Parser Combinators

João Saraiva

Department of Informatics & HASLab/INESC TEC
Universidade do Minho, Braga, Portugal
saraiva@di.uminho.pt

Context-free grammars are used to define the syntactic characteristics of programming languages. Traditionally, context-free grammars are used as the underlying language representation of parser generator tools. Such tools accept as input a context-free grammar, written in a BNF or extended BNF like notation, and produce a program that implements the parsing process of the language. There are many parser generator tools, with Yacc being the most popular of them[1].

In this chapter we show how, in a purely functional setting, we can write functions that "*are*" the context-free grammar itself. That is, we define a small set of *parsing functions* that can be used to construct *recursive descendent parsers* [2]. The notation used for those functions is very similar to the Backus Naur formalism, and consequently, writing the parsing functions is as simple as writing the correspondent context-free grammar. The parsing functions are known as *parser combinators* [3, 4, **?**]. The combinators are higher-order operators serving as "*glue*" to assemble functions into more powerful combinations.

We present parser combinators not only because they elegantly model generalised LL parsers in a purely functional language, but because they also nicely illustrate the concepts of higher-order functions, polymorphic functions and lazy evaluation. Moreover, the design of a parser combinator library is also a good example to show how a *domain specific language* can be efficiently embeded within a general purpose programming language. In our case the domain specific language is the BNF notation and the general purpose language is Haskell. By embeding BNF in Haskell we mean that the Haskell language is *extended* with the BNF one. Actually, we do not extend Haskell with BNF at all, but we make it look like if that was the case. There are several examples of constructing combinator libraries to embed other *domain specific languages* within a general purpose language. In Haskell there have been constructed several combinator libraries: languages for pretty printing [**?**], the SQL language [**?**], the XML language [**?**], the attribute grammar formalism [**?**], language-based editors [], internet programming [**?**], etc. The use of higher-ordeness, polymorphism and lazy evaluation are the basis for constructing all these combinator libraries.

[1] There are Yacc like tools for most programming languages. Happy [1] is the Yacc like parser generator for Haskell.

1 Parsing with Combinators

Before we proceed to define the parser combinator functions, let us start by defining the type of a parser function. We recall the type of a parser presented in section **??**, *i.e.*, a function that takes as argument a list of tokens and returns a tree representing the abstract structure of the input. Thus, a (complete) parser is a function of the following type:

$$parser :: [\,Token] \rightarrow Tree$$

A recursive descendent parser is organized as a set of (parser) functions each of which processing a particular (sub) syntactic structure of the input. Consequently, each parser function consumes part of the input only (recall figure **??**). So, a parser function has to return not only the result of parsing its part of the input, but also the unconsumed part of that input. So, as our first attempt to define the type of a parser function of a recursive descedent parser we have:

$$\textbf{type}\ \mathsf{Parser} = [\,Token] \rightarrow (Tree, [\,Token])$$

But, sometimes a parser function might not be able to produce a result at all. Rather than defining a special type for the success or failure of a parser function, we choose to let a parser return a list as result: the *empty list* denotes *failure*, and a *singleton list* indicates *success*. The definition of type Parser is hence:

$$\textbf{type}\ \mathsf{Parser} = [\,Token] \rightarrow [(Tree, [\,Token])]$$

Observe that, a singleton list is produced if the underlying context-free grammar is unambiguous (recall definition **??**). For ambiguous grammars there are not one, but, by definition, several ways of parsing a sentence. Each of the "ways" of parsing gives a different parse result, *i.e.*, a different parse tree (see Figure **??**). In this case, alternative parsing results constitute the elements of the list. Furthermore, in a lazy evaluation setting if only a single parsing result is required, then, we can take the head of the list and all the other possible results are not computed at all. Thus, there is no loss of efficiency[2].

Since we want to specify the type of any parser, regardless of the kind of symbols and results involved, the types *Token* and *Tree* are included as type parameters, s and r respectively. Therefore, the type definition of Parser becomes:

$$\textbf{type}\ \mathsf{Parser}\ s\ r = [s] \rightarrow [(r, [s])]$$

For example, a parser for expressions might have type Parser *Char Exp* indicating that it takes a string of characters and constructs an expression tree (denoted by type *Exp*).

[2] This method is known as the "list of successes" and was introduced by Wadler [**?**].

Now, let us focus on defining the parser combinator functions. We distinguish two kinds of parser combinators: The *basic combinators* that are a small set of basic parsing functions, and the *parser combinators* that are functions that combine parsers into more complex parsers.

The basic combinators are the building blocks of combinator parsers, exactly as grammar symbols are the building blocks of productions. In order to start with a simple example, we consider a very basic parser function called *symbola*. This parser function "*parses*" a symbol of the input stream. To be more precise, it parses the symbol a of the input. By parsing we mean that it consumes the head of a list of characters, whenever the head is the character a. This function returns the symbol a itself, tupled with the unconsumed part of the input. If no parsing is possible, then, the empty list is returned. The straightforward implementation of this function looks as follows:

```
symbola []       = []
symbola (x : xs) = if x ≡ 'a' then [(x, xs)]
                   else []
```

It gets a list (of *Char*) as argument and it returns a list of pairs as result. The first element of the pair is the character a (of type *Char*) and the second is the unconsumed part of the input (a list of *Char*). Thus, the polymorphic type of this function is $symbola :: [Char] \rightarrow [(Char, [Char])]$, which corresponds to the following Parser} type:

```
symbola :: Parser Char Char
```

This type indicates that function *symbola* returns a parser as result. In its definition, however, the function does not "follow directly" this type since it has one argument, the list of characters, and produces a result, the list of pairs. It should be noticed that the function is type correct because Parser is a functional type. In order to have a more closely correspondance between the function and the type definition we re-define the function as follows:

```
symbola = λinp → case inp of
                   (x : xs) → if x ≡ 'a' then [(x, xs)]
                              else []
                   []       → []
```

in which the (list) argument is moved to the body of the definition. In this definition, *symbola* returns a parser function has result, following closely the above type definition. Recall that an expression of the form $\backslash x \rightarrow e$ is called λ-abstraction, and denotes the function that takes an argument x and returns the value of expression e. We will use this notation in the sequel.

We can use this parser function to parse the character a of a given input. Next, we present the results of executing *symbola* with two different input sentences:

Running Haskell

```
Parser> symbola "aaab"
[('a',"aab")]
Parser> symbola "baaa"
[]
```

As expected, when the function *symbola* is executed with `"aaab"` it produces a single solution since there is only one way of parsing the input with this function, obviuosly. The result is the character `'a'` itself (the parsed input) tupled with `"aab"` (the unconsumed part of the input). It should be noticed that the list of successes method immediatly pays off, since now we return the empty list if no parsing is possible. This is the case of the second excution of *symbola*: There is no way of parsing `"baaa"` with this parsing function.

Naturally, we wish to define a basic combinator to "*define*" any grammar symbol and not the character *a* only. We conveniently call this basic combinator **symbol**. This combinator parses any symbol of the input. We shall write this combinator as a function that takes a symbol and returns a parser. Its polymorphic type is defined as follows:

$$\textbf{symbol} :: Eq\ s \Rightarrow s \rightarrow \mathsf{Parser}\ s\ s$$

This function consumes a symbol s of the input whenever the head of the input is equal to s. Symbol s may be a character and in that case the input is a string or it may be a token and the input is a sequence of them. In other words, this function is not confined to a unique type of input. We need, however, to guarantee that the symbols to be parsed can be tested for equality. This is indicated by the predicate Eq in the type of the function[3]. The result of this combinator is the symbol itself, tupled with the unconsumed part of the input. If no parsing is possible, then, the empty list is returned. This basic combinator looks as follows:

$$\begin{array}{l} \textbf{symbol}\ s = \lambda inp \rightarrow \textbf{case}\ inp\ \textbf{of} \\ \qquad\qquad\qquad\qquad (x : xs) \rightarrow \textbf{if}\ s \equiv x\ \textbf{then}\ [(s, xs)] \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ [\,] \\ \qquad\qquad\qquad\qquad [\,] \qquad\quad \rightarrow [\,] \end{array}$$

For example, we can use this basic combinator to (try to) parse the character `'b'` in the sentences used previously:

[3] In Haskell this type reads as follows: *In the context ...*.

Running Haskell

```
Parser> symbol 'b' "aaab"
[]
Parser> symbol 'b' "baaa"
[('b',"aaa")]
```

The combinator **symbol** defines a parser function for a given symbol. When writing grammars, however, we also wish to define classes of symbols, *e.g.*, the class of digits, the class of letters, the class of capital letters, etc. Thus, we introduce new combinator that is not parameterized by a symbol but instead by a class membership predicate. In other words, the combinator gets as argument a function that is the predicate that classe members must satisfy. We call this basic combinator **satisfy** and we define it as follows:

```
satisfy :: (s → Bool) → Parser s s
satisfy p = λinp → case inp of
                     []       = []
                     (x : xs) = if p x then [(x, xs)]
                                else []
```

This function takes as argument a predicate, *i.e.*, a function from polymorphic type s (denoting the type of the input symbols) to *Bool* and returns a parser (of type Parser s s).

Using this basic combinator, we can define parser functions to parse, for example, (the classe of) digits, (the classe of) lower letters, a space. They are trivially defined as follows:

```
digit         = satisfy isDigit
space         = satisfy (≡ ' ')
lowerLetter   = satisfy (λx → 'a' ⩽ x ∧ x ⩽ 'z')
letterOrDigit :: Parser Char Char
letterOrDigit = satisfy (λx → isDigit x ∨ isLetter x)
```

where the functions *isDigit* and *isLetter* are standard predicates that tests whether or not a character is a digit or a letter. It is defined as follows: $\backslash x \rightarrow$ '0' $\leqslant x^{\&\&x}\leqslant$'9'. That is, it is defined similarly to the function passed as argument to **satisfy** in the **lowerLetter** parser function. Actually, that is the definition of the standard predicate *isLower*. We can use the above parsers to parse different inputs:

Running Haskell

```
Parser> digit "3+4"
[('3',"+4")]
Parser>  lowerLetter "abcd"
[('a',"bcd")]
Parser> lowerLetter "ABCD"
[]
```

The basic combinator functions presented thus far, define parsers that consume a single symbol from the input. In practice, however, we would also like to have parser functions that accept sequences of symbols and not just one. For example, we wish to have parser functions to easily define keywords of programming languages, *e.g.*, `if`, `let`, etc. Thus, we generalize the **symbol** combinator to parse a given sequence symbols.

token :: $Eq\ s \Rightarrow [s] \rightarrow \mathsf{Parser}\ s\ [s]$
token $t = \lambda inp \rightarrow$ **if** $t \equiv take\ n\ inp$ **then** $[(t, drop\ n\ inp)]$
else $[\,]$
where $n = length\ t$

Running Haskell

```
Parser> token "let" "let x=0 in ..."
[("let"," x=0 in ...")]
```

In the BNF formalism there are productions that do not derive any symbol, *e.g.*, the . In the combinator parsing paradigm this corresponds to a parser function that does not consume any symbol of the input and always succeeds. This combinator is usually called **succeed** and always returns a given, fixed value, tupled with the input itself (since no symbol is consumed). It is defined as follows:

succeed :: $r \rightarrow \mathsf{Parser}\ s\ r$
succeed $r = \lambda inp \rightarrow [(r, inp)]$

Now that we have the parser combinators for describing grammar symbols and the , we wish to combine symbols in the same way as in the BNF formalism. In BNF we have sequences of grammar symbols: the right-hand side of productions. Furthermore, we have alternative productions applied on the same grammar symbol, denoted with “”, *i.e.*, the BNF *or* symbol. Therefore, we introduce two combinator functions: the *sequential combinator* and the *alternative combinator*. We denote these combinators with $<\!*\!>$ and $<\!|\!>$, respectively, to make the notation as similar to BNF as possible. For the same reason, we define them as infix operators.

These two combinator parsers are the *real combinators*: they take two parsers as arguments, they *combine* them and they return the resulting parser. We start by defining the simplest one: the alternative *or* combinator: $<|>$. The alternative combinator takes two parsers, say p and q, and "tries" to parse the input using both alternatives. Thanks to the method of list of successes, both p and q yield lists whose elements are results of possible parsings. To combine both results we only need to concatenate the two lists.

$(<|>) :: \mathsf{Parser}\ s\ a \to \mathsf{Parser}\ s\ a \to \mathsf{Parser}\ s\ a$
$(p <|> q)\ xs = p\ xs \mathbin{+\!\!+} q\ xs$

As stated in the type of $<|>$ combinator, the parsers p and q must produce results of exactly the same type. Otherwise, the the resulting lists would have different types and could not be concatenated.

We can use this combinator parser to combine other parsers. For example, we may combine the basic parse *symbola* and **symbol** `'b'` in order to parse both the string `"aaab"` and `"baaa"`.

$exOr = symbola <|> \mathbf{symbol}\ \texttt{'b'}$

Executing the parser function *exOr* with those input sentences we get:

Running Haskell

```
Parser> exOr "aaab"
[('a',"aab")]
Parser> exOr "baaa"
[('b',"aaa")]
```

The sequential combinator $<*>$ takes also two parsers, say p and q, and parses anything that p and q would, if placed in succession. Since the first parser p, when applied to the input, may succeed with many results, each with the unconsumed part of the input as one result, the second parser q must be applied to each of these in turn. Therefore we use a list comprehension, in which q is applied in all possible ways to the rest of the input of p. The parsers p and q also yield values of the parsing process, besides the unconsumed input. These values must be combined. The trivial solution is to tuple both values [**?**,3, 5].

$(<*>) :: \mathsf{Parser}\ s\ a \to \mathsf{Parser}\ s\ b \to \mathsf{Parser}\ s\ (a, b)$
$(p <*> q) = \lambda inp\ \ [\ ((x, y), ys)$
$\qquad\qquad\qquad\quad |\ (x, xs) \leftarrow p\ inp$
$\qquad\qquad\qquad\quad ,\ (y, ys) \leftarrow q\ xs]$

$exSeq = \mathbf{symbol}\ \texttt{'a'} <*> \mathbf{symbol}\ \texttt{'b'} <*> \mathbf{symbol}\ \texttt{'c'}$

Running Haskell

```
Parser> exSeq "abcde"
[((('a','b'),'c'),"de")]
Parser> exSeq "bcde"
[]
```

As shown by this example, howver, this approach results in a messy manipulation of nested tuples produced as the results of parsing. For example, the type of *exSeq* is:

$exSeq :: \mathsf{Parser}\ Char\ ((Char, Char), Char)$

1.1 Giving Semantics to Combinator Parsers

We choose, instead, the approach taken in [**?**]: the first parser p yields a function f of type $(a \to r)$, the second parser q yields a value v of type a, and the sequential parser yields a value of type r that is obtained by applying the function f to the value v. This $<\!*\!>$ combinator is presented next.

$$
\begin{array}{l}
(\ <\!*\!>\ ) :: \mathsf{Parser}\ s\ (a \to r) \to \mathsf{Parser}\ s\ a \to \mathsf{Parser}\ s\ r \\
(p\ <\!*\!>\ q) = \lambda inp \to\ [\ (f\ v, ys) \\
\qquad\qquad\qquad\qquad\quad \mid (f\quad, xs) \leftarrow p\ inp \\
\qquad\qquad\qquad\qquad\quad ,\ (\quad v, ys) \leftarrow q\ xs]
\end{array}
$$

The $<\!*\!>$ combinator is an excellent example which nicely exploits both the list comprehension notation and the ability of functional languages to manipulate functions.

Traditionally, a parser executes semantic actions during the parsing process. Thus, we introduce a third parser combinator: the *application combinator* for describing further processing of the values returned by the parser. It is defined as follows:

$$
\begin{array}{l}
(\ <\$>\ ) :: (a \to r) \to \mathsf{Parser}\ s\ a \to \mathsf{Parser}\ s\ r \\
(f\ <\$>\ q) = \lambda inp \to [(f\ v, ys) \mid (v, xs) \leftarrow p\ inp]
\end{array}
$$

It applies function f, the so-called *semantic function*, to the result of parser p. Having this combinator we are ready to assign a semantic meaning to the parsed symbols. For example, we can re-define the parser function **digit** to not only parse a digit from the input, but also to return its value as result:

$$
\begin{array}{l}
\mathbf{digitVal} :: \mathsf{Parser}\ Char\ Int \\
\mathbf{digitVal} = digitToInt\ <\$>\ \mathbf{digit}
\end{array}
$$

where the function *digitToInt* is the Haskell standard conversion operation that converts a character into its integer value. As expected its type is *digitToInt* ::

$Char \to Int$. Observe that, the parser function **digit** returns the parsed character, say a, tupled with the unconsumed part of the input as result. The combinator $<\$>$ applies the semantic function *digitToInt* to the parsed character a (converting it into its integer value), and tuples it with the non-yet parsed part of the input.

For example, we have:

Running Haskell

```
Parser> digitVal "3+4"
[(3,"+4")]
```

The result of parsing is an integer value, instead of a character as obtained when executed the parser function **digit** with the same input.

The parser functions constructed thus far contain a single (terminal) symbol on the right-hand side of the productions. We shall consider now a production with several grammar symbols. We start by considering a (simple) production whose right-hand side is `a b`. The straightforward aproach to model such production with combinators is to combine the parsers **symbol** `'a'` and **symbol** `'b'` with the sequencing combinator $<\ast>$ as follows:

exSeq = **symbol** `'a'` $<\ast>$ **symbol** `'b'`

Obviously, this Haskell expression does not type check: according to the type of the combinator $<\ast>$, the first parser has to return a function as result and not a value (of type *Char* in the case of **symbol** `'a'`). Thus, to sequentially combine the two parsers we can use the application combinator $<\$>$ to apply a semantic function to the result of the first parser. Such semantic function has to return as result the function that is applied to the result of the second one. This is written as follows:

```
exSeq = f <$> symbol 'a' <*> symbol 'b'
  where f :: Char → (Char → [Char])
        f a b = [a, b]
```

we may rewrite this function to make explicit the associativy and precendence of the combinators:

```
exSeq = (f <$> symbol 'a') <*> (symbol 'b')
  where f :: Char → (Char → [Char])
        f a = λb → [a, b]
```

As we can see the semantic function f has one argument, which corresponds to the result produced by the first parser, and returns a function as result. This function is the function returned by the first parser combined with $<\ast>$, indeed. It has a single argument, corresponding to the result of the second parser **symbol** `'b'`, and, in this case, returns as result a list with two elements: the results of both parsers. This is to say that the semantic function f combines the results of the two parsers into a list. Thus the type of the parser *exSeq* is Parser *Char* [*Char*].

Let us consider productions/parsers that have to combine in sequence several symbols/parsers.

$exSeq :: \mathsf{Parser}\ Char\ [\,Char\,]$
$exSeq = sf\ \texttt{<\$>}\ \textbf{symbol}\ \texttt{'a'}\ \texttt{<*>}\ \textbf{symbol}\ \texttt{'b'}\ \texttt{<*>}\ \textbf{symbol}\ \texttt{'c'}$
$\quad \textbf{where}\ sf\ a\ b\ c = [\,a, b, c\,]$

In other words, the number of arguments of the semantic function f is the number of parser functions sequencially combined.

$exSeq :: \mathsf{Parser}\ Char\ [\,Char\,]$
$exSeq = (\lambda a \to \lambda b \to \lambda c \to [\,a, b, c\,])\ \texttt{<\$>}\ \textbf{symbol}\ \texttt{'a'}\ \texttt{<*>}\ \textbf{symbol}\ \texttt{'b'}\ \texttt{<*>}\ \textbf{symbol}\ \texttt{'c'}$

Running Haskell

```
Parser> exSeq' "abcd"
[("abc","d")]
```

Consider now that we define the following grammar/parser function where we have two alternatives that start with the same sequence of symbols "ab". A regular determinotic parser needs to look into the first three symbols of the input to be able to decide which of the alternatives to choose. In terms of grammars, this is not a LL(1) grammar, but corresponds to a LL(3) one.

$exSeq \ = \ sf1\ \texttt{<\$>}\ \textbf{symbol}\ \texttt{'a'}\ \texttt{<*>}\ \textbf{symbol}\ \texttt{'b'}\ \texttt{<*>}\ \textbf{symbol}\ \texttt{'c'}$
$\qquad \texttt{<|>}\ sf2\ \texttt{<\$>}\ \textbf{symbol}\ \texttt{'a'}\ \texttt{<*>}\ \textbf{symbol}\ \texttt{'b'}$
$\quad \textbf{where}\ sf1\ a\ b\ c = [\,a, b, c\,]$
$\qquad\qquad sf2\ a\ b \quad = [\,a, b\,]$

Since our combinators (lazily) try all possible alternatives, applying *exSeq* to the input `"abcd"` gives the results `[("abc", "d"), ("ab", "cd")]`. The first results is the expected result since the sequence of letters `"abc"` has been consumed, and the unconsumed part is `"d"`. This behaviour arises because the alternative combinator <|> is *non-deterministic*: both alternatives are explored, even when the first alternative parses the input successfully. Thus, the second result produced by *exSeq* corresponds to the parse result using the second production. In this case, only the sequence `"ab"` is consumed.

Running Haskell

```
Parser> exSeq'' "abcd"
[("abc","d"),("ab","cd")]
```

We proceed now to define more interesting and usefull parser functions. In chapter**??**, we have presented a non-terminal *Spaces* and respective productions to define a possibly empty sequence of spaces.

Usually spaces do not play a syntactic role a they are usually ignored [4]. Let us define a parser function to eliminate spaces as well.

```
spaces :: Parser Char ()
spaces = sf  <$> symbol ' ' <*> s
         <|> succeed ()
           where sf _ _ = ()
```

Now, we write a new **token′** basic combinador so that it "*eats*" spaces after parsing the given token:

token′ $t = (\lambda r\ s \rightarrow r)$ <$> **token** t <*> **spaces** <*>

Running Haskell

```
Parser> token' "let" "        let x=0 in ..."
[("let"," x=0 in ...")]
```

Exercises

*Write the basic combinator **symbol** in terms of **satisfy**. Write a parser function **token** that accepts all the combinations of letters (lowercase and capital letters of its input token). Re-Write the exSeq in order to produce the nested tuple result. Re-write the combinator **spaces** in order to eliminate all kind of spaces, ie, tabs and newlines. Write a parser function for the language*

$$\mathcal{L} = \{a^n b^n \mid n \in N\}$$

The parser simply copies the input to the output. Apply the rules to transform the grammar defining the list of spaces into Haskell data types and re-write the parser to use the respective data constructors as semantic functions.

1.2 Combinators for Extended BNF

Traditionally, parser generators use a fix set of combinators, which correspond to the BNF notation. Within the parser combinator paradigm we are not forced to use such a pre-defined set of combinators. On the contrary, we may define new combinators and in this way extend our notation.

[4] Haskell is one exception

$$\begin{aligned}
&\textbf{oneOrMore}\ p = \quad f \mathbin{<\$>} p \mathbin{<*>} (\textbf{oneOrMore}\ p)\\
&\qquad\qquad\qquad \mathbin{<|>} g \mathbin{<\$>} p\\
&\qquad\qquad\qquad \textbf{where}\ f\ x\ xs = x : xs\\
&\qquad\qquad\qquad\qquad g\ x = [x]
\end{aligned}$$

$$\begin{aligned}
&\textbf{zeroOrMore}\ p = \quad f \mathbin{<\$>} p \mathbin{<*>} (\textbf{zeroOrMore}\ p)\\
&\qquad\qquad\qquad \mathbin{<|>} \ \textbf{succeed}\ [\,]\\
&\qquad\qquad\qquad \textbf{where}\ f\ x\ xs = x : xs
\end{aligned}$$

Since $aa* \equiv a^+$, we can also express **oneOrMore** in terms of **zeroOrMore** as follows:

$$\textbf{oneOrMore}\ p = (:) \mathbin{<\$>} p \mathbin{<*>} (\textbf{zeroOrMore}\ p)$$

and, we can re-define the parser of integer numbers:

$$\begin{aligned}
&\textbf{spaces} = \textbf{zeroOrMore space}\\
&\mathit{natural} = \textbf{oneOrMore digit}
\end{aligned}$$

The optional combinator: we have to give as argument the result to be returned for the ϵ-production.

$$\textbf{optional}\ p = (:) \mathbin{<\$>} p \mathbin{<*>} (\textbf{zeroOrMore}\ p)$$

$$\begin{aligned}
&\textbf{enclosedBy}\ s\ p\ e = f \mathbin{<\$>} s \mathbin{<*>} p \mathbin{<*>} e\\
&\quad \textbf{where}\ f\ a\ b\ c = b
\end{aligned}$$

And, we reuse this combinator to describe usefull language structures, like for example:

$$\begin{aligned}
&\mathit{parenthesized}\ p = \textbf{enclosedBy}\ (\textbf{symbol}\ (\texttt{'('})\ p\ (\textbf{symbol}\ (\texttt{')'})\\
&\mathit{bracketed}\qquad p = \textbf{enclosedBy}\ (\textbf{symbol}\ (\texttt{'['})\ p\ (\textbf{symbol}\ (\texttt{']'})
\end{aligned}$$

A typical syntactic charateristic of programming languages is existance of lists of elements separated by a punctuation symbol, *e.g.*, the list of formal arguments of a functions. The punctuation symbols have a sintactic meaning only: to help the programmer writer to structure the programs and to make the syntactic analysis of the texts easier. Thus, we express the sintactic charateristic by the combinator **separatedBy** as folows:

$$\begin{aligned}
&\textbf{separatedBy}\ p\ s = \quad f \mathbin{<\$>} p \mathbin{<*>} s \mathbin{<*>} (\textbf{separatedBy}\ p\ s)\\
&\qquad\qquad\qquad \mathbin{<|>} g \mathbin{<\$>} p\\
&\qquad\qquad\qquad \textbf{where}\ f\ a\ b = a : b\\
&\qquad\qquad\qquad\quad g\ a = [a]
\end{aligned}$$

we can now define the concrete language for lists (of integer numbers) in Haskell as follows:

$$\textbf{haskellList} = \mathit{bracketed}\ (\textbf{separatedBy int}\ (\textbf{symbol}\ \texttt{','}))$$

Exercises

Write the following combinators:

1. ***followedBy**: a list of elements, each of which followed by a punctuation symbol.*
2. ***block**: a block os instructions delimited by some punctuation symbols and each instruction followed by a punctuation symbol.*
3. ***charRange*** :: Parser *Char String that given two characters separated by* `-`*, returns the list of chars with all characters in that interval. For example,* ***charRange*** `"a-g"` *returns* `[("abcdefg", "")]`

1.3 A Parser for Regular Expressions

The abstract syntax of regular expressions can be easily by the following Haskell data type:

```
data RegExp = Epsilon                  -- epsilon
            | Literal Char             -- a
            | Or      RegExp RegExp    -- p + q
            | Then    RegExp RegExp    -- p q
            | Star    RegExp           -- p*
```

A grammar for defining the sintax of regular expressions has to define the notation used to write its operators. It has also to take into account the associativity and precedence of operators. There are two binary operators in regular expressions - *or* and *then*, where the *then* as a higher precedence than *or*. The operator *or* is usually denoted by symbol |, while usually there is no symbol to represent *then*: symbols are just written side by side. For example, the regular expressions $ab \mid c$, reads as *ab or c*. In other words, it is equivalente to $(ab) \mid c$. Regular expressions have an optional operator (denoted by ? expressing the algebraic rule $a? \equiv a \mid \epsilon$) and repetition operators: one or mode + and zero or mode $*$ repetitions, respectively. These operators have the highest precedence. This means that, $a?b* \mid c$ is equivalent to $((a?)(b*)) \mid c$.

To express such operator precedence, we create two nonterminals/functions *re*, *termThen* and *term* for the three levels of precedence, and an extra nonterminal *factor* for defining basic units in expressions. Thus, nonterminal *factor* defines basic units in expressions: a letter or digit, any symbol surrounded by quotes, and parenthesized expressions.

```
re :: Parser Char RegExp
re    =    f  <$> termThen <*> symbol '|' <*> re
       <|> id <$> termThen
       <|> succeed Epsilon
  where f l _ r = Or l r

termThen :: Parser Char RegExp
termThen  =  f  <$> term <*> termThen
```

```
            <|> id <$> term
   where f l r = Then l r

term :: Parser Char RegExp
term  =  f    <$> factor <*> symbol '?'   -- optional
     <|> g    <$> factor <*> symbol '*'   -- zero or more
     <|> h    <$> factor <*> symbol '+'   -- one or more
     <|> id   <$> factor
   where
     f e _ = Or e Epsilon                 -- e? = e or epsilon
     g e _ = Star e
     h e _ = Then e (Star e)              -- e+ = e e*

factor :: Parser Char RegExp
factor  =  f <$> letterOrDigit
       <|> g <$> symbol '\'' <*> satisfy (λx → True) <*> symbol '\''
       <|> h <$> symbol '(' <*> re <*> symbol ')'
   where
     f a      = Literal a
     g _ e _  = Literal e
     h _ e _  = e
```

The parser combinator *re* parses a regular expression and it produces the desired abstract syntax tree. Next we run our parser with the regular expression $a?b* \mid c$. We show the results that contains the abstract syntax tree that is built when the parser consumes all symbols of the input string.

Running Haskell

```
Parser> re "a?b*|c"
[(Or (Then (Or (Literal 'a') Epsilon) (Star (Literal 'b'))) (Literal 'c'),""),...]
```

Exercises

Express re in terms of ***separatedBy*** *and termThen in terms of* ***oneOrMore*** *Rewrite the parser* ***charRange*** *so that it returns a regular expression defining the list of characters in the range. This new parser has type charRangeAsRE ::* Parser *Char* RegExp *Extend the regular expression Language/notation so that it considers set of symbols. For example, the regular expression ... Reuse the parser of regular expressions, the functions that transform a regular expression in a deterministic finite automata, and the respective acceptance function in order to generate the Haskell code the implements an efficient acceptnace function for the language defined by a given (textual) regular expressions. Write a parser for the SL language.*

1.4 Parser Combinators and Left Recursion

To do: *Left Recursion*

explain that left recursion is not supported, as this is the case of standard recursive descent parsing techniques.

References

1. S. Marlow, Happy User Guide, Glasgow University (December 1997).
2. A. V. Aho, R. Sethi, J. D. Ullman, Compilers: Principles, Techniques and Tools, Addison Wesley, 1986.
3. J. Fokker, Functional Parsers, in: J. Jeuring, E. Meijer (Eds.), First International School on Advanced Functional Programming, Vol. 925 of LNCS, 1995, pp. 1–23.
4. G. Hutton, E. Meijer, Functional Pearl: Monadic Parsing in Haskell, Journal of Functional Programming 8 (4) (1998) 437–444.
5. S. Thompson, Haskell: The Craft of Functional Programming, Addison-Wesley, 1999.