University of Toronto
Faculty of Applied Science and Engineering
FINAL EXAMINATION, December 1996
First Year - Program 5
CSC181F Introduction to Computer Programming
Examiner - D.B. Wortman

One official 8 1/2 x 11 inch Aid Sheet permitted.
Non programable calculators permitted.
ANSWER ALL QUESTIONS. 7 questions, 100 marks total. 2 ½ hours.
WRITE LEGIBLY; unreadable answers are not answers.
If you need to make any assumptions in order to answer a question be sure to state those assumptions clearly in your test book.

DON'T PANIC KEEP COOL DON'T PANIC

1. [10 marks] Assume that the data structures shown below were used to implement Polynomials in Assignment 3. Assume that the nodes for a polynomial are kept in order from largest exponent to smallest and that a term with exponent of zero is the only one that can have a coefficient of zero.

```
type polyNode :
    record
        coeff : real      /* coefficient */
        expon : int       /* exponent */
        next  : pointer to polyNode
    end record
type PolyType : pointer to polyNode
```

   Write a Turing procedure with the header:

   function evalPoly(poly : PolyType, atVal : **real**) : **real**

   This function computes the value of its polynomial argument *poly* at the point *atVal* and returns this value as its result.

2. [10 marks] Write a function *mostFrequent* with the header:

   function mostFrequent(T : **string**(*)) : **string**(1)

   Assume (without checking) that the string T contains ONLY upper and lower case letters (i.e. "a" .. "z" and "A" .. "Z"). The function mostFrequent finds the letter that occurs most frequently in T and returns that letter as its result. If several letters tie for being the most frequent letter in T then mostFrequent returns any one of them. Examples:

```
mostFrequent( "" )                -> ""
mostFrequent( "A" )               -> "A"
mostFrequent( "AbbAbbA" )         -> "b"
mostFrequent( "XYzXYzXYzXYz" )    -> "X"  (or "Y" or "z")
```

3. [15 marks] A CSC180F student who slept through a lot of classes wrote the following procedure to (correctly) replace one element in a linked list.

```
type  listNode : record
                     val : int
                     next: pointer to listNode
                 end record
type  listPtr : pointer to listNode

procedure replace( var L : listPtr, A,B : int )
    % Replace one occurrence of A with B in list L
    var p       : listPtr     := L
    var q: listPtr     := nil
    loop        % find node to replace
        exit when p = nil
        if p->val = A then
            q := p
        end if
        p := p->next
    end loop
    if q = nil then
        return          % nothing to replace
    end if
    var temp : listPtr
    new temp            % make new node
    temp->val := B
    if q->next = nil then  % link next node
      temp->next := nil    % if there is one
    else
      temp->next := q->next
    end if
    if q = L then       % insert new node in list
        L := temp
    else
        p := L          % find place to insert
        loop                    % find place to insert
            exit when p->next = q      % must succeed
            p := p->next
        end loop
        p->next := temp        % link in node
    end if
    % free old node
    free  q
end replace
```

What changes would you make in this procedure to make it more efficient?

4. [15 marks] What is the printed output from the Turing program shown below?

```
function warped( S, P : string(*)) : string
    assert length(S) > 0
    if length(P) = 0 then
        result S
    elsif index( P, S(length(S)) )not= 0 then
        result warped( S( 1 .. * - 1), P )
    else
        var T : string := S
        loop
            const J : int := index( T, P(1) )
            exit when J = 0
            T := T( 1 .. J - 1 ) + T( J + 1 .. * )
        end loop
        result  warped( T, P( 2 .. * ) )
    end if
end warped
put warped("XUUFoUdOdRXUTdYZ df TdUWqoOUp", "XJKZfopqdU")
```

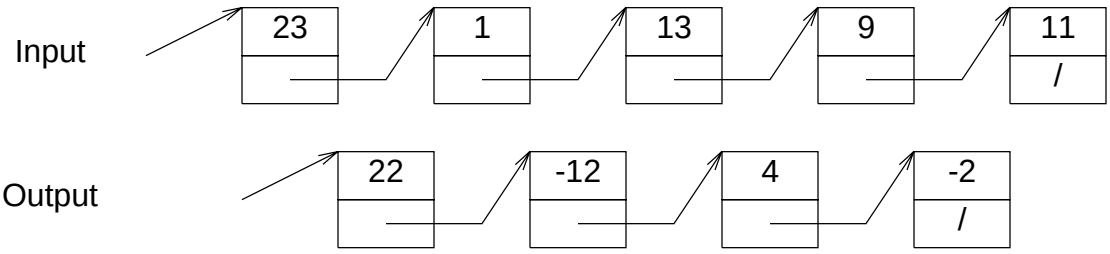
5. [15 marks] Assume the following declarations are used to create singly linked lists in C.

```
struct listNode {
    int  val ;
    struct listNode * next ;
  } ;
typedef struct listNode * listPtr ;
```

Write a C function with the header

```
listPtr deltaList( listPtr inList )  ;
```

This function computes the difference between the values of successive nodes of its argument *inList*. It returns NULL if *inList* is NULL or if *inList* contains only one node. Otherwise it returns a list of the differences. Example:



6. [5 marks] The first Gift Assumption in Assignment 5 states:

    1. You may assume that for all data you process, the *first* and *last* data records are for a filling of the car's gasoline tank (i.e. the full flag has the value 1).
    You may assume that the *window-size* command argument is >= 1.

Describe (in words) how your solution to Assignment 5 would have to be modified if this Gift Assumption was withdrawn.

7. [30 marks] For this question you are to write a collection of C functions that simulate the storage lockers often found in bookstores, railway stations and bus stations. Headers for the functions are given below.

```
typedef   ???   ObjectType  ;
```

This type defines the objects that are stored in lockers. *IT IS SUPPLIED TO YOU* by the owner of the lockers. Your functions should work for any reasonable definition for ObjectType.

```
typedef   ???   KeyType  ;
```

KeyType is the type used to represent keys to individual lockers.

```
KeyType checkIn( ObjectType object ) ;
```

The function *checkIn* takes an object and stores it in a locker. The key that it returns can later be used (once) to retrieve the object that was stored in the locker.

```
ObjectType checkOut( KeyType key ) ;
```

The procedure *checkOut* can be used to retrieve an object that was previously stored in a locker. Checking an object out of a locker resets the locker to empty and makes it available for reuse.

```
int inspect( ObjectType * object, int masterKey ) ;
```

The procedure *inspect*, allows management to inspect (but not change) the contents of all lockers. Successive calls to Inspect deliver the contents of the lockers to the parameter *object* in some arbitrary order one at a time. The function returns TRUE if there are more lockers to inspect and FALSE when all lockers have been inspected. The next call to *inspect* will start all over again. you may assume that no checkIns or checkOuts will occur during an inspection and that only one inspection will be done at a time. The user of Inspect must provide a password in the parameter *masterKey* to prove that she or he is really management.

You may assume initially that lockers are used only by *Boy Scouts*, and *Girl Guides*, i.e. they always give checkOut a valid key and never try to check an object out more than once. However Guides and Scouts come in large troops so your locker should be able to handle arbitrarily many objects.

To answer this question:

a) [7 marks] design the internal data structures and types used by the locker module. Describe any assumptions that you make about ObjectType. Describe your data structures in C.

b) [18 marks] write the functions checkIn, checkOut and Inspect in C.

c) [5 marks] describe (IN WORDS) how you would extend your answer to a) and b) if the Boy Scouts and Girl Guides were replaced by **Bad Dudes** who try to use forged keys or to reuse the same key. A secure locker system should resist such attempts. You should NOT rewrite your programs to answer this question.