Final Project: Deep Outline

Thom Dickson, Alex Cozzie

Fall 2022

1 Main Function

```
main :: IO ()
main = do
  args@[search, replace, filename] <- getArgs
  let search_term = parse $ lexer search
  putStrLn $ "Replacing: " ++ show search_term
  let replace_term = parse $ lexer replace
  putStrLn $ "With: " ++ show replace_term
  content <- readFile filename
  let text = lines content
  let replaced = map (replaceLine search_term replace_term) text
  putStrLn $ unlines replaced
```

2 Key Functions

2.1 `lexer`

The `lexer` function will lex the regular expressions given as program arguments.

```
lexer :: String -> [Token]
lexer "" = []
lexer ('.' : s) = WildcardT : lexer s
lexer ('*' : s) = StarOp : lexer s
lexer ('+' : s) = PlusOp : lexer s
lexer ('\\' : s) = Backslash : lexer s
lexer ('(' : s) = LPar : lexer s
lexer (')' : s) = RPar : lexer s
lexer ('[' : s) = LBracket : lexer s
lexer (']' : s) = RBracket : lexer s
lexer (c : s) = LiteralT c : lexer s
```

2.2 `parse/sr`

The `parse` and `sr` functions will be used to take `Token` lists from the lexer and construct valid syntax trees for Regular Expressions.

```
sr :: [Token] -> [Token] -> [Token]
-- Handle escaped characters
sr (WildcardT : Backslash : s) q = sr (PE (Literal '.') : s) q
sr (StarOp : Backslash : s) q = sr (PE (Literal '*') : s) q
sr (PlusOp : Backslash : s) q = sr (PE (Literal '+') : s) q
sr (LPar : Backslash : s) q = sr (PE (Literal '(') : s) q
```

```
sr (RPar : Backslash : s) q = sr (PE (Literal ')') : s) q
sr (LBracket : Backslash : s) q = sr (PE (Literal '[') : s) q
sr (RBracket : Backslash : s) q = sr (PE (Literal ']') : s) q
-- Capture groups
sr (LiteralT d : Backslash : s) q | isDigit d = sr (PE (CaptureReplace (digitToInt d)) : s) q
-- Repeats
sr (StarOp : PE t : s) q = sr (PE (Star t) : s) q
sr (PlusOp : PE t : s) q = sr (PE (Concat t (Star t)) : s) q -- Use star for plus
-- Wildcard
sr (WildcardT : s) q = sr (PE Wildcard : s) q
-- Concat
sr (PE b : PE a : s) q = sr (PE (Concat a b) : s) q
-- Normal Literals
sr (LiteralT c : s) q = sr (PE (Literal c) : s) q
sr s (x : xs) = sr (x : s) xs -- Shift
sr s [] = s

parse :: [Token] -> RegEx
parse t = case sr [] t of
  [PE e] -> e
  s -> error $ "Failed to parse: " ++ show s
```

2.3 replaceLine

`replaceLine` will be the top level function that is called on every line in the input file. It will iterate through the line calling `match`, which will ultimately preform the replacement operation.

```
replaceLine :: RegEx -> RegEx -> String -> String
replaceLine e r s@(x : xs) = case match e r s of
  "" -> x : replaceLine e r xs -- No match. Move on
  x -> x
```

3 Auxiliary Functions

3.1 match

`match` will iterate through the given string and find the end of the match to the regular expression.

Once the end of the match is found, it will call the `replace` function to preform the replacement.

If the `match` function doesn't find a complete match, it will return `""`.

```
match :: RegEx -> RegEx -> String -> String
```

4 replace

Preform the actual replacement. This function can assume the string given is a complete match.

This function will also handle capture groups.

```
replace :: RegEx -> RegEx -> String -> String
```