DelftX: FP101x Introduction to Functional Programming

EXERCISE 0 (1/1 point)

In these homework exercises we will use an implementation of the Parser combinators as used in the lectures that you can download from here.

Given this implementation of `Parser` and the associated combinators, what is the result of evaluating the expression: `parse item "hello"`?

- ( ) `[]`
- (•) `[('h', "ello")]`
- ( ) `[("h", "ello")]`
- ( ) `[("h", "hello")]`

You have used 1 of 1 submissions

EXERCISE 1 (1 point possible)

Assume "fast and loose" reasoning where there are no bottoms involved and all functions are total.

The parser `return 1 +++ return 2`:

- ( ) Always succeeds with the result value 2
- (•) Always succeeds
- ( ) Always fails
- ( ) Might fail

You have used 1 of 1 submissions

EXERCISE 2 (1/1 point)

What is the result of evaluating the expression `parse (return 1) "hello"` ?

- ( ) `[('h', "")]`
- ( ) `[('h', "ello")]`
- ( ) `[(1, "ello")]`
- (•) `[(1, "hello")]`

You have used 1 of 1 submissions

EXERCISE 3 (1 point possible)

What is the result of evaluating the expression `parse (item +++ return 'a') "hello"` ?

- (•) `[('h', "ello")]`
- ( ) `[('h', "aello")]`
- ( ) `[('a', "ello")]`
- ( ) `[('a', "hello")]`

You have used 1 of 1 submissions

EXERCISE 4 (1 point possible)

To make the type `Parser` (week X slide Y) a proper Monad we need to define the `(>>=)` (bind) operator and provide the required instance declaration for the Monad class such that we can use the do notation that was illustrated in the lectures.

```
newtype Parser a            = P (String -> [(a,String)])

instance Monad Parser where
   return                   :: a -> Parser a
   return v                 = P (\inp -> [(v,inp)])

   (>>=)                    :: Parser a -> (a -> Parser b) -> Parser b
   p >>= f                  = ...
```

Given the instance declaration above, choose a correct implementation of `(>>=)`, assume "fast and loose" reasoning where there are no bottoms involved and all functions are total.

- [ ]
```
p >>= f
  = P (\ inp ->
         case parse p inp of
             [] -> []
             [(v, out)] -> parse (f v) inp)
```

- [x]
```
p >>= f
  = P (\ inp ->
         case parse p inp of
             [(v, out)] -> parse (f v) out
             [] -> [])
```

- [ ]
```
p >>= f
  = P (\ inp ->
         case parse (f inp) inp of
             [] -> []
             [(v, out)] -> parse p out)
```

- [ ]
```
p >>= f
  = P (\ inp ->
         case parse (f inp) inp of
             [] -> []
             (v : out) -> parse (f v) out)
```

Read more about case expressions here:
https://www.haskell.org/onlinereport/haskell2010/haskellch3.html#x8-460003.13

You have used 1 of 1 submissions

EXERCISE 5 (1/1 point)

Assume "fast and loose" reasoning where there are no bottoms involved and all functions are total.

The parser `char 'a' +++ return 'b'`:

- ( ) Always succeeds with the result value 'a'
- ( ) Always succeeds with the result value 'b'
- (•) Always succeeds
- ( ) Always fails
- ( ) Might fail

You have used 1 of 1 submissions

EXERCISE 6 (1 point possible)

Given the following implementation of the parser `nat`, that parses a sequence of one or more digits:

```
nat :: Parser Int
nat
  = do xs <- many1 digit
       return (read xs)
```

Define a parser `int :: Parser Int` that parses an integer literal. An integer literal consists of an optional minus sign, followed by a sequence of one or more digits. Note: `"-007"` should parse as a valid integer according to this specification, but the resulting value is `-7`, just like GHCi does. Try it out!

Assume "fast and loose" reasoning where there are no bottoms involved and all functions are total.

- ( ) `int = char '-' >>= (\ c -> nat >>= (\ n -> (return (-n) +++ nat)))`
- ( ) `int = (nat +++ char '-') >>= (\ c -> nat >>= (\ n -> return (-n)))`

- [x]
```
int
  = (do char '-'
        n <- nat
        return (-n))
      +++ nat
```

- [ ]
```
int
  = (do char '-'
        nat)
      +++ nat
```

You have used 1 of 1 submissions

EXERCISE 7 (1/1 point)

Define a parser `comment :: Parser ()` for ordinary Haskell-like comments that begin with the symbol `--` and extend to the end of the current line, which is represented by the control character `'\n'` (beware Windows users!).

Note: `/=` is the syntax for "not equals" in Haskell. Yes, we know it's weird but it's not our fault.

Assume "fast and loose" reasoning where there are no bottoms involved and all functions are total.

- [ ]
```
comment
  = do string "--"
       sat (/= '\n')
       return ()
```

- [ ]
```
comment
  = do string "--"
       many (sat (/= '\n'))
```

- [ ]
```
comment
  = do string "--"
       sat (== '\n')
       return ()
```

- [x]
```
comment
  = do string "--"
       many (sat (/= '\n'))
       return ()
```

You have used 1 of 1 submissions

EXERCISE 8 (1 point possible)

Consider expressions built up from non-negative numbers, greater or equal to zero using a subtraction operator that associates to the left.

A possible grammar for such expressions would look as follows:

```
expr ::=  expr - nat | nat
nat  ::=  0 | 1 | 2 |...
```

However, this grammar is left-recursive and hence directly transliterating this grammar into parser combinators would result in a program that does not terminate because of the left-recursion. In the lectures of week 7 we showed how to factor recursive grammars using iteration.

Choose an iterative implementation of left-asociative expressions that does not suffer from non-termination.

Assume "fast and loose" reasoning where there are no bottoms involved and all functions are total.

```
expr
  = do n <- natural
       ns <- many
               (do symbol "-"
                   natural)
       return (foldl (-) n ns)
```

```
expr
  = do n <- natural
       symbol "-"
       n' <- natural
       return (n - n')
```

```
expr = do n <- natural
          ns <- many (do symbol "-"
                         natural)
```

```
expr
  = do n <- natural
       symbol "-"
       e <- expr
       return (e - n)
```

For further understanding try to implement the grammar directly using left-recursion and see what happens.

You have used 1 of 1 submissions