Lab 003: Happy (Parser Generator)

Dr. Matthew Pike

16th October 2018

Please submit a .y document containing your completed Happy parser grammar via the associated Moodle assignment by 4PM today (day of lab). This work is not assessed, but does contribute towards your attendance mark for this lab (you must also sign the register).

Introduction

In today's laboratory you are going to implement a simple grammar which is capable of performing the basic functions we may expect from a calculator. We will name this parser **calc**.

We will use the Happy (https://www.haskell.org/happy/) Parser Generator for our parser. Happy is used in both CW1 and CW2 of this module, which means this lab will help you towards understanding and answering the coursework.

Getting Up and Running

If you are using Happy on **cslinux**, then you are all set - typing the command:

```
happy --version
```

In the command prompt, should return something like the following:

```
Happy Version 1.19.9 Copyright (c) 1993-1996 Andy Gill, Simon
Marlow (c) 1997-2005 Simon Marlow

Happy is a Yacc for Haskell, and comes with ABSOLUTELY NO
WARRANTY.
This program is free software; you can redistribute it and/or
modify
it under the terms given in the file 'LICENSE' distributed with
the Happy sources.
```

If you are completing the lab on your own personal device, follow the instructions for your platform as specified on the Happy webpage (https://www.haskell.org/happy/). Generally however, running the following should install Happy for most users:

`cabal install happy`

To compile your grammar parser for this lab, use the following commands (in order) in the command prompt:

```
happy lab03.y
ghc lab03.hs
```

To run the parser:

```
./lab03
```

Instructions

Step1

Create a new Happy parser specification file, named - **lab03.y**. Add the following contents to the file:

```
{
module Main (main) where

import Data.Char
}

%name calc
%tokentype { Token }
%error { parseError }

%token
      let             { TokenLet }
      in              { TokenIn }
      int             { TokenInt  }
      var             { TokenVar  }
      '='             { TokenEq }
      '+'             { TokenPlus }
      '-'             { TokenMinus }
      '*'             { TokenTimes }
      '/'             { TokenDiv }
      '('             { TokenOB }
      ')'             { TokenCB }

%%
```

The binding of symbols and their semantic meanings as defined in `%token` are incomplete. Add the necessary operators that bind semantic values to their appropriate operators.

Note - You do not need to parse any additional symbols (LHS) or specify any

additional tokens. Rather you need to consider the special grammar items whose semantic meaning is not defined (purely) by the token itself.

Hint - Lecture 5, Slide 15.

Step 2

Insert the following code into lab03.y, below the code you've written so far (in order).

```
Exp :: { [(String, Int)] -> Int }
      : let var '=' Exp in Exp  { \p -> $6 (($2, $4 p) : p) }
      | Exp1                    { $1 }

Exp1 :: { [(String, Int)] -> Int }
      : Exp1 '+' Term           { \p -> $1 p + $3 p }
      | TODO-1
      | Term                    { $1 }

Term :: { [(String, Int)] -> Int }
      : TODO-2
      | Term '/' Factor         { \p -> $1 p `div` $3 p }
      | Factor                  { $1 }

Factor :: { [(String, Int)] -> Int }
      : int                     { \p -> $1 }
      | var                     { \p -> case lookup $1 p of
                                          Nothing -> error "no
var"
                                          Just i  -> i }
      | '(' Exp ')'             { $2 }
```

Replace **TODO-1** with the grammar rules necessary to support subtraction using the standard subtraction symbol: **-** e.g. 2 - 1.

Replace **TODO-2** with the grammar rules necessary to support multiplication using the standard multiplication symbol: * e.g. 2 * 2.

Step 3

Insert the following code into lab03.y, below the code you've written so far (in order).

```
{
parseError :: [Token] -> a
parseError _ = error "Parse error"

data Token
      = TODO
  deriving Show
```

Using the Tokens identified in Step 1, complete the definition of the data type `Token` (**TODO** section in the code).

Step 4

Insert the following code into lab03.y, below the code you've written so far (in order).

```
lexer :: String -> [Token]
lexer [] = []
lexer (c:cs)
      | isSpace c = lexer cs
      | isAlpha c = lexVar (c:cs)
      | isDigit c = lexNum (c:cs)
lexer ('=':cs) = TokenEq : lexer cs
TODO

lexNum cs = TokenInt (read num) : lexer rest
      where (num,rest) = span isDigit cs

lexVar cs =
   case span isAlpha cs of
      ("let", rest) -> TokenLet : lexer rest
      ("in", rest)  -> TokenIn : lexer rest
      (var, rest)   -> TokenVar var : lexer rest

main :: IO ()
main = do
    cs <- getContents
    print $ calc (lexer cs) []
}
```

Add the necessary lexer rules to the grammar definition (**TODO** section in the code).