15-150 Summer 2017
Homework 06

Out: Sunday, 11 June 2017
Due: Friday, 16 June 2017 at 23:59 EST

0 Introduction

This homework will focus on applications of continuation-passing style and how it compares to other styles of writing functions. It will also give you practice with using exceptions.

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/06` directory (that contains a `code` folder. 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.

To submit your written solutions (which *must* be in PDF form), log in to Gradescope and submit your PDF to the assignment **Homework 06**. 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 `hw06.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 Friday, 16 June 2017 at 23:59 EST. 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
```

0.5 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.

- You can ask your TAs about specific examples, or post on Piazza asking general questions.

If any code you submit for a problem violates our style guidelines or is otherwise difficult to understand, you can lose up to 4 points for that problem (that is, you can lose up to 4 points on Task 3.1, 4 points on Task 3.2, etc.). You can lose up to 10 style points overall on a homework assignment.

If you lose style points, you may present to a TA a compelling argument as to why your code has proper style, which, if successful, will grant you the points you lost due to style. Note that the TA you ask will have the final say in whether your code meets the style guidelines sufficiently.

1 There's the Shrub

Recall the 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 1.1 (8 pts). Using continuation-passing style, write a recursive 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'`

$$\texttt{findOne p T s k} = \begin{cases} \texttt{s v} & \text{where v is the leftmost value in T such that p v = true} \\ \texttt{k ()} & \text{if no such v exists} \end{cases}$$

"Leftmost" means that you 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 1.2 (8 pts). Write a 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` and `eq : t * t -> bool`, all values `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'`

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

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 continuations as arguments.

1.1 Get Thee to a Shrubbery!

We can generalize the result of the previous task even further: what if we wanted to find n distinct values `v` of a shrub such that `p v = true`? Unlike the previous task, where we could have our success continuation take in a tuple of type `t * t` for some type `t`, there is no way to define a general n-ary tuple in SML. Instead, we will have our success continuation take in an `'a list`.

We want to write a function

```
findN : ('a -> bool) -> ('a * 'a -> bool) -> 'a shrub -> int ->
        ('a list -> 'b) -> (unit -> 'b) -> 'b
```

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

$$\texttt{findN p eq T n s k} = \begin{cases} \texttt{s L} & \text{where } \texttt{L} = [\texttt{x}_1, \texttt{x}_2, \ldots, \texttt{x}_n], \text{ and for all } i, \texttt{x}_i \text{ is a value in } \texttt{T} \\ & \text{such that } \texttt{p x}_i = \texttt{true} \text{ and for all } j \neq i, \texttt{eq (x}_i\texttt{, x}_j\texttt{)} = \texttt{false} \\ \texttt{k ()} & \text{if no such } \texttt{x}_1, \ldots, \texttt{x}_n \text{ exist} \end{cases}$$

Task 1.3 (8 pts). In your code file, fill in the blanks in the function below so that it meets the above spec:

```
fun findN (p : 'a -> bool) (eq : 'a * 'a -> bool) (T : 'a shrub)
          (n : int) (s : 'a list -> 'b) (k : unit -> 'b) : 'b =
  case n of
       0 => (* Fill in base case *)
     | _ =>
         let
           fun success x = (* Fill in success continuation *)
         in
           findOne p T success k
         end
```

You may add other declarations to the `let` block if they help you write the success continuation, but any additional functions you write may not be recursive. You may not otherwise modify the code.

You may assume that `eq` is an equivalence relation; in particular, you can require that if `eq : t * t` for some type `t`, then for all values `x : t` and `y : t`,

$$\texttt{eq (x, y)} = \texttt{eq (y, x)}$$

2 Proof by Example

Before you start this task, you should make sure your implementation of `findOne` correctly uses continuation-passing style.

Continuations are well and good, but can we prove the correctness of continuation-passing functions in the same way that we prove other functions correct? It is your job to show that we can! (Well, that we can for `findOne`, at least.)

Consider the following non–continuation-passing implementation of `findOne`:

```
fun findOneHelp p (Leaf v) = if p v then SOME v else NONE
  | findOneHelp p (Branch (L, R)) =
      case findOneHelp p L of
           NONE => findOneHelp p R
         | SOME v => SOME v

fun findOne' p T s k =
  case findOneHelp p T of
       NONE => k ()
     | SOME v => s v
```

Convince yourself that `findOne'` satisfies the same informal spec we gave for `findOne`.

We want to show that for all total values `p : 'a -> bool`, all values `T : 'a shrub`, all values `s : 'a -> 'b`, and all values `k : unit -> 'b`,

$$\texttt{findOne' p T s k} = \texttt{findOne p T s k}$$

.

As usual, we will be proving this by structural induction on `T : 'a shrub`. Our base case is `T = Leaf v`. Then,

`findOne' p (Leaf v) s k`	= `case findOneHelp p (Leaf v)...`	1st clause of `findOne'`
	= `case (if p v then SOME v else NONE)...`	1st clause of `findOneHelp`

Since `p` is total and `p v : bool`, we know that either `p v` = `true` or `p v` = `false`.

Case 1: `p v` = `false`.

`findOne' p (Leaf v) s k`	= `case (if false then SOME v else NONE) of...`
	= `case NONE of NONE => k () |...`
	= `k ()`

Case 2: `p v` = `true`.

`findOne' p (Leaf v) s k`	= `case (if true then SOME v else NONE) of...`
	= `case SOME v of NONE => k () | SOME v => s v`
	= `s v`

To finish showing the base case, we would have to show that your implementation of `findOne` evaluates to `k ()` when `p v` = `false` and to `s v` when `p v` = `true`.

But you don't have to show this!

Task 2.1 (15 pts). Assume the inductive hypothesis and prove the inductive step.

Specifically, let P(`T`) be the following property: for all total values `p : 'a -> bool`, all values `s : 'a -> 'b`, and all values `k : unit -> 'b`,

```
findOne p T s k = findOne' p T s k
```

You get to assume P(`L`) and P(`R`), and have to show P(`Branch (L, R)`).

You may use the following lemma without proof:

Lemma 1: For total `p`, `findOneHelp p T` evaluates to a value.

Hint: Let the code guide your proof! Start with `findOne` and `findOne'`, and evaluate both sides until you get code that looks similar. Then, apply the induction hypothesis and the lemma. When applying the lemma, try to reason similarily to the proof for the base case, and things should start working out.

3 Queen of Cubes

In lecture, you solved the n-queens problem, which involved finding the placement of n queens on an $n \times n$ chessboard such that no two queens attack each other. (Recall that two queens attack each other if they are placed in the same column, the same row, or the same diagonal.) Let's add another dimension!

The 3D n-queens problem also involves finding the placement of n queens, but this time it is on an $n \times n \times n$ chessboard. The twist is that when the queens are projected onto the XY-plane, you still get a solution to the 2D n-queens problem. Likewise, when you project the queens onto the XZ- or YZ-plane, you get a solution to the 2D n-queens problem.

If that's confusing, imagine an $n \times n \times n$ box made out of $1 \times 1 \times 1$ glass cubes. In each cube, we are allowed to optionally place a queen, but we must place exactly n queens. After we place the queens, no matter what face of the box we look at, we should see a solution to the 2D n-queens problem.

For the $n = 6$ instance, we have the following solution of (x, y, z) triples:

$$(4, 5, 6), (1, 3, 5), (5, 1, 4), (2, 6, 3), (6, 4, 2), (3, 2, 1)$$

which gives the following 2D projections:

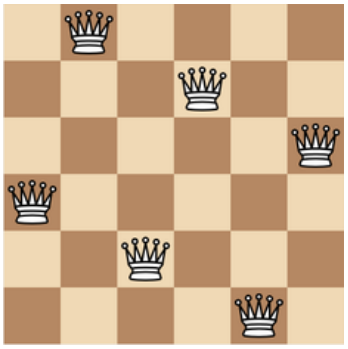

Figure 1: XY projection

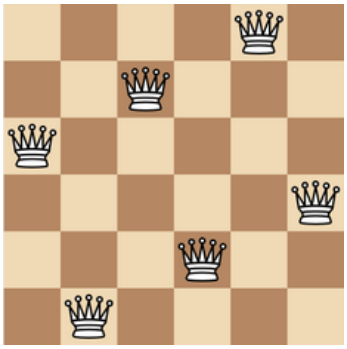

Figure 2: YZ projection

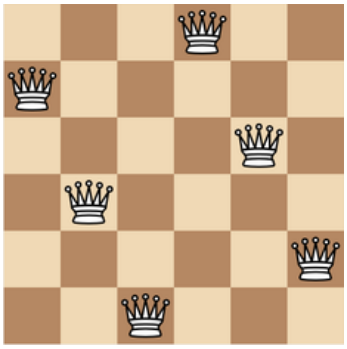

Figure 3: XZ projection

3.1 Carqueentian coordinates

We can represent the position of a queen as a tuple of an x-coordinate, a y-coordinate, and a z-coordinate. The three possible planes can be represented with a datatype:

```
type queen = int * int * int
datatype plane = XY | YZ | XZ
```

If we have a value `(x, y, z) : queen`, to project `(x, y, z)` onto a plane, we simply ignore the coordinate that doesn't exist on that plane. So the projection of `(x, y, z)` onto the `YZ` plane would be `(y, z)`.

Task 3.1 (2 pts). Write a function

```
project : plane -> queen -> int * int
```

such that `project p q` evaluates to the coordinates of the queen `q` when projected onto the plane `p`.

Task 3.2 (3 pts). Using `project`, write a function

```
conflicts : queen -> queen -> bool
```

such that `conflicts q1 q2` evaluates to `true` if `q1` and `q2` mutually attack each other when projected onto the `XY`, `XZ`, or `YZ` plane, and `false` otherwise. For this task, you should also use the code you saw in class to check whether two queens attack each other in two dimensions. (As usual, if you use code from class, there's no need to provide test cases.)

3.2 Conqueenuations

Having done this, we will find a solution to the 3D n-queens problem in a similar way to how we found a solution to the 2D n-queens problem. We will place n queens one at a time, in order. When attempting to place the ith queen at position (x, y, z), we will consider if a placement at that position conflicts with any of the $i - 1$ queens placed so far. If so, we will attempt to place the queen at the next possible position. If not, we will place the queen at that position and try to place the $(i + 1)$th queen. If the position we are trying to place at is off the board, we go back in time with the help of a failure continuation. Once we have placed n queens, we call the success continuation on the list of their placements.

Task 3.3 (20 pts). Using continuation-passing style, write a function

```
nQueens : int -> (queen list -> 'a) -> (unit -> 'a) -> 'a
```

such that for `n` > 0, `nQueens n s k` evaluates to `s L`, where `L` is a list of queen placements that constitute a valid solution to the 3D n-queens problem, or `k ()` if no such list exists. Your coordinates should be 1-indexed.

Some things to keep in mind while writing your solution:

1. Recall from lecture how in two dimensions, it was more efficient to place one queen per row. Similarly, we can imagine dividing the $n \times n \times n$ chessboard into n layers of dimension $n \times n$, where each layer has the same z-coordinate. Then, we can place the ith queen in the ith layer. If you don't do this, your solution may run very slowly! (Note also that you can choose to fix the x- or y- coordinate instead of the z-coordinate.)

2. At least one solution exists for $n = 1$, $n = 6$, $n = 7$, $n = 9$, and $n = 10$. No solution exists for $2 \leq n \leq 5$ or for $n = 8$. For any larger input, your function will probably run slowly.

3. It doesn't matter which particular solution you return, as long as it's a valid solution containing exactly n coordinates. You might want to write a helper function that verifies that the solution you return is a valid solution!

4 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 4.1 (2 pts). `if 3 < 2 then raise Fail "foo" else raise Fail "bar"`

Task 4.2 (2 pts). `(fn x => if x > 42 then 42 else raise Fail "Don't Panic") 16 handle _ => "42"`

Task 4.3 (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 4.4 (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 4.5 (2 pts).

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

Task 4.6 (3 pts). Describe what the function `mysteryMachine` does when given a `T : 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
```

5 That's A Big Number

Think back to the problem of making change. In that problem, we had a list of coin values and we needed to figure out if there was some combination of coins that added up to a target value (where repeats are allowed!).

We now ask you to solve the same problem, but for multiplication, using Continuation Passing Style.

Task 5.1 (15 pts). Write an ML function,

```
makemult: int list -> int -> (int list -> 'a) -> (unit -> 'a) -> 'a
```

such that if a sublist X (possibly with repetitions) of L whose product is n exists, then `makemult L n s k` evalutes to (s X). If such a sublist does not exist, `makemult L n s k` evaluates to (k ()).