15-150 Spring 2012
Homework 10

Out: 24 April, 2012
Due: 2 May, 2012, 0900 EST

1 Introduction

This homework will give you additional practice with imperative programming in SML. It is slightly short to give you time to prepare for the final.

1.1 Getting The Homework Assignment

The starter files for the homework assignment have been distributed through our `git` repository as usual.

1.2 Submitting The Homework Assignment

To submit your solutions, place your modified `memo.sml` and `hw10.pdf` files in your handin directory on AFS:

```
/afs/andrew.cmu.edu/course/15/150/handin/<yourandrewid>/hw10/
```

Your files must be named exactly: `memo.sml` and `hw10.pdf`. After you place your files in this directory, run the check script located at

```
/afs/andrew.cmu.edu/course/15/150/bin/check/10/check.pl
```

then fix any and all errors it reports.

Remember that the check script is *not* a grading script—a timely submission that passes the check script will be graded, but will not necessarily receive full credit.

Your `memo.sml` file must contain all the code that you want to have graded for this assignment and compile cleanly. If you have a function that happens to be named the same as one of the required functions but does not have the required type, it will not be graded. Modules must ascribe to the specified signatures or they will not be graded.

1.3 Methodology

You must use the four step methodology for writing functions for every function you write on this assignment. In particular, you will lose points for omitting the purpose or tests (unless otherwise specified) even if the implementation of the function is correct.

1.4 Style

We will continue grading this assignment based on the style of your submission as well as its correctness. Please consult course staff, your previously graded homeworks, or the published style guide as questions about style arise.

1.5 Due Date

This assignment is due on 2 May, 2012, 0900 EST. Remember that this deadline is final and that we do not accept late submissions.

1.6 The SML/NJ Build System

This assignment is comprised of multiple source files (although you are only to turn in a single file). The compilation of these is again orchestrated by CM through the file `sources.cm`. Instructions on how to use CM can be found in the previous homework handouts.

2 Pebbling

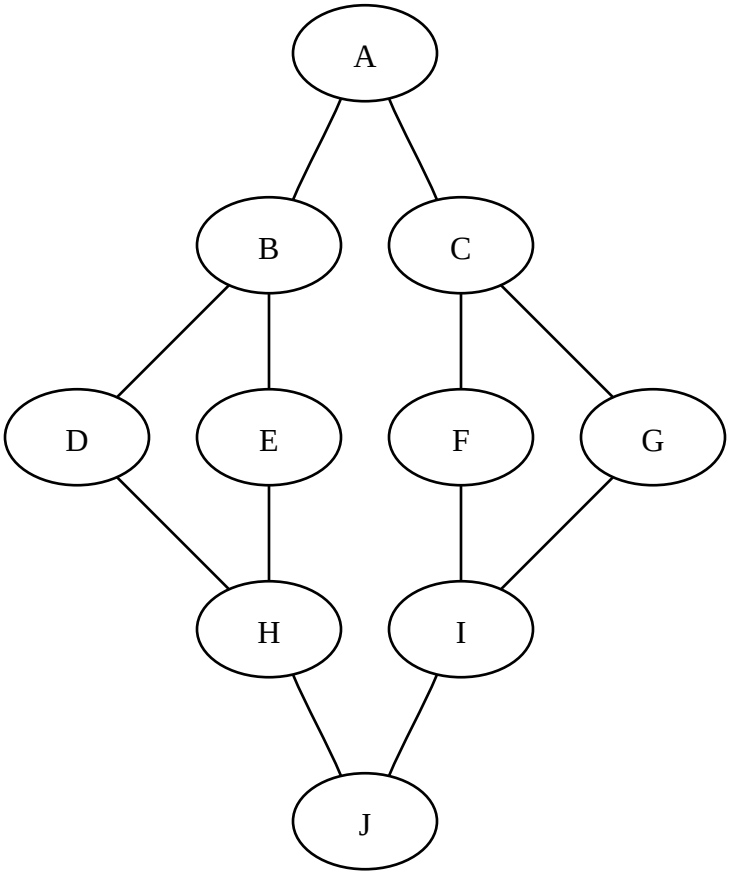

Figure 1: Example Graph for Pebbling

In lecture we discussed *pebbling* a graph to represent the scheduling of work to processors. For a series-parallel graph, a p-pebbling must obey the following rules:

1. Begin by placing a pebble on the source (*i.e.*, the top node in the graph by convention).
2. In each subsequent step, pick up the pebbles and mark the nodes that had pebbles on them as visited. Then place pebbles on up to p nodes whose predecessors have all been visited.
3. The pebbling is completed when the sink (*i.e.*, the bottom node in the graph by convention) has been visited.

There are many possible strategies for p-pebbling. In lecture, we discussed p-DFS and p-BFS. Recall that DFS prefers the left-most bottom-most available nodes, whereas BFS

prefers the higher nodes and then goes left-to-right. Consider the graph in figure 1. A 2-DFS traversal of this graph results in the assignment of pebbles to nodes indicated in the following table:

Time Step	Pebble 1	Pebble 2
1	A	
2	B	C
3	D	E
4	H	F
5	G	
6	I	
7	J	

Observe that a pebble is placed on node H before a pebble is placed on node G because this is a depth-first traversal. A 2-BFS traversal of this graph results in the following assignment of pebbles to nodes:

Time Step	Pebble 1	Pebble 2
1	A	
2	B	C
3	D	E
4	F	G
5	H	I
6	J	

Task 2.1 (5%). Give the assignment of pebbles to nodes for each time step in a 2-DFS traversal of the graph in figure 2.

Task 2.2 (5%). Give the assignment of pebbles to nodes for each time step in a 2-BFS traversal of the graph in figure 2.

3 Benign Effects

One reason to use mutation is to write a program that behaves deterministically, like a functional program, in a more convenient way. For example, mutation can be used to let different parts of your program talk to each other, without explicitly passing the relevant data around. This technique should be used with care: it is sometimes easier to write, but often harder to read. One good use of it is for debugging code: you want to write debugging code as easily as possible, without changing the structure of the main program, and it's not very important that it is readable, because it might not even stay in the final version of the code.

In this problem, we will use the MiniMax algorithm as an example. To debug game tree search algorithms, it is helpful to know how many terminal nodes of the game tree are visited. For `GAME`, we define a terminal state to be one in which `status` returns `Over` or `estimate` gets called (i.e. the non-recursive cases of MiniMax). As you saw in the previous homework, this number is different for different pruning strategies, like $\alpha\beta$-pruning and Jamboree.

In the file `minimax-counting.sml`, we have included two functors, `MiniMaxCount1` and `MiniMaxCount2`. Each of them contains a function

```
next_move : Game.state -> Game.move * int
```

that returns both the `move` computed by the MiniMax algorithm and the number of terminal states that are visited. Both of these implementations use a `ref` named `terminals` to store the count, to avoid modifying `search` and `evaluate` to pass around the count explicitly.

Parallelism in MiniMax Suppose that the call to `Seq.map` in `search` is executed in parallel (i.e. the cost graph for `Seq.map` is what we discussed in the lecture on sequences).

Task 3.1 (5%). Explain why the integer returned by `MiniMaxCount1.next_move` is dependent on the schedule. (The same explanation should apply to `MiniMaxCount2.next_move`.)

Multiple Calls to MiniMax For the remainder of this problem, we assume that all of the calls to sequence operations inside both versions of `next_move` are executed sequentially. E.g. when evaluating `Seq.map f <x1,x2,...xn>`, `f x1` is evaluated first, then `f x2`, and so on.

Suppose we apply the above functors to make two players for a particular game:

```
structure MMC1 = MiniMaxCount1(struct structure G = Con4 val search_depth = 5)
structure MMC2 = MiniMaxCount2(struct structure G = Con4 val search_depth = 5)
```

Now, suppose some client is going to make multiple calls to `MMC1.next_move` and `MMC2.next_move`—e.g. for playing multiple turns of one game of Connect 4, or for playing multiple instances of Connect 4 at once.

Recall from Lecture 28 that

- `next_move` is *benign in sequential contexts* if, for any state `s`, any two sequential executions of `next_move s` return the same move and count. This says that, when used in a sequential program, `next_move` behaves like a functional program.
- `next_move` is *benign in parallel contexts* if, for any state `s`, any two parallel executions of `next_move s` return the same move and count. This says that, when used in a parallel program, `next_move` behaves like a functional program—`next_move s` returns the same result no matter what else is running at the same time, including other instances of `next_move`.

Task 3.2 (4%). Is `MMC1.next_move` benign in sequential contexts? Briefly explain why or why not.

Task 3.3 (4%). Is `MMC1.next_move` benign in parallel contexts? Briefly explain why or why not.

Task 3.4 (4%). Is `MMC2.next_move` benign in sequential contexts? Briefly explain why or why not.

Task 3.5 (4%). Is `MMC2.next_move` benign in parallel contexts? Briefly explain why or why not.

4 This task was reversed twice

Recall the two versions of `reverse` from Lecture 5:

```
fun rev (l : 'a list) : 'a list =
    case l of
        [] => []
      | x :: xs => (rev xs) @ [x]

fun rev2 (l : 'a list) : 'a list =
    let
      fun revTwoPiles (l : 'a list, r : 'a list) : 'a list =
          case l of
            [] => r
          | (x :: xs) => revTwoPiles(xs , x :: r)
    in
      revTwoPiles(l , [])
    end
```

Both reverse their argument, but `rev2` takes linear time whereas `rev` is quadratic. In this task, you will prove that they are equivalent. More formally, you will prove:

Theorem 1. *For all values* `l : 'a list`, `rev l` $\cong$ `rev2 l`.

You should assume that `@` is implemented with the following code fragment:

```
infix @
fun (l1 : 'a list) @ (l2 : 'a list) : 'a list =
    case l1
     of [] => l2
      | x::xs => x::(xs @ l2)
```

You may assume the following lemmas without proof, but you must carefully cite them when you use them.

Lemma 1 (Append is a monoid)**.**

1. *For all valuable* `l1:'a list, l2:'a list, l3:'a list`,

   $$\texttt{(l1 @ l2) @ l3} \cong \texttt{l1 @ (l2 @ l3)}$$

2. *For all valuable* `l:'a list`, `[] @ l` $\cong$ `l`

3. *For all valuable* `l:'a list`, `l @ []` $\cong$ `l`

Lemma 2. `@` *total,* `rev` *total,* `revTwoPiles` *total*

If you try to prove Theorem 1, you will quickly realize that you need a lemma about `revTwoPiles`. In this problem, you will have to come up with this lemma on your own and prove it.

Task 4.1 (18%). State and prove a lemma about `revTwoPiles`.[1]

Task 4.2 (4%). Prove Theorem 1 using your lemma.[2]

[1] *Hint:* If your proof of the lemma is messy or impossible, consider changing your lemma!

[2] *Hint:* This proof should be almost immediate from your lemma. If it isn't, consider changing your lemma!

5 Memoization

5.1 Introduction

In the absence of effects, a function will always evaluate to the same value when applied to the same arguments. Therefore, applying a particular function to the same arguments more than once will often result in needless work. Memoization is a simple optimization that helps to avoid this inefficiency.

The idea is that you equip a function with some data structure that maps the arguments that the function has been called on to the results produced. Then, whenever the function is applied to any arguments, you first check to see if it has been applied to those arguments previously: if it has, the cached result is used instead of computing a new one; if it hasn't, the computation is actually performed and the result is cached before being returned.

If you think of a graph of a function as a set of $(input, output)$ pairs, rather than a doodle on a piece of paper representing such a set, this mapping is really storing the subset of the graph of its associated function that has been revealed so far. The optimization should let us compute each $(input, output)$ pair in the graph exactly once and refer to the already discovered graph for inputs we need more than once.

5.2 Case Study: Fibonacci

We will work through implementing this idea using the familiar Fibonacci function as a case study. Recall the naïve implementation of the Fibonacci sequence, provided in `memo.sml`:[3]

```
signature FIBO =
sig
  (* on input n, computes the nth Fibonacci number *)
  val fib : IntInf.int -> IntInf.int
end

structure Fibo : FIBO =
struct
  fun fib (n : IntInf.int) : IntInf.int =
      case n
       of 0 => 0
        | 1 => 1
        | _ => fib(n-2) + fib(n-1)
end
```

[3]Note that this code uses the built in SML/NJ type for very large integers, `IntInf.int`. This will let you run both this version of Fibonacci and the memoized versions you'll write in the next two tasks on very large input. The memoized version you write in 3.1 should be able to compute the thousandth Fibonacci number in a couple of seconds, for instance.

Task 5.1 (10%). Finish the `MemoedFibo` functor in `memo.sml` by writing a memoized version of Fibonacci.

You should represent the (*input*, *output*) mapping using a reference containing a persistent dictionary of type `D.dict`, where `D` is the argument to `MemoedFibo`. The mapping should be shared between all calls to `MemoedFibo.fib`, so that results are resued between multiple top-level calls.

If you don't know where to start, one good strategy is to use a pair of mutually recursive functions: make one function in the pair the `fib` function required by the signature; make the other function responsible for checking and updating the mapping. The benefit to this strategy is that it lets you separate memoizing from the function being memoized.

Task 5.2 (5%). Instead of hand-rolling a new version of every function that we'd like to memoize, it would be nice to have a higher order function that produces a memoized version of any function. A totally reasonable—but wrong—first attempt at writing such an automatic memoizer is shown in Figure 3.

What is wrong with this code? For example, apply the functor and use it to memoize an implementation of Fibonacci. You should observe that it is much slower than the hand-rolled version you wrote. Why?

Task 5.3 (15%). Finish the `Memoizer` functor in `memo.sml` by writing an automatic memoizer that doesn't have the problems of the `PoorMemoizer`.

Notice that `Memoizer` ascribes to a different signature than `PoorMemoizer`. Functions that can be memoized by `Memoizer` take a new argument: rather than having type

```
D.Key.t -> 'a
```

they have type

```
(D.Key.t -> 'a) -> (D.Key.t -> 'a)
```

When implementing `Memoizer`, assume that any function you memoize uses this new first argument instead of directly calling itself recursively.

Task 5.4 (5%). Finish the structure `AutoMemoedFibo` ascribing to `FIBO` using your `Memoizer` functor. This will let you test your `Memoizer` structure to make sure that you solved the problem. The Fibonacci implementation produced by your `Memoizer` function should be very nearly as fast as the hand-rolled version in `MemoedFibo`.

Task 5.5 (2%). What happens if you use `Memoizer` to memoize a function that has effects? In particular, what happens if you memoize a function that prints things?

5.3 Application: Genome Sequencing

The speed up effected by memoizing can be huge. A good practical example can be found in genetics: in sequencing a genome, one needs to find the longest common subsequence

between two sequences to get a rough idea of the amount of shared information between them. For example, a longest common subsequence of `AGATT` and `AGTCCAGT` is `AGTT`; `AGAT` is another.

More formally, a list l is a *subsequence* of l' iff l can be obtained by deleting some elements of l':

- `[]` is a subsequence of `l` for any `l`.
- `x::xs` is a subsequence of `y::ys` iff either `x::xs` is a subsequence of `ys` (delete `y`) or `x = y` and `xs` is a subsequence of `ys` (use `y` to match `x`).

Given two lists l_1 and l_2, a list l is a *longest common subsequence* of l_1 and l_2 iff l is a subsequence of l_1 and l is a subsequence of l_2 and l is at least as long as any other subsequence of both l_1 and l_2.

If DNA base-pairs are represented by the type `Base.t`, and DNA strands by the type `Base.t list`, a very straight forward program to find such longest common subsequences is

```
fun lcs (s1 : Base.t list, s2 : Base.t list) : Base.t list =
    case (s1, s2)
     of ([], _) => []
      | (_, []) => []
      | (x :: xs, y :: ys) =>
        case Base.eq (x, y)
         of true => x :: lcs (xs, ys)
          | false => Base.longerDnaOf (lcs (s1, ys),lcs (xs, s2))
```

It's pretty clear that this program takes exponential time to run. It's somewhat less clear, though, that it's also duplicating a good deal of work. For example, to find the longest subsequence of (`x1::xs`, `y1::ys`), we might have to look at (`x1::xs`, `ys`) and (`xs`, `y1::ys`). Both of these likely contain (`xs`, `ys`) as a subproblem.

Task 5.6 (5%). Use the `Memoizer` functor you've defined above to build a version of `lcs` that avoids computing subproblems more than once. Put this version in the structure `AutoMemoedLCS` in `memo.sml`. The code for the original implementation can be found in `dna.sml`.

Task 5.7 (5%). In the `SpeedExampleLCS` structure, include an example input called `speedExample` that demonstrates the speed up gained from memoizing. The original `lcs` function should take a while to run on this input, but your memoized version should terminate almost immediately.

5.4 Foreshadowing

The above uses of memoization are instances of a larger technique called dynamic programming; you'll learn more about it in 15-210. The key idea is to find problems that do a lot

of redundant work and use space to cache that work and save time. The naïve Fibonacci implementation and LCS implementation both take exponential time; when memoized, both take polynomial time.

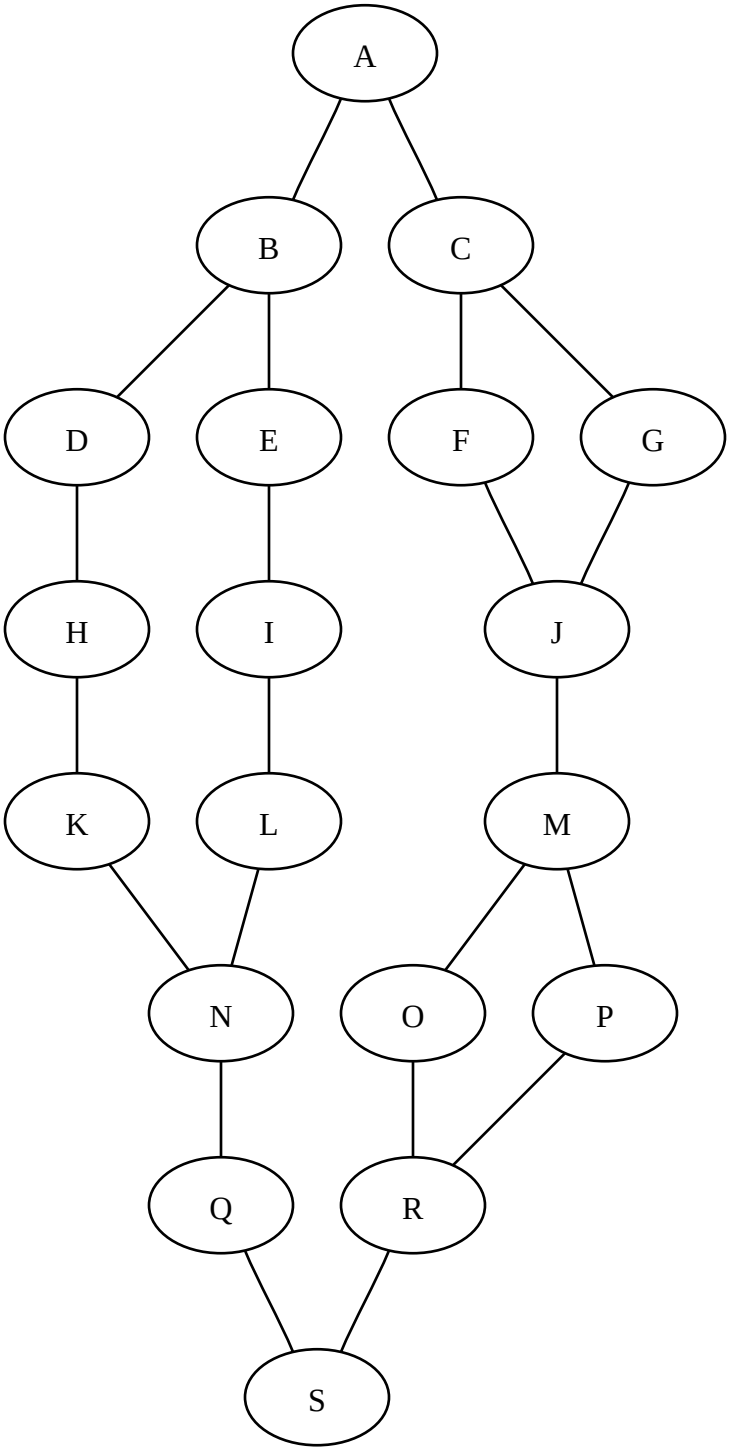

Figure 2: Graph for Tasks 2.1 and 2.2

```
functor PoorMemoizer (D : DICT) : POORMEMOIZER =
struct
  structure D = D

  fun memo (f : D.Key.t -> 'a) : D.Key.t -> 'a =
      let
        val hist : 'a D.dict ref = ref D.empty

        fun f_memoed x =
            case D.lookup (!hist) x
             of SOME(b) => b
              | NONE =>
                let
                  val res = f x
                  val _ = (hist := D.insert (!hist) (x,res))
                in
                  res
                end
      in
        f_memoed
      end
end
```

Figure 3: A Poor Memoizer