This assignment is worth 8% 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 24th March 2007. You are also to hand in a signed copy of the plagiarism form to Mr Vincent Sammut by Monday 25th March 2007. Without such a form, your assignment will not be marked. 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/

*Answer everything.*

The aim of this assignment is to build a regular expression parser in Haskell. Regular expressions are frequently used in programming and text editors to search for and manipulate text. You will be seeing more of regular expressions in the B.Sc. IT course.

Regular expressions take one of the following forms:

- **c** (where *c* is any character) matches only with the specified character.
- **?** matches with any single character.
- **e ; f** (where both *e* and *f* are themselves regular expressions) matches with a string which matches regular expression *e*, followed by regular expression *f*.
- **e + f** (where both *e* and *f* are themselves regular expressions) matches with a string which either matches regular expression *e*, or regular expression *f*.
- **e* (where *e* is itself regular expressions) matches any number of repetitions (possibly zero) of regular expression *e*. 
• # matches the empty string.

The concept can be better explained via examples:

• The regular expression a ; b matches with the string ab.

• The regular expression a ; (b + c)* ; a matches with strings which start and finish with an a, and with any number of bs and cs in between, such as abcabcaca or aba, or even aa.

• To illustrate the behaviour of ?, consider the regular expression a ; ?* ; a which matches any string that both starts and finishes with an a, and anything in between, such as abcabcaca or aringa, or even aa.

• The regular expression a ; (bc + cb + #) ; d would match a string starting with a, ending with d, and with bc, cb or nothing in between (ie abcd, acbd, ad).

• A digit (single number) can be described by 0+1+2+3+4+5+6+7+8+9. Let's call this regular expression DIGIT.

• Positive numbers can be described by DIGIT . DIGIT*. Note that we write this to ensure that it contains at least one digit. Let's call this NUM.

• Floating point numbers (with a decimal point) can now be described by NUM ; . ; NUM.

• Finally, valid variable names in various languages contain only alphabetic symbols, digits and the underscore symbol, with the additional constraint that they do not start with a digit. If the regular expression matching an alphabetic symbol a+b+...+z is named ALPHA, we can define variable names as ALPHA ; (ALPHA + DIGIT + _)*.

The most straightforward use of a regular expression is to try to match it with the prefix of a given string. Thus, for example, trying to match the regular expression a ; (b + c)* ; a with the string "abccba" would match the prefix "abccba" leaving "bcbcbcbcbcb" unparsed. Therefore, parsing a string with respect to a given regular expression in such a manner would return a pair of strings. However, as you know, life is not so simple...

Firstly, parsing does not always succeed. For example, regular expression a;b*;c does not match string "abbbbddec". Therefore, the result of parsing can be either a pair of strings (as already explained), or an error indicating no correct match. One way of encoding this in Haskell is to encode the result in the type Maybe (String, String). And there's yet another complication ...

Consider trying to parse the string "bcdefg" with regular expression b;c;d + b;c. This can match in two different ways — either matching "bcd", leaving "ef"
unparsed, or matching "bc", leaving def unparsed. Similarly, matching a* to string "aaaha" can match in multiple ways: matching with the empty string "", leaving "aaaha" or matching "a", leaving "aaha" or matching "aa", leaving "aha", or even matching "aaa", leaving "ha". To simplify matters, in both cases we take the ‘correct’ parse to be the longest matching string. Since we apply this rule with every + and * operator, and we parse from left to right, this sometimes leads to anomalous results. For example, matching regular expression a*;a;b to string aab will not match since the a* will greedily match all the initial as, leaving none to match the second part of the regular expression. Do not worry too much about this — you’ll be shown how to solve when the time is right.

In Haskell we can describe regular expressions using an abstract datatype:

```
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*
```

Write a function `multiple`, which given a number n and a regular expression e, returns a new regular expression that corresponds to regular expression e repeated n times in sequence. For example `multiple 3 (Star Any)` would return `Star Any :>: Star Any :>: Star Any`.

The minimum length of a string matching a regular expression can be calculated recursively. Clearly, ? and c always match a string of length 1. The regular expression e+f matches with a string which is at least as long as the smaller of the minimum lengths matched by e and f. The star operator e* can match the empty string, and thus has a minimum length of zero. Finally, with e;f the minimum length would be the sum of the minimum lengths. Define a function `minLen` which given a regular expression calculates the minimum length of a matching string.

As already explained, parsing 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)
```

Define `parse` using pattern matching on the regular expression. The operators ? and c are easy to define. The case of e;f is slightly more complicated, since you should parse (recursively) the given string with respect to e and then parse the remaining unparsed string with respect to f (again, recursively). If the
parsing of \( e \) fails, then the parse of \( e;f \) would completely fail, too. In the case of \( e+f \), you will have to try both \( e \) and \( f \), and then choose the one with the longest match (failing if both parses fail). Finally, the most complex operator is \( e^* \). The trick is to parse it as though you are parsing \( e;e^* + # \). The recursing would finally stop when the whole input is consumed.\(^1\)

Now use the \texttt{parse} function to define a new function \texttt{match}, which given a regular expression and a string, returns whether the whole string matches the regular expression (ie leaving no part of the string unparsed).

Good luck!

\(^1\)There is one case in which this would not work, but for the assignment we do not really care! Can you identify it?