15-150 Fall 2013
Homework 08

Out: Wednesday, 23 October 2013
Due: Tuesday, 29 October 2013 at 23:59 EST

1 Introduction

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`

In preparation for submission, your `hw/08` directory should contain a file named exactly `hw08.pdf` containing your written solutions to the homework.

To submit your solutions, run `make` from the `hw/08` directory (that contains a `code` folder and a file `hw08.pdf`). This should produce a file `hw08.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 `hw08.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.

Remember that your written solutions must be submitted in PDF format—we do not accept MS Word files or other formats.

You should put the code for each part in the files specified. **Do not change any of the provided signatures.**

1.3 Due Date

This assignment is due on Tuesday, 29 October 2013 at 23:59 EST. Remember that you may use a maximum of one late day per assignment, and that you are allowed a total of three late days for the semester.

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 v110.75 [built: Fri Feb  8 12:33:48 2013]
- 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 Exceptionally confusing code

In `exceptions.sml`, you'll find an implementation of `factorizer`. `factorizer` takes in an integer (greater than 1), and returns an integer list representing its prime factorization. For example, `factorizer 42 = [2, 3, 7]`. `factorizer` also is supposed to raise an exception (`Prime`) if it is given a prime-number argument in its top-level call. Trouble is, `factorizer` doesn't work.

Task 2.1 (5%). Find the bug in `exceptions.sml` and fix it. In the top comment in the file, write a short description of the bug and your fix.

3 Exceptions 2: Electric Boogaloo

Now that you've had the chance to read over some exceptional code, it's time to write some! Recall our definition of n-ary trees:

```
datatype 'a ntree = Empty
  | Node of 'a * 'a ntree list
```

You've seen many implementations of the `find` function on binary trees before, where `find p T` evaluates to a subtree of `T` whose root satisfies `p`. Now we extend this function to work with our n-ary trees, using exceptions!

Task 3.1 (15%). Implement `find`, in the file `treefind.sml`, using the exception-handling style. `find p T` should raise `NoSubtree` if no subtree of `T` has a root whose node satisfies `p`, and should evaluate to a subtree of `T` whose root satisfies `p` otherwise, under the assumption that `p` is total.

4 Like pancakes, but less delicious

One of the benefits of using functors is that we can apply a similar interface to structures which operate on different underlying types, or which implement the same behavior with very different code. For example, suppose we wanted a stack data structure. A stack is a simple ordered collection of elements with two basic interactions. We can push an element onto the stack, and we can pop an element off of the stack. Note that stacks are LIFO (Last In, First Out). The elements we pushed onto the stack most recently are the first to pop off. We could write a different stack structure for each type of stack we want, but if we want any cool functions that let us interact with our stack in different ways, this could get really tedious. So instead, we use a functor that lets us apply the same basic stack interface to integers, strings, books, pancakes, or anything else!

In this section, you'll be doing just that! You'll find the `STACK` signature in the file `stack.sig`. Here's a description of the behavior that you, our fantastic library-writer, are expected to implement for each function:

1. `empty` should evaluate to a `stack` which represents your implementation's empty stack.
2. `push` should take the passed `stacktype` and evaluate to the stack passed with it pushed onto the top.
3. `pop` should take the passed `stack` and evaluate to the tuple representing the stack after the top element is removed, and the top element itself. Be careful with how you handle the empty stack, as you must still ascribe to the `STACK` signature. In particular, you can't evaluate to an option type (as this wouldn't obey the type outlined in the signature), and you can't evaluate to a dummy value (because you don't know what an appropriate dummy value is for an arbitrary type and usage), so you must raise an exception. We've given you the appropriate exception in the signature.
4. `flip` should flip its given stack. That is, the first element to pop off the original stack should now be the last element to pop off of the new stack, and vice versa.
5. `append` should take two stacks, `S1, S2`, and put `S1` on top of S2. So `append (S1, S2) = S`, where `S2` is a substack of `S`, and `flip S1` is a substack of `flip (append (S1, S2))`. We define a substack as follows: `S` is a substack of `S`, and if `(e, S') = pop S`, then `S'` is a substack of `S` and all substacks of `S'` are substacks of `S`.
6. `find (S, p)` should find the largest substack of `S` which has a top element `V`, such that `p V` is true, and evaluate to that substack.

Task 4.1 (10%). Write the code for this functor in the file `stacks.sml`. You'll note that we've included a signature, `TYPEDEF`, in the file `TYPEDEF.sig`. You might find this handy.

The easiest way to test your stack implementation is to write another structure in `stacks.sml`. This structure can then contain various stacks and tests of their behavior. So a possible `stacks.sml` file might look like this:

```
functor Stack(T : TYPEDEF) : STACK =
struct
...
end

structure StackTests =
struct
<Tests>
end
```

5 What's in a queue?

After stacks, the next logical data structure to talk about is obviously queues. This time, however, we've already implemented the structure for you! So, for twenty-five points, please prove that we did it right.

5.1 Motivation

One key advantage of abstract types is that they enable *local reasoning about invariants*: if there is an invariant about the values of an abstract type—e.g. "this tree is a red-black tree"—and all of the operations in a particular implementation of the signature specifying that type preserve that invariant—e.g. "insert creates a red-black tree when given a red-black tree"—then any client code using that implementation necessarily maintains the invariant. The reason is that clients can only use the abstract type through the operations given the signature, so if these operations preserve the invariant all client code must as well.

In this problem, we will investigate a related question, allowing us to reason about several different implementations of the same abstract type. Specifically, we want to know:

> When can you replace one implementation of a signature with another without breaking any client code?

The answer is not as immediate as it may seem. Assuming all types in the signature are abstract, swapping implementations will produce a program that still typechecks; it may or may not, however, be correct.

Informally, the answer is that you can swap implementations when they behave the same.

5.2 Queues

To give you some practice with proving that two implementations of a signature behave the same, we've given you the following signature and implementation for queues:

5.2.1 Signature

```
signature QUEUE=
sig
   type queue
   val emp : queue
   val ins : int * queue -> queue
   val rem : queue -> (int * queue) option
end
```

In this signature,[1]

[1]Provided in `queues.sml`.

- `emp` represents the empty queue.
- `ins` adds an element to the back of a queue.
- `rem` removes the element at the front of the queue and returns it with the remainder of the queue (wrapped in a `SOME`), or `NONE` if the queue is empty.

Taken together, these three values codify the familiar "first-in-first-out" behaviour of a queue.

5.2.2 Implementations

We have given two implementations of this signature in the file `queues.sml`.

`LQ` The first implementation represents a queue with a list where the first element of the list is the front of the queue.

New elements are inserted by being appended to the end of the list. Elements are removed by being pulled off the head of the list. If the list is empty, we know that the queue is empty, so the removal fails.

This implementation is slow in that insertion is always a linear time operation—we have to walk down the whole list each time we add a new element.

Note that we also could have chosen to have front of the queue be the last element of the list, but then removal would be linear time and we'd have the same problem—we can't escape the fact that one of these operations will be constant time and the other will be linear.

`LLQ` The second implementation represents a queue with a pair of lists. One list stores the front of the queue, while the other list stores the back of the queue in reverse order. The split between "front" and "back" here can be anywhere in the queue; it depends on the sequence of operations that have been applied to the queue.

New elements are inserted by being put at the head of the reversed back of the queue. Elements are removed in one of two ways:

1. If the front list is not empty, the front of the queue is its head, so we peel it off and return it.
2. If the front list is empty, we reverse the reversed back list—now bringing it into order—make that the new front list, take an empty list as the back list, and try remove again on the pair of them.

If both the front and reversed back are empty, we know that the queue is empty, so the removal fails.

If we assume that reverse is implemented efficiently, this implementation needs to do a linear time operation on removal sometimes but not every time. Therefore, this represents a substantial speed up in the average case over the one-list implementation.[2]

To get an intuition for how these implementations work consider the following actions linked together in sequence, stated formally in `queue_ex.sml`:

$$\langle \text{ins } 1, \text{ ins } 2, \text{ ins } 3, \text{ rem}, \text{ ins } 4, \text{ rem}, \text{ rem}, \text{ rem}, \text{ rem}\rangle$$

Figure 1 shows the internal state of each representation through this sequence.

5.2.3 Relation

The relation that shows these two implementations are interchangeable flattens the two-lists representation into the one list representation. Formally, we define a relation between `int lists` and pairs of `int lists` as

$$\mathcal{R}(\texttt{l:int list}, \texttt{(f,b):int list * int list}) \qquad \text{iff} \qquad l = f@(\texttt{rev}\, b)$$

and $\mathcal{R}$ respects equivalence in that if $l = l'$, $(f, b) = (f', b')$, and $\mathcal{R}(\texttt{l}, \texttt{(f,b)})$ then $\mathcal{R}(\texttt{l'}, \texttt{(f',b')})$.

Showing that this relation is respected by both implementations for all the values in `QUEUE` amounts to proving the following theorem:

Theorem 1.

(i.) The empty queues are related:

$$\mathcal{R}(\texttt{LQ.emp}, \texttt{LLQ.emp})$$

(ii.) Insertion preserves relatedness:

For all `x:int`, `l:int list`, `f:int list`, `b:int list`

$$\text{If } \mathcal{R}(\texttt{l}, \texttt{(f,b)}), \text{ then } \mathcal{R}(\texttt{LQ.ins(x,l)}, \texttt{LLQ.ins(x,(f,b))})$$

(iii.) On related queues, removal gives equal integers and related queues:

For all `l:int list`, `f:int list`, `b:int list`, *if* $\mathcal{R}(\texttt{l}, \texttt{(f,b)})$ *then exactly one of the following is true:*

(a) `LQ.rem l` $=$ `NONE` *and* `LLQ.rem (f,b)` $=$ `NONE`

(b) *There exist* `x:int`, `y:int`, `l':int list`, `f':int list`, `b':int list`, *such that*

[2]In particular, you can show that if you can reverse a list in linear time, the two-lists implementation has amortized constant time insert and remove, while the one-list implementation will always have at least one operation that's always linear time. We won't cover amortized analysis in this class, but it's based on the idea of "expensive things that don't happen very often can be considered cheap."

i. `LQ.rem l` $=$ `SOME(x,l')`

ii. `LLQ.rem (f,b)` $=$ `SOME(y,(f',b'))`

iii. $x = y$

iv. $\mathcal{R}$(`l'`, `(f',b')`)

Task 5.1 (25%). Prove Theorem 1. Here are some guidelines, hints, and assumptions:

- Be sure to carefully state your assumptions and goals in each case, especially the two cases where you're proving an implication.
- You may use the following lemmas without proof, but you must carefully cite all uses.

  Lemma 1. *For all types* `t`*, and* `l1:t list`*,* `l2:t list`*,* `l3:t list`*,*

  `(l1 @ l2) @ l3` $=$ `l1 @ (l2 @ l3)`

  Lemma 2. *For all types* `t`*, and* `l:t list`*,* `[] @ l` $=$ `l`

  Lemma 3. *For all types* `t`*, and* `l:t list`*,* `l @ []` $=$ `l`

  Lemma 4. *For all values* `x:int`*,* `y:int`*,* `p:int list`*,* `q:int list`*,*

  if `x::p` $=$ `y::q`*, then* `x` $=$ `y` *and* `p` $=$ `q`

- You may assume without proof that `@`, `rev`, and all of the functions in both structures are total, however you should cite these facts if you use them.
- When you need to step through code, assume that `@` and `rev` are given by [3]

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

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

[3]This implementation of `rev` is not the fast reverse given by

`foldl op:: []`

but it is equivalent to it. All you would need to go from a proof of Theorem 1 for `LLQ` with this slow reverse to a proof for `LLQ` with fast reverse is a proof of their equivalence, so we don't really lose anything. The proof of Theorem 1 is substantially more straight-forward this way, so it's a nice assumption to make.

- From the structure of the code, we can see that `LLQ.rem` results in at most one recursive call to `LLQ.rem`, so you do *not* need induction to prove case (iii).
- When proving an existentially quantified statement, remember to explicitly instantiate each existentially quantified variable.

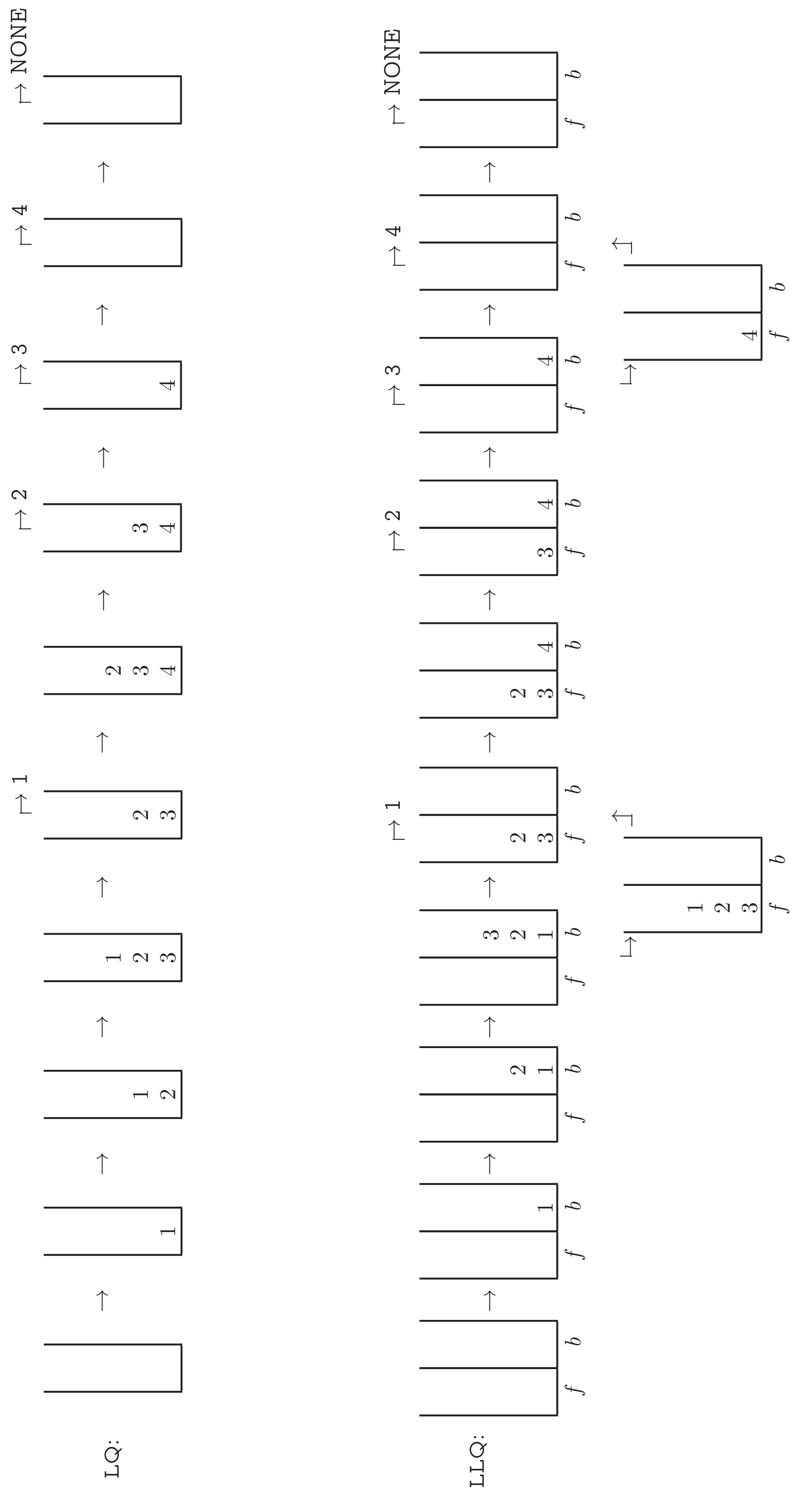

Figure 1: Queue Example

6 Learning to read, the hard way

In the `dict.sig` file we extend the signature of dictionaries from lab and lecture:

```
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
  val map : ('u -> 'v) -> 'u dict -> 'v dict
  val filter : ('v -> bool) -> 'v dict -> 'v dict

end
```

The components of the signature have the following specifications:

- `Key : ORDERED` defines the ordering of the keys in the dictionary. The signature for `ORDERED` can be found in the file `order.sml`. A structure which ascribes to `ORDERED` comtains a type, `t`, and a comparison function `compare`, for that type.
- for a type `v`, a `v dict` is a type representing a dictionary mapping keys of type `Key.t` to values of type `v`
- `empty` is a dictionary that contains no mappings.
- `insert` is a function that takes a dictionary and a key-value pair and returns the dictionary with the mapping added. If the key is already mapped to a value, the value in the dictionary should be replaced by the argument value.
- `lookup` is a function that takes a dictionary and a key, and returns `SOME v` if that key maps to the value `v` in the dictionary, or `NONE` if there is no mapping from the key.
- `remove` is a function that takes a dictionary and a key and returns the dictionary with the mapping for that key removed. If the key isn't mapped to a value, the mappings in the result dictionary should be unchanged.
- `map` is a function that takes a function `g` of type `u -> v` and a dictionary with values of type `u` and returns the dictionary with values of type `v` such that if a key is mapped to `x` by the argument dictionary then it is mapped to `g x` by the result dictionary.

- `filter` is a function that takes a function `p` of type `v -> bool` and a dictionary `d` with values of type `v` and returns the dictionary `d'` such that
  - if a key is mapped to `v` by `d` (the old dictionary), then either the key is mapped to `v` by `d'` and `p v` evaluates to `true` or there is no mapping for the key in `d'` and `p v` evaluates to `false`
  - if a key is mapped to `v` by `d'` (the new dictionary), then it was mapped to `v` in `d`.

Task 6.1 (20%). In the file `fundict.sml`, write a functor `FunDict` that takes a structure ascribing to the `ORDERED` signature and yields a structure ascribing to the above `DICT` signature using the following definition for `'v dict`:

```
datatype 'v func = Func of (Key.t -> 'v option)

type 'v dict = 'v func
```

That is, we represent a dictionary as a function from keys to value options. The intention is that, when applied to a key `k`, this function returns `SOME v` iff `k` maps to `v` in the dictionary, and it returns `NONE` iff `k` is not in the dictionary. Your functor should go in `fundict.sml`.

Task 6.2 (25%). In the file `treedict.sml`, write a functor `TreeDict` that takes a structure ascribing to the `ORDERED` signature and yields a structure ascribing to the above `DICT` signature using the following definition for `'v dict`:

```
datatype ('k, 'v) tree = Empty
                    | Node of (('k, 'v) tree * ('k * 'v) * ('k, 'v) tree)
type 'v dict = (Key.t, 'v) tree
```

The main difference here is that we represent the dictionary as a binary search tree (note, this is not just any ordinary binary tree), where the nodes contain key-value pairs. Note that you follow the same signature that you did for `FunDict`, and are implementing the same behavior (from the client's perspective) as you did in `FunDict`.

The file `dictclient.sml` has some tests for the components of the dictionary functors. You can type `use "dictclient.sml"` at the REPL after you call `CM.make` to run these tests. By default, it uses the `FunDict` functor you wrote. You may extend these tests to test your implementation of `TreeDict`.