboring region task, B, that A has not yet converged within its own region. Since A has not yet converged, its neighbor B could not yet have really converged either, in spite of what B might believe locally. Thus, we can think of this message from A to B as a command to B to keep on going. To avoid deadlock and contention problems which would result if the region tasks did entry calls on each other directly, the message is actually sent by having A do an entry call on B's coordinator task.

The consensus among the region tasks is achieved by passing messages to each other's coordinators, and by having a global counter of the number of unfinished region tasks. When a region task believes that it is done, it decrements the counter. When a coordinator task receives a message that its region could not really be done, it increments the counter. When the counter reaches zero, we know that all region tasks thought they were finished.

Each region task repeatedly executes the following actions. First, it computes one complete new set of values for the points in its region. If the region has not yet converged locally, the task advises its neighbors' coordinators to keep on going. If the region has converged locally, then the task decrements the global counter of unfinished tasks. If the counter is zero, we are done; the task must clean up the system of tasks (described below), and we may then return from the `Parallel_Relaxation` procedure. If the counter is non-zero, the region task puts itself to sleep by calling an entry of its coordinator, `Wait`. The coordinator will wake it up by accepting the `Wait` entry call when a keep-on-going message is received; if such a message had already been received, the coordinator accepts the `Wait` entry call immediately. The coordinator merges multiple keep-on-going messages that have been received since the last `Wait` entry call into one message. Multiple keep-on-going messages will thus cause only one acceptance of the `Wait` entry.

When a region task discovers in the test above that the counter of unfinished region tasks is zero, it must arrange to wake any other region tasks that are sleeping on the `Wait` entries of their respective coordinators. This must be done because all tasks which are local to a block in Ada must terminate before we may exit the block. This waking up is handled by having another entry into the coordinator tasks, the `Finish` entry. After a `Finish` entry call is received, a coordinator will immediately accept any `Wait` entry calls. After a region task returns from the `Wait` entry call, it looks again to see if the global counter has gone to zero. If it has, the region task concludes that we are done and exits.

Finally, some systems of equations will not converge. As a practical matter, it is important to provide an upper bound on the number of iterations. We do this by having another parameter to `Parallel_Relaxation`, named `MaxIterations`. If this maximum is exceeded, then we set another parameter, `DidNotConverge`, to `True` and return.

## 6.3 Program Text

### 6.3.1 A Protected Counter Task Type

Here we present a task type that provides a protected counter with operations for incrementing and decrementing. It is protected in that the increment and decrement operations are indivisible. This indivisibility is acheived by performing the operations within accept bodies.

We surround the task with a package, so that it may be separately compiled and placed in a library.

```
package Protected_Counter_Package is

   task type Protected_Counter is
      entry Initialize(Z : in Integer);
      entry Incr(Z : in Integer := 1);
      entry Decr(Z : in Integer := 1);
      entry Read(Z : out Integer);
   end Protected_Counter;

end Protected_Counter_Package;

package body Protected_Counter_Package is

   task body Protected_Counter is
      Counter: Integer;
   begin
      accept Initialize(Z : in Integer) do
         Counter := Z;
      end;

      loop
         select
            accept Incr(Z : in Integer := 1) do
               Counter := Counter + Z;
            end;
         or
            accept Decr(Z : in Integer := 1) do
               Counter := Counter - Z;
            end;
         or
            accept Read(Z : out Integer) do
               Z := Counter;
            end;
         or
            terminate;
         end select;
      end loop;
   end Protected_Counter;

end Protected_Counter_Package;
```

### 6.3.2 Parallel_Relaxation Procedure

```
generic
   type Real is digits <>;
   type RealMatrix is array(Integer range <>, Integer range <>)
            of Real;

procedure Parallel_Relaxation(U : in out RealMatrix;
                              MaxErr : in Real;
                              MaxIterations : in Natural;
                              NumberOfRegions : in Natural;
                              DidNotConverge : out Boolean;
                              Omega : in Real := 1.0);

-------------------------------------------------------------------------

with Protected_Counter_Package, Math_Lib, Integer_MaxMin_Lib;
use Integer_MaxMin_Lib;

procedure Parallel_Relaxation(U : in out RealMatrix;
                              MaxErr : in Real;
                              MaxIterations : in Natural;
                              NumberOfRegions : in Natural;
                              DidNotConverge : out Boolean;
                              Omega : in Real := 1.0) is

-- MaxErr determines when we have converged (i.e., we have converged
--   when the change in the value of all points is <= MaxErr).
-- MaxIterations is a limit on how many iterations to perform.  If
--   we perform this many iterations without converging, then we set
--   DidNotConverge to True and return.
-- NumberOfRegions is the maximum number of Ada tasks to use.
-- Omega is the acceleration parameter to the successive overrelaxation
--   formula.

   RowRegions, ColRegions : Integer;
   NumRegions : Integer;
   RowsPerRegion, ColsPerRegion : Integer;
   RowLo : constant Integer := U'FIRST(1);
   RowHi : constant Integer := U'LAST(1);
   ColLo : constant Integer := U'FIRST(2);
   ColHi : constant Integer := U'LAST(2);
   subtype InteriorRows is Integer range RowLo+1..RowHi-1;
   subtype InteriorCols is Integer range ColLo+1..ColHi-1;
   LenInteriorRows : constant Integer := (InteriorRows'LAST -
                                          InteriorRows'FIRST) + 1;

      --   There is no 'LENGTH attribute for discrete subtypes.

   LenInteriorCols : constant Integer := (InteriorCols'LAST -
                                          InteriorCols'FIRST) + 1;

   package Real_Math_Lib is new Math_Lib(Real);
   use Real_Math_Lib;
```

```
begin

   --   Initially, assume that the system will converge:

   DidNotConverge := False;

   --   See if the matrix U has any interior points. If not, return:

   if U'LENGTH(1) <= 2  or  U'LENGTH(2) <= 2 then
      return;
   end if;

   --   We zero the interior points initially:

   for I in InteriorRows loop
      for J in InteriorCols loop
         U(I, J) := 0.0;
      end loop;
   end loop;

   --   Determine the layout of the regions on the matrix.
   -- Each region is itself a rectangular sub-matrix.  We lay down a
   -- rectangular array of regions on top of the matrix.  The
   -- array is RowRegions by ColRegions:

   RowRegions := Floor(Sqrt(Real(NumberOfRegions)));
   ColRegions := NumberOfRegions / RowRegions;
   NumRegions := RowRegions * ColRegions;

      --  E.g., for NumberOfRegions = 33, we get 5, 6, and 30.
      --  Only the 30 regions are actually used.

   --   Each region is a rectangle of RowsPerRegion rows by
   -- ColsPerRegion columns.  The regions at the right edge
   -- and the bottom may be smaller however:

   RowsPerRegion := (LenInteriorRows + (RowRegions - 1)) / RowRegions;
   ColsPerRegion := (LenInteriorCols + (ColRegions - 1)) / ColRegions;

   --   Now that we know how many tasks we actually want, enter
   -- an inner block to declare them:

 ParRelax_Inner_Block:
   declare
      task type Region_Task is
         entry SetParameter(SetMyRowregion,
                            SetMyColRegion : in Integer);
      end Region_Task;

      task type Coordinator_Task is
         entry Wait;
         entry KeepOnGoing;
         entry Finish;
      end Coordinator_Task;
```

```
      Regions : array(1..RowRegions, 1..ColRegions) of Region_Task;
      Coordinators : array(1..RowRegions, 1..ColRegions) of
                        Coordinator_Task;
      Unfinished_Counter : Protected_Counter_Package.Protected_Counter;

      task body Coordinator_Task is
         Had_KeepOnGoing: Boolean := False;
         Had_Finish: Boolean := False;
      begin
         loop
            select
               accept KeepOnGoing do
                  if not Had_KeepOnGoing then
                     Unfinished_Counter.Incr;
                     Had_KeepOnGoing := True;
                  end if;
               end KeepOnGoing;
            or
               when had_KeepOnGoing or Had_Finish =>
                  accept Wait do Had_KeepOnGoing := False; end Wait;
            or
               accept Finish do Had_Finish := True; end Finish;
            or
               terminate;
            end select;
         end loop;
      end Coordinator_Task;

      procedure All_Finish is separate;

      task body Region_Body is separate;

   begin    --   The statements of ParRelax_Inner_Block

      --   The count of unfinished tasks is initially all of the
      -- region tasks:

      Unfinished_Counter.Initialize(NumRegions);

      --   Set the parameters of the regions tasks:

      for I in Regions'RANGE(1) loop
         for J in Regions'RANGE(2) loop
            Regions(I,J).SetParameter(I,J);
         end loop;
      end loop;

   -- We now simply wait at the end of the
   -- declaring block for the tasks to terminate.

   end ParRelax_Inner_Block;

end Parallel_Relaxation;
```

```
------------------------------------------------------------------------
--
-- Body of All_Finish Procedure
--
------------------------------------------------------------------------

separate (Parallel_Relaxation)

procedure All_Finish is
   -- Calls the Finish entries of all the coordinators.
begin
   for Rreg in Coordinators'RANGE(1) loop
      for Creg in Coordinators'RANGE(2) loop
         Coordinators(Rreg, Creg).Finish;
      end loop;
   end loop;
end All_Finish;

------------------------------------------------------------------------
--
-- Body of Region_Task
--
------------------------------------------------------------------------

separate (Parallel_Relexation)

task body Region_Task is
   MyRowRegion, MyColRegion : Integer;
begin
   --   Task starts by finding out what region it has been
   -- assigned:
   accept SetParameter(SetMyRowRegion, SetMyColRegion : in Integer) do
      MyRowRegion := SetMyRowRegion;
      MyColRegion := SetMyColRegion;
   end;

Region_Inner_Block:
   declare
      MyDone : Boolean;
      New_Value : Real;
      CurCount : Integer;

      --   Compute the boundaries of my region.  These are
      -- the points that will be computed by me:

      MyRowLo : constant Integer := (MyRowRegion - 1) *
                  RowsPerRegion + InteriorRows'FIRST;
      MyColLo : constant Integer := (MyColRegion - 1) *
                  ColsPerRegion + InteriorCols'FIRST;
```

```
      --   If we're at the bottom edge, then MyRowHi should
      -- not exceed InteriorRows'LAST:

      MyRowHi : constant Integer := Min(InteriorRows'LAST,
                   MyRowLo + RowsPerRegion - 1);

      --   Likewise, if we're at the right edge...:

      MyColHi: constant Integer := Min(InteriorCols'LAST,
                   MyColLo + ColsPerRegion - 1);

   begin

   ItersLoop:
      for Iters in 1..MaxIterations loop
         MyDone := True;
         --   Compute a new value for each point in my
         -- region:
         for I in MyRowLo .. MyRowHi loop
            for J in MyColLo .. MyColHi loop
               New_Value := U(I,J) + Omega *
(U(I-1,J) + U(I,J-1) + U(I+1,J) + U(I,J+1) - 4.0*U(I,J)) / 4.0;
               if Abs(New_Value - U(I,J)) >= MaxErr
                  then MyDone := False;
               end if;
               U(i,j) := New_Value;
            end loop; -- over cols
         end loop;  --over rows

         if not MyDone then   -- Tell my neighbors to keep on going:
            if MyRowRegion /= 1 then
               Coordinators(MyRowRegion-1,
                            MyColRegion).KeepOnGoing;
            end if;

            if MyRowRegion /= RowRegions then
               Coordinators(MyRowRegion+1,
                            MyColRegion).KeepOnGoing;
            end if;

            if MyColRegion /= 1 then
               Coordinators(MyRowRegion,
                            MyColRegion-1).KeepOnGoing;
            end if;

            if MyColRegion /= ColRegions then
               Coordinators(MyRowRegion,
                            MyColRegion+1).KeepOnGoing;
            end if;
```

```
            else
               Unfinished_Counter.Decr;
               Unfinished_Counter.Read(CurCount);
               if CurCount = 0 then
                  --   We're all done.  Wake up
                  -- everybody who's sleeping:
                  All_Finish;
                  goto EndOfTask;
               else
                  --   Wait to hear of some change
                  -- from my neighbors, or
                  -- for all tasks to finish:

                  Coordinators(MyRowRegion,
                               MyColRegion).Wait;

                  --   We've been woken up.  See whether
                  -- because everybody's finished,
                  -- or because of a KeepOnGoing
                  -- message:

                  Unfinished_Counter.Read(CurCount);
                  if CurCount = 0 then
                     goto EndOfTask;
                  end if;
               end if;
            end if;

            --   See if some other task has taken too many
            -- iterations already and if so stop iterating:

            exit ItersLoop when DidNotConverge;
         end loop ItersLoop;

         --   Here iff some task (either my task or some
         -- other task) has taken too many iterations:

         DidNotConverge := True;
         All_Finish;
      end Region_Inner_Block;

   <<EndOfTask>>
      null;
   end Region_Task;
```

## 6.4 Discussion

### 6.4.1 Use of Shared Variables

In our solution, we have used the array U as a shared variable. All the region tasks access it directly without any additional protocol. The effects of such simultaneous access to a shared variable are not specified in the Ada language, and will vary from implementation to implementation (see [8], section 9.11). In some systems, a floating point variable may occupy several bytes of storage, and simultaneous reads and writes may produce an interleaving of bytes: the read may receive some bytes from the value of the variable before the write and some bytes from the value after the write. Thus, the read may receive a value that is neither the previous value of the variable nor the new value. For our solution to be reasonable, we require that the operations of reading and writing the shared variables be indivisible with respect to each other. Note that we do *not* require an indivisible read-modify-write operation.

### 6.4.2 Updates of Shared Variables From Registers

Another potential difficulty with shared variables is the problem of the compiler keeping such variables in local registers (see [8], section 9.11). A compiler is required to store these back to the shared variables only at those points where tasks synchronize, e.g., via rendezvous. Fortunately, in our solution we have sufficient synchronization between the region and coordinator tasks for the shared matrix U to be updated when needed. It is thus not necessary for us to employ the predefined generic procedure `Shared_Variable_Update`.

### 6.4.3 Generics and Generic Instantiation

The procedure `Parallel_Relaxation` is a generic procedure with two generic formal parameters: the floating point data type and a two-dimensional array type of this floating point type. This allows a user to instantiate the procedure with any floating point data type he has declared. He can, of course, instantiate it several times, for example, once with a floating point type `HisShort` which is **digits** 6 and once with a floating point type `HisLong` which is **digits** 12.

Within the body of the procedure, we need some common mathematical functions on the generic formal type `Real` (e.g., `Floor` and `Sqrt`). Since these are not pre-defined subprograms in the package `Standard`, we have to arrange for their definition. We assume the existence of a generic library package, `Math_Lib`, which contains, amongst other facilities, the functions we need. Furthermore, we assume that `Math_Lib` has one generic formal parameter which is the floating point type for which its facilities are to be provided. We obtain the routines we need by instantiating `Math_Lib` with our generic parameter `Real`.

For a subprogram defined in the package `Standard` as a subprogram for the floating point types (e.g., `Abs` or "+"), the type `Real` will automatically inherit the subprogram, via the Ada derived type mechanism.

### 6.4.4 Scheduling of Ada Tasks Onto Processors

Our procedure is intended for use with an implementation of Ada on a multiprocessor computer system, which will have some number P of physical processors. Our N Ada tasks will be scheduled onto these processors by the underlying Ada system and/or by the operating system. Of course, we intend that the system will devote more than one processor to our program. However, there is no way of specifying or guaranteeing this within Ada. Suppose that P is less than N, or that the system chooses to give our program less than N dedicated processors, because, for example the system is multi-programmed or time-shared between several independent user programs. Our procedure will nevertheless execute correctly no matter how many processors are running the Ada tasks and no matter their relative speeds.

Consider the case of having one processor executing our program. Suppose that initially this processor is running some particular region task, A. Since none of this region's neighbors are changing, the region will eventually converge locally. The task A will find the global variable `Unfinished_Counter` non-zero and will block on the call to its coordinator's `Wait` entry. Since A is therefore no longer eligible, the processor will find some other eligible Ada task to run. As they block and unblock during rendezvous, our Ada tasks will multiplex themselves onto the single processor.

Observe that this ability to execute correctly on a single processor is a property of our program, and not a general property of any Ada program with multiple Ada tasks. In particular, if a running Ada task never engages in any rendezvous, there is no obligation of the underlying Ada system to ever de-schedule it and let another Ada task execute. Restating, an Ada implementation may or may not choose to do time-slicing, at the discretion of the implementors.

# References

[1] Gerard M. Baudet.
Asynchronous Iterative Methods for Multiprocessors.
*Journal of the ACM* 25(2):226-244, April, 1978.

[2] Gerard M. Baudet.
*The Design and Analysis of Algorithms for Asynchronous Multiprocessors.*
PhD thesis, Dept. of Computer Science, Carnegie-Mellon University, April, 1978.

[3] Germund Dahlquist and Ake Björck.
*Numerical Methods.*
Prentice Hall, Englewood Cliffs, New Jersey, 1974.

[4] Digital Equipment Corporation.
*PDP-11 Peripherals Handbook.*
Digital Equipment Corporation, Maynard, Massachusetts, 1976.

[5] Digital Equipment Corporation.
*PDP-11 04/34/45/55 Processor Handbook.*
Digital Equipment Corporation, Maynard, Massachusetts, 1976.

[6] Paul Hilfinger, Gary Feldman, Robert Fitzgerald, Izumi Kimura, Ralph L. London, KVS Prasad, VR Prasad, Jonathan Rosenberg, Mary Shaw, Wm. A. Wulf.
*(Preliminary) An Informal Definition of Alphard.*
Research Report CMU-CS-78-105, Carnegie-Mellon University, Computer Science Department, February, 1978.

[7] Ellis Horowitz and Sartaj Sahni.
*Fundamentals of Computer Algorithms.*
Computer Science Press, Inc., 1978.

[8] J.D. Ichbiah, B. Krieg-Brueckner, B.A. Wichmann, H.F. Ledgard, J.C. Heliard, J.R. Abrial, J.G.P. Barnes, M. Woodger, O. Roubine, P.N. Hilfinger, R. Firth.
*Reference Manual for the Ada Programming Language*
The Revised Reference Manual, July 1980 edition, Honeywell, Inc., and Cii-Honeywell Bull, 1980.

[9] Donald E. Knuth.
*The Art of Computer Programming.* Volume 1: *Fundamental Algorithms.*
Addison-Wesley, 1973.

[10] Donald E. Knuth.
*The Art of Computer Programming.* Volume 2: *Seminumerical Algorithms.*
Addison-Wesley, 1969.

[11] Donald E. Knuth.
*The Art of Computer Programming.* Volume 3: *Sorting and Searching.*
Addison-Wesley, 1973.

[12] Barbara Liskov, Alan Snyder, Russell Atkinson and Craig Schaffert.
Abstraction Mechanisms in CLU.
*Communications of the ACM* 20(8), August, 1977.

[13] Daniel D. McCracken.
*A Guide to Fortran IV Programming, 2nd Edition.*
John Wiley and Sons, New York, 1972.

[14] A. Newell, F. Tonge, E. A. Feigenbaum, B. F. Green, Jr. and G. H. Mealy.
*Information Processing Language-V Manual.*
Prentice Hall, 1964.

[15] Mary Shaw, William A. Wulf and Ralph L. London.
Abstraction and Verification in Alphard: Defining and Specifying Iteration and Generators.
*Communications of the ACM* 20(8), August, 1977.

[16] M.S. Sherman and M. Borkan.
A Flexible Semantic Analyzer for Ada.
In *Symposium on the Ada Programming Language*. ACM, Boston, December, 1980.

[17] R.J. Swan, S.H. Fuller, D.P. Siewiorek.
Cm*: A Modular, Multi-Microprocessor.
In *Proc. 1977 National Computer Conference*. American Federation of Information Processing Societies, 1977.

[18] W.A. Wulf and C.G. Bell.
C.mmp - A Multi Mini Processor.
In *Proc. 1972 Fall Joint Computer Conference*. American Federation of Information Processing Societies, 1972.

[19] William A. Wulf, Mary Shaw, Paul N. Hilfinger, and Larry Flon.
*Fundamental Structures of Computer Science.*
Addison-Wesley, 1981.