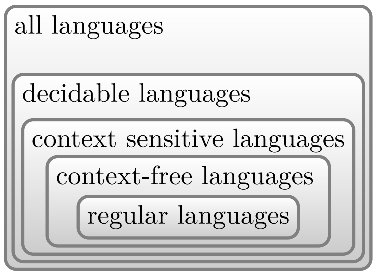
Handout 6

While regular expressions are very useful for lexing and for recognising many patterns (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 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):

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

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

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

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

$$NT \;\;\rightarrow\;\; rhs_1 \mid rhs_2$$

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

$$\begin{array}{lcl}
E & \rightarrow & N \\
E & \rightarrow & E \cdot + \cdot E \\
E & \rightarrow & E \cdot - \cdot E \\
E & \rightarrow & E \cdot * \cdot E \\
E & \rightarrow & (\cdot E \cdot) \\
N & \rightarrow & N \cdot N \mid 0 \mid 1 \mid \ldots \mid 9
\end{array}$$

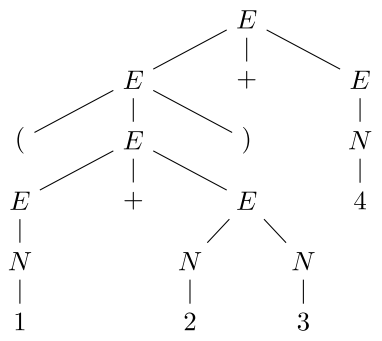
where E is the starting symbol. A *derivation* for a grammar starts with the staring symbol of the grammar and in each step replaces one non-terminal by a right-hand side of a rule. A derivation ends with a string in which only terminal symbols are left. For example a derivation for the string $(1+2)+3$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

$$\begin{array}{lcl}
E & \rightarrow & E + E \\
 & \rightarrow & (E) + E \\
 & \rightarrow & (E + E) + E \\
 & \rightarrow & (E + E) + N \\
 & \rightarrow & (E + E) + 3 \\
 & \rightarrow & (N + E) + 3 \\
 & \rightarrow^{+} & (1 + 2) + 3
\end{array}$$

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.

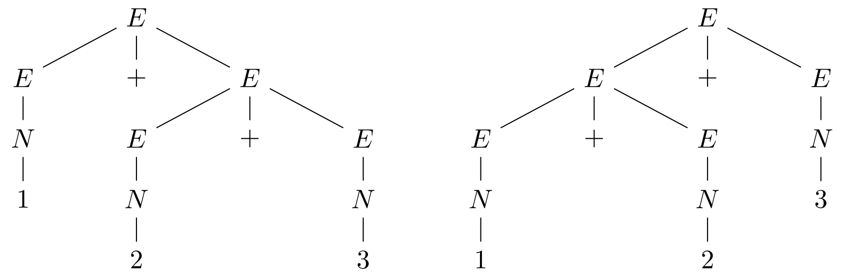
$$NT \rightarrow \ldots \rightarrow NT \cdot \ldots$$

It can be easily seems that the grammar above for arithmetic expressions is left-recursive: for example the rules $E \rightarrow E \cdot + \cdot E$ and $N \rightarrow N \cdot N$ show that this grammar is left-recursive. Some algorithms cannot cope with left-recursive grammars. 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

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

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

The other property we have to watch out 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, 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 way (for example making the + left-associative). Unfortunately already the problem of deciding whether a grammar is ambiguous or not is in general undecidable.

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. The parser combinators will be functions of type

```
I ⇒ 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.

The abstract class for parser combinators requires the implementation of the function `parse` taking an argument of type `I` and returns a set of type `Set[(T, I)]`.

```
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
}
```

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:

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