CSA1080/CSA1016
Declarative Programming:
Functional Programming in Haskell

Assignment II

This assignment is worth 7% of the final mark of the Declarative Programming course. A single file, with the code and inline documentation is to be submitted online by midnight of Sunday 6th May, 2007. You can find the online submission system on Joseph Cordina’s website under the Functional Programming link. Hopefully you have picked up your registration ticket by the time you are reading this. You are also to hand in a signed copy of the plagiarism form to Mr Vincent Sammut by Monday 7th May, 2007. Without such a form, your assignment will not be marked. If you have submitted the form found on the Assignment Submission System that applies for all submissions of this course, you do not need to resubmit the plagiarism form. Just in case, the online submission server dies, you can send your assignment by e-mail to gordon.pace(at)um.edu.mt. Assignments handed in late will be marked to zero.

The Department of Computer Science and AI takes a very serious view on plagiarism. Refer to the departmental website on plagiarism for more details:

http://www.cs.um.edu.mt/resources/plagiarism/

Regular Expressions II

The aim of this assignment is to extend the regular expressions parser in Haskell. Before attempting this assignment, reread Assignment 1. In case you got stuck in assignment 1, you will find a model answer to that assignment at the end of this one, which you may use to build upon.

Answer everything.

1. Make RegExp an instance of the Show class to be able to display regular expressions using a more readable syntax (using the syntax used in the informal description of regular expressions given in assignment 1). Use brackets to ensure that the regular expression is shown correctly.

2. Recall the parse function defined in the first assignment, which parsed a string with respect to a regular expression involves writing a function which partially consumes a string, possibly finding no match:
parse :: RegExp -> String -> Maybe (String, String)

However, recall that a regular expression can match in multiple ways with a string. For example, \((a+b)\)* can match the string \(ab\text{bcd}\) in four different ways (the empty string, \(a\), \(ab\) and \(abb\)). To avoid this problem, we adopted a greedy approach to parsing regular expressions in the first assignment.

In this question, we will extend the parser to match in different ways. The final goal of the question is to write the following function:

parses :: RegExp -> String -> [(String, String)]

Unlike \(\text{parse}\), the function \(\text{parses}\) returns a list of possible ways of matching a regular expression. For example:

\[
\text{parses} \ (\text{Star} \ (\text{Chr} \ \text{'}a\text{' => Chr \ 'b'}))) \ "ab\text{bcd}" 
\]

would return:

\[
[(\"", "ab\text{bcd}\"), (\"a\", "bb\text{bcd}\"), (\"ab\", "bc\text{d}\"), (\"abb\", "cd\")]
\]

If the regular expression does not match, \(\text{parses}\) would return the empty list. Define \(\text{parses}\) using pattern matching on the regular expression.

(a) The case for \# always succeeds, returning one result. Define \(\text{parses}\) to work for \#.

(b) Similarly \? almost always succeeds, failing only when the string is empty. Define \(\text{parses}\) to work for ?.

(c) Modify slightly the previous case to work for matching the regular expression \c\.

(d) In the case of \(e+f\), we will have to get all possibilities matching \(e\) and those matching \(f\), and then just return all the results from either of the cases. Once you realise what to do, it’s surprisingly easy. Define \(\text{parses}\) to work for \(e+f\).

(e) The case of \(e;f\) is the most complicated case, since we should parse first the given string with respect to \(e\), which gives you back a list of possibilities. For each of these possibilities, we have to parse the remaining unparsed string with respect to \(f\) and join all the results together. List comprehensions, and the \text{concat} function may be of help.

(f) To parse \(e^*\), we use the same trick as when we were defining \(\text{parse}\). We simply parse it as though we were parsing \(e;e^* + \#\). The recursion would finally stop when the whole input is consumed, or it fails.
3. Now use the `parses` function to define a new function `matches`, which given a regular expression and a string, returns whether the whole string can match the regular expression (ie leaving no part of the string unparsed). Note that not all results of `parses` need to consume the whole string, for this to return true.

If you get stuck in one of the questions, continue answering the following ones, even if they may not work well due to the earlier problems. If some things are unclear or you have some major problems, you can post your query on the discussion group at [http://groups.google.com/group/functional-programming-2007](http://groups.google.com/group/functional-programming-2007).

You need a gmail account to post your queries. Just in case you do not have one, e-mail Joseph Cordina at joseph.cordina(at)um.edu.mt or Gordon Pace at gordon.pace(at)um.edu.mt. Do not e-mail us directly with queries since we will not answer. Also do not post code or partial answers on the discussion board, since this will be penalized.

Good luck!
Model answer for Assignment I

import Data.Maybe

data RegExp
  = Nil -- corresponds to #
  | Any -- corresponds to ?
  | Chr Char -- corresponds to c
  | RegExp :+: RegExp -- corresponds to e + f
  | RegExp :>: RegExp -- corresponds to e ; f
  | Star RegExp -- corresponds to e*

multiple :: Int -> RegExp -> RegExp
multiple 1 e = e
multiple n e = e :>: multiple (n-1) e

minLen :: RegExp -> Int
minLen Nil = 0
minLen Any = 1
minLen (Chr _) = 1
minLen (e :+: f) = min (minLen e) (minLen f)
minLen (e :>: f) = minLen e + minLen f
minLen (Star _) = 0

parse :: RegExp -> String -> Maybe (String, String)
parse Nil s = Just ("", s)
parse Any [] = Nothing
parse Any (c:cs) = Just ([c], cs)
parse (Chr x) [] = Nothing
parse (Chr x) (c:cs)
  | x == c = Just ([c], cs)
  | otherwise = Nothing
parse (e :+: f) s
  | isNothing parse_e = parse_f
  | isNothing parse_f = parse_e
  | otherwise = longest (fromJust parse_e) (fromJust parse_f)
  where
    parse_e = parse e s
    parse_f = parse f s

longest (match1, rest1) (match2, rest2)
  | length match1 > length match2 = Just (match1, rest1)
  | otherwise = Just (match2, rest2)

parse (e :>: f) s =
case parse e s of
  Nothing -> Nothing
  Just (match_e, rest_e) ->
    case parse f rest_e of
      Nothing -> Nothing
Just (match_f, rest_f) -> Just (match_e++match_f, rest_f)

parse (Star e) s = parse ((e :+: Star e) :+: Nil) s

match :: RegExp -> String -> Bool
match e s =
    case parse e s of
        Nothing    -> False
        Just (_,rest) -> rest == ""