THE UNIVERSITY OF HONG KONG

COMP3258: FUNCTIONAL PROGRAMMING

Assignment 2 (Data Types and Parser)

Deadline: 23:55, November 8th, 2020 (HKT)

In this assignment, you should use the parser library provided in `Parser.hs` to implement the functions specified in the questions.

1 Parser Combinators

Implement the following utilities of parser combinators:

Problem 1. (5 pts) `lookAhead` retrieves the first character in the input, but does NOT consume any of the input.

```
lookAhead :: Parser Char
```

```
> runParser lookAhead "abc"
[('a', "abc")]
```

Problem 2. (5 pts) `optional p` *tries* to apply the parser `p`. But `optional p` does not fail even if `p` fails, it succeeds with `Nothing` and no input consumed.

```
optional :: Parser a -> Parser (Maybe a)
```

```
> runParser (optional eof) ""
[(Just (), "")]
```

```
> runParser (optional eof) "abc"
> [(Nothing, "abc")]
```

Problem 3. (5 pts) `choose ps` tries to apply all the parsers in `ps` in the order of the parsers, and gathers the successful result of any one of them.

```
choose :: [Parser a] -> Parser a
```

```
> runParser (choose [char 'a', char 'b', char 'c']) "abc"
[('a', "bc")]
> runParser (choose [string "a", string "ab", string "abcd"]) "abc"
[("a", "bc"), ("ab", "c")]
```

Problem 4. (10 pts) `followedBy p q` matches the input led by `p` and followed by a successful parse of `q`, but it does NOT consume the input consumed by `q` even if `q` parses successfully. It fails when either `p` or `q` fails.

```
followedBy :: Parser a -> Parser b -> Parser a
```

```
> runParser (char 'a' `followedBy` char 'b') "abc"
[('a', "bc")]
> runParser (char 'a' `followedBy` char 'c') "abc"
[]
```

Problem 5. (10 pts) `manyUntil p q` applies parser `p` 0 or more times until `q` succeeds and gathers the result of `p` in a list. It consumes the input consumed by both `p` and `q`. `manyUntil` fails if parser `q` fails after applying parser `p` for some number of times. The parser should returns all possible results.

```
manyUntil :: Parser a -> Parser b -> Parser [a]
```

```
> runParser (manyUntil (char 'a') (char 'b')) "aaabc"
[(['a', 'a', 'a'], "c")]
> runParser (manyUntil (char 'a') (char 'b')) "bc"
[([], "c")]
> runParser (manyUntil (char 'a') (char 'b')) "aaa"
[]
> runParser (manyUntil (char 'a') (satisfy isAlpha)) "aabc"
[("","abc"),("a","bc"),("aa","c")]
```

2 Regular Expressions

2.1 Introduction

The Regular Expression is a minimal language that describes the patterns of strings. In this assignment we consider a subset of the regular expression contains only:

- Plain characters from `a` to `z` (both lowercase and uppercase) and digits from `0` to `9`, all of them match themselves.
- `.` (dot) character that matches any single character.
- Postfix operator `*` that matches zero or more patterns before it.
- Postfix operator `+` that matches one or more patterns before it.
- Postfix operator `?` that matches zero or one pattern before it.
- Infix operator `|` that match its left-hand-side pattern *or* its right-hand-side pattern.
- Parentheses `(`, `)` which groups pattern to dictate the associativity of operators explicitly.

The precedence of operators is shown as follow (top to down means precedence from higher to lower):

Operator	Explanations
`( )`	Parentheses have the highest precedence, naturally.
`*`, `+`, `?`	Higher than concatenation, e.g. `ab+` is equivalent to `a(b+)`.
Concatenation	Concatenation is done by directly put patterns together, e.g. concatenation of `a` and `b` is `ab`.
`|`	Lower than concatenation, e.g. `ab|cde` is equivalent to `(ab)|cde`.

2.1.1 Examples

Regular Expression	Matched Strings	Explanations
`abc`	"abc"	Exact match
`a.c`	"abc", "aac", "a1c", …	Dot matches any characters
`ab|c`	"ab", "c"	`|` operator separate `ab` and `c`
`a(b|c)`	"ab", "ac"	`a` concatenate to `b|c`
`ab*`	"a", "ab", "abb", …	`a` followed by any number of `b`

Regular Expression	Matched Strings	Explanations
`(ab)*`	""“,”ab“,”abab”, …	Any number of `ab`
`ab?c`	“abc”, “ac”	Optional `b` between `a` and `c`
`(ab)+`	“abc”, “abab”, …	A positive number of `ab`
`(a|bc+|d)*`	“aa”, “bccbcc”, “ddd”…	Equivalent to `(a|(b(c+))|d)*`

2.1.2 Corner Cases

- Empty pattern is NOT allowed, e.g, empty string, `()` (empty pattern inside `( )`), `a||c` (empty pattern around `|`)
- Consecutive postfix operators are NOT allowed, e.g. `1++`, `ab+*`. But they ARE allowed with parentheses, e.g. `(1+)+` or `a(b+)*`

2.2 Datatype `Regex` and Regular Expression Parser

Problem 6. Design a data type `Regex` whose values that represent “regular expressions”. Implement the `show` function that prints the regular expression *without any unnecessary parenthesis*.

```
data Regex = -- Your design here

instance Show Regex where
  show r = undefined -- Replace undefined with your implementation
```

Problem 7. Implement a parser for regular expression with the following signature.

```
regexP :: Parser Regex
```

Question 6 and 7 together worth 40 pts.

2.2.1 Constraints and Properties

The parser `regexP`, `Regex` and its `Show` instance together must have the following properties:

- `regexP` should fail with invalid string inputs, instead of partially succeed. In the following example, the parser should NOT yield a partially successful result by parsing the prefix `"a+"`, it should fail completely.

```
> runParser regexP "a+++"
[]
```

- Successful parsing results have to consume the entirety of the input string, partially successful results shouldn't be included.
- `show` prints the `Regex` value parsed by `regexP` with NO *unnecessary parenthesis*. In other words, it prints the "simplest parenthesized form" of the original regular expression.

```
> show $ fst $ head $ runParser regexP "((ab)(c))"
abc
```

"Unnecessary parentheses" for a regular expression are pairs of parentheses which, when eliminated, the regular expression is left valid syntactically, and matches same set of strings. For example all the parentheses in all of the following examples are considerred unnecessary:

```
(a|b)|c     # equivalent to "a|b|c"
a|(b|c)     # ditto
(ab)c       # equivalent to "abc"
a(bc)       # ditto
((a(b(c)))) # ditto
```

But the following parentheses are necessary:

```
(a|b)c # NOT equivalent to a|bc
(a+)*  # NOT equivalent to "a+*", which is invalid
```

You do NOT have to be bothered with scenarios like the following example that relies on more complicated properties about some combination of `*`, `+`, `?` and `|`:

```
(a*|a)*
```

Although it matches the same string as `a*|a*`, which is a valid regular expression, it relies on properties of the `*` operators. So the parentheses above are not considered unnecessary.

2.2.2 Flexibilities and Hints

- It's NOT necessary that `regexP` yields a unique output with all possible inputs. The results are acceptable as long as all the parsing results represent the same regular expression (The result of `show`ing them are equal).
- You certainly CAN define `Regex` like `data Regex = Regex String` that just contains the original input. But you will certainly make the other parts of your implementation harder by doing so.
- It's recommended that you define `Regex` recursively by following the syntax of regular expression, or even mutual-recursively, e.g.

```
data Regex   = ... | {- mentioning Postfix -} | ...
data Postfix = ... | {- mentioning Regex   -} | ...
```

2.3 Matching Strings with `Regex`

Problem 8. (25 pts) Implement function `matcher`.

`matcher regex` is a parser, whose parsing results are ALL of the "matches" of `regex` against input strings. The parser fails when there is no matches.

For illustration purposes, we use `regexOf` function in the following examples that somehow transforms a string to a `Regex` object.

```
matcher :: Regex -> Parser String

-- magical function for illustration only
regexOf :: String -> Regex
```

```
> runParser (matcher $ regexOf "b+") "abbc"
[("b", "bc"), ("bb", "c"), ("b", "c")]
> runParser (matcher $ regexOf ".*") "ab"
[("", "abc"), ("a", "b"), ("ab", ""), ("", "b"), ("b", ""), ("", "")]
```

A "match" of the regular expression contains a portion of the string that is matched by the pattern of the regular expression, and the rest of the input string after the matched portion.

For example, the `.*` above matches the empty string in `ab`, which appears in three different positions denoted by the `!`: `!ab`, `a!b`, `ab!`. And the matches of empty strings on these three positions are considered DIFFERENT matches.

2.3.1 Flexibilities and Hints

- The order of the matching results does NOT matter.
- The match does NOT only start from the beginning of the string, as shown in the examples above.
- The design of `Regex` dictates the difficulty of this problem.

3 Submission Caveats

3.1 Grading

- The grading of every question is done by comparing the outputs against certain test cases automatically by the grading script. Each test case contributes to some points. So there's NO "condolence score" for the labor of coding.
- Time complexity of implementation is NOT assumed. However, your programs should terminate with relatively small input in a "reasonable" amount of time.

3.2 Implementation

- You are encouraged to define your own auxiliary functions or data types to solve the problems, as long as you implement the functions and data types explicitly required by the questions.
- You are NOT allowed to add additional source files besides the three in the templates (`Regex.hs`, `Parser.hs`, and `Combinator.hs`).
- You are NOT allowed to add any additional `import` statement, the imports in the templates should be more than enough for you. However you are allowed to remove `import` statements in the template in case of version compatibility issues, but it's strongly discouraged so don't do that unless it's absolutely necessary.
- With the said restrictions, you are free to use any library functions and language extensions supported by ghc 8.10 or lower.
- Ideally, your ghc compiler should be version 8.6.1 or above to be able to use the language extension in the `Parser.hs`.

3.3 Submission

- Rename the directory containing template files to `A2_XXX` where `XXX` is replaced by your student number. Pack the directory into a *zip* file named `A2_XXX.zip` when submitting (other compression formats `rar`, `7z` etc are NOT accepted), i.e. the zip file `A2_XXX.zip` contains a directory `A2_XXX`, which contains *exactly* three template source files with their names unchanged.
- Please submit your assignment to moodle before deadline.