The State monad revisited

Now it is time to revisit the state monad, in more generality. Recall the definition (using ST now instead of PST, for the state transformer function). The state monad has two parameters, one for the type of the state we try to maintain, and one for the kind of value being produced/contained:

```
data State s a = ST (s -> (a, s))
```

What we store is really a “state transformation”: Given a certain current state, a value of type `State s a` returns a pair of a transformed state and a produced value.

Here are some common methods for working with `State s a` values:

```
-- Creates state out of function. Just another name for ST
state :: (s -> (a, s)) -> State s a
state = ST

-- Runs the transformation on initial state
runState :: State s a -> s -> (a, s)
runState (ST f) s = f s

-- Runs the transformation on initial state and only returns value
evalState :: State s a -> s -> a
evalState = fst . runState

-- Returns the current state as value
get :: State s s
get = state $ \s -> (s, s)

-- Sets the state, ignoring any previous state
put :: s -> State s ()
put s = state $ \ _ -> ((), s)

-- Update the state to the result of applying the function
modify :: (s -> s) -> State s ()
modify f = state $ \ s -> ((), f s)

-- Gets specific "result" of applying function to state
gets :: (s -> a) -> State s a
gets f = state $ \ s -> (f s, s)

-- Note that we can define:  get = gets id
-- Note that we can define:  put = modify id

instance Functor (State s) where
    -- fmap :: (a -> b) -> State s a -> State s b
    fmap f st = state $ \s ->
        let (x, s') = runState st s
        in (f x, s')

instance Applicative (State s) where
    -- pure :: a -> State s a
    pure a = state $ \s -> (a, s)

    -- (<*>) :: State s (a -> b) -> State s a -> State s b
    stf <*> stx = state $ \s ->
        let (f, s')  = runState stf s
            (x, s'') = runState stx s'
        in (f x, s'')

    -- (*>) :: State s a -> State s b -> State s b
    stx *> sty = state $ \s ->
        let (_, s') = runState stx s
        in runState sty s'

instance Monad (State s) where
    -- return :: a -> State s a
    return = pure

    -- (>>=) :: State s a -> (a -> State s b) -> State s b
    stx >>= f = state $ \s ->
        let (x, s') = runState stx s
        in runState (f x) s'

    -- (>>) :: State s a -> State s b -> State s b
    -- (>>) = *>   No need to define explicitly
```

Example

Imagine a simple program that reads the string of an arithmetic expression, and wants to ensure that the expression is valid, so things like: - No multiple operators next to each other, like `2 + * 3`. - Open/close parentheses and square brackets are properly matching, i.e. nothing like `[(2+3] + 1)` or `(2+1) + 1)`. - No operators right after an opening parenthesis or before a closing parnethesis (or square brackets), i.e. no `(+ 2)` `(2*3+)`. - No numbers immediately preceding an open parenthesis/bracket or immediately following a closing parenthesis/bracket.

While there are many ways to solve this, we will solve it by creating a function:

```
checkMatching :: [Char] -> State ([Char], Char) Bool
```

So this function takes as input the string, and using a state consisting of a list(stack) of characters (to remember which parentheses/braces are currently "open" as well as a `Char` storing the last character seen. We can then use this on a string by doing:

```
runState (checkMatching s) []
```

Let's start by thinking of the big picture:

```
checkMatching [] = -- Check stack empty and last char not operator
checkMatching (x:xs) |
  opens x      =  -- Check last char not number
                  -- add x to stack
                  -- set as last char
                  -- recurse
  closes x     =  -- Check last char not operator
                  -- Pop from stack and compare for matching
                  -- set as last char
                  -- recurse
  isdigit x    =  -- Check last char not closing
                  -- set as last char
                  -- recurse
  isoperator x =  -- Check last char not operator
                  -- Check last char not open
                  -- set as last char
                  -- recurse
```

We start with some simple helper functions acting on characters:

```
operator :: Char -> Bool
operator = (`elem` "+-*/")

digit :: Char -> Bool
digit = (`elem` "0123456789")

pairs :: [(Char, Char)]
pairs = [('(', ')'), ('[', ']'), ('{', '}')]

opens :: Char -> Bool
opens = (`elem` map fst pairs)

closes :: Char -> Bool
closes = (`elem` map snd pairs)

matches :: Char -> Char -> Bool
matches c c' = (c, c') `elem` pairs

validEnd :: Char -> Bool
validEnd c = digit c || closes c
```

One of the things we will need is a way to "check" what the last character was. So for example, we would like to be able to write something like:

```
last digit :: State ([Char], Char) Bool
```

And this would be true exactly when the last character is a digit. We can use gets here. Remember that gets :: (s -> a) -> State s a, so it takes a function that processes state and gives us back a State s a value that uses the function on the current state (without changing the state). In our case we would need to provide it with the function:

```
-- ([Char], Char) -> Bool
digit . snd        -- Same as `(_, c) -> digit c`
```

Therefore the overall definition would become:

```
last :: (b -> c) -> State ([a], b) c
last p = gets (p . snd)   -- More mysteriously:  last = gets . (. snd)
```

Next we need something to help us check that the stack is empty. We can again think of that via a function that given a predicate on a stack returns a bool (we write the type more generally, with the return type being c rather than Bool:

```
stack :: ([a] -> c) -> State ([a], b) c
stack p = gets (p . fst)
```

Then we can write for example, stack null to check if the stack is empty.

So now we are almost ready to do the first part, dealing with the empty list (end of input). Conceptually what we want to do is this:

```
checkMatching [] = last validEnd && stack null  -- Not valid code!
```

The problem is the && here: It is supposed to act on Bools. But what we have instead are State ([Char], Char) Bools. The solution is exactly what Applicative was designed for, and the answer in our context is typically expressed in terms of the liftA2 function:

```
import Control.Applicative (liftA2)
-- liftA2 :: Applicative f => (a -> b -> c) -> f a -> f b -> f c
-- In our case:
liftA2 :: (Bool -> Bool -> Bool) -> State s Bool ->
                                    State s Bool -> State s Bool
```

So liftA2 will take the function to apply, in our case (&&), and the two State ... values to apply it to, and return the combination:

```
(<&&>) :: State s Bool -> State s Bool -> State s Bool
(<&&>) = liftA2 (&&)
```

Now we can write:

```
checkMatching [] = last validEnd <&&> stack null
```

For all the other cases, we need to create some more tools. For example, we need methods push, pop and setLast, which manipulate the state. Here are some possible definitions, with some helper methods there:

```
mapFst :: (a -> b) -> (a, c) -> (b, c)
mapFst f (x, y) = (f x, y)

mapSnd :: (a -> b) -> (c, a) -> (c, b)
mapSnd f (x, y) = (x, f y)

onStack :: (a -> a) -> State (a, b) ()
onStack f = modify $ mapFst f

push :: a -> State ([a], b) ()
push c = onStack (c :)

pop :: State ([a], b) a
pop = stack head <* onStack tail

setLast :: b -> State (a, b) ()
setLast c = modify $ mapSnd (\_ -> c)
```

Here we used the Applicative operator (<*) which said: Perform the two steps in sequence but then return the value of the first one. That's because we want to return the top element, which we saw via stack head, but then also remove it, which we did with onStack tail. And we cannot do those two things in opposite order, or we'll be looking at the wrong element.

Now we are ready to start thinking about our main cases. Recall the plan:

```
checkMatching (x:xs) |
  opens x      =  -- Check last char not number
                  -- add x to stack
                  -- set as last char
                  -- recurse
```

So if we encounter an opening mark, the main idea is to add it. But only if the last character was not a number. If it is a number then we have found an error, and can return False early. Note however that this "early False" behavior is not what is supported by the function <&&> we wrote earlier: In that case, both *effects* are applied, regardless of whether the final answer can be determined by the first effect. We need something that behaves more like the && operator in other languages: Only the first part will happen, if the second part is not needed.

We can accomplish this by using the monad aspect of State s a:

```
ifM :: State s Bool -> State s a -> State s a -> State s a
ifM stest strue sfalse = do
  b <- stest
  if b then strue else sfalse
-- Without do: ifM sb st sf = sb >>= (\b -> if b then st else sf)

(<&&^>) :: State s Bool -> State s Bool -> State s Bool
sb1 <&&^> sb2 = ifM sb1 (return False) sb2
```

With that in mind, we can now approach our first item:

```
checkMatching (x:xs)
  | opens x     = last (not . digit) <&&^> -- Check last char not number
                  (push x >> goOn)         -- add x to stack
      where goOn = setLast x >> checkMatching xs
```

So if the next thing on the list is an opening mark, then we check that the last character is not a digit, and if that succeeds then we proceed to push x to the stack before continuing with goOn, which is a shortcut for "set the last char to x and recursively continue".

Next we have the closing mark case:

```
  | closes x    = last (not . operator) <&&^> -- Check last char not operator
                  stack (not . null) <&&^>    -- Check stack nonempty
                  stack ((`matches` x) . head) <&&^>  -- Compare top of stack
                  (pop >> goOn)               -- Pop from stack
--  the where goOn goes here ...
```

The last two cases don't really require much explanation:

```
  | digit x     = last (not . closes) <&&^> goOn -- Check last char not closing
  | operator x = last (not . operator) <&&^> -- Check last char not operator
                  last (not . opens) <&&^>    -- Check last char not open
                  goOn
--  the where goOn goes here ...
```

So some of the key takeaways from that should be:

- Using Applicative and Monad operations we can express succinctly some pretty complex ideas.

- In order to work effectively with the State monad, we created helper functions of our own which operated with our specific state in ways that made sense for our use-case.

One thing remains: We need to initialize our state: What is the "last element" when we start the list? The state monad above is good at expressing the main loop through the list, but it is not enough to get us started. For that we will need another function, check:

```
check :: String -> Bool
check [] = False      -- The empty string is NOT a valid expression
check (x:xs)
  | operator x     = False   -- Cannot start with an operator
  | closes x       = False   -- Cannot start with a closing mark
  | otherwise      = evalState (checkMatching xs) (stack, x)
      where stack = if opens x then [x] else []
```

The interpreter using this fuller version of the state monad

We will now use the state monad more extensively in our simple language interpreter. We will make some minor changes however:

```
type Symbol = String  -- alias for strings when used as language symbols/variables.
data Expr = Numb Double       -- number literal
          | Var Symbol        -- variable lookup
          | Add Expr Expr     -- expression for addition
          | Prod Expr Expr    -- expression for multiplication
          | Seq Stmt Expr     -- first compute stmt, then expr
          deriving (Eq, Show)
data Stmt = Assign Symbol Expr  -- variable assignment
          | Print Expr          -- print the result of an expression evaluation
          | PrintMem            -- print all stored values
          deriving (Eq, Show)

type Value = Double      -- Doubles are the only possible values in this language
type Memory = [(Symbol, Value)]
```

The main difference is that we moved the sequencing operator to expressions. We can now have an expression that consists of a statement followed by an expression. So this would allow something like (x := 3; (y := 2; x + y)), via:

```
(Seq (Assign "x" (Numb 3))
     (Seq (Assign "y" (Numb 2))
          (Add (Var "x") (Var "y"))))
```

The key difference this makes is this: We must allow the possibility that when we evaluate an expression it may actually change the state. So our expressions will now also need to interact with the memory state. So the types of our two main evaluation functions would be:

```
evalExpr :: Expr -> State Memory Value
evalStmt :: Stmt -> State Memory (IO ())
```

Let's take a look at the code, there's lots to talk about. We start with an easy case:

```
evalStmt PrintMem = printMemory <$> get
```

Recall that <$> is the fmap. It takes the function printMemory :: Memory –> IO () and elevates it to act as State Memory Memory –> State Memory (IO ()). It is then applied to the value get :: State Memory Memory.

It's worth taking a look at the new version of printMemory which uses the IO Monad functions:

```
–– Using sequence_ :: [IO ()] –> IO ()
printMemory :: Memory –> IO ()
printMemory = sequence_ . map printPair

printPair :: (Symbol, Value) –> IO ()
printPair (s, v) = putStrLn $ s ++ "␣=␣" ++ show v
```

Next, we take a look at printing the result of an expression:

```
  –– print :: Value –> IO ()
evalStmt (Print expr) = print <$> evalExpr expr
```

This once again uses <$> to allow print :: Value –> IO () to turn evalExpr expr :: State Memory Value into State Memory (IO ()).

Finally, assignment.

```
evalStmt (Assign symbol expr) = do
  v <– evalExpr expr
  return <$> modify (store symbol v) –– return :: a –> IO a
  –– alternative:
  –– evalExpr expr >>= \v –> return <$> modify (store symbol v)
```

Assigning is done as follows:

- Evaluate the expression, to get a value v. This may also change the memory.
- We modify the memory using the store function. Then return <$> ... makes sure that we end in a IO () rather than a (). Recall that <$> is the fmap, so it takes the function return :: () –> IO () and "upgrades" it to a function State Memory () –> State Memory (IO ()).

The do notation makes sure that the memory gets updated through each step along the way, as needed.

Now we look at evalExpr. Some parts are easy:

```
–– return :: a –> State Memory a
evalExpr (Numb d) = return d
```

Here return is the function that turns a value into a state transformation that doesn't actually transform anything and simply returns the value.

The case of variable lookup is in reality simply, but complicated by the fact that we need to handle errors in lookup.

```
-- gets :: (Memory -> Maybe Value) -> State Memory (Maybe Value)
-- lookup :: Symbol -> Memory -> Maybe Value
-- gets (lookup s) :: State Memory (Maybe Value)
evalExpr (Var s) = valueOrError <$> gets (lookup s)
    -- valueOrError :: Maybe Value -> Value
    where valueOrError (Just v) = v
          valueOrError Nothing  = error ("Cannot␣find␣symbol:␣" ++ s)
```

Next, we have the two operations for adding and multiplying expressions. We use the do notation of the State Memory monad to chain the memory updates through each step: The expression e2 is evaluated in the memory resulting after expression e1 is evaluated.

```
evalExpr (Add e1 e2) = do
    v1 <- evalExpr e1
    v2 <- evalExpr e2
    return $ v1 + v2
evalExpr (Prod e1 e2) = do
    v1 <- evalExpr e1
    v2 <- evalExpr e2
    return $ v1 * v2
```

We can actually do those easier, using liftM2:

```
evalExpr (Add e1 e2) = liftM2 (+) (evalExpr e1) (evalExpr e2)
evalExpr (Prod e1 e2) = liftM2 (*) (evalExpr e1) (evalExpr e2)
```

Lastly, we need to handle the sequencing of a statement and an expression:

```
evalExpr (Seq st expr) = evalStmt st >> evalExpr expr
```

This evaluates the statement, then ignores its result and evaluates the expression in the updated memory state.

Notice that part: "*ignores its result*". In other languages that's not a big deal: The point of the statement was to change the state, and it did so. BUT in Haskell there is a huge difference: This result is the IO () action which does whatever printing we want from the program. And that printing will no longer happen: Only the effects of the assignments will do anything (and the prints will only matter at the "top-level" statement.

We will need to address this, and we will do so in the next segment. There are three possible "fixes":

0. Make evalExpr return an IO Value instead of a Value, for an overall type of State Memory (IO Value). This will change our code in many places, as we are effectively dealing with *two monads*.
1. This extends the previous, by using a new type for State which involves effectively three arguments, one of which is a monad (IO in our example). In brief, a State s m a will be a function that takes a state s and returns an m (a, s). Remember here that m is a monad, for example IO, so we are returning IO (a, s) instead of the (IO a, s) of the previous part. This is a fairly advanced approach, but it is the one taken by the built-in State class.

2. Add the IO actions produced as a part of the maintained State. So now our state will consist of both Memory and Output where Output is a list of IO () actions accumulated during the evaluation process.