Chapter 5

Iteration

All of the data structures examined in the first part of this book can be used to maintain collections of values. The problem posed by the task of iteration is simply to define a mechanism which will permit the programmer who is making use of such a data structure to access, one at a time, each element held by the collection. Since the task is easy to specify, and occurs with great frequency, the various alternative techniques that can be employed in solving the problem of iteration provide an excellent illustration of the range of programming techniques that are possible in a multiparadigm language. In this chapter we will explore five different solutions to the iteration problem. Each solution is inspired by, and is similar in spirit to, a mechanism found in one or more widely used programming language. Following a description of these five techniques we will explore two important variations on the problem of iteration; iteration with premature termination, and iteration of two or more structures in parallel. Our vehicle for all of our examples will be the object oriented version of the list structure described in Chapter 4.

5.1 Variation 1 – Direct Manipulation

The most obvious approach to the task of iteration is for the user of the list abstraction to directly cycle over the elements contained in the individual links. This is the technique employed, for example, in the `includes` method. First, a variable of type `Link` must be declared. Then, a `while` loop is written that iterates over each link field. An example of this approach as it could be used to print each element of a list might look something like the following:

```
var
    aList : List[integer];
    p : Link[integer];
```

```
begin
    ...

        { print out the vales maintained by a list }
    p := aList.data;
    while defined(p) do begin
        print(p.datum);
        print(" ");
        p := p.link;
        end;
```

This solution, while workable and efficient, is unsatisfactory in several regards. The major objection is that this solution forces the user of the `List` data abstraction to have detailed knowledge of the way in which lists are represented. In particular, the user of a list must not only know that elements are maintained in links, but must declare the seemingly extraneous variable `p`. But it is even worse, for not only must the variable `p` be declared, but it must be declared as an instance of the class `Link`, a class which is otherwise hidden from the user of the list abstraction.

Issues such as the necessity to make explicit use of data fields used in the implementation of a data structure are only a minor annoyance in programs developed by a single programmer working in isolation, but become a major headache in large programs developed by several programmers working together. In such situations one programmer will create software components, such as the list class, which are then used by another programmer. We would like to minimize the amount of information the second programmer needs to know in order to make use of the services provided by the first programmer.

Several years ago the computer scientist David Parnas captured the essential goals of *information hiding* in a pair of principles, which are now referred to as *Parnas' Principles*:

1. The developer of a software component should have access to all the information necessary to provide the services assigned to the component, and *nothing more*.

2. The user of a software component should have access to all the information necessary to make use of the services provided by the component in the solution of a problem, and *nothing more*.

The emphasis is purposely placed on the final limiting clause. In short, we would like to develop a mechanism whereby the user of a list class could perform the iteration task without knowledge of the internal structure of lists.

5.2 Variation 2 – Passing the Action as a Function

As an alternative to the direct manipulation of link values, the next solution we will investigate relies on the ability to create functions as expressions. In particular, functions can be passed as arguments to procedures and returned as results of execution. The use of functions in this fashion is one small aspect of *functional programming*, a topic we will investigate in much more detail in Part IV of this book.

The basic idea is to define in class `List` a method, which we will call `onEach`, that takes as argument a one-argument function. When the `onEach` method is invoked, the function supplied as argument is then executed repeatedly, once for each element contained in the list. Each time the function is executed a different value of the list is passed as argument.

```
class List [T : equality];
var
    data : Link[T];
    ...

    function onEach(fun : function(T));
    var
        p : Link[T];
    begin
        p := data;
        while defined(p) do begin
            fun(p.datum);
            p := p.next;
        end;
    end;
    ...
end;
```

Since the `onEach` method is part of the `List` class, it is permitted to directly access the link fields in the list without violating Parnas' principles.

One way this operation could be used to print out the values maintained by a list is for the user to define a special printing function. The name of this function could then be passed as argument to the method `onEach`:

```
function printValue(i : integer);
begin
    print(i);
    print(" ");
end;
```

```
var
    aList : List(integer);
    ...
        { print out the values held in the list }
    aList.onEach(printValue);
end;
```

The printValue function would then be invoked once for each value in the list named aList. An even better solution eliminates the necessity of explicitly creating the function printValue. This approach uses the ability in Leda to create unnamed functions directly as expressions, for example in an argument list. The following statement shows how the onEach method might be used in this fashion to print the values maintained by a list:

```
        { print out the values held in the list }
    aList.onEach(function(val : integer);
            begin
                print(val);
                print(" ");
            end);
```

The function passed as argument to onEach is created directly in the argument list. Notice this function does not have a name, and no other extraneous variables have been declared. In all other respects the syntax used to create the function is exactly the same as the syntax used in normal functions.

Although not used in this example, functions defined within another function or program capture the environment in which they are defined. That is, such functions can make use of variable values accessible at the time the function was defined, and the values of those variables will be retained as long as the function is accessible. We will make use of this aspect of nested functions later in this chapter when we discuss the generator approach to iteration.

5.3 Variation 3 – Using Relations and Backtracking

In chapter one we noted that a feature necessary to support the relational, or logic programming paradigm is the ability to perform automatic backtracking to revisit prior computational states. Since Leda supports the logic programming paradigm, it naturally includes an automatic backtracking facility. An alternative approach to the problem of iteration can be constructed by making use of this mechanism. While not exactly logic programming, *per se*, this solution is in the spirit of the logic programming facilities in Leda. (In Part V we will explore more traditional logic programming in Leda).

Backtracking in Leda is delimited by a `for` or `if` statement. Within the expression portion of such a statement the programmer can invoke a relation. A relation is simply an expression of type `relation`, most generally constructed out of the relational assignment operator (to be described shortly), boolean expressions, logical operators or functions which return relations. The boolean `or` operator naturally defines a "solution" to a relational expression as a sequence of alternative choices. Each alternative solution is tried in turn. In the case of lists, a "solution", as implemented by the relation named `items`, is simply a binding of a value in the list to the argument variable. Using the `for` statement in this fashion, the following code fragement will print the values held in a list, separated by spaces:

```
        { print out the values held by a list }
    for aList.items(i) do begin
        print(i);
        print(" ");
        end;
```

This approach to iteration is straight-forward and simple, and will be the technique we will most frequently employ in the remainder of the text.

As with several functions we have seen already, the implementation of the `items` relation is accomplished by a combination of methods in both class `List` and `Link`, the bulk of the work being performed in the latter class. The argument to both methods is passed by reference, so changes made within the method will be reflected as changes made to the actual argument. The definition in class `List` is as follows:

```
class List [T : equality];
var
    data : Link[T];
  ...

    function items(byRef val : T)->relation;
    begin
            { enumerate the values held in list }
        return defined(data) & data.items(val);
    end;
  ...
end;
```

The backtracking takes place within the `Link` class. At each step through this recursive procedure there are two alternatives. The first alternative invokes the `unify` relation, which you will recall from Chapter 3 (page 43) binds the value of the first argument to the quantity given in the second. The second alternative performs a recursive call, which subsequently binds the remaining values to the variable.

```
class Link [T : equality];
var
    datum : T;
    next : Link[T];
  ...

    function items(byRef val : T)->relation;
    begin
        return unify[T](val, datum)
            | defined(next) & next.items(val);
    end;
  ...
end;
```

The `or` statement, when used with relational values, is the key component of the backtracking mechanism. Each alternative possibility in an `or` expression is systematically executed in conjunction with the statement portion of the `for` or `if` statement. In this manner every element in the list will eventually be generated. (The complete details of the implementation for the logic programming paradigm will be discussed in Chapter 22.)

5.4 Variation 4 – Data Structure Iterators

In many object-oriented languages a popular approach to the problem of iteration is to employ a type of data structure called an *iterator*. An iterator is an auxiliary data structure created with the sole purpose of permitting access to the values held in a collection through a sequence of simple operations, while hiding from the user of the iterator the actual implementation of these operations. Because an iterator is usually developed by the author of the data structure over which it operates, it is permitted access to the inner working of this class without violating Parnas' principles.

A simple iterator for our `List` class might define only four operations. The first is initialization, by which means an iterator is bound to a list over which it will flow. The remaining three operators test to see if there are any remaining values to be generated, return the current value, and increment, or move the iterator to the next value. An implementation of such an iterator can be given as follows:

```
class ListIterator[T : equality];
var
    finger : Link[T];

    function init(aList : List[T]);
```

```
    begin
        finger := aList.data;
    end;

    function atEnd()->boolean;
    begin
        return ~ defined(finger);
    end;

    function current()->T;
    begin
        if defined(finger) then
            return finger.datum;
        return NIL;
    end;

    function increment();
    begin
        if defined(finger) then
            finger := finger.next;
    end;
end;
```

The iterator maintains a pointer, or "finger", into the data values held by the list structure. Upon initialization this finger is set to refer to the first element, and is moved to the next value by the increment operation. When the end of the list is reached the value held by the finger will naturally become undefined. Thus, this condition is used to determine whether or not the iterator has reached the end of the loop, while the value pointed to by the finger (if defined) is used to generate the current element being yielded by the iterator.

To iterate over the values in a list an iterator is declared and initialized, and then a simple loop is formed making use of the iterator operations:

```
var
    aList : List[string];
    listItr : ListIterator[string];
 ...
begin
 ...
        { create and initialize a list iterator}
    listItr := ListIterator[string]();
    listItr.init(aList);
```

```
        { cycle over the values held in the list }
    while ~ listItr.atEnd() do begin
        print(listItr.current());
        print(" ");
        listItr.increment();
    end;
 ...
end;
```

Iterators can be easily developed for all the common data structures. While the iterator solution seems to suffer from one of the problems of the first approach, namely; the need to introduce a seemingly extraneous variable, it nevertheless permits complex data structures to be examined piece by piece without violating the principles of information hiding.

Note that input/output operations can, in most computer languages, be considered a form of iterator. That is, input and output operations are often mediated through a variable, typically declared as type "file", with which various operations can be performed. The actual data itself is not contained in the variable, but is held in an external storage area.

5.5 Variation 5 – Data Generators

A *generator* is a function which will yield a sequence of values, one new value each time it is invoked. Probably the most common form of generator function is a random-number generator, which yields a new pseudo-random integer value on each call. But generators can be constructed for other tasks. In particular, the task of iteration can be tackled by creating a generator which will yield, on each successive invocation, a different value from the underlying data structure.

If the functional solution as implemented by the method named `onEach` is thought of as "bundling up the action to be performed and handing it to the list," then the use of generators is almost exactly the reverse. Namely, the list itself is bundled into a structure which will yield each element in turn, and it is this structure which is given to the programmer to manipulate.

While a random number generator can be considered to be producing a sequence which is infinite in length, the generators for data structures produce a finite sequence. Thus, some protocol must be observed for indicating the end of the sequence. The technique we will use is to produce the value `NIL` when the values are exhausted.

Imagine we add a method `generate` to the `List` class in order to create a list generator. Then the task of printing all the values held in a list might be implemented as follows:

```
var
    aList : List[string];
    gen : function()->string;
```

```
    val : string;
 ...

    gen := aList.generator();
    val := gen();
    while defined(val) do begin
        print(val);
        print(" ");
        val := gen();
        end;
 ...
```

Note that the generator itself is stored in a variable which is declared as type function. The generator function requires no arguments, but produces on each call a value of the appropriate type.[1]

To implement the `generator` operation we define a function which returns another function as a result. As we noted earlier, functions defined as expressions capture and retain the environment in which they are defined. Thus, within the `generator` function we can define a pointer to the current element, similar to the finger used in the iterator style of loop. This value is initially set to reference the first element in the list, and is continually modified as each execution of the returned function takes place.

Within the generator function itself there is another local variable which is used to hold the value being returned. If the pointer `current` references a legal value then this variable is set, otherwise once all values have been enumerated it remains undefined.

```
class List [T : equality];
var
    data : Link[T];
...

    function generator()->function()->T;
    var
        current : Link[T];
    begin
        current := data;
        return function()->T;
            var
                val : T;
```

[1] More complicated use of the generator programming paradigm requires the ability to "restart" a generator so as to set it again to the beginning of the sequence it is producing. Often this is specified by passing a value to the generator function, for example a zero might indicate that the next value in sequence is to be produced, while a one would indicate the sequence should be reset to the beginning.

```
            begin
                if defined(current) then begin
                    val := current.datum;
                    current := current.next;
                end;
                return val;
            end;
    end;
 ...
end;
```

The generator solution would appear to suffer from at least one problem that is also found in the iterator solution. This is that the construction of a loop requires the introduction of an extra variable to hold the generator or iterator. But in another sense this weakness can also be a strength. By breaking the task of iteration into a number of distinct steps we allow a greater flexibility in the way the components can be assembled. We will make use of this property of generators in a later section when we discuss how several lists can be traversed at one time.

5.6 Premature Termination

Often loops are used not to process an entire list, but to search for an element that satisfies a certain property. In such cases it is usually useless and inefficient to continue the iteration once the selected value has been located. Thus, it is desirable to have some ability to terminate a loop prematurely, before all elements have been examined.

In those forms of iteration we have examined which use `while` statements to control the loop (namely, the direct manipulation solution, the iterator solution, and the generator solution) it is a simple matter to add an additional condition to the expression being tested by the `while` statement. We show here the modification of the solution formed using iterators. Imagine a list contains a sequence of names in alphabetical order, and we wish to halt as soon as a name begins with a "C" or lexicographically larger letter. The loop to perform this could be written as follows:

```
 ...
    listItr := ListIterator[string]();
    listItr.init(aList);
    while ~ listItr.atEnd() & listItr.current() < "C" do begin
        print(listItr.current());
        print(" ");
        listItr.increment();
    end;
```

```
...
```

There are two general ways to handle the problem of premature termination when using the relational form of iteration. A relation used in the expression portion of an `if` statement will search for the *first* solution which satisfies the condition. Thus a search often takes the form of a relation which generates values and a condition which tests for the desired solution:

```
if aList.items(val) & val >= "C" then
    ...
else
    ...
```

An advantage of this form of search is that an `else` clause can be used to determine if the search was unsuccessful. Alternatively, the `for` statement used with relations allows an additional stopping condition. The boolean expression following the `to` keyword is evaluated at the *end* of the statement portion of the loop. Execution continues as long as the expression is false, and halts immediately the first time the expression evaluates to true.

```
for aList.items(val) to val >= "C" do begin
    print(val);
    print(" ");
end;
```

Because of a peculiar feature of the implementation technique used in conjunction with the `for` statement, it is not possible to use a `return` statement inside the body of a relational `for` loop. The following program is not permitted by the `Leda` system, and will generate a compile time error:

```
for aList.items(val) do begin
    if val >= "C" then
        return val;
end;
```

It is important to remember that the test in the relational form is performed after the statement which forms the body of the loop has been executed, while the test in the `while` loop is executed before the statement. This means that the relational version will print the first name which is lexicographically larger than "C" and then halt, while the alternative loops will halt prior to printing this value. The effect of the first loop can be achieved by including a conditional test in the body of the loop:

```
for aList.items(val) to val >= "C" do
    if val < "C" then begin
```

```
            print(val);
            print(" ");
        end;
```

It is somewhat more difficult to modify the functional technique in such a way that it will permit premature termination. One common suggestion is to have the function being passed to the `onEach` method return a boolean value, and execute as long as this boolean value remains true.

5.7 Iteration of Multiple Structures in Parallel

Consider the problem of taking two lists of equal length which each contain integer values, and generating a third list of the same length so that each value in the third list contains the sum of the corresponding values in the first two. While simple to describe, the solution to this problem is made difficult by the fact that it requires the ability to simultaneously iterate over two lists in parallel. Not all the approaches to iteration we have described in this chapter will permit this style of use. For example, consider nesting successive calls on the function `onEach`. An attempt at a solution might look something like the following:

```
var
    firstList, secondList, thirdList : List[integer];
begin
 ...
    thirdList := List[integer]();
    firstList.onEach(function (x : integer);
        begin
            secondList.onEach(function (y : integer);
                begin
                    thirdList.addToEnd(x + y);
                end);
        end);
 ...
```

If the two input lists each contain n elements, the resulting list will contain not the n pair-wise sums, but n^2 elements which correspond to each pairing of values from the first list with each value from the second.

A similar situation will occur if one tries to use the relational approach to iteration:

```
var
    firstList, secondList, thirdList : List[integer];
    x, y : integer;
```

```
begin
 ...
    thirdList := List[integer]();
    for firstList.items(x) & secondList.items(y) do
        thirdList.addToEnd(x + y);
 ...
```

Each of the remaining three approaches to iterator that we have considered in this chapter have been criticized because they required the introduction of at least one seemingly extraneous variable. In other sense, however, these variables provide an additional "handle" with which one can control the iteration process. In particular, since iteration is accomplished through the combination of a number of different statements, it is easier to combine the various components together in different manners, thereby producing a variety of effects. We illustrate this by showing how the list of sums can be produced using generators. The solution using the other two techniques is similar.

```
var
    firstList, secondList, thirdList : List[integer];
    x, y : integer;
    firstgen, secondgen : function()->integer;
begin
 ...
    thirdList := List[integer]();
    firstgen := firstList.generator();
    secondgen := secondList.generator();
    x := firstgen();
    y := secondgen();
    while defined(x) & defined(y) do begin
        thirdList.addToEnd(x + y);
        x := firstgen();
        y := secondgen();
        end;
 ...
```

Thus, while the techniques that make use of functionals or relations are, for most problems encountered in Leda, the easiest to apply solution to the problem of iteration, there are situations where one or more of the alternative techniques may be preferable.

Notes and Bibliography

Parnas' principles on information hiding were first described in [Parnas 72].

The brute force approach to iteration is the traditional technique, used in languages such as Algol and C. The technique of wrapping up the action to be performed as a function and passing this function as argument to the data structure is found not only in functional languages, but (perhaps surprisingly) in the object-oriented language Smalltalk [Budd 87, Goldberg 83]. The relational approach is inspired not only by the mechanism used in Prolog [Sterling 86], but is very similar to the technique used by the language Icon [Griswold 90]. The use of iterators is common among C++ programmers [Stroustrup 86, Coplien 92], and has been used in other languages [Grogono 91]. I myself have authored a textbook on data structures in C++ which made extensive use of iterators [Budd 94]. The generator approach is probably the least commonly used mechanism of the five, but has been employed in part by the languages Icon and Smalltalk [Griswold 81, Budd 87], and has been examined in detail in a book specifically devoted to the generator approach [Berztiss 90].

The language Smalltalk has an interesting solution to the problem of premature termination. In Smalltalk function-like structures can be created using a facility known as a *Block*. Like our functions, these can be passed as arguments to other functions, and evaluated at some later point. Like a function, when executed they act as if they were running in the context in which they were defined. However, unlike our functions, a return statement executed from within a block acts as if it were a return from the defining context, and *not* a return from the block/function.

Other interesting approaches to the iterator problem can be found embodied in the languages CLU [Liskov 81], and BETA [Madsen 93]. In particular, in BETA the inheritance mechanism is employed so that a subclass is created which implements the actions to be performed in a method which refines a function defined in the list class. In an earlier conference paper on Leda [Budd 92] I described some of the history of this issue and compare the various approaches to the techniques possible using Leda.

I am indebted to Jakub Těšínský, a student from the University of Prague, for suggesting the implementation technique used in the premature termination of iteration using relational expressions.

Exercises

1. Show how to modify the `generator` function for Lists so as to implement the ability to "restart" the generator at the beginning of the sequence, as described in the footnote on page 99.

2. Modify the `onEach` method as described in Section 5.6. That is, the method will now take as argument a function which returns a boolean value, and will execute as long as the boolean result yielded by this function is true, terminating on the first false value.

3. In Section 5.7 the problem of generating the pair-wise sum of two lists of integer values is considered. In the solution provided which makes use of generators, what happens if the two lists are of different lengths? Modify the solution so that the third list

continues with values simply copied from the longer list (in essence, as if the shorter list were extended with zero entries).

4. Write a program that uses iterators to generate a list containing the pair-wise sum of values of two argument lists, as described in Section 5.7. Compare your solution to the solution that uses generators.

5. Write a function

```
function onEachPair [X : equality]
    (list1, list2 : List[X], fun : function(X, X));
```

The function takes as arguments two lists an a function of two arguments. It traverses each list in parallel, executing the function of each pair of values. Can you implement this without violating Parnas' principles? (That is, without needing to make explicit use of the internal representation of lists?)