CSE 428, Spring 2001. Midterm 2, 5 April 2001

The exam consists of 5 exercises, totalling 100 points. Write your answers directly on these pages. **Please remember to write your name, Student ID, and section number (Section 2 for the morning section and Section 1 for the afternoon section) at the top of each sheet.**

Exercise 1 (20 pts)

Consider the following ML declaration of a datatype "bynary tree with integer labels":

```
datatype btree = leaf of int | node of btree * int * btree;
```

Define a function `minmax: btree -> int * int` which, given tree `t`, returns a pair `(m,n)` such that `m` is the minimum of `t`, and `n` is the maximum of `t`. For example, if `t1` is the tree represented in the figure of Exercise 2, then `minmax t1` should return the pair `(1,6)`.

In order to write a more elegant solution, you may find it convenient to use the following auxiliary functions:

```
fun min3(x,y,z) = (* minimum of x, y, and z *)
    if x < y
       then if x < z then x else z
       else if y < z then y else z;

fun max3(x,y,z) = (* maximum of x, y, and z *)
    if x < y
       then if y < z then z else y
       else if x < z then z else x;
```

Please do not use any other auxiliary functions. This restriction will ensure that your solution visits each node only once, and hence it has complexity $O(n)$ (where n is the number of nodes). Solutions that use other auxiliary functions (either local or external) will get, at most, 12 points for this exercise.

Note: Please write your solution in ML. Solutions written using an imperative notation will not be considered.

Exercise 2 (20 pts)

Consider the following ML declaration of a polimorphic datatype "bynary tree":

```
datatype 'a btree = leaf of 'a | node of 'a btree * 'a * 'a btree;
```

Some instances of this datatype are:

```
      t1                t2                  t3                  t4

      1              [1,2,3]               true               (1,2)
     / \              /     \              /  \                / \
    2   3          [4]    [5,5]        false  true        (3,4) (5,6)
   /   / \         / \                   /      \            /
  4   5   6      [6] [7]              true     false     (7,8)
```

Consider now the following extension of the function "map" to binary trees:

```
fun map f (leaf x) = leaf (f x)
  | map f (node(t1,x,t2)) = node(map f t1, f x, map f t2);
```

Say what is the result of each of the following expressions, including error, where `t1`, `t2`, `t3` and `t4` are the trees represented in the figure above. Remember that `+` is the plus operator on numbers, `>` is the comparison operator "greater than" on numbers, `@` is the function "append" on lists, `hd` is the function "head" on lists, `tl` is the function "tail" on lists, `not` is the negation operator on booleans, and `op` takes an infix operator and transforms it into its prefix version. Finally, `height` is the function which gives the height of a tree, defined in the following way:

```
fun height leaf _ = 1
  | height (node(t1,x,t2)) = let val h1 = height t1
                                 val h2 = height t2
                              in 1 + (if h1 > h2 then h1 else h2)
                             end;
```

1. (2 pts) `map (fn x => x + 1) t1;`

2. (3 pts) `map (fn x => height t1 ) t1;`

3. (3 pts) `map (fn x => hd x = tl x) t2;`

4. (3 pts) `map (fn x => x @ x) t2;`

5. (3 pts) `map not t3;`

6. (3 pts) `map (fn (x,y) => x > y) t4;`

7. (3 pts) `map (op +) t4;`

Exercise 3 (20 pts)

Consider the following definition of a class Stack in C++

```
class Stack {

  private:
     class Element {
     public:
        int info;
        Element* next;
        Element(int k, Element* n) { info = k; next = n; }
     };

     Element* head;

  public:
     Stack() { head = NULL; }
     ~Stack() { while (head!=NULL) pop(); }
     void push(int k) { head = new Element(k,head); }
     int pop() {
        if (head==NULL) { cout << "Empty stack!" ; return 0; }
        Element* temp = head;
        int n = head->info;
        head = head->next;
        delete temp;
        return n;
     }
  };
```

For each of the following declarations of `foo()`, say whether a call to `foo()` leaves memory leaks and/or dangling references. (Note: "a call to `foo()` leaves ml/dr" means "after a call to `foo()` returns, we have ml/dr".) For each dangling reference, specify whether it is to the stack or to the heap.

Assume that `t` is a global variable of type `Stack*` which before the call of `foo()` has value `NULL`.

1. `int foo(){ Stack s; s.push(2); s.push(3); return s.pop(); }`

2. `int foo(){ Stack* s = new Stack(); s->push(2); return s->pop(); }`

3. `void foo(){ Stack* s = new Stack(); s->push(2); delete s; }`

4. `void foo(){ Stack* s1 = new Stack(); t = s1; s1->push(2); delete s1; }`

5. `int foo(){ Stack s1; Stack* s2 = &s1; s1.push(2); return s1.pop(); }`

Exercise 4 (20 pts)

Given previous definition of a class Stack in C++, define an analogous class Stack in Java supporting multithread use (i.e. such that the same stack can be shared by several threads). Make sure that, when `pop()` is executed on a empty stack, the thread executing the `pop()` suspends, instead of printing an error message. The thread should be resumed when some other thread pushes an element in the stack. You don't need to worry about Interrupted Exceptions.

Exercise 5 (20 pts)

Consider the following Java program, where "Stack" is the Java class Stack of previous exercise:

```
import Stack; // import the definition of the class Stack

class MyThread extends Thread{
   private Stack myStack;
   private string myName;

   public MyThread(Stack s, string n){ myStack = s; myName = n; }

   public void run(){
      sleep((long)(java.lang.Math.random()*1000)); // sleep for an arbitrary amount of time
      myStack.push(myName); System.out.print("in " + myName + ", ");
      sleep((long)(java.lang.Math.random()*1000)); // sleep for an arbitrary amount of time
      System.out.print("out " + myStack.pop() + ", ");
   }
}
class Test {
   public static void main(String[] args){
      Stack s = new Stack();
      MyThread T1 = new MyThread(s,"A");
      MyThread T2 = new MyThread(s,"B");
      T1.start(); T2.start();
   }
}
```

1. (10 pts) Show all the possible output sequences that this program can produce (in different executions).

2. (10 pts) Is it possible that the program produces the "wrong" output "`in A, out B, in B, out A`"? ("wrong" in the sense that it would look like as `s` did not behave like a stack, because in this sequence `B` is extracted before being inserted). In case of negative answer, please explain why it is not possible. In case of positive answer, please discuss a way to enforce that only stack-like sequences be printed, without modifying `main()`, and in such a way that each thread performs the same operations as before (push(myName), print(myName), pop(), print the value popped).