Assignment 6

CS 4400 Programming Languages

Start early and come to us with questions.

Due: 11pm on Saturday, October 24, 2020

Submission:

1. Submit the following files via https://handins.ccs.neu.edu/courses/119:

   - `Assignment06.hs`
   - `Eval.hs`
   - `Syntax.hs`
   - `Repl.hs`

2. This assignment is meant to be worked on and submitted in pairs, but you can choose to work on your own. Note, that you need to have a team on Handins to be able to submit (a singleton team or a pair).

3. At the very top, the file should contain a preamble following this template.

```
{- |
Module      :  Assignment06
Description :  Assignment 6 submission for CS 4400.
Copyright   :  (c) <your name>

Maintainer  :  <your email>
-}

module Assignment06 where
```

   The file should contain your `main` function which starts the REPL.

4. Every top-level definition must include a purpose statement (for functions) and a type signature, followed by one or more defining equations.

5. Double-check that you have named everything as required and that functions required by this assignment have the correct type signatures.

6. Make sure your file loads into GHCi or can be compiled by GHC without any errors. **Your grade might be reduced by up to 50% if your code does not compile and run.**

Purpose: To practice interactive programming in Haskell while implementing a top-level (REPL, or shell) for a programming language.

State of the Union

At this point, your code-base for `protoScheme` should cover the following features:

1. Let-bindings and a variable references expressions
2. Arithmetic expressions with integer and floating point values
3. Boolean expressions, comparisons and equality checks, if expressions, conditionals
4. Pair values and selectors
5. Type predicates for integers, reals, numbers, pairs, and booleans
6. Global function definitions (with one or more arguments), function calls, global variable definitions

In this assignments, we will not be extending our language with additional features. Instead we will create an interactive REPL (**r**ead-**e**val-**p**rint **l**oop) environment. Let's start with an overview of how tow write interactive programs in Haskell.

Interactive Programming in Haskell. A Brief Tutorial

This is a very brief tutorial on interactive programming in Haskell. It should provide you with the necessary tools and techniques to help you build a REPL.

As we have discussed before, any interactive Haskell program will need to be written in the IO monad, that is, have an `IO a` type, where `a` can be any other type. You can think of `IO`, roughly, as the following type:

```haskell
type IO a = WorldState -> (a, WorldState)
```

where `WorldState` is some type representing the state of the world (it's not an actual type). That is, you can think of any IO *action* as a function that takes the current world state in and produces an updated state with an additional value. For example, a `main` function in a Haskell program has the type `IO ()`, meaning it is an IO action which doesn't return any meaningful value (the type of empty tuples, `()`, has only one value, also written `()`)

We usually program in the IO monad using do-notation, which conveniently expresses the sequencing of actions an provides syntax for reading values from IO actions. A basic example is a program that just prints something to the console:

```haskell
main :: IO ()
main = do
    putStrLn "Hello, world!"
```

The type of `putStrLn` is `String -> IO ()`. The next step is to read something from the console. The action for this is `getLine :: IO String`. To retrieve a value from a (monadic) action, we use the `<-` construct:

```haskell
main :: IO ()
main = do
    putStr "Type your name: "
    name <- getLine
    putStrLn ("Hello, " ++ name)
```

Since `getLine` has the type `IO String` and `<-` allows us to retrieve a value from an IO action, the type of `name` is `String`. So to speak, it allows us to "peel" the `IO` off the `String`.

Note, that IO is a monad and we talked about monads and the `>>=` (bind) operator (with type `(>>=) :: Monad m => m a -> (a -> m b) -> m b`) when discussing a convenient notation for expressions involving `Maybe`. Without using do-notation, the above main action can be also expressed as follows:

```haskell
main =
    putStr "Type your name: " >>= \_ ->
    getLine >>= \name ->
    putStrln ("Hello, " ++ name)
```

For IO, the type of `>>=` is `IO a -> (a -> IO b) -> IO b`. Note, how we ignore the result of the first `putStrLn` action.

Another interesting IO action is reading the contents of a file:

```haskell
readFile :: String -> IO String
```

which takes a filename and returns its contents as a string.

```haskell
main = do
    putStr "Type a filename: "
    filename <- getLine
    putStrLn $ "--- BEGIN " ++ filename
    contents <- readFile filename
    putStrLn contents
    putStrLn $ "--- END " ++ filename
```

Haskell doesn't have any looping constructs, like **`while`** or **`for`**. So, just like with pure functions, we need to use recursion if we want to repeat an IO action. Here's a little game:

```
guessNumber :: Integer -> IO ()
guessNumber number = do
    putStr "Guess the number I'm thinking of: "
    guess <- getLine
    if read guess == number
       then putStrLn "Congratulations! You won!"
       else do
           putStrLn "Wrong. Try again, please."
           guessNumber number
```

To return some meaningful value from an IO function, we use the `return` operation. Here, a word of caution is in order: `return` is just an ordinary function that wraps a given value in the current monad (type `Monad m => a -> m a`, or for IO: `a -> IO a`). Most importantly, it *does not terminate* the execution of the current function. Just try to run the following action in GHCi and see what is returned.

```
manyHappyReturns :: IO String
manyHappyReturns = do
    return "Happy"
    return "Happy"
    return "Happy"
    return "Happy"
    return "Well, this is not at all what I expected!"
```

For the REPL, you will need to write actions that communicate with the user while keeping track of a *context*. Thus, here is a final example of a program, which keeps reading integers and computes their sum or their product, depending on the command given. It also allows printing the current numbers (in reverse order) and it quits when the user types "quit" as the command.

```
calcLoop :: [Integer] -> IO [Integer]
calcLoop numbers = do
    line <- getLine
    processLine line
  where
    processLine "quit" = do
        putStrLn "Bye bye."
        return []
    processLine "sum" = do
        putStrLn $ "Result: " ++ show (sum numbers)
        calcLoop []
```

```
    processLine "product" = do
        putStrLn $ "Result: " ++ show (product numbers)
        calcLoop []
    processLine "print" = do
        putStrLn $ "This is what I have so far: " ++ show numbers
        calcLoop numbers
    processLine other = do
        calcLoop $ read other : numbers

main :: IO ()
main = do
    putStrLn "Welcome to the mostly useless calculator program."
    calcLoop []
    return ()
```

This should be enough to get you started with interactive console programming in Haskell.

Assignment Pack

The starter code pack for this assignment contains a parser for s-expressions, an updated SExpression module and a new version of SimpleTests with a few improvements:

`Parser.hs` Contains an s-expression parser. Feel free to look how the machinery is implemented, but you are not expected to know how it actually works. Your interface is the function `parseSExpression :: String -> Maybe S.Expr`.

`SExpression.hs` As before or minor adjustments. The `toString` function might come in handy for this assignment.

`Maps.hs` The `Maps` module has been updated to provide the function `keys`, which returns all the keys bound in the given map as a list. You will need this function. If you are using a different map implementation, you should find a similar function in the documentation.

`SimpleTests.hs` & `SimpleTestsColor.hs` Our little testing library has been updated with color! Colors are implemented using ANSI terminal sequences. This should work on Linux and macOS terminals, and most likely in a WSL (Windows Subsystem for Linux) terminal. However, since I don't want to make assumptions about how portable this is, a plain version is also provided. They have the same interface, so switching to color is as simple as changing

```
import SimpleTests (test, beginTests, endTests, testSection)
```

to

```
import SimpleTestsColor (test, beginTests, endTests, testSection)
```

Parsing S-expression from Strings

To parse s-expressions from strings, import the `Parser` module:

```
import Parser (parseSExpression)
```

Using `parseSExpression` should be relatively straightforward. Given a string, it tries to parse it into an `S.Expr`. If it cannot parse it, it returns `Nothing`

```
> parseSExpression "(a b 1 2 3.0 #f)"
Just (List [ Symbol "a"
           , Symbol "b"
           , Integer 1
           , Integer 2
           , Real 3.0
           , Boolean False
           ])

> parseSExpression "(defun f (x) (let (y 10) (+ x y)))"
Just (List [ Symbol "defun"
           , Symbol "f"
           , List [ Symbol "x"]
           , List [ Symbol "let"
                  , List [ Symbol "y"
                         , Integer 10
                         ]
                  , List [ Symbol "+"
                         , Symbol "x"
                         , Symbol "y"
                         ]]])

> parseSExpression "(f 1 2"
Nothing
```

Note, that the parser has some limitations. In particular, it does not parse otherwise valid s-expressions without spaces between list elements:

```
> parseSExpression "((a) b (c))"
Just (List [ List [ Symbol "a" ], Symbol "b", List [ Symbol "c" ] ])
```

```
> parseSExpression "((a)b(c))"
Nothing
```

Other than that, if you find it fails on some valid s-expression example, please let me know as soon as possible and include the failing example.

Questions

The only task for this assignment is to implement an interactive REPL (or shell) for `protoScheme`. The user is expected to enter global definitions and expressions on the console. If a definition is entered, the corresponding binding should be added to the current environment and the name should be available in subsequent definitions or expressions. If an expression is entered, it should be evaluated in the current global environment and the value should be printed to the console. Here is an example interaction:

```
protoScheme Version 1e-10
> 1
1
> (real? 12.43)
#t
> (define num 10)
Variable num defined.
> (define num 11)
Error: variable num is already defined.
> (defun add-num (x) (+ x num))
Function add-num defined.
> (add-num 12)
22
> (add-num num)
20
> (defun g (x) (let (y (add-num x)) (pair y (> y num))))
Function g defined.
> (g 10)
(20 . #t)
> (g -20)
(-10 . #f)
> (pair? (g 123))
#t
> (right (g 11))
#t
> (defun g (x) 10)
```

```
Error: function g is already defined.
> (+ 10 #t)
Evaluation error. Try again.
> y
Evaluation error. Try again.
> (+ 10
Parse error. Try again.
> :quit
Bye bye.
```

There are different kinds of errors which should result in the user receiving feedback, but shouldn't bring the REPL down:

1. Attempting to redefine a previously defined name.
2. A parse error – either resulting from an invalid s-expression entered or a valid s-expression which does not represent a valid `protoScheme` expression.
3. An evaluation error, resulting from, e.g., an unbound variable reference, an arithmetic error, etc.

Some of the errors are demonstrated above. The messages do not need to match the above interaction exactly.

Typing `:quit` instead of a global definition or expression should make the REPL quit gracefully.

You do not need to write new tests for the functions you implement for this assignment. You still might want to write a few tests to increase your confidence in your code. Your REPL-related code should go into `Repl.hs` and the `main` function in `Assignment06.hs` should start the REPL.

If you get stuck or need hints, let us know.