Grammars & Parser Combinators

Haskell and Cryptocurrencies

Dr. Lars Brünjes, IOG
Robertino Martinez, IOG
Karina Lopez, IOG

August, 2023

Goals

- Grammars
- Parse trees
- Parser combinators
- The `Alternative` class

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.

Grammars

Alphabets & languages

Alphabet

An alphabet is a finite set of symbols (for example the set of all UTF8-characters, corresponding to the type `Char` in Haskell).

Alphabets & languages

Alphabet

An alphabet is a finite set of symbols (for example the set of all UTF8-characters, corresponding to the type `Char` in Haskell).

Language

A language over an alphabet is a subset of all words/sentences over the alphabet (sequences of symbols from the alphabet).

Example: palindromes

The language PAL of palindromes over the alphabet $\{a, b, c\}$ is defined as follows (ϵ denotes the empty word):

- ϵ is in PAL,
- a, b and c are in PAL,
- If P is in PAL, then aPa, bPb and cPc are also in PAL.

Grammars

Grammars

A grammar is a formalism to describe a language inductively.
Grammars consist of rewrite rules, called productions.

A grammar for palindromes

$$
\begin{aligned}
P &\rightarrow \epsilon \\
P &\rightarrow a \\
P &\rightarrow b \\
P &\rightarrow c \\
P &\rightarrow aPa \\
P &\rightarrow bPb \\
P &\rightarrow cPc
\end{aligned}
$$

The language PAL is defined as follows:

- ϵ is in PAL,
- a, b and c are in PAL,
- If P is in PAL, then aPa, bPb and cPc are also in PAL.

A grammar for palindromes

$$
\begin{aligned}
P &\rightarrow \epsilon \\
P &\rightarrow a \\
P &\rightarrow b \\
P &\rightarrow c \\
P &\rightarrow aPa \\
P &\rightarrow bPb \\
P &\rightarrow cPc
\end{aligned}
$$

- A grammar consists of multiple productions. Productions can be seen as rewrite rules. If the left hand side matches, it can be replaced by the right hand side.
- The grammar uses auxiliary symbols – called nonterminals – that are not in the alphabet and hence can't appear in the final word.
- The symbols from the alphabet are also called terminals.

A grammar for palindromes

Starting from a nonterminal, we can apply productions successively until we reach a word of terminals:

$$
\begin{aligned}
P &\to \epsilon \\
P &\to a \\
P &\to b \\
P &\to c \\
P &\to aPa \\
P &\to bPb \\
P &\to cPc
\end{aligned}
$$

$$
\begin{aligned}
& P \\
\Rightarrow\ & aPa \\
\Rightarrow\ & acPca \\
\Rightarrow\ & accPcca \\
\Rightarrow\ & accbcca
\end{aligned}
$$

We call such a sequence a derivation. All words that can be derived from a nonterminal are in the language generated by the nonterminal. The nonterminal is called the start symbol of the language.

Context-free grammars

The grammars we consider are restricted:

- The left hand side of a production always consists of a single nonterminal.

Grammars with this restriction are called context-free.

Remarks about grammars

- Not all languages can be generated by a grammar.
- Even fewer languages can be generated by a context-free grammar.
- Languages that *can* be generated by a context-free grammar are called context-free languages.
- Context-free languages are relatively easy to deal with algorithmically, and therefore most programming languages have a context-free syntax.
- Multiple grammars may generate the same language.

Language of single digits

$$
\begin{aligned}
Dig &\rightarrow 0 & \qquad Dig &\rightarrow 5\\
Dig &\rightarrow 1 & \qquad Dig &\rightarrow 6\\
Dig &\rightarrow 2 & \qquad Dig &\rightarrow 7\\
Dig &\rightarrow 3 & \qquad Dig &\rightarrow 8\\
Dig &\rightarrow 4 & \qquad Dig &\rightarrow 9
\end{aligned}
$$

Language of single digits

$$\begin{aligned} \mathit{Dig} &\to 0 \\ \mathit{Dig} &\to 1 \\ \mathit{Dig} &\to 2 \\ \mathit{Dig} &\to 3 \\ \mathit{Dig} &\to 4 \end{aligned} \qquad \begin{aligned} \mathit{Dig} &\to 5 \\ \mathit{Dig} &\to 6 \\ \mathit{Dig} &\to 7 \\ \mathit{Dig} &\to 8 \\ \mathit{Dig} &\to 9 \end{aligned}$$

Multiple productions for the same nonterminal can be joined:

$$\mathit{Dig} \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

Sequences of digits & natural numbers

$$Digs \rightarrow \epsilon \mid Dig\ Digs$$

Sequences of digits & natural numbers

$$\mathit{Digs} \rightarrow \epsilon \mid \mathit{Dig}\ \mathit{Digs}$$

This grammar allows sequences with leading zeroes:

$$\begin{aligned} \mathit{Digs} &\Rightarrow \mathit{Dig}\ \mathit{Digs} \Rightarrow \mathit{Dig}\ \mathit{Dig}\ \mathit{Digs} \Rightarrow \mathit{Dig}\ \mathit{Dig}\ \mathit{Dig}\ \mathit{Digs} \\ &\Rightarrow \mathit{Dig}\ \mathit{Dig}\ \mathit{Dig}\ \epsilon \Rightarrow \ldots \Rightarrow 007 \end{aligned}$$

Sequences of digits & natural numbers

$$Digs \rightarrow \epsilon \mid Dig\ Digs$$

This grammar allows sequences with leading zeroes:

$$\begin{aligned} Digs &\Rightarrow Dig\ Digs \Rightarrow Dig\ Dig\ Digs \Rightarrow Dig\ Dig\ Dig\ Digs \\ &\Rightarrow Dig\ Dig\ Dig\ \epsilon \Rightarrow \ldots \Rightarrow 007 \end{aligned}$$

We allow the following star notation on the right hand side of a production to abbreviate zero or more occurences of a symbol:

$$Digs \rightarrow Dig^*$$

Sequences of digits & natural numbers

$$Digs \rightarrow \epsilon \mid Dig\ Digs$$

This grammar allows sequences with leading zeroes:

$$\begin{aligned} Digs &\Rightarrow Dig\ Digs \Rightarrow Dig\ Dig\ Digs \Rightarrow Dig\ Dig\ Dig\ Digs \\ &\Rightarrow Dig\ Dig\ Dig\ \epsilon \Rightarrow \ldots \Rightarrow 007 \end{aligned}$$

We allow the following star notation on the right hand side of a production to abbreviate zero or more occurences of a symbol:

$$Digs \rightarrow Dig^*$$

To disallow leading zeroes, we define non-zero digits:

$$\begin{aligned} Dig_{nz} &\rightarrow 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\ Nat &\rightarrow 0 \mid Dig_{nz}\ Dig^* \end{aligned}$$

Integers

$$
\begin{aligned}
\mathit{Sign} &\rightarrow \texttt{+} \mid \texttt{-} \\
\mathit{Int} &\rightarrow \mathit{Sign}\ \mathit{Nat} \mid \mathit{Nat}
\end{aligned}
$$

$$\begin{aligned} \mathit{Sign} &\rightarrow \texttt{+} \mid \texttt{-} \\ \mathit{Int} &\rightarrow \mathit{Sign}\ \mathit{Nat} \mid \mathit{Nat} \end{aligned}$$

The sign is optional.

Integers

$$
\begin{aligned}
\mathit{Sign} &\rightarrow \texttt{+} \mid \texttt{-} \\
\mathit{Int} &\rightarrow \mathit{Sign}\ \mathit{Nat} \mid \mathit{Nat}
\end{aligned}
$$

The sign is optional.

There is an abbreviation for optional symbols as well:

$$\mathit{Int} \rightarrow \mathit{Sign}?\ \mathit{Nat}$$

Parse Trees

Parse trees

Consider the grammar $S \to a \mid SS$. The word aaa has (at least) the following derivations:

$$S \Rightarrow SS \Rightarrow aS \Rightarrow aSS \Rightarrow aaS \Rightarrow aaa \tag{1}$$

$$S \Rightarrow SS \Rightarrow Sa \Rightarrow SSa \Rightarrow aSa \Rightarrow aaa \tag{2}$$

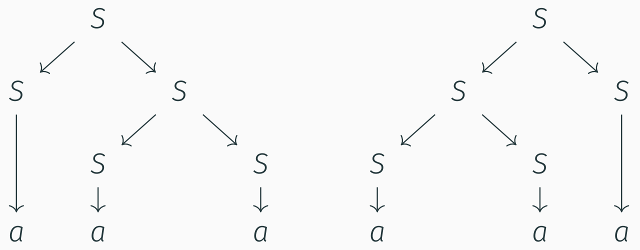
Parse trees

Consider the grammar $S \to a \mid SS$. The word aaa has (at least) the following derivations:

$$S \Rightarrow SS \Rightarrow aS \Rightarrow aSS \Rightarrow aaS \Rightarrow aaa \tag{1}$$

$$S \Rightarrow SS \Rightarrow Sa \Rightarrow SSa \Rightarrow aSa \Rightarrow aaa \tag{2}$$

We can visualize derivations as parse trees:



Ambiguity

- A grammar where every word in the generated language has a unique parse tree is called unambiguous.
- If this is not the case, the grammar is called ambiguous.
- The grammar $S \to a \mid SS$ is thus ambiguous.
- The semantics (i. e. "meaning") of a language will normally be defined via parse trees. Hence ambiguous grammars can have ambiguous semantics!
- Furthermore, ambiguity can also be bad for the *efficiency* of parsing.

Parsing problem

- Given a grammar G with generated language $L(G)$ and a word s, the parsing problem is to decide whether $s \in L(G)$.
- Furthermore, if $s \in L(G)$, we want evidence (a proof, an explanation) why this is the case, usually in the form of a parse tree.

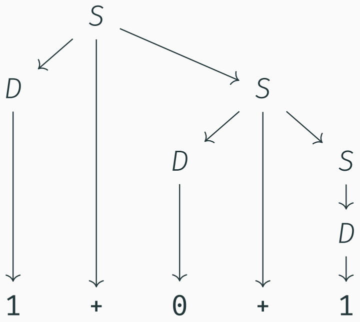
Parse trees in Haskell

Consider this grammar:

$$S \rightarrow D\texttt{+}S \mid D$$

$$D \rightarrow \texttt{0} \mid \texttt{1}$$

The word `1+0+1` has the parse tree



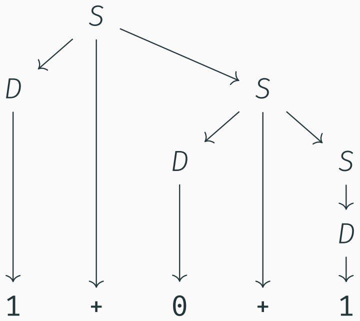
Parse trees in Haskell

Consider this grammar:

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

$$D \to \texttt{0} \mid \texttt{1}$$

The word `1+0+1` has the parse tree



How do we best represent such a tree in Haskell?

Parse trees in Haskell (contd.)

Idea

Let us represent nonterminals as datatypes:

- In any node of the parse tree, we have a choice between the productions of the nonterminal in question.
- If we want to build a value of a Haskell datatype, we have a choice between any of that datatype's constructors.

Parse trees in Haskell (contd.)

$S \to D$`+`$S \mid D$

$D \to$ `0` $\mid$ `1`

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

- We create constructors (with somewhat meaningful names) for each production.
- *Nonterminals* on the right hand side of a production turn into constructor arguments.
- *Terminals* on the right hand side of a production can be dropped – we can reconstruct them.

Concrete and abstract syntax

Both the grammar and the Haskell datatype describe the language.

concrete syntax

$S \rightarrow D\texttt{+}S \mid D$

$D \rightarrow \texttt{0} \mid \texttt{1}$

abstract syntax

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

Concrete and abstract syntax

Both the grammar and the Haskell datatype describe the language.

concrete syntax

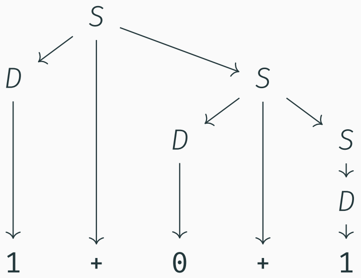
$S \rightarrow D\texttt{+}S \mid D$

$D \rightarrow \texttt{0} \mid \texttt{1}$

abstract syntax

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

The word `1+0+1` corresponds to the parse tree




```haskell
Plus
  One
  (Plus
    Zero
    (Digit One))
```

Semantic functions

concrete syntax

$S \rightarrow D$+S | D

$D \rightarrow$ 0 | 1

abstract syntax

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

Starting from the abstract syntax, we can give *meaning/ semantics* to a language, possibly in different ways.

Semantic functions

concrete syntax

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

$D \to \texttt{0} \mid \texttt{1}$

abstract syntax

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

String representation:

```
printS :: S -> String
printS (Plus d s) = printD d ++ "+" ++ printS s
printS (Digit d)  = printD d

printD :: D -> String
printD Zero = "0"
printD One  = "1"
```

```
GHCi> printS (Plus One (Plus Zero (Digit One)))
"1+0+1"
```

Semantic functions

concrete syntax

$S \rightarrow D$+S | D

$D \rightarrow$ 0 | 1

abstract syntax

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

Evaluation:

```
evalS :: S -> Int
evalS (Plus d s) = evalD d + evalS s
evalS (Digit d)  = evalD d

evalD :: D -> Int
evalD Zero = 0
evalD One  = 1
```

```
GHCi> evalS (Plus One (Plus Zero (Digit One)))
2
```

Parser Combinators

Approaches to parsing

Parser generators

Parser combinators

Approaches to parsing

Parser generators
external program

Parser combinators
library

Approaches to parsing

Parser generators
external program
bottom-up algorithm

Parser combinators
library
top-down algorithm

Approaches to parsing

Parser generators

external program

bottom-up algorithm

complex theory

Parser combinators

library

top-down algorithm

simple underlying theory

Approaches to parsing

Parser generators

external program

bottom-up algorithm

complex theory

limited look-ahead
(usually one token)

Parser combinators

library

top-down algorithm

simple underlying theory

unlimited look-ahead
(in principle)

Approaches to parsing

Parser generators	Parser combinators
external program	library
bottom-up algorithm	top-down algorithm
complex theory	simple underlying theory
limited look-ahead (usually one token)	unlimited look-ahead (in principle)
only built-in abstractions	user-definable abstractions

Approaches to parsing

Parser generators	Parser combinators
external program	library
bottom-up algorithm	top-down algorithm
complex theory	simple underlying theory
limited look-ahead (usually one token)	unlimited look-ahead (in principle)
only built-in abstractions	user-definable abstractions
very fast generated parsers	fast (for most constructs)

Approaches to parsing

Parser generators	Parser combinators
external program	library
bottom-up algorithm	top-down algorithm
complex theory	simple underlying theory
limited look-ahead (usually one token)	unlimited look-ahead (in principle)
only built-in abstractions	user-definable abstractions
very fast generated parsers	fast (for most constructs)

Both approaches place certain (but different) constraints on the grammars.

Aside: Combinators

- The term combinator denotes a self-contained function in lambda calculus, the formal system that Haskell and other functional programming languages are based upon.
- Parser combinators are thus a set of (small) library functions that can be used to construct parsers.

Lexing and parsing

Often, parsing is split into two phases:

Lexing

In a first phase, whitespace and comments are removed, and the input is organized into a list of tokens – small entities that belong together like keywords, identifiers or operators.

Parsing

In the second phase, an abstract syntax tree is constructed from the list of tokens rather than from the original list of characters.

Lexing and parsing (contd.)

In the world of generators, lexing and parsing are often performed by different generators. For example:

	Haskell	C
Lexer	alex	flex
Parser	happy	yacc / bison

Lexing and parsing (contd.)

In the world of generators, lexing and parsing are often performed by different generators. For example:

	Haskell	C
Lexer	alex	flex
Parser	happy	yacc / bison

With parser combinators, there are different options:

- Use only one phase,
- use the same parser combinators for both phases,
- use dedicated lexer combinators for lexing,
- use a hand-writen special-purpose lexer,
- combine a lexer-generator with parser combinators.

Choosing the right parser type

What Haskell type should a parser have?

First attempt: predicate on strings

```haskell
type Parser = String -> Bool
```

We can write simple parsers with this type:

```haskell
digit :: Parser
digit [c]       = c `elem` "0123456789"
digit otherwise = False
```

```haskell
eof :: Parser
eof = null
```

First attempt: predicate on strings

```haskell
type Parser = String -> Bool
```

We can write simple parsers with this type:

```haskell
digit :: Parser
digit [c]          = c `elem` "0123456789"
digit otherwise = False
```

```haskell
eof :: Parser
eof = null
```

Problem

We can't combine parsers of this type! How would we for example use `digit` to write a parser for *two* digits?

Second attempt: keep unconsumed input

```haskell
type Parser = String -> Maybe String
```

```haskell
digit :: Parser
digit []                          = Nothing
digit (c : cs)
  | c `elem` "0123456789" = Just cs
  | otherwise                 = Nothing
```

```haskell
eof :: Parser
eof []      = Just []
eof (_ : _) = Nothing
```

Second attempt: keep unconsumed input

```haskell
type Parser = String -> Maybe String
```

Now we can sequence parsers:

```haskell
combine :: Parser -> Parser -> Parser
combine p1 p2 s = do
  s' <- p1 s
  p2 s'
```

```
GHCi> digit "Asante"
Nothing
GHCi> digit "123"
Just "23"
GHCi> (digit `combine` digit) "123"
Just "3"
```

Second attempt: keep unconsumed input

```haskell
type Parser = String -> Maybe String
```

Let's define a parser for letters, too:

```haskell
letter :: Parser
letter []                                  = Nothing
letter (c : cs)
  | c `elem` ['a' .. 'z'] ++ ['A' .. 'Z'] = Just cs
  | otherwise                              = Nothing
```

Second attempt: keep unconsumed input

```haskell
type Parser = String -> Maybe String
```

Or better, let's abstract the common pattern:

```haskell
satisfy :: (Char -> Bool) -> Parser
satisfy p []   = Nothing
satisfy p (c : cs)
  | p c        = Just cs
  | otherwise = Nothing
```

```haskell
digit = satisfy (`elem`"0123456789")
```

```haskell
letter = satisfy
  (`elem` ['a' .. 'z'] ++ ['A' .. 'Z'])
```

Second attempt: keep unconsumed input

```haskell
type Parser = String -> Maybe String
```

Using `satisfy`, we can define a parser for a specific character:

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

```
GHCi> char 'x' "xyz"
Just "yz"
```

Second attempt: keep unconsumed input

```haskell
type Parser = String -> Maybe String
```

We can even define the *-combinator, called `many` in Haskell:

```haskell
many :: Parser -> Parser
many p s = case p s of
  Nothing -> Just s
  Just s'  -> many p s'
```

```
GHCi> many letter "123"
Just "123"
GHCi> many letter "abc123"
Just "123"
```

Second attempt: keep unconsumed input

```haskell
type Parser = String -> Maybe String
```

```
GHCi> (many letter `combine` char 'a') "xyzab"
Nothing
```

Problem

But what about the grammar $S \to \mathit{letter}^{*}\ a$? With this type for parsers, we only ever get at most one result, but we need to consider *all* possible results to handle cases like these.

Third attempt: returning all possibilities

```haskell
type Parser = String -> [String]
```

```haskell
satisfy :: (Char -> Bool) -> Parser
satisfy p [] = []
satisfy p (c : cs)
  | p c         = [cs]
  | otherwise = []
```

```haskell
eof :: Parser
eof []      = [[]]
eof (_ : _) = []
```

We define `digit`, `letter` and `char` as before in terms of `satisfy`.

Third attempt: returning all possibilities

```haskell
type Parser = String -> [String]
```

```haskell
combine :: Parser -> Parser -> Parser
combine p q s = do
  s' <- p s
  q s'
```

(Same code, but now in the list-monad!)

```haskell
many :: Parser -> Parser
many p s = s : combine p (many p) s
```

```
GHCi> many letter "abc123"
["abc123", "bc123", "c123", "123"]
```

Third attempt: returning all possibilities

```
type Parser = String -> [String]
```

This solves our problem with the previous attempt:

```
GHCi> (many letter `combine` char 'a') "xyzab"
["b"]
```

Third attempt: returning all possibilities

```haskell
type Parser = String -> [String]
```

With this definition for parsers, we can *choose* between parsers:

```haskell
choose :: Parser -> Parser -> Parser
choose p q s = p s ++ q s
```

```
GHCi> (letter `choose` digit) "abc"
["bc"]
GHCi> (letter `choose` digit) "123"
["23"]
```

Third attempt: returning all possibilities

```haskell
type Parser = String -> [String]
```

Problem

There is still one big problem with our definition: We only know whether a given word belongs to the language or not, but we don't get a parse tree or other result.

Let's fix that!

Fourth attempt: returning results

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

We want to make `Parser` an instance of several typeclasses, hence we wrap it in a `newtype`.

```haskell
satisfy :: (Char -> Bool) -> Parser Char
satisfy p = Parser go
  where
    go []            = []
    go (c : cs)
       | p c         = [(c, cs)]
       | otherwise = []
```

Fourth attempt: returning results

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

```haskell
digit, letter :: Parser Char
digit  = satisfy (`elem`"0123456789")
letter = satisfy
  (`elem` ['a'..'z'] ++ ['A'..'Z'])
```

```haskell
eof :: Parser ()
eof = Parser $ \ s -> case s of
  []      -> [((), [])]
  (_ : _) -> []
```

Fourth attempt: returning results

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

What about `char`? We could define.

```
char c = satisfy (== c)
```

But then `char` would produce a parser of type `Parser Char`, where it seems more natural to give the result the type `Parser ()` instead.

We need to be able to change the result type!

Fourth attempt: returning results

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

Let's make `Parser` an instance of `Functor`!

```
instance Functor Parser where

  fmap :: (a -> b) -> Parser a -> Parser b
  fmap f p = Parser $ \ s ->
    [(f a, s') | (a, s') <- runParser p s]
```

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

Fourth attempt: returning results

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

What about sequencing? – What type should

```haskell
Parser a `combine` Parser b
```

have?

`Parser (a, b)` ?

That’s awkward! – Let’s instead make `Parser` an instance of `Applicative`!

Fourth attempt: returning results

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

```
instance Applicative Parser where
  pure :: a -> Parser a
  pure a = Parser $ \ s -> [(a, s)]
  (<*>) :: Parser (a -> b) -> Parser a -> Parser b
  p <*> q = Parser $ \ s -> do
    (f, s')  <- runParser p s
    (a, s'') <- runParser q s'
    return (f a, s'')
```

Fourth attempt: returning results

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

The `Functor` and `Applicative` instances immediately give us other useful combinators:

```
(<$) :: a -> Parser b -> Parser a
(<*) :: Parser a -> Parser b -> Parser a
(*>) :: Parser a -> Parser b -> Parser b
```

These are useful when we don’t care about some of the intermediate results:

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

Fourth attempt: returning results

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

What about choice? – There is a suitable typeclass for this in `Control.Applicative`, too, that we haven't seen yet:

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

Fourth attempt: returning results

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

With `Alternative`, we get some useful combinators from the base libraries for free:

```haskell
many, some :: Alternative f => f a -> f [a]
```

(`many` means "zero or more occurences", the *-operator, `some` means "one or more occurences".)

Furthermore, we get

```haskell
optional :: Alternative f => f a -> f (Maybe a)
```

for optional values.

Fourth attempt: returning results

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

```haskell
instance Alternative Parser where

  empty :: Parser a
  empty = Parser $ const []

  (<|>) :: Parser a -> Parser a -> Parser a
  p <|> q = Parser $ \ s ->
    runParser p s ++ runParser q s
```

Example parser

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

$$D \to \texttt{0} \mid \texttt{1}$$

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

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

```haskell
parseD :: Parser D
parseD = Zero <$ char '0'
  <|> One <$ char '1'
```

Example parser

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

$D \to \texttt{0} \mid \texttt{1}$

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

```
GHCi> runParser parseS "1+0+1"
[ (Plus One (Plus Zero (Digit One)), "")
, (Plus One (Digit Zero), "+1")
, (Digit One, "+0+1")]
```

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

Fifth and final attempt: Other token types

In order to handle other tokens besides characters, we can do one more generalization:

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

```
satisfy :: (t -> Bool) -> Parser t t
satisfy p = Parser go
  where
    go []           = []
    go (t : ts)
      | p t         = [(t, ts)]
      | otherwise = []
```

Fifth and final attempt: Other token types

In order to handle other tokens besides characters, we can do one more generalization:

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

```
eof :: Parser t ()
eof = Parser $ \ts -> case ts of
  []      -> [((), [])]
  (_ : _) -> []
```

The `Functor`, `Applicative` and `Alternative` instances can easily be generalized to this setting.

Fifth and final attempt: Other token types

In order to handle other tokens besides characters, we can do one more generalization:

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

And we can generalize `char` to an analoguous function `token` that works on all tokens types (which can be compared for equality):

```
token :: Eq t => t -> Parser t ()
token t = () <$ satisfy (== t)
```