15-150 Summer 2016
Homework 07

Out: Monday, 6 June 2016
Due: Thursday, 9 June 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 Thursday, 9 June 2016 at 23:59 EDT. Remember that you have no late days this 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).

```
(case 15150 > 15251 of
   true => 15150
 | false => raise Fail "epicfail") handle EpicFail => 15150
```

Task 2.6 (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 T =
case T of
  Empty => raise EmptyTree
   | (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 shrub -> 'a
  fun search (p : 'a -> bool) (S : 'a shrub) : 'a =
      case S of
         Leaf(x) =>
          (case p(x) of
            true => x
          | false => 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 shrub -> '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

You are traveling through a maze attempting to get to a given target. We will represent this maze with a 2-dimensional rectangular $n \times m$ `square list list` where each index (i,j) in the 2D list is either Free, a Wall, or the Target. You will always start somewhere in the maze and there is only a single target.

```
datatype square = Free | Wall | Target

type board = square list list
```

In addition, you can only move in the east and south directions (rightward and downward). This means that if you are 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.

You are also unfortunately very corporeal so you are not able to go through walls.

Consider the following grid where you start from (0, 0):

```
[[YOU ARE HERE, Free, Free, Free],
 [Free, Wall, Target, 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 you start from (1, 0), there is no path to the target because we cannot move through walls.

```
[[Free, Free, Free, Free],
 [YOU ARE HERE, Wall, Target, Free],
 [Free, Free, Free, Free]]
```

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

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

such that `mazeSolver board start` is equivalent to `raise (Success L)`, where `L` is a `(int * int) list` representing a valid path from `start` to the Target (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 you start on is a wall, the function application is equivalent to `raise CorporealnessSucks`.

5 Regexp

5.1 Extending the Matcher

In class, we introduced six different operators to describe regular expressions:

- The empty set 0
- The empty string 1
- Characters **c**
- Concatenation $r_1 r_2$
- Alternative $r_1 + r_2$
- Repetition r^*

From time to time it is helpful to have some more constructs available to form regular expressions, such as

- Set Intersection $r_1 \cap r_2$, which accepts a string s if and only if s is in $L(r_1)$ and s in $L(r_2)$:

$$L(r_1 \cap r_2) = \{s \mid s \text{ in } L(r_1) \text{ and } s \text{ in } L(r_2)\}$$

- Set Difference $r_1 \setminus r_2$, which accepts a string s if and only if s is in $L(r_1)$ and s is not in $L(r_2)$:

$$L(r_1 \setminus r_2) = \{s \mid s \text{ in } L(r_1) \text{ and } s \text{ is not in } L(r_2)\}$$

The support code for this assignment includes a regular expression matcher `match` very similar to the one from lecture. We have extended the `datatype` definition of `regexp` to include the new constructor `Both` corresponding to $\cap$. (We omitted a new constructor `Diff` corresponding to $\setminus$, since you won't need to implement `Diff`.) Your job is to extend `match` to deal with `Both` and prove parts of the correctness of your implementation. In the notes for Lecture 14, you will find the full statement of the correctness theorem for `match`, including both *soundness* and *completeness*. Here we will ask you to show two cases of the soundness proof. Recall that the overall soundness and completeness theorems are as follows:

Theorem 1 (Soundness)**.** *For all values r :* `regexp`*, cs :* `char list`*, k :* `char list` $\to$ `bool`*, if* `match` *r cs $k \cong$* `true` *then there exist values p, s such that $p@s \cong cs$ with $p \in L(r)$ and $k\ s \cong$* `true`*.*

Theorem 2 (Completeness)**.** *For all values r :* `regexp`*, cs :* `char list`*, k :* `char list` $\to$ `bool`*, if there exist values p, s such that $p@s \cong cs$ with $p \in L(r)$ and $k\ s \cong$* `true` *then* `match` *r cs $k \cong$* `true`*.*

One proves these theorems simultaneously in one large induction proof, with cases for each of the regular expression constructs. You should view your proofs in this assignment as proving two of those cases within a larger overall induction proof. You may assume that termination of the code has already been proven, so that there is no issue about whether continuations loop forever.

We strongly recommend that you think through the correctness spec when you are writing the code for `Both`. If you are stuck on the implementation, try doing the proof of soundness and/or completeness—this will guide you to the answer. However, we will only ask you to hand in the soundness for `Both`.

Your first job is to implement the case of `match` for set intersection $r_1 \cap r_2$, that is, `Both`(r_1, r_2).

Task 5.1 (5 pts). Look at the function `badBoth` (located in `hw08.sml`[1]) which attempts to satisfy the spec for set intersection given above. Explain why it is not correct, and give an example set of inputs where it fails.

Task 5.2 (12 pts). Now implement the case of `match` for set intersection, i.e., `Both`.

Next you will prove the soundness for `Both`(r_1, r_2):

Theorem 3. *For all values cs :* `char list` *and k :* `char list` $\to$ `bool`*,*
if `match` (`Both`(r_1, r_2)) cs k $\cong$ `true`*, then there exist values p, s such that $p@s \cong cs$,*
$p \in L($`Both`*$(r_1, r_2))$, and k $s \cong$* `true`

You may assume the following lemmas without proof:

Lemma 1 (inversion of `andalso`) If `e1` and `e2` are expressions of type `bool` and
if `e1 andalso e2` $\cong$ `true`, then `e1` $\cong$ `true` and `e2` $\cong$ `true`.

Lemma 2 (equivalence) If `e1,e2` are expressions of type `t` (for some `t`) such that
`e1 = e2` $\cong$ `true`, then `e1` $\cong$ `e2`.

Lemma 3 (correctness of `charlisteq`) Consider `L1,L2 : char list`.
If `charlisteq(L1,L2)` $\cong$ `true`, then `L1` $\cong$ `L2`.

Lemma 4 If `p1,s1,p2,s2` are `t lists` (for some type `t`) such that
`p1@s1` $\cong$ `p2@s2` and `s1` $\cong$ `s2` , then `p1` $\cong$ `p2`.

Lemma 5 Suppose `L1:char list` and `L2:char list` are values.
If `L1` $\cong$ `L2`, then `L1` $=$ `L2`.

[1]We accidentally called it `badAlso` in the code file in one location in the specs.

Task 5.3 (18 pts). Prove Theorem 3. Remember this is one case of the larger structural induction proof on regular expressions that was discussed in lecture.

Task 5.4 (5 pts). Look at the function `badDiff` (located in `hw08.sml`) which attempts to satisfy the spec for set difference given above. Explain why it is not correct, and give an example set of inputs where it fails.

We give you the function `accept: regexp -> string -> bool`. Be sure to test your implementation of `Both` thoroughly using this function!

5.2 Secret Message

You will now use your regular expression matcher (your `accept` function) to find a secret message. You will be given a `string list` in which this message is hidden. Each string in the list will be at least 5 characters long. The first character of some strings in the list will be an “f”, a “u” or an “n”, the second character an “a”, an “r”, or an “e”, and the third character a “v”, an “a”, or an “l”. Identifying the strings that follow this format will be the first step in decoding the secret message.

Task 5.5 (2 pts). Define a value `messageKey : regexp` such that the language of `messageKey` is exactly the strings described above that contain part of the message.
So for example `accept messageKey "uel0EoN03U"` ≅ `true` and
`accept messageKey "JXgmFj1GKD"` ≅ `false`

The message is composed of the 5th character of each string in the `string list` that matches the language of `messageKey`.

For example, in the following list (`exampleMessage`), the secret message is `"HELLO WORLD"`. (The strings that conform to the given format have their first three characters in bold and the character they contribute to the message- their 5th character- in bold.)

["**naa**h**H**YDgPh",
"**uel**0**E**oN03U",
"JXgmFj1GKD",
"fektpagFXt",
"**nra**d**L**uXbpS",
"**faa**q**L** qsiw",
"V6d8CgcUbN",
"**nrl**Z**O**atgZ4",
"**ura**8 ru52F",
"**frv**w**W**eE8rx",
"wYJtmVNQxi",
"**fel**u**O** 9TWY",
"**nrv**T**R**oCGjs",
"**faa**C**L**fg67B",
"**frv**Z**D**fYUs0"]

You may find the following functions on strings useful for the next task, but you will probably only need to use some of them:

- `String.sub : string * int -> char`
  `String.sub (s,i)` returns the ith character of s.

  (Caution: By “5th character”, we really do mean the fifth character, starting with the first being “1st”. However `String.sub` is 0-indexed, so use 4 for `i`.)

- `String.str : char -> string`
  Takes a character and returns it as a string.
- `String.implode : char list -> string`
  Takes a character list and returns the string of the characters in the list.

Remember that you have all the higher order functions on lists available to you.

Task 5.6 (6 pts). Write an ML function

```
findMessage : regexp -> string list -> string
```

such that `findMessage messageKey L` returns a string consisting of the 5th character of every string in L which is accepted by the regexp `messageKey`. (Do not change the order of the characters from how they appear in L.)
So `findMessage messageKey exampleMessage` ≅ `"HELLO WORLD"`.
To test your function and regexp (and to complete Task 4.7), first type `Control.Print.stringDepth := 5000;` into SMLNJ (you will only need to do this once before you begin), and then run the command `findMessage messageKey startlist;`

If your function is correct you will find a message with directions. Our lists contain a second message that will appear if your code is slightly incorrect. This is not the real message, but it should help you fix your code.

Feel free to create your own hidden message lists and share them on Piazza.

Task 5.7 (2 pts). Use your function to play the choose-your-own-adventure game beginning with running `findMessage` on startlist, and record what you find in your PDF.