More on Parsing

Haskell and Cryptocurrencies

Dr. Lars Brünjes, IOHK
Alejandro Garcia, IOHK
Dr. Andres Löh, Well-Typed LLP

2020-08-12

INPUT | OUTPUT

Goals

- The `MonadPlus` class
- Grammar transformations
- Parsing sequences
- Operator precedence
- Parsec

Credits

This lecture is based on Johan Jeuring's lecture on "Languages and Compilers", Utrecht University, 2016–2017.

All errors are of course our own.

Alternative and MonadPlus

Reminder of the last lecture

```haskell
newtype Parser t a = Parser
  {runParser :: [t] -> [(a, [t])]}
```

```haskell
(<$>) :: (a -> b) -> Parser t a -> Parser t b
(<$)  :: a -> Parser t b -> Parser t a
```

```haskell
pure  :: a -> Parser t a
(<*>) :: Parser t (a -> b) -> Parser t a
           -> Parser t b
(<*)  :: Parser t a -> Parser t b -> Parser t a
(*>)  :: Parser t a -> Parser t b -> Parser t b
```

Reminder of the last lecture

```haskell
newtype Parser t a = Parser
  {runParser :: [t] -> [(a, [t])]}
```

```haskell
empty    :: Parser t a
(<|>)    :: Parser t a -> Parser t a -> Parser t a
many     :: Parser t a -> Parser t [a]
some     :: Parser t a -> Parser t [a]
optional :: Parser t a -> Parser t (Maybe a)
```

Reminder of the last lecture

```haskell
newtype Parser t a = Parser
  {runParser :: [t] -> [(a, [t])]}
```

```haskell
satisfy :: (t -> Bool) -> Parser t t
```

```haskell
token :: Eq t => t -> Parser t ()
```

```haskell
eof :: Parser t ()
```

```haskell
digit :: Parser Char Char
```

```haskell
letter :: Parser Char Char
```

`Alternative` for `Maybe`

Class `Alternative` is not only useful for parsing. Consider the following example:

```
type Name  = String
type Phone = String
```

```
data Person = Person
  { personName      :: Name
  , personHomePhone :: Maybe Phone
  , personWorkPhone :: Maybe Phone
  }
  deriving Show
```

`Alternative` for `Maybe` (contd.)

```haskell
phone :: Person -> Maybe Phone
phone p = case personHomePhone p of
  Just x   -> Just x
  Nothing -> personWorkPhone p
```

If the person has a home phone, we return that. Alternatively, we try to return the work phone.

`Alternative` for `Maybe` (contd.)

Type `Maybe` is an instance of `Alternative`, too:

```haskell
instance Alternative Maybe where
  empty :: Maybe a
  empty = Nothing
  (<|>) :: Maybe a -> Maybe a -> Maybe a
  Just a  <|> _ = Just a
  Nothing <|> b = b
```

`Alternative` for `Maybe` (contd.)

Now we can rewrite `phone`:

```
phone :: Person -> Maybe Phone
phone p = personHomePhone p <|> personWorkPhone p
```

`Alternative` for lists

Lists are `Alternative`, too:

```
instance Alternative [] where
  empty :: [a]
  empty = []
  (<|>) :: [a] -> [a] -> [a]
  (<|>) = (++)
```

`guard`

For instances of `Alternative`, a very useful function is defined in `Control.Monad`:

```
guard :: Alternative f => Bool -> f ()
guard False = empty
guard True  = pure ()
```

You can use it like this:

```
myFilter :: (a -> Bool) -> [a] -> [a]
myFilter p xs = do
  x <- xs
  guard (p x)
  return x
```

`MonadPlus`

There is another, similar class defined in `Control.Monad`:

```
class (Alternative m, Monad m)
   => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a
```

- `Alternative` is to `Applicative` as `MonadPlus` is to `Monad`.
- Often we have

```
mzero = empty
mplus = (<|>)
```

- `Maybe` and lists are instances of `MonadPlus`, too.

Parsers are `Monad` and `MonadPlus`

```
newtype Parser t a = Parser
  {runParser :: [t] -> [(a, [t])]}
```

Parsers are monads, too!

```
instance Monad (Parser t) where

  return :: a -> Parser t a
  return a = Parser $ \ ts -> [(a, ts)]

  (>>=) :: Parser t a -> (a -> Parser t b)
        -> Parser t b
  p >>= cont = Parser $ \ ts -> do
    (a, ts') <- runParser p ts
    runParser (cont a) ts'
```

Parsers are `Monad` and `MonadPlus`

```
newtype Parser t a = Parser
  {runParser :: [t] -> [(a, [t])]}
```

And they are `MonadPlus`:

```
instance MonadPlus (Parser t) where

  mzero :: Parser t a
  mzero = Parser $ const []

  mplus :: Parser t a -> Parser t a -> Parser t a
  p `mplus` q = Parser $ \ ts ->
    runParser p ts ++ runParser q ts
```

Grammar Transformations

Our example from last time...

$$S \to D{+}S \mid D$$
$$D \to 0 \mid 1$$

```
data S = Plus D S | Digit D
data D = Zero | One
```

```
parseS :: Parser Char S
parseS =
      Plus  <$> parseD <* token '+' <*> parseS
  <|> Digit <$> parseD
```

```
parseD :: Parser Char D
parseD =
      Zero <$ token '0'
  <|> One  <$ token '1'
```

… slightly(?) changed

$$S \to S\text{-}D \mid D$$

$$D \to 0 \mid 1$$

```
data S = Minus S D | Digit D
data D = Zero | One
```

```
parseS :: Parser Char S
parseS =
      Minus <$> parseS <* token '-' <*> parseD
  <|> Digit <$> parseD
```

```
parseD :: Parser Char D
parseD =
      Zero <$ token '0'
  <|> One  <$ token '1'
```

… slightly(?) changed

$$S \to S\text{-}D \mid D$$

$$D \to 0 \mid 1$$

```haskell
data S = Minus S D | Digit D
data D = Zero | One
```

```
GHCi> runParser parseS "1-0-1"
… infinite loop!
```

What's going on?

Left recursion

- A production is called left-recursive if the right hand side starts with the nonterminal on the left hand side.
- Example: production $S \to S\texttt{-}D \mid D$ from the last slide.
- A grammar is called left-recursive if there is a derivation $A \Rightarrow \ldots \Rightarrow Az$ for some nonterminal A of the grammar.
- Grammars can be indirectly left-recursive, i.e. without having a left-recursive production.

Left recursion and parsers

- A left-recursive production $A \rightarrow Az$ corresponds to a parser `a = a <*> z`.
- Such a parser loops!
- Removing left-recursion from grammars is essential for combinator parsing!

Removing left recursion

- Transforming a (directly) left-recursive nonterminal A such that the left-recursion is removed is relatively simple:
- First split the productions for A into left-recursive ones and others:

$$
\begin{aligned}
A &\rightarrow Ax_1 \mid Ax_2 \mid \ldots \mid Ax_n \\
A &\rightarrow y_1 \mid y_2 \mid \ldots \mid y_m
\end{aligned}
$$

- This grammar can be transformed to:

$$
\begin{aligned}
A &\rightarrow y_1 \mid y_1Z \mid y_2 \mid y_2Z \mid \ldots \mid y_m \mid y_mZ \\
Z &\rightarrow x_1 \mid x_1Z \mid x_2 \mid x_2Z \mid \ldots \mid x_n \mid x_nZ
\end{aligned}
$$

Removing left recursion (example)

- Let's try this for our left-recursive example nonterminal S!
- There is one left-recursive production and one other (so $n = m = 1$):

$$S \rightarrow S\texttt{-}D$$
$$S \rightarrow D$$

- So as transformation we get:

$$S \rightarrow D \mid DZ$$
$$Z \rightarrow \texttt{-}D \mid \texttt{-}DZ$$

Removing left recursion (example contd.)

$$S \to D \mid DZ$$
$$Z \to \texttt{-}D \mid \texttt{-}DZ$$
$$D \to 0 \mid 1$$

```
data S = Digit D | Minus D Z
data Z = Digit' D | Minus' D Z
data D = Zero | One
```

```
parseS =
      Digit <$> parseD
  <|> Minus <$> parseD <*> parseZ
parseZ =
      Digit' <$> (token '-' *> parseD)
  <|> Minus' <$> (token '-' *> parseD) <*> parseZ
parseD = Zero <$ token '0' <|> One <$ token '1'
```

Removing left recursion (example contd.)

$S \to D \mid DZ$

$Z \to \texttt{-}D \mid \texttt{-}DZ$

$D \to 0 \mid 1$

```
data S = Digit D | Minus D Z
data Z = Digit' D | Minus' D Z
data D = Zero | One
```

Now it works:

```
GHCi> runParser (parseS <* eof) "1-0-1"
[(Minus One (Minus' Zero (Digit' One)), "")]
```

Left factoring

- If several productions of a nonterminal in a grammar have a common prefix, we can perform left factoring.
- The longer the common prefix and the more productions share that prefix, the more useful left factoring becomes.
- Left factoring of a grammar corresponds to optimization of the parser. Depending on the grammar and the parser combinators used, it can be absolutely essential.

Left factoring (example)

- Let's look at our example grammar again:

$$
\begin{aligned}
S &\rightarrow D \mid DZ \\
Z &\rightarrow \texttt{-}D \mid \texttt{-}DZ \\
D &\rightarrow 0 \mid 1
\end{aligned}
$$

- Left factoring on S (common prefix D) yields

$$S \rightarrow D\big[\epsilon \mid Z\big] = D\,Z?$$

- Left factoring on Z (common prefix $\texttt{-}D$) yields

$$Z \rightarrow \texttt{-}D\big[\epsilon \mid Z\big] = \texttt{-}D\,Z?$$

Left factoring (example contd.)

$$
\begin{aligned}
S &\to D\ Z? \\
Z &\to \texttt{-}D\ Z? \\
D &\to 0 \mid 1
\end{aligned}
$$

```haskell
data S = S D (Maybe Z)
data Z = Z D (Maybe Z)
data D = Zero | One
```

```haskell
parseS = S <$> parseD <*> optional parseZ
```

```haskell
parseZ = Z <$> (token '-' *> parseD)
           <*> optional parseZ
```

```haskell
parseD = Zero <$ token '0' <|> One <$ token '1'
```

Left factoring (example contd.)

$$S \rightarrow D\ Z?$$
$$Z \rightarrow \texttt{-}D\ Z?$$
$$D \rightarrow 0 \mid 1$$

```haskell
data S = S D (Maybe Z)
data Z = Z D (Maybe Z)
data D = Zero | One
```

```
GHCi> runParser (parseS <* eof) "1-0-1"
[(S One (Just (Z Zero (Just (Z One Nothing))))
, "")]
```

Transformations without changing the result type

In the previous examples, we changed both the parser and the Haskell representation.

This is not always convenient:

- on the parser side, we aim for maximal efficiency,
- on the Haskell side, we aim for maximal clarity.

Transformations without changing the result type

In the previous examples, we changed both the parser and the Haskell representation.

This is not always convenient:

- on the parser side, we aim for maximal efficiency,
- on the Haskell side, we aim for maximal clarity.

Fortunately, we can apply left recursion removal and left factoring on the grammar, but keep the original Haskell result type.

Transformations without changing the result type (contd.)

$$S \rightarrow D\, Z?$$
$$Z \rightarrow \texttt{-}D\, Z?$$
$$D \rightarrow 0 \mid 1$$

```
data S = Minus S D | Digit D
data D = Zero | One
```

The parser for *D* is unproblematic:

```
parseD :: Parser Char D
parseD  =  Zero <$ token '0'
       <|> One  <$ token '1'
```

For the rest, the key insight is that *Z* represents an *S* from which an *S* is missing, so we give `parseZ` a result of type `S -> S` ...

Transformations without changing the result type (contd.)

```
parseS :: Parser Char S
parseS =   (\ d m -> maybe id ($) m (Digit d))
       <$> parseD
       <*> optional parseZ

parseZ :: Parser Char (S -> S)
parseZ =   (\ y m x -> maybe id ($) m (Minus x y))
       <$> (token '-' *> parseD)
       <*> optional parseZ
```

Transformations without changing the result type (contd.)

```
parseS :: Parser Char S
parseS =   (\ d m -> maybe id ($) m (Digit d))
       <$> parseD
       <*> optional parseZ

parseZ :: Parser Char (S -> S)
parseZ =   (\ y m x -> maybe id ($) m (Minus x y))
       <$> (token '-' *> parseD)
       <*> optional parseZ
```

Testing:

```
GHCi> runParser (parseS <* eof) "1-0-0"
[(Minus (Minus (Digit One) Zero) Zero, "")]
```

Parsing Sequences

Associative separators

- Consider the grammar

$$S \to S\,\mathbf{;}\,S \mid A$$

- The grammar is left-recursive and ambiguous.
- However, we could argue that this is no problem if the intended meaning of the different parse trees is the same, i.e. if **;** is assosiative:

$$(A\,\mathbf{;}\,A)\,\mathbf{;}\,A = A\,\mathbf{;}\,(A\,\mathbf{;}\,A)$$

- For this situation, we can define a special combinator `listOf` that simply collects all elements separated by a given separator into a list.

Associative separators (contd.)

```haskell
listOf :: Parser t a -> Parser t b -> Parser t [a]
listOf p s = (:) <$> p <*> many (s *> p)
```

```
GHCi> runParser (listOf digit (token ';') <* eof)
  "1;2;3;4"
[("1234", "")]
```

Left associative operators

- What if instead of an associative separator, we have a left associative operator (like `+` or `-`)?
- For example, we might want to parse something like `2-3+4` (as an `Int`).
- Idea: Given a `Parser t a` for the operands and a `Parser t (a -> a -> a)` for the operators, we parse the first operand and then `many` operator-operand pairs:

```
chainl :: Parser t a -> Parser t (a -> a -> a)
  -> Parser t a
chainl p s = foldl' (flip ($))
  <$> p
  <*> many (flip <$> s <*> p)
```

Left associative operators (contd.)

To try this out, we need parsers for plus/minus and integers:

```
pm :: Parser Char (Int -> Int -> Int)
pm = (+) <$ token '+' <|> (-) <$ token '-'
```

```
nat :: Parser Char Int
nat = read <$> some digit
```

```
sign :: Parser Char (Int -> Int)
sign = maybe id (const negate)
  <$> optional (token '-')
```

```
int :: Parser Char Int
int = ($) <$> sign <*> nat
```

Left associative operators (contd.)

```
GHCi> runParser (int <* eof) "123"
[(123, "")]
```

```
GHCi> runParser (int <* eof) "-123"
[(- 123, "")]
```

```
GHCi> runParser (chainl int pm <* eof) "2-3+4"
[(3, "")]
```

Right associative operators

For right associative operators, we first parse *many* operand-operator pairs, then a final operand:

```
chainr :: Parser t a -> Parser t (a -> a -> a)
  -> Parser t a
chainr p s = flip (foldr ($))
  <$> many (flip ($) <$> p <*> s)
  <*> p
```

```
GHCi> runParser (chainr int pm <* eof)
  "2-3+4"
[(- 5, "")]
```

Operator precedence

Operator precedence

- Consider the grammar

$$
\begin{aligned}
E &\rightarrow E\texttt{+}E \\
E &\rightarrow E\texttt{-}E \\
E &\rightarrow E\texttt{*}E \\
E &\rightarrow \texttt{(}E\texttt{)} \\
E &\rightarrow Int
\end{aligned}
$$

- This is a typical grammar for expressions with operators.
- For the same reasons as above, this grammar is ambiguous.
- Given the precedence of the operators and their associativity, we can transform the grammar so that the ambiguity is removed.

Operator precedence (contd.)

- The basic idea is to parse operators of different precedences sequentially.
- For each precedence level i we get:

$$E_i \rightarrow E_i \; Op_i \; E_{i+1} \mid E_{i+1} \qquad \text{(for left-associative operators)}$$

or

$$E_i \rightarrow E_{i+1} \; Op_i \; E_i \mid E_{i+1} \qquad \text{(for right-associative operators)}$$

or

$$E_i \rightarrow E_{i+1} \; Op_i \; E_{i+1} \mid E_{i+1} \qquad \text{(for non-associative operators)}$$

- The highest level contains the remaining productions.
- All forms of bracketing point to the lowest level of expressions.

Operator precedence (contd.)

- Applied to

$$
\begin{aligned}
E &\rightarrow E\texttt{+}E \\
E &\rightarrow E\texttt{-}E \\
E &\rightarrow E\texttt{*}E \\
E &\rightarrow \texttt{(}E\texttt{)} \\
E &\rightarrow Int
\end{aligned}
$$

- we obtain

$$
\begin{aligned}
E_1 &\rightarrow E_1\ Op_1\ E_2 \mid E_2 \\
E_2 &\rightarrow E_2\ Op_2\ E_3 \mid E_3 \\
E_3 &\rightarrow \texttt{(}E_1\texttt{)} \mid Int \\
Op_1 &\rightarrow \texttt{+} \mid \texttt{-} \\
Op_2 &\rightarrow \texttt{*}
\end{aligned}
$$

Operator precedence (contd.)

$$
\begin{aligned}
E_1 &\rightarrow E_1\ Op_1\ E_2 \mid E_2 \\
E_2 &\rightarrow E_2\ Op_2\ E_3 \mid E_3 \\
E_3 &\rightarrow (\ E_1\ ) \mid Int \\
Op_1 &\rightarrow \texttt{+} \mid \texttt{-} \\
Op_2 &\rightarrow \texttt{*}
\end{aligned}
$$

```
data E = Plus E E
       | Minus E E
       | Times E E
       | Lit Int
```

```
e1, e2, e3 :: Parser Char E
e1 = chainl e2 op1
e2 = chainl e3 op2
e3 = token '(' *> e1 <* token ')' <|> Lit <$> int
```

```
op1, op2 :: Parser Char (E -> E -> E)
op1 = Plus <$ token '+' <|> Minus <$ token '-'
op2 = Times <$ token '*'
```

A general operator parser

Using `msum` from `Data.Foldable`, we can do even better:

```
msum :: (Foldable t, MonadPlus m) => t (m a) -> m a
```

```
type Op a = (Char, a -> a -> a)
```

```
gen :: [Op a] -> Parser Char a -> Parser Char a
gen ops p = chainl p $
  msum $ map (\ (s, f) -> f <$ token s) ops
```

```
e1 = gen [('+', Plus), ('-', Minus)] e2
e2 = gen [('*', Times)] e3
```

(Mega-)Parsec

Megaparsec

- popular industrial strength parser combinator library, originally written by Daan Leijen, in this version adapted by Mark Karpov
- variants of Parsec have been ported to OCaml, Java, C#, F#, Ruby, Erlang, C++, Python, JavaScript,...
- supports arbitrary token types
- good error messages
- no multiple results: either succeeds or fails with an error message
- restricted choice

Preparations

For the rest of this lecture, we will make use of the following modules in `megaparsec`:

```
import Text.Megaparsec
import Text.Megaparsec.Char
import Text.Megaparsec.Char.Lexer as L
```

The `Parsec` type

```
data Parsec e s a  -- abstract
```

- `e` is the type for customising *errors*;
  we can use the *empty type* `Void`, because the default behaviour of **megaparsec** is already quite good.

The `Parsec` type

```haskell
data Parsec e s a  -- abstract
```

- `e` is the type for customising *errors*;
  we can use the *empty type* `Void`, because the default behaviour of **`megaparsec`** is already quite good.
- `s` is the *stream* type;
  we can use `String`, to parse from plain strings, but `ByteString`, `Text`, or user-defined token streams are also possible.

The `Parsec` type

```
data Parsec e s a  -- abstract
```

- `e` is the type for customising *errors*;
  we can use the *empty type* `Void`, because the default behaviour of **`megaparsec`** is already quite good.
- `s` is the *stream* type;
  we can use `String`, to parse from plain strings, but `ByteString`, `Text`, or user-defined token streams are also possible.
- `a` is the *return* type.

The `Parsec` type – contd.

Our own parser type

```
Parser Char a
```

corresponds to

```
Parsec Void String a
```

Revisiting our standard example

$S \to D{+}S \mid D$

$D \to 0 \mid 1$

```haskell
data S = Plus D S | Digit D
data D = Zero | One
```

```haskell
parseS :: Parsec Void String S  -- not yet ok
parseS =
      Plus  <$> parseD <* single '+' <*> parseS
  <|> Digit <$> parseD
```

```haskell
parseD :: Parsec Void String D
parseD =
      Zero <$ single '0'
  <|> One  <$ single '1'
```

Revisiting our standard example (contd.)

The `Parsec` type is an instance of `Functor`, `Applicative`, `Alternative`, `Monad` and `MonadPlus`, so we can keep using all the standard combinators.

Revisiting our standard example (contd.)

The `Parsec` type is an instance of `Functor`, `Applicative`, `Alternative`, `Monad` and `MonadPlus`, so we can keep using all the standard combinators.

Our old `token` is now called `single`:

```
single :: Char -> Parsec Void String Char
```

(This has a more general type in the library to allow for other token types.)

Running a parser

A convenient driver for a parser is

```
parseTest ::
  Show a => Parsec Void String a -> String -> IO ()
```

This tries to parse the given string, prints the result (and, in case of an error, an error message).

Running a parser

A convenient driver for a parser is

```
parseTest ::
  Show a => Parsec Void String a -> String -> IO ()
```

This tries to parse the given string, prints the result (and, in case of an error, an error message).

Other functions such as `parse` or `parseMaybe` do not force the caller into `IO` and/or provide programmatic access to the error messages.

Running a parser (examples)

```
GHCi> parseTest parseD "0"
Zero
GHCi> parseTest parseD "1"
One
```

Running a parser (examples)

```
GHCi> parseTest parseD "0"
Zero
GHCi> parseTest parseD "1"
One
```

```
GHCi> parseTest parseD "2"
1:1:
  |
1 | 2
  | ^
unexpected '2'
expecting '0' or '1'
```

Pitfall: unconsumed input

Just like our own parsers, we may not consume the entire input:

```
GHCi> parseTest parseD "1+"
One
```

A prefix is successfully parsed; there is *no warning* that we have unconsumed input.

Expecting the end of input

Fortunately, the library also provides a variant of `eof`:

```
eof :: Parsec Void String ()
```

Expecting the end of input

Fortunately, the library also provides a variant of `eof`:

```
eof :: Parsec Void String ()
```

```
GHCi> parseTest (parseD <* eof) "1+"
1:2:
  |
1 | 1+
  |  ^
unexpected '+'
expecting end of input
```

Another problem

Let's try `parseS` rather than `parseD`:

```
GHCi> parseTest (parseS <* eof) "1+1"
1:4:
  |
1 | 1+1
1 |    ^
unexpected end of input
GHCi> parseTest (parseS <* eof) "1"
1:4:
  |
1 | 1+1
1 |    ^
```

What went wrong?

```
parseS :: Parsec Void String S  -- not yet ok
parseS =
      Plus  <$> parseD <* single '+' <*> parseS
  <|> Digit <$> parseD
```

```
parseD :: Parsec Void String D
parseD =
      Zero <$ single '0'
  <|> One  <$ single '1'
```

What went wrong?

```haskell
parseS :: Parsec Void String S  -- not yet ok
parseS =
      Plus  <$> parseD <* single '+' <*> parseS
  <|> Digit <$> parseD
```

```haskell
parseD :: Parsec Void String D
parseD =
      Zero <$ single '0'
  <|> One  <$ single '1'
```

Note that *both branches* in `parseS` accept `'0'` and `'1'` as first token.

Restricted choice

- The `Alternative` instance tries the second alternative only if first failed and didn't consume input.
- This is done for efficiency ($LL(1)$ grammar).

Restricted choice

- The `Alternative` instance tries the second alternative only if first failed and didn't consume input.
- This is done for efficiency ($LL(1)$ grammar).

However, there is an "escape hatch":

```
try ::
  Parsec Void String a -> Parsec Void String a
```

If `try p` *fails*, it "pretends" not to have consumed input.

Restricted choice

- The `Alternative` instance tries the second alternative only if first failed and didn't consume input.
- This is done for efficiency ($LL(1)$ grammar).

However, there is an "escape hatch":

```
try ::
  Parsec Void String a -> Parsec Void String a
```

If `try p` *fails*, it "pretends" not to have consumed input.

Another option in this case is to apply *left factoring*.

Using `try`

```
parseS :: Parsec Void String S
parseS =
      Plus  <$> try (parseD <* single '+')
                        <*> parseS
  <|> Digit <$> parseD
```

```
parseD :: Parsec Void String D
parseD =
      Zero <$ single '0'
  <|> One  <$ single '1'
```

We have to make `try` span enough so that we are certain that this branch applies.

Using `try` – contd.

```
GHCi> parseTest (parseS <* eof) "1+1"
Plus One (Digit One)
```

Applying left factoring

```
parseS :: Parsec Void String S
parseS =
       (\ d -> maybe (Digit d) (Plus d))
   <$> parseD
   <*> optional (single '+' *> parseS)
parseD :: Parsec Void String D
parseD =
       Zero <$ single '0'
   <|> One  <$ single '1'
```

Applying left factoring – contd.

```
GHCi> parseTest (parseS <* eof) "0+1+1"
Plus Zero (Plus One (Digit One))
```

Handling whitespace

All our parsers are unsatisfactory in that they do not handle whitespace:

```
GHCi> parseTest (parseS <* eof) "0 + 1 + 1"
1:2:
  |
1 | 0 + 1 + 1
  |  ^
unexpected space
expecting '+' or end of input
```

Lexeme parsers

Parsec offers *lexeme parsers* that consume any whitespace immediately following the payload.

We have to configure what we want to consider whitespace:

```
space ::
      Parsec Void String ()  -- one or more spaces
   -> Parsec Void String ()  -- a line comment
   -> Parsec Void String ()  -- a block comment
   -> Parsec Void String ()
```

Lexeme parsers

Parsec offers *lexeme parsers* that consume any whitespace immediately following the payload.

We have to configure what we want to consider whitespace:

```
space ::
      Parsec Void String ()  -- one or more spaces
   -> Parsec Void String ()  -- a line comment
   -> Parsec Void String ()  -- a block comment
   -> Parsec Void String ()
```

Simple space consumer, no comments:

```
simpleSpaces :: Parsec Void String ()
simpleSpaces =
  L.space (() <$ some (satisfy isSpace)) empty empty
```

Lexeme parser (contd.)

Parsing a specific text followed by whitespace:

```
symbol ::
  Parsec Void String ()   -- how to consume space
   -> String              -- expected text
   -> Parsec Void String String
```

Lexeme parser (contd.)

Parsing a specific text followed by whitespace:

```
symbol ::
  Parsec Void String ()   -- how to consume space
   -> String              -- expected text
   -> Parsec Void String String
```

A useful abbreviation – our replacement for `single`:

```
sym :: String -> Parsec Void String String
sym = symbol simpleSpaces
```

A whitespace-aware version of our parser

```
parseS :: Parsec Void String S
parseS =
      (\ d -> maybe (Digit d) (Plus d))
  <$> parseD
  <*> optional (sym "+" *> parseS)
parseD :: Parsec Void String D
parseD =
      Zero <$ sym "0"
  <|> One  <$ sym "1"
```

A whitespace-aware version of our parser (contd.)

To allow whitespace at the very beginning of our input, we have to add an extra occurrence of `simpleSpaces`:

```
GHCi> wrap p = simpleSpaces *> p <* eof
GHCi> parseTest (wrap parseS) " 0 +      1+ 1 "
Plus Zero (Plus One (Digit One))
```

Adapting error messages

- As we've seen, the error messages in `megaparsec` are far better already than what we got from our own simple parser combinators.
- In case of a parse error, we get the line number and a description of the error.
- The error messages can be adapted further (and even augmented with custom data, which is what the `e` parameter of the `Parsec` type is for – but we won't cover this here).

Adapting error messages (contd.)

The combinator

```
(<?>) ::
      Parsec Void String a
   -> String
   -> Parsec Void String a
```

provides a “description” for the wrapped parser. The description will then be used in error messages.

Adapting error messages (contd.)

```
parseD :: Parsec Void String D
parseD =
      Zero <$ sym "0"
  <|> One  <$ sym "1"
```

```
parseD' :: Parsec Void String D
parseD' = parseD <?> "binary digit"
```

Adapting error messages (contd.)

```
GHCi> parseTest parseD "9"
1:1:
  |
1 | 9
  |
unexpected '9'
expecting '0' or '1'
GHCi> parseTest parseD' "9"
1:1:
  |
1 | 9
  |
unexpected '9'
expecting binary digit
```

Summary

- Many more advanced parser combinators can be defined in terms of simpler ones (e.g. for parsing operator tables).
- Grammar transformations, in particular left recursion removal and left factoring can be used to work around problematic or inefficient parsers.
- All the idea discussed are transferrable to far more advanced and efficient implementations such as `megaparsec` (many other options with slightly different emphasis exist).
- Then we can easily get whitespace-aware parsers and good error messages as well.
- One major pitfall of `megaparsec` (and the original Parsec) is that sometimes, `try` is needed in choices if multiple branches can start with the same characters.