Homework 3: Haskell Functor / Applicative

Programming Languages (Fall 2024)

Due: Same as Final Project

This homework will explore some Haskell ideas such as Monads, Fuctors, and Applicative within the context of Parsing.

Deliverables: Submit one file "∼/CS59/hw3.hs" on the pond which contains the necessary functions. In particular, you must define:

```haskell
{- Part 1 -}
term :: Char -> Parser Char
digit :: Parser Char
singularWhiteSpace :: Parser Char
endOfStream :: Parser ()
{- Part 2 -}
fmap :: (a1 -> a2) -> Parser a1 -> Parser a2
{- Part 3 -}
<*> :: Parser (a1 -> a2) -> Parser a1 -> Parser a2
<|> :: Parser a -> Parser a -> Parser a
{- Part 4 -}
greedyPlus :: Parser a -> Parser [a]
integer :: Parser Int
whiteSpace :: Parser ()
{- Part 5 -}
intCalculator :: Parser Int
```

Note: You can install `ghc` on the pond using `ghcup` without root. Just run:

```
curl --proto '=https' --tlsv1.2 -sSf https://get-ghcup.haskell.org | sh
```

Also Note: This is an extra credit assignment, but also highly encouraged!!

Part 1: A Basic Parser

Parsing is fundamentally just a way of interpreting input. A parser will take in some sequence of raw data (usually binary data in the form of bytes, but sometimes different forms of representation such as bits or text are more natural) and output some type of structure which is the parse tree of the object. For this exercise we will define the parsing operation on a string – So a parser is simply an object which can interpret a string and create some structure leaving a suffix of the string un-parsed. Of course a parser can also fail to parse the input (trying to interpret "abc" as an integer probably doesn't make a lot of sense, or, at least, likely you'd prefer that to fail than give you some unexpected result). This gives us a natural type definition:

```
newtype Parser a = Parser {
    parse :: String -> Maybe (a, String)
}
```

And then we can make some basic parsers!! Probably the simplest form of parser takes some character predicate (that is something which returns true or false depending on the character) and parses one character off of the string according to that predicate.

```
satisfy :: (Char -> Bool) -> Parser Char
satisfy pred = Parser f where
    f [] = Nothing
    f (x:xs)
        | pred x    = Just (x, xs)
        | otherwise = Nothing
```

And we can test that this does what we expect:

```
parse (satisfy ( == 'x')) "xyz" -- Returns: Just ('x',"yz")
parse (satisfy ( == '3')) "123" -- Returns: Nothing
parse (satisfy (\c -> True)) "" -- Returns: Nothing
```

Construct the following basic parsers (these should be relatively straightforward given the above examples!)

1. Write a constructor for a parser: `term`[1], which takes a `Char` and returns a `Parser Char` which will successfully parse the first character of the input if it is the character in question and will fail otherwise.
2. Write a constructor for a parser: `digit` which will parse one digit $(0 - 9)$ from the input.
3. Write a constructor for a parser `singularWhiteSpace` which will parse a whitespace character (consider only the four whitespace characters: space, tab, carriage return, and linefeed) from the input.
4. Write a constructor for a parser `endOfStream`: which succeeds if the string is empty and fails otherwise.

```
parse (term 'a') "abc"         -- Just ('a',"bc")
parse (term 'b') "abc"         -- Nothing
parse digit "a23"              -- Nothing
parse digit "4s2"              -- Just ('4',"s2")
parse digit ""                 -- Nothing
parse singularWhiteSpace " A"  -- Just (' ',"A")
parse singularWhiteSpace "\nB" -- Just ('\n',"B")
parse singularWhiteSpace "cat" -- Nothing
parse endOfStream ""           -- Just ((),"")
parse endOfStream "a"          -- Nothing
```

[1]`term` is short for "Terminal" which is the terminonigy used in parsing for a non-recursive singular-character construct. This is to say a parsing expression which will terminate consuming one symbol (usually character / byte) and without relying on the result of another parsing expression

Part 2: Parser as a Functor

Now, we made a deliberate choice above to only return a `Char` from each of the functions. But it is conceivable you would want to return something else from the parse. For instance we might want digit to return an `Int` instead of a `Char`. While we could build a wrapper which first uses `digit` to parse the input and then converts whatever character was parsed into an int, this is a highly specialized operation and we would prefer something more general. Namely, given a function of type `a -> b` we would like to be able to transform a parser which parses type `a` and construct a parser which parses type `b`. But wait!! This is just `fmap`! Let's define an `fmap` for `Parser`s:

```
instance Functor Parser where
    fmap = {- write function here! -}
```

While we can write `fmap` as some other function, making `Parser` a `Functor` comes with certain advantages. In particular, it means that you get to use some of Haskell's syntactic sugar (You might need to import Control.Applicative for `<$>` to be interpreted as fmap)!

```
-- Instead of writing:
    fmap func parser
-- We instead write:
    func <$> parser
```

This is particularly useful for associativeity of fmap:

```
    fmap f1 (fmap f2 (fmap f3 parser))
-- Simply becomes:
    f1 <$> f2 <$> f3 <$> parser
```

And to test that this works as expected, let's write a few helper functions:

```
import Text.Read

charToStr :: Char -> String
charToStr c = [c]

strToInt :: String -> Int
strToInt x = case (readMaybe x) of
    Nothing -> error "Cannot convert to Int"
    Just i -> i
```

Now we can we should test that fmap works as expected!

```
parse (charToStr <$> (term 'a')) "abc"          -- Just ("a","bc")
parse (charToStr <$> (term 'a')) "123"          -- Nothing
parse (strToInt <$> charToStr <$> digit) "6th" -- Just (6,"th")
parse (strToInt <$> charToStr <$> digit) ""    -- Nothing
```

Part 3: Parser as an Applicative

We might assume that requiring a Parser to have the Function restriction would be enough to easily build larger parsers, but we run into a problem when we want to apply a function to a longer sequence of parsers. This comes up quite frequently: say we want to parse one thing, and then parse another thing off of the input sequence. We could, of course write a function which does this using knowledge of how Parser was built, but say we want to keep things more general.
This is to say: If we have a function and we apply it to `f a` and `f b`, is there a way to get `f c`?

```
g                   :: a -> (b -> c)
fmap g              :: f a -> f (b -> c)
fmap g (f a)        :: f (b -> c)
fmap g (f a) (f b)  :: ??? -- We would like this to be: (f c)
```

It doesn't look like being a functor is enough! We also need to define an addional restruction on f, which makes it *Applicative*, which is to say, allows us to define an operation to convert `(f (b -> c)) (f b)` into `f c`. Luckily the `Control.Applicative` package gives us a nice general method of doing this!

```
import Control.Applicative
instance Applicative Parser where
    pure a = Parser (\s -> Just (a, s))
    (Parser aToB) <*> parserA = {- Insert your function here -}
```

Your implementation of `<*>` should return a new Parser which will first parse according to `aToB`. If that fails then the newly constructed parser should also fail. If that parser succeeds, then then new parser should then user the returned function in conjugation with `parserA` to parse what remains of the input.

```
combineTwoChars :: Char -> Char -> String
combineTwoChars a b = [a] ++ [b]

strAppend :: Char -> String -> String
strAppend c st = st ++ [c]

parse (combineTwoChars <$> (term 'a') <*> (term 'b')) "abc"  -- Just ("ab","c")
parse (strAppend <$> (pure "") <*> (term 'a')) "abc"         -- Just ("a","bc")
parse (strAppend <$> (pure "") <*> (term 'c')) "abc"         -- Nothing
parse (strToInt <$> (strAppend <$> (strAppend <$> (pure "") <*> digit) <*> digit)) "123"
                                                             -- Just (12,"3")
```

While this let's us build some Parsers, we are still missing a little bit of power: choice. Luckily Haskell's Applicative module also defines the Alternative class.

```
instance Alternative Parser where
    empty = Parser (\s -> Nothing)
    Parser p1 <|> Parser p2 = {- Your implementation here -}
```

Write an implementation for `<|>` which first tries to parse according to `Parser p1` and if that fails, then tries to parse according to `Parser p2`. This is an example of a "prioritized choice" operator from parsing theory.

```
parse ((term '1') <|> (term '2')) "123" -- Just ('1',"23")
parse ((term '1') <|> (term '2')) "234" -- Just ('2',"34")
parse ((term '1') <|> (term '2')) "345" -- Nothing
```

Part 4: Building Bigger Parsers

We can now use rules we have built for our Parser type to build bigger parsers! Consider wanting to build an operator `greedyStar`[2] which parses zero or more of some object. We could build it using our knowledge of Parser:

```haskell
greedyStar :: Parser a -> Parser [a]
greedyStar parseExp = Parser f where
    f s = case (parse parseExp s) of
        Nothing -> Just ([], s)
        Just (c, remaining) ->
            case (parse (greedyStar parseExp) remaining) of
                Nothing -> Just ([c], remaining)
                Just (rest, final) -> Just ([c] ++ rest, final)
```

Or we could make it far more simply (and less prone to error) using the type functions we know `Parser` is an instance of:

```haskell
greedyStar :: Parser a -> Parser [a]
greedyStar p = (:) <$> p <*> greedyStar p <|> pure []
```

And testing this out, we get what we exect:

```haskell
parse (greedyStar (term '1')) "1101"        -- Just ("11","01")
parse (greedyStar digit) "abc"              -- Just ("","abc")
parse (strToInt <$> greedyStar digit) "123" -- Just (123,"")
```

Just using the functions you have already defined (and small in-line helper functions – I.e. **without** using the definition of `Parser`) create three more parsers:

1. `greedyPlus`: which is like `greedyStar` but parses one or more of an object instead of zero or more.
2. `integer`: which parses an integer.
3. `whiteSpace`: which just strips off any amount of whitespace from the front of the string.

```haskell
parse (greedyPlus digit) "123a"   -- Just ("123","a")
parse (greedyPlus digit) "b456"   -- Nothing
parse integer "12ab"              -- Just (12,"ab")
parse integer "-456"              -- Just (-456,"")
parse integer "-00-"              -- Just (0,"-")
parse integer "--7cd"             -- Nothing
parse whiteSpace " \t\nhello! :)" -- Just ((),"hello! :)")
parse whiteSpace "42"             -- Just ((),"42")
```

[2] To those of you who have studied parsing theory before, this gets it's name from the *Kleene Star* operator. While regular expressing and context free grammars make use of a *non-deterministic choice*, our Alternative definition implements the *prioritized choice*. This means our star operator is greedy leading to a grammar language closer to Parsing Expression Grammars.

Part 5: Integer Calculator

We can use what we have built to create even more complex parsers. Consider a basic integer calculator which must be wary of matching parenthesis and consider order of operations: trying to build something from scratch might make this process daunting and error prone, but by building off of the smaller parsers we already have, this is not so bad! Classically a calculator will happen in two steps: parsing into an abstract syntax tree (AST) and then collapsing the tree via execution of arithmetic operation. In our case, the trees are simple enough we can do this in one step!

Write a parser `intCalculator` which takes in a string and either returns an int (the calculated value) if the parse was successful and nothing otherwise. You should ignore all white space and support the following operations: $\{+, -, *, /\}$ as well as properly match parenthesis.

Some tips:

- Far parsers `p2` and `p4` with the same type, and parsers `p1` and `p3`, the following are all equivalent:

  ```
  p4 = (\a1 a2 a3 -> a2) <$> p1 <*> p2 <*> p3
  p4 = (\a1 a2 -> a2) <$> p1 <*> ((\a1 a2 -> a1) <$> p2 <*> p3)
  p4 = flip const <$> p1 <*> (const <$> p2 <*> p3)
  p4 = p1 *> p2 <* p3
  ```

  This is because `Applicative f` also gives us the operations `<*` and `*>` which each check the parse is correct and implicitly fmap `const` and `flip const` over the results, respectively.

- Create several mutually recursive smaller parsers (one for `*` and `/`, one for `+` and `-`, etc...). Because of associativity, something like:

  ```
  addExp = (+) <$> multExp <* whiteSpace <* term '+' <* whiteSpace <*> addExp <|> ...
  ```

  Won't work (you'll end up with right associative parsing but math is, for better or worse, left associative). Instead you might consider `fmap`-ing `foldl (flip ($))` onto some expression (starting value) and a list of `Int -> Int` each of which partially apply their function to the second argument; things like `flip (-)` might be useful!

- For division, you may use the `div` function instead of `/` as integer division is easier than fractional division. You may also let the program error out (or whatever behavior you like) if it divides by zero.

```
parse intCalculator "  8  "                               -- Just (8,"")
parse intCalculator "1 2"                                 -- Nothing
parse intCalculator " \n      5   \t\r    *   \n \t -4  " -- Just (-20,"")
parse intCalculator "3 + -4 + 2"                          -- Just (1,"")
parse intCalculator "7 - 2 - 5"                           -- Just (0,"")
parse intCalculator "2+1*3/2"                             -- Just (3,"")
parse intCalculator "(1--2) * ( 4   + -1 ) "              -- Just (9,"")
parse intCalculator "7 - - 2"                             -- Nothing
parse intCalculator "1 + 2 * 3 - 4 * 5 / 6 + (7 - 8) * 9" -- Just (-5,"")
parse intCalculator "72 / 5 / 3 / 4"                      -- Just (1,"")
```