15-150 Summer 2017
Homework 05

Out: Tuesday, 6 June 2017
Due: Sunday, 11 June 2017

0 Introduction

This homework will focus on higher order functions. There is also one problem that is intended to give you some practice with user defined data types using regular expressions as an example.

0.1 Getting The Homework Assignment

The starter files for the homework assignment have been distributed through our `git` repository, as usual.

0.2 Submitting The Homework Assignment

Code submissions will be handled through Autolab, at

`https://autolab.andrew.cmu.edu`

Written submissions will be handled through Gradescope, at

`https://gradescope.com`

To submit your code, run `make` from the `hw/05` directory (that contains a `code` folder. This should produce a file `hw05.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 `hw05.tar` file via the "Handin your work" link.

To submit your written solutions (which *must* be in PDF form), log in to Gradescope and submit your PDF to the assignment **Homework 05**. If you are unable to do this, make a Piazza post or find a TA to help.

The Autolab handin script does some basic checks on your submission: making sure file names are correct; making sure that no files are missing; making sure your code compiles

cleanly; making sure your functions have the right types. In addition, the script tests each function that you submit individually using public tests. These public tests are in no way comprehensive, and will in most cases consist of a simple base case test. In order to receive full points, you must pass private tests that will not be run until after the submission deadline has passed, so you will still have to write your own tests to ensure correctness.

The script indicates a correct submission via a score on the Autolab website. If all files are present, your submission will receive a score of "0.0" in the "files" category. If your code compiles, it will receive a score of "0.0" in the "compile" category. If files are missing or the code does not compile, you will receive a highly negative score for the corresponding category. For each function, you will receive a score that represents the number of public tests passed for that function. If you do not receive full points for the public tests for a particular function, you will receive a score of zero for that function.

Your `hw05.sml` file must contain all the code that you want to have graded for this assignment, and must compile cleanly. If you have a function that happens to be named the same as one of the required functions but does not have the required type, it will not be graded.

0.3 Due Date

This assignment is due on Sunday, 11 June 2017. Remember that you ***do not get grace days this semester***, so we will not be accepting late submissions without a documented, university-approved excuse.

0.4 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:

```
(*  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 Types and Polymorphism

In class we discussed typing rules. In particular:

- A function expression `fn x => e` has type `t -> t'` if and only if, by assuming that `x` has type `t`, we can show that `e` has type `t'`.
- An application $\texttt{e}_1\ \texttt{e}_2$ has type `t'` if and only if there is a type `t` such that $\texttt{e}_1$ has type `t -> t'` and $\texttt{e}_2$ has type `t`.
- An expression can be used at any instance of its most general type.

For each of the following SML values, what is the most general type? If one of the expressions are not well-typed, briefly explain why.

Task 1.1 (3 pts).

```
fun elephant (dumbo, ears) =
  case dumbo of
      [] => ears
    | oooo::TRUMPET => "BEEP" ^ elephant (TRUMPET, ears)
```

Task 1.2 (3 pts).

```
fn x => (fn y => x)
```

Task 1.3 (3 pts).

```
(fn x => (fn y => x)) []
```

2 Typology

Recall the definitions of the higher-order functions map and foldl for lists:

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

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

For each of the following expressions, determine if the expression is well-typed.

If the expression is indeed well-typed, state its type and additionally state in a sentence what the expression does / what value it produces.

If the expression is not well-typed, say so and explain why.

Task 2.1 (2 pts). `foldl (fn (x,y) => (x=3) orelse y) false ["Three", "Four"]`

Task 2.2 (2 pts). `foldl (fn (x,y) => x ^ y) "" ["Hello", "Hola"]`

Task 2.3 (2 pts). `map (fn x => if x = 42 then [41,x] else [43,x]) [[42],[43]]`

Task 2.4 (3 pts). `map (fn L => foldl (fn (x,y) => x+y) 0 L)`

3 calculus throwback

We can represent a polynomial $c_0+c_1x+c_2x^2+\ldots$ as a function that maps a natural number, i, to the coefficient c_i of x^i. For these tasks we will take the coefficients to be real numbers. Therefore, we have the following type definition for polynomials:

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

For instance, see:

```
val p : poly = fn 0 => 1.0
               | 1 => 0.5
               | 2 => 7.0
               | _ => 0.0
```

The function `p` would represent the polynomial $1 + \frac{1}{2}x + 7x^2$.

As an example of how to define operations on polynomials defined in this manner, see the following definition for the addition of polynomials:

```
fun plus (p1 : poly, p2 : poly) : poly = fn e => p1 e + p2 e
```

Also note the functions such as

```
polynomialEqual : poly * poly * int -> bool
```

in `lib.sml`. In this function, the int parameter represents the term you are checking up to. Since you will not be able to pattern match against polynomials, some of the lib functions may be useful in helping you test your code.

3.1 Differentiation

Recall from calculus that differentiation of polynomials is defined as follows:

$$\frac{\mathrm{d}}{\mathrm{d}x}\sum_{i=0}^{n} c_i x^i = \sum_{i=1}^{n} i c_i x^{i-1}$$

Task 3.1 (10 pts). Define the function

```
differentiate : poly -> poly
```

that computes the derivative of a polynomial using the definition above. Note that `differentiate` should *not* be recursive.

3.2 Integration

Recall from calculus that integration of polynomials is defined as follows:

$$\int \sum_{i=0}^{n} c_i x^i \,\mathrm{d}x = C + \sum_{i=0}^{n} \frac{c_i}{i+1} x^{i+1}$$

where C is an arbitrary constant known as the constant of integration. Because C can be any number, the result of integration is a family of polynomials, one for each choice of C. Therefore, we will represent the result of integration as a function of type `real -> poly`.

Task 3.2 (10 pts). Define the function

```
integrate : poly -> (real -> poly)
```

that, given a polynomial, computes the family of polynomials corresponding to its integral. Note that `integrate` should *not* be recursive.

4 switching it up

Task 4.1 (15 pts). Write the function

```
transpose : 'a list list -> 'a list list
```

that interchanges the rows and columns of a list of lists. For example,

```
transpose [[1,2]] ==> [[1],
                       [2]]


transpose [[1,2], ==> [[1,3],
           [3,4]]      [2,4]]


transpose [[1,2],
           [3,4], ==> [[1,3,5],
           [5,6]]      [2,4,6]]
```

Also, if the inner lists are all empty, the function should return the empty list. For example,

```
transpose [[], [], []] ==> []


transpose [[]] ==> []


transpose [] ==> []
```

Your function only needs to work if all the inner lists have the same length, so for example

```
transpose [[1,2],[3]]
```

can have whatever behavior you find most convenient. Your implementation of `transpose` may use recursion, but should use higher-order functions where possible.

5 higher order warp jump

Given the following definitions:

```
fun foldl f z [ ] = z
| foldl f z (x::L) = foldl f (f(x,z)) L;
```

```
fun foldr f z [ ] = z
| foldr f z (x::L) = f(x, foldr f z L);
```

Task 5.1 (15 pts).
Prove: for all types `t1` and `t2`, total function `f: t1 * t2 -> t2`, and values `z: t2`, `L: t1 list`

```
foldr f z (L@[y]) = foldr f (f(y,z)) L
```

You may need the following lemma:

- **Lemma a**: For all types `t` and values `x: t` and `L,R: t list`, `x::(L@R)` = `(x::L)@R`

6 Forests

The year is 2037, and the nation's forests are in dire shape. As a responsible citizen who knows all about datatypes in SML (which has now become the language of choice for coders everywhere), you decide to join the Macrocosm Maintenance Group (MMG), headed by multimillionaire rapper and ex-Tic Tac Toe world champion Rick Ross.

On your first day of the job, Ross, purveyor of late-career hits such as *Stay Plantin* and *Purple Bonsai (ft. Skrillex)*, briefs you with your task. The previous resident computer scientist at MMG compiled a list of forests across the world, all represented through an SML datatype. To help Ross with his conservation efforts, your job is to implement a series of functions to deal with the SML forests.

6.1 The Forest Datatype

An SML forest is defined as follows:

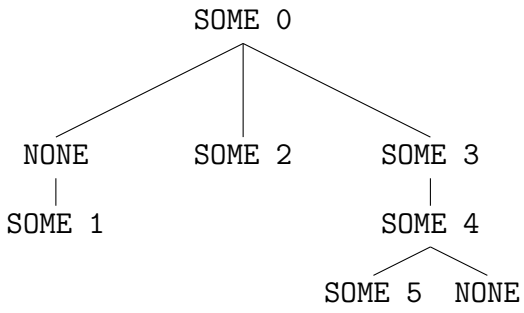
```
datatype 'a forest = Node of 'a option * 'a forest list
```

In other words, a forest can be thought of as a tree or shrub with an arbitrary number of branches, with the branches represented by the 'a forest list and the parent represented by the 'a option. In this representation, we denote an empty forest (referred to here on out as a stump) as

```
Node (NONE, [])
```

6.2 A Forest Example

Ross was kind enough to provide you with an example of a forest in SML, since your predecessor was less-than-good at reasoning about the datatype. After looking at Ross' example scrawled on the back of a napkin, and about an hour and a half fiddling with a latex library you've never used before, you finally have a good looking example of a forest!



This corresponds to the data given by:

```
Node (SOME 0,
  [Node (NONE  , [Node (SOME 1, [])])
  ,Node (SOME 2, [])
  ,Node (SOME 3, [Node (SOME 4,
                        [Node (SOME 5, [])
                        ,Node (NONE  , [])
                        ])
                 ])
  ]
)
```

6.3 Conservation Basics

Task 6.1 (1 pts). First, write the forest datatype.

Task 6.2 (5 pts). Write the functions

```
fun singleton : 'a -> 'a forest
```

where a singleton is a forest comprised of a single parent tree,

```
fun is_stump : 'a forest -> bool
```

where a stump is an empty forest,

```
fun add_children: 'a forest -> 'a forest list -> 'a forest
```

where given a parent forest and a list of child forests, appends the forest list to the current children of the parent forest.

6.4 Advanced Conservation

Task 6.3 (7 pts).

Rick Ross now comes to you with two more complex tasks. To facilitate labeling of the different types of trees, Ross wants you to create a generic map function for forests that would take in a function and return forest containing the result of applying the function to every tree in the forest.

```
fun fmap: ('a -> 'b) -> 'a forest -> 'b forest
```

Task 6.4 (7 pts).

In addition to a map function, Ross wants a generic forest filter function for classification and filing purposes that would take in a generic predicate function and apply it to every element of an input forest. `ffilter` would then return a forest that contained every element for which the predicate function returned `true`, and `NONE` in place of the elements for which the predicate function returned `false`. Note that he also wants to preserve the original structure of the forest, so do not flatten the forest (see `[*]` below).

```
fun ffilter: ('a -> bool) -> 'a forest -> 'a forest
```

`[*]` This means: **DO NOT** flatten the forest into some kind of list representation, then operate on the list in some way and turn it back into a forest. It is ok to use higher order list functions on the existing list components of `Node`s, but do not flatten the forest itself.

6.5 Better Than Pinchot

With the help of your tools, MMG has been doing so well with conservation that Rick Ross' released a celebratory hit new single, *Buy Back the Forest*. But Ross wants to keep on Hustlin' and gives you your final task of implementing two powerful functions, `freduce` that has the power to do mass accumulation operations on any given forest, and one called `fprune` that removes all dead stumps from the forest representation.

Task 6.5 (10 pts).

```
fun freduce: ('a * 'a -> 'a) -> 'a -> 'a forest -> 'a
```

`freduce` is a function that takes in an *associative* function and a base case that is an identity and applies the function to every element of the forest, in no particular order. `freduce` should work by combining each element of the forest pairwise, with the base case applied if there is only one element of the forest. **Note that it's very important that the function that freduce takes in is associative and commutative, and that the base case is an identity!** Otherwise, you'll run into some very odd behavior!