Lexical Lang

Subhajit Sahu (2018801013)

September 30, 2019

Contents

1 Imports

In lexical language, there are identifiers. For identifying them i used `isAlpha` and `isAlphaNum`, which are in `Data.Char` package. For implementing `assume`, haskell package `Data.Map` is available.

```
import Data.List
import System.IO
import Data.Char
import qualified Data.Map as Map
```

2 Values

Expressible values are the same as before.

```
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 now additionally includes identifiers, called `Ida`; logical operators called `And`, `Or`, `Not`; conditional expression `If`; and the identifier binding keyword `Assume`.

```
data Ast =
  Numa   Float   |
  Boola  Bool    |
  Ida    String  |
  Add    Ast Ast |
  Mul    Ast Ast |
  Sub    Ast Ast |
  Div    Ast Ast |
  Equals Ast Ast |
```

```
  And    Ast Ast |
  Or     Ast Ast |
  Not    Ast     |
  IsZero Ast     |
  If Ast Ast Ast |
  Assume [(Ast, Ast)] Ast
  deriving (Eq, Read, Show)
```

4 Environment

The environment called `Env` is simply a `String` to `Value` map.

```
type Env = Map.Map String Value
```

5 Run

The `main` function as before provides the REPL. It simply accepts a line and shows the output `Value` of `run` function. Use an empty (null) line to terminate.

```
main = do
  putStr "lexical: "
  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 with an empty environment (an empty map).

```
run :: String -> Value
run = (eval $ Map.empty) . parse
```

6 Evaluator

The `eval` function, as before, pattern matches with all constructors of AST. Here `Ida` is simply a fetch from current environment. `And`, `Or`, `Not`, `If` are pretty self explanatory except that you need to remember that we cannot simply return a boolean, it needs to be boxed. `Assume` is executing the body in a new environment, which is a union of bindings (from `elaborate`) and the current environment.

```
eval :: Env -> Ast -> Value
eval _ (Numa  x) = Numv  x
eval _ (Boola x) = Boolv x
eval m (Ida x)   = fetch m x
eval m (Add x y) = (eval m x) + (eval m y)
eval m (Mul x y) = (eval m x) * (eval m y)
eval m (Sub x y) = (eval m x) - (eval m y)
eval m (Div x y) = (eval m x) / (eval m y)
eval m (Equals x y)  = Boolv $ (eval m x) == (eval m y)
eval m (And x y)     = Boolv $ eval m x == Boolv True && eval m y == Boolv True
eval m (Or x y)      = Boolv $ eval m x == Boolv True || eval m y == Boolv True
eval m (Not x)       = Boolv $ if eval m x == Boolv True then False else True
eval m (IsZero x)    = Boolv $ (eval m x) == Numv 0
eval m (If c t e)    = if eval m c == Boolv True then eval m t else eval m e
eval m (Assume bs x) = eval m' x
  where m' = Map.union mb m
mb = elaborate m bs
```

The `elaborate` takes the current environment (for eval), the bindings, and returns a new environment only from the bindings. This environment needs to be composed with the current environment, as is done before.

```
elaborate :: Env -> [(Ast, Ast)] -> Env
elaborate m =  Map.fromList . map f
  where f (Ida x, e) = (x, eval m e)
```

The `fetch` does a lookup on the environment, which is a map, and if not available throws an error.

```
fetch :: Env -> String -> Value
fetch m id = case v of
    (Just x) -> x
    Nothing  -> error $ "id " ++ id ++ " not set!"
  where v = Map.lookup id m
```

7 Parser

As before, i wanted to depend upon the `read` function to generate the AST. While its simple for `Ida`, `And`, `Or`, `Not`, `If`, unfortunately it is not like that for `Assume`. In order for `Assume` to accept an array of pairs (tuples) as bindings, the first bracket needs to be square (for array) and the second needs to be round (for pair). Additionally, each item needs to be separated by comma, and not just space.

In order to perform this alteration, the whole input string is converted to words, which is then converted to a hierarchical bracket tree. All alterations are performed upon this bracket tree. Finally, the bracket tree is converted to a string which can then be directly parsed through `read` function.

Also we dont distinguish between square and round brackets, just like in racket, so square brackets are simply replaced with round brackets.

```
parse :: String -> Ast
parse s = (read . unwords . unpack . alter . Bnode "" . pack . words $ bpad) :: Ast
  where bpad = replace "(" " ( " . replace ")" " ) " . replace "[" "(" . replace "]" ")
```

Here is the alteration strategy strategy.

```
alter :: Btree -> Btree
alter (Bnode _ (Bleaf "assume":ns)) = (Bnode "(" (Bleaf "Assume":ns'))
  where (Bnode _ binds):exps = ns
ns' = (Bnode "[" binds'):exps'
binds' = intersperse comma . map toPair $ binds
toPair (Bnode _ xv) = Bnode "(" . intersperse comma . map alter $ xv
exps' = map alter exps
comma = Bleaf ","
alter (Bnode b ns) = Bnode b $ map alter ns
alter (Bleaf w) = Bleaf $ case w of
  "+" -> "Add"
  "*" -> "Mul"
  "-" -> "Sub"
  "/" -> "Div"
  "=" -> "Equals"
  "&" -> "And"
  "|" -> "Or"
  "~" -> "Not"
  "zero?" -> "IsZero"
  "if" -> "If"
```

```
  w
    | isFloat w  -> "(Numa "  ++ w ++ ")"
    | isBool  w  -> "(Boola " ++ w ++ ")"
    | isId    w  -> "(Ida \""   ++ w ++ "\")"
    | otherwise  -> w
```

Here are bracket tree functions, for converting words to bracket trees and vice versa.

```
data Btree =
  Bnode String [Btree] |
  Bleaf String
  deriving (Eq, Read, Show)

unpack :: Btree -> [String]
unpack (Bleaf w)  = [w]
unpack (Bnode b ns) = b : (foldr (++) [b'] $ map unpack ns)
  where b' = if b == "[" then "]" else (if b == "(" then ")" else "")

pack :: [String] -> [Btree]
pack [] = []
pack all@(w:ws)
  | isClose = []
  | isOpen  = node : pack ws'
  | otherwise = Bleaf w : pack ws
  where isOpen  = w == "[" || w == "("
isClose = w == "]" || w == ")"
node = Bnode w $ pack ws
ws' = drop (area node) all
win = pack ws

area :: Btree -> Int
area (Bleaf _) = 1
area (Bnode _ ns) = foldr (+) 2 $ map area ns
```

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

isId :: String -> Bool
isId (c:cs) = isAlpha c && all isAlphaNum cs
```

8 This is where you put it all together

```
import Data.List
import System.IO
import Data.Char
import qualified Data.Map as Map


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    |
  Ida    String  |
  Add    Ast Ast |
  Mul    Ast Ast |
  Sub    Ast Ast |
  Div    Ast Ast |
  Equals Ast Ast |
  And    Ast Ast |
  Or     Ast Ast |
  Not    Ast     |
  IsZero Ast     |
  If Ast Ast Ast |
  Assume [(Ast, Ast)] Ast
  deriving (Eq, Read, Show)

type Env = Map.Map String Value

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

run :: String -> Value
run = (eval $ Map.empty) . parse

eval :: Env -> Ast -> Value
```

```haskell
eval _ (Numa  x) = Numv  x
eval _ (Boola x) = Boolv x
eval m (Ida x)   = fetch m x
eval m (Add x y) = (eval m x) + (eval m y)
eval m (Mul x y) = (eval m x) * (eval m y)
eval m (Sub x y) = (eval m x) - (eval m y)
eval m (Div x y) = (eval m x) / (eval m y)
eval m (Equals x y)  = Boolv $ (eval m x) == (eval m y)
eval m (And x y)     = Boolv $ eval m x == Boolv True && eval m y == Boolv True
eval m (Or x y)      = Boolv $ eval m x == Boolv True || eval m y == Boolv True
eval m (Not x)       = Boolv $ if eval m x == Boolv True then False else True
eval m (IsZero x)    = Boolv $ (eval m x) == Numv 0
eval m (If c t e)    = if eval m c == Boolv True then eval m t else eval m e
eval m (Assume bs x) = eval m' x
  where m' = Map.union mb m
mb = elaborate m bs

elaborate :: Env -> [(Ast, Ast)] -> Env
elaborate m =  Map.fromList . map f
  where f (Ida x, e) = (x, eval m e)

fetch :: Env -> String -> Value
fetch m id = case v of
    (Just x) -> x
    Nothing  -> error $ "id " ++ id ++ " not set!"
  where v = Map.lookup id m


parse :: String -> Ast
parse s = (read . unwords . unpack . alter . Bnode "" . pack . words $ bpad) :: Ast
  where bpad = replace "(" " ( " . replace ")" " ) " . replace "[" "(" . replace "]" ")

alter :: Btree -> Btree
alter (Bnode _ (Bleaf "assume":ns)) = (Bnode "(" (Bleaf "Assume":ns'))
  where (Bnode _ binds):exps = ns
ns' = (Bnode "[" binds'):exps'
binds' = intersperse comma . map toPair $ binds
toPair (Bnode _ xv) = Bnode "(" . intersperse comma . map alter $ xv
exps' = map alter exps
comma = Bleaf ","
```

```
alter (Bnode b ns) = Bnode b $ map alter ns
alter (Bleaf w) = Bleaf $ case w of
  "+" -> "Add"
  "*" -> "Mul"
  "-" -> "Sub"
  "/" -> "Div"
  "=" -> "Equals"
  "&" -> "And"
  "|" -> "Or"
  "~" -> "Not"
  "zero?" -> "IsZero"
  "if" -> "If"
  w
    | isFloat w  -> "(Numa "  ++ w ++ ")"
    | isBool  w  -> "(Boola " ++ w ++ ")"
    | isId    w  -> "(Ida \""   ++ w ++ "\")"
    | otherwise  -> w


data Btree =
  Bnode String [Btree] |
  Bleaf String
  deriving (Eq, Read, Show)

unpack :: Btree -> [String]
unpack (Bleaf w)  = [w]
unpack (Bnode b ns) = b : (foldr (++) [b'] $ map unpack ns)
  where b' = if b == "[" then "]" else (if b == "(" then ")" else "")

pack :: [String] -> [Btree]
pack [] = []
pack all@(w:ws)
  | isClose = []
  | isOpen  = node : pack ws'
  | otherwise = Bleaf w : pack ws
  where isOpen  = w == "[" || w == "("
isClose = w == "]" || w == ")"
node = Bnode w $ pack ws
ws' = drop (area node) all
win = pack ws
```

```haskell
area :: Btree -> Int
area (Bleaf _) = 1
area (Bnode _ ns) = foldr (+) 2 $ map area ns


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

isId :: String -> Bool
isId (c:cs) = isAlpha c && all isAlphaNum cs
```