Context-Free Grammars

Context-free grammars: why we need them. In programming languages, many constructions are defined via others, e.g.:

```
<if-statement> = if(<condition>)<statement>
<if-statement> = if(<condition>)<statement>
  else <statement>
```

Here, `<condition>` indicates that different expressions can be used here. This means that whatever condition we have, we can substitute it here, and we will get a correct Java program.

What is possible to substitute does not depend on what is before or after this expression, i.e., does not depend on the *context*. Because of this, the corresponding description is known as *context-free grammar* (CFG, for short).

Variables and terminal symbols. In automata theory, notations like `<condition>` – which can mean different things – are called *variables*.

On the other hand, the text like `if(` will be verbatim reproduced in the final program that contains if-statements. In automata theory, such unchangeable pieces of text are called *terminal symbols*.

Notations: general description.

- Variables are usually described by capital letters.
- Terminal symbols are usually described by small letters.
- The fact that the right-hand side is a particular case of the left-hand side notion is described by an arrow $\rightarrow$.

Notations: example. Let us describe the above two lines in these terms. To do it, we need to describe each variable by a single capital letter. For this example, let us simply use the capitalized first letter, so:

- the variable `<if-statement>` will be described by the letter I;
- the variable `<condition>` will be described by the letter C; and
- the variable `<statement>` will be described by the letter S.

In these notations, the above rules take the following form:

$$I \to if(C)S$$

$$I \to if(C)S\ elseS$$

What is terminal symbol and what is a variable: examples. The purpose of a context-free grammar is to generate words in some language. All the symbols from that language are known as *terminal symbols*. For example, for arithmetic expressions, +, −, (,), 0, 1, ..., are all terminal symbols. For Java programs, the list of possible symbols includes =, <, etc. – all the symbols that you can use in a Java program.

To generate these words, we use rules about concepts. For example, we can say that `<if-statement>` consists of terminal symbols i, f, (, followed by `<condition>`, followed by), followed by `<statement>`:

```
<if-statement> = if(<condition>)<statement>
```

In context-free grammars this rule is described as $I \to if(C)S$.

In contrast to symbols +, 0, etc., that can be part of a Java program, the word `if-statement`, while useful to describe Java programs, is *not* part of a Java program. If you write `if-statement` in Java, the Java program will not know what to do. Symbols of this type, that are useful to generate words, but that are *not* from the desired language, are known as *variables*.

In the rule $I \to if(C)S$, symbols i, f, (, and) will appear in the resulting Java program, so they are terminal symbols. In contrast, symbols C and S are *not* part of the Java program, they have to be replaced by some condition and some statement – so they are variables.

Why do we need these notations? There exist programs that:

- take the description of a programming language in these terms and
- produce a compiler for this language.

Such programs are known as *compiler compilers*. Probably the most well known is *yacc* – this is an abbreviation for "yet another compiler compiler".

The resulting compiler is not optimal, faster compilers are usually possible, but it enables us to compile and run programs in newly developed languages without waiting months and years for programmers to come up with a very good compiler.

How do we generate words in the CFG?

- Each grammar has a *starting variable* – this is the variable with which we start.
- Every time we have an expression containing a variable V, and there is a rule $V \to w$, we can replace V with the right-hand side of the rule.

This procedure continues until we get a word that does not have any variables, only terminal symbols.

Why we suddenly study context-free grammars while we have not finished studying pushdown automata. The answer to this question is straightforward:

- just like finite automata are equivalent to regular expressions – in the sense that the language recognized by a finite automata can be described by a regular expression, and for each regular expression we can design a finite automation recognizing the corresponding language,
- non-deterministic pushdown automata are equivalent to context-free grammars: the language recognized by each pushdown automaton is context-free and, vice versa, for every context-free grammar, we can design a pushdown automaton that recognized exactly the words generated by this grammar.

Example: CFG for the language $\{a^n b^n : n = 0, 1, 2, \ldots\}$. Let us start by designing a CFG for the language

$$L = \{a^n b^n : n = 0, 1, 2, \ldots\} = \{\Lambda, ab, aabb, \ldots\}$$

for which we proved that this language cannot be described by a regular expression.

This CFG comes from the following two natural rules.

The first rule comes from considering the simplest element of this language – the empty string. This rule says that the empty string is the element of this language. In terms of the CFG, this can be described as

$$S \to \varepsilon.$$

The second rule explains how, based on a word from the language L, we can construct another word from this language. Namely, to preserve the equal number of a's and b's, we can add a to the left and b to the right. In general:

- if we have a word from the language L, i.e., a word of the type $a \ldots ab \ldots b$ that has equal number of a's and b's,
- and we add a to the left and b to the right, then we also get a word from this language.

This rule can be described as

$$S \to aSb$$

This language has only one variable S.

Let us show that by using these two rules, we can indeed generate all possible words from the language L. Indeed, we start with the variable S. If we apply the first rule to this variable, we get the empty string

$$S \to \varepsilon.$$

So, the empty string is indeed generated by this language.

If we apply the second rule, we get the expression aSb. This expression not final, it still contains a variable S. To this variable, we can again apply either the first rule or the second rule.

If we apply the first rule, we get the word ab. So, the word ab is indeed generated by this grammar. If we apply the second rule, we get the word $aaSbb$. To this new expression, we can again apply one of the two rules, so we get either $aabb$ or $aaaSbbb$, etc.

The resulting derivation can be described if we underline the variable that is replaced at each step. For example;

- the derivation of the empty string can be described as

$$\underline{S} \to \varepsilon;$$

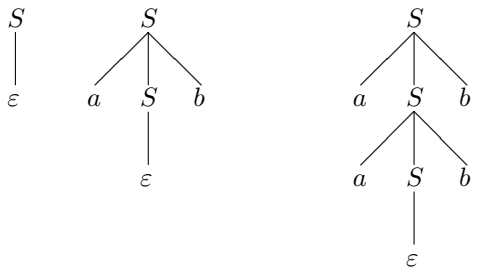
- the derivation of the word ab can be described as

$$\underline{S} \to a\underline{S}b \to ab;$$

- the derivation of the word $aabb$ can be described as

$$\underline{S} \to a\underline{S}b \to aa\underline{S}bb \to aabb.$$

Another way to describe such derivations is to draw a tree, with the starting symbol as the root:



Practice.

- Try deriving other words from this language.
- Try formulating rules for similar languages, such as

$$\{a^n b^{n+1}, n = 0, 1, 2, \ldots\} = \{b, abb, aabbb, \ldots\}$$

and

$$\{a^n b^{2n}, n = 0, 1, 2, \ldots\} = \{\Lambda, abb, aabbbb, \ldots\}.$$

Example: CFG for the class of all palindromes. A palindrome is a word that reads the same way front-to-end and end-to-front. A known joke says that the very first words said by a human being were a palindrome: when Adam saw Eve for the first time, he introduced himself: "Madam, I'm Adam".

Seriously, what rules can we use to design palindromes? As before, let us start with the simplest words from this language. First, the empty string is a palindrome, so we have a rule

$$S \to \varepsilon.$$

Also, every 1-letter word is a palindrome, so we have rules:

$$S \to a$$

$$S \to b$$

$$\dots$$

$$S \to z.$$

If we have a word which is a palindrome, how can we extend it and still keep it a palindrome? If we add a letter a to the left of the word, then, to preserve the palindrome property, we must add the same letter a to the right of the word. Similarly, we can add the same letter b to both sides of the word and still get a palindrome. This can be expressed by the following rules:

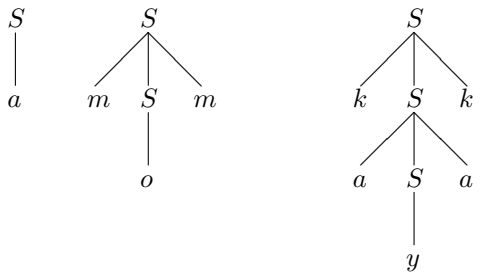
$$S \to aSa$$

$$S \to bSb$$

$$\dots$$

$$S \to zSz$$

Let us illustrate how these rules can derive the palindromes a, mom, and $kayak$:



Practice. Try deriving other palindromes, e.g., *hannah*.