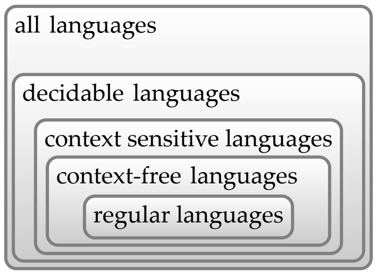
Handout 5 (Grammars & Parser)

So far we have focused on regular expressions as well as matching and lexing algorithms. 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 $a^n b^n$ (where you have strings starting with n a's followed by the same amount of b's). 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 to know whether the following two expressions are well-parenthesised or not (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 about language classes shows:



Each "bubble" stands for sets of languages (remember languages are sets of strings). As indicated the set of regular languages is fully included inside the context-free languages, meaning every regular language is also context-free, but not vice versa. Below I will let you think, for example, what the context-free grammar is for the language corresponding to the regular expression $(aaa)^*a$.

Because of their convenience, context-free languages play an important role in 'day-to-day' text processing and in programming languages. Context-free in this setting means that "words" have one meaning only and this meaning is independent from the context the "words" appear in. For example ambiguity issues like

```
Time flies like an arrow. Fruit flies like bananas.
```

from natural languages were the meaning of *flies* depends on the surrounding *context* are avoided as much as possible. Here is an interesting video about C++ being not a context-free language

https://www.youtube.com/watch?v=OzK8pUu4UfM

Context-free languages are usually specified by grammars. For example a grammar for well-parenthesised expressions can be given as follows:

$$\boldsymbol{P} ::= (\cdot \boldsymbol{P} \cdot) \cdot \boldsymbol{P} \mid \epsilon$$

or a grammar for recognising strings consisting of ones is

$$\boldsymbol{O} ::= 1 \cdot \boldsymbol{O} \mid 1$$

In general grammars consist of finitely many rules built up from *terminal symbols* (usually lower-case letters) and *non-terminal symbols* (upper-case letters written in bold like $\boldsymbol{A}$, $\boldsymbol{N}$ and so on). Rules have the shape

$$\boldsymbol{NT} ::= rhs$$

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

$$\boldsymbol{NT} ::= rhs_1 \mid rhs_2$$

means that the non-terminal $\boldsymbol{NT}$ 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

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

where $\boldsymbol{E}$ 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:

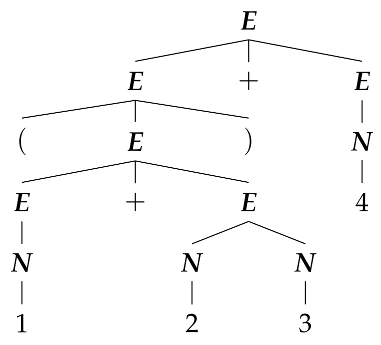
$$\begin{array}{lcll}
\boldsymbol{E} & \rightarrow & \boldsymbol{E}+\boldsymbol{E} & \text{by (2)} \\
 & \rightarrow & (\boldsymbol{E})+\boldsymbol{E} & \text{by (5)} \\
 & \rightarrow & (\boldsymbol{E}+\boldsymbol{E})+\boldsymbol{E} & \text{by (2)} \\
 & \rightarrow & (\boldsymbol{E}+\boldsymbol{E})+\boldsymbol{N}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

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 $\boldsymbol{NT}$ which leads to a string which again starts with $\boldsymbol{NT}$. This means a derivation of the form.

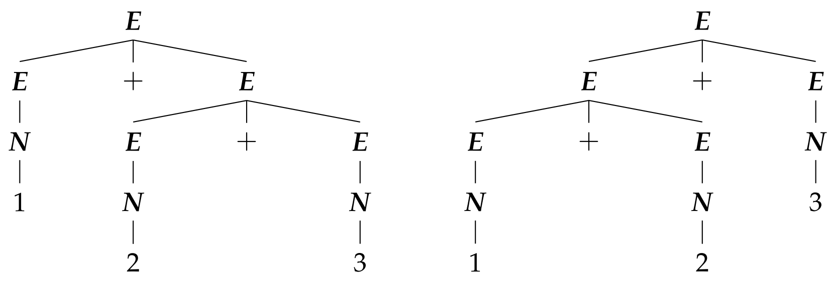
$$\boldsymbol{NT} \rightarrow \ldots \rightarrow \boldsymbol{NT} \cdot \ldots$$

It can be easily seen that the grammar above for arithmetic expressions is left-recursive: for example the rules $\boldsymbol{E} \rightarrow \boldsymbol{E} \cdot + \cdot \boldsymbol{E}$ and $\boldsymbol{N} \rightarrow \boldsymbol{N} \cdot \boldsymbol{N}$ 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: with left-recursive grammars they will 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

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

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

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 with in this module) one can usually see when a grammar is ambiguous.

Removing Left-Recursion

Let us come back to the problem of left-recursion and consider the following grammar for binary numbers:

$$\boldsymbol{B} ::= \boldsymbol{B} \cdot \boldsymbol{B} \mid 0 \mid 1$$

It is clear that this grammar can create all binary numbers, but it is also clear that this grammar is left-recursive. Giving this grammar as is to parser combinators will result in an infinite loop. Fortunately, every left-recursive grammar can be translated into one that is not left-recursive with the help of some transformation rules. Suppose we identified the "offensive" rule, then we can separate the grammar into this offensive rule and the "rest":

$$\boldsymbol{B} ::= \underbrace{\boldsymbol{B} \cdot \boldsymbol{B}}_{\textit{lft-rec}} \mid \underbrace{0 \mid 1}_{\textit{rest}}$$

To make the idea of the transformation clearer, suppose the left-recursive rule is of the form $\boldsymbol{B}\alpha$ (the left-recursive non-terminal followed by something called α) and the "rest" is called β. That means our grammar looks schematically as follows

$$\boldsymbol{B} ::= \boldsymbol{B} \cdot \alpha \mid \beta$$

To get rid of the left-recursion, we are required to introduce a new non-terminal, say $\boldsymbol{B'}$ and transform the rule as follows:

$$\begin{aligned}\boldsymbol{B} &::= \beta \cdot \boldsymbol{B'}\\ \boldsymbol{B'} &::= \alpha \cdot \boldsymbol{B'} \mid \epsilon\end{aligned}$$

In our example of binary numbers we would after the transformation end up with the rules

$$\begin{aligned}\boldsymbol{B} &::= 0 \cdot \boldsymbol{B'} \mid 1 \cdot \boldsymbol{B'}\\ \boldsymbol{B'} &::= \boldsymbol{B} \cdot \boldsymbol{B'} \mid \epsilon\end{aligned}$$

A little thought should convince you that this grammar still derives all the binary numbers (for example 0 and 1 are derivable because $\boldsymbol{B'}$ can be ϵ). Less clear might be why this grammar is non-left recursive. For $\boldsymbol{B'}$ it is relatively clear because we will never be able to derive things like

$$\boldsymbol{B'} \rightarrow \ldots \rightarrow \boldsymbol{B'} \cdot \ldots$$

because there will always be a $\boldsymbol{B}$ in front of a $\boldsymbol{B'}$, and $\boldsymbol{B}$ now has always a 0 or 1 in front, so a $\boldsymbol{B'}$ can never be in the first place. The reasoning is similar for $\boldsymbol{B}$: the 0 and 1 in the rule for $\boldsymbol{B}$ "protect" it from becoming left-recursive. This transformation does not mean the grammar is the simplest left-recursive grammar for binary numbers. For example the following grammar would do as well

$$\boldsymbol{B} ::= 0 \cdot \boldsymbol{B} \mid 1 \cdot \boldsymbol{B} \mid 0 \mid 1$$

The point is that we can in principle transform every left-recursive grammar into one that is non-left-recursive one. This explains why often the following grammar is used for arithmetic expressions:

$$\begin{aligned}\boldsymbol{E} &::= \boldsymbol{T} \mid \boldsymbol{T} \cdot + \cdot \boldsymbol{E} \mid \boldsymbol{T} \cdot - \cdot \boldsymbol{E}\\ \boldsymbol{T} &::= \boldsymbol{F} \mid \boldsymbol{F} \cdot * \cdot \boldsymbol{T}\\ \boldsymbol{F} &::= \mathit{num\_token} \mid (\cdot \boldsymbol{E} \cdot)\end{aligned}$$

In this grammar all ***E***xpressions, ***T***erms and ***F***actors are in some way protected from being left-recusive. For example if you start $\boldsymbol{E}$ you can derive another one by going through $\boldsymbol{T}$, then $\boldsymbol{F}$, but then $\boldsymbol{E}$ is protected by the open-parenthesis.

Removing ϵ-Rules and CYK-Algorithm

I showed above that the non-left-recursive grammar for binary numbers is

$$\begin{aligned}\boldsymbol{B} &::= 0 \cdot \boldsymbol{B'} \mid 1 \cdot \boldsymbol{B'}\\ \boldsymbol{B'} &::= \boldsymbol{B} \cdot \boldsymbol{B'} \mid \epsilon\end{aligned}$$

The transformation made the original grammar non-left-recursive, but at the expense of introducing an ϵ in the second rule. Having an explicit ϵ-rule is annoying to, not in terms of looping, but in terms of efficiency. The reason is that the ϵ-rule always applies but since it recognises the empty string, it does not make any progress with recognising a string. Better are rules like $(\cdot\boldsymbol{E}\cdot)$ where something of the input is consumed. Getting rid of ϵ-rules is also important for the CYK parsing algorithm, which can give us an insight into the complexity class of parsing.

It turns out we can also by some generic transformations eliminate ϵ-rules from grammars. Consider again the grammar above for binary numbers where have a rule $\boldsymbol{B'} ::= \epsilon$. In this case we look for rules of the (generic) form $\boldsymbol{A} := \alpha \cdot \boldsymbol{B'} \cdot \beta$. That is there are rules that use $\boldsymbol{B'}$ and something (α) is in front of $\boldsymbol{B'}$ and something follows (β). Such rules need to be replaced by additional rules of the form $\boldsymbol{A} := \alpha \cdot \beta$. In our running example there are the two rules for $\boldsymbol{B}$ which fall into this category

$$\boldsymbol{B} ::= 0 \cdot \boldsymbol{B'} \mid 1 \cdot \boldsymbol{B'}$$

To follow the general scheme of the transfromation, the α is either is either 0 or 1, and the β happens to be empty. So we need to generate new rules for the form $\boldsymbol{A} := \alpha \cdot \beta$, which in our particular example means we obtain

$$\boldsymbol{B} ::= 0 \cdot \boldsymbol{B'} \mid 1 \cdot \boldsymbol{B'} \mid 0 \mid 1$$

Unfortunately $\boldsymbol{B'}$ is also used in the rule

$$\boldsymbol{B'} ::= \boldsymbol{B} \cdot \boldsymbol{B'}$$

For this we repeat the transformation, giving

$$\boldsymbol{B'} ::= \boldsymbol{B} \cdot \boldsymbol{B'} \mid \boldsymbol{B}$$

In this case α was substituted with $\boldsymbol{B}$ and β was again empty. Once no rule is left over, we can simply throw away the ϵ rule. This gives the grammar

$$\begin{aligned} \boldsymbol{B} &::= 0 \cdot \boldsymbol{B'} \mid 1 \cdot \boldsymbol{B'} \mid 0 \mid 1 \\ \boldsymbol{B'} &::= \boldsymbol{B} \cdot \boldsymbol{B'} \mid \boldsymbol{B} \end{aligned}$$

I let you think about whether this grammar can still recognise all binary numbers and whether this grammar is non-left-recursive. The precise statement for the transformation of removing ϵ-rules is that if the original grammar was able to recognise only non-empty strings, then the transformed grammar will be equivalent (matching the same set of strings); if the original grammar was able to match the empty string, then the transformed grammar will be able to match the same strings, *except* the empty string. So the ϵ-removal does not preserve equivalence of grammars, but the small defect with the empty string is not important for practical purposes.

So why are these transformations all useful? Well apart from making the parser combinators work (remember they cannot deal with left-recursion and are inefficient with ϵ-rules), a second reason is that they help with getting any insight into the complexity of the parsing problem. The parser combinators are very easy to implement, but are far from the most efficient way of processing input (they can blow up exponentially with ambiguous grammars). The question remains what is the best possible complexity for parsing? It turns out that this is $O(n^3)$ for context-free languages.

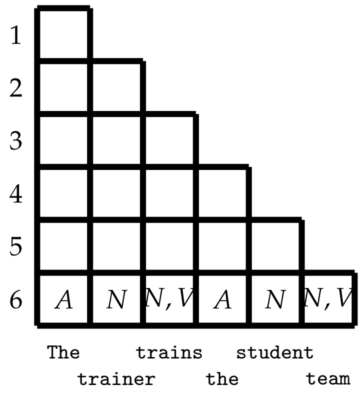
To answer the question about complexity, let me describe next the CYK algorithm (named after the authors Cocke–Younger–Kasami). This algorithm works with grammars that are in *Chomsky normalform*. In Chomsky normalform all rules must be of the form $\boldsymbol{A} ::= a$, where a is a terminal, or $\boldsymbol{A} ::= \boldsymbol{B} \cdot \boldsymbol{C}$, where $\boldsymbol{B}$ and $\boldsymbol{B}$ need to be non-terminals. And no rule can contain ϵ. The following grammar is in Chomsky normalform:

$$
\begin{array}{rcl}
\boldsymbol{S} & ::= & \boldsymbol{N} \cdot \boldsymbol{P} \\
\boldsymbol{P} & ::= & \boldsymbol{V} \cdot \boldsymbol{N} \\
\boldsymbol{N} & ::= & \boldsymbol{N} \cdot \boldsymbol{N} \\
\boldsymbol{N} & ::= & \boldsymbol{A} \cdot \boldsymbol{N} \\
\boldsymbol{N} & ::= & \texttt{student} \mid \texttt{trainer} \mid \texttt{team} \mid \texttt{trains} \\
\boldsymbol{V} & ::= & \texttt{trains} \mid \texttt{team} \\
\boldsymbol{A} & ::= & \texttt{The} \mid \texttt{the}
\end{array}
$$

where $\boldsymbol{S}$ is the start symbol and $\boldsymbol{S}, \boldsymbol{P}, \boldsymbol{N}, \boldsymbol{V}$ and $\boldsymbol{A}$ are non-terminals. The "words" are terminals. The rough idea behind this grammar is that $\boldsymbol{S}$ stands for a sentence, $\boldsymbol{P}$ is a predicate, $\boldsymbol{N}$ is a noun and so on. For example the rule $\boldsymbol{P} ::= \boldsymbol{V} \cdot \boldsymbol{N}$ states that a predicate can be a verb followed by a noun. Now the question is whether the string

```
The trainer trains the student team
```

is recognised by the grammar. The CYK algorithm starts with the following triangular data structure.



The last row contains the information about all words and their corresponding non-terminals. For example the field for `trains` contains the information $\boldsymbol{N}$ and $\boldsymbol{V}$ because it can be a "verb" and a "noun" according to the grammar. The row above, let's call the corresponding fields 5a to 5e, contains information about 2-word parts of the sentence, namely

5a) $\underbrace{\texttt{The}}_{A} \mid \underbrace{\texttt{trainer}}_{N}$
5b) $\underbrace{\texttt{trainer}}_{N} \mid \underbrace{\texttt{trains}}_{N,V}$
5c) $\texttt{trains} \mid \texttt{the}$
5d) $\texttt{the} \mid \texttt{student}$
5e) $\texttt{student} \mid \texttt{team}$

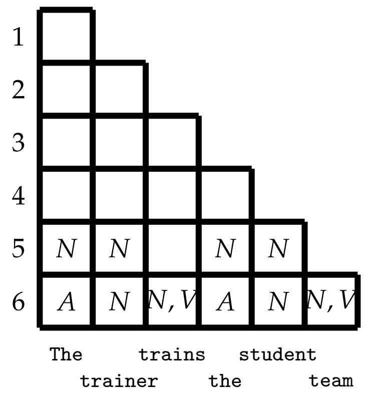
For each of them, we look up in row 6 which non-terminals it belongs to (indicated above for 5a and 5b). For 5a, with the non-terminals $\boldsymbol{A}$ and $\boldsymbol{N}$, we find the grammar rule

$$\boldsymbol{N} ::= \boldsymbol{A} \cdot \boldsymbol{N}$$

which means into field 5a we put the left-hand side of this rule, which in this case is the non-terminal $\boldsymbol{N}$. For 5b we have to check for both $\boldsymbol{N} \cdot \boldsymbol{N}$ and $\boldsymbol{N} \cdot \boldsymbol{V}$ whether there is a right-hand side of this form in the grammar. But only the grammar rule

$$\boldsymbol{N} ::= \boldsymbol{N} \cdot \boldsymbol{N}$$

matches, which means 5b gets also an $\boldsymbol{N}$. Continuing for all fields in row 5 gives:



Now row 4 is in charge of all 3-word parts of the sentence, namely

4a)	The | trainer trains
	The trainer | trains
4b)	trainer | trains the
	trainer trains | the
4c)	trains | the student
	trains the | student
4d)	the | student team
	the student | team

Note that in case of 3-word parts we have two splits. For example for 4a: $\underbrace{\texttt{The}}_{A}$ and $\underbrace{\texttt{trainer trains}}_{N}$; and also $\underbrace{\texttt{The trainer}}_{N}$ and $\underbrace{\texttt{trains}}_{N,V}$. For each of these splits we have to look up in the rows below, which non-terminals we already computed. This allows us to look for right-hand sides in our grammar: $\boldsymbol{A} \cdot \boldsymbol{N}$, $\boldsymbol{N} \cdot \boldsymbol{N}$ and $\boldsymbol{N} \cdot \boldsymbol{V}$, which yield the only left-hand side $\boldsymbol{N}$. This is what we fill in for 4a. And so on for row 4.

Row 3 is about all 4-word parts in the sentence, namely

3a)	The trainer trains the
3b)	trainer trains the student
3c)	trains the student team

Each of them can be split up in 3 ways, for example

3a)	The | trainer trains the
	The trainer | trains the
	The trainer trains | the

and we have to consider them all in turn to fill in the non-terminals for 3a. You can guess how it continues: row 2 is for all 5-word parts, and finally the field on the top is for the whole sentence. The idea of the CYK algorithm is that if in the top-field the starting symbol $\boldsymbol{S}$ appears (possibly together with other non-terminals), then the sentence is accepted by the grammar. If it does not, then the sentence is not accepted.

Let us very quickly calculate the complexity of the CYK algorithm. Lookup operations inside the triangle and in the grammar are assumed to be of constant time, $O(1)$, meaning they do not matter. How many fields are in the triangle… $\frac{n}{2} * (n+1)$, where n is the size of the input. That means roughly $O(n^2)$ fields. How much work do we have to do for each field? Well, for the top-most we have to consider $n-1$ splits, which means roughly $O(n)$ for each field. The overall result is a $O(n^3)$ time-complexity for CYK. It turns out that this is the best worst-time complexity for parsing with context-free grammars.