Writing a basic RPN interpreter in Haskell

Introduction

Our goal is to write a simple interpreter for reverse Polish notation[1] in Haskell. It should read a file line by line, interpreting the results, possibly as assignment statements, and print the result of computing each line in an output file. All input values are to be positive integers, or strings representing previously assigned variables (an unassigned variable in an expression will trigger an error, we could just as easily replace it by 0). The operations allowed in the present instance are `+` `*` and unary −. As will be clear from the code, adding additional operators would require adding precisely one line per operator.

When I first specified this problem I intended to allow only single character variable names. It turned out to be just as convenient, and in fact easier, to allow any variable name beginning with an alphabetic character (and not containing any whitespace of course!)

```
module RPNInterpreter where
import IO
```

These header lines define our module, and note that we need to make use of the IO library.

The I/O bit

The program breaks naturally into two parts:

- An outer framework for handling the I/O,
- A functional core which interprets a line, represented as a list of tokens.

In this section we define the I/0 framework. Since the intended syntax of the command is:

`Main>` *rpn "foo"*

meaning that the interpreted lines be read from "*foo.in*" and written to "*foo.out*", our top level function is simply called *rpn*. All that it does is to open two handles for reading and writing, pass them to the interpreter, and when that finishes, closes the handles and exits.

```
rpn fileName = do
                 inHandle ← openFile (fileName++".in") ReadMode
                 outHandle ← openFile (fileName++".out") WriteMode
                 interpreter inHandle outHandle []
                 hClose inHandle
                 hClose outHandle
                 return ()
```

The interpreter processes the input file a line at a time. As usual, this is done recursively, it processes a line, and then calls itself. When this raises an error (owing to EOF) it exits (this is the meaning of the "catch" block). An important addition at this level is the *state*. This is a list of

[1] See `http://www-stone.ch.cam.ac.uk/documentation/rrf/rpn.html`

pairs: (variableName, variableValue) that needs to be maintained line by line in order to look up any variables found in an expression. In particular the *processLine* function has a primary effect of returning a new state, and a secondary side effect of putting the output value onto the output file.

```
interpreter hIn hOut state = catch
             (do newState ← processLine hIn hOut state
                 interpreter hIn hOut newState)
             (\_ → return ([]))

processLine hIn hOut state =
  do
      thisLine ← hGetLine hIn
      let (newState, outValue) = evaluateLine (words thisLine) state
       in
           do hPutStrLn hOut (show outValue)
              return newState
```

The *processLine* function, finally accesses the functional guts of the program by calling *evaluateLine*. It prepares a list of tokens by using the built in function *words* to break up this line, and then evaluates it. The *evaluateLine* function returns a pair consisting of the value of the line and a new state (which differs from the old state only for assignment statements). The "let ... in" statement pattern matches this result to isolate the output value, and the new state, and then the second "do" block places the output value in the output file, and returns the new state.

The functional core

This part of the is where all the real work happens. The I/O section was just dull window dressing. The basic idea is that we have: a sequence of tokens representing a line, and a sequence of pairs representing the present "state" (values of assigned variables). The tasks relating to state are to look up, or find the values of variables, and to add new values to the state when we make an assignment. Evaluation of the line itself is just a matter of pushing the values of integer or variable tokens onto a stack (list), or evaluating an operation on the top or top two elements of the stack.

Matters of state In these brief functions we describe the type in which we hold the state. It is a simple list of pairs consisting of a `String` and an `Int`. The first member of the pair is a variable name and the second its value. Whenever we need to update the state, we simply add a new element to the front. We don't have to worry about duplicate values because the *lookup* function (built in) takes `(key,value)` pairs from a list and returns the first value which matches a key (so, as long as our updates precede old values in the list, the old values are hidden). One slight subtlety is that *lookup* uses the *Maybe* type to indicate success or failure, so our *find* function has to allow for this in terms of returning a value or an error message.

```
type State = [(String, Int)]

update :: (String, Int) → State → State
update = (:)

find c state = case (lookup c state) of
        (Just v) → v
        (Nothing) → error "Attempt to use uninitialised variable"
```

Evaluation To evaluate a line we must check whether or not we have an assignment statement. If we do, we make use of the *asg* function to both produce the value and update the state. If not, the state will not change, and we simply evaluate the whole token list with an initial empty stack.

```
evaluateLine :: [String] → State → (State, Int)

evaluateLine tokenList state
        | assignmentStatement   = asg tokenList state
        | otherwise             = (state, eval tokenList [] state)
        where assignmentStatement = ((length tokenList) ≥ 2) &&
                                        tokenList!!1 == "="
```

To carry out the *asg* function, we evaluate the remainder of the token list (after dropping its first two parts), and return an updated state and the value.

```
asg :: [String] → State → (State, Int)

asg tokenList state = (newState, outValue)
                      where outValue = eval (drop 2 tokenList) [] state
                            newState = update (newVar, outValue) state
                            newVar   = (tokenList!!0)
```

At last we come to the actual evaluation function. The first argument is the token list, the second the stack, and the third the state. If we reach empty token list and empty stack something has gone wrong! Otherwise when we reach empty token list, we wish to report the top element on the stack.

```
eval :: [String] → [Int] → State → Int

eval [] [] _ = error "Empty stack at end of evaluation?"
eval [] (s:_) _ = s
```

Finally, we come to the case where there are still tokens to be processed. These are either binary operators, a unary operator, variables, numbers, or errors. We handle each case with a guard. An appropriate action is taken, and then the *eval* function is called recursively with the remainder of the tokens. Variables are identified as strings whose first character is alphabetic (`isAlpha (t!!0)`). Numbers are identified as strings all of whose characters are digits (`and (map isDigit t)`).

```
eval (t:ts) stack state
        | (t == "*")    = eval ts (evalBinOp (*) stack) state
        | (t == "+")    = eval ts (evalBinOp (+) stack) state
        | (t == "-")    = eval ts (evalUnOp negate stack) state
        | isVar         = eval ts ((find t state):stack) state
        | isNumber      = eval ts ((read t :: Int):stack) state
        | otherwise     = error "In eval: unacceptable input."
        where isVar    = isAlpha (t!!0)
              isNumber = and (map isDigit t)
```

Evaluating a binary operator on a stack with at least two elements is easy. If there are too few, then an error has ocurred. Unary operators are even easier.

```
evalBinOp op (s0:s1:ss) = ((s0 `op` s1):ss)
evalBinOp _ _           = error "Binary operation on short stack"

evalUnOp op (s:ss)      = (op s):ss
evalUnOp _ _            = error "Unary operation on empty stack"
```

And that's all! I make it 52 actual lines of code, including some which are merely type declarations and not strictly speaking necessary. By the way, both the program and the explanatory text are in a single `.lhs` file which can be loaded directly to Haskell, but is also imported into a `.tex` file using the `listings` package and producing this document!