15-150 Fall 2013
Homework 12

Out: Wednesday, 27 November 2013
Due: Friday, 6 December 2013, 23:59 EST
For this assignment, NO LATE DAYS will be allowed.

1 Introduction

This homework will give you the opportunity to work with lazy and imperative programming in SML.

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

Submissions will be handled through Autolab, at

`https://autolab.cs.cmu.edu`

To submit your solutions, run `make` from the `hw/12` directory (that contains a `code` folder). This should produce a file `hw12.tar`, containing the files that should be handed in for this homework assignment. Open the Autolab web site, find the page for this assignment, and submit your `hw12.tar` file via the "Handin your work" link.

The Autolab handin script does some basic checks on your submission: making sure that the file names are correct; making sure that no files are missing; making sure that your code compiles cleanly. Note that the handin script is *not* a grading script—a timely submission that passes the handin script will be graded, but will not necessarily receive full credit. You can view the results of the handin script by clicking the number (usually either 0.0 or 1.0) corresponding to the "check" section of your latest handin on the "Handin History" page. If this number is 0.0, your submission failed the check script; if it is 1.0, it passed.

There is no written part to this homework.

Your solutions must compile cleanly, and all functions you submit should have the correct type. 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.

1.3 Due Date

This assignment is due on Friday, 6 December 2013, 23:59 EST. You cannot use a late day for this assignment.

1.4 Methodology

You must use the five step methodology discussed in class for writing functions, for **every** function you write in this assignment. Recall the five step methodology:

1. In the first line of comments, write the name and type of the function.
2. In the second line of comments, specify via a `REQUIRES` clause any assumptions about the arguments passed to the function.
3. In the third line of comments, specify via an `ENSURES` clause what the function computes (what it returns).
4. Implement the function.
5. Provide testcases, generally in the format
   `val <return value> = <function> <argument value>`.

For example, for the factorial function presented in lecture:

```
(*  fact : int -> int
 *  REQUIRES:  n >= 0
 *  ENSURES: fact(n) ==> n!
*)

fun fact (0 : int) : int = 1
  | fact (n : int) : int = n * fact(n-1)

(* Tests: *)

val 1 = fact 0
val 720 = fact 6
```

1.5 The SML/NJ Build System

We will be using several SML files in this assignment. In order to avoid tedious and error-prone sequences of `use` commands, the authors of the SML/NJ compiler wrote a program that will load and compile programs whose file names are given in a text file. The structure `CM` has a function

```
val make: string -> unit
```

`make` reads a file usually named `sources.cm` with the following form:

```
Group is

$/basis.sml
file1.sml
file2.sml
file3.sml
...
```

Loading your code using the REPL is simple. Launch SML in the directory containing your work, and then:

```
$ sml
Standard ML of New Jersey v110.69 [built: Wed Apr 29 12:25:34 2009]
- CM.make "sources.cm";
[autoloading]
[library $smlnj/cm/cm.cm is stable]
[library $smlnj/internal/cm-sig-lib.cm is stable]
...
```

Simply call

```
CM.make "sources.cm";
```

at the REPL whenever you change your code instead of a `use` command like in previous assignments. The compilation manager offers a better interface to the command line. There is less typing and less of an issue with name shadowing between iterations of your code. In short, on this assignment, the development cycle will be:

1. Edit your source files.
2. At the REPL, type

   ```
   CM.make "sources.cm";
   ```

3. Fix errors and debug.

4. If done, consider doing 251 homework; else go to 1.

Be warned that `CM.make` will make a directory in the current working directory called `.cm`. This is populated with metadata needed to work out compilation dependencies, but can become quite large. The `.cm` directory can safely be deleted at the completion of this assignment.

It's sometimes the case that the metadata in the `.cm` directory gets in to an inconsistent state—if you run `CM.make` with different versions of SML in the same directory, for example. This often produces bizarre error messages. When that happens, it's also safe to delete the `.cm` directory and compile again from scratch.

1.5.1 Emphatic Warning

CM will not return cleanly if any of the files listed in the sources have no code in them. Because we want you to learn how to write modules from scratch, we have handed out a few files that are empty except for a few place holder comments. That means that there are a few files in the `sources.cm` we handed out that are commented out, so that when you first get your tarball `CM.make "sources.cm"` will work cleanly.

You must uncomment these lines as you progress through the assignment! If you forget, it will look like your code compiles cleanly even though it almost certainly doesn't.

2 Basic Imperative

A signature `DICT` for "standard" dictionaries is defined below:

```
signature ORDERED =
sig
  type t
  val compare : t * t -> order
end

signature DICT =
sig
  structure Key : ORDERED
  type 'v dict

  val empty  : 'v dict
  val insert : 'v dict -> (Key.t * 'v) -> 'v dict
  val lookup : 'v dict -> Key.t -> 'v option
  val remove : 'v dict -> Key.t -> 'v dict
end
```

Imagine a dictionary that keeps track of the number of times its keys have been looked up. A "counting dictionary", ascribing to the signature `COUNTING_DICT` below, can be built (using the function `build`) by inserting a list of key-value pairs into an initially empty dictionary. A counting dictionary also supports the additional function:

```
hits : 'a dict -> Key.t -> int
```

where `hits d k` evaluates to the number of times the key `k` has been successfully looked up (that is, the number of times a key `EQUAL` to `k` has been looked up and its value returned) in the dictionary `d`. When `d` has no key `EQUAL` to `k`, `hits d k` returns 0.

```
signature COUNTING_DICT =
sig
  structure Key : ORDERED
  type 'a dict

  val build : (Key.t * 'a) list -> 'a dict
  val lookup : 'a dict -> Key.t -> 'a option
  val hits : 'a dict -> Key.t -> int
end
```

We can implement a counting dictionary using an ordinary dictionary (ascribing to `DICT`), suitably adjusted to incorporate references and imperative programming.

Task 2.1 (20%). Implement a functor `AddCounting` that takes a structure ascribing to `DICT` and builds a structure ascribing to `COUNTING_DICT`. You should not create a new `datatype`.

3 Infinite Lazy Lists

Recall from class the datatype for infinite lazy lists:

```
datatype 'a lazylist = Cons of 'a * (unit -> 'a lazylist)
```

Also recall the function `show : int -> 'a lazylist -> 'a list`. These are specified in `lazylist.sig` and defined in `lazylist.sml` along with several other functions that might be useful for debugging. We suggest you take a look at these files.

Given a function `cmp : 'a -> 'a -> order`, we say that a value `L` of type `'a lazylist` is `cmp`-*sorted* if for all `n` ≥ 0, `show n L` evaluates to a `cmp`-sorted finite list. Recall that a finite list is `cmp`-sorted iff each of its elements is less-than-or-equal to the elements further down the list according to `cmp`. So, putting these definitions together, an infinite lazy list is also `cmp`-sorted iff each of its elements is less-than-or-equal to the elements further down the lazy list.

Task 3.1 (10%). In `lazy.sml`, write an ML function

```
lazy_merge : ('a * 'a -> order) -> 'a lazylist -> 'a lazylist -> 'a lazylist
```

such that for any type `t`, whenever `L1` and `L2` are `cmp`-sorted lazy lists of type `t`, and `cmp` compares elements of type `t`, `lazy_merge cmp L1 L2` evaluates to a `cmp`-sorted lazy list of type `t`. Your function should have the property that for each `n` ≥ 0,

```
show n (lazy_merge cmp L1 L2)
```

consists of the `n` smallest elements from the set of items belonging to `L1` or `L2`. Elements that appear multiple times in one or both of the lazy lists should also appear multiple times in the merged lazy list. Your function should keep the relative ordering of elements within a lazy list intact, and place the elements from `L1` before `L2` if they are equal according to `cmp`.

Task 3.2 (10%). Merging two lists might be useful, but what if we had more than 2 lists? What if we had an infinite lazy list of lazy lists? Combining all of them into a sorted lazy list would be interesting! Write the function

```
combine : ('a * 'a -> order) -> 'a lazylist lazylist -> 'a lazylist
```

such that on input `L` of `cmp`-sorted lazy lists, `combine` evaluates to a `cmp`-sorted lazy list. Your function should have any property that for each `n` ≥ 0, `show n (combine cmp L)` consists of the `n` smallest elements from the set of items belonging to the lazy lists in `L`. Similarly to `merge`, `combine` should keep the ordering of elements within a lazy list intact, and place elements from earlier lazy lists first (when there are multiple options). To ensure that `combine` is total, we will require that the lazy list consisting of the first elements of each lazy list in `L` is also `cmp`-sorted. You can, and should, use this fact to your advantage. Can

you tell why we need this invariant (you do not need to turn in an answer)?

Task 3.3 (10%). Let's prove that the set of all pairs of natural numbers (that is $\mathbb{N}^2$) is countable. To do this, we need to list every single pair of natural numbers in some order. In `lazy.sml`, define the value `pairs : int lazy list` that contains every pair of natural numbers exactly once. `pair` should satisfy the following invariants:

1. If `(a1, a2)` occurs before `(b1, b2)`, then `a1 + a2 <= b1 + b2`,
2. If `(a1, a2)` occurs before `(b1, b2)` and `a1 + a2 = b1 + b2`, then `a1 < b1`,
3. If `a1` : `int` $>=$ `0` and `a2` : `int` $>=$ `0`, then `(a1, a2)` occurs in `pairs`.

According to these invariants, `show 10 pairs` should look like:

```
[(0,0),(0,1),(1,0),(0,2),(1,1),(2,0),(0,3),(1,2),(2,1),(3,0)].
```

To do this task, you might find `LazyList.iterate` and `combine` functions useful.

4 Memoization

4.1 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.

4.2 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 `fib.sml`:[1]

```
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
```

[1]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 4.1 (15%). 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 reused 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 (using the keyword `and`): 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 4.2 (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 1.

```
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 1: A Poor Memoizer

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? Put your answer in a comment in `memo.sml`.

Task 4.3 (20%). 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)
```

or, parenthesized slightly differently:

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

You should assume that your input to the memoizer is the result of the following transformation: take the text of an un-memoized recursive function; add a new first argument; everywhere in the text of that function where you used to make a recursive call, apply that new first argument to the arguments of the recursive call.

Also, when the memoized function `m` is called on arguments that have already been seen, the work of this function $W_m(n)$ should be within a constant factor of $W_{\texttt{D.lookup}}(n)$, where $W_{\texttt{D.lookup}}(n)$ is the work of the dictionary lookup function `D.lookup`.

Task 4.4 (10%). 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`. Make sure that the `fib` that you provide does as little repeated work as possible.

Task 4.5 (0%). 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? You do not need to turn in an answer for this task.

4.3 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.