15-150 Fall 2013
Lab 7

9 October 2013

1 Introduction

The goal for this lab is to give you more practice with continuations and expose you more to regular expressions. Please take advantage of this opportunity to practice writing functions and proofs with the assistance of the TAs and your classmates. You are, as always, encouraged to collaborate with your classmates and to ask the TAs for help.

1.1 Getting Started

Update your clone of the `git` repository to get the files for this weeks lab as usual by running

```
git pull
```

from the top level directory (probably named `15150`).

1.2 Methodology

You must use the five step methodology for writing functions for every function you write on this assignment. In particular, every function you write should have REQUIRES and ENSURES clauses and tests.

2 Types

Task 2.1 Give the most general type and value for the following functions and expressions (if it exists).

1. `map (fn x => 3)`

2. `List.tabulate (4, (fn x => map (fn y => (x*x, #"z")) [#"a", #"b", #"c"]))`

   where `List.tabulate : int * (int -> 'a) -> 'a list` and `List.tabulate n f` generates a list of length `n` where the ith element is generated by `f i`.

3. `([]::[[]])::[[[]]]`

4. `(fn x => fn y => fn z => if y x then x z else z y)`

5. `fun f x = f x`

3 Continuations

Task 3.1 Step the following code and be sure that you understand how it works:

```
fun fact 0 k = k 1
  | fact n k = fact (n - 1) (fn x => k (n * x))


fact 3 (fn x => x)
```

Task 3.2 Write `size` on trees using Continuation-Passing Style. Try it with pen/pencil and paper rather than on a computer. You will be surprised at how different coding feels when you don't have a keyboard. It's a good idea to practice this before the exam.

4 Regular Expressions

Regular expressions are powerful tools in the computing world. Here you can practice declaring some functions related to regular expressions. For your convenience, the datatype for regular expressions has been reproduced here:

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

Now we can define a function of type

```
anyChar : char list -> regexp
```

that will create a `regexp` which will accept any character in the input list. We provide you this code as an example:

```
fun anyChar (L : char list) : regexp =
    foldr (fn (c,R) => Plus(Char c, R)) (One) L
```

The function `foldr` will likely be helpful in the other functions you create as well.

Task 4.1 Define the function

```
fromString : string -> regexp
```

that generates a regular expression accepting exactly the string inputted to the function.

Hint: You may want to use String.explode.

Task 4.2 Define the function

```
anyString : string list -> regexp
```

such that `anyString(L)` returns a regular expression that will match any string in `L`.

Task 4.3 Define the function

```
emailer : string -> regexp
```

such that `emailer d` creates a regular expression that will accept any email address from the domain `d` or any subdomain of `d`. That is to say that it must have the @ symbol in it and it must end in `d`. We have provided the variable `validChars` which contains a list of all of the characters we will say can be used in an email address.

For example, `emailer "cmu.edu"` would create a regular expression accepting strings *andrewid@andrew.cmu.edu* or *webmaster@cmu.edu*, but not *user@gmail.com* since it is not under the correct domain, nor *cs.cmu.edu* since it does not contain an @ symbol.

5 Polynomials

We can consider a polynomial as a sum of monomials of increasing powers with coefficients. More formally, we can write the function $f(x)$ as

$$\sum_{i=0}^{\infty} a_i x^i$$

where $\{a_i\}$ represents a sequence of coefficients. Thus, we can define a polynomial as a function `f : int -> real` which takes in the index of which coefficient we want to retrieve.

We can declare our type as

```
type poly = int -> real
```

Task 5.1 Write the function

```
add : poly * poly -> poly
```

which returns a `poly` that would be equivalent to adding the two polynomials inputted.

Now consider the two polynomials

$$f_1(x) = \sum_{i=0}^{\infty} a_i x^i, f_2(x) = \sum_{i=0}^{\infty} b_i x^i$$

We can then write the multiplication $f(x) = f_1(x)f_2(x)$ as

$$f(x) = \sum_{i=0}^{\infty} \sum_{j=0}^{i} x^i a_j b_{i-j}$$

Less formally, the term of degree n in the sum will have a coefficient that represents the sum of multiplied pairs from the original two series whose indices add up to n. You may want to multiply a few example polynomials to make sure you understand how the coefficients are determined. For example,

$$(a_0 + a_1x)(b_0 + b_1x + b_2x^2) = a_0b_0 + (a_0b_1 + b_0a_1)x + (a_1b_1 + a_0b_2)x^2$$

Task 5.2 Write the function

```
mult : poly * poly -> poly
```

which multiplies the two inputted polynomials together.

Although we cannot fully evaluate a function in this form, it is possible to approximate the value of the function at a point by having some threshold number of terms to evaluate.

Task 5.3 Write a function

```
eval : poly -> int -> int -> real
```

such that `eval f n x` returns the sum of the first `n` terms of the polynomial `f` evaluated at `x`.