How to build a monadic interpreter in one day - Summary

Binbin Weng (binbinw2)

This article introduces us how to build a monadic interpreter step by step.

To build a simple monadic interpreter, the task is split into two parts: front-end and back-end. The first step to build an interpreter is to construct the abstract syntax tree AST which is involved in the back-end, so the plan is to build the back-end in the morning and the front-end in the afternoon.

In the morning, what should be done for the back-end is as the follows:

1. declare the AST. We define two data structures for expression and command separately.

```
data Exp = Constant Int
         | Variable String
         | Plus Exp Exp
         | Minus Exp Exp
         | Greater Exp Exp
         | Less Exp Exp
         | Equal Exp Exp
         | Times Exp Exp
         | Divide Exp Exp
         deriving Show
data Com = Assign String Exp
         | Seq Com Com
         | Cond Exp Com Com
         | While Exp Com
         | Declare String Exp Com
         | Print Exp
         deriving Show
```

2. prepare a small program including commands for testing purpose - (referred as s1 program). The testing program should involve almost all kinds of expressions and commands so that it can test each kind of expressions and commands.

```
-- this is for parsing; after parsing, the result should the same as
-- the AST below
s1String = "declare x = 150 in declare y = 200 in {while y > 0 do
           if x > 100 then x:=x/2 else y:=y-100; print x}"
-- this is the AST for interpreting
s1 = Declare "x" (Constant 150)
       (Declare "y" (Constant 200)
          (Seq (While (Greater (Variable "y") (Constant 0))
                  Cond (Greater (Variable "x") (Constant 100))
                    (Assign "x" (Divide (Variable "x") (Constant 2)))
                    (Assign "y" (Minus (Variable "y") (Constant 100)))))
          (Print (Variable "x"))))
```

3. declare the data structures of environment and some functions used to manipulate it, so that values of variables can be stored and reused later in the program.

```
-- this is what the environment will look like.
-- Index is for the variable names, Stack is for the variable values
-- Location is the location of the variable in the Stack
type Location = Int
type Index = [String]
```

```
type Stack = [Int]

-- declare functions used to manipulate environment
-- including the functions to get the location of a variable,
-- to get the nth value from the Stack, to replace a value in the Stack.
-- Please see the detailed functions in the program file provided.
```

4. choose the “State and Output” monad. The monadic semantics allows us to use do-notation.

```
newtype M a = StOut (Stack -> (a, Stack, String))
-- remember to declare Functor M and Applicative M
-- because if something is declared as Monad,
-- it must be Functor and Applicative
instance Monad M where
    return x = StOut (\\n -> (x, n, ""))
    e >>=  f = StOut (\\n -> let (a, n1, s1) = (unStOut e) n
                                (b, n2, s2) = unStOut (f a) n1
                            in (b, n2, s1++s2))
--unStOut is to extract the embeded function from a monadic capsule
unStOut (StOut f) = f
```

5. define some basic operations with monadic values. All the computations will take place in the monad, so we need the operations working on monads to fulfill the computations.

```
-- return a value from the environment
getfrom :: Location -> M Int
getfrom i = StOut (\\ns -> (fetch i ns, ns, ""))
-- modify the stack
write :: Location -> Int -> M()
write i v = StOut (\\ns -> ((), put i v ns, ""))
-- push the value as top/head of the stack
push :: Int -> M ()
push x = StOut (\\ns -> ((), x:ns, ""))
-- remove the element from the top of the stack
pop::M ()
pop = StOut (\\m -> let (n:ns) = m in ((), ns, ""))
```

6. prepare the evaluator for expressions

```
eval1 :: Exp -> Index -> M Int
eval1 exp index =
    case exp of
        Constant n -> return n
        Variable x -> let loc = position x index
                      in getfrom loc
        Plus x y -> do { a <- eval1 x index;
                         b <- eval1 y index;
                         return (a+b)}
        Minus x y -> do { a <- eval1 x index;
                          b <- eval1 y index;
                          return (a-b)}
        Greater x y -> do { a <- eval1 x index;
                            b <- eval1 y index;
                            return (if a > b then 1 else 0)}
        Less x y -> do { a <- eval1 x index;
                         b <- eval1 y index;
                         return (if a < b then 1 else 0)}
        Equal x y -> do { a <- eval1 x index;
                          b <- eval1 y index;
                          return (if a == b then 1 else 0)}
        Times x y -> do { a <- eval1 x index;
```

```
                        b <- eval1 y index;
                        return (a*b)}
        Divide x y -> do { a <- eval1 x index;
                         b <- eval1 y index;
                         return (if b /= 0 then a `div` b else  0)}
```

7. prepare the evaluator for commands

```
interpret1 :: Com -> Index -> M ()
interpret1 stmt index =
    case stmt of
        Assign name e ->
            let loc = position name index
            in do { v <- eval1 e index;
                    write loc v}
        Seq s1 s2 ->
            do {x <- interpret1 s1 index;
                y <- interpret1 s2 index;
                return () }
        Cond e s1 s2 ->
            do {x <- eval1 e index;
                if x == 1 then interpret1 s1 index
                else interpret1 s2 index}
        While e b ->
            let loop () = do { v <- eval1 e index;
                               if v == 0 then return ()
                               else do {interpret1 b index;
                               loop()} }
            in loop ()
        Declare nm e stmt ->
            do { v <- eval1 e index;
                 push v;
                 interpret1 stmt (nm:index);
                 pop}
        Print e ->
            do { v <- eval1 e index;
                 output v}

output :: Show a => a -> M ()
output v = StOut (\\n -> ((),n,show v))
```

In the afternoon, what should be done for the front-end is as the follows:

1. prepare the grammar of the language. The grammar as the follows should be in your mind, not in the interpreter.

```
<rexp> ::= <rexp> <relop> <expr> | <expr>
<expr> ::= <expr> <addop> <term> | <term>
<term> ::= <term> <mulop> <factor> | <factor>
<var> ::= <var> | <digiti> | (<expr>)
<digiti> ::= <digit> | <digit><digit>
<digit> ::= 0 | 1 | ... | 9
<addop> ::= + | -
<mulop> ::= * | /
<relop> ::= = > | < | =
<com> ::= <assign> | <seqv> | <cond> | <while> | <declare> | <printe>
<assign> ::= <identif> ":=" <rexp>
<seqv> ::= "{" <com> ";" <com> "}"
<cond> ::= "if" <rexp> "then" <com> "else" <com>
<while> ::= "while" <rexp> "do" <com>
<declare> ::= "declare" <identif> "=" <rexp "in" <com>
```

```
<printe> ::= "print" <rexp>
```

2. prepare a parser combinator or choose a parser combinator library like Parsec or ParseLib.

```
newtype Parser a = Parser (String -> [(a,String)])
-- parse is to extract the function from the monadic capsule
parse :: Parser a -> String -> [(a, String)]
parse (Parser p) = p
-- remember to declare Parser as Functor and Applicative
instance Monad Parser where
    return a = Parser (\\cs -> [(a,cs)])
    p >>= f = Parser (\\cs -> concatMap

                          (\\(a,cs')-> parse (f a) cs') $ parse p cs)
-- MonadPlus is to deal with the possibility of multiple parsing
instance MonadPlus Parser where
    mzero = Parser (\\cs -> [])
    p `mplus` q = Parser (\\cs -> parse p cs ++ parse q cs)
-- remember to declare it as Alternative as well

-- the parser combinator includes many functions which are able to produce
-- other parsers like functions to create a specific Char, String, spaces,
-- blanks, tabs, symbol,and variables
```

3. combine simple parser until you get a parser of the whole language. We need parsers for digits, characters, expressions, and commands, and then combine them together. (Please see the detailed parsers in the program file provided.)
4. parse the source of s1 program to test if the interpreter works. (please see the README for detailed testing)
5. combine front-end, back-end in one file. We can define the main function as the follows so that we can test the language we just built. (Please see the README for detailed testing)

```
run::String -> Com
run = runParser com
runParser :: Parser a -> String -> a
runParser m s =
    case parse m s of
        [(res,[])] -> res
        [(_,_)] -> error "Parse error."

main::IO()
main = forever $ do
    putStr ">"
    a <- getLine
    case unStOut(interpret1 (run a) [])[] of
        (_, _, a) -> print a
        _ -> error "error."
```

Conclusion

This is just a simple monadic interpreter. There is a large space we can work on to perfect our language based on our needs. This article introduces steps of building such an interpreter, which gives us a whole clear picture of building monadic interpreter.

The code and README file is available on github at https://github.com/binbinweng/cs421-finalproject.git.

The link of the article, *How to build a monadic interpreter in one day*, is https://wiki.haskell.org/wikiupload/c/c6/ICMI45-paper-en.pdf