15-150 Fall 2015 Homework 07

Out: Wednesday, 14 October 2015
Due: Tuesday, 27 October 2015 at 23:59 EDT

1 Introduction

In this homework, you will get some practice with continuations and extend the regular expression matcher from lecture, using it to decode a secret message. You will also get some more practice with shrubs and trees.

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.

All the code that you want to have graded for this assignment should be contained in `hw07.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, 27 October 2015 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, 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
```

2 Continuations

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 2.1 (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'`

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

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 2.2 (10 pts). Write an ML function

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

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

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

For example:

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

NOTE: `findTwo` should NOT be recursive. Think about how you can use `findOne` and pass appropriate continutations as arguments.

3 Advanced Path-Finding

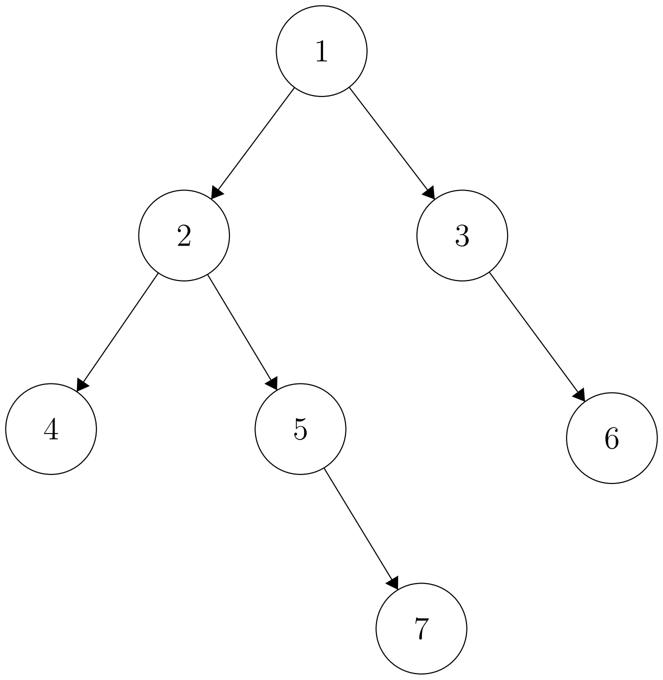
Consider

```
datatype 'a tree = Empty | Node of ('a tree) * 'a * ('a tree).
```

Given a tree and an element of that tree, we should be able to find a path from the root to that element. We can describe that path as a list of directions with type `dir list` where `dir` is defined as

```
datatype dir = Left | Right
```

Example: The direction list of the following tree to `7` would be `[Left,Right,Right]`.



We can create the `dir list` using a continuation-passing style function. You can only take one path in each recursive call, so your challenge in `path` is to encode the path not taken in the failure continuation somehow.

Task 3.1 (10 pts). Implement the function:

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

where `path T x s k` = `s A`, where `A` is a path in `T` to `x` if such a path exists, otherwise `path T x s k` = `k ()`. Notice that for `path T x s k`, `x` is the target element we are trying to find the path to, `T` is the tree, and `s, k` are the success and failure continuations respectively.

If the element is found, call `s A`, where `A` is a value that represents the direction list to that element, and if the element is not found, call `k ()`.

If there are multiple elements that match the target value, your function should find the path to the element that would appear first in the inorder traversal, i.e. the "leftmost" path.

Your function must not do any meaningful work after performing the recursive call.

Do not write any helper functions other than the success and failure continuations.

Task 3.2 (5 pts). Implement the function:

```
optionPath:  ''a tree -> ''a -> (dir list option)
```

where `optionPath T x` returns `SOME` of the direction list to the leftmost occurrence of x in T if it exists, otherwise, `NONE`. You must use `path`.

You must use the `path` function from Task 3.1

Task 3.3 (4 pts). Is the span of `path` less than the work of `path`? Justify why or why not. (You may include recurrences and big-O bounds as justification, but this is not necessary.)

A classic algorithms problem is to find the shortest path between two nodes in a graph. In this section we will consider a special case of this problem on trees. But first we need a helper function.

Task 3.4 (5 pts). Write the function

```
chop :  (dir list * dir list) -> (dir list * dir list)
```

which takes two direction lists and removes the longest common prefix between them.
For example:

```
chop ([Left,Right,Left,Left],[Left,Right,Right,Right,Left]) =
                ([Left,Left],[Right,Right,Left])
```

Task 3.5 (7 pts). Now implement the function

```
pathLength :  ''a tree -> ''a -> ''a -> int
```

which takes a `T : t tree` for some equality type `t`, as well as two values in the tree, `x` and `y`, and evaluates to `k` where `k` is the length of the path connecting them.
You should use the `path` and `chop` functions you wrote in the previous tasks.

Use of **builtin** functions including but not limited to `List.length` or `ListPair.zip` is **allowed** for both `chop` and `pathLength`, and you may write **at most one** helper function of your own in addition to `path` and `chop`.
Assume that all values in the tree are unique and that `x` and `y` are present in the tree.
Note that your solution should run in O(n) time where n is the size of the tree; solutions that do not will receive little or no credit.

4 Regexp

Note: we will go over the code and proofs for `regexp` *in lecture on Tuesday/Thursday; you may want to wait until after Tuesday to begin the next section.*

4.1 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^*

We have also extended the matcher for you to include:

- Wildcard _

Which matches all strings.
From time to time it is helpful to have some more constructs available to form regular expressions, such as

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

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

The support code for this assignment includes a regular expression matcher `match` very similar to the one from lecture.

Your job is to now write a function `diff` that takes two regular expressions `r1 : regexp`, `r2 : regexp`, a `cs : char list`, and a suffix continuation `k : char list -> bool`, such that `diff r1 r2 cs k` evaluates to true if there exist values p, s such that $p@s = cs$ with $p \in L(r_1/r_2)$ and `k s = true`. `diff r1 r2 cs k` evaluates to false, otherwise.

Task 4.1 (5 pts). Look at the function `badDiff` (located in `hw07.sml`) which attempts to satisfy the spec above. Explain why it is not correct, and give an example set of inputs where it fails.

Task 4.2 (12 pts). Now write the function

```
diff :  regexp -> regexp -> char list -> (char list -> bool) -> bool
```

which satisfies the spec above. Think about how you can really leverage the continuation function to check if $cs \in L(r_1)$ and make sure $cs \notin L(r_2)$. Your function must call `match`, and should not be recursive.
We will now prove soundness for `diff`.

Theorem 1 (Soundness). *For all values $r1$: `regexp`, $r2$: `regexp`, cs : `char list`, k : `char list` $\rightarrow$ `bool`, if `diff` $r1\ r2\ cs\ k =$ `true` then there exist values p, s such that $p@s = cs$ with $p \in L(r1/r2)$ and $k\ s =$ `true`.*

You may assume the soundness and completeness of `match` in your proof:

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

Theorem 3 (Completeness). *For all values r : `regexp`, cs : `char list`, k : `char list` $\rightarrow$ `bool`, if there exist values p, s such that $p@s = cs$ with $p \in L(r)$ and $k\ s =$ `true`, then* `match` $r\ cs\ k =$ `true`

You may also assume that termination of the code has already been proven, so that there is no issue about whether continuations loop forever. Also you may assume that all continuations are total.

Task 4.3 (15 pts). Prove Theorem 1. 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` $=$ `true`, then `e1` $=$ `true` and `e2` $=$ `true`.

Lemma 2 (inversion of `not`) If `e1` is an expression of type `bool` and
if `not e1` $=$ `true`, then `e1` $=$ `false`

Lemma 3 (equivalence) If `e1,e2` are expressions of type `t` (for some equality type `t`) such that
`(e1 = e2)` $=$ `false`, then `e1` $\neq$ `e2`.

You should be using both the Soundness and Completenss of `match` in this proof.

4.2 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 “a”, or a “l”, the second character an “y”, or an “m”, the third character a “y”, or an “a”, and lastly the fourth character a “ ”, or an “o”. Identifying the strings that follow this format will be the first step in decoding the secret message.

Task 4.4 (5 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 "amyoEoN03U"` = `true` and
`accept messageKey "JXgmFj1GKD"` = `false`
You will need to use `All`, the wildcard regexp that we've included for you.

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 four characters underlined and the character they contribute to the message- their 5th character- in bold.)

["ayy **H**YDgPh",
"lmao**E**oN03U",
"JXgmFj1GKD",
"fektpagFXt",
"lyyo**L**uXbpS",
"amy **L** qsiw",
"V6d8CgcUbN",
"lyao**O**atgZ4",
"lmy ru52F",
"amao**W**eE8rx",
"wYJtmVNQxi",
"lyyo**O** 9TWY",
"aya **R**oCGjs",
"amyo**L**fg67B",
"ayy **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 *i*th 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 4.5 (10 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 4.6), 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 4.6 (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.