15-150 Spring 2016
Homework 07

Out: Thursday, 25 February 2016
Due: Tuesday, 1 March 2016 at 23:59 EDT

1 Introduction

This homework will focus on continuations and exceptions.

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/07` directory should contain a file named exactly `hw07.pdf` containing your written solutions to the homework.

To submit your solutions, run `make` from the `hw/07` directory (that contains a `code` folder and a file `hw07.pdf`). This should produce a file `hw07.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 `hw07.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.

Your `hw07.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.

1.3 Due Date

This assignment is due on Tuesday, 1 March 2016 at 23:59 EDT. 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.

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, 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 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 720 = fact 6
```

2 Down the Rabbit-hole

Exceptions allow us to control what a function can and cannot do, and give meaningful feedback when code does something it shouldn't. Perhaps more importantly, we can use `handle` to evaluate an expression that may result in an exception.

For each of the following expressions, determine what the expression evaluates to. If it is not well-typed, say so and explain why. If it raises an exception, state the exception. Assume that `Fail` is the only `exn` defined at top-level.

Task 2.1 (2 pts). `if 3 < 2 then raise Fail "foo" else raise Fail "bar"`

Task 2.2 (2 pts). `(fn x => if x > 42 then 42 else raise Fail "Don't Panic") 16 handle _ => "42"`

Task 2.3 (2 pts).

```
let
  exception Less
  fun bar(x,y) = case Int.compare(y, x) of LESS => raise Less
                                         | _ => y
  fun foo f [] = []
    | foo f [x] = [x]
    | foo f (x::y::L) =
        f(x,y) :: (foo f (x::L)) handle Less => x::(foo f (y::L))
in
  foo bar [3,2,1,42]
end
```

Task 2.4 (2 pts).

```
let
  exception bike
  exception bars
in
  "I can " ^ "ride my " ^ (raise bike) ^ "with no " handle bars =>
  "but I always wear my helmet"
end
```

Task 2.5 (2 pts).

```
(if 15150 > 15251 then 15150
 else raise Fail "epicfail") handle EpicFail => 15150
```

Task 2.6 (3 pts). Describe what the function `mysteryMachine` does when given a `T : int tree`

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

fun mysteryMachine Empty = raise EmptyTree
  | mysteryMachine (Node (L, n, R)) =
      ((mysteryMachine L) + n + (mysteryMachine R)) handle EmptyTree => n
```

3 Looking-Glass Insects

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

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

Consider the following implementation of `findOne` from the previous homework using continuation passing style.

```
fun findOne (p:'a -> bool) (T:'a shrub) (s:'a -> 'b) (k:unit -> 'b):'b =
    case T
     of Leaf(x) => if p(x) then s x else k ()
      | Branch(L,R) => findOne p L s (fn () => findOne p R s k)
```

Recall 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'`

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

Task 3.1 (5 pts). Prove the correctness of `findOne` (meaning `findOne p T s k` will always evaluate to `s(v)` where v is the leftmost element that satisfies p if such an element exists, `k()` otherwise.)

A function `f : 'a -> 'b` is called *weakly total* if, for all values `x` of type `'a`, `f x` either evaluates to a value or raises an exception. (So `f x` doesn't loop forever.)

Consider the following recursive ML function

```
  search : ('a -> bool) -> 'a
  fun search (p : 'a -> bool) (S : 'a shrub) : 'a =
      case S
       of Leaf(x) => if p(x) then x else raise NotFound
        | Branch(L,R) => search p L handle NotFound => search p R
```

Task 3.2 (15 pts). Prove that for all total functions `p` and `S : 'a shrub`, `findOne p S (fn x => x) (fn _ => raise NotFound)` $\cong$ `search p S`.

Task 3.3 (2 pts). Write a wrapper for `search`

```
search' : ('a -> bool) -> 'a option
```

that uses the weakly total function `search` to make a total function that returns `NONE` instead of raising an exception and `SOME x` if an element is found.

4 It's My Own Invention

Alice is venturing across the chess board, fielded with chess pieces, in hopes of checkmating the Red King. We will represent the chess board with a 2-dimensional rectangular $n \times m$ `square list list` where each square is either Free, occupied by the Red King, or occupied by another chess piece. Alice always starts on the chess board, and there is only one Red King.

```
datatype color = White | Red
datatype square = Free | RedKing | Occupied of color

type board = square list list
```

Since the White Knight is on vacation, Tweedledum and Tweedledee will guide Alice instead. However, the bumbling pair can never agree on anything. Tweedledee always wants to go down. Tweedledum always wants to go right. Alice can only choose one of the two. If Alice is at the location (r, c) on the chess board, she can either move to $(r+1, c)$ or $(r, c+1)$. $(0, 0)$ represents the top-left corner.

There are also several pieces on the board. Some of the squares are occupied by friendly White pieces, others are occupied by enemy Red pieces. Alice can only move to a square if it either Free or it is occupied by a friendly piece.

Consider the following grid where Alice starts from (0, 0):

```
[[ALICE, Occupied White, Free,    Free ],
 [Free,  Occupied Red,   RedKing, Free ],
 [Free,  Free,           Free,    Free ]]
```

In the grid, the only path to the target is two rights followed by a down, represented by the list of points, $[(0, 0), (0, 1), (0, 2), (1, 2)]$.

However, in the following setup where Alice starts from (1, 0), there is no path to the target because we cannot move to a square occupied by a defending red piece.

```
[[Free,  Free,           Free,    Free ],
 [ALICE, Occupied Red,   RedKing, Free ],
 [Free,  Occupied White, Free,    Free ]]
```

Task 4.1 (15 pts). Write the function in exception handling style,

```
checkmate : square list list -> (int * int) -> 'a
```

such that `checkmate board start` is equivalent to `raise (CaptureKing L)`, where `L` is a `(int * int) list` representing a valid path from `start` to the Red King (including both the start and destination) if such a path exists. If no such path exists, the function application is equivalent to `raise NoPath`. If the square that Alice starts out on is occupied by a red piece, the function application is equivalent to `raise Jabberwock`.