Programming by Contract

Eiffel supports the development of correct programs in two main ways:

1. *Eiffel is a strongly typed language.*

Each object has a type; the class from where it is defined. The compiler can check whether routines have the correct arguments by checking the types.

2. *Assertion Checking.*

In Eiffel assertions can be checked at run-time. Assertions are boolean expressions that specify what the program means. A routine can be specified by a Pre/Post condition pair. For example, a program for Floor_Sqrt, for getting the square root of a number x, can be specified by the Pre/Post condition pair

$\{ x \geq 0 \}$
Floor_Sqrt
$\{ \text{result} = \lfloor \sqrt{x} \rfloor \}$

The contract between the programmer and user of Floor_Sqrt is such that:
If the user fulfills the precondition, $\{x \geq 0\}$, then the programmer guarantees that the program will fulfill the postcondition $\{ \text{result} = \lfloor \sqrt{x} \rfloor \}$.

The syntax of Assertions in Eiffel is illustrated by

```
proc_name(parameter_list) is
        require         --Precondition
                label : Boolean expression
        local
                        . . .
        do
                <Proc_body>
        ensure          --Postcondition
                label: boolean expression

        end --proc_name
```

Assertions are more than formal or precise comments, as Eiffel can also, at run-time, check whether the assertions are true and if false report the offending assertion.

BINARY_SEARCH class

As an example of the use of assertions we will consider implementing the binary search algorithm which searches an ordered array of n items in log(n) time.

We consider first what the class is supposed to deliver rather than how it delivers.

The parameter passing mechanism of Eiffel does not allow "var" or "in out" parameters and so class attributes (index and found) are used to return values from the routines.

Eiffel adheres to the policy of having no side-effect in functions; hence, for example, the two routines read_integer (a procedure) and last_integer (a function) together are used to read in an integer. In designing the language, Meyer decided against have a language restriction in Eiffel so as to prevent side-effects in functions. It is up to the programmer to make sure functions have no side-effects.

It is due to this policy that the routine for Search is a procedure and not a function. As well as returning an index of the found element, the program allows for the situation where the item x is not in the array. To allow for this situation the attribute 'found' is introduced; when the item x is not found, the attribute 'found' is set to false.

```
class  BINARY_SEARCH[G -> COMPARABLE]
feature
        found : BOOLEAN
        index : INTEGER

        Is_Ordered(A:ARRAY[G], L,H:INTEGER):BOOLEAN is
                require         --Precondition
                        Pre : A /= void and L < = H and A.lower <= L  and H <= A.upper

                ensure          --Postcondition
                        -- (∀ i | L ≤ i < H : A@i ≤ A@(i+1))
                end -- Is_Ordered

        Search(A:ARRAY[G]; L, H : INTEGER; x:G) is
                require         --Precondition
                        Is_Ordered(A, L, H)

                ensure          --Postcondition
                -- (found → x = A@index)  &  (¬found →  x ∉  A[L .. H]
                end --Search

end -- BINARY_SEARCH
```

Comment: -- Is_Ordered
The precondition, Pre, checks for a void array and then ensures that the array has at least one item.

The postcondition is a comment as Eiffel's assertion language is limited; it does not have ∀ ('for all') or ∃ ('there exists').

For the procedure Search we use an auxilary recursive function Bin_Search_r, implementing the recursive algorithm for Binary Search.
i.e. to search an ordered array we check which half the item is in and then recursively search that half.

The specificaton for Bin_Search_r is

```
Bin_Search_r(A:ARRAY[STRING];  i,j : INTEGER; x:STRING):INTEGER is
        --      Binary Search the array segment A[i..j] for item x
        require
                                i  < j;
                Within:         A@i <= x and  x < A@(j)

        ensure
                A@result <= x and x < A@(result+1)
        end
```

Note:

Bin_Search_r assumes that A has more than one item.
Given the procedure, Bin_Search_r , we can write a procedure for Search as:

```
search (A: ARRAY [G]; L, H: INTEGER; x: G) is
        require
                ordered: is_ordered (A, L, H)
        do
                if  x < A.item (L) then
                        found := false ;
                        index := L - 1
                elseif  x > A.item (H) then
                        found := false ;
                        index := H
                elseif  equal (A.item (H), x) then
                        found := true ;
                        index := H
                else  -- A@L ≤ x < A@H
                        index := Bin_Search_r (A, L, H, x);
                        found := equal (A.item (index), x)
                end

        ensure

                (found → x = A@index)
        --      & (¬found →  x ∉ A[L..H])
        end  -- search
```

```
Bin_Search_r(A:ARRAY[STRING];  i,j : INTEGER; x:STRING) INTEGER is

        -- Binary Search the array segment A[i..j] for item x

        require
                                i < j;
                Within:         A@i <= x and  x < A@(j)

        local
                mid: INTEGER
        do
                if j > i+1 then
                        mid := (i+j)//2
                        if a.item(mid) <= x then
                                result := Bin_Search_r(A,mid,j,x)
                        else  -- a.item(mid) > x
                                result := Bin_Search_r(A,i,mid,x)
                        end
                else
                        result := i
        end
        ensure
                Post:  A@result <= x and x < A@(result+1)
        end --Bin_Search_r
```

<u>*Comment:*</u>

The function, Bin_Search_r, captures x so that

A@result $\leq$ x < A@(result+1).

We then have to check if x is actually in the array.

In the procedure, Search, we used

found := equal(A.item(index), x)

Alternative Version of Search

We can make the procedure for Search alot shorter by considering the following. We consider that A@(L-1) has the virtual value of -∞, and that A@(H+1) has the virtual value of +∞. Then

$$A@(L-1) \leq x < A@(H+1)$$

Note: We never access these values in the program.

Eiffel will not allow such a virtual value so let us restrict the precondition by dropping the 'within assertion'

Within: A@i <= x and x < A@(j)

The specification for Bin_Search_r is now

```
Bin_Search_r(A:ARRAY[G];  i,j : INTEGER; x: G) : INTEGER is
        -- Binary Search the array segment A[i..j] for item x
        require
                        i < j

        ensure
                        A@result <= x and x < A@(result+1) end
```

The implementation of Bin_Search_r remains exactly as it was and we now rewrite Search as:

```
Search (A: ARRAY [G]; L, H: INTEGER; x: G) is
        require
                Ordered:     Is_Ordered(A, L, H)

        do
                index := Bin_Search_r(A,L-1,H+1,x)
                found:=(index >= L) and then equal(A.item(index),x)

        ensure
        --{ (found  →  x = A@index)  &  (¬found → x ∉  A[L..H]}

        Found_It:     found  implies  equal(A@index,x)

        Failed:
                (not found) implies
                L-1 <= index and index <= H
                index = L-1 or else A@index < x
                index = H or else x < A@(index+1)

        --      (L-1 ≤ i ≤ H)
        --      &  (i ≠ L-1 → A@i ≤ x)
        --      &  (i ≠ H → x < A@(i+1))   --using 'i' for 'index'

end --Search
```

Comment:

Both versions of Search have the advantage that when the item x is not found the procedure indicates where to insert it, if desired, i.e.

$\neg$found $\rightarrow$ A@index < x < A@(index+1)

We expressed this in Eiffel using the assertion, Failed.

Iterative version of Binary Search

```
Bin_Search (A : ARRAY[G]; L,H : INTEGER;  x : G) : INTEGER is
        require
                        L < H
        local
                mid: INTEGER
        do
                from
                        i := L
                        j := H
                invariant
                        Within: A.item(i) <= x and x < A.item(j)
                        Range:        L<=i and i < j and j <=H
                until
                        j = i+1
                loop
                        mid := (i+j)//2
                        if a.item(mid) <= x then
                                i:=mid
                        else --A.item(mid) > x
                                j:= mid
                        end
                end
                result := i
        end
        ensure
                Got_It:        A.item(result) <= x and x < A.item(result+1)
        end -- Bin_Search
```

Final Version of Search

Combining the iterative function Binary Search with its calling procedure, Search, we get a new version of Search. In effect, the function, Binary Search, has been expanded 'in-line' in the procedure Search. The procedure Seach works for anays of size > 0.

```
Search(A:ARRAY[G]; L,H:INTEGER; x:G) is
        require
                Ordered : Is_Ordered(A,L,H)
        local
                i,j,mid: INTEGER
        do
                from
                        i := L-1
                        j := H+1
                until
                        j = i+1
                loop
                        mid := (i+j)//2
                        if a.item(mid) <= x then
                                i:=mid
                        else -- A.item(mid) > x
                                j:= mid
                        end
                end
                index := i
                found := L <= index and then equal(A.item(index), x)
        ensure
                Found_it: (found  implies  equal(A.item(index),x))
                Failed:         (not found implies
                                ((L-1 <= index and index <= H) and
                                (index = L-1 or else A.item(index) < x) and
                                (index = H or else x < A.item(index+1))))
        end -- Search
```

The Boolean Function, Is_Ordered

We use the technique of forcing termination of the loop when we find an item out of order.

```
Is_Ordered(A:ARRAY[G], L, H: INTEGER):BOOLEAN is
        -- Is A[L..H] ordered
        require
                Non_Trivial: A /= void and L < = H
                A.lower <= L  and H <= A.upper
        local
                i,j : INTEGER
        do
                from
                        i := L
                        j := H
                until
                        i = j
                loop
                        if A.item(i) <= A.item(i+1) then
                                i := i+1
                        else
                                j := i
                        end
                end
        Result  :=   i = H
        ensure
                -- (∀ i | L ≤ i < H : A@i ≤ A@(i+1))
        end—Is_Ordered
```

Another Version of Is_Ordered

As an alternative, we could use a boolean to force termination of the loop

```
Is_Ordered(A:ARRAY[G], L,H: INTEGER):BOOLEAN is
        require
                Non_Trivial: A /= void and L < = H
                A.lower <= L  and H <= A.upper
        local
                k : INTEGER
                b : BOOLEAN
        do
                from
                        k := L
                        b := false -- by default in Eiffel
                until
                        k = H or  b
                loop
                        b:= a.item(k) > a.item(k+1)
                        k := k+1
                end
                Result :=  not b
        ensure
                -- (∀ i | L ≤ i < H : A@i ≤ A@(i+1))
        end --Is_Ordered
```

A class BINARY_ROOT that will test Binary Search.

```
class
        BINARY_ROOT
creation
        make
feature


        make is
                local
                        bs : BINARY_SEARCH[STRING]
                        s :  STRING
                        A: ARRAY[STRING]
                do
                        !!bs
                        !!A.make (1,6)
                        A := << “Andy”,”Dick”,”Harry”,”John”,”Pat”,”Tom”>>
                        io.put_string(“%NLooking for which Name:”)
                        io.read_string
                        s := io.last_string
                        bs.Search(A, A.lower, A.upper,s)
                        io.put_string(s)
                        if bs.found then
                                io.put_string(“ was found at pos “)
                                io.put_integer(bs.index)
                        else
                                io.put_string(“ was not found: Index is “)
                                io.put_integer( bs.index)
                        end
                        io.readchar
                end --make


end—BINARY_ROOT
```

From Standish, T.A.
-- "Data Structures, Algorithms and Software Principles" p.181

```
Search(a:ARRAY[G]; L,H:INTEGER; x:G) is
        require
                Ordered : Is_Ordered(a,L,H)
        local
                i,j, mid : INTEGER
        do
                from
                        i := L-1
                        j := H+1
                        found := false
                        mid := (i+j)//2
                until
                        mid <= i or found
                loop
                        if equal(A.item(mid), x) then
                                index := mid
                                found := true
                        elseif A.item(mid) < x then
                                i := mid
                        else
                                j := mid
                        end
                        mid := (i+j)//2
                end
                if not found then
                        index := mid
                end
        end --Search
```

Correct & Incorrect Binary Search Programs

<u>*Exercise:*</u>

Which of the following 5 programs for Binary Search are correct/incorrect and why.

Assume that

A:ARRAY[REAL]

has n elements with indexing from 0 to N-1.

We are searching for an item x in the array; it may not be in the array.

Version 1:

```
from
        i := 0
        j := N
until
        j <= i +1
loop
        mid := (i+j) // 2
        if x >= A@mid then
                i := mid
        else
                j := mid
        end
end
found := (x = A@i)
```

Version 2:

```
from
        i := 0
        j := N-1
        found := false
until
        i >= j or found
loop
        mid := (i+j)//2
        if  A@mid < x  then
                i := mid +1
        elseif  A@mid = x then
                found := true
        elseif A@mid > x then
                j := mid - 1
        end
end
```

Version 4:

```
from
        i := 0
        j := N-1
        mid := (i+j)//2
until
        i > j
loop
        if  x >= A@mid  then
                i := mid-1
        end
        if  x <= A@mid then
                j := mid + 1
        end
        mid := (i+j) // 2
end
```

Version 3:

```
from
        i := 0
        j := N-1
        mid := (i+j)//2
until
        A@mid = x or i >= j
loop
        if A@mid < x then
                i := mid
        else
                j := mid
        end
        mid := (i+j)//2
end
```

Version 5:

```
from
        i := 0
        j := N-1
        mid := (i+j)//2
until
        i > j
loop
        if A@mid < x then
                i := mid
        else
                j := mid+1
        end
        mid := (i+j) // 2
end
```