Computer Science 1030.03 Fall 1997 Midterm Test

SOLUTIONS

1. The OOT built-in *string* type includes a way to declare variables of type string, and a number of common string operations, e.g., comparison of two strings, converting strings to integers, etcetera. Strings are of maximum size 255 characters. A string in OOT might be declared as follows.

```
var name : string
name := "Oscar the Grouch"
```

(The quotes are not part of the string.)

A *substring* is part of another string. In OOT, to select the substring of string `S` starting at position `i` and up to position `j`, the syntax `S(i..j)` is used. For example, to select the substring `Oscar` from `name`, above, we would write `name(1..5)`.

Write a small set of test cases for the OOT substring operation `S(i..j)`. That is, write down a set of values for `S`, `i`, and `j` that will thoroughly test the correct functioning of the substring operation. For each item in the set of test data, explain what specific part of substring it is intended to test. **NOTE:** you should need no more than 7-10 test cases.

```
Let valid(i,j) = i <= j.
Let empty(S) be true iff S is the empty string.

Test cases:

1. valid(i,j) and empty(S):  returns empty string.
2. not valid(i,j) and (empty(S) or not empty(S)): error message.
3. valid(i,j) and length(S)=255 (i=1,j=255): maximum sized string.
4. i=1 and j=length(S): whole maximal substring.
5. i<1 and any S: out of left subrange.
6. j>length(S) and any S: out of right subrange.
7. i=j and any S: unit-length substring.
8. general case: valid(i,j) and any non-maximal, non-empty S.
```


2. In Assignment 1, you implemented *sets of integers* as an abstract data type, using OOT modules and classes. Sets could contain only one copy of each element; that is, duplicate integers could not be entered into the set.

A *bag* is an extension of a set. The main difference between bags and sets is that a bag can hold more than one copy of an element. Bags are unsorted, like sets.

You are to write three small OOT functions or procedures that implement basic bag operations. The following type definition should be used.

```
type Bag :
  record
    elements : array 1..MaxSize of elementType
    count : array 1..MaxSize of int
  end record

var numElements : int
var theBag : Bag
```

Each element to be inserted into `theBag` is represented *only once* in the array `elements`. Instead of having multiple copies of an element in the array, a count of the number of occurrences of an element is stored. `count(i)` is the number of occurrences of the element `elements(i)` in the bag.

Write OOT code for the following bag operations. Each operation should be implemented as either a function or a procedure.

(a) *Initialize:* initializes the bag before any elements are added. It must be called before any other operations are applied to a bag. This operation sets `theBag` and `numElements` to suitable starting values.

```
procedure Initialize()
  numElements := 0
  % Initializing theBag is unnecessary.
end Initialize
```

(b) *Insert(x):* inserts x to the bag. If x is already in the bag, the number of occurrences for x is incremented by one.

```
procedure Insert( x : elementType )
  var i : int := 1
  loop exit when i>numElements or theBag.elements(i)=x
    i+=1
  end loop
  if i<= numElements then
    theBag.count(i) += 1
  else
    if numElements < MaxSize then
      numElements += 1
      theBag.elements(numElements) := x
      theBag.count(numElements) := 1
    else
      put "The bag is full. Insert failed."
    end if
  end if
end Insert
```

(c) *IsElement(x):* returns the number of occurrences of x in the bag.

```
procedure IsElement( x : elementType )
  var i : int := 1
  loop exit when i>numElements or theBag.elements(i)=x
    i+=1
  end loop
  if i <= numElements then
    result theBag.count(i)
  else result 0
  end if
end IsElement
```


3. Consider the following type and variable for creating a linked list of integers.

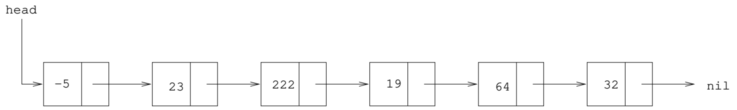
```
type nodetype :
  record
    data : int
    next : ^nodetype
  end record

var head : ^nodetype
```

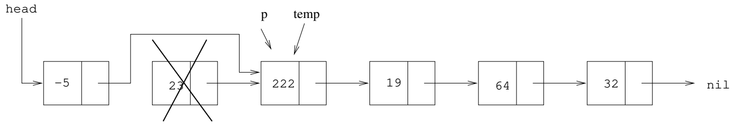
The following code fragment operates on a linked list that is pointed to by `head`.

```
var n : ^nodetype
n := head
loop
  exit when n=nil
  var temp : ^nodetype
  temp := n->next
  if temp not= nil then
    temp := temp->next
  end if
  free n->next             % Free one node
  n->next := temp
  n := n->next
end loop
```

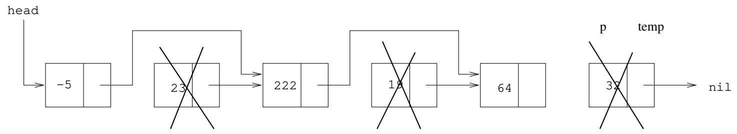
(a) Suppose that `head` points to the following linked list.



Next, suppose that the code above is executed. Draw a picture below of the linked list and the variables `n`, `head`, and `temp` after the *first* iteration of the `loop`.



(b) Draw a picture below of the linked list and the variables `head, n`, and `temp` after the code has terminated.



(c) Write an appropriate comment, and a pre- and postcondition that could go above this block of code.

```
% pre: true
% post: removes and frees every second element of the list
%       pointed to by head, starting with element 2.

% This program removes and frees every second element of the
% list pointed to by head.
```


4. The following program fragment reverses the order of elements in an array `A`.

```
% pre: upper(A) >= 1
i := 1
j := upper(A)
loop
  exit when i >= j
  % Swap elements at positions i and j
  t := A(i)
  A(i) := A(j)
  A(j) := t
  i := i+1
  j := j-1
end loop
```

(a) The statement $i \leq j$ is not a loop invariant for this code. Explain why not.

```
If upper(A)=2 then at the beginning of the second iteration,
i=2 and j=1, so i<=j is false.
```

(b) The statement $A(i) < A(j)$ is not a loop invariant for this code. Explain why not.

```
Nothing has been assumed about the values of the array A.
So we don't know if A(i) < A(j) at any time.
```

(c) Write a statement that *is* a loop invariant for this code, and which mentions at least one of the variables `i, j`, and `A`.

```
There are many decent answers:

  i>=1,  j>=1,  i<=j+1,  j<=upper(A),  i<=upper(A),
  i+j=upper(A)+1
```

5. Consider the following OOT function. (**Note:** ^ is the exponentiation operator.)

```
% <pre:  p>=0 and n has a value >
function pow( n,p : int ) : int
  var answer : int := 1
  var count  : int := 0
  loop
    <INV: n^count = answer >
    exit when count=p
    answer := answer*n
    count += 1
  end loop
  result answer
end pow
% <post:  answer = n^p >
```

(a) Trace a call to `pow(2,4)` and fill out the table below to show the sequence of values taken by the variables.

n	p	answer	count
2	4	1	0
2	4	2	1
2	4	4	2
2	4	8	3
2	4	16	4

(b) To prove termination of a loop, we can define a loop variant that decreases on each iteration, yet can never be less than zero. Give a *decreasing* loop variant for the loop in `pow`.

```
count is a loop variant (bounded by p) but is increasing.
p-count  is a decreasing loop variant (bounded by 0).
```

(c) Prove the partial correctness of `pow`. That is, show that:

If the preconditions are true, the code is executed, and the code terminates, then the postconditions will be true.

You should use the three proof rules for loops given in class, and the invariant `INV` given in the code above. Show all your steps!


```
Follow the checklist given in class, ie., show that the 3 proof steps
hold for this loop.

1.  preloop => INV

    (answer=1 and count=0) => (answer=n^count)

    which is trivial, substituting 1 and 0 for answer and count.

2.  INV and b => postloop

    (answer=n^count and count=p) => answer=n^p

    which is trivial, substituting p for count.

3.  INV and not b and codeA => INV

    (answer=n^count and count<p and answer*=n and count+=1) => INV

    Substitute RHS of answer and count in the RHS of the implication,
    giving us:

    (answer=n^count and count<p) => answer*n=n^(count+1)

    But n^(count+1) = n^count*n^1, so cancel n from each side, and
    the implication is proven.

So the loop is partially correct. Now to prove that pow is correct,
we must show that

pre and powcode => post

p>=0 and n has a value and powcode => answer=n^p

But we know that powcode establishes answer=n^p, so the result holds.
```