Part II

Chapter 5
Parser Combinators

| 5.1       | The type of parsers      | 5.9       | Arithmetical expressions |
| 5.2       | Elementary parsers       | 5.10      | Generalized expressions  |
| 5.3       | Grammars                 | 5.11      | Monadic parsers          |
| 5.4       | Parser combinators       | 5.12      | Context sensitivity      |
| 5.5       | Parser transformers      | 5.13      | Common traps             |
| 5.6       | Matching parentheses     | 5.14      | Error handling           |
| 5.7       | More parser combinators  | 5.15      | Self application         |
| 5.8       | Analyzing options        |           |                         |

This chapter is an informal introduction to writing parsers in a lazy functional language using parser combinators. The combinators are operators manipulating parse functions. Using these combinators it is possible to write parsers for ambiguous grammars in an elegant way. These parser combinators relay on the possibility to manipulate with higher order functions. Parsers are usually generated by special purpose parser generators, in functional programming languages such a tools appears to be superfluous.

We will start by motivating the definition of the type of parser functions. Using that type, we will be capable to build parsers for the language of ambiguous grammars. Next, we will introduce some elementary parsers that can be used for parsing the terminal symbols of a language. Your knowledge of grammars is shortly refreshed in section 3. In the rest of this chapter we will show some examples and introduce more powerful parser combinators interleaved.

The parsing problem is: given a string, construct a tree that describes the structure of the string according to its grammar. In this chapter we will use the word string for any sequence (list) of input symbols. Do not confuse this with the type `String` in Clean.

Grammars and parsing are one of the success stories of computer science. The context free syntax of languages can be described concise by the BNF-formalism (Backus-Naur Form). Theoretical computer science has developed tests to determine if a BNF-grammar is ambiguous or vacuous. Transformations can be used to make equivalent grammars that are easier to parse. Applied computer science developed parsers compilers that can
turn a high-level specification of a parser into an efficient program. Theoretical computer science can state some properties about grammar formalisms, grammars and parsing algorithms. In this chapter we will introduce tools that enables you to construct recursive descent parsers in an easy way. Recursive descent parsing is a top-down method of syntax analysis which uses a family of functions to process the input. In general we associate one function to each nonterminal in the grammar.

5.1 The type of parsers

In a functional language we can define a datatype Tree. A parser could be implemented by function of the following type:

```plaintext
::Parser :== [Char] -> Tree
```

For parsing substructures, a parser could call other parsers (or itself) recursively. These calls need not only communicate to their result, but also which part of the input string is left unprocessed. The unprocessed input string has to be part of the result of the parser. The two results can be grouped in a tuple. A better definition for the type Parser is thus:

```plaintext
::Parser :== [Char] -> ([Char],Tree)
```

The type Tree, however, is not yet defined. The type of tree that is returned depends on the application. Therefore, it is better to make the parser type into a polymorphic type, by parameterizing it with the type of the parse tree. Thus we abstract from the type of the parse tree at hand, substituting the type variable r for the result type Tree:

```plaintext
::Parser r :== [Char] -> ([Char],r)
```

For example, a parser that returns a structure of type Oak now has type Parser Oak. For parse trees that represent an expression we could define a type Expr, making it possible to develop parsers returning an expression: Parser Expr. Another instance of a parser is a parse function that recognizes a string of digits, and yields the number represented by it as a parse tree. In this case the function is of type Parser Int.

Until now, we have been assuming that every string can be parsed in exactly one way. In general, this need not be the case: it may be that a single string can be parsed in various ways, or that there is no possible way of parsing a string. As another refinement of the type definition, instead of returning one parse tree (and its associated rest string), we let a parser return a list of trees. Each element of the result consists of a tree, paired with the rest string that was left unprocessed while parsing it. The type definition of Parser therefore had better be:

```plaintext
::Parser r :== [Char] -> (([Char],r))
```

If there is just one parsing, the result of the parse function will be a singleton list. If no parsing is possible, the result will be an empty list. In the case of an ambiguous grammar, alternative parsings make up the elements of the result.

This programming style is called the list of successes method [Wadler 85]. It can be used in situations where in other languages you would use backtracking techniques. If only one solution is required rather than all possible solutions, you can take the head of the list of successes. Thanks to lazy evaluation, only those elements that are actually needed are evaluated. Lazy evaluation enables the use of the list of successes method instead of ordinary backtracking without loss of efficiency.

Parsers with the type described so far operate on lists of characters. There is however no reason to be this restrictive. You may imagine a situation in which a preprocessor prepares a list of tokens, which is subsequently parsed. A token represents a collection of coherent input characters. We can represent a token as a list of characters, a String, or some tailor

200 Functional programming in Clean
made datatype. To cater for arbitrary input symbols, as a final refinement of the parser
type we again abstract from a type: that of the elements of the input string. Calling it \( s \),
and the result type \( r \), the type of parsers is defined by:
\[
:\text{Parser } s \rightarrow [([s], r)]
\]
or if you prefer meaningful identifiers over conciseness:
\[
:\text{Parser symbol result } \rightarrow [([\text{symbol}], \text{result})]
\]
We will use this type definition in the rest of this chapter.

5.2 Elementary parsers
We will start quite simply, defining a parse function that just recognizes the symbol \('a'\).
The type of the input string symbols is \( \text{Char} \) in this case, and as a parse tree we also simply use \( \text{Char} \):
\[
symbola :: \text{Parser Char Char}
\]
\[
symbola = p
\]
\[
\begin{align*}
p ['a':xs] &= (xs, 'a') \\
p _ &= []
\end{align*}
\]

The list of successes method immediately pays off, because now we can return an empty
list if no parsing is possible (because the input is empty, or does not start with an \( 'a' \)). In
the same fashion, we can write parsers that recognize other symbols. As always, rather
than defining a lot of closely related functions, it is better to abstract from the symbol to
be recognized by making it an extra parameter of the function. Also, the function can op-
erate on strings other than characters, so that it can be used in other applications than char-
acter oriented ones. The only prerequisite is that the symbols to be parsed can be tested
for equality. In Clean, this is indicated by the \( \text{Eq} \) predicate in the type of the function:
\[
symbol :: s \rightarrow \text{Parser } s s \mid \text{Eq } s
\]
\[
symbol s = p
\]
\[
\begin{align*}
p [x:xs] | s == x &= (xs, x) \\
p _ &= []
\end{align*}
\]
The function \( \text{symbol} \) is a function that, given a symbol \( s \), yields a parser for that symbol.
The parser in turn is a function, too.

We will now define some elementary parsers that can do the work traditionally taken
care of by lexical analyzers. For example, a useful parser is one that recognizes a fixed
string of symbols, such as \( ['begin'] \) or \( ['end'] \). We will call this function \( \text{token} \).
\[
token :: [s] \rightarrow \text{Parser } s [s] \mid \text{Eq } s
\]
\[
token k = p
\]
\[
\begin{align*}
p xs | k == take n xs &= ([\text{drop } n \hspace{1em} xs, k]) \\
n &= \text{length } k
\end{align*}
\]
As in the case of the symbol function we have parameterized this function with the string
to be recognized. Of course, this function is not confined to strings of characters. How-
ever, we do need an equality test on the input string type. This is reflected in the
type of \( \text{token} \).
The function \( \text{token} \) is a generalization of the \( \text{symbol} \) function, in that it recognizes more
than one character. The combinator \( \text{symbol} \) can be written in terms of \( \text{token} \) as:
\[
symbol s = \text{token } [s]
\]
Another generalization of \( \text{symbol} \) is a function which may, depending on the input, return
different parse results. The function \( \text{satisfy} \) is an example of this. Where the \( \text{symbol} \) func-
tion tests for equality to a given symbol, in \( \text{satisfy} \) an arbitrary predicate \( \text{t} \) can be speci-
fied. Again, \( \text{satisfy} \) effectively is a family of parser functions.
satisfy :: (s->Bool) -> Parser s s
satisfy f = p
  where p [x:xs] | f x = [(xs,x)]
        p _            = []

The final generalisation in this line is to add a function to determine the result of a successful parsing:
satisfy2 :: (s->Bool) (s->t) -> Parser s t
satisfy2 f g = p
  where p [x:xs] | f x = [(xs,g x)]
        p _            = []

In grammar theory an empty string is often called epsilon, written as the greek character \( \varepsilon \). In this tradition, we will define a function \( \text{epsilon} \) that parses the empty string. It does not consume any input, and thus always returns an empty list as parse tree and unmodified input.

\[
\text{epsilon} :: \text{Parser} s [r]
\]

\[
\text{epsilon} = p
  \text{where} p \text{xs} = [(\text{xs},[])]
\]

A variation is the function \( \text{succeed} \), that neither consumes input, but does always return a given, fixed value (or parse tree, if you could call the result of processing zero symbols a parse tree)

\[
\text{succeed} :: r -> \text{Parser} s r
\]

\[
\text{succeed} v = p
  \text{where} p \text{xs} = [(\text{xs}, v)]
\]

Of course, \( \text{epsilon} \) can be defined using \( \text{succeed} \):

\[
\text{epsilon'} :: \text{Parser} s [r]
\]

\[
\text{epsilon'} = \text{succeed} []
\]

Dual to the function \( \text{succeed} \) is the function \( \text{fail} \), that fails to recognize any symbol on the input string. It always returns an empty list of successes:

\[
\text{fail} :: \text{Parser} s r
\]

\[
\text{fail} = p
  \text{where} p \text{xs} = []
\]

We will need this trivial parser as a neutral element for \( \text{foldr} \) later. Note the difference with \( \text{epsilon} \), which does have one element in its list of successes (albeit an empty one).

### 5.3 Grammars

A grammar is a formalism to describe syntax. In a grammar we distinguish terminal symbols and nonterminal symbols. The terminal symbols are the elementary building blocks in the grammar. All composed elements are called nonterminal symbols. The grammar describes which constructs are allowed for the nonterminals. The expression \( 1+2*3 \) is composed at the top-level of the terminals \( 1 \) and \( + \) and the nonterminal \( 2*3 \). This nonterminal is composed of the terminals \( 2, \star \) and \( 3 \).

In a grammar we usually write the terminals between quotes. Sequential composition of elements is denoted by juxtaposition. A definition of a nonterminal consists of its name, the symbol ::=, a body and ends in a full stop. Various alternatives inside a body are separated by the | symbol. Optional pieces are placed between the square brackets [ and ]. Zero or more occurrences are indicated by an postfix asterisk. One or more occurrences by a postfix + symbol. An empty alternative is indicated by \( \varepsilon \). Grouping is indicated by { and }.

As example we show a simple grammar for expressions:

\[
\text{Expr} ::= \text{Term} ( \text{Operator} \text{Term} )
\]
The meta-grammar, the grammar of all grammars, is:

Grammar ::= Rule +.
Rule ::= NonTerm ::= 'Exp' '.'.
Exp ::= NonTerm | Term | Exp + | '{' Exp '}' | '[' Exp ']' | Exp '|' Exp | Exp '*' | 'ε'.

The terminals in the meta grammar are the symbols between quotes and NonTerm and Term. Each of them is a non-empty sequence of characters.

5.4 Parser combinators

Using the elementary parsers from above, parsers can be constructed for terminal symbols from a grammar. More interesting are parsers for nonterminal symbols. Of course, you could write these by hand in an ad-hoc way, but it is more convenient to construct them by partially parameterizing higher-order functions.

Important operations on parsers are sequential and alternative composition. We will develop two functions for this, which for notational convenience are defined as operators: <&> for sequential composition, and <|> for alternative composition. Priorities of these operators are defined to minimize parentheses in practical situations: the operator <&> associates to the right and will have priority level 6, whereas the operator <|> has priority level 4. Both operators have two parsers as parameter, and yield a parser as result. By combining the result of the composition with other parsers, you may construct even more involved parsers.

In the definitions below, the functions operate on parsers \( p_1 \) and \( p_2 \). The parser combinators yield a new parser; a function that takes a list of symbols as input and yields the list of possible parsings.

To start, we will write the operator <&> for sequential composition. First \( p_1 \) must be applied to the input. After that, \( p_2 \) is applied to the rest of the input string \( xs_1 \). This rest input is part of the result of \( p_1 \). Because \( p_1 \) yields a list of solutions, we use a list comprehension in which \( p_2 \) is applied to all rest strings in the list:

\[
<\&> \text{ infixr 6 :: (Parser s a) (Parser s b) \rightarrow Parser s (a,b)} \\\n\text{where p xs = } \{
\begin{align*}
& (xs_2, (v_1, v_2)) \\
& (xs_1, v_1) \leftarrow p_1 \; xs \\
& (xs_2, v_2) \leftarrow p_2 \; xs_1
\end{align*}
\}
\]

The result of the function is a list of all possible tuples \((xs_2, (v_1, v_2))\) with rest string \(xs_2\), where \(v_1\) is the parse tree computed by \(p_1\), and where rest string \(xs_1\) is used to let \(p_2\) compute \(v_2\) and \(xs_2\).

Apart from sequential composition we need a parser combinator for representing choice. For this, we have the parser combinator operator <|>:

\[
<|> \text{ infixr 4 :: (Parser s a) (Parser s a) \rightarrow Parser s a} \\\n<|> \text{ p1 p2 = p} \\\n\text{where p xs = p1 xs ++ p2 xs}
\]

Thanks to the list of successes method, both \( p_1 \) and \( p_2 \) yield lists of possible parsings. To obtain all possible successes of choice between \( p_1 \) and \( p_2 \), we only need to concatenate these two lists.
The result of parser combinators is again a parser, which can be combined with other parsers. The resulting parse trees are intricate tuples which reflect the way in which the parsers were combined. Thus, the term parse tree is really appropriate. For example, the parser \( p_{abc} \) defined as
\[
p_{abc} = \text{symbol 'a' } \&\& \text{symbol 'b' } \&\& \text{symbol 'c'}
\]
is of type \( \text{Parser Char (Char,(Char,Char))} \).

Although the tuples clearly describe the structure of the parse tree, it is a problem that we cannot combine parsers in an arbitrary way. For example, it is impossible to alternatively compose the parser \( p \) above with symbol \( 'a' \), because the latter is of type \( \text{Parser Char Char} \), and only parsers of the same type can be composed alternatively. Even worse, it is not possible to recursively combine a parser with itself, as this would result in infinitely nested tuple types. What we need is a way to alter the structure of the parse tree that a given parser returns.

### 5.5 Parser transformers

Apart from the operators \( \&\& \) and \( \|\| \), that combine parsers, we can define some functions that modify or transform existing parsers. We will develop three of them: \( \text{sp} \) lets a given parser neglect initial spaces, \( \text{just} \) transforms a parser into one that insists on empty rest string, and \( \&\&\& \) applies a given function to the resulting parse trees.

The first parser transformer is \( \text{sp} \). It drops spaces from the input, and then applies a given parser:
\[
\text{sp :: (Parser Char a) -> Parser Char a}
\]
\[
\text{sp p = p o dropWhile isSpace}
\]

The second parser transformer is \( \text{just} \). Given a parser \( p \) it yields a parser that does the same as \( p \), but also guarantees that the rest string is empty. It does so by filtering the list of successes for empty rest strings. Because the rest string is the first component of the tuple, the function can be defined as:
\[
\text{just :: (Parser s a) -> Parser s a}
\]
\[
\text{just p = filter (isEmpty o fst) o p}
\]

The most important parser transformer is the one that transforms a parser into a parser which modifies its result value. We will define it as an operator \( \&\&\& \), that applies a given function to the resulting parse trees of a given parser. We have chosen the symbol so that you might pronounce it as apply; the arrow points away from the function towards the argument. Given a parser \( p \) and a function \( f \), the operator \( \&\&\& \) returns a parser that does the same as \( p \), but in addition applies \( f \) to the resulting parse tree. It is most easily defined using a list comprehension:
\[
(<@) \text{ infixl 5 :: (Parser s a) (a->b) -> Parser s b}
\]
\[
(<@) p0 f = p
\]
\[
\text{where p xs = \[(ys, f v) \|\| (ys, v) <- p0 xs\]}
\]

Using this operator, we can transform the parser that recognizes a digit character into one that delivers the result as an integer
\[
\text{digit :: Parser Char Int}
\]
\[
\text{digit = satisfy isDigit <@ digtoInt}
\]

In practice, the \( <@ \) operator is used to build a certain value during parsing (in the case of parsing a computer program this value may be the generated code, or a list of all variables with their types, etc.). Put more generally: using \( <@ \) we can add semantic functions to parsers.
While testing your self-made parsers, you can use \texttt{just} for discarding the parses which leave a non-empty rest string. But you might become bored of seeing the empty list as rest string in the results. Also, more often than not you may be interested in just some parsing rather than all possibilities.

As we have reserved the word parser for a function that returns all parsings, accompanied with their rest string. Let's therefore define a new type for a function that parses a text, guarantees empty rest string, picks the first solution, and delivers the parse tree only (discarding the rest string, because it is known to be empty at this stage). The functional program for converting a parser in such a deterministic parser is more concise and readable than the description above:

\[
\begin{align*}
\text{:: DetPars symbol return} & :== [\text{symbol}] \rightarrow \text{return} \\
\text{some} & : (\text{Parser s a}) \rightarrow \text{DetPars s a} \\
\text{some p} & = \text{snd o hd o just p}
\end{align*}
\]

Use the function \texttt{some} with care: this function assumes that there is at least one solution, so it fails when the resulting \texttt{DetPars} is applied to a text which contains a syntax error.

## 5.6 Matching parentheses

Using the parser combinators and transformers developed thus far, we can construct a parser that recognizes matching pairs of parentheses. A first attempt, that is not type correct however, is:

\[
\begin{align*}
\text{parens} & :: \text{Parser Char Tree} \\
\text{parens} & = \text{symbol} '('
\langle&\rangle \text{parens}
\langle&\rangle \text{symbol ')'}
\langle&\rangle \text{parens}
)\]
\langle|> \text{epsilon}
\end{align*}
\]

This definition is inspired strongly by the well-known grammar for nested parentheses:

\[
\text{Parentheses} ::= '(' \text{Parentheses} ')' \text{Parentheses} \\
| \varepsilon.
\]

The type of the parse tree, however, is a problem. If this type would be \(a\), then the type of the composition of the four subtrees in the first alternative would be \((\text{Char},(a,(\text{Char},a)))\), which is not the same or unifiable. Also, the second alternative (epsilon) must yield a parse tree of the same type. Therefore we need to define a type for the parse tree first, and use the operator \(\langle\rangle\) in both alternatives to construct a tree of the correct type. The type of the parse tree can be for example:

\[
\begin{align*}
\text{:: Tree} & = \text{Nil} \\
\text{Bin (Tree,Tree)}
\end{align*}
\]

Now we can add semantic functions to the parser:

\[
\begin{align*}
\text{parens} & :: \text{Parser Char Tree} \\
\text{parens} & = \text{symbol} '('
\langle&\rangle \text{parens}
\langle&\rangle \text{symbol ')'}
\langle&\rangle \text{parens}
)\text{\(\langle@\\rangle\) epsilon}
\langle|> \text{epsilon} \text{\(\langle@\rangle\) K Nil}
\end{align*}
\]

Using the combinator \(\text{K}\) from the standard environment:

\[
\begin{align*}
\text{K} & :: x y \rightarrow x \\
\text{K x y} & = x
\end{align*}
\]
The rather obscure text \((\ _, (x, (\ _, y)))\) is a lambda pattern describing a function with as parameter a tuple containing the four parts of the first alternative, of which only the second and fourth matter.

In the lambda pattern, underscores are used as placeholders for the parse trees of symbol `'(' and symbol `')'`, which are not needed in the result. In order to not having to use these complicated tuples, it might be easier to discard the parse trees for symbols in an earlier stage. For this, we introduce two auxiliary parser combinators, which will prove useful in many situations. These operators behave the same as `<&>`, except that they discard the result of one of their two parser arguments:

\[
\begin{align*}
(<&) & \text{ infixr 6 :: (Parser s a) (Parser s b) } \rightarrow \text{ Parser s a} \\
(>&) & \text{ infixr 6 :: (Parser s a) (Parser s b) } \rightarrow \text{ Parser s b}
\end{align*}
\]

We can use these new parser combinators for improving the readability of the parser parens:

\[
\begin{align*}
\text{open} & = \text{ symbol } '(' \\
\text{close} & = \text{ symbol } ')'
\end{align*}
\]

\[
\text{parens :: Parser Char Tree} \\
\text{parens} = (\text{open } &> \text{ parens } <& \text{ close}) <& \text{ parens } <& \text{ Bin} \\
\hspace{1cm} \langle | \rangle \text{ succeed } \text{ Nil}
\]

By judiciously choosing the priorities of the operators involved we minimized on the number of parentheses needed. This application shows why we have packed the subtrees of Bin in a tuple.

By varying the function used after `<&` (the semantic function), we can yield other things than parse trees. As an example we write a parser that calculates the maximum nesting depth of nested parentheses:

\[
\begin{align*}
\text{nesting :: Parser Char Int} \\
\text{nesting} = (\text{open } &> \text{ nesting } <& \text{ close}) <& \text{ nesting} \\
\hspace{1cm} <& \text{ \((\ (x,y) \rightarrow \text{ max } (x+1) \ y)\) } \\
\hspace{2cm} \langle | \rangle \text{ succeed } 0
\end{align*}
\]

An example of the use of nesting is:

\[
\text{Start} = \text{ just nesting ['(())(())(())']}
\]

which results in

\[
[[[.3]]]
\]

Indeed nesting only recognizes correctly formed nested parentheses, and calculates the nesting depth on the fly. Without just, this program yields also a number of partial parse results:

\[
[[[.3],[['1'.3],[['11111(())(1)',1],[('11111(())(1)',0)]]]]
\]

If more variations are of interest, it may be worthwhile to make the semantic function and the value to yield in the empty case into two additional parameters. The higher order function foldparens parses nested parentheses, using the given function \(f\) and constant \(e\) respectively, after parsing one of the two alternatives:

\[
\begin{align*}
\text{foldparens :: ((a,a) -> a) a -> Parser Char a} \\
\text{foldparens \ f \ e = p} \\
\hspace{1cm} \text{ where } \ p = (\text{open } &> \text{ p } <& \text{ close}) <& \text{ p } <& \text{ f} \\
\hspace{2cm} \langle | \rangle \text{ succeed } e
\end{align*}
\]

In exercise 7 you are asked to use this function to write some of the parser introduced above.
5.7 More parser combinators

Although in principle you can build parsers for any context-free language using the combinators `<&>` and `<|>`, in practice it is easier to have some more parser combinators available. In traditional grammar formalisms, too, additional symbols are used to describe for example optional or repeated constructions. Consider for example the BNF formalism, in which originally only sequential and alternative composition could be used (denoted by juxtaposition and vertical bars, respectively), but which was later extended to also allow for repetition, denoted by asterisks.

It is very easy to make new parser combinators for extensions like that. As a first example we consider repetition. Given a parser \( p \) for a construction, \( <*> p \) is a parser for zero or more occurrences of that construction. The name of this function is inspired by the habit in BNF-notation to write a (postfix) asterisk to indicate zero or more occurrences.

\[
<*> :: (Parser s a) -> Parser s [a];
<*> p = p <&> <*> p <@ list
\]

The auxiliary function \( \text{list} \) is defined as the uncurried version of the list constructor:

\[
\text{list} :: (x,[x]) -> [x];
\text{list} (x,xs) = [x:xs]
\]

The recursive definition of the parser follows the recursive structure of lists. Perhaps even nicer is the version in which the \( \text{epsilon} \) parser is used instead of \( \text{succeed} \):

\[
<*> :: (Parser s a) -> Parser s [a];
<*> p = p <&> <*> p <@ (\(x,xs\) -> [x:xs])
\]

\[
\text{epsilon} <@ (\_ -> [])
\]

The order in which the alternatives are given only influences the order in which solutions are placed in the list of successes.

But to obtain symmetry, we could also try and avoid the \( <&> \) operator in both alternatives. Earlier we defined the operator \( <@ \) as an abbreviation of applying \( <@ \text{fst} \) to the result of \( <&> \). In the function \( <*> \), also the result of \( <&> \) is postprocessed. We define \( <@&> \) as an abbreviation of postprocessing \( <*> \) with the list function:

\[
(<@&>) :: (Parser s a) (Parser s [a]) -> Parser s [a]
(<@&>) p q = p <&> q <@ \text{list}
\]

Then we can define

\[
<@> :: (Parser s a) -> Parser s [a];
<@> p = p <&> <*> p <@ succeed []
\]

An example in which the combinator \( <@> \) can be used is parsing of a natural number:

\[
\text{natural} :: \text{Parser Char Int}
\text{natural} = <*> \text{digit} <@ \text{foldl nextDigit 0}
\text{where nextDigit a b = a*10+b}
\]

Defined in this way, the \( \text{natural} \) parser also accepts empty input as a number. If this is not desired, we'd better use the \( <@> \) parser combinator, which accepts one or more occurrences of a construction. This function is again inspired by the BNF-notation.

\[
<@> :: (Parser s a) -> Parser s [a]
<@> p = p <&> <*> p
\]

Another combinator that you may know from other formalisms is the option combinator \( <@> \). The constructed parser generates an optional element, depending on whether the construction was recognized or not. The parser yields a list of successes.

\[
<@> :: (Parser s a) -> Parser s [a]
<@> p = p <@ \emptyset (\x -> [x])
\]

\[
\text{epsilon} <@ \emptyset (\_ -> [])
\]
For aesthetic reasons we used \( \epsilon \) in this definition; another way to write the second alternative is `succeed []`.

Using the parser `natural` we can define a parser for a (possibly negative) integer number, which consists of an optional minus sign followed by a natural number. The easiest way is to do case analysis:

```plaintext
integer :: Parser Char Int
integer = <?> (symbol '-') <?&> natural <@ f
  where f ([],n) = n
  f (_,n)  = -n
```

A nicer way to write this parser is by using the `<?&` operator, yielding the identity or negation function in absence or presence of the minus sign, which is finally applied to the natural number. The operator `<?&` is defined in section 5.8.

```plaintext
integer :: Parser Char Int
integer =     (<?> (symbol '-') <?@ (I,K ~))
<&> natural
<@  (f,n) -> f n
```

By the use of the `<&>` and `<@>` functions, a large amount of backtracking possibilities are introduced. This is not always advantageous. For example, if we define a parser for identifiers by

```plaintext
identifier :: Parser Char String
identifier = <+> (satisfy isAlpha) <@ toString
```

a single word may also be parsed in several ways as identifier. As a matter of fact `identifier ['Clean']` yields

```
[[],'Clean'],[['n','Clean']].[['an','Cle'],['ean','Cl'],['lean','C']]
```

The order of the alternatives in the definition of `<+>` and `<*>` caused the greedy parsing, which accumulates as many letters as possible in the identifier is tried first, but if parsing fails elsewhere in the sentence, also less greedy parsings of the identifier are tried – in vain.

In some situations we can predict that it is useless to try non-greedy successes of `<*>` or `<+>` from the structure of the grammar. Examples are the parsers `natural` and `fracpart`. We can define a parser transformer `first`, that transforms a parser into a parser that only yields the first possibility. It does so by taking the first element of the list of successes.

```plaintext
first :: (Parser s a) -> Parser s a
first p = take 1 o p
```

Using this function, we can create special take all or nothing versions of `<*>` and `<+>`:

```plaintext
<!*> :: ((Parser s a) -> Parser s [a])
<!*> = first o <*> 
<!+> :: ((Parser s a) -> Parser s [a])
<!+> = first o <+>
```

when we define

```plaintext
identifier :: Parser Char String
identifier = <!+> (satisfy isAlpha) <@ toString
```

The expression `identifier ['Clean']` now yields

```
[[],'Clean']
```

In the parsing of numbers we have a similar situation. Usually it does not make sense to stop parsing of a number within a sequence on digits. We want at least on digit for the number and consume all subsequent digits:

```plaintext
natural :: Parser Char Int
natural = <!+> digit <@ foldl nextDigit 0
  where nextDigit a b = a*10+b
```
If we compose the `first` function with the option parser combinator:

```hs
<!?> :: ((Parser s a) -> Parser s [a])
<!?> = first o <?>
```
we get a parser which must accept a construction if it is present, but which does not fail if it is not present.

The combinators `<*>, ↔ and <!?>` are classical in parser constructions, but there is no need to leave it at that. For example, in many languages constructions are frequently enclosed between two meaningless symbols, most often some sort of parentheses. For this we design a parser combinator `pack`. Given a parser for an opening token (`s1`), a body (`p`), and a closing token (`s2`), it constructs a parser for the enclosed body:

```hs
pack :: (Parser s a) (Parser s b) (Parser s c) -> Parser s b
pack s1 p s2 = s1 &> p <& s2
```

Applications of this combinator are:

- `parenthesized p = pack (symbol '(') p (symbol ')')`
- `bracketed p = pack (symbol '[') p (symbol ']')`
- `compound p = pack (token ['begin']) p (token ['end'])`

Another frequently occurring construction is repetition of a certain construction, where the elements are separated by some symbol. You may think of lists of parameters (expressions separated by commas), or compound statements (statements separated by semicolons). For the parse trees, the separators are of no importance. The function `listOf` below generates a parser for a (possibly empty) list, given a parser for the items and a parser for the separators:

```hs
listOf :: (Parser s a) (Parser s b) -> Parser s [a]
listOf p s = p <:&> <*> (s &> p) <|> succeed []
```

An useful application is:

```hs
commaList :: (Parser Char a) -> Parser Char [a]
commaList p = listOf p (symbol ',')
```

A somewhat more complicated variant of the function `listOf` is the case where the separators carry a meaning themselves. For example, arithmetical expressions, where the operators that separate the subexpressions have to be part of the parse tree. For this case we will develop the functions `chainr` and `chainl`. These functions expect that the parser for the separators yields a function (!); that function is used by chain to combine parse trees for the items. In the case of `chainr` the operator is applied right-to-left, in the case of `chainl` it is applied left-to-right. The basic structure of `chainl` is the same as that of `listOf`. But where the function `listOf` discards the separators using the operator `&>`, we will keep it in the result now using `<&>`. Furthermore, postprocessing is more difficult now than just applying list.

```hs
chainl :: (Parser s a) (Parser s (a a->a)) -> Parser s a
chainl p s = p <&> <*> (s <&> p) @ f
```

The function `f` should operate on an element (yielded by the first `p`) and a list of tuples (yielded by `<&>` (s `<&>` p)), each containing an operator and an element. For example, `f `e1` = (`e0`, `o1`, `e2`, `o2`, `e3`, `o3`, `e4`)` should return `((e0, o1, e2) o2 e3) e4`. Since we cannot use infix notation here we actually have to write: `o3 (o2 (o1 e0 e1) e2) e3`. You may recognize a version of `foldl` in this (albeit an uncurried one), where a tuple `(c1, e1)` from the list and intermediate result `e` are combined applying `c1 e e1`. We define

```hs
chainl :: (Parser s a) (Parser s (a a->a)) -> Parser s a
chainl p s = p <&> <*> (s <&> p) @ (f `e1` = (`e0`, `o1`, `e2`, `o2`, `e3`, `o3`, `e4`) )
```

II.4 Parsers DRAFT 12 augustus 1997
Dual to this function is \texttt{chainr}, which applies the operators associating to the right. To obtain \texttt{chainr}, change \texttt{foldl} into \texttt{foldr}, flip the list and initial element and reorder the distribution of \texttt{<*>} over the \texttt{<&>} operators:

\begin{verbatim}
chainr :: (Parser s a) (Parser s (a a->a)) -> Parser s a
chainr p s = <*> (p <*> s) <*> p @ (\(x,op\) y -> op x y) e0 1
\end{verbatim}

The function \texttt{chainl} is convenient to parse expressions. Expressions composed of integers and the addition or subtraction of integers can be parsed and evaluated by:

\begin{verbatim}
  expr :: Parser Char Int
  expr = chainl integer (symbol '+' <*> K (+) <|> symbol '-' <*> K (-))
\end{verbatim}

An application of this function is:

\begin{verbatim}
Start = just expr ['54--32-1']
\end{verbatim}

This program yields \[(\[\],85)\]

Note that using \texttt{chainr} instead of \texttt{chainl} in \texttt{expr} changes the associativity of operators. The result of our example program would have been \[(\[\],87)\].

This example shows also that we does not have to construct a parse tree which is a direct representation of the items parsed. The parsed expressions of this example are evaluated immediately.

### 5.8 Analyzing options

The option function \texttt{<?>} constructs a parser which yields an optional element. Post-processing often perform a case analysis whether the element was found or not. You will therefore often need constructions like:

\begin{verbatim}
<?> p <$> f
  where f [] = a
       f [x] = b x
\end{verbatim}

As this necessitates a new function name for every optional symbol in our grammar, we had better provide a higher order function for this situation. We will define a special version \texttt{<?>} of the \texttt{<>} operator, which provides a semantics for both the case that the optional construct was present and that it was not. The right argument of \texttt{<?>} consists of two parts: a constant to be used in absence, and a function to be used in presence of the optional construct. The new transformer is defined by:

\begin{verbatim}
(<?>) infixl 5 :: (Parser s [a]) (b,a->b) -> Parser s b
(<?>) p (no,yes) = p <$> f
  where f [] = yes x
       f [x] = no
\end{verbatim}

To illustrate a practical use of this, let's extend the parser for natural numbers to floating point numbers. We have for natural numbers:

\begin{verbatim}
natural :: Parser Char Int
natural = <*> digit <$> foldl nextDigit 0
  where nextDigit a b = a*10+b
\end{verbatim}

The fractional part of a floating point number is parsed by:

\begin{verbatim}
fractpart :: Parser Char Real
fractpart = <*> digit <$> foldr nextDigit 0.0
  where nextDigit d n = (n + toReal d)/10.0
\end{verbatim}

But the fractional part is optional in a floating point number.

\begin{verbatim}
real :: Parser Char Real
real = (integer <$> toReal)
  <$> (<?> (\(symbol '.*' &\) fractpart) <$> (\(0.0,1)\))
  <$> uncurry (+)
\end{verbatim}
The decimal point is for separation only, and therefore immediately discarded by
the operator \&>. The decimal point and the fractional part together are optional. In their ab-

sence, the number 0.0 should be used, in their presence, the identity function should be
applied to the fractional part. Finally, integer and fractional part are added.

We can use this approach for yet another refinement of the \texttt{chainr} function. It was defined
in the previous section using the function \texttt{<\&>}. The parser yields a list of tuples (element, operator), which immediately afterwards is destroyed by \texttt{foldr}. Why bothering building the list, then, anyway? We can apply the function that is folded with directly during pars-
ing, without first building a list. For this, we need to substitute the body of \texttt{<\&>} in the
definition of \texttt{chainr}. We can further abbreviate the phrase \texttt{p <\&> epsilon} by \texttt{?\& p}. By di-
rectly applying the function that was previously used during \texttt{foldr} we obtain:

\begin{verbatim}
chainr' :: (Parser s a) (Parser s (a a->a)) -> Parser s a
chainr' p s = q
  where q = p ?& (s ?& q) ?& (flip Fun)
       q ?& (x,op) -> op x
\end{verbatim}

5.9 Arithmetical expressions

In this section we will use the parser combinators in a concrete application. We will de-
velop a parser for arithmetical expressions, of which parse trees are of type \texttt{Expr}. O
operators will be stored as functions with the corresponding names.

\begin{verbatim}
::Expr = Int Int
       | Var String
       | Fun String [Expr]
\end{verbatim}

In order to account for the priorities of the operators, we will use a grammar with non-
terminals expression, term and factor: an expression is composed of terms separated by +
or -; a term is composed of factors separated by * or /, and a factor is a constant, a vari-
able, a function call, or an expression between parentheses.

\begin{verbatim}
expr ::= term {{ '+' | '-' } term}*
term ::= fact {{ '*' | '/' } fact}*
fact ::= integer
       | identifier [ '(' expr { ',' expr } * ')' ]
       | '(' expr ')'.
\end{verbatim}

This grammar is represented in the functions below:

\begin{verbatim}
fact :: Parser Char Expr
fact =     integer <@ Int
         <|> identifier
         <$> (<!?> (parenthesized (commaList expr))
             <$> (Var,flip Fun))
         <$> (x,op) -> op x)
         <$> parenthesized expr
\end{verbatim}

The first alternative is a constant, which is fed into the semantic function \texttt{Con}. The second
alternative is a variable or function call, depending on the presence of a parameter list. In
absence of the latter, the function \texttt{Var} is applied, in presence the function \texttt{Fun}. The function
\texttt{flip} is used to place the function name between the constructor \texttt{Fun} and the arguments.

\begin{verbatim}
flip :: (a b -> c) b a -> c
flip f b a = f a b
\end{verbatim}

For the third alternative of \texttt{fact} there is no semantic function; the meaning of an expres-
sion between parentheses is the same as that of the expression without parentheses. If you
insists of adding such a function you can use \texttt{?@ I}.

For the definition of a term as a list of factors separated by multiplicative operators we
will use the function \texttt{chainl}:
Recall that \texttt{chainl} repeatedly recognizes its first parameter (\texttt{fact}), separated by its second parameter (an \texttt{*} or \texttt{/}). The parse trees for the individual factors are joined by the function \texttt{mkFun} supplied after \texttt{@}.

The function \texttt{expr} is analogous to \texttt{term}, only with additive operators instead of multiplicative operators, and with terms instead of factors:

\begin{verbatim}
expr :: Parser Char Expr
expr = chainl term ((symbol '+' <|> symbol '-') <@ mkFun)
\end{verbatim}

From this example the strength of the method becomes clear. There is no need for a separate formalism for grammars; the production rules of the grammar are joined using higher-order functions. Also, there is no need for a separate parser generator (like yacc); the functions can be viewed both as description of the grammar and as an executable parser.

5.10 Generalized expressions

Arithmetical expressions in which operators have more than two levels of priority can be parsed by writing more auxiliary functions between \texttt{term} and \texttt{expr}. The function \texttt{chainl} is used in each definition, with as first parameter the function of one priority level higher.

If there are nine levels of priority, we obtain nine copies of almost the same text. This would not be as it should be. Functions that resemble each other are an indication that we should write a generalized function, where the differences are described using extra parameters. Therefore, let's inspect the differences in the definitions of \texttt{term} and \texttt{expr} again. These are:

- The operators that are used in the second parameter of \texttt{chainl}
- The parser that is used as first parameter of \texttt{chainl}

The generalized function will take these two differences as extra parameters: the first in the form of a list of operator names, the second in the form of a parse function. In general there is no reason to assume that operators are exactly one symbol long. So, we use \texttt{token} instead of \texttt{symbol}. Finally, we replace \texttt{mkFun} by a parameter.

\begin{verbatim}
gen :: ([s] e e -> e) [[s]] (Parser s e) -> Parser s e | == s
  gen f ops p = chainl p (choice [token t <@ f \ t <- ops])
\end{verbatim}

The function \texttt{choice} is the generalisation of \texttt{|>} to a list of parsing alternatives developed in exercise 12.

If furthermore we define as shorthand:

\begin{verbatim}
add_op = ['+', '-']
mul_op = ['*', '/']
\end{verbatim}

then \texttt{expr} and \texttt{term} can be defined as applications of \texttt{gen}:

\begin{verbatim}
expr = gen mkFun add_op term
  term = gen mkFun mul_op fact
\end{verbatim}

By expanding the definition of \texttt{term} in that of \texttt{expr} we obtain:

\begin{verbatim}
expr = gen mkFun add_op (gen mkFun mul_op fact)
\end{verbatim}

Which an experienced functional programmer immediately recognizes as an application of \texttt{foldr}. The function to apply to the list elements is \texttt{gen mkFun}, the unit element for an empty list is \texttt{fact} and the list of items to process is \texttt{[add_op, mul_op]}.
expr :: Parser Char Expr
expr = foldr (gen mkFun) fact [add_op, mul_op]

From this definition a generalization to more levels of priority is simply a matter of extending the list of operator-lists.

The very compact formulation of the parser for expressions with an arbitrary number of priority levels was possible because the parser combinators could be used in conjunction with the existing mechanisms for generalization and partial parametrization in the functional language.

Contrary to conventional approaches, the levels of priority need not be coded explicitly with integers. The only thing that matters is the relative position of an operator in the list of lists with operators. All operators in a sub-list have the same priority. The insertion of additional levels of priority is very easy.

5.11 Monadic parsers

The result of the sequential composition of two parsers by the operator `<&>` is grouped in a tuple. Often this tuple is immediately destructed in order to construct the intended parse tree. Especially when a long sequence of items is parsed, handling these tuples can become ugly.

By using the idea of monads we can avoid the construction and destruction of tuples. The operator `<&=>` is the equivalent of the operator `bind` used to compose two monad manipulations. The name and use of `<&=>` are similar to the operator `<&>`. The left argument of the operator `<&=>` is an ordinary parser. The right argument is a function that takes the parse tree of the first parser as argument and yields a parser. These two arguments are composed to a new parser:

```haskell
(<&=>) infixr 6 :: (Parser s a) (a -> Parser s b) -> Parser s b
(<&=>) p1 p2 = p
  where p xs = [  tuples
    \ (xs1,v1) <- p1 xs
    ,  tuples   <- p2 v1 xs1
    ]
```

When you compare this with the definition of `<&>` it is obvious that also the definitions of `<&=>` and `<&>` are very similar.

The parser `succeed` can be used where you expect the monad construct `return`. For instance, a parser that will recognise two integers separated by the symbol `+` and yields their sum as parse tree is defined in the monadic style as\(^\text{1}\):

```haskell
ints :: Parser Char Int
ints = integer  <&=> (\i ->
  symbol '+' <&=> (\_ ->
    integer  <&=> (\j ->
        succeed (i+j)))))
```

There is no need to treat the operator `<&=>` as an opposite of `<&>`. These operators, as well as all other parser combinators, can be combined to write new parsers. This is illustrated by the following examples.

The first example is the definition of the parser `nesting` from section 5.6. This parser computes the maximum depth of nesting pairs of parentheses. Using the new parser combinator it can be defined as:

\(^{1}\)In Clean 1.2 the parentheses around the lambda expressions can be omitted.
neste:: Parser Char Int
neste = (open &> nes & close) &=> (\x ->
 nes &=> (\y ->
  succeed (max (x+1) y))
<|> succeed 0

In fact, we can often do without the 'result' operator succeed. We can replace the last operator &=> by @. Using this the parser nesting can be defined as:

neste:: Parser Char Int
neste = (open &> nes & close) &@ (\x ->
 nes &@ (\y ->
  max (x+1) y))
<|> succeed 0

The second example is the parser fact introduced in the section about parsing expressions above.

fact:: Parser Char Expr
fact = integer &@ Con
<|> identifier &@ (\v ->
  (!?) (parenthesized (commaList expr))
  ?@ (Var v, args -> Fun v args))
<|> parenthesized expr

These example show that the monadic style can be combined with the other parser combinators. The obtained parsers contain less combinators due to improved handling of the parse results. Since the monadic parsers are a little more compact and contain no tuple handling they are more appealing.

5.12 Context sensitivity

Until now all decisions in parsing are taken on the syntactical structure of the input. In general this is not always possible. For instance, when we are parsing a functional language like Clean an identifier can be the name of a function are a function argument. We cannot make a decision based on the syntax alone. Whether an identifier is a argument or a function depends on the context of the identifier.

A similar problem arises in the parsing of functions that consists of several alternatives. It is easy to construct a parser for alternatives.

:: Definition = FunDef [Alt]
:: Alt = Alt Fsymb [Pattern] Expr
:: Fsymb ::= String

pAlt:: Parser Char Alt
pAlt = identifier &@ (\f ->
  !* pattern &@ (\args ->
  spsymbol '=' &@ spexpr &@ (spsymbol ',' &@ (\b -> Alt f args b))))

pattern:: Parser Char Expr
pattern = sp (identifier @ Var @ integer @ Int)

Combining these alternatives to functions is a bit tricky. When we would simply write

pFun:: Parser Char Definition
pFun = <> pAlt

all alternatives are combined into one function. This is clearly not the intention. Delaying the generation of functions to the processing of the list of alternatives is not always desirable. What we really want is to parse one alternative and then collect all other alternatives that start with the same name.

We cannot achieve this by a condition on the list of parsed alternatives. The alternatives are parsed at that moment and there is no way to undo the parsing. We should only in-
clude the alternative in the list when it belongs to this function. To realise this we can add an additional argument to the parser of alternatives that checks the appropriate condition on the functions symbol.

\[
p\text{Fun2} :: \text{Parser Char} \\
p\text{Fun2} = p\text{Alt2} \bowtie \text{\{\_\_\->True\}} \bowtie \text{\{\_a\_:(Alt \ f \ _ \ _) ->\}} \\
\bowtie \text{\{\_**\} (p\text{Alt2} \bowtie \text{\{\_g\_->f==g\})} \\
\bowtie \text{\{\_@ \ (r \ -> \text{FunDef [a;r]}))}
\]

\[
p\text{Alt2} :: (\text{String} \rightarrow \text{Bool}) \rightarrow (\text{Parser Char Alt}) \\
p\text{Alt2} \ p = \text{Psymb} \bowtie \text{\{f \ ->\}} \\
\bowtie \text{\{if \ (p \ f) \ ( \text{pattern} \bowtie \text{\{args ->\}} \\
\bowtie \text{\{spsymbol '=' \& \}} \\
\bowtie \text{\{sp expr <\&} \\
\bowtie \text{\{spsymbol ',' \&} \\
\bowtie \text{\{Alt f args b\)) \}} \\
\bowtie \text{\{fail\}}}
\]

However, it is more elegant to do impose this kind of conditions with a new parser combinator. The combinator \(\bowtie\) is defined as:

\[
(<\bowtie>) \ \text{infix 9} :: (\text{Parser s a}) \ (a->\text{Bool}) \rightarrow \text{Parser s a} \\
(<\bowtie>) \ p \ f = \ \text{\{xs \rightarrow [ t \ \}} \\
\bowtie \text{\{t:=(_,_v) <-> p xs \}} \\
\bowtie \text{\{f v \}}
\]

This parser imposes an condition on the parsed items and only succeeds when the condition yields \(T\) true. Using this combinator, the parsing of functions can be defined as:

\[
p\text{Fun} = p\text{Alt} \bowtie \text{\{a\_:(Alt \ f \ _ \ _) ->\}} \\
\bowtie \text{\{**\} (p\text{Alt} \bowtie (<\bowtie>) \bowtie \text{\{(Alt \ g \ _ \ _) -> f==g\})} \\
\bowtie \text{\{@ \ (r \ -> \text{FunDef [a;r]}))}
\]

Although this definition is very elegant, it is slightly less efficient than the parser \(p\text{Fun2}\). Parser \(p\text{Fun}\) parses the entire alternative before the test is executed, while \(p\text{Fun2}\) evaluates the test as soon as the function symbol is parsed. The parser \(p\text{Alt2}\) can be written more elegant using the conditional combinator \(\bowtie\).

5.13 Common traps

The parsers themselves can be read equally well as grammar rules. At some places the specified parsers look more complicated than the grammar, this is not caused by the parsing part, but by the manipulation of recognised items. Manipulation of the recognised items is usually done in attribute grammars \([??]\) or affix grammars \([??]\). Here we have the advantage that parsing and manipulation of the recognised items are done in the same high level language. Whenever necessary we can employ the full power of this language.

This looks all very well and simple, but we cannot use this without some understanding of what is actually happening. Here it is much easier to understand what is happening than in some parser generator, since the implementation of all constructs used is available.

Left recursion

Problems arise for instance with left-recursive rules in a syntax description. A rule in a grammar is left-recursive when its name occurs immediately after the ::= symbol. Also when its name occurs in an other rule of the grammar that can be called without consuming any input the same problem occurs. An example of a direct left recursive grammar is:

\[
\text{exp} ::= \text{exp oper exp} \mid '(' \text{exp} ')' \mid \text{integer} . \\
\text{oper} ::= ']' \mid '['.
\]
When we would write naively a parser for this syntax this will look like:

```plaintext
def expres :: Parser Char Int
  = expres <|> oper <|> expres <@ (\y -> op x y))
     <|> parenthesized expres
     <|> integer

def oper :: Parser Char (a a -> a) | +, - a
  = symbol '+' <@ K (+) <|> symbol '-' <@ K (-)
```

Although that it looks obvious correct, it will always generates an overflow (either stack overflow or heap full). This is caused by the fact that `expres` is a recursive function that will enter the recursion immediately. The function does not consume any input nor performs any test before entering the recursion: this cannot terminate. In part I we saw that recursive functions are accomplished as long as we guarantee that they will yield a result in some finite number of iterations. In this situation it is fine to write parsers as recursive function as long as we guarantee that these parsers consume some input symbols. Since the input is assumed to be finite the parser will terminate. Fortunately it can be show that it is always possible to transform a left-recursive grammar to a grammar that can be parsed safely.

In our example we can solve the termination problems by writing

```plaintext
def exp := integer oper exp | ('(' exp ')') | integer.
```

The corresponding parser is

```plaintext
def expres2 :: Parser Char Int
  = integer <|> expres2 <@ (\y -> op x y))
     <|> parenthesized expres2
     <|> integer
```

This parser has an other problem. It associates operators to the right, while in mathematics this is usually done to the left. For instance the input `['1-2-3']` is evaluated to 2 insted of -4. A correct way to construct such a parsers for expressions is.

```plaintext
def expres3 :: Parser Char Int
  = elem <|> (oper <|> elem) <@ foldl (\a (op,b) -> op a b) x list)
```

An other solution is to use the function `chainl` defined above.

### Parsing the same structure again

Apart from the associativity problem, the parser `expres2` has an other problem. When you supply an input containing only an integer, e.g. `['42']`, this integer is parsed twice. The function `expres2` first tries to find a compound expression. Parsing a compound expression consists of parsing an integer, parsing an operand and parsing an other expression.

Applied to our example parsing an integer secedes, but parsing an operator fails. Now the next element in the list of successes is evaluated. The second possibility is a parenthesized `expres2`. This fails immediately since the first symbol in the input is not a `('`.

---

2 When we have not defined `integer` using `<!+>`, the parser will take `'4'` as an integer and tries to parse `['2']` as an operator and an expression as second element in the list of successes.
Now the integer will be parsed for the second time and the parser will successfully terminate. Parsing an integer twice is not much of a problem, but when you use a complicated structure instead of integer this can slow down the compiler considerably. Especially when the surrounding parser is applied in a similar situation.

This problem can always be avoided by writing the parser slightly different. The essential step is to take care that the initial part is parsed only once. All possible continuations are grouped to one parsed that is applied after recognition of the first element.

```haskell
expres4 :: Parser Char Int
expres4 = integer <|> (
  succeed x
  <|> oper <|> expres4 @ (y ->
    op x y))
  <|> parenthesized expres4
```

The parser expres3 shows an other way to avoid this problem. The same technique is used to parse an identifier only once in fact.

### 5.14 Error handling

The parsers constructed in the way outlined in the previous sections works fine when the input can be parsed according to the grammar. On an erroneous input however, the constructed parsers show undesirable behaviour. At the spot of the error the parser just generates a failing parse of the sub-structure at hand. The parser starts trying all possible alternatives and can generate lots of partial parses. In this section we introduce some extensions that enable the detection of errors. When an error is found, it can be handled in several ways. We show how to interrupt parsing at an error, error detection, and also how the continue parsing after the detection of an error, error recovery.

In general you should equip your compilers with some error detection. Whether you will also include error recovery or not, depends on your needs and the effort you want to put in writing the parser. In general it is hard to write a parser that is good in error recovery: people, including yourself, appear to be more creative in making syntax errors than you can imagine while you are writing the error recovery part of the parser.

#### Detecting errors

To enable the detection of errors we introduce the or-else operator <!>. The ordinary or operator, <|>, yields all solutions of the first parser and all solutions of the second parser. The or-else operator <!> only activates the second parser when the first one fails.

```haskell
(!<>) infixr 4 :: (Parser s r) (Parser s r) -> Parser s r
(!<>) p q = p' // apply q when p fails
  where p' xs = case p xs of
    [] -> q xs
    r -> r
```

To limit the amount of parentheses to write, we can give this parser combinator a higher priority than the operators <&> and <|>. This might be convenient for detecting errors, but is too confusing when you use the or-else operator as alternative for the operator <|>.

The operator <!> is used in situations where the syntax of the language to parse guarantees that some construct must be present. This implies that there is an error when the corresponding parser p fails. It is also possible to use <!> in ordinary parsers. The result of the expressions p1 <!> p2 <!> p3 is the result of p1 when it is not empty. When p1 fails the result of this expression is the result of p2 unless that is also empty. When p1 and p2 fails the result of this expression is the result of p3. Do not confuse this with first (p1 <|> p2 <|>
The last expression yields the first result of the concatenation of the results of \( p_1, p_2 \) and \( p_3 \). The first expression yields the result of the first parser that does not fail, this does not need to be a single result.

Now we will give some suggestions of what can be done when an error is detected.

**Interrupting the parser**

The simplest thing to do is to interrupt the parser immediately when an error is detected. This simple error handling is in many situations superior to the standard way of handling errors in the parser described until now. Without special measurements the parser starts to try all alternatives and returns usually one or more partial parses when an error in the input occurs.

Consider the following grammar for a simple imperative language:

\[
\begin{align*}
tiny &::= \text{'BEGIN'} \text{ statements } \text{'END'}. \\
statements &::= \text{ stmt } [';'] \text{ statements } . \\
stmt &::= \text{ identifier } ':=' \text{ expression} \\
&| \text{ 'IF' expression 'THEN' statements [ 'ELSE' statements ] } \\
&| \text{ 'WHILE' expression 'DO' statements} \\
&| \text{ 'PRINT' expression} \\
&| \text{ 'VAR' identifier [ integer ]}. \\
stmts &::= \text{ stmt } [ '\text{'BEGIN'}' \text{ statements } '\text{'END'}'.
\end{align*}
\]

The following data-structure is used to store parse trees of this language, the type \( \text{Expr} \) introduced above is used for expressions in \( \text{Tiny} \):

\[
\begin{align*}
:: \text{Tiny} &::= [\text{TStatement}] \\
:: \text{TStatement} &::= \text{ Declare Variable Int} \\
&| \text{ Assign Variable Expr} \\
&| \text{ If Expr [TStatement] [TStatement]} \\
&| \text{ While Expr [TStatement]} \\
&| \text{ Print Expr} \\
\end{align*}
\]

As soon as the parser has seen the keyword \text{WHILE} it must detect a complete while-statement as the next item. The keyword \text{WHILE} must always be followed by an expression, the keyword \text{DO}, \text{stmts} etc. This is used in the construction of the following parser for statements. As soon as a piece of input is recognized as a statement, it is useless to look whether it is perhaps also an other statement. So, we will use \text{<!>} instead of \text{<|>}.

\[
\begin{align*}
\text{stmt} &::= \text{ Parser Char TStatement} \\
\text{stmt} &::= \text{ IFstmt <!> WHILEstmt <!> PRINTstmt <!> ASSIGNstmt <!> declaration} \\
\text{WHILEstmt} &::= \text{ Parser Char TStatement} \\
\text{WHILEstmt} &::= \text{ WHILEtok} \\
&| \text{ expr <!> pError "WHILE: condition expected"} \\
&| \text{ DOtok <!> pError "WHILE: DO expected"} \\
&| \text{ stmts <!> pError "WHILE: Body expected"} \\
&| \text{ @ (c,b) -> While c b} \\
\text{stmts} &::= \text{ Parser Char [TStatement]} \\
\text{stmts} &::= \text{ BEGINtok} \\
&| \text{ statements <!> pError "stmts expected"} \\
&| \text{ ENDtok <!> pError "END expected"} \\
&| \text{ stmt <!> stmt <@ (s) -> [s]}} \\
\text{statements} &::= \text{ Parser Char [TStatement]} \\
\text{statements} &::= \text{ listOf TStatement (spsymbol ';')} \\
\end{align*}
\]

Using:

\[
\begin{align*}
\text{WHILEtok} &::= \text{ sptoken ['WHILE']} \\
\text{DOTok} &::= \text{ sptoken ['DO']} \\
\text{BEGINtok} &::= \text{ sptoken ['BEGIN']} \\
\end{align*}
\]

218 Functional programming in Clean
The most naive implementation of `pError` is just

\[ pError s = abort s \]

A slightly more sophisticated implementation of this function also shows the piece of input that cannot be parsed.

\[
\begin{align*}
pError :: & \text{String} \rightarrow \text{Parser s r} \mid \text{ToString} s \\
& \text{pError mes} = q \\
& \text{where } q \text{ xs} = \text{abort } ("\text{Parse error: }+\)\text{mes}+\). \text{Input} = "+\text{show } (\text{take } 20 \text{ xs})+\)\text{\"n}"
\end{align*}
\]

\[
\begin{align*}
\text{show :: [s]} & \rightarrow \text{String} \mid \text{ToString s} \\
\text{show list} = & \"[\mapsto \text{showTl list} \\
& \text{where } \text{showTl } [\emptyset] = \"\)"
\end{align*}
\]

\[
\begin{align*}
\text{showTl } [a:r] = & \text{toString } a + \)\text{,,} + \text{showTl } r
\end{align*}
\]

Since it is unknown which part of the input caused the error, we have arbitrary chosen to show the first 20 input symbols.

**Error recovery**

In the previous sub-section we showed how the parser can be interrupted when an error is detected. It is not always appropriate to abort a program when an error is detected. A way to indicate an error and to continue parsing is to store an indication of the error in the parse tree constructed. In order to do this we extend the algebraic type(s) for the parse tree with an error indicator. The type for statements in Tiny becomes:

\[
\begin{align*}
\text{:: TStatement} & = \text{Declare Variable Int} \\
& \mid \text{Assign Variable Expr} \\
& \mid \text{If Expr }[\text{TStatement}] [\text{TStatement}] \\
& \mid \text{While Expr }[\text{TStatement}] \\
& \mid \text{Print Expr} \\
& \mid \text{StmtError } \mid \text{The error indication}
\end{align*}
\]

Whenever necessary we can give `StmtError` some arguments to indicate the error that has been detected. A similar extension can be made to the type `Expr` which represent the expressions. This can be used to replace an omitted expression or body in a while-loop:

\[
\begin{align*}
\text{WHILEstmt} = & \text{WHILEtok} \\
& \& (\text{expr } <!> \text{succeed ExprError}) \\
& \& \text{DOTok} \\
& \& (\text{stmts } <!> \text{succeed }[\text{StmtError}]) \\
& \langle\theta \ (c,b) \rightarrow \text{While } c \ b\rangle
\end{align*}
\]

In general an error does not consists of the omission of an entire language construct, but consists of some illegal language construct. This kind of error is not handled properly by the kind of parsers in the previous section: the erroneous construct remains present in the input. It is usually difficult to indicate where the erroneous construct stops. The functions `skip` and `skipAlso` can be used to resynchronise the parser and the input. These functions drop at most \(n\) elements of the input stream until the parser `until` given as arguments succeeds. When the parser `until` succeeds the result is created by the parser `result` given as second argument to `skip` or `skipAlso`.

\[
\begin{align*}
\text{skip :: Int } (\text{Parser s x}) (\text{Parser s r}) \rightarrow & \text{Parser s r} \\
\text{skip n until result} = & \text{p n} \\
\text{where } p 0 \text{ xs} = [] \\
& \text{\// } n \text{ symbols are skipped fail} \\
& p n \text{ xs} = \text{case until } xs \text{ of} \\
& [] \rightarrow \text{case } xs \text{ of} \\
& & [] \rightarrow [] \\
& & [...] \rightarrow p (n-1) tl \\
& ne \rightarrow \text{result } xs \\
& \text{\// until succeeds apply result parser}
\end{align*}
\]

The function `skipAlso` is similar. The difference with `skip` is that also the input fragment that stopped the skipping is removed from the input.
Using this function the error handling in the parser for the while-loop can be improved. We assume that only one error occurs in each while-loop. This implies for instance that when parsing the condition fails the keyword **do** will be present. The parser for while statements can now be improved to:

```plaintext
WHILEstmt = WHILEtok
&> (expr <!> skip 20 DOTok (succeed ExprError)
    <!> skipAlso 20 expr (succeed ExprError))
<&> (DOTok <!> skipAlso 100 DOTok (succeed []))
<&> (stmts <!> endStmt [StmtError]
    <!> skipAlso 20 stmt (succeed [StmtError])))
@ (\(c,b\) -> While c b)
```

Although this parser is able to recover from many errors in the input we cannot be satisfied with it. The major reason for rejecting this parser is that it is too good in recovering. Consider for instance the following input

```plaintext
WHILE n >> 0
  DO BEGIN x := x+1 END;
```

The condition of this loop is an erroneous expression since >> is not an valid operator. Parsing the conditions yields the expression \texttt{VAR "n"} as condition. The parser **DOTok** fails since the input contains \texttt{[' >> 0 DO …']}. By dropping some characters the error recovery can resynchronise the input with the parser and the parser recognise the token **DO**. Unfortunately the result of parsing the token **DO** is thrown away. So, there seem to be no way to include an error message in the parse tree!

The solution is to detect this type of error one level higher. Instead of detecting that the keyword **DO** is missing and start skipping to find this keyword, we let the parser fail. We can check whether parsing the whole while-loop succeeds or not.

```plaintext
WHILEstmt = WHILEtok &> ( (expr <!> skip 100 DOTok (succeed ExprError))
    <!> DOTok
    &> (stmts <!> endStmt [StmtError])
    @ (\(c,b\) -> While c b)
    <!> skipAlso 100 endStmt (succeed StmtError))
```

**Listing errors**

An other reason for being unsatisfied with the current approach of error recovery is that an error is not detected by the rest of the program before the part of the parse tree representing the error is actually used. A way to generate error messages without aborting the program is to write these error messages to a file. In order to prevent carrying this file around we will use the file **stderr** (standard error) as target. The appropriate version of the function **pError** is:

```plaintext
pError :: String (Parser s r) -> Parser s r | toString s
pError mes p = q
  where
    q xs = let! out = fwrite message stderr
           in K (p xs) out
    where message = "Parse error: "+mes+". Input = "+show (take 20 xs)+"\n"
```

220 Functional programming in Clean
Although this approach serves our goals, one can argue that this is not a very nice solution since printing error messages is a kind of side-effect. The price we have to pay for being fully referential transparent is that a file has to be passed around in the parser.

An other way to handle errors is of course to change the type `Parser`. Instead of a list of two-tuples a parser yields a list of three tuples. The additional field in the results is used to pass a list of error messages.

Combining the tools introduced in the previous sub-sections we can write parsers that does error detection, error recovery and list the errors found on `stderr`. As example we show again the parser for while statements in Tiny.

```haskell
WHILEstmt = WHILEtok &> (  
  (expr <!> pError "WHILE: condition expected"  
    (skip 100 DOTok (succeed ExprError)))
  &< (DOtok <!> pError "DO expected" fail)
  &< (stmts <!> pError "Body expected" (endStmt [StmtError])))
  <@ (
    <@
      (c,b) -> While c b
    @>
  )<!> pError "Invalid WHILE" (endStmt StmtError))

endStmt :: r -> Parser Char r
endStmt r = skipAlso 100
  (    ENDtok <|> (BEGINtok &> (listOf stmt (spsymbol ';') <|> ENDtok)))
  (succeed r)
  <!> abort "endStmt failed"
```

In order to show that it is still possible to interrupt the parser in when error recovery fails we force `endStmt` to stop the program when the skipping fails to find a new synchronisation point.

### 5.15 Self application

Although in the preceding sections it is shown that a separate formalism for grammars is not needed, users might want to stick to, for example, BNF-notation for writing grammars. Therefore in this section we will write a function that transforms a BNF-grammar into a parser. The BNF-grammar is given as a string, and is analyzed itself of course by a parser. This parser is a parser that as parse tree yields a parser! Thus, the title of this section is justified.

This section is structured as follows. First we write some functions that are needed to manipulate an environment. Next, we describe how a grammar can be parsed. Then we will define a data structure in which parse trees for an arbitrary grammar can be represented. Finally we will show how the parser for grammars can yield a parser for the language described by the grammar.

#### Environments

An environment is a list of pairs, to represent a finite mapping from arguments to results (a function in the mathematical sense of the word). The function `assoc` can be used to associate a value to its image under the mapping in the given environment (applying a `function' to an argument).

\[::Env a b ::= [(a,b)]\]

\[assoc :: (Env a b) a \rightarrow b | Eq, toString a\]

\[assoc [(u,v):ws] x | x \equiv u = v = assoc ws x\]

\[assoc _ x = abort \("No association for " ++ toString x)\]

We also define the function `mapenv` that applies a `Clean` function to all images in an environment.
mapenv :: (a->b) (Env s a) -> Env s b
mapenv f env = [(x,f v) \ (x,v) <- env]

Grammars

In a grammar, terminal symbols and nonterminal symbols are used. Both are represented by a list of characters. We provide a datatype with two cases for the two kinds of symbols:

```
::Symbol = Term [Char]
    | Nont [Char]
```

Equality on these symbols is defined in the obvious way:

```
instance == Symbol
  where (==) :: !Symbol !Symbol -> Bool
       (==) (Term s1) (Term s2) = s1 == s2
       (==) (Nont s1) (Nont s2) = s1 == s2
       (==) _         _         = False
```

The right hand side of a production rule consists of a number of alternatives, each of which is a list of symbols:

```
::Alt ::= [Symbol]
::Rhs ::= [Alt]
```

Finally, a grammar is an association between a (nonterminal) symbol an the right hand side of the production rule for it:

```
::Gram ::= Env Symbol Rhs
```

Grammar can easily be denoted using the BNF-notation. For this notation we will write a parser, that as a parse tree yields a value of type Gram. The parser for BNF-grammars is parameterized with a parser for nonterminals and a parser for terminals, so that we can adopt different conventions for representing them later. We use the elementary parsers sp-token and spsymbol rather than token and symbol to allow for extra spaces in the grammar representation.

```
bnf :: (Parser Char [Char]) (Parser Char [Char]) -> Parser Char Gram
bnf nontp termp = <*> rule
  where rule = (   nont
            <&> sptoken ['::='] &> rhs <& spsymbol '.'
        )
            rhs  = listOf (<*> (term <|> nont)) (spsymbol '|')
            term = sp termp <@ Term
            nont = sp nontp <@ Nont
```

A BNF-grammar consists of many rules, each consisting of a nonterminal separated by a ::= symbol from the rhs and followed by a full stop. The rhs is a list of alternatives, separated by |-symbols, where each alternative consists of many symbols, terminal or nonterminal. Terminals and nonterminals are recognized by the parsers provided as parameter to the bnf function.

An example of a grammar representation that can be parsed with this parser is the grammar for block structured statements:

```
blockgram = ['BLOCK ::= begin BLOCK end BLOCK | .']
```

Here we used the convention to denote nonterminals by upper case and terminals by lower case characters. In a call of the bnf functions we should specify these conventions. For example:

```
Start :: Gram
Start = some (bnf nont term) blockgram
  where nont = <!+> (satisfy isUpper)
        term = <!+> (satisfy isLower)
```

The output of this test is the following environment:

```
[(Nont ['BLOCK']),[(Term ['begin'])]
```
We can no longer use a data structure that is specially designed for one particular grammar, like the `Expr` type above. Instead, we define a generic data structure, that describes parse trees for sentences from an arbitrary grammar. We simply call them `RTree`; they are instances of multibranching trees or rose trees:

```
::RTree = Node Symbol [RTree]
```

**Parsers instead of grammars**

Using the `bnf` function, we can easily generate values of the `RTree` type. But what we really need in practice is a parser for the language that is described by a BNF-grammar. So let’s define a function

```
parsGram :: Gram Symbol -> Parser Symbol RTree
```

that given a grammar and a start symbol generates a parser for the language described by the grammar. Having defined it, we can let is postprocess the output of the `bnf` function.

The function `parsGram` uses some auxiliary functions, which generate a parser for a symbol, an alternative, and the rhs of a rule, respectively:

```
parsSym :: Symbol -> Parser Symbol RTree
parsSym s=:(Nont n) = parsRhs (assoc gram s) <@ Node s
parsSym s=:(Term t) = symbol s <@ K [] <@ Node s

parsRhs :: (Rhs -> Parser Symbol [RTree])
parsRhs = choice o (map parsAlt)

parsAlt :: (Alt -> Parser Symbol [RTree])
parsAlt = sequence o (map parsSym)
```

The `parsSym` function distinguishes cases for terminal and nonterminal functions. For terminal symbols a parser is generated that just recognizes that symbol, and subsequently a `Node` for the parse tree is build.

For nonterminal symbols, the corresponding rule is looked up in the grammar, which is an environment after all. Then the function `parsRhs` is used to construct a parser for a rhs. The function `parsRhs` generates parsers for each alternative, and applies the function `choice` over the list of alternatives. The function `parsAlt`, finally, generates parsers for the individual symbols in the alternative, and combines them with the `sequence` function.

**A parser generator**

In theoretical textbooks, a context-free grammar is usually described as a four-tuple consisting of a set of nonterminals, a set of terminals, a set of rules and a start symbol. Let’s do so, representing a set of symbols by a parser:

```
::SymbolSet :== Parser Char [Char]
::CFG       :== (SymbolSet,SymbolSet,[Char],Symbol)
```

Now we will define a function that takes such a four-tuple and returns a parser for its language. Would it be too immodest to call this a parser generator?

```
parsgen :: CFG -> Parser Symbol RTree
```
The sets of nonterminals and terminals are represented by parsers for them. The grammar is a string in BNF notation. The resulting parser accepts a list of (terminal) symbols and yields a parse tree.

**Lexical scanners**

The parser that is generated accepts Symbols instead of Chars. If we want to apply it to a character string, this string first has to be tokenized by a lexical scanner.

For this, we will make a function twopass, which takes two parsers: one that converts characters into tokens, and one that converts tokens into trees. The function does not need any properties of character, token and tree, and thus has a polymorphic type:

\[
\text{twopass} \quad :: \quad (\text{Parser } a \, b) \, (\text{Parser } b \, c) \rightarrow \text{Parser } a \, c \\
\text{twopass} \quad \text{lex} \, \text{synt} = \; p \\
\quad \text{where} \quad p \, \text{xs} = \begin{cases} \\
\quad \; (\text{rest}, \text{tree}) \\
\quad \; (\text{rest}, \text{tokens}) \leftarrow \ast \leftarrow \text{lex} \, \text{xs} \\
\quad \quad (\_, \text{tree}) \leftarrow \text{just synt tokens} \\
\end{cases}
\]

Using this function, we can finally parse a string from the language that was described by a BNF grammar:

\[
\begin{align*}
\text{blockgram} &= [\text{"BLOCK ::= begin BLOCK end BLOCK | ."}] \\
\text{upperIdent} &= \langle \! + \! \rangle \, (\text{sat}\text{i}\text{fy isUpper}) \\
\text{lowerIdent} &= \langle \! + \! \rangle \, (\text{sat}\text{i}\text{fy isLower}) \\
\text{block4t} &= (\text{upperIdent}, \text{lowerIdent}, \text{blockgram}, \text{Nont } [\text{"BLOCK"]}) \\
\text{final} &= \text{twopass } (\text{sp lowerIdent } \langle \! @ \! \text{Term} \rangle) \, (\text{parsgen block4t}) \\
\text{input} &= [\text{"begin end begin begin end end"}]
\end{align*}
\]

This can really be used in a program like:

```
Start = some final input
```

**References**


**Exercises**

5.1 Since satisfy is a generalization of symbol, the function symbol could have been defined as an instance of satisfy. How can this be done?

5.2 When defining the priority of the <\|> operator, using the infixr keyword we also specified that the operator associates to the right. Why is this a better choice than association to the left?
5.3 Define the function just using a list comprehension instead of the filter function.

5.4 Why don't we use a four-tuple in the lambda pattern in section 5.6 instead of a two-tuple with as second element a two-tuple with as second element a two-tuple?

5.5 Why is the function K, which is defined by K x y = x in the standard environment, needed in the function parens in section 5.6? Can you write the second alternative more concisely without using K and @?

5.6 The parentheses around open &> parens close in the first alternative, of the function parens in section 6 are required in spite of our clever priorities. What would happen if we left them out?

5.7 The function foldparens is a generalization of parens and nesting. Write the latter two as an application of foldparens.

5.8 What would happen if we omit the just transformer in the examples in section 5.6?

5.9 Write a parser that yields the numbers of parentheses pairs in a given sequence of characters.

5.10 Redefine the parser token in terms of symbol and &>: symbol should be used to parse the individual elements of the token.

5.11 Consider application of the parser &> (symbol 'a') to the string ['aaa']. In what order do the four possible parsings appear in the list of successes?

5.12 As another variation on the theme repetition, define a parser sequence combinator that transforms a list of parsers for some type into a parser yielding a list of elements of that type. Also define a combinator choice that iterates the operator &>.

5.13 As an application of sequence, define the function token that was discussed in section 2.

5.14 Let the parser for reals recognize an optional exponent.

5.15 The parser sp given in section 5 is written in an ad-hoc way. Define this parser using parsing combinators to recognize spaces and to discard them.

5.16 Is it correct to use chainr in the parser for expressions instead of chainl?

5.17 Define the operator &: using the definition of &:.

5.18 Write a parser that computes the value of constant parts (containing no identifiers) in the expressions. The parser should yield ([], Int 3) in response to the input ['1+2'].

5.19 Extend the given parser fragments to a complete parser for Tiny, generating error messages whenever appropriate.

5.20 Extend the given parser fragments to a complete parser for Tiny, generating error messages whenever appropriate and doing error recovery from a number of errors. Consider to change skip such that it uses perror to indicate that error recovery fails.

5.21 What is the &: K [] transformation in the function ParsGram used for?

5.22 We used uppercase and lowercase identifiers to distinguish between nonterminals an terminals. If the namespaces of nonterminals and terminals overlap, we have to adopt other mechanisms to distinguish them, for example angle brackets around nonterminals and quotes around terminals. How can this be done?

5.23 Make a parser for your favourite language.