# 15-150 Summer 2016
# Lab 15

April 27th, 2016

## 1 Introduction

### 1.1 Getting Started

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

```
git pull
```

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

### 1.2 Methodology

You should practice writing requires and ensures specifications, and tests on the functions you write on this assignment. In particular, every function you write should have both specs and tests.

## 2 Types

A portion of the final will be basics questions that test your knowledge of types. Thus, it is essential to understand how to look at a function and figure out the type of it. You will get some practice today.

**Task 2.1** For each of the following expressions, determine if the expression is well-typed. If it is, give its most general type. If not, explain why. Also determine if the expression reduces to a value. If it does, state what the value is. If not, explain why not.

(a) `fun x y = y`

(b) `fun fact a = a !`

(c) `(fn c => ref c)`

(d) `ref ref ref`

(e) `[] :: [] :: []`

(f) `fun y x = (fn y => case y of x => map map x)`

(g) `! o !`

(h) `fn x => !`

(i) `(1,3)+(2,4)`

(j) `[fn x=>x, fn x=>2]`

(k) `(fn x=>fn y=> x+y) 42`

(l) `foldr map`

(m) `map foldr`

(n) `SOME NONE`

(o) `case SOME 3 of NONE => 4 | _ => SOME 4`

(p) `let fun f x = f x in f 3 end`

(q) `([], []) :: []`

# 3 Cyclic Lists

Recall the imperative linked list data structure from lab 13:

```
datatype 'a cell = Nil
                 | Cons of 'a * 'a llist
withtype 'a llist = ('a cell) ref
```

Sometimes it's helpful to have a way print instances of our data structures for debugging purposes. This can be particularly useful for mutable data structures liked linked lists, which can be destructively modified as a side-effect of evaluation.

**Task 3.1** Define a function

```
printLlist : ('a -> string) -> 'a llist -> unit
```

`printLlist toString l` should print a representation of `l` using `toString` to get the representation of each element.

We say a linked list is cyclic if there exists a path from any node in the list to itself. A linked list is said to be acyclic if there is no such path. For example, the list in Figure 3 is cyclic but the list in Figure 3 is acyclic.

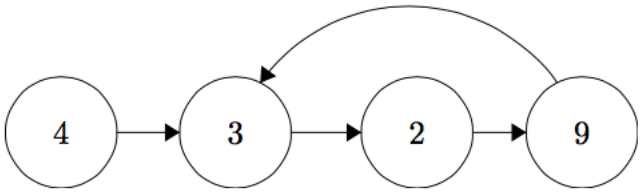


Figure 1: a cyclic linked list of integers

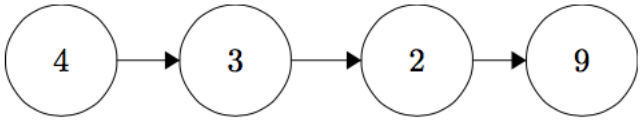


Figure 2: an acyclic linked list of integers

There's nothing wrong with cyclic lists per se, but code that that assumes acyclic input will almost always fail on cyclic lists. For example, we can define the aforementioned cyclic list by

```
val testend = ref Nil
val testmid = ref (Cons (3, ref (Cons (2, ref (Cons (9, testend))))))
val () = testend := !testmid
val testcyclic = ref (Cons (4, testmid))
```

**Task 3.2** What happens when you run `printLlist` on `testcyclic`? What happens when you run `toList` on `testcyclic`?

**Task 3.3** Define a function

```
llistEq : 'a llist * 'a llist -> bool
```

`llistEq` should return true if `l` and `l'` are the same ref cell and false otherwise.
(Hint: Don't overthink it.)

**Task 3.4** Define a function

```
isCyclic : 'a llist -> bool
```

The function `isCyclic` evaluates to `false` if the argument list is acyclic and `true` if the argument list is cyclic.

Your implementation of `isCyclic` must use only a constant amount of space and run in time at most linear in the number of unique cells in the argument list. You will need to follow the references in the linked list structure, but you may not destroy the linked list. Your implementation should be purely functional, so you may not have any ephemeral storage.

# 4 Zero-simplifying Regular Expressions

Recall the datatype that we use to represent regular expressions:

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

It happens that there are infinitely many ways of writing a regular expression that accepts nothing. For example, all of the following are regular expressions that accept nothing:
`Zero, Plus(Zero, Zero), Plus(Zero, Times(Zero, One)), Times(Char #"a", Zero)`

**Task 4.1** Write a function `zeroout: regex -> regex` such that `zeroout R` evaluates to `Zero` if `R` accepts nothing and otherwise evaluates to a regular expression `R'` that is zero-simple and has the same language as `R`. `R` is a zero-simple expression if there are no instances of the constructor `Zero` within `R`. We have written the base cases. It remains for you to write the case for: `Plus, Times, and Star`.

```
fun zeroout (R: regex) : regex =
    case R of
        Zero => Zero
      | One => One
      | Char x => Char x
      | Plus(R1, R2) => (* TODO *)
      | Times(R1, R2) => (* TODO *)
      | Star(R1) => (* TODO *)
```

# 5 Proof Practice

Recall that `map` takes a function `f` and a list `L`, then builds a new list with `f` applied to each element in the `L`. We are interested in what happens when we map multiple functions on a list `L`. We want to show that loosely speaking, mapping multiple functions iteratively on a list is equivalent to mapping the composition of those functions once on the list.

Consider the following function definitions:

```
(*id is the identity function*)
fun id x = x

fun fold f b [] = b
| fold f b (x::L) = f(fold f b L, x)

fun map f [] = []
| map f (x::L) = (f x)::(map f L)
```

The infix operator `o` is defined such that `f o g = comp (f, g)`, where comp is defined by `fun comp (f, g) x = f(g(x))`.

**Task 5.1**

Prove:
For all types `t` and all values `L` of type `t list` and all values `FL` of type `(t -> t) list`, where all the elements of `FL` are total, the following equality holds. Remember, two expressions are equal if they both fail to terminate or they evaluate to equivalent values.

```
fold (fn (x,y) => map y x) L FL = map (fold (fn (x,y) => y o x) id FL) L
```

You may use the following lemmas without proof:

**Lemma 1:** For all types `s, t, u`, and for all values `L` of type `s list`, and all functions `g` of type `s -> t` and all functions `f` of type `t -> u`:

```
map f (map g L) = map (f o g) L
```

**Lemma 2:** For all types `t` and all values `L` of type `t list`,

```
map id L = L
```

Hints:

1. Evaluational reasoning will introduce complexity that you may not want to deal with.
2. Let `L` be arbitrary and fixed. Induct on `FL`.
3. Equality is transitive.

# 6 More Practice

Recall the tree datatype:

```
datatype tree = Empty
              | Node of tree * int * tree
```

For this question, we want to sum up the numbers in a tree in a special way. Consider a leaf and the path from the root to the leaf. We will call the sum at a leaf to be the sum of the value at the leaf added with the values on the path from the root to the leaf where each value on the path is multiplied by 10 raised to the power corresponding to the level that the value is on. Then, the sum of the tree is the sum of all the leaf sums.

Consider the following tree:

```
  4
 / \
2   3
```

Given the description of how we will sum up trees in this question, you should interpret this tree sum as $(2 + 4 * 10^1) + (3 + 4 * 10^1) = 42 + 43 = 85$.

**Task 6.1** Write the function

```
treeSum : tree -> int
```

that computes the sum in this way.

Now, moving forward. Let's work on some continuations.

**Task 6.2** Please convert map for lists into one with continuation passing style.

Also, remember that function that found the number of nodes of a tree.

```
fun numNode Empty = 0
  | numNode (Node(l, x, r)) = 1 + numNode(l) + numNode(r)
```

**Task 6.3** Please convert that into continuation passing style.

**Task 6.4** Also, pretend like that for any empty tree, we raised an exception. How would you change the `numNode` function?

Lastly, a question for you. If you were given a list of positive numbers and a positive integer, `total`, such that the numbers in the array represent coefficients of the equation $x_0y_0 + x_1y_1 + ... + x_iy_i =$`total`. How would you find the number of solutions of $(y_0, ..., y_i)$?

**Task 6.5** Write a pure and total function

```
combinations : int list -> int -> int
```

such that `combinations L total` gives the number of ways to choose $(y_0, ..., y_i)$ such that $x_0y_0 + x_1y_1 + ... + x_iy_i =$`total`.

# 7 Work and Span

Work and span analysis, which is an important aspect of Computer Science was covered in this class. Hence, it is important that these concepts are understood.

**Task 7.1** Rearrange the following function classes such that they are ordered from smallest to largest

(a) $42^n$

(b) $15150n$

(c) $2n^{1.5}$

(d) $n^n$

**Task 7.2** Explain what the following function does, provide its cost bounds and justify them.

```
fun fact (t : int tree) : int*int =
  case t of
    Empty=>(0,0)
  | Node(Left, x, Right)=>
      let
        val (m1, c1)=sumUp Left
        val (m2, c2)=sumUp Right
      in
        (Int.max(Int.max(m1, m2), x), c1+c2+1)
      end
```

**Task 7.3** Do the same for this function as you did in the previous task.

```
fun equivalenceClass (s: int seq) =
  Seq.tabulate (fn i=>Seq.filter (fn B=>B mod i=0) s)) (Seq.length s)
```

# 8 Sequences

We covered sequences, a data structure that allows for parallel operations. Using such a data structure will often allow us to achieve better span than if we use other structures, for instance, lists.

**Task 8.1** Write a function, `prefixSum : int seq -> int seq`, that given `s : int seq` evaluates to an `int seq` of elements whose index in the sequence corresponds to the prefix sum of the original sequence. For example, if `s = <1,2,3>`, `prefixSum s = <0,1,3>`.

**Task 8.2** Write a function `concatInt : int seq -> string`, that given `s : int seq`, returns a string that is all the string representation of the integers in `s` concatenated together. For example, `concatInt <1,2,3> = "123"`.