Arithmetic Lang

Subhajit Sahu (2018801013)

September 14, 2019

Contents

1 Imports

I needed to use `isPrefixOf` for doing string replacement. It is part of module `Data.List`. In the `main` function, using `putStr` does not flush the string to `stdout` immediately. For that, module `System.IO` nneded to be imported.

```
import Data.List
import System.IO
```

2 Values

Unlike the arithmetic language interpreter (using racket), discussed in class, this arithmetic lang needs to support both numbers and booleans. To support that, i used a boxing type, here called `Value`. To keep the code simple, i made it an instance of `Eq`, `Show`, `Num`, and `Fractional`. Nummbers are called `Numv`, and booleans `Boolv`, to distinguish them from builtin typeclass / type.

```
data Value =
  Numv  Float |
  Boolv Bool
  deriving (Eq)

instance Show Value where
  show (Numv x)  = show x
  show (Boolv x) = show x
```

```
instance Num Value where
  (Numv x) + (Numv y) = Numv $ x + y
  (Numv x) * (Numv y) = Numv $ x * y
  abs (Numv x) = Numv $ abs x
  signum (Numv x) = Numv $ signum x
  fromInteger x = Numv $ fromInteger x
  negate (Numv x) = Numv $ negate x

instance Fractional Value where
  (Numv x) / (Numv y) = Numv $ x / y
  fromRational x = Numv $ fromRational x
```

3 Abstract Syntax Tree

The AST is as expected, and again numbers and booleans are called differently.

```
data Ast =
  Numa   Float   |
  Boola  Bool    |
  Add    Ast Ast |
  Mul    Ast Ast |
  Sub    Ast Ast |
  Div    Ast Ast |
  Equals Ast Ast |
  IsZero Ast
  deriving (Eq, Read, Show)
```

4 Run

There is a `main` function which provides the REPL. It simply accepts a line and shows the output `Value` of `run` function. Use an empty (null) line to terminate. After displaying the "arithmetic: " prompt, it was necessary to flush to `stdout`.

```
main = do
  putStr "arithmetic: "
  hFlush stdout
  exp <- getLine
  if null exp
    then return ()
    else do
      putStrLn (show . run $ exp)
      main
```

The run function is simply parses and evaluates a string.

```
run :: String -> Value
run = eval . parse
```

5 Evaluator

The `eval` function pattern matches with all constructors of AST, and does the desired action. It finally returns a `Value`. Since `Value` is already an instance of `Eq`, `Num`, `Fractional` we can directly use arithmetic operators. However, equals operator is an exception, since it always returns a `Bool` and thus it had to be boxed in a `Boolv`.

```
eval :: Ast -> Value
eval (Numa  x) = Numv  x
eval (Boola x) = Boolv x
eval (Add x y) = (eval x) + (eval y)
eval (Mul x y) = (eval x) * (eval y)
eval (Sub x y) = (eval x) - (eval y)
eval (Div x y) = (eval x) / (eval y)
eval (Equals x y) = Boolv $ (eval x) == (eval y)
eval (IsZero x)   = Boolv $ (eval x) == Numv 0
```

6 Parser

I wanted to depend upon the `read` function to generate the AST. But a simple `read` on the string would not work since it would read numbers as `Num` and not `Numa` (AST needs that). Same is true for the boolean values. It would also not work with operators.

However if the input string is transformed into a suitable format, using our AST constructors, read should work fine. So, as you can see we split the input into words, transform it, and rejoin it back, before feeding it back to read, which then directly gives us the AST.

Also, since brackets and operators, or brackets and numbers are not separated by a space, it causes problems for word splitting, so it needed to be done beforehand.

```
parse :: String -> Ast
parse s = (read . unwords . map token . words $ bpad) :: Ast
  where bpad = replace "(" " ( " . replace ")" " ) " $ s
```

Here is the token replacement strategy.

```
token :: String -> String
token "+" = "Add"
token "*" = "Mul"
token "-" = "Sub"
token "/" = "Div"
token "=" = "Equals"
token "zero?" = "IsZero"
token t
  | isFloat t  = "(Numa "  ++ t ++ ")"
  | isBool  t  = "(Boola " ++ t ++ ")"
  | otherwise  = t
```

And, here are a few utility functions we are using.

```
replace :: (Eq a) => [a] -> [a] -> [a] -> [a]
replace _ _ [] = []
replace from to all@(x:xs)
  | from ‘isPrefixOf‘ all = to ++ (replace from to . drop (length from) $ all)
  | otherwise             = x : replace from to xs

isFloat :: String -> Bool
isFloat s = case (reads s) :: [(Float, String)] of
  [(_, "")] -> True
  _         -> False

isBool :: String -> Bool
isBool s = case (reads s) :: [(Bool, String)] of
  [(_, "")] -> True
  _         -> False
```

7 This is where you put it all together

```
import Data.List
import System.IO


data Value =
  Numv  Float |
  Boolv Bool
  deriving (Eq)

instance Show Value where
  show (Numv x)  = show x
  show (Boolv x) = show x

instance Num Value where
  (Numv x) + (Numv y) = Numv $ x + y
  (Numv x) * (Numv y) = Numv $ x * y
  abs (Numv x) = Numv $ abs x
  signum (Numv x) = Numv $ signum x
  fromInteger x = Numv $ fromInteger x
  negate (Numv x) = Numv $ negate x

instance Fractional Value where
  (Numv x) / (Numv y) = Numv $ x / y
  fromRational x = Numv $ fromRational x


data Ast =
  Numa   Float   |
  Boola  Bool    |
  Add    Ast Ast |
  Mul    Ast Ast |
  Sub    Ast Ast |
  Div    Ast Ast |
  Equals Ast Ast |
  IsZero Ast
  deriving (Eq, Read, Show)
```

```
main = do
  putStr "arithmetic: "
  hFlush stdout
  exp <- getLine
  if null exp
    then return ()
    else do
      putStrLn (show . run $ exp)
      main

run :: String -> Value
run = eval . parse

eval :: Ast -> Value
eval (Numa  x) = Numv  x
eval (Boola x) = Boolv x
eval (Add x y) = (eval x) + (eval y)
eval (Mul x y) = (eval x) * (eval y)
eval (Sub x y) = (eval x) - (eval y)
eval (Div x y) = (eval x) / (eval y)
eval (Equals x y) = Boolv $ (eval x) == (eval y)
eval (IsZero x)   = Boolv $ (eval x) == Numv 0

parse :: String -> Ast
parse s = (read . unwords . map token . words $ bpad) :: Ast
  where bpad = replace "(" " ( " . replace ")" " ) " $ s

token :: String -> String
token "+" = "Add"
token "*" = "Mul"
token "-" = "Sub"
token "/" = "Div"
token "=" = "Equals"
token "zero?" = "IsZero"
token t
  | isFloat t  = "(Numa "  ++ t ++ ")"
  | isBool  t  = "(Boola " ++ t ++ ")"
  | otherwise  = t
```

```
replace :: (Eq a) => [a] -> [a] -> [a] -> [a]
replace _ _ [] = []
replace from to all@(x:xs)
  | from ‘isPrefixOf‘ all = to ++ (replace from to . drop (length from) $ all)
  | otherwise             = x : replace from to xs

isFloat :: String -> Bool
isFloat s = case (reads s) :: [(Float, String)] of
  [(_, "")] -> True
  _         -> False

isBool :: String -> Bool
isBool s = case (reads s) :: [(Bool, String)] of
  [(_, "")] -> True
  _         -> False
```