Name:____

CSE 341, Spring 2011, Midterm Examination
27 April 2011

Please do not turn the page until everyone is ready.

Rules:

- The exam is closed-book, closed-note, except for one side of one 8.5×11 in. piece of paper.
- **Please stop promptly at 1:20.**
- You can rip apart the pages, but please staple them back together before you leave.
- There are **60 points** total, distributed **unevenly** among **5** questions (most with multiple parts).
- When writing code, style matters, but don't worry about indentation.

Advice:

- Read questions carefully. Understand a question before you start writing.
- **Write down thoughts and intermediate steps so you can get partial credit.**
- The questions are not necessarily in order of difficulty. **Skip around.**
- If you have questions, ask.
- Relax. You are here to learn.

Name:____

1. This problem uses this datatype definition for a binary tree:

```
datatype ('a, 'b) Tree =
         Leaf of 'a
       | Node of ('b * ('a,'b) Tree * ('a,'b) Tree)
```

   (a) (**5** points) Write a function `sum` that returns the sum of the values stored in the `Leaf` nodes of a `Tree`. Your function only needs to work correctly on trees whose `Leaf` nodes contain integer values.

   (b) (**8** points) Write a function `height` that returns the height of a `Tree`. For the purposes of this question, the height of a tree consisting of only a `Leaf` node is 0; the height of any other tree is 1 greater than the height of its tallest subtree.

   For full credit, your solution may not use any additional functions (either defined by you or in the standard SML library), and may not compute the height(s) of any subtrees more than once. (Hint: `let` may be your friend.)

   (c) (**2** points) What are the types of functions `sum` and `height` from parts (a) and (b)?

Solution:

(a)
```
fun sum tree =
    case tree of
        Leaf n => n
      | Node(_,l,r) => (sum l) + (sum r)
```

(b)
```
fun height tree =
    case tree of
        Leaf _ => 0
      | Node(_,l,r) =>
        let
            val lheight = 1 + height l
            val rheight = 1 + height r
        in
            if lheight > rheight then lheight else rheight
        end
```

(c)
```
sum : (int, 'a) Tree -> int
height : ('a, 'b) Tree -> int
```

Name:____

2. (**8** points) Write a tail-recursive function `fib n` that returns a list of the first n Fibonocci numbers for $n > 0$. For example,

```
fib 8 = [0,1,1,2,3,5,8,13]
fib 1 = [0]
```

(Recall that the Fibonocci numbers are the series beginning with 0 and 1, and each successive number is the sum of the previous two.)

For full credit the number of tail-recursive calls made by your function should be linear in n. However, you do not need to worry about the costs of accumulating the answers in a list (i.e, don't worry about the costs of the list append operations).

Solution:

```
fun fib n =
    let
        fun fibaux(acc,n,curr,next) =
            if n = 0
            then acc
            else fibaux(acc@[curr],n-1,next,curr+next)
    in
        fibaux([],n,0,1)
    end
```

Name:____

3. For each of the following programs, give the value that `ans` is bound to after evaluation.

   (a) (**3** points)

   ```
   val x = 1
   val y = 2
   val f = fn y => x + y
   val x = 5
   val y = 10
   val ans = f x + y
   ```

   (b) (**3** points)

   ```
   val x = 1
   val y = 2
   val z = 3
   fun f x = let val x = 10 in fn y => x+y+z end
   val y = 2
   val ans = f y 7
   ```

   (c) (**3** points)

   ```
   val q = fn x => List.map tl x
   val ans = q [[1,2,3],[4,5]]
   ```

   Solution:

   (a) `16`

   (b) `20`

   (c) `[[2,3],[5]]`

Name:____

4. One of your colleagues has been insisting that a language without a "real" `for`-loop must be useless. You have decided to settle the argument by showing that an equivalent loop can be written in SML.

   (a) (**9** points) Write a function named `for` with parameters `loopvarinit test incr body` and `accinit` that has the same effect as the following loop in one of those lesser languages:

   ```
   acc = accinit;
   for ( loopvar = loopvarinit; test(loopvar); loopvar = incr(loopvar) ) {
     acc = body(loopvar, acc);
   }
   yield acc;  /* i.e., the final value of acc is the "value" of the loop */
   ```

   In the above description, `loopvar` and `acc` are additional values that are not parameters of `for`. You will want to have similar values in your solution. Your solution must be properly tail-recursive and may not use any other functions defined outside of function `for`.

   ```
   fun for loopvarinit test incr body accinit =
   ```

   (b) (**5** points) Complete the following function so that it uses function `for` to compute the sum of the numbers 1 through n. You should supply appropriate values (integers and anonymous functions) for the parameters of `for`.

   ```
   fun sum(n: int) =

       for ____________________________________________

           ____________________________________________

           ____________________________________________

           ____________________________________________

           ____________________________________________
   ```

Solution:

(a)
```
fun for loopvarinit test incr body accinit =
    let
        fun loop(loopvar, acc) =
            if test loopvar
            then loop(incr loopvar, body(loopvar,acc))
            else acc
    in
        loop(loopvarinit, accinit)
    end
```

(b)
```
for 1 (fn x => x <= n) (fn n => n+1) (fn (x,y) => x+y) 0;
```

Name:____

5. Suppose we are implementing a dictionary (collection) of words. Version 1.0 of the software uses this SML structure definition:

```
structure D :> DICT =
struct
  datatype dictionary = None | Cons of string * dictionary

  fun member(s,dict) =
    case dict of
        None => false
      | Cons(str,rest) => s=str orelse member(s,rest)

  fun add(s,dict) = if member(s,dict) then dict else Cons(s,dict)
end
```

Now suppose that in version 2.0 of the software we want to replace that structure with this new one, where the `dictionary` datatype in version 1.0 is replaced with a regular SML `string list`.

```
structure D :> DICT =
struct
  type dictionary = string list
  ... (* see part a *)
end
```

(a) (**8** points) Write the necessary function bindings to complete version 2.0 of the structure so that it has the same functionality as the version 1.0 structure.

Hint: Be sure that your new version of the structure includes sufficient operations so that client code can construct new dictionaries and process them.

Write your answer on the next page.

(b) (**6** points) Complete this signature so that *both* version 1.0 and version 2.0 of the structure will type-check without errors. Use one abstract type definition and any necessary `val` bindings.

```
signature DICT =
sig
  ...
end
```

Write your answer on the next page.

Name:____

Answers for question 5 can be written here.

Solution:

(a)
```
structure D :> DICT =
struct
type dictionary = string list

val None = [];
fun member(s,dict) =
    case dict of
        [] => false
      | x::xs => x=s orelse member(s,xs)
fun add(s,dict) = if member(s,dict) then dict else s::dict
end
```

(b)
```
signature DICT =
sig
  type dictionary
  val None : dictionary
  val add : string * dictionary -> dictionary
  val member : string * dictionary -> bool
end
```