Part II

Chapter 5
Parser Combinators

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

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 test 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:

```
::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:

```
::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`:

```
::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:

```
::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 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:

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

or if you prefer meaningful identifiers over conciseness:

```
::Parser symbol result :== [symbol] -> [([symbol],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 `Char` in this case, and as a parse tree we also simply use a `Char`:

```
symbola :: Parser Char Char
symbola = p
  where p ['a':xs] = [(xs, 'a')]
        p _        = []
```

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 operate on strings other than characters, so that it can be used in other applications than character oriented ones. The only prerequisite is that the symbols to be parsed can be tested for equality. In Clean, this is indicated by the `Eq` predicate in the type of the function:

```
symbol :: s -> Parser s s  | Eq s
symbol s = p
  where p [x:xs] | s == x = [(xs, x)]
        p _               = []
```

The function `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 `token`.

```
token :: [s] -> Parser s [s] | Eq s
token k = p
  where p xs | k == take n xs = [(drop n xs, k)]
                              = []
        n = length k
```

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. However, we do need an equality test on the input string type. This is reflected in the type of `token`.

The function `token` is a generalization of the `symbol` function, in that it recognizes more than one character. The combinator `symbol` can be written in terms of `token` as:

```
symbol s = token [s]
```

Another generalization of `symbol` is a function which may, depending on the input, return different parse results. The function `satisfy` is an example of this. Where the `symbol` function tests for equality to a given symbol, in `satisfy` an arbitrary predicate `t` can be specified. Again, `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 succesful 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 ε. In this tradition, we will define a function `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.

```
epsilon :: Parser s [r]
epsilon = p
  where p xs = [(xs,[])]
```

A variation is the function `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)

```
succeed :: r -> Parser s r
succeed v = p
  where p xs = [(xs, v)]
```

Of course, `epsilon` can be defined using `succeed`:

```
epsilon` :: Parser s [r]
epsilon` = succeed []
```

Dual to the function `succeed` is the function `fail`, that fails to recognize any symbol on the input string. It always returns an empty list of successes:

```
fail :: Parser s r
fail = p
  where p xs = []
```

We will need this trivial parser as a neutral element for `foldr` later. Note the difference with `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`, `*` 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 ε. Grouping is indicated by { and }.

As example we show a simple grammar for expressions:

```
Expr          ::=   Term { Operator Term }.
```

```
Term        ::=  Digit + [ '.' Digit +]
             |   '(' Expr ')'.
Operator    ::=  '+' | '-' | '*' | '/'.
```

The meta-grammar, the grammar of all grammars, is:

```
Grammar     ::=  Rule +.
Rule        ::=  NonTerm '::=' Exp '.'.
Exp         ::=  NonTerm       |  '"' Term '"'
             |   Exp +         |  '{' 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 p1 and p2. 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 p1 must be applied to the input. After that, p2 is applied to the rest of the input string xs1. This rest input is part of the result of p1. Because p1 yields a list of solutions, we use a list comprehension in which p2 is applied to all rest strings in the list:

```
(<&>) infixr 6 :: (Parser s a) (Parser s b) -> Parser s (a,b)
(<&>) p1 p2 = p
    where p xs = [  (xs2,(v1,v2))
                 \\ (xs1,v1) <- p1 xs
                 ,  (xs2,v2) <- p2 xs1
                 ]
```

The result of the function is a list of all possible tuples (xs2,(v1,v2)) with rest string xs2, where v1 is the parse tree computed by p1, and where rest string xs1 is used to let p2 compute v2 and xs2.

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

```
(<|>) infixr 4 :: (Parser s a) (Parser s a) -> Parser s a
(<|>) p1 p2 = p
    where p xs = p1 xs ++ p2 xs
```

Thanks to the list of successes method, both p1 and p2 yield lists of possible parsings. To obtain all possible successes of choice between p1 and p2, 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 = symbol 'a' <&> symbol 'b' <&> symbol 'c'
```

is of type `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 `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: `sp` lets a given parser neglect initial spaces, `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 `sp`. It drops spaces from the input, and then applies a given parser:

```
    sp :: (Parser Char a) -> Parser Char a
    sp p = p o dropWhile isSpace
```

The second parser transformer is `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:

```
    just :: (Parser s a) -> Parser s a
    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 result 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:

```
    (<@) infixl 5 :: (Parser s a) (a->b) -> Parser s b
    (<@) p0 f = p
        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

```
    digit :: Parser Char Int
    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 `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:

```
:: DetPars symbol result :== [symbol] -> result

some :: (Parser s a) -> DetPars s a
some p = snd o hd o just p
```

Use the function `some` with care: this function assumes that there is at least one solution, so it fails when the resulting `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:

```
parens :: Parser Char ???
parens =     (   symbol '('
             <&> parens
             <&> symbol ')'
             <&> parens
             )
          <|> epsilon
```

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

```
Parentheses  ::=  '(' Parentheses ')' Parentheses
               |   ε.
```

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 `(Char,(a,(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 `<@` in both alternatives to construct a tree of the correct type. The type of the parse tree can be for example:

```
:: Tree = Nil
        | Bin (Tree,Tree)
```

Now we can add semantic functions to the parser:

```
parens :: Parser Char Tree
parens =     (   symbol '('
             <&> parens
             <&> symbol ')'
             <&> parens
             )              <@ (\( _,(x,(_,y))) -> Bin (x,y))
          <|> epsilon       <@ K Nil
```

Using the combinator `K` from the standard environment:

```
K :: x y -> x
K x y = x
```

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:

```
(<&) infixr 6 :: (Parser s a) (Parser s b) -> Parser s a
(<&) p q = p <&> q  <@  fst

(&>) infixr 6 :: (Parser s a) (Parser s b) -> Parser s b
(&>) p q = p <&> q  <@  snd
```

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

```
open  = symbol '('
close = symbol ')'

parens :: Parser Char Tree
parens =     (open &> parens <& close) <&> parens <@ Bin
         <|> succeed 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:

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

An example of the use of `nesting` is:

```
Start = 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),(['()'],3),(['(()(()()))()'],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:

```
foldparens :: ((a,a)->a) a -> Parser Char a
foldparens f e = p
  where p =     (open &> p <& close) <&> p <@ f
            <|> succeed e
```

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
        <|> succeed []
```

The auxiliary function `list` is defined as the uncurried version of the list constructor:

```
list :: (x,[x]) -> [x]
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 `epsilon` parser is used instead of `succeed`:

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

```
natural :: Parser Char Int
natural = <*> digit <@ foldl nextDigit 0
  where nextDigit a b = a*10+b
```

Defined in this way, the `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       <@  (\x -> [x])
         <|> epsilon  <@  (\_ -> [])
```

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:

```
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.

```
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

```
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 `idenitifier ['Clean']` yields

```
[([],"Clean"),(['n'],"Clea"),(['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 `fractpart`. 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.

```
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 `<+>`:

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

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

when we define

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

The expression `idenitifier ['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:

```
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:

```
<!?> :: ((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:

```
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:

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

An useful application is:

```
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.

```
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` $(e_0,[(o_1,e_1),(o_2,e_2),(o_3,e_3)]$ should return $((e_0\ o_1\ e_1)\ o_2\ e_2)\ o_3\ e_3$. Since we cannot use infix notation here we actually have to write: $o_3\ (o_2\ (o_1\ e_0\ e_1)\ e_2)\ e_3$. You may recognize a version of `foldl` in this (albeit an uncurried one), where a tuple (o_1,e_1) from the list and intermediate result `e` are combined applying o_1 `e` e_1. We define

```
chainl ::  (Parser s a) (Parser s (a a->a)) -> Parser s a
chainl p s = p <&> <*> (s <&> p) <@ \(e0,l) -> foldl (\x (op,y) -> op x y) e0 l
```

Dual to this function is `chainr`, which applies the operators associating to the right. To obtain `chainr`, change `foldl` into `foldr`, flip the list and initial element and reorder the distribution of `<*>` over the `<&>` operators:

```
chainr :: (Parser s a) (Parser s (a a->a)) -> Parser s a
chainr p s = <*> (p <&> s) <&> p <@ \(l,e0) -> foldr (\(x,op) y -> op x y) e0 l
```

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

```
expr :: Parser Char Int
expr = chainl integer (symbol '+' <@ K (+) <|> symbol '-' <@ K (-))
```

An application of this function is:

```
Start = just expr ['54--32-1']
```

This program yields

```
[([],85)]
```

Note that using `chainr` instead of `chainl` in `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 `<?>` constructs a parser which yields an optional element. Post-processing ofter perferm a case analysis whether the element was found or not. You will therefore often need constructions like:

```
<?> p <@ f
   where f []  = a
         f [x] = b x
```

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 `<?@` of the `<@` operator, which provides a semantics for both the case that the optional construct was present and that it was not. The right argument of `<?@` 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:

```
(<?@) infixl 5 :: (Parser s [a]) (b,a->b) -> Parser s b
(<?@) p (no,yes) = p <@ f
           where  f [x] = yes x
                  f []  = no
```

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

```
natural :: Parser Char Int
natural = <+> digit <@ foldl nextDigit 0
  where nextDigit a b = a*10+b
```

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

```
fractpart :: Parser Char Real
fractpart = <*> digit <@ foldr nextDigit 0.0
  where nextDigit d n = (n + toReal d)/10.0
```

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

```
real ::  Parser Char Real
real =     (integer <@ toReal)
       <&> (<?> (symbol '.' &> fractpart) <?@ (0.0,I))
       <@ uncurry (+)
```

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 absence, the number `0.0` should be used, in there 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 `chainr` function. It was defined in the previous section using the function `<*>`. The parser yields a list of tuples `(element, operator)`, which immediately afterwards is destroyed by `foldr`. Why bothering building the list, then, anyway? We can apply the function that is folded with directly during parsing, without first building a list. For this, we need to substitute the body of `<*>` in the definition of `chainr`. We can further abbreviate the phrase `p <|> epsilon` by `<?> p`. By directly applying the function that was previously used during `foldr` we obtain:

```
chainr` :: (Parser s a) (Parser s (a a->a)) -> Parser s a
chainr` p s  = q
 where q = p <&> (<?> (s <&> q) <?@ (I,\(op,y) x -> op x y) ) <@ \(x,op) -> op x
```

5.9 Arithmetical expressions

In this section we will use the parser combinators in a concrete application. We will develop a parser for arithmetical expressions, of which parse trees are of type `Expr`. Operators will be stored as functions with the corresponding names.

```
::Expr = Int Int
       | Var String
       | Fun String [Expr]
```

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 variable, a function call, or an expression between parentheses.

```
expr    ::=  term { { '+' | '-' } term }* .
term    ::=  fact { { '*' | '/' } fact }* .
fact    ::=  integer
          |  identifier [ '(' exp { ',' expr } * ')' ]
          |  '(' expr ')' .
```

This grammar is represented in the functions below:

```
fact :: Parser Char Expr
fact =     integer  <@  Int
       <|> identifier
           <&> ( <!?> (parenthesized (commaList expr))
                 <?@ (Var,flip Fun))
           <@  (\(x,op) -> op x)
       <|> parenthesized expr
```

The first alternative is a constant, which is fed into the semantic function `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 `Var` is applied, in presence the function `Fun`. The function `flip` is used to place the function name between the constructor `Fun` and the arguments.

```
flip :: (a b -> c) b a -> c
flip f b a = f a b
```

For the third alternative of `fact` there is no semantic function; the meaning of an expression between parentheses is the same as that of the expression without parentheses. If you insists of adding such a function you can use `<@ I`.

For the definition of a term as a list of factors separated by multiplicative operators we will use the function `chainl`:

```
term :: Parser Char Expr
term = chainl fact ((symbol '*' <|> symbol '/') <@ mkFun)

mkFun :: a Expr Expr -> Expr | toString a
mkFun n x y = Fun (toString n) [x,y]
```

Recall that `chainl` repeatedly recognizes its first parameter (`fact`), separated by its second parameter (an `*` or `/`). The parse trees for the individual factors are joined by the function `mkFun` supplied after `<@`.

The function `expr` is analogous to `term`, only with additive operators instead of multiplicative operators, and with terms instead of factors:

```
expr :: Parser Char Expr
expr = chainl term ((symbol '+' <|> symbol '-') <@ mkFun)
```

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 `term` and `expr`. The function `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 `term` and `expr` again. These are:

- The operators that are used in the second parameter of `chainl`
- The parser that is used as first parameter of `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 the is no reason to assume that operators are exactly one symbol long. So, we use `token` instead of `symbol`. Finally, we replace `mkFun` by a parameter.

```
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])
```

The function `choice` is the generalisation of `<|>` to a list of parsing alternatives developed in exercise 12.

If furthermore we define as shorthand:

```
add_op = [['+'],['-']]
mul_op = [['*'],['/']]
```

then `expr` and `term` can be defined as applications of `gen`:

```
expr = gen mkFun add_op term
term = gen mkFun mul_op fact
```

By expanding the definition of `term` in that of `expr` we obtain:

```
expr = gen mkFun add_op (gen mkFun mul_op fact)
```

Which an experienced functional programmer immediately recognizes as an application of `foldr`. The function to apply to the list elements is `gen mkFun`, the unit element for an empty list is `fact` and the list of items to process is `[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:

```
(<&=>)  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[1]:

```
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 parantheses. Using the new parser combinator it can be defined as:

[1]In Clean 1.2 the parentheses around the lambda expressions can be omitted.

```
nesting :: Parser Char Int
nesting =     (open &> nesting <& close) <&=> (\x ->
              nesting                    <&=> (\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:

```
nesting :: Parser Char Int
nesting =     (open &> nesting <& close) <&=> (\x ->
              nesting                    <@   (\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 '='  &>
       sp expr      <&
       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 include 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.

```
pFun2 :: Parser Char Definition
pFun2 = pAlt2 (\_->True) <&=> (\a=:(Alt f _ _) ->
        <!*> (pAlt2 (\g->f==g))
        <@ (\r -> FunDef [a:r]))

pAlt2 :: (String -> Bool) -> (Parser Char Alt)
pAlt2 p = Fsymb <&=> (\f ->
          if (p f) ( <!*> pattern   <&=> (\args ->
                     spsymbol '='    &>
                     sp expr        <&
                     spsymbol ';'   <@   (\b ->
                     Alt f args b))
                   )
                   fail)
```

However, it is more elegant to do impose this kind of conditions with a new parser combinator. The combinator `<??>` is defined as:

```
(<??>) infix 9 :: (Parser s a) (a->Bool) -> Parser s a
(<??>) p f = \xs -> [   t
                    \\  t=:(_,v) <- p xs
                    |   f v
                    ]
```

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

```
pFun =  pAlt <&=> (\a=:(Alt f _ _) ->
        <!*> (pAlt <??> (\(Alt g _ _) -> f==g))
        <@ (\r -> FunDef [a:r]))
```

Although this definition is very elegant, it is slightly less efficient than the parser `pFun2`. Parser `pFun parses` the entire alternative before the test is executed, while `pFun2` evaluates the test as soon as the function symbol is parsed. The parser `pAlt2` can be written more elegant using the conditional combinator `<??>`.

5.13 Common traps

The parsers themselves can be read equally well as grammar rules. At some places the specified parsers look more complicated then 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:

```
exp     ::=   exp oper exp  |  '(' exp ')'  |  integer .
oper    ::=   '+'  |  '-' .
```

When we would write naively a parser for this syntax this will look like:

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

oper :: Parser Char (a a -> a) | +,- a
oper = 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

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

The corresponding parser is

```
expres2 :: Parser Char Int
expres2 =     integer <&=> (\x  ->
              oper    <&=> (\op ->
              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 evaluted to `2` insted of `-4`. A correct way to construct such a parsers for expressions is.

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

elem :: Parser Char Int
elem = integer <|> parenthesized expres3
```

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.[2] 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.

```
expres4 :: Parser Char Int
expres4 =     integer <&=> (\x  ->
                  succeed x
              <|> oper    <&=> (\op ->
                  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.

```
(<!>) 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 <|>`

p3). The last expression yields the first result of the concatenation of the results of p1, p2 and p3. 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 situation 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 an simple imperative language:

```
tiny            ::=  'BEGIN' statements 'END'.
statements      ::=  stmt [ ';' statements ].
stmt            ::=  identifier ':=' expression
                 |   'IF' expression 'THEN' stmts [ 'ELSE' stmts ]
                 |   'WHILE' expression 'DO' stmts
                 |   'PRINT' expression
                 |   'VAR' identifier [ integer ].
stmts           ::=  stmt | 'BEGIN' statements 'END' .
```

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

```
:: Tiny     :== [TStatement]

:: TStatement   =   Declare Variable Int
                |   Assign  Variable Expr
                |   If      Expr [TStatement] [TStatement]
                |   While   Expr [TStatement]
                |   Print   Expr
```

As soon as the parser has seen the keyword WHILE it must detect a complete while-statement as the next item. The keyword WHILE must always be followed by an expression, the keyword DO, 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 <!> instead of <|>.

```
stmt :: Parser Char TStatement
stmt = IFstmt <!> WHILEstmt <!> PRINTstmt <!> ASSIGNstmt <!> declaration

WHILEstmt :: Parser Char TStatement
WHILEstmt =     WHILEtok
             &> (expr  <!> pError "WHILE: condition expected")
            <&> (DOtok <!> pError "WHILE: DO expected")
             &> (stmts <!> pError "WHILE: Body expected")
                <@ (\(c,b) -> While c b)

stmts :: Parser Char [TStatement]
stmts =         BEGINtok
             &> statements  <!> pError "stmts expected")
            <&  (ENDtok     <!> pError "END expected")
        <|> stmt <@ (\s -> [s])

statements :: Parser Char [TStatement]
statements = listOf stmt (spsymbol ';'
```

Using:

```
WHILEtok   = sptoken ['WHILE']
DOtok      = sptoken ['DO']
BEGINtok   = sptoken ['BEGIN']
```

```
ENDtok     = sptoken ['END']
sptoken t  = sp (token t)
spsymbol s = sp (symbol s)
```

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.

```
pError :: String -> Parser s r | ToString s
pError mes = q
  where q xs = abort ("Parse error: "+mes+". Input = "+show (take 20 xs)+"\n")

show :: [s] -> String | ToString s
show list = "["+ showTl list
  where showTl []    = "]"
        showTl [a:r] = toString a + "," + showTl r
```

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:

```
:: TStatement   =   Declare Variable Int
                |   Assign  Variable Expr
                |   If      Expr [TStatement] [TStatement]
                |   While   Expr [TStatement]
                |   Print   Expr
                |   StmtError   // The error indication
```

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:

```
WHILEstmt =   WHILEtok
               &> (expr  <!> succeed ExprError)
              <&> DOtok
               &> (stmts <!> succeed [StmtError])
                  <@ (\(c,b) -> While c b)
```

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`.

```
skip :: Int (Parser s x) (Parser s r) -> Parser s r
skip n until result = p n
  where p 0 xs = []                           // n symbols are skipped fail
        p n xs = case until xs of
                   [] -> case xs of             // parser until failed
                          []     -> []          // empty input: skipping fails
                          [_:tl] -> p (n-1) tl  // skip current input token
                   ne -> result xs              // until succeeds: apply result parser
```

The function `skipAlso` is similar. The difference with `skip` is that also the input fragment that stopped the skipping is removed from the input.

```
skipAlso :: Int (Parser s x) (Parser s r) -> Parser s r
skipAlso n until result = skip n until (until &> result)
```

Using this function the error handling in the parser for the while-loop can be improved. We assume that only one error occurs is 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:

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

endStmt :: r -> Parser Char r
endStmt r = skipAlso 100
                    (    ENDtok <@ K [StmtError]
                     <|> (BEGINtok &> (listOf stmt (spsymbol ';') <&  ENDtok)))
                    (succeed r)
```

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

```
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 `VAR "n"` as condition. The parser `DOtok` fails since the input contains `[' >> 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.

```
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:

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

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.

```
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 <@ K [StmtError]
                      <|> (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 -> b | Eq, toString a
assoc [(u,v):ws] x | x == 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
```

Grammars 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 in 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 `sptoken` 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']
                       , Nont ['BLOCK']
                       , Term ['end']
                       , Nont ['BLOCK']
                       ]
                     , []
                     ]
   )
 ]
```

Parse trees

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 `bfn` 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:

```
parsGram :: Gram Symbol -> Parser Symbol RTree
parsGram gram start = parsSym start
  where
    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 `parseAlt`, 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
```

```
    parsgen (nontp, termp, bnfstring, start)
        = some (bnf nontp termp <@ parsGram) bnfstring start
```

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:

```
    twopass :: (Parser a b) (Parser b c) -> Parser a c
    twopass lex synt = p
      where p xs = [  (rest,tree)
                   \\ (rest,tokens) <- <*> lex xs
                   ,  (_,tree)      <- just synt tokens
                   ]
```

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

```
    blockgram  = ['BLOCK ::= begin BLOCK end BLOCK | .']
    upperIdent = <!+> (satisfy isUpper)
    lowerIdent = <!+> (satisfy isLower)
    block4t    = (upperIdent, lowerIdent, blockgram, Nont ['BLOCK'])
    final      = twopass (sp lowerIdent <@ Term) (parsgen block4t)
    input      = ['begin end begin begin end end']
```

This can really be used in a program like:

```
    Start = some final input
```

References

Bird, R. and P. Wadler, *Introduction to Functional Programming*, Prentice Hall, 1988.

Burge, W.H. *Parsing*. In Recursive Programming Techniques, Addison-Wesley, 1975.

Fokker, J.: *Functional Parsers*. In: Advanced Functional Programming (Jeuring and Meijer eds.) LNCS 925. 1995.

Hutton, Graham. *Higher-order functions for parsing*. J. Functional Programming 2 pp: 323–343. 1992.

Wadler, P. *How to replace failure by a list of successes: a method for exception handling, backtracking, and pattern matching in lazy functional languages*. In Functional Programming Languages and Computer Architecture, (J.P. Jouannaud, ed.), Springer, 1985 (LNCS 201), pp. 113–128.

Wadler, Philip. *Monads for functional programming*. In Program design calculi, proc. of the Marktoberdorf Summer School, (M. Broy. ed.), Springer, 1992.

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 6are 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?

5.23 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?

5.22 Make a parser for your favourite language.