Functional Parsers

Jeroen Fokker

Dept. of Computer Science, Utrecht University
P.O.Box 80.089, 3508 TB Utrecht, The Netherlands
e-mail jeroen@cs.ruu.nl

Abstract. In an informal way the 'list of successes' method for writing parsers using a lazy functional language (Gofer) is described. The library of higher-order functions (known as 'parser combinators') that is developed is used for writing parsers for nested parentheses and operator expressions with an arbitrary number of priorities. The method is applied on itself to write a parser for grammars, that yields a parser for the language of the grammar. In the text exercises are provided, the solutions of which are given at the end of the article.

1 Introduction

This article is an informal introduction to writing parsers in a lazy functional language using 'parser combinators'. Most of the techniques have been described by Burge [2], Wadler [5] and Hutton [3]. Recently, the use of so-called *monads* has become quite popular in connection with parser combinators [6, 7]. We will not use them in this article, however, to show that no magic is involved in using parser combinators. You are nevertheless encouraged to study monads at some time, because they form a useful generalization of the techniques described here.

In this article we stick to standard functional language constructs like higher-order functions, lists, and algebraic types. All programs are written in Gofer [4]. List comprehensions are used in a few places, but they are not essential, and could be easily rephrased using the `map`, `filter` and `concat` functions [1]. Type classes are only used for overloading the equality and arithmetic operators.

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.

In section 4 the first parser combinators are introduced, which can be used for sequentially and alternatively combining parsers. In section 5 some functions are defined, which make it possible to calculate a value during parsing. You may use these functions for what traditionally is called 'defining semantic functions': some useful meaning can be associated to syntactic structures. As an example, in section 6 we construct a parser for strings of matching parentheses, where different semantic values are calculated: a tree describing the structure, and an integer indicating the nesting depth.

In sections 7 and 8 we introduce some new parser combinators. Not only these will make life easier later, but also their definitions are nice examples of using parser combinators. A real application is given in section 9, where a parser for arithmetical expressions is developed. Next, the expression parser is generalized to expressions with an arbitrary number of precedence levels. This is done without coding the priorities of operators as integers, and we will avoid using indices and ellipses.

In the last section parser combinators are used to parse the string representation of a grammar. As a semantic value, a parser is derived for the language of the grammar, which in turn can be applied to an input string. Thus we will essentially have a parser generator.

2 The type 'Parser'

The *parsing problem* is: given a string, construct a tree that describes the structure of the string. In a functional language we can define a datatype `Tree`. A parser could be implemented by function of the following type:

```
type Parser  =  String -> Tree
```

For parsing substructures, a parser could call other parsers, or itself recursively. These calls need not only communicate their result, but also which part of the input string is left unprocessed. As this cannot be done using a global variable, 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:

```
type Parser  =  String -> ( String, Tree )
```

The type `String` is defined in the standard prelude as a list of characters. 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 `a` for it:

```
type Parser a  =  String -> ( String , a )
```

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 whil parsing it. The type definition of `Parser` therefore had better be:

```
type Parser a  =  String -> [ (String,a) ]
```

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 method is called the *list of successes* method, described by Wadler [5]. It can be used in situations where in other languages you would use backtracking techniques. In the Bird and Wadler textbook it is used to solve combinatorial problems like the eight queens problem [1]. 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, not all elements of the list are determined if only the first value is needed, so there will be no loss of efficiency. Lazy evaluation provides a backtracking approach to finding the first solution.

Parsers with the type described so far operate on strings, that is lists of characters. There is however no reason for not allowing parsing strings of elements other than characters. You may imagine a situation in which a preprocessor prepares a list of tokens, which is subsequently parsed. To cater for this situation, as a final refinement of the parser type we again abstract from a type: that of the elements of the input string. Calling it `a`, and the result type `b`, the type of parsers is defined by:

```
type Parser a b  =  [a] -> [([a],b)]
```

or if you prefer meaningful identifiers over conciseness:

```
type Parser symbol result  =  [symbol] -> [([symbol],result)]
```

We will use this type definition in the rest of this article.

3 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 []                  =  []
symbola (x:xs) | x=='a'     =  [ (xs, 'a') ]
               | otherwise  =  []
```

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 Gofer, this is indicated by the `Eq` predicate in the type of the function:

```
symbol  ::  Eq s  =>  s -> Parser s s
symbol a []                    =  []
symbol a (x:xs)  |  a==x       =  [ (xs,x) ]
                 |  otherwise  =  []
```

As usual, there are a number of ways to define the same function. If you like list comprehensions, you might prefer the following definition:

```
symbol a []      =  []
symbol a (x:xs)  =  [ (xs,a) | a==x ]
```

In Gofer, a list comprehension with no generators but only a condition is defined to be empty or singleton, depending on the condition.

The function `symbol` is a function that, given a symbol, yields a parser for that symbol. A parser in turn is a function, too. This is why two parameters appear in the definition of `symbol`.

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 k xs  |  k==take n xs  =  [ (drop n xs, k) ]
            |  otherwise     =  []
                       where  n = length k
```

As in the case of the `symbol` function we have parameterized this function with the string to be recognized, effectively making it into a family of functions. Of course, this function is not confined to strings of characters. However, we do need an equality test on the input string type; the type of `token` is:

```
token  ::  Eq [s]  =>  [s] -> Parser s [s]
```

The function `token` is a generalization of the `symbol` function, in that it recognizes more than one character.

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 can be specified. Again, `satisfy` effectively is a family of parser functions. It is defined here using the list comprehension notation:

```
satisfy  ::  (s->Bool) -> Parser s s
satisfy p []     =  []
satisfy p (x:xs) =  [ (xs,x) | p x ]
```

Exercise 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?

In books on grammar theory an empty string is often called 'epsilon'. 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 parse tree and unmodified input. A zero-tuple can be used as a result value: `()` is the only value of the type `()`.

```
epsilon    ::  Parser s ()
epsilon 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 xs   =  [ (xs,v) ]
```

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

```
epsilon  ::  Parser s ()
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 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).

4 Parser combinators

Using the elementary parsers from the previous section, parsers can be constructed for terminal symbols from a grammar. More interesting are parsers for *non*terminal symbols. Of course, you could write these by hand, 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 so as to minimize parentheses in practical situations:

```
infixr 6 <*>
infixr 4 <|>
```

Both operators have two parsers as parameter, and yield a parser as result. By again combining the result with other parsers, you may construct even more involved parsers.

In the definitions below, the functions operate on parsers `p1` and `p2`. Apart from the parameters `p1` and `p2`, the function operates on a string, which can be thought of as the string that is parsed by the parser that is the result of combining `p1` and `p2`.

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 string part of the result. Because `p1` yields a *list* of solutions, we use a list comprehension in which `p2` is applied to all rest strings in the list:

```
(<*>)          ::  Parser s a -> Parser s b -> Parser s (a,b)
(p1 <*> p2) xs  =  [  (xs2, (v1,v2))
                   |  (xs1, v1)  <-  p1 xs
                   ,  (xs2, v2)  <-  p2 xs1
                   ]
```

The result of the function is a list of all possible tuples `(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 `<|>`:

```
(<|>)          ::  Parser s a -> Parser s a -> Parser s a
(p1 <|> p2) 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.

Exercise 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?

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` where

```
p  =  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 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 . dropWhile (==' ')
```

or if you prefer functional definitions:

```
sp   =  ( . dropWhile (==' '))
```

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 null rest strings. Because the rest string is the first component of the list, the function can be defined as:

```
just     ::  Parser s a -> Parser s a
just p    =  filter (null.fst) . p
```

Exercise 3. Define the function `just` using a list comprehension instead of the `filter` function.

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. 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:

```
infixr 5 <@
(<@)          ::  Parser s a -> (a->b) -> Parser s b
(p <@ f) xs    =  [ (ys, f v)
                  | (ys,   v) <- p 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  <@  f
         where  f c = ord c - ord '0'
```

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:

```
type DetPars symbol result  =  [symbol] -> result
some   ::  Parser s a -> DetPars s a
some p  =  snd . head . just p
```

Use the `some` function 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.

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. 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:

```
data 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  <@  const Nil
```

The rather obscure text `\(_,(x,(_,y)))` is a lambda pattern describing a function with as first parameter a tuple containing the four parts of the first alternative, of which only the second and fourth matter.

Exercise 4. Why don't we use a four-tuple in the lambda pattern instead of a tuple with as second element a tuple with as second element a tuple?

Exercise 5. Why is the function `const`, which is defined by `const x y = x` in the prelude, needed? Can you write the second alternative more concisely without using `const` and `<@`?

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 more situations. These operators behave the same as `<*>`, except that they discard the result of one of their two parser arguments:

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

(*>)    ::  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:

```
infixr 6  <*> , <*  , *>
infixl 5  <@
infixr 4  <|>
```

we minimize on the number of parentheses needed.

Exercise 6. The parentheses around `open*>parens<*close` in the first alternative, are required in spite of our clever priorities. What would happen if we left them out?

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 nesting depth of nested parentheses:

```
nesting  ::  Parser Char Int
nesting   =  (open *> nesting <* close) <*> nesting  <@  f
             <|> succeed 0
      where  f (x,y) = (1+x) `max` y
```

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 and constant 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
```

Exercise 7. The function `foldparens` is a generalization of `parens` and `nesting`. Write the latter two as an instantiation of the former.

A session in which `nesting` is used may look like this:

```
? just nesting "()(())()"
[(2,[])]
? just nesting "())"
[]
```

Indeed `nesting` only recognizes correctly formed nested parentheses, and calculates the nesting depth on the fly.

Exercise 8. What would happen if we omit the `just` transformer in these examples?

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 for a construction, `many` makes a parser for zero or more occurrences of that construction:

```
many   ::  Parser s a  -> Parser s [a]
many p  =  p <*> many p  <@  list
           <|> succeed []
```

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

```
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`:

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

Exercise 9. 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 `many`, also the result of `<*>` is postprocessed. Define an utility function `<:*>` for this case, and use it to simplify the definition of `many` even more.

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

Exercise 10. Consider application of the parser `many (symbol 'a')` to the string `"aaa"`. In what order do the four possible parsings appear in the list of successes?

An example in which the `many` combinator can be used is parsing of a natural number:

```
natural  ::  Parser Char Int
natural   =  many digit  <@  foldl f 0
      where  f 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 `many1` parser combinator, which accepts one or more occurrences of a construction.

Exercise 11. Define the `many1` parser combinator.

Another combinator that you may know from other formalisms is the `option` combinator. The constructed parser generates a list with zero or one element, depending on whether the construction was recognized or not.

```
option   ::  Parser s a -> Parser s [a]
option p  =      p        <@  (\x->[x])
             <|> epsilon  <@  (\x->[] )
```

For aesthetic reasons we used `epsilon` in this definition; another way to write the second alternative is `succeed []`.

The combinators `many`, `many1` and `option` are classical in compiler 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, a body, and a closing token, 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
```

Special cases 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 <:*> many (s *> p)  <|>  succeed []
```

Useful instatiations are:

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

Exercise 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 `<|>`.

Exercise 13. As an application of `sequence`, define the function `token` that was discussed in section 3.

A somewhat more complicated variant of the function `listOf` is the case where the separators carry a meaning themselves. For example, an 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 <*> many (s <*> p)  <@  f
```

The function `f` should operate on an element and a list of tuples, each containing an operator and an element. For example, $f(e_0, [(\oplus_1, e_1), (\oplus_2, e_2), (\oplus_3, e_3)])$ should return $((e_o \oplus_1 e_1) \oplus_2 e_2) \oplus_3 e_3$. You may recognize a version of `foldl` in this (albeit an uncurried one), where a tuple $(\oplus, y)$ from the list and intermediate result x are combined applying $x \oplus y$. If we define

```
ap2 (op,y) x = x `op` y
```

or even

```
ap2 (op,y)  =  (`op` y)
```

then we may define

```
chainl     :: Parser s a -> Parser s (a->a->a) -> Parser s a
chainl p s  = p <*> many (s <*> p)
              <@  uncurry (foldl (flip ap2))
```

Dual to this function is `chainr`, which applies the operators associating to the right.

Exercise 14. Try to define `chainr`. The definition is beautifully symmetric to `chainl`, but you only experience the beauty when you discover it yourself...

8 Analyzing options

The `option` function constructs a parser which yields a list of elements: an empty list if the optional construct was not recognized, and a singleton list if it was present. Postprocessing functions may therefore safely assume that the list consists of zero or one element, and will in practice do a case analysis. You will therefore often need constructions like:

```
option 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:

```
p <?@ (no,yes)  =  p <@ f
           where  f []   =  no
                  f [x] =  yes x
```

For a practical use of this, let's extend the parser for natural numbers to floating point numbers:

```
natural  ::  Parser Char Int
natural   =  many digit  <@  foldl f 0
      where  f n d = n*10 + d
```

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

```
fract    ::  Parser Char Float
fract     =  many digit  <@  foldr f 0.0
      where  f d x = (x + fromInteger d)/10.0
```

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

```
fixed    ::  Parser Char Float
fixed     =  (integer <@ fromInteger)
             <*>
             (option (symbol '.' *> fract) <?@ (0.0,id))
             <@  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.

Exercise 15. Define a parser for a (possibly negative) integer number, which consists of an optional minus sign followed by a natural number.

Exercise 16. Let the parser for floating point numbers recognize an optional exponent.

In the solution of exercise 15 you will find a nice construct, in which the first construct parsed yields a function which is subsequently applied to the second construct parsed. We can use that for yet another refinement of the **chainr** function. It was defined in the previous section using the **many** function. The parser yields a list of tuples (operator,element), 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 **many** in the definition of **chainr**. We can further abbreviate the phrase `p <|> epsilon` by `option p`. By directly applying the function that was previously used during **foldr** we obtain:

```
chainr' p s   =  q
     where  q =  p <*> (option (s <*> q) <?@ (id,ap2) )
                 <@ flip ap
```

Exercise 17. You want to try `chainl` yourself?

By the use of the `option` and **many** 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  =  many1 (satisfy isAlpha)
```

a single word may also be parsed as two identifiers. Caused by the order of the alternatives in the definition of **many**, 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 situations where from the way the grammar is built we can predict that it is hopeless to try non-greedy successes of **many**. 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 a b -> Parser a b
first p xs | null r      =  []
           | otherwise   =  [head r]
                  where  r = p xs
```

Using this function, we can create a special 'take all or nothing' version of **many**:

```
greedy   =  first . many
greedy1  =  first . many1
```

If we compose the **first** function with the `option` parser combinator:

```
compulsion  =  first . option
```

we get a parser which must accept a construction if it is present, but which does not fail if it is not present.

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`:

```
data Expr  =  Con Int
           |  Var String
           |  Fun String [Expr]
           |  Expr :+: Expr
           |  Expr :-: Expr
           |  Expr :*: Expr
           |  Expr :/: 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, variable, function call, or expression between parentheses.

This grammar is represented in the functions below:

```
fact  ::  Parser Char Expr
fact   =      integer  <@  Con
          <|> identifier
              <*> ( option (parenthesized (commaList expr))
                    <?@ (Var,flip Fun))
              <@  ap'
          <|> parenthesized expr
```

The first alternative is a constant, which is fed into the 'semantic function' `Var`. The second alternative is a variable or function call, depending on the presence of a parameterlist. In absence of the latter, the function `Var` is applied, in presence the function `Fun`. For the third alternative there is no semantic function, because the meaning of an expression between parentheses is the same as that of the expression without parentheses.

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

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

Recall that `chainr` repeatedly recognizes its first parameter (`fact`), separated by its second parameter (an $*$ or $/$). The parse trees for the individual factors are joined by the constructor functions mentioned after `<@`.

The function `expr` is analogous to `term`, only with additive operators instead of multiplicative operators, and with `term`s instead of `fact`ors:

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

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.

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 `chainr` is used in each definition, with as first parameter the function of one priority level lower.

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 and associated tree constructors that are used in the second parameter of `chainr`
- The parser that is used as first parameter of `chainr`

The generalized function will take these two differences as extra parameters: the first in the form of a list of pairs, the second in the form of a parse function:

```
type Op a  =  (Char, a->a->a)
gen       ::  [Op a] -> Parser Char a -> Parser Char a
gen ops p  =  chainr p (choice (map f ops))
       where  f (s,c) = symbol s <@ const c
```

If furthermore we define as shorthand:

```
multis  =  [ ('*',(:*:)), ('/',(:/:)) ]
addis   =  [ ('+',(:+:)), ('-',(:-:)) ]
```

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

```
expr  =  gen addis  term
term  =  gen multis fact
```

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

```
expr  =  addis `gen` (multis `gen` fact)
```

which an experienced functional programmer immediately recognizes as an application of **foldr**:

```
expr  =  foldr gen fact [addis, multis]
```

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 'list with operators of the same priority'. Also, the insertion of new priority levels is very easy.

11 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 a 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, in which a finite mapping can be represented. The function `assoc` can be used to associate a value to its image under the mapping.

```
type Env a b    =   [(a,b)]
assoc :: Eq s  =>  Env s d -> s -> d
assoc ((u,v):ws) x  |  x==u        =  v
                    |  otherwise   =  assoc ws x
```

We also define a function `mapenv` that applies a function to all images in an environment.

```
mapenv :: (a->b) -> Env s a -> Env s b
mapenv f [] = []
mapenv f ((x,v):ws) = (x,f v) : mapenv f ws
```

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

```
data Symbol  =  Term String
             |  Nont String
```

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

```
type Alt   =  [Symbol]
type Rhs   =  [Alt]
```

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

```
type 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 String -> Parser Char String
       -> Parser Char Gram
bnf nontp termp = many rule
    where rule =  (    nont
                  <*>  sptoken "::="  *> rhs <*  spsymbol '.'
                  )
          rhs  =  listOf alt (spsymbol '|')
          alt  =  many (term <|> nont)
          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:

```
test = some (bnf nont term) blockgram
     where nont = greedy1 (satisfy isUpper)
           term = greedy1 (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 in section 9. Instead, we define a generic data structure, that describes parse trees for sentences from an arbitrary grammar. We simply call them `Tree`; they are instances of multibranching trees or 'rose trees':

```
data Tree  =  Node Symbol [Tree]
```

Parsers instead of grammars Using the `bfn` function, we can easily generate values of the `Gram` 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 Tree
```

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 Tree
parsGram gram start =  parsSym start
   where
     parsSym :: Symbol -> Parser Symbol Tree
     parsSym s@(Term t)  =  symbol s <@ const []   <@ Node s
     parsSym s@(Nont n)  =  parsRhs (assoc gram s) <@ Node s
     parsAlt ::  Alt -> Parser Symbol [Tree]
     parsAlt  =  sequence . map parsSym
     parsRhs ::  Rhs -> Parser Symbol [Tree]
     parsRhs  =  choice . map parsAlt
```

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.

Exercise 18. What is the `<@ const []` transformation used for?

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 makes a `choice` from them. Finally, the function `parseAlt` generates parsers for the individual symbols in the alternative, and combines them using the `sequence` function.

A parser generator In theoretical textbooks a context-free grammar is usually described as a four-tuple (N, T, R, S) 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:

```
type SymbolSet = Parser Char String
type CFG = (SymbolSet, SymbolSet, String, 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 Tree
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) `Symbol`s and yields a parse `Tree`.

Lexical scanners The parser that is generated accepts `Symbol`s instead of `Char`s. 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 library 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 xs = [ (rest,tree)
                      | (rest,tokens) <- many 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  |  ."
block4tup =  (upperId, lowerId, blockgram, Nont "BLOCK")
upperId   =  greedy1 (satisfy isUpper)
lowerId   =  greedy1 (satisfy isLower)
final     =  twopass (sp lowerId <@ Term) (parsgen block4tup)
input     =  "begin end begin begin end end"
```

This can really be used in a session:

```
? some final input
Node (Nont "BLOCK") [Node (Term "begin") [], Node (Nont
"BLOCK") [], Node (Term "end") [], Node (Nont "BLOCK")
[Node (Term "begin") [], Node (Nont "BLOCK") [Node (Term
"begin") [], Node (Nont "BLOCK") [], Node (Term "end")
[], Node (Nont "BLOCK") []], Node (Term "end") [], Node
(Nont "BLOCK") []]]
(1061 reductions, 2722 cells)
```

Exercise 19. 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?

Exercise 20. Make a parser for your favourite language.

Acknowledgement

I would like to thank Doaitse Swierstra and Erik Meijer for their comments on a draft of this paper and stimulating ideas.

References

1. R. Bird and P. Wadler, *Introduction to Functional Programming.* Prentice Hall, 1988.
2. W.H. Burge, 'Parsing'. In *Recursive Programming Techniques*, Addison-Wesley, 1975.
3. Graham Hutton, 'Higher-order functions for parsing'. *J. Functional Programming* **2**:323–343.
4. Mark Jones, *Gofer 2.30 release notes.* `http://www.cs.nott.ac.uk:80/Department/Staff/mpj/`.
5. P. Wadler, '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.
6. Philip Wadler, 'Monads for functional programming'. In *Program design calculi, proc. of the Marktoberdorf Summer School*, (M. Broy, ed.) Springer, 1992.
7. Philip Wadler, 'Monads for functional programming'. In *Lecture notes of the First International Spring School on Advanced Programming Techniques*, (J. Jeuring, ed.) Springer, 1995.

Solutions to exercises

1. A symbol equal to a satisfies equality to a:

```
symbol a  =  satisfy  (==a)
```

2. As `<|>` is a lifted version of `++`, it is more efficiently evaluated right associative.
3. The function `just` can be written as a list comprehension:

```
just p xs  =  [ ([],v)
              | (ys,v) <- p xs
              , null ys
              ]
```

4. The operator `<*>` associates to the right, so `a <*> b <*> c <*> d` really means
`a <*> (b <*> (c <*> d))`, which explains the structure of the result.
5. The parser `epsilon` yields the empty tuple `()` as parse tree. The function `const Nil` is applied to this result, thus effectively discarding the empty tuple and substituting `Nil` for it. Instead of `epsilon <@ const Nil` we can also write `succeed Nil`.
6. Without parentheses, we obtain `open *> (parens <* (close<*>parens))`, and we would only keep the result of the first recursive use of the `parens` parser.
7. The functions `parens` and `nesting` can be written as partial parametrizations of `foldparens`, by supplying the functions to be used for the first and second alternative:

```
parens   =  foldparens Bin        Nil
nesting  =  foldparens (max.(1+)) 0
```

8. Without the `just` transformer, also partial parses are reported in the successes list

```
? nesting "()(())()"
[([],2), ("()",2), ("(())()",1), ("()(())()",0)]
? nesting "())"
[(")",1), ("())",0)]
```

9. The empty alternative is presented last, because the `<|>` combinator uses list concatenation for concatenating lists of successes. This also holds for the recursive calls; thus the 'greedy' parsing of all three `a`'s is presented first, then two `a`'s with singleton rest string, then one `a`, and finally the empty result with untouched rest string.
10. We define `<:*>` as an abbreviation of postprocessing `<*>` with the `list` function:

```
p <:*> q  =  p <*> q  <@  list
```

Then we can define

```
many p  =   p <:*> many p  <|>  succeed []
```

11. The `many1` combinator can be defined using the `many` combinator:

```
many1   ::  Parser s a -> Parser s [a]
many1 p  =  p <*> many p  <@  list
```

12.
```
sequence     ::  [Parser s a] -> Parser s [a]
sequence      =  foldr (<:*>) (succeed [])
choice       ::  [Parser s a] -> Parser s a
choice        =  foldr (<|>) fail
```

13.
```
token  ::  Eq [s]  =>  [s] -> Parser s [s]
token   =  sequence . map symbol
```

14. This was `chainl`:

```
chainl :: Parser s a -> Parser s (a->a->a) -> Parser s a
chainl p s  = p <*> many (s <*> p)
              <@ uncurry (foldl (flip ap2))
```

To obtain `chainr`, change `foldl` into `foldr`, swap `flip` and `fold`, change `ap2` into `ap1` and reorder the distribution of `many` over the `<*>` operators:

```
chainr :: Parser s a -> Parser s (a->a->a) -> Parser s a
chainr p s  = many (p <*> s) <*> p
              <@ uncurry (flip (foldr ap1))
```

The auxiliary functions used are:

```
ap2 (op,y)   = (`op` y)
ap1 (x,op)   = (x `op`)
```

15. Easiest is to do the case analysis explicitly:

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

But nicest is to use the `<?@` operator, yielding the identity or negation function in absence or presence of the minus sign, which is finally applied to the natural number:

```
integer ::  Parser Char Int
integer  =  (option (symbol '-')  <?@  (id,const negate))
            <*> natural
            <@  ap
     where ap (f,x) = f x
```

16. A floating point number is a fixed point number with an optional exponent part:

```
float  ::  Parser Char Float
float   =  fixed
           <*>
           (option (symbol 'E' *> integer) <?@ (0,id) )
           <@ f
    where  f (m,e)  =  m * power e
           power e | e<0       = 1.0 / power (-e)
                   | otherwise = fromInteger(10^e)
```

17. This would be nice:

```
chainl' p s   =  q
     where  q = (option (q <*> s) <?@ (id,ap1) )
                <*> p <@ ap
```

   Alas, this function will not terminate...
18. The symbol `s` that is parsed is discarded, and an empty list is substituted for it. Then the function `Node s` is applied to this empty list, resulting in `Node s []`, which is a terminal node in the parse tree.