13 Parsing

Exercise 13.1 (*Warm-up*: Monads and Functors, `LiftM.hs`).
Given a function `f :: a → b` and an object `mx :: m a`, where `m` is an instance of `Monad` (i.e. `mx` is a monad holding a value of type `a`), the function `liftM` (in the module `Control.Monad`) applies the function `f` on the value inside the monad. Since every monad is also an applicative functor, it can be implemented using `fmap`:

```
liftM :: (Monad m) ⇒ (a → b) → m a → m b
liftM f mx = fmap f mx
```

or, equivalently, since `fmap` = `(<$>)`:

```
liftM f mx = f <$> mx
```

Define `liftM` **using only monadic operations**, i.e. `pure`, `(>>=)` and/or *do-notation*. Test your function on couple of simple examples:

```
⋙ liftM (+1) (Just 5)
Just 6
⋙ liftM (*2) [1..10]
[2,4,6,8,10,12,14,16,18,20]
⋙ liftM ("Hello, "++) getLine
Dave
"Hello, Dave"
```

(Practical note: you may encounter `liftM` in Haskell code written before 2018, but it has no advantage over `fmap` or `<$>` nowadays. Still, it is good to know how `fmap` relates to `(>>=)` and `pure`.)

Exercise 13.2 (*Warm-up*: Questions about parsing, `Trivia.hs`).
Parser combinators can be thought of as describing a tiny new language for describing grammars, fully implemented using Haskell operators. This idea is more generally called an *embedded domain specific language*. When developing grammars using this embedded language, problems can of course arise. Try to answer the following questions by reasoning in terms of the *grammar*, instead of the actual implementation of the Haskell functions:

1. Given this definition:

   ```
   dot :: Parser Char
   dot = char '.'
   ```

   Explain the output of:

   ```
   ⋙ parse dot ""
   ⋙ parse dot "."
   ⋙ parse dot ".."
   ⋙ parse dot "...etc"
   ```

2. Changing `dot` above in `many dot` (which is a `Parser [Char]`), explain the output of:

   ```
   ⋙ parse (many dot) ""
   ⋙ parse (many dot) "."
   ⋙ parse (many dot) ".."
   ⋙ parse (many dot) "...etc"
   ```

3. Why does using `many (many dot)` as a parser cause an infinite loop?

4. Instead of `many dot`, we can also use the equivalent parser:

   ```
   dots = (++) <$> many dot ⟨*⟩ many dot
   ```

   How many dots are being parsed by the first invocation of `many dot`, and how many by the second? Check your answer! *(Tip: change `(++)` in `(\x y→(x,y))` to be able to examine the individual components)*

Exercise 13.3 (*Warm-up*: Defining parser combinators, `Parser.hs`).
We can of course add many more parser combinators to the ones shown during the lecture. Here a few suggestions; try to implement them.

1. `space1 :: Parser ()`, which like `space` consumes white space, except that `space1` expects at least one space character:

   ```
    ⋙ parse space ""
   Just ((), "")
    ⋙ parse space "   "
   Just ((), "")
    ⋙ parse space1 ""
   Nothing
    ⋙ parse space1 "   "
   Just ((), "")
   ```

2. `endBy :: Parser a → Parser b → Parser [a]`; which is similar to `sepBy`, except that the second argument is a *terminator* instead of a *separator*. E.g.:

   ```
    ⋙ parse (integer ‘sepBy‘ symbol ",") "1,2,3,4,5,"
   Just ([1,2,3,4,5],",")
    ⋙ parse (integer ‘endBy‘ symbol ";") "1;2;3;4;5;"
   Just ([1,2,3,4,5],"")
   ```

3. `atMost :: Int → Parser a → Parser [a]`, which is similar to `many` in that it should parse zero or more items using the parser supplied as its second argument; except that there is a maximum amount of elements it is allowed to parse, given by its first argument.

4. *(Optional)* `between :: (Int,Int) → Parser a → Parser [a]`; with the intent that `between (a,b) p` parses *at least a* items and *at most b* items. (*Tip:* `replicateM` can be used to parse *exactly n* items.)

5. *(Optional)* `option :: Parser a → Parse (Maybe a)`; with the intent that `option p` should try to use the parser `p` and return its parsed value (wrapped in `Just`) if it succeeds; if `p` fails, `option p` should succeed without consuming any input and parse the value `Nothing`.

6. *(Optional)* `anythingBut :: Parser a → Parser ()`; with the intent that `anythingBut p` fails if `p` would succeed, and succeeds if `p` fails (in which case no input is consumed).

Exercise 13.4 (*Warm-up*: Worked example: a tiny parser, `Person.hs`).
We have seen this type before:

```
data Person = Person { studentName::String, studentAge::Int, favouriteCourse::String }
```

For which we can easily write a pretty printer:

```
pretty :: Person → String
pretty p = studentName p ++ " (" ++ show (studentAge p) ++ "), likes " ++ favouriteCourse p
```

1. We have already provided a parser `name :: Parser String` for parsing student and course names. Define a parser `person :: Parser Person` that is capable of processing the input of the pretty printer back into an actual `Person` value. You can use the `test :: Person → Person` function to test your parser.

2. Instead of having to type *"Kees (54), likes ICT and Society"*, we also want to support the shorter syntax *"Betsy (32): Assembly Language"*. This results in the grammar:

   ```
   person = name "(" natural ")" likes course
   course = name
   likes  = "," "likes"
          | ":"
   ```

   Extend the parser to implement this grammar.

Exercise 13.5 (Mandatory: Monadic parsing, `ListParse.hs`).
Parser combinators in the *applicative style* can describe context-free grammars. However, when using them in the *monadic style* (i.e. do-notation) we can also describe *context-sensitive* grammars.

1. Using the applicative style, define a parser `intList :: Parser [Int]` to process zero or more integers, separated by white space and surrounded by braces. Examples:

   ```
    ⋙ parseAll intList "{1 2 3}"
   Just [1,2,3]
    ⋙ parseAll intList "{}"
   Just []
    ⋙ parseAll intList "{1,2,3}"   -- "," are not allowed
   Nothing
   ```

2. Using the monadic style, define a parser `intRecord :: Parser [Int]` to processes zero or more integers in a similar notation, but where the input additionally contains a prefix denoting the number of elements that follow. Parsing for the entire record should fail if not exactly that amount of integers can be parsed:

   ```
    ⋙ parseAll intRecord "{2#1 2 3}"
   Nothing
    ⋙ parseAll intRecord "{3#1 2 3}"
   Just [1,2,3]
    ⋙ parseAll intRecord "{4#1 2 3}"
   Nothing
   ```

   (Tip: start by copying `intList` and translating it to do-notation, as practised in Exercise 12.3)

Exercise 13.6 (Mandatory: Parsing dice expression, `DiceParse.hs`). In Exercise 12.6, we have created an analyser for expressions containing dice throws:

```haskell
data Expr = Lit Int | Dice Int
          | Expr :+: Expr | ...
          | Min Expr Expr | Max Expr Expr
```

Which can represent expressions such as (2d6+2)/4, which is short-hand for "roll two 6-sided dice, compute their sum, add 2, and divide the result by 4, rounding down". It would be nice, of course, to have a parser for these short-hand expressions. We give a grammar for it here:

```
expr     = fraction

fraction = formula
         | term "/" positive

formula  = formula "+" term
         | formula "-" term
         | term

term     = "(" expr ")"
         | natural
         | [positive] "d" positive

positive = <a natural number greater than 0>
```

Write a parser `expr :: Parser Expr` that parses valid expressions into the `Expr` type.

- Don't try to define this parser all at once; start simple and reload often. We have provided *stubs* for every part of the grammar, which already allows you to parse very simple constant expressions such as *"23"*.
- In some cases (especially for the production rule for *formula*), the grammar cannot be translated directly to Haskell combinators, as discussed during the lecture; this can be solved by re-ordering the production rules and eliminating *left-recursion*.
- A reminder about the *meaning* of the short-hand notation: as discussed before, 4d6 means the *sum* of four dice rolls; 1d6 is commonly abbreviated to d6, but both are valid.
- Your parser should accept as precisely as possible the language described by the grammar above. For instance, the grammar above does not accept the input *"1+2/3"*, but forces the use of parenthesis to make clear whether *"(1+2)/3"* or *"1+(2/3)"* is meant. Similarly something like a 0-sided die (*"d0"*) is invalid and should be rejected. You *are* allowed to add extra grammar rules for min and max, and implement these, if you wish.
- The template for this exercise contains several test cases!

Exercise 13.7 (*Extra:* Error handling in parsers, `ParserErr.hs`).
Producing good quality parsing error messages is more of an art than a science; and you will probably already be familiar with compilers that are not exceptionally good at it (due to Haskell's terse syntax, GHCi's parsing errors are also not always clear).

While producing high quality error messages would be way too much work, we can at least extend the `Parser` type class so that instead of returning a `Nothing`, it also returns the first part of the input that could not be parsed. We can do this by changing the definition of `Parser` to:

```
type ParseResult t = Either String t
newtype Parser   a = P { parse :: String → ParseResult (a, String) }
```

Change the parser accordingly: whenever the parser previously returned a `Just (v,xs)`, it should now produce a `Right (v,xs)`, and `Nothing` will become `Left inp`, where `inp` is the offending input:

```
 >>> parse integer "123"
Right (123,"")
 >>> parse integer "FNORD"
Left "FNORD"
 >>> parse (symbol "##>") "##> abc"
Right ("##>", "abc")
 >>> parse (symbol "##>") "#@> abc"
Left "@> abc"
 >>> parse person "Vivian (21): Cryptography"
Right (Person {studentName = "Vivian", studentAge = 21, favouriteCourse = "Cryptography"},"")
 >>> parse person "Vivian (21), Likes Cryptography"
Left ", Likes Cryptography"
```

- Try type-oriented programming: GHCi's type errors will indicate where changes are needed!
- You will have to define an `instance Alternative ParseResult` (or change the instance for `Parser`), since the type class `Alternative` is not pre-defined for the `Either` type.
- If you succeed, you may find that the first incorrect character of input is not being reported; one way to fix this is to change the definition of `sat` to:

```
sat :: (Char → Bool) → Parser Char
sat p = do c ← getchar
           if p c then pure c
           else putback c
  where
  putback :: Char → Parser Char
  putback c = P (Left . (c:))
```