21.5a Double dispatching

Another form of passive iterator uses interfaces and dispatching (this is perhaps an obsolescent technique since anonymous access to subprogram parameters were introduced in Ada 2005, however, it has a certain symmetry which is interesting). Assuming the parent package Lists as described in Section 21.5, we can write

```
package Lists.Iterators is
   type Iterator is interface;
   procedure Iterate(IC: in Iterator'Class; L: in List);
   procedure Action(It: in out Iterator; C: in out Colour) is abstract;
end;

package body Lists.Iterators is

   procedure Iterate(IC: in Iterator'Class; L: in List) is
      This: access Cell := L;
   begin
      while This /= null loop
         Action(IC, This.C);                    -- dispatches
                                                -- or IC.Action(This.C);
         This := This.Next;
      end loop;
   end Iterate;

end Lists.Iterators;
```

and the subprograms to perform the specific operations can now be declared in a package as follows

```
package Lists.Iterators.Ops is
   function Count(L: List) return Natural;
   procedure Green_To_Red(L: in List);
end;

package body Lists.Iterators.Ops is

   function Count(L: List) return Natural is
      type Count_Iterator is new Iterator with null record;
      Result: Natural := 0;

      procedure Action(It: in out Count_Iterator; C: in out Colour) is
      begin
         Result := Result + 1;
      end Action;

      It: Count_Iterator;
   begin
      Iterate(It, L);                           -- or It.Iterate(L);
      return Result;
   end Count;
```

```
   procedure Green_To_Red(L: in List) is
      type GTR_Iterator is new Iterator with null record;

      procedure Action(It: in out GTR_Iterator; C: in out Colour) is
      begin
         if C = Green then C := Red; end if;
      end Action;

      It: GTR_Iterator;
   begin
      Iterate(It, L);                          -- or It.Iterate(L);
   end Green_To_Red;

end Lists.Iterators.Ops;
```

The workings should be noted carefully. The subprograms call Iterate and pass a particular iterator as parameter. The tag of this identifies the associated Action which is then called from within the loop of Iterate. Each iterator is a null extension of the abstract type Iterator and acts as a call-back handle.

Observe that the extensions are not within a package specification and so no new primitive operations can be added but nevertheless the existing operation Action can be overridden. Moreover, the extensions are inside the subprograms such as Count and so are at an inner level – this was not permitted in Ada 95 but is permitted in Ada 2005 and Ada 2012.

It is instructive to note the similarity between the dispatching procedures Action and the procedures such as Count_Action when using the access to subprogram form of passive iterator described in Section 21.5. In both cases variables such as Result are global and in fact the text of the procedures is the same.

As another example, if we wanted to change all balls of a given colour to another given colour then we would write

```
procedure Change_Colour(L: in List; From, To: in Colour) is
   type Change_Iterator is new Iterator with null record;

   procedure Action(It: in out Change_Iterator; C: in out Colour) is
   begin
      if C = From then
         C := To;
      end if;
   end Action;

   It: Change_Iterator;
begin
   Iterate(It, L);
end Change_Colour;
```

Clearly the same technique can be used with any data structure and we leave the reader to explore how it might be generalized as an exercise.

Exercise

1 Consider how to generalize the passive iterator approach described above to work on any structure by declaring a package containing interfaces Structure and Iterator plus primitive operations Iterate and Action. Apply the generalization to a binary tree by declaring a type Tree as an extension of Structure. Then declare a function that counts the number of balls of a given colour in any structure and apply it to determine how many Green balls are in a tree. Hint: use double dispatching.

Answer

1

```
package Iterators is
   type Structure is interface;
   type Iterator is interface;
   procedure Iterate(S: in Structure;
                     IC: in Iterator'Class) is abstract;
   procedure Action(It: in out Iterator;
                    C: in out Colour) is abstract;
end;

package Trees is
   type Tree is new Structure with private;
   ...
   procedure Iterate(T: in Tree; IC: in Iterator'Class);
private
   type Node;
   type Node_Ptr is access Node;

   type Node is
      record
         Left, Right: Node_Ptr;
         C: Colour;
      end record;
   type Tree is new Structure with
      record
         Root: Node_Ptr;
      end record;
end;

package body Trees is
   ...
   procedure Iterate(T: in Tree; IC: in Iterator'Class) is
      procedure Inner(N: in Node_Ptr) is
      begin
         if N /= null then
            Action(IC, N.C);        -- dispatches on IC
            Inner(N.Left);
            Inner(N.Right);
         end if;
      end Inner;
   begin
      Inner(T.Root);
   end Iterate;
end Trees;
```

The package Iterators has no body. It serves just as a means of establishing the interfaces and their primitive operations; Iterate is a primitive operation of Structure and Action is a primitive operation of Iterator. The type Trees is then extended from Structure but note that an extra level is required in order to hold the pointer to the root of the tree. A consequence of this is that the recursive walk over the tree has to be done by a local procedure Inner within Iterate. We now declare the general counting function thus

```
function Count(S: Structure'Class; C: Colour)
                                        return Natural is
   type Count_Iterator is new Iterator with
                                           null record;
   Result: Natural := 0;

   procedure Action(It: in out Count_Iterator;
                    C: in out Colour) is
   begin
      if C = Count.C then
         Result := Result + 1;
      end if;
   end Action;

   It: Count_Iterator;
begin
   Iterate(S, It);                  -- dispatch on S
   return Result;
end Count;

Oak: Tree;                          -- declare some tree
...                                 -- build the tree
N := Count(Oak, Green);
```

The final statement counts how many nodes have the colour Green in the Tree called Oak. Note the double dispatching. The function Count dispatches to the particular Iterate for the tree structure and then that Iterate dispatches to the Action for counting.