Recursive Descent Parsing:

- Recursive descent parsers can be nicely expressed in an embedded domain-specific language, built from a few primitives and composed using the connectives from Functor, Applicative, and Monad. There is one more relevant connective type class, Alternative, for failures and choices.
- The Alternative type class is used for backtracking.

Parser representation:

- Each parser can be represented as a function taking an input string, consuming a prefix of it, and giving one of:
    1. failure
    2. success, with unconsumed suffix and answer
- When the result is "success", it is important to give the unconsumed suffix rather than losing it. When liftA2 and >>= chain up two parsers, the second parser needs to see the leftover from the first parser. You can also think of a state variable for the current string to parse. Overall we are combining two effects, failure and state.
- So we use the below function type to define our parser type:

```
data Parser a = MkParser (String -> Maybe (String, a))

unParser :: Parser a -> String -> Maybe (String, a)
unParser (MkParser sf1) = sf1
```

- Each parser, there could be multiple, takes in an input string and consumes the first few characters of that input string. Then, the parser can either:
    1. Declare that this is not what I'm looking for. (Declare failure)
    2. Declare that this is what I'm looking for. The parser will give the unconsumed suffix to the next parser and give an answer. (Declare success)

  The parser is a function that takes in a string and returns the rest of the string and an answer upon success or nothing upon failure.
- We can use unParser to use our parser.
- I use Maybe because I anticipate at most one valid answer, and for simplicity I don't include error information for failures.
- It is also possible to use [] to anticipate ambiguous grammars and multiple valid answers, with the empty list for failure.
- Here is the function for using a parser. You give it an input string and it gives you an overall final answer (success or failure). It discards the final leftover as usually we aren't interested in it. If you're interested, use unParser above.

```
runParser :: Parser a -> String -> Maybe a
runParser (MkParser sf) inp = case sf inp of
                                Nothing -> Nothing
                                Just (_, a) -> Just a
                              -- OR: fmap (\(_,a) -> a) (sf inp)
```

  In the case of Nothing, we get Nothing back.
  In the case of any string and an answer, a, we get the answer, a, back.

Parsing primitives (character level):

- In this example, a basic parser reads a character and gives it to you. It fails when/if there's no character to read.

Here's the code:

```
anyChar :: Parser Char
anyChar = MkParser sf
  where
    sf "" = Nothing
    sf (c:cs) = Just (cs, c)
```

E.g.

```
*ParserLib> unParser anyChar "xyz"
Just ("yz",'x')
*ParserLib> unParser anyChar ""
Nothing
```

```
*ParserLib> runParser anyChar "xyz"
Just 'x'
*ParserLib> runParser anyChar ""
Nothing
```

- In this example, the parser is expecting a specific character and wants to read and check it. It fails if the character it's reading is not the expected character or if there's no character to read.

Here's the code:

```
char :: Char -> Parser Char
char wanted = MkParser sf
  where
    sf (c:cs) | c == wanted = Just (cs, c)
    sf _ = Nothing
```

E.g.

```
*ParserLib> unParser (char 'A') "xyz"
Nothing
*ParserLib> unParser (char 'A') "xyza"
Nothing
*ParserLib> unParser (char 'A') "xyzA"
Nothing
*ParserLib> unParser (char 'A') "Axyz"
Just ("xyz",'A')
*ParserLib> runParser (char 'A') "xyz"
Nothing
*ParserLib> runParser (char 'A') "xyzA"
Nothing
*ParserLib> runParser (char 'A') "Axyz"
Just 'A'
```

- In this example, the parser is expecting a character that satisfies a specific condition or predicate.

Here's the code:

```
satisfy :: (Char -> Bool) -> Parser Char
satisfy pred = MkParser sf
  where
    sf (c:cs) | pred c = Just (cs, c)
    sf _ = Nothing
```

If you expect a letter, you say “satisfy isAlpha” (isAlpha is from Data.Char.).
E.g.

```
*ParserLib> unParser (satisfy isAlpha) "Axyz"
Just ("xyz",'A')
*ParserLib> unParser (satisfy isAlpha) "1xyz"
Nothing
*ParserLib> unParser (satisfy isAlpha) "xyz"
Just ("yz",'x')
*ParserLib> runParser (satisfy isAlpha) "xyz"
Just 'x'
*ParserLib> runParser (satisfy isAlpha) ""
Nothing
*ParserLib> runParser (satisfy isAlpha) "1xyz"
Nothing
```

- In this example, the parser is checking that the input string is empty. So its failure/success criterion is the opposite of char's.

Here’s the code:

```
eof :: Parser ()
eof = MkParser sf
  where
    sf "" = Just ("", ())
    sf _ = Nothing
```

E.g.

```
*ParserLib> unParser eof ""
Just ("",())
*ParserLib> unParser eof " xyz"
Nothing
*ParserLib> unParser eof "xyz"
Nothing
*ParserLib> runParser eof ""
Just ()
*ParserLib> runParser eof " xyz"
Nothing
*ParserLib> runParser eof "xyz"
Nothing
```

Functor, Applicative, Monad, Alternative connectives:

- The effects of the Parser type are a combination of failure and state. Accordingly, the implementation of the Functor, Applicative, and Monad methods also combine those of Maybe and State.
  I.e. Checking for Nothing vs Just, and plumbing for state values (input strings and leftovers).

- Fmap, Applicative and Monad implementations:

```
instance Functor Parser where
  -- fmap :: (a -> b) -> Parser a -> Parser b
  fmap f (MkParser sf) = MkParser sfb
    where
      sfb inp = case sf inp of
                  Nothing -> Nothing
                  Just (rest, a) -> Just (rest, f a)
                -- OR: fmap (\(rest, a) -> (rest, f a)) (sf inp)

instance Applicative Parser where
  -- pure :: a -> Parser a
  pure a = MkParser (\inp -> Just (inp, a))

  -- liftA2 :: (a -> b -> c) -> Parser a -> Parser b -> Parser c
  -- Consider the 1st parser to be stage 1, 2nd parser stage 2.
  liftA2 op (MkParser sf1) p2 = MkParser g
    where
      g inp = case sf1 inp of
                Nothing -> Nothing
                Just (middle, a) ->
                  case unParser p2 middle of
                    Nothing -> Nothing
                    Just (rest, b) -> Just (rest, op a b)

instance Monad Parser where
  -- return :: a -> Parser a
  return = pure

  -- (>>=) :: Parser a -> (a -> Parser b) -> Parser b
  MkParser sf1 >>= k = MkParser g
    where
      g inp = case sf1 inp of
                Nothing -> Nothing
                Just (rest, a) -> unParser (k a) rest
```

- In Control.Applicative there are more utility connectives, two of which are useful for parsing.

```
(*>) :: Applicative f => f a -> f b -> f b
p *> q = liftA2 (\a b -> b) p q
-- Drop p's answer, give only q's answer.  Like (>>) but Applicative.

(<*) :: Applicative f => f a -> f b -> f a
p <* q = liftA2 (\a b -> a) p q
-- Drop q's answer, give only p's answer.
```

Example of chaining up several primitive parsers sequentially: I want a letter, then a digit, then '!'; the answer is the letter and the digit in a string, and drop the '!'.
I can use the following code to do it:

```
Ide :: Parser String
Ide = liftA2 (\x y -> [x,y]) (satisfy isAlpha) (satisfy isDigit) <* (char '!')
```

```
*ParserLib> unParser Ide "B6!A"
Just ("A","B6")
*ParserLib> unParser Ide "B36!A"
Nothing
*ParserLib> unParser Ide "2B6!A"
Nothing
*ParserLib> unParser Ide "B6a!"
Nothing
```

```
*ParserLib> unParser Ide "B6!"
Just ("","B6")
```

- There is one more type class "Alternative" in the standard library containing methods for failure and choice.
  choice is an associative binary operator, and failure is the identity element.
  Note: You need to import it from Control.Applicative.

```
class Applicative f => Alternative f where
   empty :: f a
   (<|>) :: f a -> f a -> f a
   many :: f a -> f [a]  -- has default implementation
   some :: f a -> f [a]  -- has default implementation
```

This type class was actually inspired by parsing.
The <|> operator came from the "|" in BNF, and many and some came from "0 or more times" and "1 or more times".

Here's the implementation for our Parser:

```
instance Alternative Parser where
   -- empty :: Parser a
   -- Always fail.
   -- Putting empty under if-then-else or some conditional branching makes it
     useful
   empty = MkParser (\_ -> Nothing)

   -- (<|>) :: Parser a -> Parser a -> Parser a
   -- Try the 1st one. If success, done; if failure, do the 2nd one
   MkParser sf1 <|> p2 = MkParser g
    where
      g inp = case sf1 inp of
              Nothing -> unParser p2 inp
              j -> j       -- the Just case
```

```
-- many :: Parser a -> Parser [a]
-- 0 or more times, maximum munch, collect the answers into a list.
-- Can use default implementation. And it goes as:
many p = some p <|> pure []
-- How to make sense of it: To repeat 0 or more times, first try 1 or more
-- times!  If that fails, then we know it's 0 times, and the answer is the
-- empty list.

-- some :: Parser a -> Parser [a]
-- 1 or more times, maximum munch, collect the answers into a list.
-- Can use default implementation. And it goes as:
some p = liftA2 (:) p (many p)
-- How to make sense of it: To repeat 1 or more times, do 1 time, then 0 or
-- more times!  Use liftA2 to chain up and collect answers.
```

E.g.

```
*ParserLib> unParser (many (satisfy isAlpha)) "abc012"
Just ("012","abc")
*ParserLib> unParser (many (satisfy isAlpha)) "a012"
Just ("012","a")
*ParserLib> unParser (many (satisfy isAlpha)) "012"
Just ("012","")
*ParserLib> unParser (some (satisfy isAlpha)) "abc012"
Just ("012","abc")
*ParserLib> unParser (some (satisfy isAlpha)) "a012"
Just ("012","a")
*ParserLib> unParser (some (satisfy isAlpha)) "012"
Nothing
*ParserLib> unParser (some (satisfy isAlpha)) "abc012xyz"
Just ("012xyz","abc")
*ParserLib> unParser (many (satisfy isAlpha)) "abc012xyz"
Just ("012xyz","abc")
```

Example use of <|>: I want ‘A’ or ‘B’, followed by ‘0’ or ‘1’:

```
ab01 :: Parser String
ab01 = liftA2 (\x y -> [x,y]) (char 'A' <|> char 'B') (char '0' <|> char '1')
```

E.g.

```
*ParserLib> ab01 = liftA2 (\x y -> [x,y]) (char 'A' <|> char 'B') (char '0' <|> char '1')
*ParserLib> unParser ab01 "x1"
Nothing
*ParserLib> unParser ab01 "A1"
Just ("","A1")
*ParserLib> unParser ab01 "B0"
Just ("","B0")
*ParserLib> unParser ab01 "B1"
Just ("","B1")
*ParserLib> unParser ab01 "A0"
Just ("","A0")
```

- In Control.Applicative there is also a utility connective based on Alternative. It's very handy when an ENBF rule says “0 or 1 time”.

optional :: Alternative f => f a -> f (Maybe a)
optional p = fmap Just p <|> pure Nothing

E.g.

```
*ParserLib> optional p = fmap Just p <|> pure Nothing
*ParserLib> unParser (optional (char '0')) "xyz"
Just ("xyz",Nothing)
*ParserLib> unParser (optional (char '0')) "xyz0"
Just ("xyz0",Nothing)
*ParserLib> unParser (optional (char '0')) "0"
Just ("",Just '0')
*ParserLib> unParser (optional (char '0')) "0000"
Just ("000",Just '0')
*ParserLib> unParser (optional (char '0')) "0000xyz"
Just ("000xyz",Just '0')
```

Furthermore, if we do **unParser (char '0') "0xyz"**, we get back **Just ("xyz",'0')**.
If we do **unParser (optional (char '0')) "0xyz"**, we get back **Just ("xyz",Just '0')**.

```
*ParserLib> unParser (char '0') "0xyz"
Just ("xyz",'0')
```

```
*ParserLib> unParser (optional (char '0')) "0xyz"
Just ("xyz",Just '0')
```

Parsing Primitives (lexeme/token level):

- We won't actually use the character-level primitives directly. A reason is that spaces will get into the way. Another is that we think at the token level. We only use character-level primitives to implement token-level primitives such as the ones below. Then we use connectives and token-level primitives for the grammar.
- Whitespace handling convention: Token-level primitives assume there are no leading spaces, and skip trailing spaces, so the next token primitive may assume no leading

spaces. Something else at the outermost level will have to skip initial leading spaces. This will be discussed later.

```
-- | Space or tab or newline (unix and windows).
whitespace :: Parser Char
whitespace = satisfy (\c -> c `elem` ['\t', '\n', '\r', ' '])

-- | Consume zero or more whitespaces, maximum munch.
whitespaces :: Parser String
whitespaces = many whitespace

-- | Read a natural number (non-negative integer), then skip trailing spaces.
natural :: Parser Integer
natural = fmap read (some (satisfy isDigit)) <* whitespaces
-- read :: Read a => String -> a
-- For converting string to your data type, assuming valid string.  Integer
-- is an instance of Read, and our string is valid, so we can use read.

-- | Read an identifier, then skip trailing spaces.  Disallow the listed keywords.
identifier :: [String] -> Parser String
identifier keywords =
   satisfy isAlpha
   >>= \c -> many (satisfy isAlphaNum)
   >>= \cs -> whitespaces
   >> let str = c:cs
   in if str `elem` keywords then empty else return str

-- | Read the wanted keyword, then skip trailing spaces.
keyword :: String -> Parser String
keyword wanted =
   satisfy isAlpha
   >>= \c -> many (satisfy isAlphaNum)
   >>= \cs -> whitespaces
   *> if c:cs == wanted then return wanted else empty
```