15-150 Spring 2016
Homework 06

Out: Wednesday, 17 February 2015
Due: Tuesday, 23 February 2015 at 23:59 EST

1 Introduction

This homework will focus on applications of higher order functions, polymorphism, and continuations.

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.andrew.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.

This homework will be partially autograded. When you submit your code some *very* basic tests are run. These are the public tests, and you will immediately see your score on these. After the final submission deadline, we will run a more comprehensive suite of private tests on your code. Your final score will be a function of your public score, private score, and any manual grading we perform. Non-compiling code will automatically receive a 0.

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, your code will not compile in our environment.

1.3 Due Date

This assignment is due on Tuesday, 23 February 2015 at 23:59 EST. Remember that you may use a maximum of one late day per assignment, and that you are allowed a total of three late days for the semester. You may submit as many times as you would like until the due date.

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 five 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.

Note that if any code you submit for a problem violates our style guidelines, you will receive a 0 for that problem. You will have two weeks from the time the homework is handed back to the class to fix your style and bring your corrected code and handed-back homework to a TA. If your code is satisfactory, you will then receive the grade your original code would have gotten had it satisfied the style guidelines.

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 (3 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 Continuations and Trees

Recall the ML datatype for shrubs (trees with data at the leaves):

```
datatype 'a shrub = Leaf of 'a | Branch of 'a shrub * 'a shrub
```

A function `f : t -> bool` is called *total* iff, for all values `x` of type `t`, there is a value `y` of type `bool` such that `f x` evaluates to `y`. (So `f x` doesn't loop forever, and `f x` doesn't raise any unhandled exception.)

Task 3.1 (10 pts). Write a recursive ML function

```
findOne : ('a -> bool) -> 'a shrub -> ('a -> 'b) -> (unit -> 'b) -> 'b
```

such that for all types `t` and `t'`, all total functions `p : t -> bool`, all values `T : t shrub`, and for all values `s : t -> t'`, `k : unit -> t'`

$$\texttt{findOne p T s k} \cong \begin{cases} \texttt{s v} & \text{where } \texttt{v} \text{ is the leftmost value in } \texttt{T} \text{ such that } \texttt{p v} \cong \texttt{true}, \\ & \texttt{if there is one.} \\ \texttt{k ()} & \text{otherwise} \end{cases}$$

By leftmost, we mean that your function should give priority to the left subtree over the right subtree. For example:

```
val T = Branch(Leaf 1, Leaf 2)
findOne (fn x => x > 0) T s k ≅ s 1
```

Do NOT use any helper functions here other than continuations!

Task 3.2 (15 pts). Write an ML function

```
findTwo : ('a -> bool) -> ('a * 'a -> bool) -> 'a shrub ->
          ('a * 'a ->'b) -> (unit -> 'b) -> 'b
```

such that for all types `t'` and types `t`, all total functions `p : t -> bool`, all shrubs `T : t shrub` with distinct values at the leaves whose values could be checked for equality using `eq`, and all values `s : t * t -> t'`, and
`k : unit -> t'`

$$\texttt{findTwo p eq T s k} \cong \begin{cases} \texttt{s (v1, v2)} & \text{where } \texttt{v1} \text{ and } \texttt{v2} \text{ are two distinct values} \\ & (\texttt{eq(v1,v2)} \cong \texttt{false}) \text{ in } \texttt{T} \text{ such that} \\ & \texttt{p v1} \cong \texttt{true} \text{ and } \texttt{p v2} \cong \texttt{true} \\ \texttt{k ()} & \text{if no such } \texttt{v1,v2} \text{ exist} \end{cases}$$

For example:

```
val T = Branch(Leaf 1, Leaf 2)
findTwo (fn x => x > 0) (fn (x,y) => x = y) T s k ≅ s (1,2)
findTwo (fn x => x = 1) (fn (x,y) => x = y) T s k ≅ k ()
```

NOTE: `findTwo` should NOT be recursive. Think about how you can use `findOne` and pass appropriate continutations as arguments. You should not use `op =` to check for equality in `findTwo`.

4 Tower of Hanoi

The Tower of Hanoi is a classic puzzle in mathematics. It consists of three towers and a number of disks of different sizes which can slide onto any tower. The puzzle starts with the disks in a neat stack in ascending order of size on one tower, the smallest at the top. The objective of the puzzle is to move the entire stack to another tower, obeying the following simple rules:

1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack (i.e. a disk can only be moved if it is the uppermost disk on a stack).
3. No disk may be placed on top of a smaller disk.

We use a new data type `tower` to represent the three towers we have in a Hanoi game.

```
datatype tower = TowerA | TowerB | TowerC
```

Similarly, we represent the disks using data type `disk`

```
datatype disk = Disk of int
```

`Disk n` represents a disk of size n (larger numbers correspond to larger sizes).

Task 4.1 (25 pts). Assume you are given a set of Hanoi Towers, write a recursive ML function in continuation-passing style:

```
hanoi : disk list * tower * tower * tower ->
        ((disk * tower * tower) list -> 'a) -> 'a
```

such that `hanoi (L, tower1, tower2, tower3) k` ≅ `k M` where `L` is a sorted list of disks in ascending size, `M` is a list of `(disk, fromTower, toTower)` tuples, representing the moves needed to move all the disks in `L` from `tower1` to `tower2` via `tower3`.

Taking the trivial example of moving a single disc from `TowerA` to `TowerB` via `TowerC`:

```
hanoi ([Disk 1], TowerA, TowerB, TowerC) k ≅ k [(Disk 1, TowerA, TowerB)]
```

To move two discs from `TowerA` to `TowerB` via `TowerC`:

```
hanoi ([Disk 1, Disk 2], TowerA, TowerB, TowerC) k  ≅
k [(Disk 1, TowerA, TowerC), (Disk 2, TowerA, TowerB), (Disk 1, TowerC, TowerB)]
```

since we need to make the following moves:

1. Move the smaller disc on top of `TowerA` to `TowerC`
2. Move the larger disc from `TowerA` to `TowerB`
3. Move the smaller disc from `TowerC` to `TowerB`

You may reference https://en.wikipedia.org/wiki/Tower_of_Hanoi.

5 Depth Trees

In this question, we will look at depth trees. A depth tree is simply a regular binary tree whose vertices have been augmented with their depth (with the root node having depth 1).

```
datatype 'a depthTree = DEmpty
                      | DNode of 'a depthTree * ('a * int) * 'a depthTree
```

For example,

```
          (2, 1)
         /     \
        /       \
      (3, 2)   (5, 2)
      /
     /
   (1, 3)
```

Task 5.1 (15 pts). A group of friends are making plans for their Spring Break trip. Their trip advisor, a former 15-150 student, provides them a depthTree containing all the possible routes and their associated costs.

Consider the depth tree shown above. There are two routes available in this depthTree: one with 3 stops and total cost of (2+3+1) = 6 (hundred dollars) and the other with 2 stops and total cost of (5+2) = 7 (hundred dollars).

In order not to have a rushed trip, the group now decide that they will choose a route with only a limited number of stops. Besides, they also have a limited budget and do not wish to take a very costly trip.

To help the group find a route in the tree that satisfies their requirement, write a recursive ML function:

```
findRoute: int depthTree -> (int * int) -> ((int * int) -> 'a) ->
           (unit -> 'a) -> 'a
```

such that `findRoute dt (budget, stop) s k` $\cong$ `s (cost, stop')` where `cost` is the total cost in hundreds of dollars and `stop'` is the number of stops of the leftmost route in the tree which does not exceed the budget (`budget`) and the maximum number of stops desired (`stop`); `findRoute dt (budget, stop) s k` $\cong$ `k ()` if no such route exists.

6 Lunchtime

At Gates High School, students are frequently suspended for getting into fights. To reduce this problem, the school principal has tasked the lunch monitors with ensuring that no two students who are enemies sit at the same table at lunch.
We will denote that two students, Anshu and Vincent, are enemies by Anshu # Vincent. Note, # is a relation with the following properties:

1. # is symmetric. If Anshu # Vincent, then Vincent # Anshu.
2. # is anti-reflexive. No student is their own enemy.
3. # may not be transitive. If Anshu # Vincent and Vincent # Steven, we cannot infer that Anshu # Steven.

A student's enemies will be represented by values of the form `type relationship = string * string list`. For example, Anshu # Vincent and Anshu # Steven is represented by `("Anshu", ["Vincent", "Steven"])`.

Each table in the lunchroom is numbered, so we represent the `list` of tables with an `int list`.

We will use values of `type assignment = string * int` to represent the assignment of a student to a table. If Anshu is assigned to table 1, then we would represent this with `("Anshu", 1)`.

Task 6.1 (25 pts). Unfortunately, the lunch monitors have been having great difficulty with this problem and are not sure if it is possible. Help them out by writing the function:

```
seatable : relationship list -> int list ->
           (assignment list -> 'a) -> (unit -> 'a) -> 'a
```

such that

$$\texttt{seatable enemies tables s k} \cong \begin{cases} \texttt{s res} & \text{where res is a list of assignments such that no} \\ & \text{two students who are enemies sit at the same table} \\ \texttt{k ()} & \text{if no such arrangement exists} \end{cases}$$