# Handout 5 (Grammars & Parser)

While regular expressions are very useful for lexing and for recognising many patterns in strings (like email addresses), they have their limitations. For example there is no regular expression that can recognise the language $a^n b^n$. Another example for which there exists no regular expression is the language of well-parenthesised expressions. In languages like Lisp, which use parentheses rather extensively, it might be of interest whether the following two expressions are well-parenthesised (the left one is, the right one is not):

$$(((()()))()) \qquad (((()()))()))$$

Not being able to solve such recognition problems is a serious limitation. In order to solve such recognition problems, we need more powerful techniques than regular expressions. We will in particular look at *context-free languages*. They include the regular languages as the picture below shows:
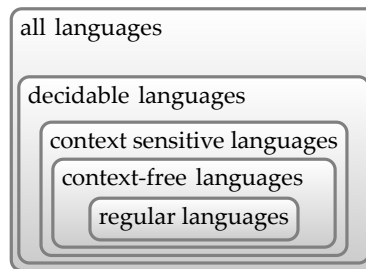
Context-free languages play an important role in 'day-to-day' text processing and in programming languages. Context-free languages are usually specified by grammars. For example a grammar for well-parenthesised expressions is

$$P \;\rightarrow\; (\cdot P \cdot) \cdot P \mid \epsilon$$

or a grammar for recognising strings consisting of ones is

$$O \;\rightarrow\; 1 \cdot O \mid 1$$

In general grammars consist of finitely many rules built up from *terminal symbols* (usually lower-case letters) and *non-terminal symbols* (upper-case letters). Rules have the shape

$$NT \;\rightarrow\; rhs$$

where on the left-hand side is a single non-terminal and on the right a string consisting of both terminals and non-terminals including the ϵ-symbol for indicating the empty string. We use the convention to separate components on the right hand-side by using the $\cdot$ symbol, as in the grammar for well-parenthesised expressions. We also use the convention to use $\mid$ as a shorthand notation for several rules. For example

$$NT \quad \rightarrow \quad rhs_1 \mid rhs_2$$

means that the non-terminal *NT* can be replaced by either rhs_1 or rhs_2. If there are more than one non-terminal on the left-hand side of the rules, then we need to indicate what is the *starting* symbol of the grammar. For example the grammar for arithmetic expressions can be given as follows

$$
\begin{array}{rcll}
E & \rightarrow & N & (1)\\
E & \rightarrow & E \cdot + \cdot E & (2)\\
E & \rightarrow & E \cdot - \cdot E & (3)\\
E & \rightarrow & E \cdot * \cdot E & (4)\\
E & \rightarrow & (\cdot E \cdot) & (5)\\
N & \rightarrow & N \cdot N \mid 0 \mid 1 \mid \ldots \mid 9 & (6\ldots)
\end{array}
$$

where *E* is the starting symbol. A *derivation* for a grammar starts with the starting symbol of the grammar and in each step replaces one non-terminal by a right-hand side of a rule. A derivation ends with a string in which only terminal symbols are left. For example a derivation for the string $(1+2)+3$ is as follows:

$$
\begin{array}{rll}
E & \rightarrow & E + E & \text{by (2)}\\
& \rightarrow & (E) + E & \text{by (5)}\\
& \rightarrow & (E+E) + E & \text{by (2)}\\
& \rightarrow & (E+E) + N & \text{by (1)}\\
& \rightarrow & (E+E) + 3 & \text{by (6\ldots)}\\
& \rightarrow & (N+E) + 3 & \text{by (1)}\\
& \rightarrow^+ & (1+2) + 3 & \text{by (1, 6\ldots)}
\end{array}
$$

where on the right it is indicated which grammar rule has been applied. In the last step we merged several steps into one.

The *language* of a context-free grammar *G* with start symbol *S* is defined as the set of strings derivable by a derivation, that is

$$\{c_1 \ldots c_n \mid S \rightarrow^* c_1 \ldots c_n \ \text{with all } c_i \text{ being non-terminals}\}$$

A *parse-tree* encodes how a string is derived with the starting symbol on top and each non-terminal containing a subtree for how it is replaced in a derivation. The parse tree for the string $(1+23)+4$ is as follows:
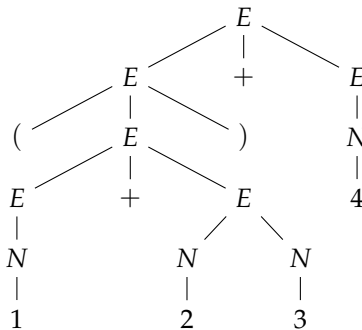
We are often interested in these parse-trees since they encode the structure of how a string is derived by a grammar. Before we come to the problem of constructing such parse-trees, we need to consider the following two properties of grammars. A grammar is *left-recursive* if there is a derivation starting from a non-terminal, say NT which leads to a string which again starts with NT. This means a derivation of the form.

$$NT \to \ldots \to NT \cdot \ldots$$

It can be easily seen that the grammar above for arithmetic expressions is left-recursive: for example the rules $E \to E \cdot + \cdot E$ and $N \to N \cdot N$ show that this grammar is left-recursive. But note that left-recursiveness can involve more than one step in the derivation. The problem with left-recursive grammars is that some algorithms cannot cope with them: they fall into a loop. Fortunately every left-recursive grammar can be transformed into one that is not left-recursive, although this transformation might make the grammar less "human-readable". For example if we want to give a non-left-recursive grammar for numbers we might specify

$$N \;\to\; 0 \mid \ldots \mid 9 \mid 1 \cdot N \mid 2 \cdot N \mid \ldots \mid 9 \cdot N$$

Using this grammar we can still derive every number string, but we will never be able to derive a string of the form $N \to \ldots \to N \cdot \ldots$.

The other property we have to watch out for is when a grammar is *ambiguous*. A grammar is said to be ambiguous if there are two parse-trees for one string. Again the grammar for arithmetic expressions shown above is ambiguous. While the shown parse tree for the string $(1 + 23) + 4$ is unique, this is not the case in general. For example there are two parse trees for the string $1 + 2 + 3$, namely
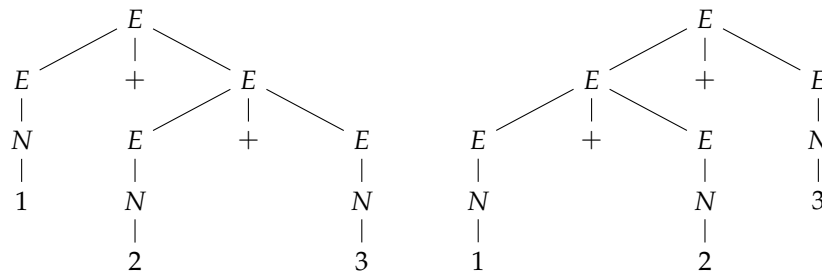
In particular in programming languages we will try to avoid ambiguous grammars because two different parse-trees for a string mean a program can be interpreted in two different ways. In such cases we have to somehow make sure the two different ways do not matter, or disambiguate the grammar in some other way (for example making the $+$ left-associative). Unfortunately already the problem of deciding whether a grammar is ambiguous or not is in general undecidable. But in simple instance (the ones we deal in this module) one can usually see when a grammar is ambiguous.

Let us now turn to the problem of generating a parse-tree for a grammar and string. In what follows we explain *parser combinators*, because they are easy to implement and closely resemble grammar rules. Imagine that a grammar describes the strings of natural numbers, such as the grammar N shown above. For all such strings we want to generate the parse-trees or later on we actually want to extract the meaning of these strings, that is the concrete integers "behind" these strings. In Scala the parser combinators will be functions of type

$$\texttt{I} \Rightarrow \texttt{Set[(T, I)]}$$

that is they take as input something of type `I`, typically a list of tokens or a string, and return a set of pairs. The first component of these pairs corresponds to what the parser combinator was able to process from the input and the second is the unprocessed part of the input. As we shall see shortly, a parser combinator might return more than one such pair, with the idea that there are potentially several ways how to interpret the input. As a concrete example, consider the case where the input is of type string, say the string

$$\boxed{\texttt{i}\,|\,\texttt{f}\,|\,\texttt{f}\,|\,\texttt{o}\,|\,\texttt{o}\,|\,\texttt{⎵}\,|\,\texttt{t}\,|\,\texttt{e}\,|\,\texttt{s}\,|\,\texttt{t}\,|\,\texttt{b}\,|\,\texttt{a}\,|\,\texttt{r}}$$

We might have a parser combinator which tries to interpret this string as a keyword (`if`) or an identifier (`iffoo`). Then the output will be the set

$$\left\{\left(\boxed{\texttt{i}\,\texttt{f}}\,,\,\boxed{\texttt{f}\,\texttt{o}\,\texttt{o}\,\texttt{⎵}\,\texttt{t}\,\texttt{e}\,\texttt{s}\,\texttt{t}\,\texttt{b}\,\texttt{a}\,\texttt{r}}\right),\left(\boxed{\texttt{i}\,\texttt{f}\,\texttt{f}\,\texttt{o}\,\texttt{o}}\,,\,\boxed{\texttt{⎵}\,\texttt{t}\,\texttt{e}\,\texttt{s}\,\texttt{t}\,\texttt{b}\,\texttt{a}\,\texttt{r}}\right)\right\}$$

where the first pair means the parser could recognise `if` from the input and leaves the rest as 'unprocessed' as the second component of the pair; in the other case it could recognise `iffoo` and leaves `⎵testbar` as unprocessed. If the parser cannot recognise anything from the input then parser combinators just return the empty set {}. This will indicate something "went wrong".

The main attraction is that we can easily build parser combinators out of smaller components following very closely the structure of a grammar. In order to implement this in an object oriented programming language, like Scala, we need to specify an abstract class for parser combinators. This abstract class requires the implementation of the function `parse` taking an argument of type `I` and returns a set of type `Set[(T, I)]`.

```scala
abstract class Parser[I, T] {
  def parse(ts: I): Set[(T, I)]

  def parse_all(ts: I): Set[T] =
    for ((head, tail) <- parse(ts); if (tail.isEmpty))
      yield head
}
```

From the function `parse` we can then "centrally" derive the function `parse_all`, which just filters out all pairs whose second component is not empty (that is

has still some unprocessed part). The reason is that at the end of parsing we are only interested in the results where all the input has been consumed and no unprocessed part is left.

One of the simplest parser combinators recognises just a character, say *c*, from the beginning of strings. Its behaviour is as follows:

- if the head of the input string starts with a *c*, it returns the set $\{(c, \textit{tail of } s)\}$

- otherwise it returns the empty set $\varnothing$

The input type of this simple parser combinator for characters is `String` and the output type `Set[(Char, String)]`. The code in Scala is as follows:

```scala
case class CharParser(c: Char) extends Parser[String, Char] {
  def parse(sb: String) =
    if (sb.head == c) Set((c, sb.tail)) else Set()
}
```

The `parse` function tests whether the first character of the input string `sb` is equal to `c`. If yes, then it splits the string into the recognised part `c` and the unprocessed part `sb.tail`. In case `sb` does not start with `c` then the parser returns the empty set (in Scala `Set()`).

More interesting are the parser combinators that build larger parsers out of smaller component parsers. For example the alternative parser combinator is as follows.

```scala
class AltParser[I, T]
        (p: => Parser[I, T],
         q: => Parser[I, T]) extends Parser[I, T] {
  def parse(sb: I) = p.parse(sb) ++ q.parse(sb)
}
```

The types of this parser combinator are polymorphic (we just have `I` for the input type, and `T` for the output type). The alternative parser builds a new parser out of two existing parser combinator `p` and `q`. Both need to be able to process input of type `I` and return the same output type `Set[(T, I)]`. (There is an interesting detail of Scala, namely the `=>` in front of the types of `p` and `q`. They will prevent the evaluation of the arguments before they are used. This is often called *lazy evaluation* of the arguments.) The alternative parser should run the input with the first parser `p` (producing a set of outputs) and then run the same input with `q`. The result should be then just the union of both sets, which is the operation `++` in Scala.

This parser combinator already allows us to construct a parser that either a character `a` or `b`, as

```scala
new AltParser(CharParser('a'), CharParser('b'))
```

Scala allows us to introduce some more readable shorthand notation for this, like `'a' || 'b'`. We can call this parser combinator with the strings

input string   output

$\boxed{a}\boxed{c} \quad \rightarrow \quad \{(\boxed{a},\boxed{c})\}$

$\boxed{b}\boxed{c} \quad \rightarrow \quad \{(\boxed{b},\boxed{c})\}$

$\boxed{c}\boxed{c} \quad \rightarrow \quad \varnothing$

We receive in the first two cases a successful output (that is a non-empty set).

A bit more interesting is the *sequence parser combinator* implemented in Scala as follows:

```scala
class SeqParser[I, T, S]
      (p: => Parser[I, T],
       q: => Parser[I, S]) extends Parser[I, (T, S)] {
  def parse(sb: I) =
    for ((head1, tail1) <- p.parse(sb);
         (head2, tail2) <- q.parse(tail1))
           yield ((head1, head2), tail2)
}
```

This parser takes as input two parsers, `p` and `q`. It implements `parse` as follows: let first run the parser `p` on the input producing a set of pairs (`head1`, `tail1`). The `tail1` stands for the unprocessed parts left over by `p`. Let `q` run on these unprocessed parts producing again a set of pairs. The output of the sequence parser combinator is then a set containing pairs where the first components are again pairs, namely what the first parser could parse together with what the second parser could parse; the second component is the unprocessed part left over after running the second parser `q`. Therefore the input type of the sequence parser combinator is as usual `I`, but the output type is

```
Set[((T, S), I)]
```

Scala allows us to provide some shorthand notation for the sequence parser combinator. So we can write for example `'a' ~ 'b'`, which is the parser combinator that first consumes the character `a` from a string and then `b`. Calling this parser combinator with the strings

input string   output

$\boxed{a}\boxed{b}\boxed{c} \quad \rightarrow \quad \{((\boxed{a},\boxed{b}),\boxed{c})\}$

$\boxed{b}\boxed{a}\boxed{c} \quad \rightarrow \quad \varnothing$

$\boxed{c}\boxed{c}\boxed{c} \quad \rightarrow \quad \varnothing$

A slightly more complicated parser is (`'a' || 'b'`) ~ `'b'` which parses as first character either an `a` or `b` followed by a `b`. This parser produces the following results.

input string		output
`a b c`	$\rightarrow$	$\{((\boxed{a},\boxed{b}),\boxed{c})\}$
`b b c`	$\rightarrow$	$\{((\boxed{b},\boxed{b}),\boxed{c})\}$
`a a c`	$\rightarrow$	$\varnothing$

Note carefully that constructing the parser `'a' || ('a'` $\sim$ `'b')` will result in a tying error. The first parser has as output type a single character (recall the type of `CharParser`), but the second parser produces a pair of characters as output. The alternative parser is however required to have both component parsers to have the same type. We will see later how we can build this parser without the typing error.

The next parser combinator does not actually combine smaller parsers, but applies a function to the result of the parser. It is implemented in Scala as follows

```scala
class FunParser[I, T, S]
        (p: => Parser[I, T],
         f: T => S) extends Parser[I, S] {
  def parse(sb: I) =
    for ((head, tail) <- p.parse(sb)) yield (f(head), tail)
}
```

This parser combinator takes a parser `p` with output type `T` as input as well as a function `f` with type `T => S`. The parser `p` produces sets of type `(T, I)`. The `FunParser` combinator then applies the function `f` to all the parer outputs. Since this function is of type `T => S`, we obtain a parser with output type `S`. Again Scala lets us introduce some shorthand notation for this parser combinator. Therefore we will write `p ==> f` for it.