Effective Functional Programming
Effect APIs
Assignment 4
`Calculator`

Paul Downen

Parsing is tricky business. It is one thing to read a flat file format, like an initialization file containing individual settings all listed separately on their own line. It is another thing to read documents organized with more complex, nested structure, like XML, JSON, or even arithmetic expressions. For example, consider the arithmetic expression `1-2+3*4^(-5+6)`; how would you write a program to correctly parse this string? There are binary operations like `+` and `*` that sit in the middle of sub-expressions that also have to be parsed before and after. You have to make sure that certain operations take precedence over others: the exponent `^` must be before the multiplication `*` that appears before it to the left, which must be before the very first addition `+` in the string. But this default precedence can be overridden by parenthesis: even though addition `+` happens last, the right-most `-5+6` must be first because it is grouped in parenthesis. This subexpression also includes a negative number, so the `-` symbol is ambiguous: it may be read a binary operation as in `1-2` or a unary negative sign as in `-5`.

So lets think of all the things a parser might be responsible for. In a perfect world, a parser of `a`s is a function that reads a `String` in its entirety and returns the one `a` value it represents. This perfect-world parser of `a`s has the type:

```
parsePerfect :: String -> a
```

But in the real world, parsing might fail because some `String`s don't correspond to any possible value that the function needs to return. For example, there isn't any sensible way to parse the string `"chair"` as a number, because no number is sufficiently sturdy to sit in. A parser that might fail when trying to produce an `a` has the type:

```
parseFail :: String -> Maybe a
```

where a result of `Nothing` signals a failed parse, and `Just x` signals that the input is successfully read as the value `x`.

Sometimes parsing can come across ambiguous words or phrases, which might need to be clarified by the context they appear in. For example, consider trying to parse the parts of speech (*e.g.,* nouns, verbs, adjectives, *etc.*). The string `"flies"`

has multiple different interpretations depending on the context it appears in. It could be read as the verb "flies" meaning soaring through the air, as in the sentence "Time flies like an arrow." It could also be read as the noun "flies" meaning the insect, as in the sentence "Fruit flies like a banana." Because of this ambiguity, a parser might have to give multiple different answers for the same string. A parser of ambiguous `a`s has the type:

```
parseAmbiguous :: String -> [a]
```

Finally, sometimes parsing can only read part of a string, rather than the whole thing. For example, parsing the string `"12"` as a number gives the result 12. If we tried to parse the larger string `"12+30"` as a number, we can still give the result 12, but there is also a left-over string `"+30"` that hasn't been parsed yet. A partial parser of `a`s which has only read some of the string, and returns how much more string remains to be processed, has the type:

```
parsePartial :: String -> (a, String)
```

Putting it all together a, parsing function might involve these complications:

- It might fail, producing no results.
- It might have multiple interpretations, producing two or more results.
- It might only parse part of the input, producing the remaining string that has yet to be parsed alongside its answer.

Each of these possibilities is captured by this single type for a parsing function:

```
parse :: String -> [(a, String)]
```

If `parse` fails it returns the empty list `[]`. If `parse` is given ambiguous input, it returns a list of two or more answers. And if an individual answer of `parse` is produced by only reading part of the input string, the unread portion of the string is returned along with that individual answer; answers created by reading the whole input have the empty `""` as their remainder.

This functionality can be captured in `N`on-`D`eterministic `Parser` data type, which contains an arbitrary parsing function of the same type as `parse` above.

```
data NDParser a = Parse (String -> [(a, String)])
```

If you're ever lost and forget how to parse things, remember this rhyme:

> *A Parser of Things*
> *is a function from Strings*
> *to lists of pairs*
> *of Things and Strings*

1 Parser Combinators (25 points& 5 extra credit)

The `NDParser` module found in the template file `src/NDParser.hs` already contains the above `NDParser` data type, as well as instances of the `Functor`, `Applicative`, `Alternative`, and `Monad` type classes which let you chain together the "effect" of parsing into bigger and bigger parsing operations. The `NDParser` template also provides the function

```
everyParse :: NDParser a -> String -> [(a, String)]
```

that runs the given parser of `a`s on the given input string, to produce a list of *every* possible `a` answer that can be parsed from the input, each `a` answer paired alongside the remaining portion of the input string that hasn't been read yet.

Your job in this section is to create some basic parser *combinators*. "Combinator" is just a fancy word for some simple building blocks and functions that stitch them together to create more sophisticated, complex programs. The `<*>` and `<|>` operations from `Applicative` and `Alternative`, and the `do`-notation made possible by `Monad` already fill the role of "stitching parsers together." What's missing, that you will fill in, is the basic parsing building blocks to be combined.

To complete the exercises below, fill in all the `undefined` parts of the `NDParser` module found in the template file `src/NDParser.hs`.

Exercise 1.1 (10 points)**.** Implement the `end` parser with this type signature

```
end :: NDParser ()
```

The `end` parser checks that it is at the end of the input string without reading any characters. So `end` should only "succeed" (meaning it returns a list of at least one answer) when it reads the empty string, and fails (meaning it returns an empty list of answers) when given any non-empty string.

In other words, `end` should be a `NDParser ()` of the form `Parse f`, where `f` is a function that takes a string `s`, and returns a different list of answers depending on the shape of `s`:

- If `s` is the empty `""`, then the function inside `Parse` should return a list containing one element: the pair of the unit value `()` as the "result" of the parser and the empty string `""` as the "remainder" of the unparsed input.
- If `s` is any other string of the form `c:cs`, then the function inside `Parse` should return an empty list signaling failure, since it is not already at the end of the input string. **End Exercise 1.1**

Hint 1.1. For help with how to approach your implementation of `end`, look at the definition of `next` right below it in the template file. `next :: NDParser Char` reads the next character of its given input string, and fails if there is no more input. A correct definition of `end` will look very similar to `next`, but with two important differences:

- `next` "fails" and returns an empty list when given an empty string, and "succeeds" in returning a non-empty list when given a non-empty string. These two cases are the opposite of `end` when "fails" on a *non-empty* input and "succeeds" on an *empty* input string.
- When `next` "succeeds" in reading a character from the non-empty input `c:cs`, it's single answer is `(c,cs)` which pairs the first character of the input with remaining unread characters that follow. Instead, when `end` "succeeds" by being given an empty string as input, it returns the single answer `((), "")` which pairs the placeholder value `()` with no remaining unread characters that follow. *End Hint 1.1*

Bonus Exercise 1.2 (5 extra credit)**.** Implement the parser function

```
feed :: Char -> NDParser ()
```

which does not read any characters, but instead *feeds* in a new character as the next one to be read in the input stream. The parser `feed c` should always succeed for any input string `s`, and its single answer should be the placeholder value `()` paired with the longer string `c:s` of unread characters that remain to be parsed. **End Bonus 1.2**

Hint 1.2. `feed` is the reverse operation as `next`. `next` undoes `feed`: running `feed c` followed by `next` on any input string `s` will always return the character `c` alongside the entire original input `s` as the unread remainder. For example,

```
> everyParse (do {feed '0'; x <- next; return x}) "abcd"
[('0',"abcd")]
```

In the opposite direction, `feed` undoes `#next#:`running `next` followed by `feed x`, where `x` is the answer of `next`, will "do nothing" if the input string is not empty. For example,

```
> everyParse (do {x <- next; feed x; return ()}) "abcd"
[((),"abcd")]
```

If the input string happens to be empty, then the first `next` operation will fail, so in that case `next` followed by `feed` will also fail. For example,

```
> everyParse (do {x <- next; feed x; return ()}) ""
[]
```

End Hint 1.2

Exercise 1.3 (10 points)**.** Implement the parser function `char` with the type signature

```
char :: Char -> NDParser Char
```

similar to `next`, `char c` can read the next character of the input string, and fails if the input string is empty. However, unlike `next`, `char c` only successfully reads the next character if it is *exactly* `c`, and fails if any other character is found as the next input. **End Exercise 1.3**

Hint 1.3. The template `NDParser` module provides the parser function

```
check :: (Char -> Bool) -> NDParser Char
```

which takes a test (a function of type `Char -> Bool` which returns `True` or `False` for any `Char`acter) and returns the next character of the input only if it passes the test (that is, the given function returns `True` for that next character). Any character which does not pass the test fails, as does an empty input string. See if you can use `check` to help implement `char`. *End Hint 1.3*

Exercise 1.4 (5 points)**.** Implement the `digit` parser with the type signature

```
digit :: NDParser Char
```

which parses just one digit character, that could be any one of `'0'`, `'1'`, `'2'`, ..., `'9'`.

Parsing one or more digits is implemented by the `digits :: NDParser String` parser is provided in the template `NDParser` module, which applies the generic `some` operation from `Alternative` to repeat your definition of a single `digit` above multiple times. **End Exercise 1.4**

Hint 1.4. The template `NDParser` module provides the parser function

```
oneOf :: [Char] -> NDParser Char
```

`oneOf [a, b, ..., z]` successfully parses the next character of the input string if it is *exactly one of* the characters `a`, `b`, ..., `z`. If the next character of the input string is not one of these, it fails, as does the empty input string. `oneOf` is implemented using the `<|>` operator from the `Alternative` type class which puts together two parsers of the same type. `p1 <|> p2` can be read as "parse the input using EITHER `p1` OR `p2`: if both succeed, combine the results; if only one succeeds, use its results, and if neither succeeds, fail."

For example, look at the definition of `space :: NDParser Char` which `oneOf` to parse any of the ASCII blank space characters: single space (`' '`), tab (`'\t`), new line (`'\n'`), or line feed (`'\f'`). You can implement `digit` similar to `space` by providing a different list of characters — those which represent a single digit rather than blank space — to the `oneOf` function. *End Hint 1.4*

2 Arithmetic Expressions (50 points& 30 extra credit)

Now that you have some basic building blocks, and ways to combine them together into more complex parsers, you can now start to write code for parsing real arithmetic expressions.

The `Calculate` module found in the template file `src/Calculate.hs` provides an outline and some helpful helper functions for your arithmetic expression parser. Fill in all the `undefined` parts of the `Calculate` module to complete the exercises below.

Exercise 2.1 (10 points)**.** The `Calculate` module template already includes the parser

```
naturalNumber :: NDParser Double
```

which parses any sequence of one or more `digits` as a natural number, 0, 1, 2, . . . , 42, . . . , and so on.

First, use `naturalNumber` to implement the `negativeNumber` parser with the type signature:

```
negativeNumber :: NDParser Double
```

The `negativeNumber` parser starts by reading the character `'-'`, then reads a `naturalNumber` that comes from one or more `digits`, and then returns the negation of that natural number. For example, `negativeNumber` parses the string `"-100"` as the number −100.

Second, use both the `naturalNumber` and `negativeNumber` parsers to implement the `integer` parser with the type signature

```
integer :: NDParser Double
```

`integer` will parse EITHER a `negativeNumber` OR a `naturalNumber`. Remember, the `<|>` operator lets you combine two parsers in an either-or fashion! **End Exercise 2.1**

Hint 2.1. Look at how `naturalNumber` is defined for help with how to write `negativeNumber`. You can use `naturalNumber` directly to parse a sequence of digits as a number; just remember to negate that number before you `return` it! *End Hint 2.1*

Bonus Exercise 2.2 (10 extra credit)**.** In addition to supporting integers, you can also parse fractional decimal points directly, too. To do so, implement the following parsers with the type signatures:

```
decimalFraction :: NDParser Double
float           :: NDParser Double
number          :: NDParser Double
```

The `decimalFraction` parser should read any sequence of one or more `digits` and interpret them as the fractional part that follows a decimal point. That means that the number `return`ed by `decmalFraction` should *always* be between 0 and 1. For example, if `decimalFraction` reads the `digits` `"12345"`, it should return the `Double`-precision floating-point number `0.12345`.

The `float` parser should start by parsing any `integer`, then it should parse the `char`acter `'.'`, next it should parse a `decimalFraction`, and finally it should return the addition of the parsed `integer` and the parsed `decimalFraction`.

`number` should parse EITHER an `integer` OR a `float`, whichever matches its input string. In the following Exercise 2.7, you can use your more general `number` parser instead of `integer` in defining `baseExpr`. **End Bonus 2.2**

Exercise 2.3 (10 points)**.** Implement the following three parsers with these type signatures:

```
times, divide, multiplicative :: NDParser (Double -> Double -> Double)
```

`times` parses exactly the `char`acter `'*'` and returns the Haskell multiplication function `(*)` as a result. `divide` is similar, and parses exactly the `char`acter `'/'` and returns the Haskell floating-point division function `(/)`.

Putting them together, the `multiplicative` parser should parse EITHER a `times` operator OR a `divide` operator. Remember, the `<|>` operator lets you combine parsers in an either-or way! **End Exercise 2.3**

Hint 2.2. For help with how to write `times`, `divide`, and `multiplicative`, look at the similar definitions provided in `Calculate` for

```
plus, minus, additive :: NDParser (Double -> Double -> Double)
```

Your definitions for `times` and `divide` should be very similar to `plus` and `minus`: the change is that a different character will be read, and a different Haskell function for the numeric operation corresponding to that character is returned. *End Hint 2.2*

Bonus Exercise 2.4 (5 extra credit)**.** Implement the `power` parser:

```
power :: NDParser (Double -> Double -> Double)
```

for the exponential operator. `power` should parse exactly the `char`acter `'^'` and returns the floating-point exponential operation named `(**)` in Haskell. **End Bonus 2.4**

Exercise 2.5 (10 points)**.** Implement the `trim` parser function with the type signature

```
trim :: NDParser a -> NDParser a
```

which surrounds any parser in any mount of blank `spaces`. The template `NDParser` already implements `spaces` which parses zero or more blank `space` characters. `trim p` should:

1. first, parse any `spaces` that might be found at the start of the input string,
2. second, parse the given `p`, and bind its result to `x` to remember it for later,
3. third, parse any additional `spaces` that might be found after step 2, and
4. fourth, return the `x` from step 2.

End Exercise 2.5

Hint 2.3. For help with how to write `trim`, look at the definition provided for `parenthesized :: NDParser a -> NDParser a`. `parenthesized p` modifies the parser `p` by parsing `p` surrounded by the open `'('` and close `')'` parentheses characters, which may be separated from `p` by any number of blank spaces. The result `return`ed by `parenthesized p` is whatever `p` returns inside of the parentheses and blank spaces. *End Hint 2.3*

Exercise 2.6 (10 points)**.** One of the main challenges with parsing arithmetic expressions is to correctly interpret the groupings of operations, called *associativity*. For example, the string `"10-1-2-3"` should be read as the *left associative* grouping $(((10-1)-2)-3)$ instead of the *right associative* grouping $(10-(1-(2-3)))$.

Implement these three pure functions with the following type signatures:

```
applyBinR :: (a -> b -> c) -> b -> (a -> c)
compose   :: (a -> b) -> (b -> c) -> (a -> c)
applyL    :: a -> (a -> b) -> b
```

`applyBinR f y` takes a two-argument, binary function `f` and the second argument `y` for `f`, and returns a function taking one argument `x` and calling `f` with `x` and `y`.

`compose f g` takes two functions, and returns a function taking one argument `x` and calling the composition of `g` and `f` on `x` so that `f` is applied first and `g` is applied second: `g (f x)`.

`applyL x f` takes an argument `x` for the function `f`, and returns the result of calling `f` with `x`.

These pure functions are used to implement the following parser combinators, provided for you in the template `Calculate` module:

```
binopR :: NDParser (a -> b -> c) -> NDParser b -> NDParser (a -> c)
chainR :: NDParser (a -> a -> a) -> NDParser a -> NDParser (a -> a)
assocL :: NDParser (a -> a -> a) -> NDParser a -> NDParser a
```

`binopR op right` parses a binary operator via `op` followed by its second argument via `right`, and returns the partial application of `op` to its second argument on the `right`. `chainR op subexpr` combines together a chain of partial, rightward applications of the binary operator parsed by `op` to the sub-expressions parsed by `subexpr`. Finally, `assocL op subexpr` parses left-associative applications of operations parsed by `op` applied to sub-expressions parsed by `subexpr`.

End Exercise 2.6

Hint 2.4. Do not change the type signatures of `applyBinR`, `compose`, or `applyL`!! While there are options for a few minor syntactic variations, in the end each of these type signatures has only *one* answer that fits. If you can replace the `undefined` with a value that type checks, then your answer will be correct.

End Hint 2.4

Exercise 2.7 (10 points)**.** Implement the following expression parsers with the type signature:

```
baseExpr, mulExpr :: NDParser Double
```

The base expression `baseExpr` should parse EITHER a `parenthesized addExpr` OR an `integer`. (Remember the either-or combinator `<|>`!)

The multiplication expression `mulExpr` should use `assocL` to parse left-associative `multiplicative` operators applied to sub-expressions from `baseExpr`.

End Exercise 2.7

Hint 2.5. The template `Calculate` module provides the addition expression `addExpr` (as needed to implement `baseExpr`) which parses left-associative `additive` operations applied to sub-expressions from `mulExpr`. You can use the definition of `addExpr` as an example for help with how to implement `mulExpr`.

The template also provides a top-level parser `expr :: NDParser Double` which `trim`s blank spaces before and after the expression, and makes sure the entire input string is read until the `end`. *End Hint 2.5*

Bonus Exercise 2.8 (15 extra credit)**.** Add exponential operators to your expression parser. To do so, first implement the parser combinators

```
binop  :: NDParser a -> NDParser (a -> b -> c) -> NDParser b -> NDParser c
assocR :: NDParser (a -> a -> a) -> NDParser a -> NDParser a
```

`binop left op right` parses the `left` sub-expression, then parses the operation given by `op`, then parses the `right` sub-expression, and finally combines all three together by returning the application of `op`'s operator to the results of `left` and `right`.

`assocR op subexpr` parses right-associative applications of operators parsed by `op` applied to sub-expressions parsed by `subexpr`. `assocR op subexpr` will be EITHER just a single `subexpr` or the `binop` application of a `subexpr` followed by an `op` followed by another right-associative `assocR op subexpr`.

Then, implement the parser of exponential power expressions:

```
powExpr :: NDParser Double
```

The power expression `powExpr` should parse right-associative (via `assocR`) applications of the `power` operator applied to `baseExpr` sub-expressions.

Finally, modify your definition of `mulExpr` from Exercise 2.7 to use `powExpr` as its sub-expressions, rather than `baseExpr`. **End Bonus 2.8**

3 *Bonus:* Parser API (45 extra credit)

Previously in section 1 you defined some basic general building blocks as `NDParser`s, and in section 2 you put those building blocks together to parse arithmetic expressions. However, there is nothing about parsing arithmetic that specifically requires `NDParser`; but there are other, potentially more efficient, implementations of parsing different from `NDParser` that would work just as well.

The `Parser` API defined in the `GenericParser` module

```
class (Alternative m) => Parser m where
  end    :: m ()
  char   :: Char -> m Char
```

defines the type signatures for the two primitive parsing operations `end` and `char` that you previously saw in section 1.

In addition to the primitives above, the `Parser` API also inherits the operations from `Alternative` (including the either-or combination `<|>` and the `empty`

parser that always fails) which in turn inherits the operations from `Applicative` (including the `pure` parser that returns a value without reading any input just like `return`, and the application `<*>` operator that parses a function, parses an argument, and then applies the function to the argument).

The optional bonus exercises that follow in this section have you generalize your arithmetic expressions to use *any* type of parser that has the right building blocks. This generalization involves creating a `Parser` type class that defines an API with the only essential primitive parsing-specific operations—on top of generic ones already provided by `Alternative`, `Applicative`, and `Functor`—needed to complete the rest of the exercises. All the other building blocks and arithmetic-specific parsers are then built on those two primitives. Abstraction away the specifics of parsing with the `Parser` API lets you quickly swap out one parsing implementation, like `NDParser` above, with another, like `ParseTable` below in section 4 while changing very few lines of code.

To complete this section, you will fill in the `undefined` parts of the `GenericParser` module in `src/GenericParser.hs`, and rewrite parts of the `Calculate` module in `src/Calculate.hs` to work with any generic `Parser` instead of the specific `NDParser`.

Bonus Exercise 3.1 (5 extra credit)**.** Define an `instance` of the `Parser` type class for `NDParser`. In other words, fill in the `instance Parser NDParser` block with definitions of the `end` and `char` functions. **End Bonus 3.1**

Hint 3.1. You already gave definitions of `end` and `char` in Exercises 1.1 and 1.3, so you can use that same code to implement the `end` and `char` functions of the `instance Parser NDParser` block. *End Hint 3.1*

Bonus Exercise 3.2 (10 extra credit)**.** Implement the generic `string` parser with the following type signature:

```
string :: Parser m => String -> m String
```

The `string` operation is similar to `char`, but it matches a whole `String` of multiple characters instead of just a single `Char`. Given a string `s` of zero or more characters, `string s` successfully returns `s` only if the input string starts with *exactly* `s`, otherwise `string s` fails. **End Bonus 3.2**

Hint 3.2. Note that the `Parser` API does not include the operations from the `Monad` API. That means that you cannot use `do`-notation for a generic `Parser`, which might not implement the `Monad` operations. Instead, you will have to implement `string` using only the operations made available from `Functor`, `Applicative`, `Alternative`, and `Parser` itself. *End Hint 3.2*

Bonus Exercise 3.3 (10 extra credit)**.** Implement generic versions of these parser combinators which work for *any* generic `Parser`, and not just the specific `NDParser`:

```
oneOf  :: Parser m => [Char] -> m Char
space  :: Parser m => m Char
```

```
spaces :: Parser m => m String
digit  :: Parser m => m Char
digits :: Parser m => m String
```

End Bonus 3.3

Hint 3.3. The definitions of `oneOf`, `space`, `spaces`, `digit`, and `digits` you give in `GenericParser` will look a lot like the ones from `NDParser`. The difference is that the `NDParser` versions are defined specifically to only work for parsers of the type `NDParser a`. In contrast, the `GenericParser` versions only rely on the `char` function from the generic `Parser` type class, as well as other operations from the type classes inherited by `Parser`: `Alternative` and `Applicative` and `Functor`. *End Hint 3.3*

Bonus Exercise 3.4 (20 extra credit)**.** Generalize the `Calculate` module by rewriting the top-level type signatures to use these more general types:

```
naturalNumber  :: (Num a, Read a, Parser m) => m a
negativeNumber :: (Num a, Read a, Parser m) => m a
integer        :: (Num a, Read a, Parser m) => m a

plus,  minus,  additive       :: (Num a,        Parser m) => m (a -> a -> a)
times, divide, multiplicative :: (Fractional a, Parser m) => m (a -> a -> a)

trim          :: Parser m => m a -> m a
parenthesized :: Parser m => m a -> m a

binopR :: Parser m => m (a -> b -> c) -> m b -> m (a -> c)
chainR :: Parser m => m (a -> a -> a) -> m a -> m (a -> a)
assocL :: Parser m => m (a -> a -> a) -> m a -> m a

baseExpr, mulExpr, addExpr :: (Floating a, Read a, Parser m) => m a
```

Additionally, if you previously implemented any bonus exercises in section 2, also generalize those type signatures like so:

```
decimalFraction :: (Fractional a, Read a, Parser m) => m a
float           :: (Fractional a, Read a, Parser m) => m a
number          :: (Fractional a, Read a, Parser m) => m a

power :: (Floating a, Parser m) => m (a -> a -> a)

binop  :: Parser m => m a -> m (a -> b -> c) -> m b -> m c
assocR :: Parser m => m (a -> a -> a) -> m a -> m a

powExpr :: (Floating a, Read a, Parser m) => m a
```

End Bonus 3.4

Hint 3.4. A generic `Parser m` might not be a `Monad`. That means that you can't use the monad operations like `return` and `(>>=)`, or `do`-notation in your generalized definitions. Make sure to replace all `return`s with `pure`s, and find a way to use the `Applicative` operations like `(<$>)` and `(<*>)` to write these parsers without using `do`. *End Hint 3.4*

4 *Bonus:* Parse Tables (60 extra credit)

While `NDParser` is an effective type of parser, it has a problem: it's entirely opaque. A value of `NDParser` is just a function, and there is no way to look inside and inspect that function without giving it a specific `String` input to parse and look at what it returns. As a consequence, `NDParser`s are hard to analyze, modify, and optimize.

If we want to be able to look more closely at how a particular parser is implemented, we need a more concrete representation that shows the individual steps of the parser. Such a concrete representation is given by the `ParseTable` data type given in the `src/ParseTable.hs` file:

```
data ParseTable a = Done a
                  | Fork [ParseTable a]
                  | Look (Map (Maybe Char) (ParseTable a))
                    deriving Show
```

There are three different basic kinds of `ParseTable`s: `Done x` immediately returns `x`, `Fork [p_1, ..., p_n]` combines the results from each alternative sub-parser `p_1`...`p_n`, and `Look table` checks the first character of the input string with the lookup `table` to decide what to do next.

As an example of how `ParseTable` is easier to analyze, notice how (unlike `NDParser`) we can derive an implementation of `Show` which prints out a full string representation of any `ParseTable`. For example, we can define this basic `ParseTable` using the `Parser` API that parses either one of the strings `"aba"`, `"abb"` or `"abc"`

```
abc :: ParseTable String
abc = string "aba" <|> string "abb" <|> string "abc"
```

`show`ing the value of `abc` in the interpreter prints this result (with better indentation):

```
> abc
Fork
  [ Fork
      [ Look (fromList
          [ (Just 'a',
             Look (fromList
               [ (Just 'b',
                 Look (fromList [(Just 'a', Done "aba")]))]))]),
```

```
      Look (fromList
        [ (Just 'a',
           Look (fromList
             [ (Just 'b',
               Look (fromList
                 [ (Just 'b', Done "abb")]))]))]),
  Look (fromList
    [ (Just 'a',
       Look (fromList
         [ (Just 'b',
            Look (fromList [(Just 'c',Done "abc")]))]))])]
```

That's quite a mouthful! And the worst part isn't just the size of that output. The description of `abc` says that it will try to read the input string three different times to look for a successfull parse. `abc` will first check that the input starts with the characters `'a'`, `'b'`, and `'a'` in that order. If it doesn't, it will go back to the beginning and check that the input starts with `'a'`, `'b'`, and `'b'`. If that fails, too, it goes back to the beginning yet again, and finally checks if the input starts with `'a'`, `'b'`, `'c'`, before giving up entirely.

This retracing and going back to the beginning is inefficient, because we keep rechecking that the first two characters are `'a'` and `'b'` every time. If we try to parse the string `"abd"`, then the initial `"ab"` will be checked three times before we figure out that none of the options match. It would be better for both performance, and for the size of the `ParseTable`, to combine these common checks into an optimized version of `abc` which never retraces back to earlier parts of the string; only looking at each character exactly once and then never again. This `optimize`d version of `abc` would look like:

```
> optimize abc
Look (fromList
  [ (Just 'a',
     Look (fromList
       [ (Just 'b',
          Look (fromList
            [ (Just 'a', Done "aba"),
              (Just 'b', Done "abb"),
              (Just 'c', Done "abc")]))])])
```

Now, this parser begins by just checking that the input starts with `'a'` and then `'b'`—anything else will fail. Assuming those two checks pare passed, then it looks for any one of `'a'`, `'b'`, or `'c'`, and returns the appropriate final result in each case.

This optional section has bonus exercises for implementing a parser based on `ParseTable`s, making basic `ParseTable` building blocks using the same generic `Parser` API from section 3, and finally analyzing and optimizing hask `ParseTable`s into, that you can use to more efficiently implement arithmetic calculations. To get started, the template file `src/ParseTable.hs` includes the

`ParseTable` data type above, along with instances of the `Functor`, `Applicative`, and `Alternative` type classes for it (but not `Monad`; why not?).

Bonus Exercise 4.1 (10 extra credit)**.** Implement the function `parseTable` of type

```
parseTable :: ParseTable a -> String -> [(a, String)]
```

which uses a `ParseTable a` data structure to process a given string input and return a list of possible `a` results paired with unread remainders of the input, similar to `everyParse` for `NDParser`. To implement this function, `parseTable p s` should pattern-match on its first argument `p` to say what to do in each case.

- `parseTable (Done x) s` doesn't look at the input string at all, and just immediately finishes parsing and returns a list containing only `x` paired with `s`.
- `parseTable (Fork ps) s` combines *all* results of parsing each alternative `parseTable p s`, where `p` is drawn from the list of `ParseTable`s `ps`.
- `parseTable (Look table) s` checks the first character of the input string `s` to decide what to do. The parameter `table` is represented as a `Map` (*i.e.,* a *dictionary* or *hash table*) that maps certain characters, which might appear at the beginning of the string, to the next `ParseTable` to use to parse the rest of the string.
  - In the empty string case `parseTable (Look table) ""`, `table` might map `Nothing` to a `ParseTable` which says what to do at the end of the input.
  - In the non-empty string case `parseTable (Look table) (c:s)`, `table` might map `Just c` to a `ParseTable` which says what to do for the character `c`.

  In either case, if `table` doesn't associate anything to the next character (or end of string), then `parseTable (Look table) s` fails, and just returns the empty list of results. **End Bonus 4.1**

Bonus Exercise 4.2 (10 extra credit)**.** Define an `instance` of the generic `Parser` type class for `ParseTable`. In other words, fill in the definitions for `end` and `char` in the `instance Parser Parsetable` declaration block. These `end` and `char` parser operations should do equivalent things as the ones for `NDParser`, but they return a `ParseTable` value instead. Specifically, it should be the case that, for any string `s` and character `c`, both of these equalities are `True`:

```
parseTable end s      == everyParse end s
parseTable (char c) s == everyParse (char c) s
```

End Bonus 4.2

Bonus Exercise 4.3 (20 extra credit)**.** Implement the two analysis functions

```
immediate :: ParseTable a -> [a]
lookahead :: ParseTable a -> Map (Maybe Char) (ParseTable a)
```

`immediate` searches through a `ParseTable a` and returns a list of all the values it can immediately return without looking at the input string at all. So `immediate (Done x)` gives just the one immediate result `[x]`, `immediate (Look table)` gives no immediate results, and `immediate (Fork ps)` combines all the immediate results from each `ParseTable a` in `ps`.

`lookahead` searches through a `ParseTable a` and returns a *single* `Map` which says what the parser will do after looking at just 1 character (or detecting the end of the input). `lookahead (Look table)` is just `table` itself, `lookahead (Done x)` is the `empty` map since `Done` doesn't look at the input string, and `lookahead (Fork ps)` is the combination of the `lookahead` map of each `ParseTable a` in the list `ps`. **End Bonus 4.3**

Hint 4.1. The `Data.Map` module from the `containers` package defines the `empty`map and an operation `unionsWith` for combining together several maps with these type signatures:

```
empty      :: Map k a
unionsWith :: Ord k => (a -> a -> a) -> [Map k a] -> Map k a
```

The first parameter to `unionsWith` is a function that says how to combine the `a` values associated with the same key in two different `Map`s. So for this exercise, you will want to use `empty` and `unionsWith` with these specialized types:

```
empty      :: Map (Maybe Char) (ParseTable a)
unionsWith :: (ParseTable a -> ParseTable a -> ParseTable a)
           -> [Map (Maybe Char) (ParseTable a)]
           -> Map (Maybe Char) (ParseTable a)
```

If you encounter a scenario where you have a `Fork` of several alternative parsers, and two of them both associate some `ParseTable a` response to the same `Maybe Char` input, then those two responses to the same input need to be merged into one that combines both results. Remember that `<|>` from `Alternative` can combine the ambiguous results from multiple parsers. *End Hint 4.1*

Bonus Exercise 4.4 (25 extra credit)**.** Implement the two optimization functions

```
inlineFork1 :: ParseTable a -> ParseTable a
optimize    :: ParseTable a -> ParseTable a
```

`inlineFork1` just eliminates the redundant case of a `Fork` with only one alternative. So `inlineFork1 (Fork [p])` is just `p`, and otherwise `inlineFork1` returns the same `ParseTable a` it was given.

`optimize` performs a more serious optimization over an entire `ParseTable a`. `optimize p` will rewrite `p` into the following special form:

- If `p` *does not* look at the input string at all (in other words, `lookahead p` is the empty `Map`), then `optimize p` returns a single `Fork` between `Done` parsers that return each of the `immediate` results of `p`.
- If `p` *does* look at the input string (in other words, `lookahead p` is not the empty `Map`), then `optimize p` returns a single `Fork` starting with `Done` parsers that return each of the `immediate` results of `p`, followed by the single non-empty `Look` table given by `lookahead p`.

In either case, if the result of `optimize p` would be a single-alternative `Fork [p']`, then the result should just be `p'` itself, so make sure to use `inlineFork1` on the final result of `optimize`! **End Bonus 4.4**

Hint 4.2. The function `null :: Map k a -> Bool` from `Data.Map` checks if a `Map` is empty or not. *End Hint 4.2*

Bonus Exercise 4.5 (5 extra credit)**.** Now, if you have successfully generalized your arithmetic parsers following the exercises in section 3, you can quickly switch to your implementation of parse tables and optimize the final calculator expression parser used by the main program.

First, remove the `import NDParser` from the top of the `Calculate` module and uncomment the two lines that `import GenericParser` and `ParseTable`.

Second, change the type signature of the `expr` parser in the `Calculate` module to be

```
expr :: ParseTable Double
```

and then `optimize` the right-hand side of `expr`. In otherwords, change the definition from `expr = ...` to `expr = optimize (...)`.

Third, in the `app/Main.hs` file defining the app, remove the `import NDParser` from the top of the `Main` module and uncomment the `import ParseTable` line.

Fourth, inside the definition of the `calculate` action, replace the `everyParse` function (which expects an `NDParser a`) with `parseTable` (which expects a `ParseTable a`).

These four steps entirely replace the parser engine used by the app, swapping out `NDparser` for `ParseTable` everywhere! You can now try running the calculator app with

```
> stack run
```

and check that your program can correctly calculate some arithmetic expressions! **End Bonus 4.5**