Basic Parsing and Interpreting

Homework 3

Christopher Salinas | Gavin McGuire

We wrote a program in Haskell to parse and interpret BASIC programs.

Our Basic Interpreter `BASIC.hs`, uses two primary modules:

- `BasicParser.hs` parses `.bas` files into `Program` data structures using `Parselib.hs`.
- `BasicInterpreter.hs`. interprets parsed `Program` data structures.

and three helper modules that we wrote:

- `BasicData.hs` to define data types and data structures.
- `Utils.hs` to manipulate program execution control.
- `Variables.hs` to update the variable environment the environment.

Usage

`Parselib.hs` that was provided to us in the #homework Slack channel.

Process

- Parser

  - The parser is ordered and organized using the Chipmunk Basic Grammar.
  - Uses `Parser` monads from `parselib.hs` to create a `Program` data structure.
  - It parses the strings into a list of Line Numbers and `Statements.`

- Interpreter

  - The `Interpreter` is a `ReaderT` monad which reads each `Statement`, and alters the `Environment`.
  - Each `Statement` execution is defined by Chipmunk Basics Manual.
  - It interprets given there are no errors. Otherwise, it uses minimal error handling.
  - Prints values and and strings.

Results

Running our makeFile with `make` and then `make run`, tests each `.bas` file with test cases and yields the following results:

```
Running amazing.bas for a 10x10
                            AMAZING PROGRAM
               CREATIVE COMPUTING  MORRISTOWN, NEW JERSEY
```

```
WHAT ARE YOUR WIDTH AND LENGTH




.--.  .--.--.--.--.--.--.--.--.
I     I              I        I
:  :--:--:--:  :  :--:  :--:  .
I  I           I     I     I  I
:  :  :--:--:  :--:  :--:--:  .
I     I     I  I     I     I  I
:  :--:  :  :--:  :--:  :  :  .
I        I     I        I     I
:  :--:--:--:  :--:--:--:--:--.
I  I     I     I           I  I
:  :--:  :  :--:  :--:--:  :  .
I        I        I  I        I
:  :--:--:--:--:--:  :  :  :  .
I  I           I     I  I  I  I
:  :  :--:--:  :  :--:  :  :--.
I  I  I        I     I  I     I
:  :  :--:--:--:  :  :  :--:  .
I  I     I     I  I     I     I
:  :--:  :  :  :  :--:--:  :--.
I        I  I     I           I
:--:--:--:  :--:--:--:--:--:--.


Running bubblesort.bas
RANDOM NUMBERS:
9.501768, 2.1173687, 7.572983, 0.59691966, 6.652684, 5.0798187, 1.9747853, 1.8351316, 7.4396667, 1.1871171
AFTER SORTING:
0.59691966, 1.1871171, 1.8351316, 1.9747853, 2.1173687, 5.0798187, 6.652684, 7.4396667, 7.572983, 9.501768


Running fib.bas
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
987
1597
2584
4181
6765
10946


Running foo.bas
14


Running gcd.bas with 15 and 20
WHAT IS THE VALUE OF X
WHAT IS THE VALUE OF Y
THE GCD OF 15 AND 20 IS 5


Running guess.bas
GUESS A NUMBER BETWEEN 1 AND 10
TOO LOW. GUESS AGAIN
TOO LOW. GUESS AGAIN
TOO LOW. GUESS AGAIN
TOO LOW. GUESS AGAIN
TOO LOW. GUESS AGAIN
TOO LOW. GUESS AGAIN
YOU WIN!


Running pascal.bas
NUMBER OF ROWS
     1



    1 1


   1 2 1


  1 3 3 1


 1 4 6 4 1


Running root.bas with 16
WHAT NUMBER DO YOU WANT TO KNOW THE SQUARE ROOT OF
THE SQUARE ROOT IS 4

Running root.bas with 11
WHAT NUMBER DO YOU WANT TO KNOW THE SQUARE ROOT OF
THAT NUMBER IS TOO HARD. MY BEST GUESS IS 3.3166246

Running sieve.bas
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 103 107 109 113 127 131 137 139 149 151 157
163 167 173 179 181 191 193 197 199 211 223 227 229 233 239 241 251 257 263 269 271 277 281 283 293 307 311 313 317
331 337 347 349 353 359 367 373 379 383 389 397 401 409 419 421 431 433 439 443 449 457 461 463 467 479 487 491 499
503 509 521 523 541 547 557 563 569 571 577 587 593 599 601 607 613 617 619 631 641 643 647 653 659 661 673 677 683
691 701 709 719 727 733 739 743 751 757 761 769 773 787 797 809 811 821 823 827 829 839 853 857 859 863 877 881 883
887 907 911 919 929 937 941 947 953 967 971 977 983 991 997

Running test.bas with 5
5
1
2
3
4
5
```

Code Examples

`BasicInterpreter.hs`
statement interpretation

```haskell
-------------------- EVAL STATEMENT --------------------
evalStatement :: Statement -> Interpreter (Bool, Bool)
---------- DIM ----------
evalStatement (DIM []) = return (True, True)
evalStatement (DIM (a:as)) = do
  initArray a
  evalStatement (DIM as)

---------- LET ----------
evalStatement (LET var rhs) = do
  val <- evalExpr rhs
  setVar var val
  return (True, True)

---------- PRINT ----------
-- Print New Line Statement
evalStatement (PRINT []) = do
  liftIO $ putStrLn ""
  liftIO $ hFlush stdout
  return (True, True)

-- Print Tab Statement
evalStatement (PRINT ((SepExpr _ (Tab e)):es)) = do
  n <- evalExpr e
  let s = replicate (truncate n) ' '
  ss <- evalPrintList es
  liftIO $ putStrLn $ s ++ ss
  liftIO $ hFlush stdout
  return (True, True)

-- Print SepExpr List
evalStatement (PRINT ((SepExpr _ e):es)) = do
  s <- evalPrintList $ e:es
  liftIO $ putStr s
  return (True, True)

-- Print List
evalStatement (PRINT es) = do
  s <- evalPrintList es
  liftIO $ putStrLn s
  liftIO $ hFlush stdout
  return (True, True)

---------- FOR ----------
-- If variable is Nothing, initialize it and handle it appropriately
-- If variable is (Just _), handle appropriately
evalStatement (FOR (Id var) init l ms) = do
  mv <- getVariable var
  limit <- evalExpr l
  step <- evalStep ms
  case mv of
    Nothing -> do
      val <- evalExpr init
      setVariable var val
      handleFor var val limit step
    Just val -> do handleFor var val limit step

---------- NEXT ----------
-- For each Id in the list, check what the variable resolves to
--   1. If Nothing, continue to next Id
--   2. If Just val,
--     a. Set var = val + step
--     b. Check limit
--       i. if met, continue
--       ii. GOTO associate FOR statement
evalStatement (NEXT []) = return (True, True)
evalStatement (NEXT ((Id var):ids)) = do
  v <- getVariable var
  case v of
    Nothing -> evalStatement (NEXT ids)
    Just val -> do
      (Line ln [FOR (Id var) _ l s]) <- findFor var
      limit <- evalExpr l
      step <- evalStep s
      let newVal = val + step
      if metLimit newVal limit step
        then do
          clearVariable var
          evalStatement (NEXT ids)
          return (True, True)
        else do
          setVariable var newVal
          evalStatement $ GOTO $ Num $ fromIntegral ln

---------- IFTHEN ----------
evalStatement (IFTHEN e ln) = do
  b <- evalComparison e
  if b
    then evalStatement $ GOTO ln
    else return (True, True)

---------- INPUT ----------
evalStatement (INPUT str vars) = do
  when (str /= "") $ do liftIO $ putStrLn str
  inputLines vars
  return (True, True)

---------- GOTO ----------
evalStatement (GOTO ln) = do
  nextLine <- findLine $ truncate (val ln)
  setLineIx nextLine
  return (False, True)

---------- GOSUB ----------
evalStatement (GOSUB ln) = do
  (program, vars, _, arrs) <- ask
  nextLine <- findLine $ truncate (val ln)
  newLineArr <- liftIO $ newArray (1, 1) nextLine
  liftIO $ runReaderT evalProgram (program, vars, newLineArr, arrs)
  return (True, True)

---------- ONGOTO ----------
evalStatement (ONGOTO e lns) = do
  ix <- evalExpr e
  ln <- evalExpr $ lns !! (truncate ix - 1)
  evalStatement $ GOTO (Num ln)

---------- RETURN ----------
evalStatement RETURN = return (False, False)

---------- REM ----------
evalStatement (REM _) = return (True, True)

---------- END ----------
evalStatement END = liftIO exitSuccess
```

`Variables.hs`

variable manipulation

```
-- Variable
setVariable :: Char -> Float -> Interpreter ()
setVariable c v = do
```

```haskell
  (_, vars, _, _) <- ask
  liftIO $ writeArray vars c $ Just v

getVariable :: Char -> Interpreter (Maybe Float)
getVariable c = do
  (_, vars, _, _) <- ask
  liftIO $ readArray vars c

clearVariable :: Char -> Interpreter ()
clearVariable c = do
  (_, vars, _, _) <- ask
  liftIO $ writeArray vars c Nothing

-- 1D Array Variable
create1D :: Int -> IO Array1D
create1D w = newArray (0,w-1) 0

dim1D :: Char -> Int -> Interpreter ()
dim1D c w = do
  (_, _, _, arrs) <- ask
  arr <- liftIO $ create2D 1 (w+1)
  liftIO $ writeArray arrs c $ Just arr

setVariable1D :: Char -> Int -> Float -> Interpreter ()
setVariable1D c = setVariable2D c 0

getVariable1D :: Char -> Int -> Interpreter (Maybe Float)
getVariable1D c = getVariable2D c 0

-- Create a 2D array
-- 1. Repeat w, h times. For example: h=2, w=3 => [3, 3]
-- 2. Map create1D over each of the elements in the list from step 1
-- 3. Create a new array using the list from step 2
-- Result is a 2x3 matrix:
--   Nothing, Nothing, Nothing
--   Nothing, Nothing, Nothing
create2D :: Int -> Int -> IO Array2D
create2D h w = do
  l1 <- replicateM h (create1D w)
  newListArray (0,h-1) l1

-- Initialize a 2D array for the variable and size given
-- This creates a 2D array of size h+1 by w+1. It has indices (0,h) and (0,w)
dim2D :: Char -> Int -> Int -> Interpreter ()
dim2D c h w = do
  (_, _, _, arrs) <- ask
  arr <- liftIO $ create2D (h+1) (w+1)
  liftIO $ writeArray arrs c $ Just arr

-- ID(I,J) = V
setVariable2D :: Char -> Int -> Int -> Float -> Interpreter ()
setVariable2D c i j v = do
  (_, _, _, arrs) <- ask
  ma2 <- liftIO $ readArray arrs c
  case ma2 of
    Nothing -> error $ "Found Nothing for " ++ [c]
    Just a2 -> do
      a1 <- liftIO $ readArray a2 i
      liftIO $ writeArray a1 j v

-- Get value at ID(I,J)
getVariable2D :: Char -> Int -> Int -> Interpreter (Maybe Float)
getVariable2D c i j = do
  (_, _, _, arrs) <- ask
  ma2 <- liftIO $ readArray arrs c
  case ma2 of
    Nothing -> return Nothing
    Just a2 -> do
      a1 <- liftIO $ readArray a2 i
```

```
        v <- liftIO $ readArray a1 j
        return $ Just v
```

`BasicData.hs`

Data structures and definitions

```
    data Expr = Id Char
              | Num { val :: Float }
              | Int' Expr
              | Rnd Expr
              | Tab Expr
              | OrExpr Expr Expr
              | AndExpr Expr Expr
              | NotExpr Expr
              | CompareExpr String Expr Expr
              | MultExpr Char Expr Expr
              | AddExpr Char Expr Expr
              | NegateExpr Expr
              | PowerExpr Expr Expr
              | Array' Expr [Expr]
              | String' { str :: String }
              | SepExpr String Expr

  ...

    data Statement = DIM [Expr]
                   | END
                   | FOR Expr Expr Expr (Maybe Expr)
                   | GOSUB Expr
                   | GOTO Expr
                   | IFTHEN Expr Expr
                   | INPUT String [Expr]
                   | LET Expr Expr
                   | NEXT [Expr]
                   | ONGOTO Expr [Expr]
                   | PRINT [Expr]
                   | REM String
                   | RETURN
  ...
    -- Program is an array of BasicLines
    type Program = [BasicLine]

    -- Vars
    type Vars = IOArray Char (Maybe Float)

    -- Arrays is an array that stores all float variables
    type Array1D = IOArray Int Float
    type Array2D = IOArray Int Array1D
    type Arrays = IOArray Char (Maybe Array2D)

    -- CurrentLine is an array of size 1 that holds the current line number
    type CurrentLine = IOArray Int Int

    -- The Environment is a tuple of the program, variables, and current line
    type Environment = (Program, Vars, CurrentLine, Arrays)

    -- The Interpreter is a ReaderT that holds the Environment of the program
    type Interpreter = ReaderT Environment IO
```