Monadic LL Parser

Overview

The motivation for this project was to get a better understanding of how Monads work and to implement a monadic parser.

The intention of the project was to implement a parser inspired by the following paper

- https://www.cs.nott.ac.uk/~pszgmh/pearl.pdf

Where combinators and parsing components could be built and then used to build more expressive LL grammars as needed.

The goal of the project was to support the following grammar

```
expr ::= expr addop term | term | letop
term ::= term mulop factor | factor
factor ::= number | ( expr )
letop ::= letsym var equalsym (integer
addop ::= + | -
mulop ::= * | /
appsep := ;
letsym ::= 'let'
equalsym ::= =
var ::= [a-zA-Z]+
number ::= <integer> | var -> <integer>
```

The grammar defined in the above mentioned paper implements one that is consistent with a calculator. This project adds an enhancement that allows variable creation and substitution in our expression.

Implementation

I think one of the more important aspects of this application are the monadic parser and the combinators that can be used to build up more complicated parsers.

The monadic parser was interesting because it gave me a better feel for how to implement a monad and how it could be used to bind different parsers together.

I believe a nice example of a combinator is the below shown token parser that uses the space parser to produce a parser that can ignore spaces.

```
-- Parser to read zero or more spaces.
space :: Parser ()
space = many (sat isSpace) >> return ()

-- Parser that decorates other parsers to handle
-- beginning and ending spaces
token :: Parser a -> Parser a
token p = space >> p >>= (\a -> space >> return a)
```

An enhancement that was made during this project was to add the ability to declare and then use a variable in the calculator syntax. This was achieved by adding the following capability to the grammar:

- let a = 10 ; a + 1

The above would yield a result of **11**.

This was achieved by adding the following function:

```
letop env = symbol "let" >>
 var >>= (\r1 ->
 (symbol "=") >>= (\r2 ->
 int env >>= (\r3 ->
 (symbol ";") >>= (\r4 ->
   let newEnv = H.insert r1 r3 env
   in expr newEnv))))
```

Enhancing the definition of the int parser as show below:

```
-- building blocks for integers
-- Parser deals with positive integers
-- some reads 1 or more digits
-- read function makes 'd' in int.
posint :: Parser Int
posint =
 do d <- some (sat isDigit)
    return (read d)

-- building blocks for integers
-- Parser deals with negative integers
negint :: Parser Int
negint = char '-' >> posint >>= (\d -> return (-d))

-- building blocks for integers
-- Parser deals with environment variable
-- substitution for variables used that have
```

```haskell
-- integer values in the environment.
intvar :: H.HashMap String Int -> Parser Int
intvar env =
 var >>= (\r ->
     let val = H.lookup r env
     in case val of
       Just a -> (return a)
       otherwise ->  Parser(\s -> []))


int :: Env -> Parser Int
int env = posint <|> negint <|> intvar env
```

And plumbing the environment appropriately, as shown in the code snippets above.

Examples of CLI Execution patterns

There are a number of ways to exercise the code for this project. Individual parsers can be executed and, additionally, there is a remedial interpreter that has been built for you to try and type in expressions at a prompt.

The following are examples of some cli commands that can be used to test the system manually.

Example of running the **int** parser.
At the root of the project run the following cli commands

1. stack ghci src/Parser.hs
2. import Data.HashMap.Strict as H
3. parse (int H.empty) "123" ->
    a. This will produce the following result: `[(123,"")]`

Below are screen grabs of some example parsing runs:

Example of a successful int parser execution:

```
ghci> import Data.HashMap.Strict as H
ghci> parse (int H.empty) "123"
[(123,"")]
ghci>
```

Example of an unsuccessful int parser execution:

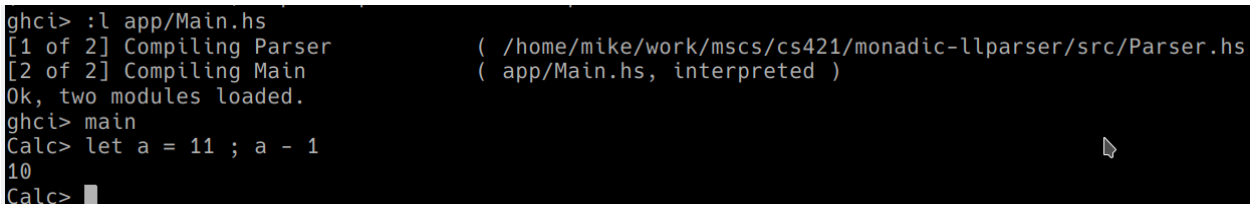
```
ghci> parse (int H.empty) "a123"
[]
ghci>
```

Example of running the interpreter
At the root of the project run the following cli commands

1. stack ghci app/Main.hs
2. ghci>main
3. Calc>let a = 11 ; a - 1
4. **10** – which is the result of the above expression
5. Calc> buy – ends the interpreter session

Below is some screen grabs of this

Example of a successful let interpretation



Example of invalid syntax for a let expression

Future Implementation Considerations

I would like to investigate using the State Monad Transformer and replace the environment plumbing. Additionally I would like to enhance the remedial interpreter that was created for this project and have a clear separation between the parser and interpreter via the implementation of an AST of sorts.

Build and Tests

The **stack** build environment is used for this project.

The following are the cli commands that can be used:

- stack build - used to build the project
- stack test - used to run the tests associated with the project

The tests are implemented using the HUnit framework. There are **30 tests** that will run when 'stack test' is executed. These tests are designed to test all of the different parsers that were built for this project. Please refer to the test/Spec.hs file for a full review of the tests that have been implemented for this project.