15–150: Functional Programming

Final Examination

3 May 2011

- There are 21 pages in this examination, comprising 5 questions worth a total of 180 points.
- You have 180 minutes to complete this examination.
- Please answer all questions in the space provided with the question.
- You may refer to your personal notes and to the text, but to no other person or source, during the examination.

Full Name: ______________________________

Andrew ID: ______________________________

Survey Question: Based on your experience, do you think students in future years should take

150 then 122   122 then 150   it doesn't matter   no opinion

Question	Points	Score
Short Answer	25	
Google MapReduce	44	
Work/Span	20	
BST Insert	40	
Regular Expressions	51	
Total:	180	

Question 1 [25]: Short Answer

(a) (2 points) What are the type and value of the expression `(fn (x : int) => x div 0)`?

(b) (4 points) What are the type and value of the following expression?

```
let
  val account = ref 100
  val f = (fn () => (fn () => account := 200)) ()
  val (ref x) = account
in
  x
end
```

(c) (5 points) What is wrong with the use of `Seq.tabulate` in the following program?

```
val concat : string -> string ref -> unit =
    fn s => fn r => let val (ref s') = r in r := s' ^ s end
val account = ref "ab"
val _ = Seq.tabulate (fn 0 => concat "cd" account
                       | 1 => concat "ef" account) 2
```

(d) (5 points) Consider the following signature and implementation of sets of integers:

```
signature INTSET = sig
  type set
  val empty : set
  val insert : set -> int -> set
  val member : set -> int -> bool
end
structure ListSet : INTSET = struct
  (* Representation Invariant: elements are sorted by <;
                               no duplicate elements *)
  type set = int list
  (* ... implementations of empty, insert, and member
         that preserve the representation invariant ... *)
end
```

A client of the `ListSet` structure **can** construct a value of type `ListSet.set` that breaks this representation invariant. For example:

```
val x : ListSet.set = [5,4,5,2]
```

Describe how you would change the definition of `ListSet` so that all values of type `ListSet.set` obey the invariant:

(e) (4 points) True or false: The span of an expression can be strictly greater than its work. Circle one:

True False

(f) (5 points) The list function `foldr` has type

```
foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```

It is a variant of the familiar list function `reduce`, except that it has an inherently sequential meaning, namely

```
foldr c b [x1, x2, ..., xn]
```

returns

```
c(x1, c(x2, ..., c(xn, b)...))
```

Note that if $n = 0$ then this is just `b`. Implement `foldr`.

```
fun foldr c b l =

    case ____________

     of ____________________ => ________________________________________

      | ____________________ => ________________________________________
```

Question 2 [44]: Google MapReduce

In this problem, you will define Google MapReduce, which is a useful higher-order function on sequences. Google MapReduce is inspired by the `map` and `reduce` functions on sequences that we have studied but, as you will see, it's not just `Seq.mapreduce`.

You'll start by defining a dictionary that supports a `merge` operation. The extended dictionary signature that you will be working with is:

```
signature DICT =
sig
  structure Key : ORDERED
  type 'v dict
  ...
  val merge  : ('v * 'v -> 'v) -> 'v dict * 'v dict -> 'v dict
end
```

`merge combine (d1,d2)` merges the entries of `d1` and `d2`, combining values with the function `combine` if the same key appears in both.

You will complete an implementation of this signature using binary search trees without balancing by implementing the new `merge` function to complete the following functor:

```
functor TreeDict(Key : ORDERED) : DICT =
struct
  structure Key : ORDERED = Key
  (* invariant: sorted according to Key.compare *)
  datatype 'v tree =
      Leaf
    | Node of 'v tree * (Key.t * 'v) * 'v tree
  type 'v dict = 'v tree
  ...
  fun merge c (d1,d2) = ...
end
```

In implementing `merge`, you may assume that all the other functions are implemented correctly. You may also assume that there is a function

```
val split  : 'v dict -> Key.t -> 'v dict * 'v option * 'v dict
```

implemented inside the functor such that `split d k ==> (d1 , vo , d2)`, where

- `d1` contains exactly those keys in `d` that are less than `k`, and maps each key to the same value as in `d`
- `d2` contains exactly those keys in `d` that are greater than `k`, and maps each key to the same value as in `d`
- `vo` is `SOME v` if `k |-> v` is in `d`, and `NONE` otherwise[1]

If `d` has n keys and is balanced, `split d k` has $O(\log n)$ work and span.

[1] Recall that the notation `< k |-> v, ...>` represents a dictionary where `k` maps to `v`

(a) (20 points) **Merge**

Implement

```
val merge  : ('v * 'v -> 'v) -> 'v dict * 'v dict -> 'v dict
```

such that `merge combine (d1,d2) ==> d` where

- `k` is in `d` if and only if `k` is in `d1` or `k` is in `d2`
- If `k |-> v` in `d1` and `k` is not in `d2`, then `k |-> v` in `d`.
- If `k |-> v` in `d2` and `k` is not in `d1`, then `k |-> v` in `d`.
- If `k |-> v1` in `d1` and `k |-> v2` in `d2`, then `k |-> combine v1 v2` in `d`.

If `d1` and `d2` both have n keys and are balanced, and `combine` takes constant time, then `merge combine (d1,d2)` should have $O((\log n)^2)$ span.

```
fun merge combine (d1,d2) =
```

Google MapReduce is defined by the following signature:

```
signature GOOGLE_MAPREDUCE =
sig
    structure D : DICT
    val gmapred : ('a -> (D.Key.t * 'v) Seq.seq)
                -> ('v * 'v -> 'v)
                -> 'a Seq.seq
                -> 'v D.dict
end
```

The module `D` specifies an ordered type `D.Key.t` of keys, as well as a dictionary that supports merging on such keys.

Think of `s` as a sequence of documents. `gmapred extract combine s` takes (1) a function `extract` that extracts key-value pairs from each document (the keys in the sequence it returns need not be unique) and (2) a function `combine` that combines two values into a single value. It extracts key-value pairs from each document in `s`, and then builds a dictionary mapping each key to the value produced by applying `combine` pairwise to the values extracted with that key.

For example, if `MR : GOOGLE_MAPREDUCE` where `D.Key.t` is `string`, then we can count the words in a document as follows:

```
val words : string -> string Seq.seq
fun wordCounts (d : string Seq.seq) : int MR.D.dict =
   MR.gmapred (fn s => Seq.map (fn w => (w, 1)) (words s))
              (fn (x,y) => x + y)
              d
```

Specifically,

```
     wordCounts <"this is is document 1", "this is document 2">
==>  <("1",1),("2",1),("document",2),("is",3),("this",2)>
```

This is because the `extract` function pairs each word in the document with `1`:

```
     (Seq.map (fn w => (w, 1)) (words "this is is document 1"))
==>  <("this",1),("is",1),("is",1),("document",1),("1",1)>
```

and the `combine` function sums the counts.

You will implement the following functor:

```
functor GoogleMapReduce (Key : ORDERED) : GOOGLE_MAPREDUCE =
struct
  structure D = TreeDict(Key)
  fun gmapred extract combine s = ...
end
```

(a) (12 points) Implement a helper function

```
val collect : ('v * 'v -> 'v) -> (Key.t * 'v) Seq.seq -> 'v D.dict
```

that takes a sequence of key-value pairs, and produces a dictionary mapping each key in the sequence to the value resulting from applying `combine` pairwise to all of the values associated with it.

For example,

```
     collect (fn (x,y) => x + y) <("this",1),("is",1),("is",1)>
==>  ("this" |-> 1, "is" |-> 2)
```

Hint: use `D.merge` and other functions from the dictionary `D`.

```
fun collect combine s =
```

(b) (12 points) Implement a function

```
val gmapred : ('a -> (D.Key.t * 'v) Seq.seq) -> ('v * 'v -> 'v)
              -> 'a Seq.seq -> 'v D.dict
```

as described above. Hint: use `collect`. You may also wish to use `Seq.flatten`.

```
fun gmapred extract combine s =
```

Question 3 [20]: Work/Span

In this problem, we will define a datatype of arbitrarily branching trees known as Rose Trees. We will first represent the children using a list:

```
datatype 'a rosetree = Node of 'a * (('a rosetree) list)
```

Throughout this problem you should assume that the `rosetree`s are balanced with n nodes total and a branching factor of b (*i.e.*, each node has either zero or b children).

The following function maps an argument function over each element of a `rosetree`:

```
fun listrosemap (f : 'a -> 'b) (Node (x, s) : 'a rosetree) : 'b rosetree =
    Node (f x, List.map (listrosemap f) s)
```

For the following analysis, you should assume that argument function `f` has constant work and span.

(a) (4 points) **Work:** Give a recurrence for the work of the `listrosemap` function in terms of the number of nodes n and branching factor b.

(b) (4 points) **Span:** Give a recurrence for the span of the `listrosemap` function in terms of the number of nodes n and branching factor b. Recall that `List.map` applies its argument function sequentially.

We can instead represent the children using a sequence:

```
datatype 'a rosetree = Node of 'a * (('a rosetree) Seq.seq)
```

The following function maps an argument function to each element of a rose tree:

```
fun seqrosemap (f : 'a -> 'b) (Node (x, s) : 'a rosetree) : 'b rosetree =
    Node (f x, Seq.map (seqrosemap f) s)
```

(a) (6 points) **Work:** Give a recurrence for the work of the `seqrosemap` function in terms of the number of nodes n and branching factor b. Is this recurrence different from your work recurrence for the `listrosemap` function? Explain why or why not in one sentence.

(b) (6 points) **Span:** Give a recurrence for the span of the `seqrosemap` function in terms of the number of nodes n and branching factor b. Recall that `Seq.map` applies its argument function in parallel. Is this recurrence different from your span recurrence for the `listrosemap` function? Explain why or why not in one sentence.

Question 4 [40]: BST Insert

In this question, you will prove a property of binary search trees (BSTs). Recall the definition of insertion on BSTs:

```
structure Key : ORDERED
datatype 'a bst = Empty
                | Node of 'a bst * (Key.t * 'a) * 'a bst

fun insert (t : 'a bst) (knew : Key.t, vnew : 'a) : 'a bst =
  case t
    of Empty => Node (Empty, (k,v), Empty)
     | Node (t1, (k,v), t2) =>
       case Key.compare (knew, k)
         of EQUAL => Node (t1, (k,vnew), t2)
          | LESS => Node (insert t1 (knew,vnew), (k,v), t2)
          | GREATER => Node (t1, (k,v), insert t2 (knew,vnew))
```

BSTs obey the representation invariant that they are *sorted*: intuitively, in any internal node all keys in the left subtree are less than the key in the node, which is in turn less than all keys in the right subtree.

To express this carefully, first we define what we mean by "all keys in a BST T":

$$\begin{aligned} keys(\texttt{Empty}) &= \emptyset \\ keys(\texttt{Node}(\texttt{T}_1, (\texttt{k}, \texttt{v}), \texttt{T}_2)) &= \{k\} \cup keys(\texttt{T}_1) \cup keys(\texttt{T}_2) \end{aligned}$$

This states that there are no keys in `Empty` and that the keys in $\texttt{Node}(\texttt{T}_1, (\texttt{k}, \texttt{v}), \texttt{T}_2)$ consist of `k` and all keys in either $\texttt{T}_1$ or $\texttt{T}_2$.

Using this, we define when a BST T is sorted as follows:

- `Empty` is sorted.
- $\texttt{Node}(\texttt{T}_1, (\texttt{k}, \texttt{v}), \texttt{T}_2)$ is sorted if and only if all of the following are true:
  - $\texttt{T}_1$ is sorted, and
  - $\texttt{T}_2$ is sorted, and
  - For any $\texttt{x} \in keys(\texttt{T}_1)$, $\texttt{Key.compare}(\texttt{x}, \texttt{k}) \Longrightarrow \texttt{LESS}$, and
  - For any $\texttt{x} \in keys(\texttt{T}_2)$, $\texttt{Key.compare}(\texttt{x}, \texttt{k}) \Longrightarrow \texttt{GREATER}$.

Part of the correctness of `insert` is that it preserves this representation invariant:

Theorem 1. *If* $\texttt{T} : {'}\texttt{a bst}$ *is sorted then* $\texttt{insert T (knew, vnew)} \Longrightarrow \texttt{T}'$ *and* $\texttt{T}'$ *is sorted.*[2]

In this problem, you will prove certain cases of this theorem by structural induction on $\texttt{T} : {'}\texttt{a bst}$. You may use the following lemma in your proof but must carefully cite where you use it.

> **Lemma 1 (Domain)**: If $\texttt{insert T (knew, vnew)} \Longrightarrow \texttt{T}'$, then $keys(\texttt{T}') \subseteq \{\texttt{knew}\} \cup keys(\texttt{T})$.

[2] Throughout this problem we assume `Key.compare` is total.

(a) (10 points) **Base case**

Prove the case of the theorem when `T` = `Empty`.

To show: ____

____

(b) **Inductive case**

Next you will prove subcases of the inductive step of the theorem when $\texttt{T} = \texttt{Node}(\texttt{T}_1, (\texttt{k}, \texttt{v}), \texttt{T}_2)$.

i. (5 points) State the inductive hypotheses.

ii. (2 points) Assume that $\texttt{Node}(\texttt{T}_1, (\texttt{k}, \texttt{v}), \texttt{T}_2)$ is sorted. State what this means according to the definition of sorted.

iii. (10 points) Prove the subcase of the inductive case when `Key.compare (knew, k)` $\Longrightarrow$ `EQUAL`.

Explicitly indicate uses of inductive hypotheses or the lemma, if any.

To show: ____

____

iv. (13 points) Prove the subcase of the inductive case where `Key.compare (knew, k)` $\Longrightarrow$ `LESS`.
Explicitly indicate uses of inductive hypotheses or the lemma, if any.

To show: ____

____

Question 5 [51]: Regular Expressions

Recall the type of regular expressions from earlier in the course:

```
datatype regexp =
    Zero
  | One
  | Char of char
  | Star of regexp
  | Plus of regexp * regexp
  | Times of regexp * regexp
```

One way to implement regular expression matching is to compile a regular expression into a *deterministic finite automaton* (DFA). A DFA is a state machine with finitely many states, including a distinguished start state and distinguished final (or accept) states, equipped with a transition function that assigns each state and character a unique successor state.

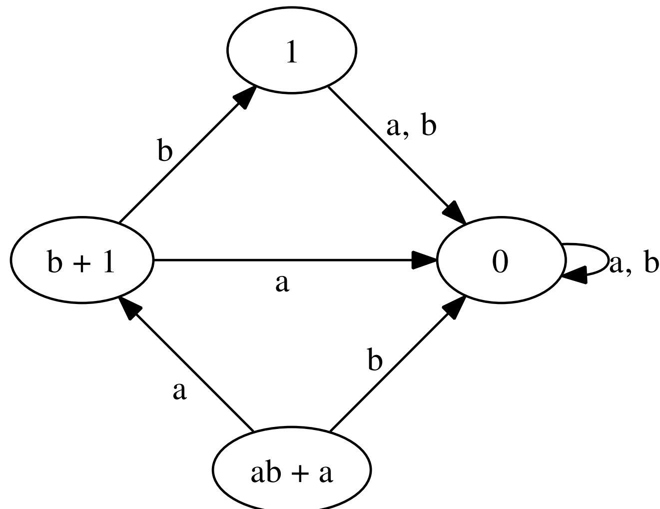
A state representing the regexp `r` transitions by the character `c` to a state representing "`r` after `c` has already been matched". That is, `(r,c)` transitions to the *derivative* of `r` *with respect to* `c`, written $\partial_c\, r$. Its language is defined as follows:

$$\mathcal{L}(\partial_c\, r) \quad = \quad \{s \mid c \cdot s \in \mathcal{L}(r)\}$$

For example,

$$\partial_a\,(ab + a) = b + 1$$
$$\partial_b\,(b + 1) = 1 + 0 = 1$$
$$\partial_a\,(b + 1) = 0 + 0 = 0$$

Thus, the regexp $ab + a$ compiles into the DFA



First you are to implement the derivative of a regexp. To do so requires an auxiliary concept: a regexp is defined to be *nullable* if and only if it matches the empty string. Using nullability, we can write the rules for the derivative of the regexp r with respect to the character c, as follows:

$$\begin{aligned}
\partial_c\, 0 &= 0 \\
\partial_c\, 1 &= 0 \\
\partial_c\, b &= \begin{cases} 1 & \text{if } b = c \\ 0 & \text{otherwise} \end{cases} \\
\partial_c\, r* &= (\partial_c\, r) \cdot (r*) \\
\partial_c\, (r_1 + r_2) &= (\partial_c\, r_1) + (\partial_c\, r_2) \\
\partial_c\, (r_1 \cdot r_2) &= \begin{cases} ((\partial_c\, r_1) \cdot r_2) + (\partial_c\, r_2) & \text{if } r_1 \text{is nullable} \\ (\partial_c\, r_1) \cdot r_2 & \text{otherwise} \end{cases}
\end{aligned}$$

(a) (15 points) Assume that you are given a function `nullable : regexp -> bool` such that `nullable r ==> true` iff `r` is nullable. Write the function

```
deriv : regexp -> char -> regexp
```

such that `deriv r c ==>` $\partial_c\, r$.

Fill in the template on the next page

```
fun deriv r c =
  case r of

    One => ______________________________

  | Zero => ______________________________

  | Char b =>

      ____________________________________________________________

      ____________________________________________________________

      ____________________________________________________________

      ____________________________________________________________

  | Star r' => ____________________________________________

               ____________________________________________

  | Plus (r1, r2) => ____________________________________________

                     ____________________________________________

  | Times (r1, r2) =>

      ____________________________________________________________

      ____________________________________________________________

      ____________________________________________________________

      ____________________________________________________________

      ____________________________________________________________

      ____________________________________________________________

      ____________________________________________________________

      ____________________________________________________________
```

Given a regexp r, our goal is to compute the transition table of the corresponding DFA. Each entry in this table maps a regexp and character (r, c) to the regexp $\partial_c r$. We represent the transition table in a curried manner: each r is mapped to a dictionary mapping each c to $\partial_c r$. Thus, we use the following type:

```
type transtable = (regexp CharDict.dict) RegDict.dict
```

These dictionaries are defined as follows:

```
structure RegDict : DICT
structure CharDict : DICT
```

Here, `RegDict.Key.t` is `regexp`. To ensure termination, it is necessary to equate regular expressions up to commutativity, associativity, and idempotence[3] of $+$, which `RegDict.Key.compare` does. `CharDict.Key.t` is `char`, compared by the standard dictionary ordering.

(a) (11 points) First, we will write a function that generates the edges out of a given regexp: Implement a function

```
val successors : regexp -> char list -> regexp CharDict.dict
```

such that `successors r alphabet` generates the dictionary containing $c \mapsto \partial_c r$ for all c in `alphabet`.

```
fun successors r alphabet =
```

[3] that is, $r + r$ is equivalent to r

Now, you will implement

```
val transitions : regexp -> transtable
```

Assume a variable

```
val alphabet : char list
```

representing a fixed alphabet.

To compute the transition table for r, we use `successors` to generate the immediate successors, and then recursively build the transition table for each $\partial_c\, r$. Because the derivative of r is not necessarily smaller than r, termination of this process is somewhat tricky. It suffices to keep track of the regexps that have already been considered, and to stop extending the table when you reach a regexp that is already in the table—similar to the visited set in depth-first search.

Thus, we will write this code in an imperative style, creating a mutable cell containing a `transtable` and updating it as we process the regular expresssion.

(b) (25 points) Fill in the code on the next page to complete the implementation of `transitions`:

1. Create the mutable cell `theTable`.
2. Write the function `trans : regexp -> unit`, where `trans r` updates `theTable` to contain the transition table for `r`. If `r` is already in the table, then no updates need to be made. If `r` is not, then we must (1) update the table by mapping r to its successors and (2) update the table for each $\partial_c\, r$.
   To loop over the `alphabet`, you may either use a higher-order function, such as `List.map`, `List.foldr`, `List.mapreduce`, or write a helper function.
3. Call `trans` to update the `theTable`.
4. Read the `theTable` and return the final result.

Fill in the template on the next page

```
val alphabet : char list = ...
fun transitions (r : regexp) : transtable =
  let
    val theTable : transtable ref = ____________________________
    fun trans (r : regexp) : unit =






    val () = ________________________________________________________


    val ________________________________________________________
  in


    ____________________________
  end
```