Exercise (Week 6)

DUE: Tue 16 July 2019 14:00:00

CSE Stack

Download the exercise tarball and extract it to a directory in your home directory at CSE. This tarball contains a file, called `Ex04.hs`, wherein you will do all of your programming.

To test your code, run the following shell commands to open a GHCi session:

```
$ 3141
newclass starting new subshell for class COMP3141...
$ cabal repl
Resolving dependencies...
Configuring Ex04-1.0...
Preprocessing executable 'Ex04' for Ex04-1.0..
GHCi, version 8.2.2: http://www.haskell.org/ghc/  :? for help
[1 of 1] Compiling Ex04             ( Ex04.hs, interpreted )
Ok, one module loaded.
*Ex04> calculate "1 1 +"
...
```

Note that you will only need to submit `Ex04.hs`, so only make changes to that file.

Reverse Polish Notation

The Reverse Polish Notation is a common way to represent arithmetic expressions, where each operator appears after the operands it works on. Below are a number of examples in a sample GHCi session. Your task in this exercise is to develop a RPN calculator using a variety of monadic functions. You should be able to evaluate the same examples to the same results after you have completed this exercise:

```
*Ex04> calculate "3 4 +"
Just 7
*Ex04> calculate "3 4 - 5 +"
Just 4
*Ex04> calculate "3 4 2 / *"
Just 6
*Ex04> calculate "3 4 2 / * +"
Nothing
```

Generally, evaluation of RPN proceeds by looking at each token (number or operator), one at a time. We evaluate a number by pushing it on to a stack, and we evaluate an operator `o` by popping two numbers `y` and `x` from the stack, and pushing `o x y` back on. The number left on the top of the stack at the end of evaluating is the result of the expression.

Parsing (1 Mark)

We define a `Token` as either a number or a binary operator:

```
data Token = Number Int | Operator (Int -> Int -> Int)
```

First, we need a way to translate `String` values into proper calculable expressions, that is, lists of `Token`. We have already defined a function that will attempt to convert a single word into a single `Token`, called `parseToken`:

```
parseToken :: String -> Maybe Token
parseToken "+" = Just (Operator (+))
parseToken "-" = Just (Operator (-))
parseToken "/" = Just (Operator div)
parseToken "*" = Just (Operator (*))
parseToken str = fmap Number (readMaybe str)
```

Your first task is to implement the function `tokenise`:

```
tokenise :: String -> Maybe [Token]
```

This function must break the string into words (the built-in `words` function will be useful), and then attempt to parse each one into a `Token`, returning a non-`Nothing` result only if every word can be parsed.

Note that you can (and should) exploit the fact that `Maybe` is an instance of `Monad`, `Applicative` and `Functor`. This will enable you to make the above function into a short, one-line definition.

Stack Operations (2 Marks)

Next, in order to evaluate RPN expressions, we need a convenient way to model computations which manipulate a stack, and can possibly fail (for example, if we were to try to pop from an empty stack).

In Haskell, such a computation could be modelled as a function that, given an input stack (of `Int`), will either fail (returning `Nothing`) or return `Just` an output stack with an additional return value.

```
newtype Calc a = C ([Int] -> Maybe ([Int], a))
```

Implement the two basic stack operations, `push` and `pop` as `Calc` computations.

```
pop  :: Calc Int
push :: Int -> Calc ()
```

Evaluating (3 Marks)

We have provided instances of `Monad`, `Applicative` and `Functor` for `Calc`:

```
instance Functor Calc where
  fmap f (C sa) = C $ \s ->
      case sa s of
        Nothing      -> Nothing
        Just (s', a) -> Just (s', f a)

instance Applicative Calc where
  pure x = C (\s -> Just (s,x))
  C sf <*> C sx = C $ \s ->
      case sf s of
          Nothing     -> Nothing
          Just (s',f) -> case sx s' of
              Nothing      -> Nothing
              Just (s'',x) -> Just (s'', f x)

instance Monad Calc where
  return = pure
```

```haskell
  C sa >>= f = C $ \s ->
      case sa s of
          Nothing     -> Nothing
          Just (s',a) -> unwrapCalc (f a) s'
    where unwrapCalc (C a) = a
```

All three instances make sure that the whole computation will return `Nothing` if any subcomputation returns `Nothing`. Furthermore, the `Applicative` and `Monad` instances allow us to *thread the stack* through multiple computations without manually keeping track of this state. For example, the `Calc` computation that adds the two topmost elements of the stack and pushes the result back to the stack can be defined as:

```haskell
addTwo :: Calc ()
addTwo = do
  x <- pop
  y <- pop
  push (x + y)
```

By threading the stack through in the `Monad` instance here, we can just treat the stack as some implicit state that we manipulate abstractly with `Calc` computations. This allows us to make code much shorter and cleaner.

Using the above instances for `Calc`, define a function `evaluate`:

```haskell
evaluate :: [Token] -> Calc Int
```

This should evaluate a list of `Token` as described above, using the stack operations you defined earlier, finally returning the topmost element of the stack.

Putting it together (1 Mark)

Lastly, define a function `calculate`:

```haskell
calculate :: String -> Maybe Int
```

That will first attempt to parse the given string with `tokenise` to get a list of tokens. If that succeeds, it should use `evaluate` to get a `Calc` computation corresponding to that list of tokens. Then, it should run that computation starting with an empty stack, returning just the resultant integer from that computation, if any. If the stack has more than one element when `calculate` runs, it should return the topmost element.

Note that once again you can use the `Monad`, `Applicative` and `Functor` instances of `Maybe` to succinctly implement this function.

After implementing this function, test it with the examples listed at the beginning of this exercise, and try your own examples also.

Peer review (2 Marks)

Now that you've all made some fantastic artworks, we'd like each of you that submitted a picture to review one other student's picture. To do this, just log in to the peer review form here, and answer the questions the form asks you. Make sure you review **carefully** and **kindly**, because your review will be posted on the gallery page for that picture, and your identity is revealed to the person who made the image. To check the reviews for your own artworks, you can click here.

Submission instructions

You can submit your exercise by typing:

```
$ give cs3141 Ex04 Ex04.hs
```

on a CSE terminal, or by using the `give` web interface. Your file *must* be named `Ex04.hs` (case-sensitive!). A dry-run test will partially autotest your solution at submission time. To get full marks, you will need to perform further testing yourself.