Chapter 7

Sets

The problem we will consider in this chapter is the implementation of a data structure that corresponds roughly to the notion of a *set* in mathematics. Our purpose is not to explain the concept of a set (we assume the reader is already familiar with the idea) but to further explore some of the techniques that can be used in the representation of common data structures.

For this chapter we will define a set to be an unordered collection of objects of the same type, with no value appearing more than once in the set. In order to simplify our presentation, we will consider only three operations on sets. The action of *addition* incorporates a new element into the set, while *inclusion* is a test used to determine if an element is already present in the set. Finally, a *union* of one set with a second is formed by adding all the elements from the second set that do not appear already in the first. The exercises at the end of the chapter explore the implementation of various other set operations.

Unlike the chapter on Lists (Chapter 4), which developed a single implementation of a data structure that made use of facilities from multiple different paradigms, in this chapter we will consider four different implementation approaches, each more-or-less pure in their own paradigm. Two of the approaches will make use of object-oriented programming, and two of the approaches will use functional techniques. The presentation of different implementations permits the reader to explore the range of potential solution techniques, and to more clearly see some of the advantages and disadvantages of the different methods.

7.1 Object-Oriented Sets

Object-oriented programming is in large part a paradigm that emphasizes the reuse of existing software abstractions in the development of new software components. We saw this in the previous chapter, where we constructed in an object-oriented fashion the `Table` data type on top of the `List` abstraction from Chapter 4. In the first part of this chapter we will once again illustrate how this is accomplished by building the object-oriented versions of our set data abstraction by making use of the `List` data structure.

There are two primary mechanisms available for software reuse using object-oriented programming, and both are applicable in the development of a set data abstraction. In the first of the following two sections we illustrate software reuse using the idea of *composition*, while in the second section we develop an entirely different approach using the technique of *inheritance*.

7.1.1 Implementing Sets using Composition

You will recall that an object is simply an encapsulation of state (data values) and behavior. When using composition to reuse an existing data abstraction in the development of a new data type, a portion of the state of the new data structure is simply an instance of the existing structure. This is illustrated in Figure 7.1, where the data type `Set` contains a field of type `List`. In a similar fashion, in Chapter 6 we constructed the `Table` data structure which held values in a field of type `List`.

Operations used to manipulate the new structure are implemented by making use of the existing operations provided for the earlier data type. For example, the implementation of the `includes` operation for our set data structure simply invokes the similarly-named function already defined for lists. Notice that both the includes test and the addition function check to make sure the list data field has been initialized before they perform their operation, and initialize the data field to hold a newly created empty list if needed.

The addition of a new element to a set first checks to see if the value is already contained in the collection. If it is already in the list it is not added, since our definition of a set stipulates that each element appears only once in the collection. Only if the element is not already in the set is it added to the list.

The operation of union is made difficult by the necessity of looping over the elements in the second set. Since this is will undoubtedly also be an operation users of the set data abstraction will wish to perform, we make available the function named `items` that was used with the list abstraction to perform iteration. The `items` function simply invokes the functions with the same name in the list class.

Figure 7.2 illustrates the use of our set data abstraction. An array literal is defined to hold the names of different types of animals. Two sets are defined, one containing animals that bark and one containing animals that are found in a zoo. A loop then prints out information concerning each type of animal. The output of the program would be as follows:

```
a cat is a non-zoo animal that does not bark
a dog is a non-zoo animal that barks
a seal is a zoo animal that barks
a lion is a zoo animal that does not bark
a horse is a non-zoo animal that does not bark
```

```
class Set [T : equality];
var
    values : List[T];    { hold data values in a list }

    function add (newValue : T);
    begin        { add new value to set, by adding to list }
        if ~ defined(values) then
            values := List[T]();
        if ~ values.includes(newValue) then
            values.add(newValue);
    end;

    function includes (value : T)->boolean;
    begin        { see if value is held by set }
        return defined(values) & values.includes(value);
    end;

    function items(byRef val : T)->relation;
    begin        { iterate over values from set }
        return defined(values) & values.items(val);
    end;

    function unionWith(secondSet : Set[T]);
    var        { add all elements from second set }
        val : T;
    begin
        for secondSet.items(val) do
            add(val);
    end;
end;
```

Figure 7.1: Sets Constructed using Composition

```
const
    animals := ["cat", "dog", "seal", "lion", "horse"];
var
    barkingAnimals : set[string];
    zooAnimals : set[string];
    name : string;
begin
    barkingAnimals := set[string](); { make an empty set }
    zooAnimals := set[string]();

    barkingAnimals.add("dog");
    barkingAnimals.add("seal");

    zooAnimals.add("seal");
    zooAnimals.add("lion");
    zooAnimals.add("lion");    { second add has no effect }

    for animals.items(name) do begin
        print("a ");
        print(name);
        if zooAnimals.includes(name) then
            print(" is a zoo animal")
        else
            print(" is a non-zoo animal");
        if barkingAnimals.includes(name) then
            print(" that barks\n")
        else
            print(" that does not bark\n");
    end;
end;
```

Figure 7.2: An Example Program using Object-Oriented Sets

```
class Set [T : equality] of List[T];

    function add (newValue : T);
    begin         { add only if not already in list }
        if ~ includes(newValue) then
            List[T].add(self, newValue);
    end;

    function unionWith(secondSet : Set[T]);
    var
        val : T;
    begin         { add all elements from second set }
        for secondSet.items(val) do
            add(val);
    end;
end;
```

Figure 7.3: A Set defined using Inheritance

7.1.2 Using Inheritance

An entirely different mechanism for software reuse in object-oriented programming is the concept of inheritance. Using inheritance, a new class can be declared to be a *subclass*, or *child class* of an existing class. By doing so, all data areas and functions associated with the original class are automatically associated with the new data abstraction. The new class can in addition define new data values or new functions. The new class can also *override* functions in the original class, by simply defining new functions with the same name.

These possibilities are illustrated in Figure 7.3, which implements a different version of the set data abstraction. By naming the class `List` following the keyword `of` we indicate that the new abstraction is an extension, or a refinement, of the earlier class. This means that all operations associated with lists (Figure 4.1, page 63) are immediately applicable to sets as well. Notice that the class does not define any new data fields. Instead, the data fields defined in the `List` class will be used to maintain the set elements. Similarly, functions defined in the parent class can be used without any further effort. In particular, this means that no work is required to implement the inclusion test, since the list data structure already provides a function named `includes` that performs exactly the task we desire.

The addition of an element to a set, on the other hand, has a slightly different meaning than the addition of a value to a list. Thus, the implementation of the addition function must supersede, or override, the function with the same name found in the list class. Contrast the definition of the `add` function shown in Figure 7.3 with the similarly named function defined for lists (Figure 4.2, page 68). The function for sets first checks to see if the element is in the collection, only adding the new element if it is not present in the set. If it is not yet present in the list, then we want to invoke the `add` function from the class `List`.[1]

Simply saying `self.add(...)` would not work, since the compiler would interpret that construct as a recursive call on the overridden `add` function, and the result would be an infinite execution loop. The solution to this problem involves a more general mechanism in the Leda language. When a class name is used in an expression that references a function defined within the class, as in

```
List[T].add
```

a transformation is applied to generate a new function out of the named function. This new function takes one more argument than the original. Thus, the `add` function, which in the class `List` required only a single argument, is converted into a new function that requires two arguments. The additional argument is used to specify which list is being manipulated. In effect, it is as if the expression given above had been translated into the following function:

```
function [T : equality] (aList : List[T], newValue : T);
begin
    aList.add(newValue);
end
```

This transformation is applied in the `add` function, which converts the one-argument function from the parent class into a two-argument function, which is then immediately invoked. The technique, however, can in general be applied to any function defined within a class.

A subtle point to note in relation to this transformation is the fact that the first argument to the function being generated is declared as a `List`, but in fact is passed a value of class `Set` (namely, the value referred to by the pseudo-variable `self` from within the `Set` class). This is permitted, since the class `Set` inherits from class `List`, and thus can in all reasonable situations be used as a substitute for class `List`. We will explore this issue in more detail later in Chapter 10.

The method for performing union in the `Set` class formed using inheritance uses exactly the same technique as the function written for the class using composition. Note, however, that since the `items` function is inherited by the `Set` class, it does not need to be repeated in the class declaration, as we were required to do with the first technique.

A programmer using the set abstraction might want to use inheritance one more time in order to avoid the repeated use of fully qualified names. To do this, the programmer creates a new class with no data areas and no behavior, as follows:

[1] The question of whether the add function in the class `Set` must invoke the add function in the class `List` is tied to a fundamental difference between the so-called "American" school of object-oriented languages versus the so-called "Scandinavian" school. The references at the end of this chapter contain further pointers to discussions of this distinction.

```
class stringSet of set[string]
    { no data, no new behavior }
end;
```

The variables `barkingAnimals` and `zooAnimals` in Figure 7.2 can then be defined as instances of `stringSet`. Similarly the constructor for `stringSet` can be invoked, which simply constructs an instance of the class `Set[string]`. In all other respects the example program shown in Figure 7.2 remains unchanged, since exactly the same syntax is used with both forms of sets, and exactly the same output will be generated by each.

The is-a relation

Given the existence of these two different software reuse mechanisms it is natural to ask under what conditions it is more appropriate to use inheritance and under what conditions it is better to use composition. While it is difficult to provide a hard-and-fast rule that is applicable in all situations, a simple heuristic is useful in answering this question in the large majority of cases. This heuristic is commonly known as the "is-a" test.

To apply the "is-a" test, form an English sentence by placing the name of the child class first, followed by the words "is a", and lastly the name of the parent class. If the resulting assertion appears to "sound right", then inheritance is likely the proper technique to employ. If the sentence seems to sound a little odd, then inheritance is likely not the proper technique to use.

Let us apply the is-a heuristic to the present case. Consider the sentence:

A *Set* is a *List*.

It is likely that this assertion does not seem entirely accurate. For example, there are many features of a set that do not seem to be necessarily applicable to lists. A set is unordered, and contains only unique elements, to name just two. On the other hand, the sentence

A *Set* can be implemented using a *List*.

makes much more sense. This simple test seems to indicate, then, that composition, rather than inheritance, is the better implementation technique for constructing sets out of lists. We will return to a discussion of the is-a test when we discuss co- and contravariance in object oriented languages, in Chapter 10.

7.1.3 Contrasting Composition and Inheritance

Having illustrated two different mechanisms for software reuse, and having seen that they are both applicable to the implementation of sets, we are now in a position to comment on some of the advantages and disadvantages of the two approaches.

The mechanism of composition is the simpler of the two techniques. The advantage of composition is that it more clearly indicates exactly what operations can be performed on a particular data structure. Looking at the declaration shown in Figure 7.1, it is clear that the only operations provided for the `Set` data type are addition, the inclusion test, and set union. This is true regardless of what operations are defined for lists.

Using inheritance, on the other hand, the operations of the new data abstraction are a superset of the operations provided for the original data structure on which the new object is built. Thus, for the programmer to know exactly what operations are legal for the new structure it is necessary to examine the declaration for the original. An examination of the declaration shown in Figure 7.3, for example, does not immediately indicate that the `includes` test can legally be applied to sets. It is only by examining the declaration for the earlier data abstraction (in this case, the `List` class defined in Figure 4.1, page 63), that the entire set of legal operations can be ascertained. The observation that for the programmer to understand a class constructed using inheritance it is often necessary to flip back and forth between two (or more) class declarations has been labeled the "yo-yo" problem. (See bibliography section for references).

However, the brevity of data abstractions constructed using the abstraction mechanism is, in another light, an advantage. Using inheritance it is not necessary to write any code to access the functionality provided by the class on which the new structure is built. For this reason implementations using inheritance are almost always, as in the present case, considerably shorter in code length than implementations constructed using composition, while at the same time often providing greater functionality. For example, the inheritance implementation not only makes available the `includes` test for sets, but also the functions `remove` and `onEach`.

Finally, data structures implemented using inheritance tend to have a very small advantage in execution time over those constructed using composition, since one additional function call is avoided. (The preceding statement is true of Leda, but should not be construed to be true for all programming languages. Some languages, such as C++, have a feature that permits function calls to be replaced directly by their associated function bodies. In these languages using this mechanism the overhead of the additional function call can sometimes be eliminated.)

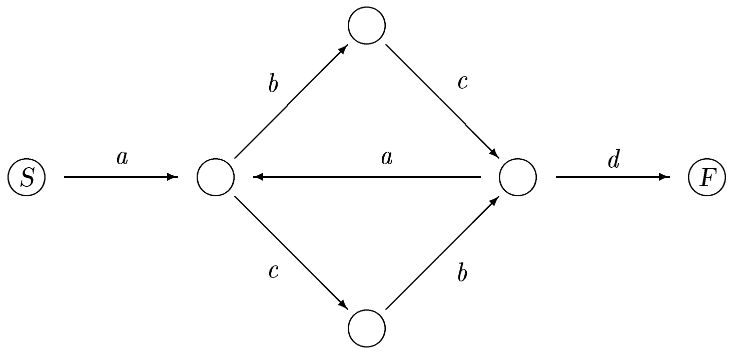
The bottom line, however, is that both techniques are very useful, and an object-oriented programmer should be familiar with either form.

7.2 Implementing Sets using Characteristic Functions

While it may be natural to first think about representing sets as data, there are many examples in both computer theory and software engineering where sets are more easily represented as actions; that is, as functions. For example, formal languages are consider to be characterized either by a description or by a recognition function. A set can be defined equally well by a regular expression:

```
( a ( bc | cb ) )* d
```

or by a finite automata:



The two forms are usually considered to be equivalent. That is, the finite automata returns true if and only if the input matches the set description.

In a similar manner, in the C standard library there is a function used to tell whether a character value represents an upper case letter. In set terms, we can think of this as testing whether the character is in the set of upper case letters. But operationally the function simply performs a calculation which returns true if the argument happens to be located within a given range:

```
int isUpperCase(char c)
{
    return c >= 'A' && c <= 'Z';
}
```

In both these cases the underlying idea is the same. Namely, to characterize a set it is sufficient to define a function which return `true` if the argument is contained in the set, and `false` otherwise. Such a function is known as a *characteristic function* for the set.

In this first section our characteristic functions will exactly match this basic idea. That is, the characteristic functions will take as argument a value, and return a boolean result that is true if the argument is contained in the set and false otherwise. We will see in the second section that this rather direct approach may have certain disadvantages, and these disadvantages can be overcome by using a slightly more indirect representation.

To implement our three example set operations we will provide a quartet of utility functions, named `emptySet`, `setAdd`, `setIncludes` and `setUnion`. The first takes no arguments, and simply generates a new empty set. Each of the latter three will take two arguments. The first argument in all cases is the characteristic function representing the set, while the second argument represents either an individual set element (in the case of addition or inclusion) or the characteristic function associated with another set (in the case of union). The first and third functions, `setAdd` and `setUnion`, will return a new characteristic function that represents the set with the addition of the new element (or elements). The second function (`setIncludes`) returns a boolean value which indicates whether or not the second argument is contained in the set. Both functions are generic, and thus must be qualified by an argument representing the type of object held in the set.

To see how these operations might be used, consider the program shown in Figure 7.4, which is a rewriting of the test program used in the earlier Figure 7.2. Changes include the following:

- A type definition defines the name `stringSet` to be a synonym for a function that takes a string as argument and returns a boolean value as result.
- The two sets are now declared simply as instances of the type `stringSet`.
- Adding values to the set is accomplished by invoking the function `setAdd`, which returns a new characteristic function value. Although we could have written the statement that initializes the set as a series of assignments, it is more characteristic of functional programming to generate the set value once, using the result of one addition as argument to the next.
- To test to see if a value is contained in a set we simply invoke the characteristic function on the value. The function returns true if it is in the set, and false otherwise.

Figures 7.5 and 7.6 illustrate the implementation of the set manipulation functions. The empty set function simply returns a new characteristic function which answers false to every query. Thus, no elements are contained in an empty set.

The implementation of the `setAdd` function is more complex. The function first checks to see if the element being added is already present in the set, by simply invoking the characteristic function with the new value as argument. If it is in the set then there is no reason to add it again, so the current characteristic function is returned. If, on the other hand, the element is not in the set, then a new characteristic function is generated.

The new function captures and retains the environment in which it was defined. Then, when invoked, the value of the variable named `addedValue` will be the value it held when the function was created. The new function takes as argument a set element, and compares the argument to the captured quantity in the variable `addedValue`. If they are equal the result `true` is returned, otherwise the existing characteristic function, which was also captured when the function was defined, is invoked to see if the argument is present in the set. This new characteristic function is then returned as the result of the `setAdd` operation.

The `setUnion` function (Figure 7.6) is similar. A new function is created that takes as argument a value, and returns true if the value is found in either the first set or in the second set.

An interesting feature to note in relation to the set union function is the way in which the code mirrors the definition of a union. A value is in the union if either it is in the first set, or it is in the second set. Thus, it is not surprising that the logical *or* operator

```
type
    stringSet : function(string)->boolean;

var
    animals : array[string];
    barkingAnimals : stringSet;
    zooAnimals : stringSet;
    name : string;

begin
    animals := ["cat", "dog", "seal", "lion", "horse"];

    barkingAnimals := setAdd[string](
        setAdd[string](emptySet[string](), "dog"), "seal");

    zooAnimals := setAdd[string](
        setAdd[string](emptySet[string](), "seal"), "lion");

    for animals.items(name) do begin
        print("a ");
        print(name);
        if zooAnimals(name) then
            print(" is a zoo animal")
        else
            print(" is a non-zoo animal");
        if barkingAnimals(name) then
            print(" that barks\n")
        else
            print(" that does not bark\n");
    end;
end;
```

Figure 7.4: An Example Program using Characteristic Functions

```
    { return a new empty set }
function emptySet [X : equality] ()-> function(X)->boolean;
begin
    return function (v : X)->boolean;
        begin        { always just say no }
            return false;
        end;
end;

    { return characteristic function for set containing new element }
function setAdd [X : equality]
    (theFun : function(X)->boolean, addedValue : X)->function(X)->boolean;
begin
        { if already in the set then do not add again }
    if setIncludes[X] (theFun, addedValue) then
        return theFun;

        { return a new characteristic function }
    return function(newVal : X)->boolean;
        begin
            if (newVal = addedValue) then
                return true;
            return theFun(newVal);
        end;
end;
```

Figure 7.5: Implementation of Sets using Characteristic Functions

```
    { see if element is contained in set characterized by function }
function setIncludes [X : equality]
    (theFun : function(X)->boolean, value : X) -> boolean;
begin
        { simply return the characteristic function value }
    return theFun(value);
end;

    { return characteristic function represent union of two sets }
function setUnion [X : equality]
    (firstSet, secondSet : function(X)->boolean)->function(X)->boolean;
begin
    return function(newVal : X)->boolean;
    begin
            { true if either in first or second }
        return firstSet(newVal) | secondSet(newVal);
    end;
end;
```

Figure 7.6: Set Union with Characteristic Functions

appears at the heart of the function shown in Figure 7.6. What is surprising, perhaps, is that nowhere does an *or* appear in the equivalent function constructed in the object-oriented version (Figure 7.3). The object-oriented algorithm describes *how* a union can be constructed, whereas the function version is more declarative, describing *what* a union represents.

For completeness we have also given in Figure 7.6 a definition of the function `setIncludes`; although it simply turns around and invokes the first argument using the second argument as value. Thus, in practice, there would be little reason to use the includes function with this representation. This is not true of the representation to be studied in the next section.

To motivate the need for a different representation, consider that a difficult problem to overcome using sets represented by characteristic functions is the task of iteration. Imagine, for example, the formation of the union of the sets of zoo animals and barking animals, followed by the printing of the elements contained in the new set. This can be accomplished, but only by explicitly iterating over *all* animals and testing each one for inclusion in the new set:

```
    thirdSet := setUnion[string] (zooAnimals, barkingAnimals);
    print("union of barking animals and zoo animals is: ");
    for animals.items(name) do
        if setIncludes[string] (thirdSet, name) then begin
```

```
            print(name);
            print(" ");
            end;
    print("\n");
```

In the next section we will consider a different representation that simplifies this problem.

7.2.1 Characteristic Functional Arguments

Using the set implementation technique illustrated in Figure 7.5, it is easy to determine if any particular element is contained in a set; one merely executes the characteristic function. If, however, we do not have a particular element in mind, but merely wish to determine which elements are present in the set, then the task is much more difficult. If the possible range of elements is relatively small, then we can simply loop over each potential value and use the inclusion test. This was the technique employed by the example program shown in Figure 7.4. Suppose, on the other hand, that the range of values is very large, for example the set of integer numbers. We would certainly not want to loop over all integers to see which values were contained in the set.[2] To solve this problem, we will propose a slightly different representation for a set.

We will continue to represent a set by a characteristic function; however, instead of the function returning a boolean value, we will take as argument to the characteristic function another function as argument, and simply apply the function argument to each value contained in the set. Thus, the characteristic function is performing the same task as the function `onEach` provided for the `List` abstraction.

To form a new empty set we invoke the function `emptySet`, shown in Figure 7.7. The value returned is a function which, when provided with an action to perform on every element, simply does nothing.

The `setIncludes` function, also shown in Figure 7.7, perhaps better illustrates the way in which this representation of functions is utilized. The purpose of the function is to test whether a specific value is found in the set being characterized by the first argument. To do this, the set is passed a new function, generated directly by a function expression. This function will be then invoked on each element of the set. When invoked, the function compares the element value against the test value which was passed to the `setIncludes` function. If any element matches, then the variable `result` is set to true. If result remains false after execution, it can only be because the tested value is not contained in the set.

The action of the `setAdd` function (Figure 7.8) is also different in the new representation than the similarly named function in the previous formulation. As before, the `setAdd` function first tests to see if a value is already contained in a collection. If so, then no further action is necessary and the existing characteristic function is returned. If, on the other

[2] Exercise 11 investigates some of the difficulties caused by this limitation in the implementation technique of Figure 7.5.

```
    { return a new empty set }
function emptySet [X : equality] ()->function(function(X));
begin
    return function(action :function(X));
    begin
        { do nothing } ;
    end;
end;

    { see if element is contained in set characterized by function }
function setIncludes [X : equality]
    (theFun : function(function(X)), value : X)->boolean;
var
    result : boolean;
begin
    result := false;

        { test each value in turn }
        { if any match argument then set the result flag }
    theFun(function(test : X);
        begin
            if test = value then
                result := true;
        end);

        { return the final value of the result flag }
    return result;
end;
```

Figure 7.7: Creation of empty set and set inclusion test using characteristic functionals

```
    { return characteristic function of set containing element }
function setAdd [X : equality]
    (theFun : function(function(X)), value : X)->function(function(X));
begin
    { if value is already in set then do not add }
    if setIncludes[X](theFun, value) then
        return theFun;

    { return a new characteristic function }
    return function (action: function(X));
        begin
            { first execute action of new value }
            action(value);

            { then execute on remainder of the values }
            if defined(theFun) then theFun(action);
        end;
end;

    { return characteristic function for set with element removed }
function setRemove [X : equality]
    (theFun : function(function(X)), value : X)->function(function(X));
begin
        { if value is not in set then do not remove it }
    if ~ setIncludes[X](theFun, value) then
        return theFun;

        { return new characteristic function}
    return function (action : function(X));
        begin
            if defined(theFun) then
                theFun(function(testVal : X);
                    begin
                        if testVal <> value then
                            action(testVal);
                    end);
        end;
end;
```

Figure 7.8: Implementation of Set Addition and Set Removal using Characteristic Functionals

```
    { return characteristic function representing union of two sets }
function setUnion [X : equality]
    (firstSet, secondSet : function(function(X)))-> function(function(X));
begin
    return function(action: function(X));
        begin
                { execute action on elements in first set }
            firstSet(action);

                { and only on elements in second set }
                { that were not in first set }
            secondSet(function(value : X);
                begin
                    if ~setIncludes[X](firstSet, value) then
                        action(value);
                end);
        end;
end;
```

Figure 7.9: Implementation of Set Union

hand, the value is not already in the set then a new characteristic function must be created. This new function takes as argument a function, named `action`, which describes the action to be performed on each element of the set. When invoked, the action will be performed using as argument the new value of the set. To perform the action on the previous values of the set the action function is passed as argument to the previous characteristic function, which was captured when the new characteristic function was constructed.

The development of a function to remove an element from a set will serve as a vehicle to further illustrate the manipulation of data structures implemented using the characteristic function technique. The `setRemove` function is shown in Figure 7.8. If the given value is not present in the set, then there is no need to remove it and thus the characteristic function given as argument can be returned. Otherwise, a new characteristic function is generated. This new function takes as argument an action to be performed, but passes to the existing characteristic function a function that only applies this action if the argument fails to match the value that has been removed. Thus, the removed value will never be executed with the given action, and it will effectively be eliminated from the set.

Finally, the implementation of set union (Figure 7.9) is a *tour-de-force* in the use of this representation. The task of forming a union is made complicated by the requirement that values must appear in a set only once. If a value appears in both portions of a set union, the revised characteristic function must ensure that an action is only performed on the element once. The new characteristic function first performs the given action on all elements in the first set. Next, it performs the action only on those elements in the second set that do not appear in the first set.

With a suitable change in the declaration for the variable `zooAnimals` and the variable `barkingAnimals`, and by changing the invocation of the characteristic function to instead use the predicate `setIncludes`, the example program shown in Figure 7.4 works exactly as before using this new representation. The new type declaration for sets is as follows: following:

```
type
    stringSet : function(function(string));
```

To illustrate the additional power provided by this new technique, we return again to the problem of forming the union of the set of barking animals and zoo animals, and then printing out the elements in this third set. In fact, we will first solve the more general problem of printing the values held in an arbitrary set. You will recall that it was not possible to write such a function for sets maintained in the first characteristic functional form without explicit knowledge of the range of values, and even then only by iterating over the entire range of values and testing each one. But such a function is very easy to write in the new format:

```
function printSet [ X : equality ] (aSet : function(function(X)));
begin
        { print each value in the set }
    aSet(function(val : X);
        begin
            print(val);
            print(" ");
        end);
end;
```

With this function the formation and printing of the union can be accomplished in a single expression:

```
print("the union of barking animals and zoo animals is:");
printSet[string] (setUnion[string] (barkingAnimals, zooAnimals));
print("\n");
```

7.3 A Comparison of the Techniques

Although many readers will undoubtedly find the object-oriented implementation of sets easier to understand, several of the exercises at the end of this chapter together suffice to show that there is essentially no functionality provided by the object-oriented version that cannot also be simulated using the functional techniques described in Figures 7.7, 7.8 and 7.9. Thus, based on functionality alone there is little reason to prefer one technique over the other. Nevertheless, several important differences can be noted:

- Using the functional approach the representation of a set always increases in size, never getting smaller. Even removing an element from a set increases the amount of storage used to represent the set. Using the object-oriented technique the size of the set increases and decreases as elements are added and removed. (For this reason the characteristic functional approach is most often used only when the elements in the set are fixed, or change very slowly).
- The object-oriented version is able to leverage considerable functionality off of the existing `List` data structure. Using the functional technique basically all new functionality must be provided from scratch. (This is not to say that the use of functional techniques does not encourage software reuse. In fact, functional programming supports a very powerful, if different, style of software reuse. We will explore this topic later in Chapter 14.)
- In some circumstances, the functional approach yields procedures that are closer to the definition of the corresponding set operation than are the equivalent methods using the object-oriented technique. Compare, for example, the set union function shown in Figure 7.6 with the earlier function shown in Figure 7.1. The object-oriented version describes operationally how to form the union, but not what a union represents. The functional solution is much closer to the definition of the union.
- The declaration of a set using the object-oriented version is slightly more self-evident then the declaration of a similar structure using the functional approach.
- Interestingly, the functional approach can result in programs that are faster. The following table provides timings for an example program that created a set containing 100 random elements, then performed 10,000 tests for inclusion. This is perhaps why the most common use of characteristic functions occurs in those situations where speed is very important (see comments in the bibliography for further discussion).

Implementation	execution time (seconds)
Figure 7.1	103.9
Figure 7.3	93.9
Figure 7.7	72.0

Notes and Bibliography

The "yo-yo" problem, which can occur when a new data structure is implemented using inheritance, was first discussed by Taenzer [Taenzer 89]. I also discuss this problem in my book on Object-Oriented programming [Budd 91b]. The functional implementation of sets was suggested by an example program in Samuel Kamin's book on programming languages [Kamin 90].

Sets are found as a primitive data type in a number of programming languages. Examples include Setl [Baxter 89], ABC [Geurts 85], Icon [Griswold 90] and others.

The difference between the "American" and the "Scandinavian" schools of object-oriented program concerns the question of whether an overridden function is a *refinement* or a *replacement* of the function named in the parent class. In the Scandinavian school (so named because it is characterized by the languages Simula [Dahl 66] and BETA [Madsen 93], which were created in Scandinavian countries), an overridden function must always first invoke the function in the parent class, then invoke the function in the child class. Thus, it would be more difficult to write the `Set` class using inheritance in such a system, since the `Add` function from class `List` would always be invoked for each call on `Add` in the class `Set`. This set class would then need to see if the element was already in the collection, and if so remove it once again. On the other hand, in the "American" school (characterized by languages such as Smalltalk [Goldberg 83] and C++ [Stroustrup 86]) an overridden function totally replaces the function in the parent class. This has both advantages, as illustrated by the implementation of sets from lists, and disadvantages. We will explore some of the disadvantages later in Chapter 10.

While the implementation of sets as characteristic functions may strike many programmers used to more conventional techniques as odd, the principle that data can often be replaced by code is found in a number of different situations. Finite state automata, for example, are easily described in a tabular form. Yet when efficiency is a concern, such as in parsers or lexical analyzers, the states of the finite state automata are frequently coded directly as program statements [Aho 86, Fraser 92, Horspool 90].

Exercises

1. In the same way that the addition operator was overloaded to mean addition to lists, redefine the meaning of this operator for each of the set structures defined in this chapter.

2. Add a function named `remove` to the set abstraction defined in Figure 7.1. This function should remove the item, if present, from the collection, performing no action if the item is not present in the set. For example, subsequent to the following set of statements the collection should contain exactly one item.

```
zooAnimals.add(lion);
```

```
zooAnimals.add(dog);
zooAnimals.add(dog);
zooAnimals.remove(dog);
```

3. Explain why it is not necessary to define a function to remove items from the set abstraction shown in Figure 7.3.

4. Two other operations commonly associated with sets are intersection and difference. The intersection of two sets is the set of elements contained in both sets. To intersect one set with another, remove all elements from the first set that do not appear in the second. The difference of two sets is the inverse of the intersection; it is the elements in the first set that do not appear in the second. Write functions to implement these operations using the object-oriented form of sets. Can you write functions that will work for both the sets built using inheritance and the sets built using composition?

5. Two sets are equal if they contain the same elements. Show how the equality testing operator, =, could be implemented for the set data type described in Figure 7.3. To do so, make the set class a subclass of equality, and follow the techniques used in Chapter 6 to redefine the meaning of the equality testing operator in class `Association`. How would this differ if you used the set data type described in Figure 7.1?

6. One set is said to be a subset of another set if all the values in the first are also found in the second. (The second set can also, although need not, contain additional elements). We can overload the meaning of the "less than" operator, <, to mean the subset test when used with two set values. Provide an implementation for this operator using the set data type described in Figure 7.3. How would this differ if you used the set data type described in Figure 7.1?

7. Chapter 4 introduced two functional uses of data structures, mapping and application. In application a function is provided as argument and applied to each element of a collection. This functionality was provided by the function `onEach` in the list data abstraction shown in Figure 4.1 (page 63). Implement the function `onEach` for the set data abstraction shown in Figure 7.1.

8. Explain why, to achieve the functionality described in the previous question, it is not necessary to implement the `onEach` function for the set abstraction shown in Figure 7.3.

9. Implement a function named `setRemove` which will remove an element from a set constructed using the functions in Figure 7.5. Hint: it is not possible (or at least, not easy) to actually remove the element from the characteristic function. Rather, simply create a new characteristic function that will return the value `false` when queried about the given element.

10. Implement the set operations of intersection and difference using the characteristic function representation on Figure 7.5.

11. As we note in Section 7.2.1, there is no easy way to determine which elements are contained in a set constructed in the fashion of Figure 7.5, short of simply looping over all possible values and testing each one individually. Explain how this complicates the implementation of a set equality or subset operator using this representation. Does it also complicate the implementation of `onEach`? Why does it not complicate the set operations described in the previous question?

12. Consider the following sequence of statements, which make use of the set implementation shown in Figures 7.7 and 7.8.

```
type
    intSet : function(function(integer));

begin
    intSet := setAdd:(integer)(intSet, 12);
    intSet := setAdd:(integer)(intSet, 237);
    intSet := setAdd:(integer)(intSet, 237);
    intSet := setAdd:(integer)(intSet, 1632);
    intSet := setRemove:(integer)(intSet, 237);
    intSet := setAdd:(integer)(intSet, 237);
```

    Describe the structure of the final value of `intSet`. How many elements does the set contain?

13. Show how to implement the set equality operation and subset operation using the representation provided in Figure 7.7.

14. Implement the set operations of intersection and difference using the characteristic function representation on Figure 7.7.

15. Show how to implement the functionality of the `onEach` operation using the set representation provided in Figure 7.7.