COMP3141 Assignment 2

Haskell-Augmented Regular Expressions (H.A.R.E.)
Version 1.1

Liam O’Connor

Semester 1, 2018

Marking  Total of 20 marks (20% of practical component)

Due Date  Friday, 25th May, 2018, 23:55

Late Penalty  The maximum available mark is reduced by 10% if the assignment is one day late, by 25% if it is 2 days late and by 50% if it is 3 days late. Assignments that are late 4 days or more will be awarded zero marks. So if your assignment is worth 88% and you submit it one day late you still get 88%, but if you submit it two days late you get 75%, three days late 50%, and four days late zero.

Submission Instructions  The assignment can be submitted using the ‘give’ system.

To submit from a CSE terminal, type:

$ give cs3141 Hare Hare.hs

Overview

In this assignment, you will refactor a text-matching regular expressions engine to use a more strongly-typed interface. These stronger types not only reduce the possibility for bugs, but also improve the readability and clarity of the code base. On top of the strongly-typed version, you will implement a number of additional convenience combinators for working with regular expressions. This assignment will give you experience using Haskell type system extensions (specifically GADTs), experience programming with monads and applicative functors, and introduce you to concepts such as alternatives.
**Provided Code**

The provided code consists of a number of modules:

- **Hare.hs** contains the stubs of the code you should implement.
- **Untyped.hs** contains an untyped, unextended version of the regex engine.
- **HareMonad.hs** contains the monadic type used to write the matching algorithm.
- **Tests.hs** contains the main function for running tests.
- **Tests/Support.hs** contains the support code for running tests.
- **Tests/UnitTests.hs** contains properties for the basic regular expressions.
- **Tests/Transcript.hs** contains some acceptance tests for your combinators, for analysing a UNSW transcript.
- **Tests/Examples.hs** contains all examples from this spec, as unit tests.

**Note**: The only file you can submit is ‘**Hare.hs**’, so make sure your submission compiles with the original versions of all other files.

**Regular Expressions**

Regular expressions are a popular way to describe and manipulate patterns of strings. Often they are used to search for a substring that matches a particular pattern, and then extracting information from that substring.

Many languages include regular expressions as a built in feature or in their standard library. If you have done COMP2041, you will have been exposed to several such languages. Unfortunately, Haskell is not such a language. To remedy this, Helen the Hare started to design H.A.R.E, a regular expressions library for Haskell.

She started by encoding the basic building blocks of regular expressions as a Haskell data type.

```haskell
data RE = Empty               -- Matches the empty string
               | Fail                 -- Matches no strings
               | Char [Char]         -- Matches a single character from the list
               | Seq RE RE           -- Matches the two expressions in sequence
               | Choose RE RE        -- Matches either the first, or the second
               | Star RE             -- Matches the expression zero or more times
```

**Note**: If you have seen regular expressions before, you may be used to more features being available than the ones above. In general, those features can be translated into the features given above.

Some simple examples of these regular expressions are given in the table below.
In order to define a matching algorithm for regular expressions, Helen defined a type called \texttt{Hare} (found in \texttt{HareMonad.hs}), which is similar to \texttt{State String} except that the return type is wrapped in a monad \texttt{f}:

\begin{verbatim}
newtype Hare f a = Hare { runHare :: String -> f (String, a) }

The \texttt{hare} function can be used to “run” a \texttt{Hare} computation on a string:

\begin{verbatim}
hare :: Functor f => Hare f a -> String -> f a
hare a s = fmap snd (runHare a s)
\end{verbatim}

Helen has parameterised the \texttt{Hare} type by \texttt{f} so that it could be instantiated to \texttt{Maybe} if the user only wishes to find the \textit{first} match for a particular regular expression, or to the list type constructor \texttt{[]} if the user wishes to find \textit{all possible} matches of the regular expression. This requires us to make use of the \texttt{Alternative} type class, which has two built in methods:

\begin{verbatim}
class Applicative f => Alternative f where
  empty :: f a
  (<>|>) :: f a -> f a -> f a
\end{verbatim}

The \texttt{empty} method denotes \textit{failure}. For the list instance, it is the empty list \texttt{[]} , whereas for the \texttt{Maybe} instance it is \texttt{Nothing}. The \texttt{(<|>)} method denotes \textit{alternation}. For the list instance it is the same as concatenation (\texttt{++}), whereas for the \texttt{Maybe} instance, it is a left-biased choice, like so:

\begin{verbatim}
instance Alternative Maybe where
  empty = Nothing
  Just a <|> b = Just a
  Nothing <|> b = b
\end{verbatim}
To avoid confusing `empty` with the `Empty` constructor for regular expressions, Helen also defined an alias for `empty`, called `failure`:

```haskell
failure :: (Alternative f, Monad f) => Hare f a
failure = empty
```

The `readCharacter` action is also of note, which removes a character from the `String` state and returns it. If there are no characters left in the string, it will fail:

```haskell
readCharacter :: (Alternative f, Monad f) => Hare f Char
readCharacter = Hare $ \s -> case s of
  [] -> empty
  (x:xs) -> pure (xs,x)
```

Armed with just `readCharacter` and the `Monad`, `Applicative` and `Alternative` instances for `Hare`, Helen defines the matching algorithm, in a function called `match` (found in `Untyped.hs`):

```haskell
match :: (Monad f, Alternative f) => RE -> Hare f Results
```

Here the `Results` type describes the string that was matched by each part of the regular expression:

```haskell
data Results = None
  | Character Char
  | Tuple Results Results
  | Repetition [Results]
```

Seeing as the `Empty` expression matches any string, Helen simply returns `None` in that case:

```haskell
match Empty = pure None
```

Conversely, because `Fail` matches no strings at all, Helen always calls `failure` if attempting to match this expression:

```haskell
match Fail = failure
```

To define the case for `Char`, Helen uses the built-in function `guard`, which will fail if the given condition evaluates to false:

```haskell
match (Char cs) = do
  x <- readCharacter
  guard (x `elem` cs)
  pure (Character x)
```

To define the case for `Seq`, Helen makes use of the `Monad` instance, first matching on the first expression `a`, then the second `b`: 

```haskell
match (Seq a b) = do
  xa <- match a
  xb <- match b
  pure (Tuple xa xb)
```
match (Seq a b) = do
  ra <- match a
  rb <- match b
  pure (Tuple ra rb)

Because there is no dependencies between ra and rb, this could also have been defined using the Applicative instance, as follows:

match (Seq a b) = Tuple <$> match a <*> match b

For the case for Choose, Helen makes use of the Alternative (<|>) method:

match (Choose a b) =
  match a <|> match b

For Star, Helen observed the following bijection\[1\]

\[
\text{Star } r \iff (r \text{ 'Seq' Star } r) \text{ 'Choose' Empty}
\]

For this reason, Helen implements Star a by first matching the expression a, then matching on Star a recursively. If either match fails, it falls back to returning the empty match:

match (Star a) =
  addFront <$> match a <*> match (Star a) <|> pure (Repetition [])
where
  addFront x (Repetition xs) = Repetition (x:xs)
  addFront _ _ = error "(should be) impossible!"

Having defined match, Helen can now define the regex matching operator (= ), which searches through a string for a match to the expression:

\[
(\sim) :: (\text{Monad } f, \text{Alternative } f) \to \text{String} \to \text{RE} \to f \text{ Results}
\]

\[
\text{str } \sim \text{ re } = \text{ hare matchAnywhere str}
\]

where matchAnywhere = match re <|> (readCharacter >> matchAnywhere)

If we look at some examples, we can see that invoking it with the Maybe monad returns just the first result, whereas the list version returns all results:

*> "COMP3141" \sim (Char \[\text{\text{'0'}..\text{'9'}\}]) :: Maybe Results
Just (Character '3')
*> "COMP3141" \sim (Char \[\text{\text{'0'}..\text{'9'}\}]) :: [Results]
[Character '3', Character '1', Character '4', Character '1']

\[1\]While they return different results, there is a bijection between them
A Typed Implementation (12 Marks)

Helen’s untyped implementation works, but it’s not very clean. For example, we know that the Results returned by the regular expression

```
Char ['0'..'9'] 'Seq' Star (Char ['0'..'9'])
```

will be of the format:

```
Tuple (Character c) (Repetition [Character x, Character y, ...])
```

but any function designed to process the Results from this regular expression would have to be partial, because we did not encode the format of the results returned into the type system.

Instead of having a Results type, your first task will be to define a type-indexed version of the RE type and associated match function, in Hare.hs.

```haskell
data RE :: * -> * where
  Empty :: RE ()
  Fail :: RE a
  -- Your constructors here
```

Each constructor has an additional type argument which specifies the type of the result returned from each expression. We have defined the first two constructors for you. You will have to find appropriate definitions for the rest of them.

For implementing match, feel free to look at Helen’s original implementation in the Untyped.hs module as a guide. The test cases provided may also be instructive in getting your implementation type-checking and correct.

<table>
<thead>
<tr>
<th>Marks</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Empty type correct; match implementation correct.</td>
</tr>
<tr>
<td>2</td>
<td>Fail type correct; match implementation correct.</td>
</tr>
<tr>
<td>2</td>
<td>Char type correct; match implementation correct.</td>
</tr>
<tr>
<td>2</td>
<td>Seq type correct; match implementation correct.</td>
</tr>
<tr>
<td>2</td>
<td>Choose type correct; match implementation correct.</td>
</tr>
<tr>
<td>2</td>
<td>Star type correct; match implementation correct.</td>
</tr>
<tr>
<td>12</td>
<td>Total</td>
</tr>
</tbody>
</table>

6
Adding Actions (1 Marks)

Next, we will add an additional constructor to RE that does not change the set of strings matched, but allows us to run an arbitrary transformation on the results returned from the match:

\[
\text{Action :: (a -> b) -> RE a -> RE b}
\]

Your task in this part is to implement the `match` function for this constructor. You may find the `Functor` instance for `Hare` useful.

Examples

*\[ *** =^ Action \text{length} \ (\text{Star} \ (\text{Char} \ ['*'])) :: [\text{Int}] \]
\[ [3, 2, 1, 0, 2, 1, 0, 1, 0, 0] \]

*\[ **A**xax =^ Action \text{isUpper} \ (\text{Char} \ ['a', 'A']) :: [\text{Bool}] \]
\[ [\text{True}, \text{False}] \]

*\[ \text{let } f \ (x, y) = [x, y] \]
*\[ \text{let atoz = Char} \ ['a'..'z'] \]
*\[ **ab01cd20 =^ Action f \ (atoz \ 'Seq' \ atoz) :: [\text{String}] \]
\[ ['ab', 'cd'] \]

<table>
<thead>
<tr>
<th>Marking Criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Marks</strong></td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>
Utility Combinators (7 Marks)

Using our newly minted Action constructor, you must now define a series of combinator functions that allow us to define a lot of the features people normally expect to find in a regular expressions system.

cons function

The cons function matches a regular expression for an element of type a, followed by a regular expression for a list [a], and returns that element prepended to the list.

\[
\text{cons} :: \text{RE } a \rightarrow \text{RE } [a] \rightarrow \text{RE } [a]
\]

Examples:

\[
\begin{align*}
\text{"10100" } & \sim \text{ cons (Char } [\text{'}1\text{'}]) (\text{Star (Char } [\text{'}0\text{'}])) :: [\text{String}] \\
["10","1","100","10","1"] \\
\text{"10100" } & \sim \text{ cons (Char } [\text{'}1\text{'}]) (\text{Action (const } []) \text{ Empty}) :: [\text{String}] \\
["1","1"]
\end{align*}
\]

plus function

The plus function is like Star, but requires the expression given to occur \textit{at least once}. Thus, this function can be defined using Action, Seq and Star.

\[
\text{plus} :: \text{RE } a \rightarrow \text{RE } [a]
\]

Examples:

\[
\begin{align*}
\text{"10100" } & \sim \text{ plus (Char } [\text{'}0\text{'}]) :: [\text{String}] \\
["0","00","0","0"] \\
\text{let atoz } = \text{ Char } [\text{'}a\text{'}..\text{'}z\text{'}] \\
\text{let digits } = \text{ Char } [\text{'}0\text{'}..\text{'}9\text{'}] \\
\text{"ab1c3" } & \sim \text{ plus (atoz } \text{ Seq digits) :: [[[Char,Char]]]} \\
[[\text{'(b','1')},\text{'(c','3')}],[[\text{'(b','1')},\text{'(c','3')}]]
\end{align*}
\]

string function

The string function produces a regex that only matches the given string, and returns it. You should be able to implement it using Char, cons, Empty and Action.

\[
\text{string} :: \text{String } \rightarrow \text{RE } \text{String}
\]

Examples:

\[
\begin{align*}
\text{let comp3141 } = \text{ string } "\text{COMP3141}" \\
\text{"My favourite subject is COMP3141" } & \sim \text{ comp3141 :: Maybe String} \\
\text{Just } "\text{COMP3141}" \\
\text{"My favourite subject is MATH1141" } & \sim \text{ comp3141 :: Maybe String} \\
\text{Nothing}
\end{align*}
\]
choose function

The choose function is like the Choose constructor, but generalised to lists. That is, choose [a, b] is equivalent to a ‘Choose’ b. What should choose [] be?

choose :: [RE a] -> RE a

Examples:

```haskell
*> let re = choose [string "COMP", string "MATH", string "PHYS"]
*> "COMP3141, MATH1081, PHYS1121, COMP3121" =~ re :: [String]
["COMP","MATH","PHYS","COMP"]

*> "abc" =~ choose [] :: Maybe String
Nothing
```

option function

The option function matches the given expression zero or one times. In other words, if the expression matches, it returns Just that match, otherwise Nothing. You can implement this combinator with Action, Choose and Empty.

option :: RE a -> RE (Maybe a)

Examples:

```haskell
*> let digits = Char ['0'..'9']
*> let sign = Action (fromMaybe '+') (option (Char ['-']))
*> "-30 5 3" =~ (sign 'cons' plus digits) :: [String]
["-30","-3","+30","+3","+0","+5","+3"]

*> "foo" =~ option (Char ['a']) :: [Maybe Char]
[Nothing,Nothing,Nothing,Nothing]
```

rpt function

The rpt combinator allows a regular expression to be repeated a fixed number of times. The expression rpt n x is equivalent to repeating x in sequence n times, returning the results in a list.

rpt :: Int -> RE a -> RE [a]

Examples:

```haskell
*> let digits = Char ['0'..'9']
*> let programs = choose [string "COMP", string "PHYS", string "MATH"]
*> let courseCode = programs 'Seq' rpt 4 digits
*> "COMP3141, MATH1081, and PHYS1121" =~ courseCode :: [(String,String)]
```
```
["COMP","3141"],("MATH","1081"),("PHYS","1121")]

>> "foo" =~ rpt 0 (Char ['a']) :: Maybe [Char]
Just ""

`rptRange` function

Lastly, the `rptRange` function takes a range of numbers (x, y). You may assume that x ≤ y. It will match the given regex at least x times and at most y times.

`rptRange :: (Int, Int) -> RE a -> RE [a]`

Examples:

>> "1234" =~ rptRange (2,4) (Char ['0'..'9']) :: [String]
["12","123","1234","23","234","34"]

>> "1234" =~ rptRange (3,3) (Char ['0'..'9']) :: [String]
["123","234"]

<table>
<thead>
<tr>
<th>Marks</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>cons implementation correct.</td>
</tr>
<tr>
<td>1</td>
<td>plus implementation correct.</td>
</tr>
<tr>
<td>1</td>
<td>string implementation correct.</td>
</tr>
<tr>
<td>1</td>
<td>choose implementation correct.</td>
</tr>
<tr>
<td>1</td>
<td>option implementation correct.</td>
</tr>
<tr>
<td>1</td>
<td>rpt implementation correct.</td>
</tr>
<tr>
<td>1</td>
<td>rptRange implementation correct.</td>
</tr>
<tr>
<td><strong>7</strong></td>
<td><strong>Total</strong></td>
</tr>
</tbody>
</table>
Compiling and Building

In addition to the standard library, this project depends on the QuickCheck and HUnit testing libraries and the test framework called tasty. For CSE machines, we have already configured a package database on the course account that should build the assignment without difficulty using the standard Haskell build tool cabal. For students using their own computer, we instead recommend the alternative build tool stack, available from the Haskell Stack website at [https://www.haskellstack.org](https://www.haskellstack.org). We have provided a stack configuration file that fixes the versions of each of our dependencies to known-working ones. If you use stack to set up your toolchain, you should have minimal compatibility difficulties regardless of the platform you are using. If you are using versions of GHC or Haskell build tools supplied by your Linux distribution, these are commonly out of date or incorrectly configured. We cannot provide support for these distributions.

If you are using a Mac computer, you may be interested in using the Haskell for Mac IDE. The Lecturer in Charge has access to special (free) student licenses for COMP3141 students, so contact the lecturer for more information.

Detailed, assignment-specific instructions for each build tool are presented below.

On CSE Machines

Enter a COMP3141 subshell by typing 3141 into a CSE terminal:

```
$ 3141
  newclass starting new subshell for class COMP3141...
```

From there, if you navigate to the directory containing the assignment code, you can build the assignment by typing:

```
$ cabal build
```

To run the program from ‘Tests.hs’, which contains all the unit, acceptance, and property tests, type

```
$ ./dist/build/HareTests/HareTests
```

To start a ghci session, type:

```
$ cabal repl
```

Lastly, if for whatever reason you want to remove all build artefacts, type:

```
$ cabal clean
```

For stack users

Firstly, ensure that GHC has been setup for this project by typing, in the directory that contains the assignment code:

```
$ stack setup
```
If `stack` reports that it has already set up GHC, you should be able to build the assignment with:

```bash
$ stack build
```

This `build` command will, on first run, download and build the library dependencies as well, so be sure to have an internet connection active. To run the program from ‘Tests.hs’, which contains all the unit, acceptance, and property tests, type

```bash
$ stack exec HareTests
```

To start a `ghci` session, type:

```bash
$ stack repl
```

Lastly, if for whatever reason you want to remove all build artefacts, type:

```bash
$ stack clean
```

**For Haskell for Mac users**

The provided Haskell for Mac code bundle should include everything you need, however you will need to install the command line tools and the libraries used by following these instructions:

- In the `Preferences` menu, go to the `Command Line` tab and click the `Download Installer` button. This should get you to the website where you can download the HaskellCLI Installer. Once it is installed, there should be a `Target` option in the menu bar.

- Go to `Target`, then `Package Management`. Install the following three packages:
  - `tasty`
  - `tasty-quickcheck`
  - `tasty-hunit`

- You should now be able to run tests by going to `Tests.hs` and then going to `Target`, and `Run`.

**Marking and Testing**

**Testing**

A number of different test suites are provided:

- The `Tests/Transcript.hs` file contains some high-level acceptance tests that extract various bits of information from an anonymised UNSW student transcript, located in `Tests/transcript.txt`. 
• The Tests/UnitTests.hs file contains some QuickCheck properties that, while not complete specifications, will help you to gain some assurance for your implementation of `match` for each constructor.

• The Tests/Examples.hs file contains each example presented in this spec document as a unit test.

• The Tests.hs file contains a `tasty` test suite containing all of the above tests.

Marking

All marks for this assignment are awarded based on automatic marking scripts, which are comprised of QuickCheck properties, acceptance tests, and unit tests. Marks are not awarded subjectively, and are allocated according to the criteria presented in each section.

Barring exceptional circumstances, the marks awarded by the automatic marking script are final. For this reason, please make sure that your submission compiles and runs correctly on CSE machines. We will use similar machines to mark your assignment.

A dry-run script that runs the tests provided in the assignment code will be provided. When you submit the assignment, please make sure the script does not report any problems.

Late Submissions

Unless otherwise stated if you wish to submit an assignment late, you may do so, but a late penalty reducing the maximum available mark applies to every late assignment. The maximum available mark is reduced by 10% if the assignment is one day late, by 25% if it is 2 days late and by 50% if it is 3 days late. Assignments that are late 4 days or more will be awarded zero marks. So if your assignment is worth 88% and you submit it one day late you still get 88%, but if you submit it two days late you get 75%, three days late 50%, and four days late zero.

Extensions

Assignment extensions are only awarded for serious and unforeseeable events. Having the flu for a few days, deleting your assignment by mistake, going on holiday, work commitments, etc do not qualify. Therefore aim to complete your assignments well before the due date in case of last minute illness, and make regular backups of your work.

Plagiarism

Many students do not appear to understand what is regarded as plagiarism. This is no defense. Before submitting any work you should read and understand the UNSW plagiarism policy [https://student.unsw.edu.au/plagiarism](https://student.unsw.edu.au/plagiarism).
All work submitted for assessment must be entirely your own work. We regard unacknowledged copying of material, in whole or part, as an extremely serious offence. In this course submission of any work derived from another person, or solely or jointly written by and or with someone else, without clear and explicit acknowledgement, will be severely punished and may result in automatic failure for the course and a mark of zero for the course. Note this includes including unreferenced work from books, the internet, etc.

Do not provide or show your assessable work to any other person. Allowing another student to copy from you will, at the very least, result in zero for that assessment. If you knowingly provide or show your assessment work to another person for any reason, and work derived from it is subsequently submitted you will be penalized, even if the work was submitted without your knowledge or consent. This will apply even if your work is submitted by a third party unknown to you. You should keep your work private until submissions have closed.

If you are unsure about whether certain activities would constitute plagiarism ask us before engaging in them!