The ADT List_Set

Classes are generally used to define Abstract Data Types (ADTs). Eiffel systems come with large libraries of classes and Eiffel, the language, has practically no built-in types.

We may regard the Basic types (INTEGER, REAL etc) as being built-in. The type or class ARRAY is defined as a library class as is the class STRING. The class ARRAY or STRING are not built-in types although, due to mainfest arrays/strings, they can be regarded as been part of the language definition. These are special classes in that one is allowed to have: manifest arrays e.g. <<3,1,4,16>> and manifest strings e.g. “The Provost” as part of the language syntax. As an (elementary) example of a user (as supplier) defined ADT we consider the ADT, LIST_SET.

We consider first how we are going to use the type or class

```
class LIST_TEST
creation    make
feature
        make is
                local
                        s : LIST_SET[STRING]
                do
                        !!s
                        io.put_string(“%N Enter words—lowercase--%N”)
                        from
                                io.read_word
                        until
                                io.last_string = “quit”
                        loop
                                s.add(io.last_string)
                                io.read_word
                        end
                        io.put_string(“%NSearching for a word, enter word:”)
                        io.read_word
                        if s.has(io.last_string) then
                                io.put_string(“%NWord Found%N”)
                        else
                                io.put_string(“%NWord is not in the List_Set%N”)
                        end
                        io.put_string(“%NRemove a word: Which one?”)
                        io.read_word
                        if s.has(io.last_string) then
                                s.remove(io.last_string)
                                io.put_string(“%NWord has been removed%N”)
                        else
                                io.put_string(“%NWord is not is list%N”)
                        end
                end—make
end—LIST_TEST
```

The class LIST_SET[G] is based directly on the class LIST given in Chs 4 and 8 in Switzer, Robert "Eiffel, an Introduction" (Prentice-Hall) (The book uses Eiffel/S, now Visual Eiffel). The class is a generic class (as in Ch 8 Switzer) in that we are allowed to have a LIST_SET[G] of arbitrary type
i.e. in defining the class we use the class variable/parameter G which will be instantiated with a particular type when used by a client,

e.g.   s : LIST_SET [STRING]

The class is implemented as a 'linked-list' where the nodes are objects from the class NODE. The NODE class defines the cells or nodes used to hold each item in the LIST_SET.

```
class NODE [G]
feature
	item : G
	next : NODE [G

	set_item (x : G) is
		do
			item := clone(x)
		end—set_item

	set_next (n :NODE [G]) is
		do
			next := n
		end—set_next
end—NODE
```

```
class  LIST_SET [G]
feature {NONE}
      first_node : NODE[G]  -- hidden attribute
feature

      has (x : G) : BOOLEAN is
            local
                  n : NODE [G]
            do
                  from
                        n := first_node
                  until
                        n = void or else equal(x, n.item)
                  loop
                        n := n.next
                  end
                  result := (n /= void)
            end—has

      put(x : G) is
            local
                  n : NODE [G]
            do
                  if not has(x) then
                        !!n
                        n.set_item(x)
                        n.set_next(first_node)
                        first_node := n
                        count := count+1
                  end
            end—put
```

```
    remove (x : G) is
        local
            prev, pres : NODE[G]
        do
            from
                pres := first_node
            until
                pres = void or else equal(x, pres.item)
            loop
                prev := pres
                pres := pres.next
            end
                if pres /= void then
                    if prev = void then
                        first_node := pres.next
                    else
                        prev.set_next(pres.next)
                    end
                    count := count-1
                end
        end—remove

    count : INTEGER

    empty : BOOLEAN is
        do
            result := (count = 0)
        end—empty

end—class LIST_SET
```

Exporting features/ Information Hiding

The keyword "feature" is used to control the 'visibility' of the features i.e. the attributes and routines. If "feature" is unqualified or equivalently qualified by ANY, i.e. if we have

feature {ANY}

then all the features up to the next keyword "feature" are exported to any class, i.e. to all classes inherited from the class ANY.

If "feature" is qualified by NONE then the following features are exported to no class, i.e. exported all classes inheriting from NONE, but no classes inherit from NONE. The class NONE is an empty or virtual class inheriting from all classes.
e.g.

feature {NONE}
    first_node : NODE[G]
feature

This means that the attribute "first_node" is not exported.

In general the keyword "feature" is used to control <u>information hiding</u>. The features between

feature {C1,C2, C3}

and the next keyword "feature" are exported only to the classes C1,C2 and C3.

If one wants to let objects of just the same class have access to the features of its class, A say, then we would use

feature {A}

Accessing features of a Class.

Given two classes B and C, let C be the <u>client</u> of B and so B is a <u>supplier</u> to C,
i.e. in class C we may have x:B then through the entity x we can have access to features of B.
Let a:D be an attribute of B and let p be a procedure in B.
In class C we may have y:D and so then we can have y:=x.a
and also use x.p (or x.p(e1,e2) if procedure p has args.).
x.p calls the procedure p via the object x:B
If class B had a function f declared as a feature that returned an object of type D then we could have y := x.f(e1,e2) or if f had no args. then y := x.f
From the class C, we can't judge whether f is an attribute or a function. This benign ambiguity is intentional in Eiffel.
In these examples, x:B in class C must be bound to some object otherwise x = void. An object of the class must be created using e.g. !!x
x.a is attribute of B and so we cannot update this attribute in class C

WRONG!!   x.a := u   WRONG!!

Only class B, using its own routine e.g. set_a, can change the attribute, a, by e.g. x.set_a(new_a) where set_a is a procedure which updates the attribute a.

Objects as 'machine/devices'

We can view objects in another way; as machines or devices in which the functions and attributes give information (the dials / meters on the machine) and the procedures (the switches / buttons) change the state of the machine. The other view of an object was to regard an object as an instantiation of an ADT

e.g. s : LIST_SET[STRING]

and !!s creates an LIST_SET object,
i.e. s is a list_set of strings which can have strings added to and removed.

As an example of regarding an object as a abstract machine consider the library class SINGLE_MATH. To use functions from this class in a class C let m: SINGLE_MATH, then !!m creates an object (abstract machine) and, for example, m.floor(z) returns $\lfloor z \rfloor$.

Adding Traversal Routines to LIST_SET

We need routines to traverse a list in order, for example, to print out the list. To facilitate traversal we intoduce a 'hidden' attribute named `cursor` that references a NODE. The `cursor` will be used to move from node to node in the list. The notion is similar to the notion of `cursor` in an editor. In our extension of the the LIST_SET class the `cursor` is used just for traversal; later the notion of `cursor` may be used to add and remove items from a list. Since the traversal routines are very short we can give them directly in Eiffel.

```
start is        -- set cursor back to start
        require
                not empty
        do
                cursor := first_node
        end—start

first : G is -- The item at first_node
        require
                not empty
        do
                result := first_node.item
        end—first

item : G is-- item at cursor
        require
                not empty
        do
                result := cursor.item
        end—item
```

```
forth is      -- move cursor forward
        require
                not empty
        do
                cursor := cursor.next
        end—forth

off : Boolean is -- Is cursor beyond end
        require
                not empty
        do
                result :=  cursor = void
        end—off
```

Exercise: Write a procedure that will set the `cursor` on the last item in the list and a function that returns the value of the last list item.

In using the traversal routines it is assumed that the list is not empty, hence the precondition “not empty” on the routines.

This implementation of LIST_SET is not efficient as consider the case of removing the last item in the list. We first traverse the list to find its value and then we call the routine Remove which in effect traverses the list again to remove the item.

Consider also a procedure that will print out the items in the list in sorted order assuming that the items are comparable.

Difficulties with the cursor:

Introducing a cursor may cause side-effects in functions. For example, if the cursor is at at a particular item in the middle of the list and we have a function that returns the last item in the list it is likely that the cursor will be moved.

A more difficult problem is when we remove the node/item at the cursor. When we remove an item do we leave the cursor at the left or right of the removed item. If the list has one item then there is no left or right item when the item is removed.