15-150 Fall 2015
Homework 06

Out: Wednesday, 7 October 2015
Due: Tuesday, 13 October 2015 at 23:59 EST

1 Introduction

This homework will focus on higher order functions and on continuation-passing style

1.1 Getting The Homework Assignment

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

1.2 Submitting The Homework Assignment

Submissions will be handled through Autolab, at `https://autolab.cs.cmu.edu`

In preparation for submission, your `hw/06` directory should contain a file named exactly `hw06.pdf` containing your written solutions to the homework.

To submit your solutions, run `make` from the `hw/06` directory (that contains a `code` folder and a file `hw06.pdf`). This should produce a file `hw06.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 `hw06.tar` file via the "Handin your work" link.

The Autolab handin script does some basic checks on your submission: making sure that the file names are correct; making sure that no files are missing; making sure that your code compiles cleanly. Note that the handin script is *not* a grading script—a timely submission that passes the handin script will be graded, but will not necessarily receive full credit. You can view the results of the handin script by clicking the number corresponding to the "check" section of your latest handin on the "Handin History" page. **If this number is 0.0, your submission failed the check script; if it is 1.0, it passed.**

Remember that your written solutions must be submitted in PDF format—we do not accept MS Word files or other formats.

All the code that you want to have graded for this assignment should be contained in `hw06.sml`, 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.

1.3 Due Date

This assignment is due on Tuesday, 13 October 2015 at 23:59 EST.

1.4 Methodology

You must use the five step methodology discussed in class for writing functions, for **every** function you write in this assignment. Recall the four step methodology:

1. In the first line of comments, specify the type of the function.
2. In the second line of comments, specify via a `REQUIRES` clause any assumptions about the arguments to be passed to the function.
3. In the third line of comments, specify via an `ENSURES` clause what the function computes (what it returns when applied to an argument that satisfies the assumptions in `REQUIRES`).
4. Implement the function.
5. Provide testcases, generally in the format
   `val <return value> = <function> <argument value>`.

For example, for the factorial function presented in lecture:

```
(*  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 6 = fact 3
val 720 = fact 6
```

1.5 Style

For this assignment, we will be grading your submissions based on your coding style. There are several ways that you can learn what is good style and what isn't:

- Your returned and graded homework submissions have been graded for style so use the markups for a reference.

- We have published solution code for the previous assignments, labs, and lectures.
- We have published a style guide at

  `https://www.cs.cmu.edu/~15150/resources/handouts/style.pdf`.

  There is also a copy in the `docs` subdirectory of your git clone.
- You can ask your TAs about specific examples, or post on Piazza asking general questions.

2 Polymorphic Universe

Consider the following functions:

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

fun map1 [ ] f = [ ]
  | map1 (x::L) f = (f x) :: (map1 L) f;

fun map2 [ ] _ = [ ]
  | map2 (x::L) f = (x f) :: (map2 L) f;

fun map3 (f, x::L) = (f x) :: map3 (f, L)
  | map3 (_, [ ]) = [ ];

fun map4 (f, x::L) = (f x) :: map4 f L
  | map4 (_, [ ]) = [ ];
```

Task 2.1 (4 pts). For each of the provided function definitions, what does the ML runtime system say?

Task 2.2 (2 pts). Using the typing rules, explain the reasons for the responses for map3 and map4.

Task 2.3 (4 pts). For each of the following expressions, indicate:

- If it is well typed
- The type if it is well typed
- A short informal summary of the behavior, value, or meaning of the code if it is well typed, or why it is not well typed. Note that non-termination is a behavior.

For example, for `(fn (x,y) => x)` is well typed, with type `'a * 'b -> 'a`, and is a function that when applied, evaluates to the first element of the input tuple.

```
map (fn x => x ^ "!") ["Functions ", "are ", "Values"]

map (fn (x,y) => (x-1,y ^ y)) [(1,"f"),("o",2),("o",3)]
```

3 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);


fun flattenleft L = foldl (fn (x,A) => A@x) [] L;


fun flattenright L = foldr (fn (x,A) => A@x) [] L;
```

We want to show that `flattenright` and `flattenleft` do similar things. More specifically, they produce the same output on reversed lists.

To prove this, we will need a lemma.

Task 3.1 (13 pts).
Prove Lemma 1: 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`

Task 3.2 (12 pts). Prove that for all types `t` and values `L: t list list`

```
flattenleft L = flattenright (rev L)
```

You may need the following lemmas:

- **Lemma b**: for all types `t` and values `x: t`, `L: t list`, `rev (x::L)` = `(rev L)@[x]` and `rev [] = []`
- **Lemma c**: `@` and `rev` are total

Task 3.3 (8 pts). Define recursive functions

```
costleft, costright :  'a list list -> int
```

such that, for any suitably typed value `L`, `costleft L` and `costright L` return the number of cons operations needed to evaluate `flattenleft L` and `flattenright L`, respectively. Recall that evaluating `L1@L2` uses `length L1` cons operations.

Task 3.4 (2 pts).
Estimate the big-O of the number of cons operations needed to evaluate `flattenleft L` and `flattenright L` when L is a list of n lists, each of length n.

Rather than coming up with a recurrence and then solving for the closed form, use the functions from task 3.3 on different sizes of inputs.

4 Short Circuitry

It can be rather annoying that we cannot use `op orelse` or `op andalso` as the function argument for `foldl` or `foldr`. Recall that in SML, `orelse` uses short circuit evaluation, that is, `exp2` is evaluated in `exp1 orelse exp2` if and only if `exp1` evaluates to false.

Task 4.1 (5 pts). Prove or disprove Theorem 1:

Theorem 1: For all types `t1, t2` and values `f : t1 -> bool`, `g : t2 -> bool`, `x : t1`, `y : t2`,

`(fn (a,b) => a orelse b) (f x, g y)` = `f x orelse g y`

5 Big Data

In this problem we consider the task of generating word frequencies from bodies of text.

We will generalize words to be of an arbitrary type with an associated comparison functions. You may think of them as strings, but having them be arbitrary also allows us to deal with characters, integers, etc. easily.
A histogram is a mapping from the words to their counts in a list of words. We introduce the type definition to represent histograms:

```
type 'a hist = ('a * int) list
```

If the words are of type `t` and with comparison function `cmp`, a value `h` of type `t hist` is a histogram iff it has the following properites.

- The words in `h` are sorted in increasing order by `cmp`
- Each word occurs at most once in `h`
- If `(w,c)` occurs in `h`, then `c` is positive
- If a word does not occur in the histogram, it is assumed to have count of 0

For example, for the text

```
["types", "are", "pretty", "useful", "types", "types", "types", "types", "types"]
```

and the standard string comparison function, the histogram will be exactly

```
[("are", 1), ("pretty", 1), ("types", 6), ("useful", 1)]
```

One method to generate a histogram over some text is to break the text into individual words, create a separate histogram for each word, and then merge them all together.

Task 5.1 (10 pts). Define the function

```
merge_histogram :  ('a * 'a -> order) -> ('a hist * 'a hist) -> 'a hist
```

such that if `h1` and `h2` are histograms with `cmp` for the comparison function, then `merge_histogram cmp (h1, h2)` must also be a histogram, such that if word `w` has count of `c1` in `h1` and count `c2` in `h2`, then `(w, c1+c2)` occurs in the merged histogram. Notice that `merge_histogram` is associative and commutative.

Task 5.2 (10 pts). Define the function

```
create_histogram:  ('a * 'a -> order) -> 'a list -> 'a hist
```

such that if `w` appears n times in `L`, then `(w,n)` is in `create_histogram cmp L`. If `w` doesn't appear in `L`, then it should not be in the histogram.

Do not write helper functions specifically for this task, other than anonymous functions to be used as arguments for higher-order functions. You should use `merge_histogram`

Task 5.3 (5 pts). Sometimes, the text we want to create a histogram of is too large to fit in memory. Then if we want to find the histogram for the entire text (here represented as a list of word lists), we need to generate the histogram of each sub-document and merge them together.

Define the function

```
corpus_histogram:  ('a * 'a -> order) -> 'a list list -> 'a hist
```

that generates the histogram of a collection of lists. Do not flatten the list of list or write any helpers. You may use the functions from task 5.1 and 5.2.

6 The Real GridWorld

Given a 2-d grid of nodes, we want to know if there is a path, i.e. list of directions, that we can take to get to a target node from the upper left corner. Our movements are constrained thusly:

- You may only move to the node vertically down and horizontally to the right per step.
- Certain nodes are blocked, so you cannot enter them. These nodes will be labeled `invalid`

We will use the following datatypes:

```
datatype node = valid | invalid | target
datatype dir = down | right
```

We will represent the grids as values of type `node list list`, with the constraint that all of the node lists have the same length, the number of columns in the grid. The first list represents the uppermost row in the grid, and the first element of each list is in the leftmost column.

We will represent the paths as values of type `dir list`

Consider the following grid:

```
[[valid, valid,   valid,  valid  ],
 [valid, invalid, target, valid  ],
 [valid, valid,   valid,  invalid]]
```

In the grid, the only path to the target is two rights followed by a down, represented by `[right, right, down]`

However, in the following grid there is no path to the target because we cannot move through or around the invalid nodes blocking the target.

```
[[valid, valid,   valid,   valid],
 [valid, valid,   invalid, valid],
 [valid, invalid, target,  valid],
 [valid, valid,   valid,   valid]]
```

Task 6.1 (25 pts). Define the function

```
path: node list list -> (dir list -> 'a) -> (unit -> 'a) -> 'a
```

such that `path M s k` is equivalent to `s L`, where `L` is a `dir list` representing a valid path to a target if such a path exists. If no such path exists, the function application is equivalent to `k ()`.
Do not use helper functions other than continuations.

Additional information and hints:

- It may be better to think of `invalid` as a sinkhole where you can't move from rather than somewhere you can't move to.
- The upper left corner may be `invalid`. In that case, there is no path to a target.
- Think of what the recursive subproblems are.
- Think of how moving right will affect the set of potentially reachable nodes.
- `[]` and `[[],[],[],[],[],[],[],[]]` are both empty grids.
- A target may not exist in the grid. In that case there is no path to a target.
- There may be more than one target, and there may be multiple paths to the same target. If this is the case, any of the paths will suffice.
- Feel free to use the function `tl`, especially in conjunction with `map`
- Use the success continuation `s` to build up the result list.
- Use the failure continuation `k` to "go back in time" to check a different path.