2009 November 23

Example Test Questions for Assertions & Verification

1.

A Give a general template for refining an operation into a sequence and state what questions a designer must answer to verify the sequence is a correct refinement.

B Give a general template for refining an operation into a choice and state what questions a designer must answer to verify the choice is a correct refinement.

C Give a general template for redefining an operation into a loop and state what questions a designer must answer to verify the loop is a correct refinement.

2.

Explain what is meant by strong and weak assertions.

3.

There are 5 types of assertions that can be used in program design. Describe each type of assertion and how that assertion type is used in the design of programs.

4.

Give the best precondition, postcondition and loop invariant for the following algorithm to do a binary search on an array A[1 .. N]. Your work should use as much mathematical notation as possible to capture the relevant information.

```
low ← 1
high ← N
Result ← 0

while Result = 0 and low ≤ high do
    mid ← (low + high)/2
    if A[mid] = k then Result ← mid
    elseif A[mid] < k then low ← mid + 1
    else high ← mid - 1
    fi
end while
```

5.

What would be the best precondition, postcondition and loop invariant for the following algorithm to find a span of dash characters in an array.

```
charPointer := 1
loop exit when textLine(charPointer) not equal dashChar
  charPointer := charPointer + 1
end loop
```

6.

What would be the best precondition, postcondition and loop invariant for the following **correct** algorithm to find the maximum integer and its index in the array A[1..N].

```
max := 0  ;  maxIndex :=  0
j := 1
while j <= N do
  if max < A[j] then max = A[j] ; maxIndex := j end if
  j := j + 1
```

```
    end while
```

7.

Prove the following algorithm to sum the odd integers in the range 1 to N inclusive is correct. Clearly show the correspondence with the questions a designer must ask when verifying a loop is the correct refinement of an operation.

Precondition:

Postcondition:

Loop invariant:

```
sum := 0
p := 0
while p < N
    p := p + 1
    if odd(p) then sum := sum + p fi
end while
```

8.

What is a loop invariant? When is it used? How is it used? Why is it used? Where is it used?

9.

Given a loop, pre & post conditions and loop invariant prove the loop is correct or is incorrect.

10.

Given pre & post conditions and a loop invariant create the corresponding program text in Eiffel.

11.

What would be the best loop invariant and variant for the following algorithm.

```
require x = a * a

from
invariant ???
variant  ???
until x = 0
loop
    y := y + 1 ; x := x - 1
end

ensure y = a*a + old y
```

Using your loop invariant and variant, verify that the algorithm given above achieves the post-condition.

12.

Consider the following pseudo code which implements the *selection* sort. It works by looking through an array of n elements, A, picking the smallest element and moving it to the 1st position. It then, repeats the same thing on the sub-array, A[2..n], defined by positions 2…n, then, A[3..n], defined on positions 3..n and so on.

What is the best loop invariant for the outer loop? First find the loop invariant for the inner loop (lines 3-7) and then use this to find the invariant of the outer loop (lines 1-11).

Explain how the loop invariant can be used to prove the correctness of the main loop (lines 1-11)? (Hint: what is the relationship between A[1..k] and A[k+1..n] after kth iteration of the main loop?)

```
for i from 1 to n-1 do
  small = i
  for j from i+1 to n do
     if A[j] < A[small] then
       small = j
     end if
 end for
 temp := A[small]
 A[small] := A[i]
 A[i] := temp
end for
```

13.

What would be the best precondition, postcondition and loop invariant for the following correct algorithm to find the longest string in an array of strings **A[1..N]**

```
   s := null; longestIndex := 0
   j := 1
   while j <= N do
     if length(s) < A[j] then
           s := A[j]; longestIndex := j
     end-if
 end-while
```

14.

Give, both in English and in mathematical notation, the best precondition, postcondition, loop invariant and loop variant for the following algorithm to search the array `a[lb .. ub]` for the position of the value key. The Eiffel notation "`a @ j`" is equivalent to "`a[j]`" in Java/C++/C.

```
search(a : ARRAY[ITEM]; key : ITEM) : INTEGER is
  require  ???
  local j, n : INTEGER
  do
  from j := a.upperbound ; Result := a.lowerbound – 1
  invariant ???
  variant  ???
  until j = Result
  loop
    if a @ j = key then Result := j
    else j := j – 1
    end
  end
  ensure  ???
end
```

15.

The following code segment computes the square of n

```
square := 0
increment := 1
for  i:= 1 to n do
      square := square + increment
      increment := increment + 2
end for
```

A What are the best loop invariant and variant.

B Use the invariant and the variant from part A to argue the correctness of the loop.

C Explain why an invariant such as "i >= 1" is not of much value

16.

The greatest common divisor (GCD) of two positive integers, a and b, denoted by GCD(a, b) is the largest natural number that divides both a and b. Some examples include GCD(9,6) = 3, GCD(16, 5) = 1. Euclid's algorithm computes the GCD of two numbers as follows

Step A: Write a = q *b + r where 0 <= r <= b.
Step B: If r > 0, then set a = b, b = r and go to Step A. Otherwise last b is the GCD

The equation . a = q *b + r implies that GCD(a, b) = GCD(b, r) and hence the process works.
Here are several iterations:

a = q1*b + r1 where 0 <= r1 <= b
b = q2*r1 + r2 where 0 <= r2 < r1
r1 = q3*r2 + r3 where 0 <= r3 < r2

Complete *require, ensure, variant* and *invariant* clauses for the gcd method.

```
gcd (a, b: INTEGER): INTEGER is
            -- Greatest common divisor of a and b.
      require
            ???
      local
            x, y, remainder: INTEGER
      do
          from
            x := a
            y := b
            remainder := x \\ y  -- remainder of x divided by y
          invariant
                  ???
          variant
                  ???
          until
            remainder = 0
          loop
            x := y
            y := remainder
            remainder := x \\ y
          end
          Result := y
      ensure
            ???
end
```

17.

Verify that the following algorithm to partition an array, `a[1 .. N]`, into those elements that are smaller than `a[1]` and those that are larger than `a[1]` is correct.

Array on exit	elements of a < old a[1]	old a[1]	elements of a > old a[1]

```
partition(a : ARRAY[INTEGER]): INTEGER is
  require a.lower_bound ≤ 1  ∧  a.upper_bound ≥ 1
          ∀ i : 2 .. a.upper_bound • a[1] ≠ a[i]
  local split, last : INTEGER
  do
    from split := 1 ; last := 2
    invariant ∀ k : 2 .. split • a[k] < a[1]  ∧  ∀ k : split+1 .. last-1 • a[k] > a[1]
     variant a.upperbound – last + 1
     until last > a.upper_bound do
       if a[last] < a[1] then split := split + 1
                              swap(a[last], a[split])
                              last := last + 1
       else last := last + 1
       end
    end
    swap(a[1], a[split])
    Result := split
  ensure  a[Result] = old a[1]
          ∀ k : 1 .. Result-1 • a[k] < a[Result]
          ∀ k : Result+1 .. a.upper_bound • a[Result] < a[k]
  end
```

18.

The following routine correctly computes $2^n - 1$ for a given integer n. Determine the loop invariant and variant and write it in the space provided within the program text. Note that both assertions have to be Eiffel executable.

```
pow_2 (n : INTEGER) : INTEGER is
    -- result is 2^n - 1
  require n >= 0
  local i, increment: INTEGER
  do
    from
        i := 0
        increment := 1
        Result := 0
    invariant ???
    variant ???
    until i >= n
    loop
        Result := Result + increment
        increment := increment * 2
        i := i + 1
    end
  ensure Result = 2 ^ n - 1
  end
```

19.

The following routine correctly computes the 2^n – 1 for a given integer n. 2^n is 2 to the power of n. Prove the algorithm is correct.

```
pow_2 (n : INTEGER) : INTEGER is
    -- result is 2^n - 1
  require n >= 0
  local i, increment: INTEGER
  do
    from
        i := 0
        increment := 1
        Result := 0
    invariant  Result = 2^i – 1 and increment = 2 ^ i
    variant  n - i
    until i >= n
    loop
```

```
            Result := Result + increment
            increment := increment * 2
            i := i + 1
        end
        ensure Result = 2 ^ n - 1
      end
```

20.

The following routine employs a loop to compute n*(n+1) for a given non-negative integer n. There is, however, an error in its implementation. Your task is to use the given loop invariant and the variant to prove the "correctness" of the loop. If you are careful with your proof, then the source of the error should reveal itself at the proper step of the proof. Follow the normal course of proof until you reach an inconsistency, then indicate what is the source of the error in the routine that gives the inconsistency, correct it and complete the proof.

```
sum2 (n : INTEGER) : INTEGER is
    -- Compute n * (n + 1)
  require n >= 0
  local i : INTEGER
  do
    from i := 0
    invariant Result = i * (i + 1)
    variant n - i + 1
    until i >  n
    loop
        i := i + 1
        Result := Result + 2 * i
    end
  ensure Result = n * (n + 1)
  end
```

21.

The following routine employs a loop to compute n^2 (n^2) for a given non-negative integer n. Verify the routine is correct. There is, however, an error in its implementation. Your task is to use the given loop invariant and the variant to prove the "correctness" of the loop. If you are careful with your proof, then the source of the error should reveal itself at the proper step of the proof. Follow the normal course of proof until you reach an inconsistency, then indicate what is the source of the error in the routine that gives the inconsistency, correct it and complete the proof.

```
square (n: INTEGER): INTEGER is
  -- result is n^2
  require n >= 0
  local i, increment: INTEGER
  do
    from i := 0 ; increment := -1
    invariant Result = i^2 and increment = 2*i - 1
```

22.
```
   until i > n
   loop
    i := i + 1
    increment := 2 * i - 1
    Result := Result + increment
   end
   ensure Result = n ^ 2
  end
```

23.

Consider the following pseudo code, which implements the *selection* sort. It works by looking through an array of n elements, A, picking the smallest element and moving it to the 1st position. It then, repeats the same thing on the sub-array, A[2..n], defined by positions 2…n, then, A[3..n], defined on positions 3..n and so on.

```
1      i := 1
2      while i <= n do {
3          small := i
4          j = i + 1
5          while j <= n do {
6              if A[j] < A[small] then small := j fi
7              j := j + 1
8              }
9      temp := A[small]
10     A[small] := A[i]
11     A[i] := temp
12     i := i + 1
13     }
```

A What is the best loop invariant for the inner loop (lines 5..8)?

B What is the best loop invariant for the outer loop (lines 2-13)?

C Use the inner loop invariant to show the correctness of the inner loop (lines 3..8).

24.

A positive integer n is said to be *prime* if it is only divisible by 1 and itself.

$\forall$ i : INTEGER • i | n $\Rightarrow$ (i = 1 $\vee$ i = n)   where i | n is read as i divides n

The following Eiffel routine correctly tests if a given integer, n, is prime by verifying that none of the integers 2, 3, …, floor(sqrt(n)) divides n. For this question you need to perform the following tasks.

a) Complete the postcondition for the function prime.
b) Write a proper loop invariant, using BON assertion language, and variant for the loop.
c) Use results of the previous two steps to prove the correctness of the loop.

```
prime (n: INTEGER): BOOLEAN is
     -- Asserts that n a prime number
  local
     i   : INTEGER  -- Test divisor
     max : INTEGER  -- Maximum integer to try
     do
       from
         i := 2
         max := floor (sqrt (n))
         Result := True
       until
         i > max  or  not Result
       loop
```

```
            Result := not ((n \\ i) = 0)   -- \\ is the mod function
            i := i + 1
          end
        ensure  ???

    end
```

Loop invariant
Write the loop invariant, using BON assertion notation **not** Eiffel programming language, and loop variant.
Loop variant ???

25.

The following algorithm multiplies two integers using addition. Prove the algorithm works correctly. The more mathematically precise you are, the higher the grade.

Pre-condition:	$a \neq 0$ and $b \neq 0$
Post-condition:	$z = abs(a*b)$ – abs is the absolute value, $z > 0$
Loop invariant:	$z + u*y = x*y$

```
if a < 0 then x := –a else x := a
if b < 0 then y := -b else y := b
z := 0
u := x
repeat
  z := z + y
  u := u – 1
until u = 0
```

26.

Give, both in English and in mathematical notation, the best precondition, postcondition, loop invariant and loop variant for the following algorithm to search the array `a[lb .. ub]` for the position of the value key. The Eiffel notation "`a @ j`" is equivalent to "`a[j]`" in Java/C++/C.

```
search(a : ARRAY[ITEM]; key : ITEM) : INTEGER is
  require ???
  local j, n : INTEGER
  do
  from j := a.upperbound ; Result := a.lowerbound – 1
  invariant ???
  variant ???
  until j = Result
  loop
    if a @ j = key then Result := j
    else j := j – 1
    end
  end
  ensure ???
end
```