Parser of λ-programs
Homework Assignment 4

Rostislav Horčík

April 29, 2022

1 Introduction

This homework assignment aims to practice programming with monads. You implemented a normal-order evaluator for lambda expressions in the previous homework assignment. To make a complete interpreter of lambda calculus, we have to provide further a parser that reads an input program and translates it into the corresponding λ-expression that your evaluator can evaluate. Your solution should follow Lecture 12, where I show how to create a type constructor **`Parser`** and make it into an instance of **`Monad`**. Create a module `Parser.hs` containing the code in Section 4 that you import into your solution.

The parser should be implemented as a Haskell module called `Hw4`. Note the capital `H`. The module names in Haskell have to start with capital letters. As the file containing the module code must be of the same name as the module name, **all your code is required to be in a single file called `Hw4.hs`.** Your file `Hw4.hs` has to start with the following lines:

```haskell
module Hw4 where

import Control.Applicative
import Data.Char
import Parser
import Hw3
```

There are four imports. The first two necessary libraries so that the definitions from Lecture 12 work. The third is the module `Parser` whose code is in Section 4. The last import is your previous homework assignment. So be sure that you have your `Hw3.hs` and `Parser.hs` in the same folder as your `Hw4.hs`. You can use `eval :: Expr -> Expr` from your `Hw3` module in your tests. However, `eval` is not needed for the homework evaluation. The only necessary definition from `Hw3.hs` is the definition of the following data types:

```haskell
type Symbol = String
data Expr = Var Symbol | App Expr Expr | Lambda Symbol Expr deriving Eq
```

2 Parser specification

First, we have to specify the structure of λ-programs. λ-calculus does not have a mechanism allowing to give a name to an λ-expression that can be used to build up more complex λ-expressions in a human-readable way. Thus we extend λ-programs by such a mechanism.

The grammar specifying λ-programs is in Figure 1. The terminal symbols are alphanumeric characters and any whitespace like the space `" "` or the newline character `"\n"`. Variables `<var>` are non-empty sequences of alphanumeric characters. Separators `<sep>` are non-empty sequences of whitespaces. The program `<program>` has two parts. The first part consists of a (possibly empty) list of definitions. The second one is the main λ-expression. Definitions and the main

```
<program> -> (<definition> <sep>)* <expr> <space>*

<definition> -> <var> <sep> ":=" <sep> <expr>

<expr> -> <var>
        | "(\\" <var> '.' <expr> ')'
        | '(' <expr> <sep> <expr> ')'

<var> -> <alphanum>+
<alphanum> -> any alphanumeric character recognized by isAlphaNum

<sep> -> <space>+
<space> -> any whitespace recognized by isSpace, e.g. ' ', '\n'
```

Figure 1: The grammar for λ-programs.

λ-expressions have to be separated by `<sep>`, i.e., any non-empty sequence of whitespaces. After the main λ-expression, any (possibly empty) sequence of whitespaces is allowed.

Each definition `<definition>` consists of a variable followed by a separator `<sep>`, then the string `":="` followed by a separator `<sep>` and finally a λ-expression `<expr>`. Each λ-expression `<expr>` is either a variable `<var>` or a λ-abstraction or an application. The λ-abstraction starts with an opening parenthesis, followed by a single backslash character (note that in Haskell, we have to write `"\\"` to create a string containing a single backslash character), then a variable followed by the dot character, and finally, a λ-expression followed by a closing parenthesis. The application consists of two λ-expressions separated by a separator `<sep>` and enclosed in parentheses.

The definitions should behave like `let*` in Scheme, i.e., later definitions can use already defined names from the previous definitions. So the first definition introduces a name for a λ-expression not using any defined names. The second definition can use in its λ-expression the name from the first definition etc. The main λ-expression may use any defined name.

Your task is to implement in Haskell the function `readPrg :: String -> Maybe Expr` which takes a string containing a λ-program and returns either `Nothing` if the program is syntactically incorrect or `Just` a λ-expression. The returned λ-expression is the main λ-expression specified at the end of λ-program. It must have all the defined names resolved. So you first have to resolve the definitions so that their expressions do not use any defined names, and then you can resolve the defined names in the main λ-expression.

The substitutions in the resolution of definitions do not have to deal with any name conflicts like the substitutions in β-reductions. Consequently, you can simply replace a variable by the corresponding λ-expression without checking whether a free variable would become bound.

3 Test cases

Let us go through several examples of λ-programs and the corresponding outputs of `readPrg`.

If there are no definition, `readPrg` just parses the input λ-expression.

```
> readPrg "(\\s.(\\z.z))"
Just (\s.(\z.z))
```

The above output assumes that the `Show` instance for `Expr` is defined in your `Hw3.hs`. If we used the automatically derived `Show` instance, it would display:

```
Just (Lambda "s" (Lambda "z" (Var "z")))
```

In the further examples, I will assume that the `Show` instance is defined (not the automatic one). If the input program is grammatically incorrect, `readPrg` returns `Nothing`.

```
> readPrg "(\\s.\\z.))"
Nothing
```

A more complex λ-expressions can be defined via definitions. E.g. we define the identity function as `I` and then we define its application to itself.

```
> readPrg "I := (\\x.x)\n (I I)"
Just ((\x.x) (\x.x))
```

For larger λ-programs, it is better to write them in a text editor and then pass the whole file as input. Create a new Haskell source file `execHw4.hs` in the same folder as `Hw3.hs` and `Hw4.hs` containing the following code:

```haskell
import Hw3
import Hw4

main :: IO ()
main = do inp <- getContents
          case readPrg inp of
            Nothing -> putStrLn "Incorrect program"
            Just e -> print $ e
```

Then you can pass a file with a λ-program `filename.lmb` by the following command in the Bash prompt:

```
$ runghc execHw4.hs < filename.lmb
```

The above haskell program just reads everything from the standard input by `getContents`. Then it calls `readPrg` and if it is successful, it displays the parsed λ-expression. If you wish, you can easily turn this program into a λ-calculus interpreter. Just call your `eval` function on the λ-expression `e` before printing it out.

Let us return to the examples. Consider following λ-program written in a text editor:

```
0 := (\s.(\z.z))
S := (\w.(\y.(\x.(y ((w y) x)))))
1 := (S 0)
2 := (S 1)

((2 S) 1)
```

Note that now there are only single backslash characters. As we pass λ-programs from the standard input, we do not have to escape them.

The above λ-program captures the computation $2 + 1$. There are four definitions. The first two definitions do not use any previously defined names. The third one is resolved by substituting `(\s.(\z.z))` for `0` and `(\w.(\y.(\x.(y ((w y) x)))))` for `S`. So we obtained

```
((\w.(\y.(\x.(y ((w y) x))))) (\s.(\z.z)))
```

Then the definition of `2` is resolved by substituting the above λ-expression for `1` and the λ-expression `(\w.(\y.(\x.(y ((w y) x)))))` for `S`. This leads to

```
((\w.(\y.(\x.(y ((w y) x))))) ((\w.(\y.(\x.(y ((w y) x))))) (\s.(\z.z))))
```

Finally, the λ-expression `((2 S) 1)` is resolved by substituting the corresponding λ-expressions for `2, S, 1` respectively. The result looks like

```
((((\w.(\y.(\x.(y ((w y) x))))) ((\w.(\y.(\x.(y ((w y) x))))) (\s.(\z.z))))
(\w.(\y.(\x.(y ((w y) x)))))) ((\w.(\y.(\x.(y ((w y) x))))) (\s.(\z.z))))
```

You see that it can quickly get unreadable. Thus I advise you to evaluate these expressions in your tests by your `eval` function transforming it into the normal form. If you change the above Haskell program as follows:

```
import Hw3
import Hw4

main :: IO ()
main = do inp <- getContents
          case readPrg inp of
            Nothing -> putStrLn "Incorrect program"
            Just e -> print $ eval e
```

then you can test the λ-program `((2 S) 1)` by calling

```
$ runghc execHw4.hs < 2S1.lmb
(\y.(\x.(y (y (y x)))))
```

where `2S1.lmb` is the filename with the λ-program.

4 Parser module

The code of the parser module `Parser.hs`. It contains all the necessary definitions making the type constructor **`Parser`** a monad and alternative. Moreover, it includes basic parsers for specific characters, strings, alphanumeric characters, and spaces.

```
module Parser where

import Control.Applicative
import Data.Char

newtype Parser a = P { parse :: String -> Maybe (a, String) }

instance Functor Parser where
    -- fmap :: (a -> b) -> Parser a -> Parser b
    fmap f p = P (\inp -> case parse p inp of
                            Nothing -> Nothing
                            Just (v,out) -> Just (f v, out))

instance Applicative Parser where
    -- (<*>) :: Parser (a -> b) -> Parser a -> Parser b
    pg <*> px = P (\inp -> case parse pg inp of
                             Nothing -> Nothing
                             Just (g,out) -> parse (fmap g px) out)
    pure v = P (\inp -> Just (v,inp))

instance Monad Parser where
    -- (>>=) :: Parser a -> (a -> Parser b) -> Parser b
    p >>= f = P (\inp -> case parse p inp of
                           Nothing -> Nothing
                           Just (v,out) -> parse (f v) out)

instance Alternative Parser where
    -- empty :: Parser a
    empty = P (\_ -> Nothing)
    -- (<|>) :: Parser a -> Parser a -> Parser a
    p <|> q = P (\inp -> case parse p inp of
                           Nothing -> parse q inp
                           Just (v,out) -> Just (v,out))

-- Parsers
item :: Parser Char
item = P (\inp -> case inp of
                    "" -> Nothing
                    (x:xs) -> Just (x,xs))

sat :: (Char -> Bool) -> Parser Char
sat pr = item >>= \x -> if pr x then return x
                        else empty

alphaNum :: Parser Char
alphaNum = sat isAlphaNum

char :: Char -> Parser Char
char c = sat (== c)

string :: String -> Parser String
string "" = pure ""
string (x:xs) = (:) <$> char x <*> string xs

sep :: Parser ()
sep = some (sat isSpace) *> pure ()
```