15-150 Spring 2016
Lab 7

24 February 2015

1 Exam Review

This lab tries to be as comprehensive as possible in covering the topics you have learned in class so far. You should answer the questions on a sheet of paper.
You are encouraged to start working on problems you're most unsure about first.

1.1 Disclaimer!

The difficulty and length of this lab does *not* reflect the actual difficulty, length, or topic focus of the exam. The format of the questions however, will likely be similar to questions on the exam.

1.2 Exam Information

Time: Thursday, Feb 25, noon-1:20 PM.
Location: McConomy (in the UC)

2 Short Answer (Values, Types, etc)

For each of the following expressions, state the most general type *and* syntactic value of the expression. If the expression is not well-typed or does not reduce, explain briefly why or why not.

(a) `1/2`

(b) `[]::[]`

(c) `"abcd"+"f"`

(d) `SOME NONE`

(e) `(fn a => a, fn b => b)`

(f) `(fn a => fn b => b a) 6`

(g) `(fn a => 1::a)`

(h) `(fn a => 1::a) [1]`

(i) `let fun f (x::L) = x in f "abc" end`

(j) `(op o)`

(k) `fun f f x = x f f`

For each of the following expressions, state the most general type. If the expression is not well-typed, briefly explain why or why not.

(l) `map filter`

(m) `filter map`

3 Currying

Write two total functions `curry`, `uncurry` with the most general types being

```
uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c
```

```
curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c
```

4 HOFs

Recall that the SML built-in functions, `foldl` and `foldr`, have the following type:

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

However, they combine data in a different evaluation order:

```
foldr f b [x1,x2,x3,...,xn] = f (x1, f (x2, f (x3, ... f (xn,b)...)))
foldl f b [x1,x2,x3,...,xn] = f (xn, ... f (x3, f (x2, f (x1,b)...)))
```

Implement the following functions using only `foldl`, `foldr`, and any anonymous function. Your function must not be recursive. You may not use or define any other helper functions. You may use builtin operators such as `case`, `::`, `if`, `andalso`, `orelse`, but you may NOT use `@`.

(a)
```
(* reverse L evaluates to list L reversed *)
fun reverse (L : 'a list) :  'a list =
```

(b)
```
(* length L evaluates to length of L *)
fun length (L : 'a list) :  int =
```

(c)
```
(*find L evaluates to SOME x if x exists in L, NONE otherwise *)
fun find (L : ''a list) (x :  ''a) :  ''a option =
```

(d)
```
(* map f L evaluates to list with f applied to all elements in L,
* kept in same order *)
fun map (f :  'a -> 'b) (L : 'a list) :  'b list =
```

5 Tree Product (Proof, Big-O)

Recall the tree datatype:

```
datatype 'a tree = Empty | Node of 'a tree * 'a * 'a tree
```

Consider this function, which computes the product of integers in a tree:

```
fun mult Empty = 1 (* multiplicative identity *)
  | mult (Node(L,x,R)) = x * (mult L) * (mult R)
```

(a) Using induction, prove that `mult T` evaluates to the product of all elements in tree `T`.

(b) Given that `T` is a balanced `int tree` with depth `d`, write recurrences that will represent the work and span of evaluating `mult T`, in terms of `d`.

(c) Using the recurrences you found in part (b), what are the respective big-$\mathcal{O}$ bounds? Show your work.

6 Fib (Proof, Big-O)

Consider the following two implementations of `fib`:

```
fun fib1 0 = 1
  | fib1 1 = 1
  | fib1 n = fib1(n-1)+fib1(n-2)
```

and

```
fun fib2_helper 0 = (0,1)
  | fib2_helper n = let val (n1,n2) = fib2_helper (n-1) in (n2,n1+n2) end

fun fib2 0 = 1
  | fib2 1 = 1
  | fib2 n = let (_,x) = fib2_helper n in x end
```

(a) Prove that `fib1 n` $\cong$ `fib2 n`

(b) Find the big-$\mathcal{O}$ runtime for both of the functions. Show your work.

7 Binary Generation (HOFs, Continuation)

Given two non-negative integers `m` and `n`, we are interested in all the possible binary numbers that can be formed using `m` 1's and `n` 0's. We will represent binary numbers as an `int list` of 0's and 1's. (For example, the binary number 100 would be represented as `[1,0,0]`.)

(a) Define the following recursive helper function that returns an `int list` of length `d` that only contains `n`'s, given that `d` is non-negative. The type of `listOfNs` is `int -> int -> int list`.

```
fun listOfNs n 0 =
  | listOfNs n d =
```

(b) Define a recursive function `bingen` where `m` and `n` are defined as above. (For example, `bingen 1 2 =>* [[1,0,0],[0,1,0],[0,0,1]]`. You are allowed to use `listOfNs`, `map`, `::`, `@`, and any anonymous functions, but no other helpers. The type of `bingen` is `int -> int -> int list list`.

```
fun bingen 0 n =
  | bingen m 0 =
  | bingen m n =
```

(c) Using continuation, implement function `bingenC` of type `int -> int -> (int list list -> 'a) -> 'a`, where `bingenC m n k` $\cong$ `k (bingen m n)`. You are allowed to use `listOfNs`, `map`, `::`, `@`, and any anonymous functions, but no other helpers.

```
fun bingen 0 n k =
  | bingen m 0 k =
  | bingen m n k =
```