# *CIS 194: Homework 10*

*Due Monday, April 1*

- Files you should submit: `AParser.hs`. You should take the versions that we have provided and add your solutions to them.

## *Introduction*

A *parser* is an algorithm which takes unstructured data as input (often a `String`) and produces structured data as output. For example, when you load a Haskell file into `ghci`, the first thing it does is *parse* your file in order to turn it from a long `String` into an *abstract syntax tree* representing your code in a more structured form.

Concretely, we will represent a parser for a value of type `a` as a function which takes a `String` represnting the input to be parsed, and succeeds or fails; if it succeeds, it returns the parsed value along with whatever part of the input it did not use.

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

For example, `satisfy` takes a `Char` predicate and constructs a parser which succeeds only if it sees a `Char` that satisfies the predicate (which it then returns). If it encounters a `Char` that does not satisfy the predicate (or an empty input), it fails.

```
satisfy :: (Char -> Bool) -> Parser Char
satisfy p = Parser f
  where
    f [] = Nothing    -- fail on the empty input
    f (x:xs)          -- check if x satisfies the predicate
                        -- if so, return x along with the remainder
                        -- of the input (that is, xs)
        | p x       = Just (x, xs)
        | otherwise = Nothing  -- otherwise, fail
```

Using satisfy, we can also define the parser `char`, which expects to see exactly a given character and fails otherwise.

```
char :: Char -> Parser Char
char c = satisfy (== c)
```

For example:

```
*Parser> runParser (satisfy isUpper) "ABC"
Just ('A',"BC")
```

```
*Parser> runParser (satisfy isUpper) "abc"
Nothing
*Parser> runParser (char 'x') "xyz"
Just ('x',"yz")
```

For convenience, we've also provided you with a parser for positive integers:

```
posInt :: Parser Integer
posInt = Parser f
  where
    f xs
      | null ns   = Nothing
      | otherwise = Just (read ns, rest)
      where (ns, rest) = span isDigit xs
```

## *Tools for building parsers*

However, implementing parsers explicitly like this is tedious and error-prone for anything beyond the most basic primitive parsers. The real power of this approach comes from the ability to create complex parsers by *combining* simpler ones. And this power of combining will be given to us by... you guessed it, `Applicative`.

### Exercise 1

First, you'll need to implement a `Functor` instance for `Parser`.
*Hint*: You may find it useful to implement a function

```
first :: (a -> b) -> (a,c) -> (b,c)
```

### Exercise 2

Now implement an `Applicative` instance for `Parser`:

- `pure a` represents the parser which consumes no input and successfully returns a result of `a`.
- `p1 <*> p2` represents the parser which first runs `p1` (which will consume some input and produce a function), then passes the *remaining* input to `p2` (which consumes more input and produces some value), then returns the result of applying the function to the value. However, if either `p1` or `p2` fails then the whole thing should also fail (put another way, `p1 <*> p2` only succeeds if both `p1` and `p2` succeed).

So what is this good for? Recalling the `Employee` example from class,

```
type Name = String
data Employee = Emp { name :: Name, phone :: String }
```

we could now use the `Applicative` instance for `Parser` to make an employee parser from name and phone parsers. That is, if

```
parseName  :: Parser Name
parsePhone :: Parser String
```

then

```
Emp <$> parseName <*> parsePhone :: Parser Employee
```

is a parser which first reads a name from the input, then a phone number, and returns them combined into an `Employee` record. Of course, this assumes that the name and phone number are right next to each other in the input, with no intervening separators. We'll see later how to make parsers that can throw away extra stuff that doesn't directly correspond to information you want to parse.

## Exercise 3

We can also test your `Applicative` instance using other simple applications of functions to multiple parsers. You should implement each of the following exercises using the `Applicative` interface to put together simpler parsers into more complex ones. Do *not* implement them using the low-level definition of a `Parser`! In other words, pretend that you do not have access to the `Parser` constructor or even know how the `Parser` type is defined.

- Create a parser

  ```
  abParser :: Parser (Char, Char)
  ```

  which expects to see the characters `'a'` and `'b'` and returns them as a pair. That is,

  ```
  *AParser> runParser abParser "abcdef"
  Just (('a','b'),"cdef")
  *AParser> runParser abParser "aebcdf"
  Nothing
  ```

- Now create a parser

  ```
  abParser_ :: Parser ()
  ```

  which acts in the same way as `abParser` but returns `()` instead of the characters `'a'` and `'b'`.

```
*AParser> runParser abParser_ "abcdef"
Just ((),"cdef")
*AParser> runParser abParser_ "aebcdf"
Nothing
```

- Create a parser `intPair` which reads two integer values separated by a space and returns the integer values in a list. You should use the provided `posInt` to parse the integer values.

```
*Parser> runParser intPair "12 34"
Just ([12,34],"")
```

## Exercise 4

`Applicative` by itself can be used to make parsers for simple, fixed formats. But for any format involving *choice* (*e.g.* "…after the colon there can be a number **or** a word **or** parentheses…") `Applicative` is not quite enough. To handle choice we turn to the `Alternative` class, defined (essentially) as follows:

```
class Applicative f => Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a
```

`(<|>)` is intended to represent *choice*: that is, `f1 <|> f2` represents a choice between `f1` and `f2`. `empty` should be the identity element for `(<|>)`, and often represents *failure*.

Write an `Alternative` instance for `Parser`:

- `empty` represents the parser which always fails.
- `p1 <|> p2` represents the parser which first tries running `p1`. If `p1` succeeds then `p2` is ignored and the result of `p1` is returned. Otherwise, if `p1` fails, then `p2` is tried instead.

*Hint*: there is already an `Alternative` instance for `Maybe` which you may find useful.

## Exercise 5

Implement a parser

```
intOrUppercase :: Parser ()
```

which parses either an integer value or an uppercase character, and fails otherwise.

```
*Parser> runParser intOrUppercase "342abcd"
Just ((), "abcd")
*Parser> runParser intOrUppercase "XYZ"
Just ((), "YZ")
*Parser> runParser intOrUppercase "foo"
Nothing
```

Next week, we will use your parsing framework to build a more sophisticated parser for a small programming language!