Mark Karpov

Megaparsec tutorial

Published on February 23, 2019, last updated October 30, 2021

This is the Megaparsec tutorial which originally was written as a chapter for the Intermediate Haskell book. Due to lack of progress with the book in the last year, other authors agreed to let me publish the text as a standalone tutorial so that people can benefit at least from this part of our work.

Japanese translation, Chinese translation.

- `ParsecT` and `Parsec` monads
- Character and binary streams
- Monadic and applicative syntax
- Forcing consumption of input with `eof`
- Working with alternatives
- Controlling backtracking with `try`
- Debugging parsers
- Labeling and hiding things
- Running a parser
- The `MonadParsec` type class
- Lexing
  - White space
  - Char and string literals
  - Numbers
- `notFollowedBy` and `lookAhead`
- Parsing expressions
- Indentation-sensitive parsing
  - `nonIndented` and `indentBlock`
  - Parsing a simple indented list
  - Nested indented list
  - Adding line folds
- Writing efficient parsers
- Parse errors
  - Parse error definitions
  - How to signal a parse error
  - Displaying parse errors
  - Catching parse errors in a running parser
  - Controlling location of parse errors
  - Reporting multiple parse errors
- Testing Megaparsec parsers

- Working with custom input streams

The toy parser combinators developed in chapter “An Example: Writing Your Own Parser Combinators” are not suitable for real-world use, so let’s continue by taking a look at the libraries in the Haskell ecosystem that solve the same problem, and note various trade-offs they make:

- `parsec` has been the “default” parsing library in Haskell for a long time. The library is said to be focused on quality of error messages. It however does not have good test coverage and is currently in maintenance mode.
- `attoparsec` is a robust, fast parsing library with focus on performance. It is the only library from this list that has full support for incremental parsing. Its downsides are poor quality of error messages, inability to be used as a monad transformer, and limited set of types that can be used as input stream.
- `trifecta` features good error messages but is under-documented and hard to figure out. It can parse `String` and `ByteString` out-of-the-box, but not `Text`.
- `megaparsec` is a fork of `parsec` that has been actively developed in the last few years. The current version tries to find a nice balance between speed, flexibility, and quality of parse errors. As an unofficial successor of `parsec`, it stays conventional and immediately familiar for users who have used that library or who have read `parsec` tutorials.

It would be impractical to try to cover all these libraries, and so we will focus on `megaparsec`. More precisely, we are going to cover the version 9, which by the time this book is published will probably have replaced the older versions almost everywhere.

`ParsecT` and `Parsec` monads

`ParsecT` is the main parser monad transformer and the central data type in `megaparsec`. `ParsecT e s m a` is parametrized like this:

- `e` is the type of custom component of error messages. If we do not want anything custom (and for now we do not), we just use `Void` from the `Data.Void` module.
- `s` is the type of input stream. `megaparsec` works out-of-the-box with `String`, strict and lazy `Text`, and strict and lazy `ByteString`s. It is also possible to work with custom input streams.
- `m` is the inner monad of the `ParsecT` monad transformer.
- `a` is the monadic value, result of parsing.

Since most of the time `m` is nothing but `Identity`, the `Parsec` type synonym is quite useful:

```
type Parsec e s a = ParsecT e s Identity a
```

`Parsec` is simply the non-transformer version of `ParsecT`.

We can also draw an analogy between the monad transformers in `megaparsec` and MTL monad transformers and classes. Indeed, there is also the `MonadParsec` type class which is similar in its purpose to type classes such as `MonadState` and `MonadReader`. We will return to `MonadParsec` later and discuss it in more details.

Speaking of type synonyms, the best way to start writing parser with `megaparsec` is to define a custom type synonym for your parser. This is a good idea for two reasons:

- It will be easier to add top level signatures like `Parser Int` where `Parser` is your parsing monad. Without the signatures, things like `e` will often be ambiguous—it is the flip side of the polymorphic API of the library.
- Working with concrete types with all type variables fixed helps GHC optimize a lot better. GHC cannot do much in terms of optimization if your parsers stay polymorphic. Although `megaparsec` API is polymorphic, it is expected that end user will stick to a concrete type of parsing monad, so inlining and the fact that most functions have their definition dumped into so-called *interface files* will allow GHC produce very efficient non-polymorphic code.

Let's define a type synonym (typically called `Parser`) like this:

```
type Parser = Parsec Void Text
--                   ^    ^
--                   |    |
-- Custom error component Type of input stream
```

Until we start dealing with custom parsing errors, when you see `Parser` in the chapter, assume this type.

Character and binary streams

It has been said that `megaparsec` can work with five types of input stream out-of-the-box: `String`, strict and lazy `Text`, and strict and lazy `ByteString`s. This is possible because the library makes these types instances of the `Stream` type class which abstracts the functionality that every data type should support to be used as input to a `megaparsec` parser.

A simplified version of `Stream` could look like this:

```
class Stream s where
  type Token  s :: *
  type Tokens s :: *
  take1_ :: s -> Maybe (Token s, s) -- aka uncons
  tokensToChunk :: Proxy s -> [Token s] -> Tokens s
```

The real definition of `Stream` has more methods, but knowing about them is not necessary for using the library.

Note that the type class has two type functions associated with it:

- `Token s` for stream `s` is the type of single token. Common examples are `Char` and `Word8`, although it may be something else for custom streams.
- `Tokens s` for stream `s` is the type of a "chunk" of stream. The concept of *chunk* was only introduced for performance reasons. Indeed, it is often possible to have a more efficient representation of part of a stream which is isomorphic to list of tokens `[Token s]`. For example, input stream of the type `Text` has `Tokens s ~ Text`: chunk of `Text` is just `Text`. Although the type equality `Tokens s ~ s` often holds, `Tokens s` and `s` may differ for custom streams, and thus we separate these types in `megaparsec`.

We can put all the default input streams into a single table like this:

`s`	`Token s`	`Tokens s`
`String`	`Char`	`String`
strict `Text`	`Char`	strict `Text`
lazy `Text`	`Char`	lazy `Text`
strict `ByteString`	`Word8`	strict `ByteString`
lazy `ByteString`	`Word8`	lazy `ByteString`

It is important to get used to the `Token` and `Tokens` type functions because they are ubiquitous in the types of `megaparsec` API.

You may have noticed that if we group all default input streams by token type, we will get two groups:

- character streams, for which `Token s ~ Char`: `String` and strict/lazy `Text`,
- binary streams, for which `Token s ~ Word8`: strict and lazy `ByteString`s.

It turns out that with `megaparsec` it is not necessary to code the same parsers for every type of input stream (this is the case, for example, with the `attoparsec` library), but still we must have different code for different token types:

- to get common combinators for character streams, import the `Text.Megaparsec.Char` module;
- to get the same for binary streams, import `Text.Megaparsec.Byte`.

These modules contain two similar sets of helper parsers such as:

Name	`Text.Megaparsec.Char`	`Text.Megaparsec.Byte`
`newline`	`(MonadParsec e s m, Token s ~ Char) => m (Token s)`	`(MonadParsec e s m, Token s ~ Word8) => m (Token s)`
`eol`	`(MonadParsec e s m, Token s ~ Char) => m (Tokens s)`	`(MonadParsec e s m, Token s ~ Word8) => m (Tokens s)`

Let's introduce a couple of primitives on which the modules are built, so we understand the tools we are going to use.

The first primitive is called `token`, and correspondingly it allows us to parse a `Token s`:

```
token :: MonadParsec e s m
  => (Token s -> Maybe a)
    -- ^ Matching function for the token to parse
  -> Set (ErrorItem (Token s))
     -- ^ Expected items (in case of an error)
  -> m a
```

The first argument of `token` is the matching function for the token to parse. If the function returns something in `Just`, that value becomes the result of parsing. `Nothing` indicates that the parser does not accept this token and so the primitive fails.

The second argument is a `Set` (from the `containers` package) that contains all expected `ErrorItem`s to be displayed to the user in case of failure. We will explore the `ErrorItem` type in details when we will be discussing parse errors.

To better understand how `token` works, let's see some definitions from the `Text.Megaparsec` module which contains, among other things, some combinators that work with all types of input stream. `satisfy` is a fairly common combinator, we give it a predicate that returns `True` for tokens we want to match and it gives us a parser back:

```
satisfy :: MonadParsec e s m
  => (Token s -> Bool) -- ^ Predicate to apply
  -> m (Token s)
satisfy f = token testToken Set.empty
  where
    testToken x = if f x then Just x else Nothing
```

The job of `testToken` is to turn the `f` function which returns `Bool` into a function that returns `Maybe (Token s)` that `token` expects. With `satisfy` we do not know exact sequence of tokens that we expect to match, thus we pass `Set.empty` as the second argument.

`satisfy` should look understandable, let's see how it works. To play with a parser we need a helper function that would run it. For testing in GHCi `megaparsec` provides `parseTest`.

First, let's start GHCi and import some modules:

```
λ> import Text.Megaparsec
λ> import Text.Megaparsec.Char
λ> import Data.Text (Text)
λ> import Data.Void
```

We add the `Parser` type synonym that we will use to resolve ambiguity in the type of the parsers:

```
λ> type Parser = Parsec Void Text
```

We also need to enable the `OverloadedStrings` language extension so we can use string literals as `Text` values:

```
λ> :set -XOverloadedStrings

λ> parseTest (satisfy (== 'a') :: Parser Char) ""
1:1:
  |
1 | <empty line>
  | ^
unexpected end of input

λ> parseTest (satisfy (== 'a') :: Parser Char) "a"
'a'

λ> parseTest (satisfy (== 'a') :: Parser Char) "b"
1:1:
  |
1 | b
  | ^
unexpected 'b'

λ> parseTest (satisfy (> 'c') :: Parser Char) "a"
1:1:
  |
1 | a
  | ^
unexpected 'a'

λ> parseTest (satisfy (> 'c') :: Parser Char) "d"
'd'
```

The `:: Parser Char` annotation is necessary because `satisfy` by itself is polymorphic, so `parseTest` cannot know what to use in place of `e` and `s` in `MonadParsec e s m (m` is

assumed to be `Identity` with these helpers). If we worked with a pre-existing parser which had a type signature, the explicit clarification of parser type would be unnecessary.

That seems to work all right. The problem with `satisfy` is that it does not say what is expected when it fails, because we cannot analyze the function which the caller of `satisfy` provides. There are other combinators that are less general, but they can generate more helpful error messages. For example `single` (with type-constrained synonyms called `char` in `Text.Megaparsec.Byte` and `Text.Megaparsec.Char`) which matches a specific token value:

```
single :: MonadParsec e s m
  => Token s           -- ^ Token to match
  -> m (Token s)
single t = token testToken expected
  where
    testToken x = if x == t then Just x else Nothing
    expected    = Set.singleton (Tokens (t:|[]))
```

The `Tokens` data type constructor has nothing in common with the type function `Tokens` that we have discussed previously. In fact, `Tokens` is one of constructors of `ErrorItem` and it is used to specify concrete sequence of tokens we expected to match.

```
λ> parseTest (char 'a' :: Parser Char) "b"
1:1:
  |
1 | b
  | ^
unexpected 'b'
expecting 'a'

λ> parseTest (char 'a' :: Parser Char) "a"
'a'
```

We can now define `newline` from the table above:

```
newline :: (MonadParsec e s m, Token s ~ Char) => m (Token s)
newline = single '\n'
```

The second primitive is called `tokens` and it allows us to parse `Tokens s`, that is, it can be used to match a fixed chunk of input:

```
tokens :: MonadParsec e s m
  => (Tokens s -> Tokens s -> Bool)
    -- ^ Predicate to check equality of chunks
  -> Tokens s
    -- ^ Chunk of input to match against
  -> m (Tokens s)
```

There are also two parsers defined in terms of `tokens`:

```
-- from "Text.Megaparsec":
chunk :: MonadParsec e s m
  => Tokens s
  -> m (Tokens s)
chunk = tokens (==)

-- from "Text.Megaparsec.Char" and "Text.Megaparsec.Byte":
string' :: (MonadParsec e s m, Data.CaseInsensitive.FoldCase (Tokens s))
  => Tokens s
  -> m (Tokens s)
string' = tokens ((==) `on` Data.CaseInsensitive.mk)
```

They match fixed chunks of input, `chunk` (which has type-constrained synonyms called `string` in `Text.Megaparsec.Byte` and `Text.Megaparsec.Char`) case-sensitively, while `string'` case-insensitively. For case-insensitive matching the `case-insensitive` package is used, thus the `FoldCase` constraint.

Let's try to use the new combinators:

```
λ> parseTest (string "foo" :: Parser Text) "foo"
"foo"

λ> parseTest (string "foo" :: Parser Text) "bar"
1:1:
  |
1 | bar
  | ^
unexpected "bar"
expecting "foo"

λ> parseTest (string' "foo" :: Parser Text) "FOO"
"FOO"

λ> parseTest (string' "foo" :: Parser Text) "FoO"
"FoO"

λ> parseTest (string' "foo" :: Parser Text) "FoZ"
1:1:
  |
1 | FoZ
  | ^
unexpected "FoZ"
expecting "foo"
```

OK, we can match a single token and a chunk of input. The next step is to learn how to combine the building blocks to write more interesting parsers.

Monadic and applicative syntax

The simplest way to combine parsers is to execute them in succession. `ParsecT` and `Parsec` are monads, and monadic bind is exactly what we use for sequencing our parsers:

```
mySequence :: Parser (Char, Char, Char)
mySequence = do
  a <- char 'a'
  b <- char 'b'
  c <- char 'c'
  return (a, b, c)
```

We can run it to check that everything works as expected:

```
λ> parseTest mySequence "abc"
('a','b','c')

λ> parseTest mySequence "bcd"
1:1:
  |
1 | bcd
  | ^
unexpected 'b'
expecting 'a'

λ> parseTest mySequence "adc"
1:2:
  |
1 | adc
  |  ^
unexpected 'd'
expecting 'b'
```

An alternative syntax for sequential execution is possible if we remember that every monad is also an applicative functor, and so we can use applicative syntax:

```
mySequence :: Parser (Char, Char, Char)
mySequence =
  (,,) <$> char 'a'
       <*> char 'b'
       <*> char 'c'
```

The second version works just like the first. Which style to use is often a matter of taste. Monadic style is arguably more verbose and sometimes clearer, while applicative style is often more concise. That said, monadic style is of course more powerful because monads are more powerful than applicative functors.

Forcing consumption of input with `eof`

`Applicative` is often powerful enough to allow doing quite interesting things. Equipped with an associative operator which has identity, we get a monoid on applicative functors

expressed in Haskell via the `Alternative` type class. The parser-combinators package provides quite a few abstract combinators built on the concepts of `Applicative` and `Alternative`. The `Text.Megaparsec` module re-exports them from `Control.Applicative.Combinators`.

One of the most common combinators is called `many`. It allows us to run a given parser *zero* or more times:

```
λ> parseTest (many (char 'a') :: Parser [Char]) "aaa"
"aaa"

λ> parseTest (many (char 'a') :: Parser [Char]) "aabbb"
"aa"
```

The second result may be a bit surprising. The parser consumed `a`s that matched, but stopped after that. Well, we did not say what we want to do after `many (char 'a')`!

Most of the time we want to actually force parser to consume entire input, and report parse errors instead of being shy and stopping silently. This is done by demanding that we reach the end of input. Happily, although the end of input is nothing but a concept, there is a primitive called `eof :: MonadParsec e s m => m ()` that does not ever consume anything and only succeeds at the end of input. Let's add it to our parser and try again:

```
λ> parseTest (many (char 'a') <* eof :: Parser [Char]) "aabbb"
1:3:
  |
1 | aabbb
  |   ^
unexpected 'b'
expecting 'a' or end of input
```

We did not say anything about `b`s in our parser, and they are certainly unexpected.

Working with alternatives

From now on we will be developing a real, useful parser that can parse URIs of the following form:

```
scheme:[//[user:password@]host[:port]][/]path[?query][#fragment]
```

We should remember that things in square brackets `[]` are optional, they may or may not appear in a valid URI. `[]` may be even nested to express a possibility inside another possibility. We will handle all of this [1].

Let's start with `scheme`. We will accept only schemes that are known to us, such as: `data`, `file`, `ftp`, `http`, `https`, `irc`, and `mailto`.

To match a fixed sequence of characters we use `string`. To express a choice, we use the `(<|>)` method from the `Alternative` type class. So we can write:

```
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE RecordWildCards   #-}

module Main (main) where

import Control.Applicative
import Control.Monad
import Data.Text (Text)
import Data.Void
import Text.Megaparsec hiding (State)
import Text.Megaparsec.Char
import qualified Data.Text as T
import qualified Text.Megaparsec.Char.Lexer as L

type Parser = Parsec Void Text

pScheme :: Parser Text
pScheme = string "data"
  <|> string "file"
  <|> string "ftp"
  <|> string "http"
  <|> string "https"
  <|> string "irc"
  <|> string "mailto"
```

Let's try it:

```
λ> parseTest pScheme ""
1:1:
  |
1 | <empty line>
  | ^
unexpected end of input
expecting "data", "file", "ftp", "http", "https", "irc", or "mailto"

λ> parseTest pScheme "dat"
1:1:
  |
1 | dat
  | ^
unexpected "dat"
expecting "data", "file", "ftp", "http", "https", "irc", or "mailto"

λ> parseTest pScheme "file"
"file"
```

```
λ> parseTest pScheme "irc"
"irc"
```

Looks good, but the defintion of `pScheme` is a bit repetitive. There is a way to write `pScheme` with the `choice` combinator:

```
pScheme :: Parser Text
pScheme = choice
  [ string "data"
  , string "file"
  , string "ftp"
  , string "http"
  , string "https"
  , string "irc"
  , string "mailto" ]
```

`choice` is just a synonym for `asum` —an operation that folds a list putting `(<|>)` between its elements, so the two definitions of `pScheme` are actually the same, although the one which uses `choice` may look a bit nicer.

After the scheme, there should be a colon `:`. Recall that to require something to go after something else, we use monadic bind or `do`-notation:

```
data Uri = Uri
  { uriScheme :: Text
  } deriving (Eq, Show)

pUri :: Parser Uri
pUri = do
  r <- pScheme
  _ <- char ':'
  return (Uri r)
```

If we try to run `pUri`, we will see that it requires `:` to follow the scheme name now:

```
λ> parseTest pUri "irc"
1:4:
  |
1 | irc
  |    ^
unexpected end of input
expecting ':'

λ> parseTest pUri "irc:"
Uri {uriScheme = "irc"}
```

We are not done with the scheme parsing though. A good Haskell programmer tries to define types in such a way so that incorrect data cannot be represented. Not every `Text` value is a valid scheme. Let's define a data type to represent schemes and make our `pScheme` parser return value of that type:

```haskell
data Scheme
  = SchemeData
  | SchemeFile
  | SchemeFtp
  | SchemeHttp
  | SchemeHttps
  | SchemeIrc
  | SchemeMailto
  deriving (Eq, Show)

pScheme :: Parser Scheme
pScheme = choice
  [ SchemeData   <$ string "data"
  , SchemeFile   <$ string "file"
  , SchemeFtp    <$ string "ftp"
  , SchemeHttp   <$ string "http"
  , SchemeHttps  <$ string "https"
  , SchemeIrc    <$ string "irc"
  , SchemeMailto <$ string "mailto" ]

data Uri = Uri
  { uriScheme :: Scheme
  } deriving (Eq, Show)
```

The `(<$)` operator just puts the value on its left-hand side into a functorial context replacing whatever is there at the moment. `a <$ f` is the same as `const a <$> f`, but can be more efficient for some functors.

Let's continue playing with our parser:

```
λ> parseTest pUri "https:"
1:5:
  |
1 | https:
  |     ^
unexpected 's'
expecting ':'
```

Hmm, `https` should be a valid scheme. Can you figure out what went wrong? The parser tries the alternatives one by one, and `http` matches, so it does not go further to try `https`. The solution is to put the `SchemeHttps <$ string "https"` line before the `SchemeHttp <$ string "http"` line. Remember: *with alternatives, order matters!*

Now `pUri` works correctly:

```
λ> parseTest pUri "http:"
Uri {uriScheme = SchemeHttp}

λ> parseTest pUri "https:"
```

```
Uri {uriScheme = SchemeHttps}

λ> parseTest pUri "mailto:"
Uri {uriScheme = SchemeMailto}

λ> parseTest pUri "foo:"
1:1:
  |
1 | foo:
  | ^
unexpected "foo:"
expecting "data", "file", "ftp", "http", "https", "irc", or "mailto"
```

Controlling backtracking with `try`

The next part to handle is `[//[user:password@]host[:port]]` —the authority. Here we have nested optional parts, let us update the `Uri` type to reflect this:

```haskell
data Uri = Uri
  { uriScheme    :: Scheme
  , uriAuthority :: Maybe Authority
  } deriving (Eq, Show)

data Authority = Authority
  { authUser :: Maybe (Text, Text) -- (user, password)
  , authHost :: Text
  , authPort :: Maybe Int
  } deriving (Eq, Show)
```

Now we need to discuss an important concept called *backtracking*. Backtracking is a way to travel back in time "un-consuming" input in the process. This is important primarily with branching. Here is an example:

```haskell
alternatives :: Parser (Char, Char)
alternatives = foo <|> bar
  where
    foo = (,) <$> char 'a' <*> char 'b'
    bar = (,) <$> char 'a' <*> char 'c'
```

Looks reasonable, let's try it:

```
λ> parseTest alternatives "ab"
('a','b')

λ> parseTest alternatives "ac"
1:2:
  |
1 | ac
  |  ^
```

```
unexpected 'c'
expecting 'b'
```

What happens here is that `char 'a'` part of `foo` (which is tried first) succeeded and consumed an `a` from the input stream. `char 'b'` then failed to match against `'c'` and so we ended up with this error. An important detail here is that `(<|>)` did not even try `bar` because `foo` has consumed some input!

This is done for performance reasons and because it would make no sense to run `bar` feeding it leftovers of `foo` anyway. `bar` wants to be run from the same point in the input stream as `foo`. `megaparsec` does not go back automatically, unlike for example `attoparsec` or the toy combinators from the previous chapter, so we must use a primitive called `try` to express our wish to backtrack explicitly. `try p` makes it so that if `p` fails consuming input, `try p` fails as if no input has been consumed (in fact, it backtracks the entire parser state). This allows `(<|>)` to try its right-hand alternative:

```
alternatives :: Parser (Char, Char)
alternatives = try foo <|> bar
  where
    foo = (,) <$> char 'a' <*> char 'b'
    bar = (,) <$> char 'a' <*> char 'c'

λ> parseTest alternatives "ac"
('a','c')
```

All primitives that actually consume input (there are also primitives that alter behavior of existing parsers, such as `try` itself) are "atomic" in terms of input consumption. This means that if they fail, they backtrack automatically, so there is no way they can consume some input and then fail halfway through. This is why `pScheme` with its list of alternatives works: `string` is defined on top of `tokens` and `tokens` is a primitive. We either match the entire string with `string` or we fail without consuming input stream at all.

Back to parsing URIs, `(<|>)` can be used to build a handy combinator called `optional`:

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

If `p` in `optional p` matches, we get its result in `Just`, otherwise `Nothing` is returned. Just what we want! There is no need to define `optional`, `Text.Megaparsec` re-exports this combinator for us. We can now use it in `pUri`:

```
pUri :: Parser Uri
pUri = do
  uriScheme <- pScheme
  void (char ':')
  uriAuthority <- optional . try $ do           -- (1)
    void (string "//")
    authUser <- optional . try $ do             -- (2)
```

```
      user <- T.pack <$> some alphaNumChar       -- (3)
      void (char ':')
      password <- T.pack <$> some alphaNumChar
      void (char '@')
      return (user, password)
    authHost <- T.pack <$> some (alphaNumChar <|> char '.')
    authPort <- optional (char ':' *> L.decimal) -- (4)
    return Authority {..}                        -- (5)
  return Uri {..}                                -- (6)
```

I took the liberty of accepting any alpha-numeric sequences of characters as username and password, and made similarly arbitrary simplifications in the format of the host.

Some important points here:

- In (1) and (2) we need to wrap the argument of `optional` with `try` because it is a composite parser, not a primitive.
- (3) `some` is just like `many`, but demands that its argument parser matches at least once: `some p = (:) <$> p <*> many p`.
- (4) Do not use `try` unless necessary! Here if `char ':'` succeeds (which is by itself built on top of `token`, so it does not need a `try`), we know for sure that port must follow after it, so we just demand a decimal number with `L.decimal`. After matching `:`, we are committed and do not need a way to go back.
- In (5) and (6) we assemble `Authority` and `Uri` values using the `RecordWildCards` language extension.
- `void :: Functor f => f a -> f ()` is used to explicitly discard the result to parsing, without it we would get warnings about unused values from GHC.

Play with `pUri` in GHCi and see for yourself that it works:

```
λ> parseTest (pUri <* eof) "https://mark:secret@example.com"
Uri
  { uriScheme = SchemeHttps
  , uriAuthority = Just (Authority
    { authUser = Just ("mark","secret")
    , authHost = "example.com"
    , authPort = Nothing } ) }

λ> parseTest (pUri <* eof) "https://mark:secret@example.com:123"
Uri
  { uriScheme = SchemeHttps
  , uriAuthority = Just (Authority
    { authUser = Just ("mark","secret")
    , authHost = "example.com"
    , authPort = Just 123 } ) }
```

```
λ> parseTest (pUri <* eof) "https://example.com:123"
Uri
  { uriScheme = SchemeHttps
  , uriAuthority = Just (Authority
    { authUser = Nothing
    , authHost = "example.com"
    , authPort = Just 123 } ) }

λ> parseTest (pUri <* eof) "https://mark@example.com:123"
1:13:
  |
1 | https://mark@example.com:123
  |             ^
unexpected '@'
expecting '.', ':', alphanumeric character, or end of input
```

Debugging parsers

However, you may find that there is a problem:

```
λ> parseTest (pUri <* eof) "https://mark:@example.com"
1:7:
  |
1 | https://mark:@example.com
  |       ^
unexpected '/'
expecting end of input
```

The parse error could be better! What to do? The easiest way to figure out what is going on is to use the built-in `dbg` helper from the `Text.Megaparsec.Debug` module:

```
dbg :: (VisualStream s, ShowToken (Token s), ShowErrorComponent e, Show a)
  => String            -- ^ Debugging label
  -> ParsecT e s m a   -- ^ Parser to debug
  -> ParsecT e s m a   -- ^ Parser that prints debugging messages
```

The `VisualStream` type class is defined for input streams that can be printed out on the screen in readable form. We will not dwell on it here.

Let's use it in `pUri`:

```
pUri :: Parser Uri
pUri = do
  uriScheme <- dbg "scheme" pScheme
  void (char ':')
  uriAuthority <- dbg "auth" . optional . try $ do
    void (string "//")
    authUser <- dbg "user" . optional . try $ do
      user <- T.pack <$> some alphaNumChar
```

```
      void (char ':')
      password <- T.pack <$> some alphaNumChar
      void (char '@')
      return (user, password)
    authHost <- T.pack <$> dbg "host" (some (alphaNumChar <|> char '.'))
    authPort <- dbg "port" $ optional (char ':' *> L.decimal)
    return Authority {..}
  return Uri {..}
```

Then let's try running `pUri` on that unfortunate input again:

```
λ> parseTest (pUri <* eof) "https://mark:@example.com"
scheme> IN: "https://mark:@example.com"
scheme> MATCH (COK): "https"
scheme> VALUE: SchemeHttps

user> IN: "mark:@example.com"
user> MATCH (EOK): <EMPTY>
user> VALUE: Nothing

host> IN: "mark:@example.com"
host> MATCH (COK): "mark"
host> VALUE: "mark"

port> IN: ":@example.com"
port> MATCH (CERR): ':'
port> ERROR:
port> 1:14:
port> unexpected '@'
port> expecting integer

auth> IN: "//mark:@example.com"
auth> MATCH (EOK): <EMPTY>
auth> VALUE: Nothing

1:7:
  |
1 | https://mark:@example.com
  |       ^
unexpected '/'
expecting end of input
```

We can see what exactly is going on inside `megaparsec` now:

- `scheme` matches successfully.
- `user` fails: although there is a username in place `mark`, there is no password after the column `:` (we demand that the password is not empty here). We fail and thanks to `try`, backtrack.

- `host` starts from the same point as `user` and tries now to interpret input as hostname. We can see that it succeeds and returns `mark` as host name.
- There may be a port number after host, so `port` gets its chance now. It sees `:`, but after that there is no integer, so `port` fails as well.
- The whole `auth` parser thus fails (`port` is inside of `auth` and it has failed).
- The `auth` parser returns `Nothing` because it could not parse anything. Now `eof` demands that we have reached the end of input, but it is not the case, so we get the final error message.

What to do? This is an example of a situation when using `try` enclosing large portions of code may make parse errors worse. Let us take another look at the syntax we want to parse:

```
scheme:[//[user:password@]host[:port]][/]path[?query][#fragment]
```

What are we looking for? Something that would allow us to commit to certain branch of parsing. Just like with port where when we see column `:` we are sure port number must follow. If you look carefully, you will see that the double slash `//` is the sign that we have the authority part in our URI. Since we match `//` with an "atomic" parser (`string`), matching on it backtracks automatically, and after we have matched `//`, we can be sure to demand the authority part. Let us remove the first `try` from `pUri`:

```
pUri :: Parser Uri
pUri = do
  uriScheme <- pScheme
  void (char ':')
  uriAuthority <- optional $ do -- removed 'try' on this line
    void (string "//")
    authUser <- optional . try $ do
      user <- T.pack <$> some alphaNumChar
      void (char ':')
      password <- T.pack <$> some alphaNumChar
      void (char '@')
      return (user, password)
    authHost <- T.pack <$> some (alphaNumChar <|> char '.')
    authPort <- optional (char ':' *> L.decimal)
    return Authority {..}
  return Uri {..}
```

Now we get a nicer parse error:

```
λ> parseTest (pUri <* eof) "https://mark:@example.com"
1:14:
  |
1 | https://mark:@example.com
  |              ^
```

```
unexpected '@'
expecting integer
```

Although it is still a bit misleading, but well, that is a tricky example I have picked. Lots of `optional`s.

Labeling and hiding things

Sometimes the list of expected items may get rather long. Remember what we get when we try to use a non-recognized scheme?

```
λ> parseTest (pUri <* eof) "foo://example.com"
1:1:
  |
1 | foo://example.com
  | ^
unexpected "foo://"
expecting "data", "file", "ftp", "http", "https", "irc", or "mailto"
```

`megaparsec` provides a way to override expected items with something custom, typically called a *label*. This is done with the help of the `label` primitive (which has a synonym in the form of the `(<?>)` operator):

```
pUri :: Parser Uri
pUri = do
  uriScheme <- pScheme <?> "valid scheme"
  -- the rest stays the same
```

```
λ> parseTest (pUri <* eof) "foo://example.com"
1:1:
  |
1 | foo://example.com
  | ^
unexpected "foo://"
expecting valid scheme
```

We can go on and add more labels to make errors messages more human-readable:

```
pUri :: Parser Uri
pUri = do
  uriScheme <- pScheme <?> "valid scheme"
  void (char ':')
  uriAuthority <- optional $ do
    void (string "//")
    authUser <- optional . try $ do
      user <- T.pack <$> some alphaNumChar <?> "username"
      void (char ':')
      password <- T.pack <$> some alphaNumChar <?> "password"
      void (char '@')
      return (user, password)
```

```
    authHost <- T.pack <$> some (alphaNumChar <|> char '.') <?> "hostname"
    authPort <- optional (char ':' *> label "port number" L.decimal)
    return Authority {..}
  return Uri {..}
```

For example:

```
λ> parseTest (pUri <* eof) "https://mark:@example.com"
1:14:
  |
1 | https://mark:@example.com
  |              ^
unexpected '@'
expecting port number
```

Another primitive is called `hidden`. If `label` renames things, `hidden` just removes them altogether. Compare:

```
λ> parseTest (many (char 'a') >> many (char 'b') >> eof :: Parser ()) "d"
1:1:
  |
1 | d
  | ^
unexpected 'd'
expecting 'a', 'b', or end of input

λ> parseTest (many (char 'a') >> hidden (many (char 'b')) >> eof :: Parser ()) "d"
1:1:
  |
1 | d
  | ^
unexpected 'd'
expecting 'a' or end of input
```

`hidden` is useful when it is desirable to make error messages less noisy. For example, when parsing a programming language it is a good idea to drop “expecting white space” messages because usually there may be white space after each token anyway.

EXERCISE: Finishing the `pUri` parser is left as an exercise for the reader, now that all the tools that are necessary for this have been explained.

Running a parser

We explored in details how to construct parsers, but we haven’t inspected the functions that allow us to run them, except for `parseTest`.

Traditionally, the “default” function to run a parser from your program has been `parse`. But `parse` is actually a synonym for `runParser`:

```
runParser
  :: Parsec e s a -- ^ Parser to run
  -> String       -- ^ Name of source file
  -> s            -- ^ Input for parser
  -> Either (ParseErrorBundle s e) a
```

The second argument is just a file name which will be included in the generated parse errors, `megaparsec` is not going to read anything from that file, because the actual input comes as the third argument of the function.

`runParser` allows us to run the `Parsec` monad which, as we already know, is the non-transformer version of `ParsecT`:

```
type Parsec e s = ParsecT e s Identity
```

`runParser` has 3 siblings: `runParser'`, `runParserT`, and `runParserT'`. The versions with the `T` suffix run `ParsecT` monad transformer, and the "prime" versions take and return parser state. Let's put all the functions into a table:

Arguments	Runs `Parsec`	Runs `ParsecT`
Input and file name	`runParser`	`runParserT`
Custom initial state	`runParser'`	`runParserT'`

Custom initial state may be necessary if you e.g. want to set tab width to some non-standard value (the default value is `8`). As an example, here is the type signature of `runParser'`:

```
runParser'
  :: Parsec e s a -- ^ Parser to run
  -> State s      -- ^ Initial state
  -> (State s, Either (ParseErrorBundle s e) a)
```

Modifying `State` manually is advanced usage of the library, and we are not going to describe it here.

If you wonder what is `ParseErrorBundle`, we'll discuss it in one of the following sections.

The `MonadParsec` type class

All tools in `megaparsec` work with any instance of the `MonadParsec` type class. The type class abstracts *primitive combinators*—the elementary building blocks of all `megaparsec` parsers, combinators that cannot be expressed via other combinators.

Having primitive combinators in a type class allows the principal concrete monad transformer of `megaparsec` `ParsecT` to be wrapped in the familiar transformers of the MTL family achieving different interactions between layers of a monadic stack. To better

understand the motivation, recall that the order of layers in a monadic stack matters. If we combine `ReaderT` and `State` like this:

```
type MyStack a = ReaderT MyContext (State MyState) a
```

the outer layer, `ReaderT` cannot inspect the internal structure of the underlying `m` layer. The `Monad` instance for `ReaderT` describes the binding strategy:

```
newtype ReaderT r m a = ReaderT { runReaderT :: r -> m a }

instance Monad m => Monad (ReaderT r m) where
  m >>= k = ReaderT $ \r -> do
    a <- runReaderT m r
    runReaderT (k a) r
```

In fact, the only thing that we know about `m` is that it is an instance of `Monad` and so the state of `m` can only be passed to `k` via monadic bind. That is what we typically want from `(>>=)` of `ReaderT` anyway.

The `(<|>)` method of the `Alternative` type class works differently—it "splits" state and the two branches of parsing do not contact anymore, so we get *backtracking state* in the sense that if the first branch is discarded changes to its state are also discarded and cannot influence the second branch (we "backtrack" the state when the first branch fails).

To illustrate, let us see the definition of `Alternative` for `ReaderT`:

```
instance Alternative m => Alternative (ReaderT r m) where
  empty = liftReaderT empty
  ReaderT m <|> ReaderT n = ReaderT $ \r -> m r <|> n r
```

This all is very nice, because `ReaderT` is a "stateless" monad transformer and it is easy to delegate the actual work to the inner monad (the `Alternative` instance of `m` comes in handy here) without needing to combine monadic state associated with `ReaderT` itself (it has none).

Now let's take a look at `State`. Since `State s a` is just a type synonym for `StateT s Identity a`, we should look at the `Alternative` instance for `StateT s m` itself:

```
instance (Functor m, Alternative m) => Alternative (StateT s m) where
  empty = StateT $ \_ -> empty
  StateT m <|> StateT n = StateT $ \s -> m s <|> n s
```

Here we can see the splitting of state `s`, just like we saw sharing of the reader context `r`. There is a difference though, because the expressions `m s` and `n s` produce stateful results: together with monadic value, they return the new state in a tuple. Here we either go with `m s` or with `n s`, naturally achieving backtracking.

What about `ParsecT`? Let us consider now putting `State` inside `ParsecT` like this:

```
type MyStack a = ParsecT Void Text (State MyState) a
```

`ParsecT` is more complex than `ReaderT` and its implementation of `(<|>)` has to do more:

- managing of the state of the parser itself;
- merging of parse errors (when appropriate), should they happen.

Implementation of `(<|>)` in `ParsecT`'s instance of `Alternative` thus cannot delegate its work to the `Alternative` instance of the underlying monad `State MyState` and so no splitting of `MyState` happens—we have no backtracking.

Let us demonstrate this with an example:

```
{-# LANGUAGE OverloadedStrings #-}

module Main (main) where

import Control.Applicative
import Control.Monad.State.Strict
import Data.Text (Text)
import Data.Void
import Text.Megaparsec hiding (State)

type Parser = ParsecT Void Text (State String)

parser0 :: Parser String
parser0 = a <|> b
  where
    a = "foo" <$ put "branch A"
    b = get   <* put "branch B"

parser1 :: Parser String
parser1 = a <|> b
  where
    a = "foo" <$ put "branch A" <* empty
    b = get   <* put "branch B"

main :: IO ()
main = do
  let run p          = runState (runParserT p "" "") "initial"
      (Right a0, s0) = run parser0
      (Right a1, s1) = run parser1

  putStrLn  "Parser 0"
  putStrLn ("Result:      " ++ show a0)
  putStrLn ("Final state: " ++ show s0)

  putStrLn  "Parser 1"
  putStrLn ("Result:      " ++ show a1)
  putStrLn ("Final state: " ++ show s1)
```

Here is the result of running the program:

```
Parser 0
Result:      "foo"
Final state: "branch A"
Parser 1
Result:      "branch A"
Final state: "branch B"
```

With `parser0` we can see that the branch `b` is not tried. With `parser1` however it is obvious that the final result—the value returned by `get` —comes from the branch `a` even though it fails because of `empty` and it is the branch `b` that succeeds (`empty` in the context of parsing means "fail instantly and without any information about what has happened"). No backtracking happens.

What to do if we want backtracking custom state in our parser? We can provide that if we allow to wrap `ParsecT` *inside* `StateT` :

```
type MyStack a = StateT MyState (ParsecT Void Text Identity) a
```

Now if we use `(<|>)` in `MyStack` the instance used is that of `StateT` :

```
StateT m <|> StateT n = StateT $ \s -> m s <|> n s
```

Which gives us backtracking state and then delegates the rest of the work to `Alternative` instance of its inner monad— `ParsecT` . This behavior is exactly what we want:

```
{-# LANGUAGE OverloadedStrings #-}

module Main (main) where

import Control.Applicative
import Control.Monad.Identity
import Control.Monad.State.Strict
import Data.Text (Text)
import Data.Void
import Text.Megaparsec hiding (State)

type Parser = StateT String (ParsecT Void Text Identity)

parser :: Parser String
parser = a <|> b
  where
    a = "foo" <$ put "branch A" <* empty
    b = get   <* put "branch B"

main :: IO ()
main = do
  let p            = runStateT parser "initial"
      Right (a, s) = runParser p "" ""
```

```
  putStrLn ("Result:      " ++ show a)
  putStrLn ("Final state: " ++ show s)
```

The program prints:

```
Result:      "initial"
Final state: "branch B"
```

To make this approach feasible, `StateT` should support the whole set of primitive parsers, so we can work with it just like with `ParsecT`. In other words, it should be an instance of `MonadParsec`, just like it is an instance of not only `MonadState`, but also e.g. `MonadWriter` if its inner monad is an instance of `MonadWriter` (in MTL):

```
instance MonadWriter w m => MonadWriter w (StateT s m) where …
```

Indeed, we can lift primitives from inner instance of `MonadParsec` into `StateT`:

```
instance MonadParsec e s m => MonadParsec e s (StateT st m) where …
```

`megaparsec` defines instances of `MonadParsec` for all MTL monad transformers so that the user is free to insert the transformers inside of `ParsecT` or wrap `ParsecT` in those transformers achieving different kinds of interactions between the layers of monadic stack.

Lexing

Lexing is the process of transforming the input stream into a stream of tokens: integers, keywords, symbols, etc. which are easier to parse than the raw input directly, or are expected as input to parsers created with parser generators. Lexing can be performed in a separate pass with an external tool such as `alex`, but `megaparsec` also provides functions that should simplify writing a lexer in a seamless fashion, as part of your parser.

There are two lexer modules `Text.Megaparsec.Char.Lexer` for character streams and `Text.Megaparsec.Byte.Lexer` for byte streams. We will be using `Text.Megaparsec.Char.Lexer` because we work with a strict `Text` as the input stream, but most functions are mirrored in `Text.Megaparsec.Byte.Lexer` as well if you wish to work with `ByteString`s.

White space

The first topic we need to cover is dealing with white space. It is helpful to consume white space in a consistent manner either before every token or after every token. Megaparsec's lexer modules follow the strategy "assume no white space before token and consume all white space after token".

To consume white space we need a special parser that we will refer to as *space consumer*. The `Text.Megaparsec.Char.Lexer` module provides a helper allowing to build a general space consumer:

```
space :: MonadParsec e s m
  => m () -- ^ A parser for space characters which does not accept empty
          -- input (e.g. 'space1')
  -> m () -- ^ A parser for a line comment (e.g. 'skipLineComment')
  -> m () -- ^ A parser for a block comment (e.g. 'skipBlockComment')
  -> m ()
```

The documentation for the `space` function is quite comprehensive by itself, but let us complement it with an example:

```
{-# LANGUAGE OverloadedStrings #-}

module Main (main) where

import Data.Text (Text)
import Data.Void
import Text.Megaparsec
import Text.Megaparsec.Char
import qualified Text.Megaparsec.Char.Lexer as L -- (1)

type Parser = Parsec Void Text

sc :: Parser ()
sc = L.space
  space1                         -- (2)
  (L.skipLineComment "//")       -- (3)
  (L.skipBlockComment "/*" "*/") -- (4)
```

Some notes:

- The `Text.Megaparsec.Char.Lexer` is intended to be imported qualified because it contains names that collide with names from e.g. `Text.Megaparsec.Char`, for example `space`.
- The first argument of `L.space` should be a parser that is to be used to pick up white space. An important detail is that it should not accept empty input because then `L.space` would go into an infinite loop. `space1` is a parser from `Text.Megaparsec.Char` that meets the requirements perfectly.
- The second argument of `L.space` defines how to skip line comments, that is, comments that start with a given sequence of tokens and end with the end of line. The `skipLineComment` helper allows us to craft an auxiliary parser for line comments easily.
- The third argument of `L.space` in turn defines how to pick up block comments: everything between starting and ending sequences of tokens. The `skipBlockComment` helper allows us to deal with non-nested block comments. If supporting nested block comments is desirable, `skipBlockCommentNested` should be used instead.

Operationally, `L.space` tries all three parsers in turn as many times as it can till all of them cannot be applied anymore meaning that we have consumed all white space there is. Knowing this, it should make sense that if your grammar does not include block or line comments, you can just pass `empty` as the second and/or third argument of `L.space`. `empty`, being the identity of `(<|>)`, will just cause `L.space` to try the parser for the next white space component—exactly what is desirable.

Having the space consumer `sc`, we can then define various white space-related helpers:

```
lexeme :: Parser a -> Parser a
lexeme = L.lexeme sc

symbol :: Text -> Parser Text
symbol = L.symbol sc
```

- `lexeme` is a wrapper for lexemes that picks up all trailing white space using the supplied space consumer.
- `symbol` is a parser that matches given text using `string` internally and then similarly picks up all trailing white space.

We will see how it all works together in a moment, but first we need to introduce a couple more helpers from `Text.Megaparsec.Char.Lexer`.

Char and string literals

Parsing character and string literals can be tricky because of various escaping rules. To make life easier, `megaparsec` provides the `charLiteral` parser:

```
charLiteral :: (MonadParsec e s m, Token s ~ Char) => m Char
```

The job of `charLiteral` is to parse a single character that may be escaped according to the syntax for character literals described in the Haskell report. Note that it does not parse quotes around the literal though for two reasons:

- so the user can control how character literals are quoted,
- so `charLiteral` can be used to parse string literals as well.

Here are some example parsers built on top of `charLiteral`:

```
charLiteral :: Parser Char
charLiteral = between (char '\'') (char '\'') L.charLiteral

stringLiteral :: Parser String
stringLiteral = char '\"' *> manyTill L.charLiteral (char '\"')
```

- To turn `L.charLiteral` into a parser for char literals we only need to add the enclosing quotes. Here we follow Haskell syntax and use single quotes. The `between` combinator is defined simply as: `between open close p = open *> p <* close`.

- `stringLiteral` uses `L.charLiteral` to parse individual characters inside a string literal enclosed in double quotes.

The second function is also interesting because of its use of the `manyTill` combinator:

```
manyTill :: Alternative m => m a -> m end -> m [a]
manyTill p end = go
  where
    go = ([] <$ end) <|> ((:) <$> p <*> go)
```

`manyTill` tries to apply the `end` parser on every iteration and if it fails, it then runs the `p` parser and accumulates results of `p` in a list.

There is also `someTill` for when you want to demand that at least one item is present.

Numbers

Finally, a very common need is to parse numbers. For integral numbers, there are three helpers that can parse values in decimal, octal, and hexadecimal representations:

```
decimal, octal, hexadecimal
  :: (MonadParsec e s m, Token s ~ Char, Num a) => m a
```

Using them is easy:

```
integer :: Parser Integer
integer = lexeme L.decimal
```

```
λ> parseTest (integer <* eof) "123  "
123

λ> parseTest (integer <* eof) "12a  "
1:3:
  |
1 | 12a
  |   ^
unexpected 'a'
expecting end of input or the rest of integer
```

`scientific` accepts integer and fractional grammars, while `float` accepts only fractional grammars. `scientific` returns the `Scientific` type from the `scientific` package, while `float` is polymorphic in its result type and can return any instance of `RealFloat`:

```
scientific :: (MonadParsec e s m, Token s ~ Char)              => m Scientific
float      :: (MonadParsec e s m, Token s ~ Char, RealFloat a) => m a
```

For example:

```
float :: Parser Double
float = lexeme L.float
```

```
λ> parseTest (float <* eof) "123"
1:4:
  |
1 | 123
  |    ^
unexpected end of input
expecting '.', 'E', 'e', or digit

λ> parseTest (float <* eof) "123.45"
123.45

λ> parseTest (float <* eof) "123d"
1:4:
  |
1 | 123d
  |    ^
unexpected 'd'
expecting '.', 'E', 'e', or digit
```

Note that all these parsers do not parse signed numbers. To make a parser for signed numbers, we need to wrap an existing parser with the `signed` combinator:

```
signedInteger :: Parser Integer
signedInteger = L.signed sc integer

signedFloat :: Parser Double
signedFloat = L.signed sc float
```

The first argument of `signed` —the space consumer—controls how white space is consumed between the sign and actual numeral. If you do not want to allow space in there, just pass `return ()` instead.

`notFollowedBy` and `lookAhead`

There are two more primitives (in addition to `try`) that can perform look ahead in the input stream without actually advancing the parsing position in it.

The first one is called `notFollowedBy`:

```
notFollowedBy :: MonadParsec e s m => m a -> m ()
```

It succeeds only when its argument parser fails and never consumes any input or modifies the parser state.

As an example when you may want to use `notFollowedBy`, consider parsing of keywords:

```
pKeyword :: Text -> Parser Text
pKeyword keyword = lexeme (string keyword)
```

This parser has a problem: what if the keyword we are matching against is just a prefix of an identifier? In that case it is definitely not a keyword. Thus we must eliminate that case by using `notFollowedBy`:

```
pKeyword :: Text -> Parser Text
pKeyword keyword = lexeme (string keyword <* notFollowedBy alphaNumChar)
```

Another primitive is `lookAhead`:

```
lookAhead :: MonadParsec e s m => m a -> m a
```

If the argument `p` of `lookAhead` succeeds, the whole construct `lookAhead p` also succeeds but the input stream (and the entire parser state) stays untouched, i.e. nothing is consumed.

One example of where this may be useful is performing a check on an already parsed value and then either failing or continuing successfully. The idiom can be expressed in code like this:

```
withPredicate1
  :: (a -> Bool)       -- ^ The check to perform on parsed input
  -> String            -- ^ Message to print when the check fails
  -> Parser a          -- ^ Parser to run
  -> Parser a          -- ^ Resulting parser that performs the check
withPredicate1 f msg p = do
  r <- lookAhead p
  if f r
    then p
    else fail msg
```

This demonstrates a use of `lookAhead`, but we also should note that when the check if successful we perform the parsing twice, which is not good. Here is an alternative solution using the `getOffset` function:

```
withPredicate2
  :: (a -> Bool)       -- ^ The check to perform on parsed input
  -> String            -- ^ Message to print when the check fails
  -> Parser a          -- ^ Parser to run
  -> Parser a          -- ^ Resulting parser that performs the check
withPredicate2 f msg p = do
  o <- getOffset
  r <- p
  if f r
    then return r
    else do
      setOffset o
      fail msg
```

This way we just set offset in the input stream to what if was before running `p` and then fail. There is a mismatch now in what remains unconsumed vs offset position, but it does not

matter in this case because we end parsing immediately by calling `fail`. It may matter in other cases. We will see how to do better in situations like this later in this chapter.

Parsing expressions

By "expression" we mean a structure formed from terms and operators applied to those terms. Operators can be prefix, infix, and postfix, left and right-associative, with different precedence. An example of such a construct would be arithmetic expressions familiar from school:

```
a * (b + 2)
```

Here we can see two kinds of terms: variables (`a` and `b`) and integers (`2`). There are also two operators: `*` and `+`.

Writing an expression parser may take a while to get right. To help with that, the `parser-combinators` package comes with the `Control.Monad.Combinators.Expr` module which exports only two things: the `Operator` data type and the `makeExprParser` helper. Both are well documented, so in this section we will not repeat the documentation, instead we are going to write a simple but fully functional expression parser.

Let's start by defining a data type representing an expression as AST:

```
data Expr
  = Var String
  | Int Int
  | Negation Expr
  | Sum      Expr Expr
  | Subtr    Expr Expr
  | Product  Expr Expr
  | Division Expr Expr
  deriving (Eq, Ord, Show)
```

To use `makeExprParser` we need to provide it with a term parser and an operator table:

```
makeExprParser :: MonadParsec e s m
  => m a               -- ^ Term parser
  -> [[Operator m a]]  -- ^ Operator table, see 'Operator'
  -> m a               -- ^ Resulting expression parser
```

Let's start with the term parser. It is helpful to think about term as a box that that is to be considered as an indivisible whole by the expression parsing algorithm when it works with things like associativity and precedence. In our case there are three things that fall into this category: variables, integers, and entire expressions in parentheses. Using the definitions from previous chapters we can define the term parser as:

```
pVariable :: Parser Expr
pVariable = Var <$> lexeme
```

```
    ((:) <$> letterChar <*> many alphaNumChar <?> "variable")

pInteger :: Parser Expr
pInteger = Int <$> lexeme L.decimal

parens :: Parser a -> Parser a
parens = between (symbol "(") (symbol ")")

pTerm :: Parser Expr
pTerm = choice
  [ parens pExpr
  , pVariable
  , pInteger
  ]

pExpr :: Parser Expr
pExpr = makeExprParser pTerm operatorTable

operatorTable :: [[Operator Parser Expr]]
operatorTable = undefined -- TODO
```

The definitions of `pVariable`, `pInteger`, and `parens` should go without questions by now. We are also quite lucky here in that we do not need `try`s in `pTerm` because the grammars do not overlap:

- if we see an opening parenthesis `(`, we know that an expression in parentheses is to follow, so we commit to that branch;
- if we see a letter, we know that it is the start of an identifier;
- if we see a digit, we know that it is the start of an integer.

Finally, to finish `pExpr` we need to define the `operatorTable`. We can see from the type that it is a nested list. Every inner list is a list of operators we want to support, they all have equal precedence. The outer list is ordered in descending precedence, so the higher we place a group of operators in it, the tighter they bind:

```
data Operator m a -- N.B.
  = InfixN  (m (a -> a -> a)) -- ^ Non-associative infix
  | InfixL  (m (a -> a -> a)) -- ^ Left-associative infix
  | InfixR  (m (a -> a -> a)) -- ^ Right-associative infix
  | Prefix  (m (a -> a))      -- ^ Prefix
  | Postfix (m (a -> a))      -- ^ Postfix

operatorTable :: [[Operator Parser Expr]]
operatorTable =
  [ [ prefix "-" Negation
    , prefix "+" id
    ]
```

```
  , [ binary "*" Product
    , binary "/" Division
    ]
  , [ binary "+" Sum
    , binary "-" Subtr
    ]
  ]

binary :: Text -> (Expr -> Expr -> Expr) -> Operator Parser Expr
binary  name f = InfixL  (f <$ symbol name)

prefix, postfix :: Text -> (Expr -> Expr) -> Operator Parser Expr
prefix  name f = Prefix  (f <$ symbol name)
postfix name f = Postfix (f <$ symbol name)
```

Note how we place `Parser (Expr -> Expr -> Expr)` inside `InfixL` in `binary` and similarly `Parser (Expr -> Expr)` in `prefix` and `postfix`. That is, we run `symbol name` and return a function to apply to the terms in order to get the final result of the type `Expr`.

We can now try our parser, it is ready!

```
λ> parseTest (pExpr <* eof) "a * (b + 2)"
Product (Var "a") (Sum (Var "b") (Int 2))

λ> parseTest (pExpr <* eof) "a * b + 2"
Sum (Product (Var "a") (Var "b")) (Int 2)

λ> parseTest (pExpr <* eof) "a * b / 2"
Division (Product (Var "a") (Var "b")) (Int 2)

λ> parseTest (pExpr <* eof) "a * (b $ 2)"
1:8:
  |
1 | a * (b $ 2)
  |        ^
unexpected '$'
expecting ')' or operator
```

Documentation for the `Control.Monad.Combinators.Expr` module contains some hints that are useful in certain less-standard situations, so it is a good idea to read it as well.

Indentation-sensitive parsing

The `Text.Megaparsec.Char.Lexer` module contains tools that should be helpful when parsing indentation-sensitive grammars. We are going to review the available combinators first, then put them into use by writing an indentation-sensitive parser.

`nonIndented` and `indentBlock`

Let's start with the simplest thing— `nonIndented`:

```
nonIndented :: (TraversableStream s, MonadParsec e s m)
  => m ()              -- ^ How to consume indentation (white space)
  -> m a               -- ^ Inner parser
  -> m a
```

It allows us to make sure that its inner parser consumes input that is *not* indented. It is a part of a model behind high-level parsing of indentation-sensitive input. We state that there are top-level items that are not indented and that all indented tokens are directly or indirectly children of those top-level definitions. In `megaparsec`, we do not need any additional state to express this. Since indentation is always relative, our idea is to explicitly tie parsers for reference tokens and indented tokens, thus defining indentation-sensitive grammar via pure combination of parsers.

So, how do we define a parser for indented block? Let's take a look at the signature of `indentBlock`:

```
indentBlock :: (TraversableStream s, MonadParsec e s m, Token s ~ Char)
  => m ()              -- ^ How to consume indentation (white space)
  -> m (IndentOpt m a b) -- ^ How to parse “reference” token
  -> m a
```

First, we specify how to consume indentation. An important thing to note here is that this space-consuming parser *must* consume newlines as well, while tokens (reference token and indented tokens) should not normally consume newlines after them.

As you can see, the second argument allows us to parse reference token and return a data structure that tells `indentBlock` what to do next. There are several options:

```
data IndentOpt m a b
  = IndentNone a
    -- ^ Parse no indented tokens, just return the value
  | IndentMany (Maybe Pos) ([b] -> m a) (m b)
    -- ^ Parse many indented tokens (possibly zero), use given indentation
    -- level (if 'Nothing', use level of the first indented token); the
    -- second argument tells how to get the final result, and the third
    -- argument describes how to parse an indented token
  | IndentSome (Maybe Pos) ([b] -> m a) (m b)
    -- ^ Just like 'IndentMany', but requires at least one indented token to
    -- be present
```

We can change our mind and parse no indented tokens, we can parse *many* (that is, possibly zero) indented tokens or require *at least one* such token. We can either allow `indentBlock` to detect the indentation level of the first indented token and use that, or manually specify the indentation level.

Parsing a simple indented list

Let's parse a simple indented list of some items. We begin with the import section:

```
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE TupleSections     #-}

module Main (main) where

import Control.Applicative hiding (some)
import Control.Monad (void)
import Data.Text (Text)
import Data.Void
import Text.Megaparsec
import Text.Megaparsec.Char
import qualified Text.Megaparsec.Char.Lexer as L

type Parser = Parsec Void Text
```

We will need two kinds of space-consumers: one that consumes new lines `scn` and one that does not `sc` (actually it only parses spaces and tabs here):

```
lineComment :: Parser ()
lineComment = L.skipLineComment "#"

scn :: Parser ()
scn = L.space space1 lineComment empty

sc :: Parser ()
sc = L.space (void $ some (char ' ' <|> char '\t')) lineComment empty

lexeme :: Parser a -> Parser a
lexeme = L.lexeme sc
```

Just for fun, we allow line comments that start with `#`.

`pItemList` is a top-level form that itself is a combination of reference token (header of list) and indented tokens (list items), so:

```
pItemList :: Parser (String, [String]) -- header and list items
pItemList = L.nonIndented scn (L.indentBlock scn p)
  where
    p = do
      header <- pItem
      return (L.IndentMany Nothing (return . (header, )) pItem)
```

For our purposes, an item is a sequence of alpha-numeric characters and dashes:

```
pItem :: Parser String
pItem = lexeme (some (alphaNumChar <|> char '-')) <?> "list item"
```

Let's load the code into GHCi and try it with the help of `parseTest` built-in:

```
λ> parseTest (pItemList <* eof) ""
1:1:
  |
1 | <empty line>
  | ^
unexpected end of input
expecting list item

λ> parseTest (pItemList <* eof) "something"
("something",[])

λ> parseTest (pItemList <* eof) "  something"
1:3:
  |
1 |   something
  |   ^
incorrect indentation (got 3, should be equal to 1)

λ> parseTest (pItemList <* eof) "something\none\ntwo\nthree"
2:1:
  |
2 | one
  | ^
unexpected 'o'
expecting end of input
```

Remember that we are using the `IndentMany` option, so empty lists are OK, on the other hand the built-in combinator `space` has hidden the phrase “expecting more space” from error messages, so this error message is perfectly reasonable.

Let’s continue:

```
λ> parseTest (pItemList <* eof) "something\n  one\n    two\n  three"
3:5:
  |
3 |     two
  |     ^
incorrect indentation (got 5, should be equal to 3)

λ> parseTest (pItemList <* eof) "something\n  one\n  two\n three"
4:2:
  |
4 |  three
  |  ^
incorrect indentation (got 2, should be equal to 3)

λ> parseTest (pItemList <* eof) "something\n  one\n  two\n  three"
("something",["one","two","three"])
```

Let's replace `IndentMany` with `IndentSome` and `Nothing` with `Just (mkPos 5)` (indentation levels are counted from 1, so it will require 4 spaces before indented items):

```
pItemList :: Parser (String, [String])
pItemList = L.nonIndented scn (L.indentBlock scn p)
  where
    p = do
      header <- pItem
      return (L.IndentSome (Just (mkPos 5)) (return . (header, )) pItem)
```

Now:

```
λ> parseTest (pItemList <* eof) "something\n"
2:1:
  |
2 | <empty line>
  | ^
incorrect indentation (got 1, should be greater than 1)

λ> parseTest (pItemList <* eof) "something\n  one"
2:3:
  |
2 |   one
  |   ^
incorrect indentation (got 3, should be equal to 5)

λ> parseTest (pItemList <* eof) "something\n    one"
("something",["one"])
```

The first message may be a bit surprising, but `megaparsec` knows that there must be at least one item in the list, so it checks the indentation level and it is 1, which is incorrect, so it reports it.

Nested indented list

Let's allow list items to have sub-items. For this we will need a new parser, `pComplexItem`:

```
pComplexItem :: Parser (String, [String])
pComplexItem = L.indentBlock scn p
  where
    p = do
      header <- pItem
      return (L.IndentMany Nothing (return . (header, )) pItem)

pItemList :: Parser (String, [(String, [String])])
pItemList = L.nonIndented scn (L.indentBlock scn p)
  where
    p = do
      header <- pItem
      return (L.IndentSome Nothing (return . (header, )) pComplexItem)
```

If we feed something like this:

```
first-chapter
  paragraph-one
      note-A # an important note here!
      note-B
  paragraph-two
    note-1
    note-2
  paragraph-three
```

into our parser, we will get:

```
Right
  ( "first-chapter"
  , [ ("paragraph-one",   ["note-A","note-B"])
    , ("paragraph-two",   ["note-1","note-2"])
    , ("paragraph-three", [])
    ]
  )
```

This demonstrates how this approach scales for nested indented construts without requiring additional state.

Adding line folds

A *line fold* consists of several elements that can be put on one line or on several lines as long as the indentation level of the subsequent items is greater than the indentation level of the first item.

Let's make use of another helper called `lineFold`:

```
pComplexItem :: Parser (String, [String])
pComplexItem = L.indentBlock scn p
  where
    p = do
      header <- pItem
      return (L.IndentMany Nothing (return . (header, )) pLineFold)

pLineFold :: Parser String
pLineFold = L.lineFold scn $ \sc' ->
  let ps = some (alphaNumChar <|> char '-') `sepBy1` try sc'
  in unwords <$> ps <* scn -- (1)
```

`lineFold` works like this: we give it a space consumer that accepts newlines `scn` and it gives back a special space consumer `sc'` that we can use in the callback to consume space between elements of line fold.

Why use `try sc'` and `scn` on the line (1)? The situation is the following:

- Components of a line fold can only be more indented than its start.
- `sc'` consumes whitespace with newlines in such a way that after consuming whitespace the column number is greater than initial column.
- To stop, `sc'` should encounter the opposite situation, that is, the column number after consumption should be less than or equal to the initial column. At that point it fails without consuming input (thanks to `try`) and `scn` is used to pick up whitespace before that new thing that will start at that column.
- Previously used `sc'` already probed whitespace with space consumer which consumes newlines. So, it is only logical to also consume newlines when picking up trailing whitespace. This is why `scn` is used on the line (1) and not `sc`.

EXERCISE: Playing with the final version of our parser is left as an exercise for the reader. You can create "items" that consist of multiple words and as long as they are line-folded they will be parsed and concatenated with single space between them.

Writing efficient parsers

Let's discuss what to attempt in order to improve performance of a `megaparsec` parser. It should be noted right away that one should always check if there is any improvement through profiling and benchmarking. That is the only way to understand if we are doing the right thing when tuning performance.

Common pieces of advice:

- If your parser uses a monad stack instead of the plain `Parsec` monad (recall that it is the `ParsecT` monad transformer over `Identity`, which is quite lightweight), make sure you use at least version 0.5 of `transformers` library, and at least version 7.0 of `megaparsec`. Both libraries have critical performance improvements in these versions, so you can just get better performance for free.

- `Parsec` monad will be always faster then `ParsecT`-based monad transformers. Avoid using `StateT`, `WriterT`, and other monad transformers unless absolutely necessary. The more you add to the monadic stack, the slower your parser will be.

- Backtracking is an expensive operation. Avoid building long chains of alternatives where every alternative can go deep into input before failing.

- Do not keep your parsers polymorphic unless you really have a reason to do so. It is best to fix the types of parsers specifying concrete types, such as `type Parser = Parsec Void Text` for every top-level definition. This way GHC will be able to optimize better.

- Inline generously (when it makes sense, of course). You may not believe your eyes when you see how much of a difference inlining can do, especially for short functions. This is especially true for parsers that are defined in one module and used in another one, because `INLINE` and `INLINEABLE` pragmas make GHC dump function definitions into interface files and this facilitates specializing.

- Use the fast primitives such as `takeWhileP`, `takeWhile1P`, and `takeP` whenever you can. This blog post explains why they are so fast.
- Avoid `oneOf` and `noneOf` preferring `satisfy` and `notChar` whenever possible.

While most of the points above do not require additional comment, I think it would be beneficial to get into the habit of using the newer fast primitives: `takeWhileP`, `takeWhile1P`, and `takeP`. The first two are especially common as they allow us to replace `many` and `some`-based constructs making them faster and changing the type of returned data to chunk of input stream, i.e. the `Tokens s` type we have discussed previously.

For example, recall that when we parsed URIs, we had this code for parsing username in the authority component:

```
user <- T.pack <$> some alphaNumChar
```

We can replace it by `takeWhile1P`:

```
user <- takeWhile1P (Just "alpha num character") isAlphaNum
--                  ^                            ^
--                  |                            |
-- label for tokens we match against         predicate
```

When we parse `ByteString`s and `Text`, this will be a lot faster than the original approach. Also note that `T.pack` is not necessary anymore as we get `Text` directly from `takeWhile1P`.

These equations may be helpful for understanding the meaning of the `Maybe String` argument of `takeWhileP` and `takeWhile1P`:

```
takeWhileP  (Just "foo") f = many (satisfy f <?> "foo")
takeWhileP  Nothing      f = many (satisfy f)
takeWhile1P (Just "foo") f = some (satisfy f <?> "foo")
takeWhile1P Nothing      f = some (satisfy f)
```

Parse errors

Now that we have explored how to use most features of `megaparsec`, it is time to learn more about parse errors: how they are defined, how to signal them, and how to process them inside a running parser.

Parse error definitions

The `ParseError` type is defined like this:

```
data ParseError s e
  = TrivialError Int (Maybe (ErrorItem (Token s))) (Set (ErrorItem (Token s)))
    -- ^ Trivial errors, generated by Megaparsec's machinery. The data
    -- constructor includes the offset of error, unexpected token (if any),
    -- and expected tokens.
```

```
  | FancyError Int (Set (ErrorFancy e))
    -- ^ Fancy, custom errors.
```

In English: a `ParseError` is either a `TrivialError` with at most one unexpected item and a (possibly empty) collection of expected items or a `FancyError`.

`ParseError s e` is parametrized over two type variables:

- `s` is the type of input stream.
- `e` is the type of custom component of parse error.

`ErrorItem` is defined as:

```
data ErrorItem t
  = Tokens (NonEmpty t)      -- ^ Non-empty stream of tokens
  | Label (NonEmpty Char)    -- ^ Label (cannot be empty)
  | EndOfInput               -- ^ End of input
```

`NonEmpty` is a type for non-empty lists, it comes from `Data.List.NonEmpty`. And here is `ErrorFancy`:

```
data ErrorFancy e
  = ErrorFail String
    -- ^ 'fail' has been used in parser monad
  | ErrorIndentation Ordering Pos Pos
    -- ^ Incorrect indentation error: desired ordering between reference
    -- level and actual level, reference indentation level, actual
    -- indentation level
  | ErrorCustom e
    -- ^ Custom error data, can be conveniently disabled by indexing
    -- 'ErrorFancy' by 'Void'
```

`ErrorFancy` includes data constructors for two common cases `megaparsec` supports out-of-the-box:

- Use of the `fail` function that causes parser to fail reporting an arbitrary `String`.
- Indentation-related issues which we have seen in a previous section. Since we provide tools for working with indentation-sensitive grammars out-of-the-box, we need a way to store well-typed information about problems with indentation.

Finally, `ErrorCustom` is a sort of an "extension slot" which allows to embed arbitrary data into the `ErrorFancy` type. When we do not need any custom data in our parse errors, we parametrize `ErrorFancy` by `Void`. Since `Void` is not inhabited by non-bottom values, `ErrorCustom` becomes "cancelled out" or, if we follow the analogy between algebraic data types and numbers, "multiplied by zero".

In older version of the library, `ParseError`s were returned directly by functions like `parse`, but version 7 delays calculation of line and column for each error, as well as fetching of the

relevant line on input for displaying in case of an error. This is done to be make parsing faster, because all this information is usually useful only when a parser fails. Another problem of older versions of the library is that displaying several parse errors at once required re-traversal of input each time to fetch the right line.

The problem is solved with the `ParseErrorBundle` data type:

```
-- | A non-empty collection of 'ParseError's equipped with 'PosState' that
-- allows to pretty-print the errors efficiently and correctly.

data ParseErrorBundle s e = ParseErrorBundle
  { bundleErrors :: NonEmpty (ParseError s e)
    -- ^ A collection of 'ParseError's that is sorted by parse error offsets
  , bundlePosState :: PosState s
    -- ^ State that is used for line\/column calculation
  }
```

All parser-running functions return `ParseErrorBundle` with a correctly set `bundlePosState` and a collection `ParseError` s inside.

How to signal a parse error

Let's discuss different ways to signal a parse error. The simplest function for that is `fail` :

```
λ> parseTest (fail "I'm failing, help me!" :: Parser ()) ""
1:1:
  |
1 | <empty line>
  | ^
I'm failing, help me!
```

For many people who are familiar with simpler parsing libraries such as `parsec` this is often enough. However, displaying a parse error to the user is not everything, we may have a need to analyze and/or manipulate it. This is where `String` s are not very convenient.

Trivial parse errors are usually generated by `megaparsec` , but we can signal any such an error ourselves using the `failure` combinator:

```
failure :: MonadParsec e s m
  => Maybe (ErrorItem (Token s)) -- ^ Unexpected item (if any)
  -> Set (ErrorItem (Token s)) -- ^ Expected items
  -> m a

unfortunateParser :: Parser ()
unfortunateParser = failure (Just EndOfInput) (Set.fromList es)
  where
    es = [Tokens (NE.fromList "a"), Tokens (NE.fromList "b")]

λ> parseTest unfortunateParser ""
1:1:
```

```
  |
1 | <empty line>
  | ^
unexpected end of input
expecting 'a' or 'b'
```

Unlike the `fail`-based approach, trivial parse errors are easy to pattern-match on, inspect, and modify.

For fancy errors we correspondingly have the `fancyFaliure` combinator:

```
fancyFailure :: MonadParsec e s m
  => Set (ErrorFancy e) -- ^ Fancy error components
  -> m a
```

With `fancyFailure`, it is often desirable to define a helper like the one we have in the lexer modules instead of calling `fancyFailure` directly:

```
incorrectIndent :: MonadParsec e s m
  => Ordering  -- ^ Desired ordering between reference level and actual level
  -> Pos               -- ^ Reference indentation level
  -> Pos               -- ^ Actual indentation level
  -> m a
incorrectIndent ord ref actual = fancyFailure . Set.singleton $
  ErrorIndentation ord ref actual
```

As an example of adding a custom parse error component to your parser, let's go through defining a special parse error that says that given `Text` value is not a keyword.

First, we need to define the data type with constructors representing scenarios we want to support:

```
data Custom = NotKeyword Text
  deriving (Eq, Show, Ord)
```

And tell `megaparsec` how to display it in parse errors:

```
instance ShowErrorComponent Custom where
  showErrorComponent (NotKeyword txt) = T.unpack txt ++ " is not a keyword"
```

Next we update our `Parser` type synonym:

```
type Parser = Parsec Custom Text
```

After that we can define the `notKeyword` helper:

```
notKeyword :: Text -> Parser a
notKeyword = customFailure . NotKeyword
```

Where `customFailure` is a useful helper that comes from the `Text.Megaparsec` module:

```
customFailure :: MonadParsec e s m => e -> m a
customFailure = fancyFailure . E.singleton . ErrorCustom
```

Finally, let us try it:

```
λ> parseTest (notKeyword "foo" :: Parser ()) ""
1:1:
  |
1 | <empty line>
  | ^
foo is not a keyword
```

Displaying parse errors

Displaying of `ParseErrorBundle` s is done with the `errorBundlePretty` function:

```
-- | Pretty-print a 'ParseErrorBundle'. All 'ParseError's in the bundle will
-- be pretty-printed in order together with the corresponding offending
-- lines by doing a single efficient pass over the input stream. The
-- rendered 'String' always ends with a newline.

errorBundlePretty
  :: ( VisualStream s
     , TraversableStream s
     , ShowErrorComponent e
     )
  => ParseErrorBundle s e -- ^ Parse error bundle to display
  -> String               -- ^ Textual rendition of the bundle
```

In 99% of cases you will only need this one function.

Catching parse errors in a running parser

Another useful feature of `megaparsec` is that it is possible to “catch” a parse error, alter it in some way, and then re-throw, just like with exceptions. This is enabled by the `observing` primitive:

```
-- | @'observing' p@ allows to “observe” failure of the @p@ parser, should
-- it happen, without actually ending parsing, but instead getting the
-- 'ParseError' in 'Left'. On success parsed value is returned in 'Right'
-- as usual. Note that this primitive just allows you to observe parse
-- errors as they happen, it does not backtrack or change how the @p@
-- parser works in any way.

observing :: MonadParsec e s m
  => m a             -- ^ The parser to run
  -> m (Either (ParseError (Token s) e) a)
```

Here is a complete program demonstrating typical usage of `observing` :

```haskell
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE TypeApplications  #-}

module Main (main) where

import Control.Applicative hiding (some)
import Data.List (intercalate)
import Data.Set (Set)
import Data.Text (Text)
import Data.Void
import Text.Megaparsec
import Text.Megaparsec.Char
import qualified Data.Set as Set

data Custom
  = TrivialWithLocation
    [String] -- position stack
    (Maybe (ErrorItem Char))
    (Set (ErrorItem Char))
  | FancyWithLocation
    [String] -- position stack
    (ErrorFancy Void) -- Void, because we do not want to allow to nest Customs
  deriving (Eq, Ord, Show)

instance ShowErrorComponent Custom where
  showErrorComponent (TrivialWithLocation stack us es) =
    parseErrorTextPretty (TrivialError @Text @Void undefined us es)
      ++ showPosStack stack
  showErrorComponent (FancyWithLocation stack cs) =
    parseErrorTextPretty (FancyError @Text @Void undefined (Set.singleton cs))
      ++ showPosStack stack

showPosStack :: [String] -> String
showPosStack = intercalate ", " . fmap ("in " ++)

type Parser = Parsec Custom Text

inside :: String -> Parser a -> Parser a
inside location p = do
  r <- observing p
  case r of
    Left (TrivialError _ us es) ->
      fancyFailure . Set.singleton . ErrorCustom $
        TrivialWithLocation [location] us es
    Left (FancyError _ xs) -> do
      let f (ErrorFail msg) = ErrorCustom $
            FancyWithLocation [location] (ErrorFail msg)
          f (ErrorIndentation ord rlvl alvl) = ErrorCustom $
            FancyWithLocation [location] (ErrorIndentation ord rlvl alvl)
```

```
          f (ErrorCustom (TrivialWithLocation ps us es)) = ErrorCustom $
            TrivialWithLocation (location:ps) us es
          f (ErrorCustom (FancyWithLocation ps cs)) = ErrorCustom $
            FancyWithLocation (location:ps) cs
      fancyFailure (Set.map f xs)
    Right x -> return x

myParser :: Parser String
myParser = some (char 'a') *> some (char 'b')

main :: IO ()
main = do
  parseTest (inside "foo" myParser) "aaacc"
  parseTest (inside "foo" $ inside "bar" myParser) "aaacc"
```

EXERCISE: Understand in details how this program works.

If I run this program, I see the following output:

```
1:4:
  |
1 | aaacc
  |    ^
unexpected 'c'
expecting 'a' or 'b'
in foo
1:4:
  |
1 | aaacc
  |    ^
unexpected 'c'
expecting 'a' or 'b'
in foo, in bar
```

Thus, the feature can be used to attach location labels to parse errors, or indeed define *regions* in which parse errors are processed in some way. The idiom is quite useful, so there is even a non-primitive helper called `region` defined in terms of the `observing` primitive:

```
-- | Specify how to process 'ParseError's that happen inside of this
-- wrapper. This applies to both normal and delayed 'ParseError's.
--
-- As a side-effect of the implementation the inner computation will start
-- with empty collection of delayed errors and they will be updated and
-- “restored” on the way out of 'region'.

region :: MonadParsec e s m
  => (ParseError s e -> ParseError s e)
     -- ^ How to process 'ParseError's
  -> m a
     -- ^ The “region” that the processing applies to
```

```
  -> m a
region f m = do
  r <- observing m
  case r of
    Left err -> parseError (f err) -- see the next section
    Right x -> return x
```

EXERCISE: Rewrite the `inside` function in the program above using `region`.

Controlling location of parse errors

The definition of `region` uses the `parseError` primitive:

```
parseError :: MonadParsec e s m => ParseError s e -> m a
```

It is the fundamental primitive for error reporting and all other functions we have seen so far are defined in terms of `parseError`:

```
failure
  :: MonadParsec e s m
  => Maybe (ErrorItem (Token s)) -- ^ Unexpected item (if any)
  -> Set (ErrorItem (Token s)) -- ^ Expected items
  -> m a
failure us ps = do
  o <- getOffset
  parseError (TrivialError o us ps)

fancyFailure
  :: MonadParsec e s m
  => Set (ErrorFancy e) -- ^ Fancy error components
  -> m a
fancyFailure xs = do
  o <- getOffset
  parseError (FancyError o xs)
```

One thing `parseError` allows you to do is to set error offset (that is, position) to something else than current position in input stream. Let's return to the example with rejecting results of parsing retroactively:

```
withPredicate2
  :: (a -> Bool)       -- ^ The check to perform on parsed input
  -> String            -- ^ Message to print when the check fails
  -> Parser a          -- ^ Parser to run
  -> Parser a          -- ^ Resulting parser that performs the check
withPredicate2 f msg p = do
  o <- getOffset
  r <- p
  if f r
    then return r
    else do
```

```
      setOffset o
      fail msg
```

We noted that `setOffset o` will make the error to be located correctly, but it will also invalidate the parser state as a side effect—the offset will not reflect reality anymore. This may be a real problem in more complex parsers. For example, imagine that you enclose `withPredicate2` with `observing` so that there will be some code running after `fail`.

With `parseError` and `region` we finally have proper solution to the problem—either use `region` to reset the parse error location, or use `parseError` in the first place:

```
withPredicate3
  :: (a -> Bool)       -- ^ The check to perform on parsed input
  -> String            -- ^ Message to print when the check fails
  -> Parser a          -- ^ Parser to run
  -> Parser a          -- ^ Resulting parser that performs the check
withPredicate3 f msg p = do
  o <- getOffset
  r <- p
  if f r
    then return r
    else region (setErrorOffset o) (fail msg)

withPredicate4
  :: (a -> Bool)       -- ^ The check to perform on parsed input
  -> String            -- ^ Message to print when the check fails
  -> Parser a          -- ^ Parser to run
  -> Parser a          -- ^ Resulting parser that performs the check
withPredicate4 f msg p = do
  o <- getOffset
  r <- p
  if f r
    then return r
    else parseError (FancyError o (Set.singleton (ErrorFail msg)))
```

Reporting multiple parse errors

Finally, `megaparsec` allows us to signal several parse errors in a single run. This may be helpful for the end users because they will be able to fix several issues at once and so they will need to run your parser fewer times.

One prerequisite for having a multi-error parser is that it should be possible to skip over a problematic part of input and resume parsing from a position that is known to be good. This part is accomplished by using the `withRecovery` primitive:

```
-- | @'withRecovery' r p@ allows continue parsing even if parser @p@
-- fails. In this case @r@ is called with the actual 'ParseError' as its
-- argument. Typical usage is to return a value signifying failure to
-- parse this particular object and to consume some part of the input up
```

```
-- to the point where the next object starts.
--
-- Note that if @r@ fails, original error message is reported as if
-- without 'withRecovery'. In no way recovering parser @r@ can influence
-- error messages.

withRecovery
  :: (ParseError s e -> m a) -- ^ How to recover from failure
  -> m a                     -- ^ Original parser
  -> m a                     -- ^ Parser that can recover from failures
```

Before Megaparsec 8 users had to pick the type `a` to be a sum type including the possibilities for success and failure. For example, it could be `Either (ParseError s e) Result`. The parse errors had to be collected and later manually added to the `ParseErrorBundle` before displaying. Needless to say, all of this was an example of advanced usage that was not user friendly.

Megaparsec 8 supports *delayed parse errors*:

```
-- | Register a 'ParseError' for later reporting. This action does not end
-- parsing and has no effect except for adding the given 'ParseError' to the
-- collection of “delayed” 'ParseError's which will be taken into
-- consideration at the end of parsing. Only if this collection is empty
-- parser will succeed. This is the main way to report several parse errors
-- at once.

registerParseError :: MonadParsec e s m => ParseError s e -> m ()

-- | Like 'failure', but for delayed 'ParseError's.

registerFailure
  :: MonadParsec e s m
  => Maybe (ErrorItem (Token s)) -- ^ Unexpected item (if any)
  -> Set (ErrorItem (Token s)) -- ^ Expected items
  -> m ()

-- | Like 'fancyFailure', but for delayed 'ParseError's.

registerFancyFailure
  :: MonadParsec e s m
  => Set (ErrorFancy e) -- ^ Fancy error components
  -> m ()
```

These errors can be registered in the error-processing callback of `withRecovery` making the resulting type `Maybe Result`. This takes care of including the delayed errors in the final `ParseErrorBundle` as well as making the parser fail in the end if the collection of delayed errors in not empty.

With all this, we hope that the practice of writing multi-error parsers will become more common among the users.

Testing Megaparsec parsers

Testing a parser is a practical task most people face sooner or later, so we are bound to cover it. The recommended way to test `megaparsec` parsers is by using the [hspec-megaparsec](#) package. The package adds utility expectations such as `shouldParse`, `parseSatisfies`, etc. which work with the `hspec` testing framework.

Let's start with an example:

```
{-# LANGUAGE OverloadedStrings #-}

module Main (main) where

import Control.Applicative hiding (some)
import Data.Text (Text)
import Data.Void
import Test.Hspec
import Test.Hspec.Megaparsec
import Text.Megaparsec
import Text.Megaparsec.Char

type Parser = Parsec Void Text

myParser :: Parser String
myParser = some (char 'a')

main :: IO ()
main = hspec $
  describe "myParser" $ do
    it "returns correct result" $
      parse myParser "" "aaa" `shouldParse` "aaa"
    it "result of parsing satisfies what it should" $
      parse myParser "" "aaaa" `parseSatisfies` ((== 4) . length)
```

`shouldParse` accepts `Either (ParseErrorBundle s e) a` —the result of parsing and a thing of the type `a` to compare with. It is probably the most common helper. `parseSatisfies` is quite similar, but instead of comparing for equality with the expected result, it allows to check the result by applying an arbitrary predicate.

Other simple expectations are `shouldSucceedOn` and `shouldFailOn` (although they are rarely used):

```
    it "should parse 'a's all right" $
      parse myParser "" `shouldSucceedOn` "aaaa"
    it "should fail on 'b's" $
      parse myParser "" `shouldFailOn` "bbb"
```

With `megaparsec` we want to be precise about parse errors our parsers produce. To test parse errors there is `shouldFailWith`, which can be used like this:

```
it "fails on 'b's producing correct error message" $
  parse myParser "" "bbb" `shouldFailWith`
    TrivialError
      0
      (Just (Tokens ('b' :| [])))
      (Set.singleton (Tokens ('a' :| [])))
```

Writing out a `TrivialError` like this is tiresome. The definition of `ParseError` contains "inconvenient" types like `Set` and `NonEmpty` which are not handy to enter directly as we have just seen. Fortunately, `Test.Hspec.Megaparsec` also re-exports the `Text.Megaparsec.Error.Builder` module which provides an API for easier construction of `ParserError`s. Let us instead use the `err` helper:

```
it "fails on 'b's producing correct error message" $
  parse myParser "" "bbb" `shouldFailWith` err 0 (utok 'b' <> etok 'a')
```

- The first argument of `err` is offset of the parse error (the number of tokens that had been consumed before we got the error). In this example it is simply 0.
- `utok` stands for "unexpected token", similarly `etok` means "expected token".

EXERCISE: Familiarize yourself with `errFancy`, which is used to construct fancy parse errors.

Finally, it is possible to test what part of input remains unconsumed after parsing using `failsLeaving` and `succeedsLeaving`:

```
it "consumes all 'a's but does not touch 'b's" $
  runParser' myParser (initialState "aaabbb") `succeedsLeaving` "bbb"
it "fails without consuming anything" $
  runParser' myParser (initialState "bbbccc") `failsLeaving` "bbbccc"
```

These should by used with `runParser'` or `runParserT'` which accept custom initial state of parser and return its final state (this is what allows us to check the leftover of the input stream after parsing):

```
runParser'
  :: Parsec e s a      -- ^ Parser to run
  -> State s           -- ^ Initial state
  -> (State s, Either (ParseError (Token s) e) a)

runParserT' :: Monad m
  => ParsecT e s m a   -- ^ Parser to run
  -> State s           -- ^ Initial state
  -> m (State s, Either (ParseError (Token s) e) a)
```

The `initialState` function takes the input stream and returns the initial state with that input stream and other record fields filled with their default values.

Other sources of inspiration for using `hspec-megaparsec` are:

- Megaparsec's own test suite is written using `hspec-megaparsec`.
- The toy test suite that comes with `hspec-megaparsec` itself.

Working with custom input streams

`megaparsec` can be used to parse any input that is an instance of the `Stream` type class. This means that it may be used in conjunction with a lexing tool such as `alex`.

Not to digress from our main topic by presenting how a stream of tokens could be generated with `alex`, we will assume it in the following form:

```
{-# LANGUAGE LambdaCase        #-}
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE RecordWildCards   #-}
{-# LANGUAGE TypeFamilies      #-}

module Main (main) where

import Data.List.NonEmpty (NonEmpty (..))
import Data.Proxy
import Data.Void
import Text.Megaparsec
import qualified Data.List          as DL
import qualified Data.List.NonEmpty as NE
import qualified Data.Set           as Set

data MyToken
  = Int Int
  | Plus
  | Mul
  | Div
  | OpenParen
  | CloseParen
  deriving (Eq, Ord, Show)
```

To report parse errors though we need a way to know the token's starting position, ending position, and length, so let's add `WithPos`:

```
data WithPos a = WithPos
  { startPos :: SourcePos
  , endPos :: SourcePos
  , tokenLength :: Int
```

```
  , tokenVal :: a
  } deriving (Eq, Ord, Show)
```

Then we can have a data type for our stream:

```
data MyStream = MyStream
  { myStreamInput :: String -- for showing offending lines
  , unMyStream :: [WithPos MyToken]
  }
```

Next, we need to make `MyStream` an instance of the `Stream` type class. This requires the `TypeFamilies` language extension because we want to define the associated type functions `Token` and `Tokens`:

```
instance Stream MyStream where
  type Token  MyStream = WithPos MyToken
  type Tokens MyStream = [WithPos MyToken]
  -- …
```

`Stream`, `VisualStream`, and `TraversableStream` are documented in the `Text.Megaparsec.Stream` module. Here we go straight to defining the methods:

```
instance Stream MyStream where
  type Token  MyStream = WithPos MyToken
  type Tokens MyStream = [WithPos MyToken]

  tokenToChunk Proxy x = [x]
  tokensToChunk Proxy xs = xs
  chunkToTokens Proxy = id
  chunkLength Proxy = length
  chunkEmpty Proxy = null
  take1_ (MyStream _ []) = Nothing
  take1_ (MyStream str (t:ts)) = Just
    ( t
    , MyStream (drop (tokensLength pxy (t:[])) str) ts
    )
  takeN_ n (MyStream str s)
    | n <= 0    = Just ([], MyStream str s)
    | null s    = Nothing
    | otherwise =
        let (x, s') = splitAt n s
        in case NE.nonEmpty x of
          Nothing -> Just (x, MyStream str s')
          Just nex -> Just (x, MyStream (drop (tokensLength pxy nex) str) s')
  takeWhile_ f (MyStream str s) =
    let (x, s') = DL.span f s
    in case NE.nonEmpty x of
      Nothing -> (x, MyStream str s')
      Just nex -> (x, MyStream (drop (tokensLength pxy nex) str) s')
```

```haskell
instance VisualStream MyStream where
  showTokens Proxy = DL.intercalate " "
    . NE.toList
    . fmap (showMyToken . tokenVal)
  tokensLength Proxy xs = sum (tokenLength <$> xs)

instance TraversableStream MyStream where
  reachOffset o PosState {..} =
    ( Just (prefix ++ restOfLine)
    , PosState
        { pstateInput = MyStream
            { myStreamInput = postStr
            , unMyStream = post
            }
        , pstateOffset = max pstateOffset o
        , pstateSourcePos = newSourcePos
        , pstateTabWidth = pstateTabWidth
        , pstateLinePrefix = prefix
        }
    )
    where
      prefix =
        if sameLine
          then pstateLinePrefix ++ preLine
          else preLine
      sameLine = sourceLine newSourcePos == sourceLine pstateSourcePos
      newSourcePos =
        case post of
          [] -> pstateSourcePos
          (x:_) -> startPos x
      (pre, post) = splitAt (o - pstateOffset) (unMyStream pstateInput)
      (preStr, postStr) = splitAt tokensConsumed (myStreamInput pstateInput)
      preLine = reverse . takeWhile (/= '\n') . reverse $ preStr
      tokensConsumed =
        case NE.nonEmpty pre of
          Nothing -> 0
          Just nePre -> tokensLength pxy nePre
      restOfLine = takeWhile (/= '\n') postStr

pxy :: Proxy MyStream
pxy = Proxy

showMyToken :: MyToken -> String
showMyToken = \case
  (Int n)    -> show n
  Plus       -> "+"
  Mul        -> "*"
  Div        -> "/"
```

```
  OpenParen  -> "("
  CloseParen -> ")"
```

More background information about the `Stream` type class (and why it looks like this) can be found in this blog post. Note that in the version 9 of `megaparsec` certain methods of `Stream` were moved to the classes `VisualStream` and `TraversableStream` to make it easier to define instances of `Stream` for certain custom input streams.

Now we can define `Parser` for our custom stream:

```
type Parser = Parsec Void MyStream
```

The next step is to define basic parsers on top of `token` and (if it makes sense) `tokens` primitives. For the streams that are supported out-of-the-box we have `Text.Megaparsec.Byte` and `Text.Megaparsec.Char` modules, but if we are to work with custom tokens, we need custom helpers.

```
liftMyToken :: MyToken -> WithPos MyToken
liftMyToken myToken = WithPos pos pos 0 myToken
  where
    pos = initialPos ""

pToken :: MyToken -> Parser MyToken
pToken c = token test (Set.singleton . Tokens . nes . liftMyToken $ c)
  where
    test (WithPos _ _ _ x) =
      if x == c
        then Just x
        else Nothing
    nes x = x :| []

pInt :: Parser Int
pInt = token test Set.empty <?> "integer"
  where
    test (WithPos _ _ _ (Int n)) = Just n
    test _ = Nothing
```

Finally let us have a test parser which parses a sum:

```
pSum :: Parser (Int, Int)
pSum = do
  a <- pInt
  _ <- pToken Plus
  b <- pInt
  return (a, b)
```

And an example input for it:

```
exampleStream :: MyStream
exampleStream = MyStream
```

```
  "5 + 6"
  [ at 1 1 (Int 5)
  , at 1 3 Plus         -- (1)
  , at 1 5 (Int 6)
  ]
  where
    at  l c = WithPos (at' l c) (at' l (c + 1)) 2
    at' l c = SourcePos "" (mkPos l) (mkPos c)
```

Let's try it:

```
λ> parseTest (pSum <* eof) exampleStream
(5,6)
```

If we change `Plus` on the line (1) to `Div`, we will get the correct parse error:

```
λ> parseTest (pSum <* eof) exampleStream
1:3:
  |
1 | 5 + 6
  |   ^^
unexpected /
expecting +
```

In other words, we have now a fully functional parser that parses a custom stream.

1. There is actually the `modern-uri` package which contains a real-world Megaparsec parser which can parse URIs according to RFC 3986. The parser from the package is much more complex than the one we describe here, though. ↩