Grammar Formalism

Regular Grammars-Right Linear and Left Linear Grammars

- In theoretical computer science and formal language theory, a **regular grammar** is a formal grammar that is right-regular or left-regular. Every regular grammar describes a regular language.
- **Strictly regular grammars**

A **right regular grammar** (also called right linear grammar) is a formal grammar (N, Σ, P, S) such that all the production rules in P are of one of the following forms:

1. B → a - where B is a non-terminal in N and a is a terminal in Σ
2. B → aC - where B and C are non-terminals in N and a is in Σ
3. B → ε - where B is in N and ε denotes the empty string, i.e. the string of length 0.

In a **left regular grammar** (also called left linear grammar), all rules obey the forms

1. A → a - where A is a non-terminal in N and a is a terminal in Σ
2. A → Ba - where A and B are in N and a is in Σ
3. A → ε - where A is in N and ε is the empty string.

- A **regular grammar** is a left or right regular grammar. Some textbooks and articles disallow empty production rules, and assume that the empty string is not present in languages.

  Example: An example of a right regular grammar G with N = {S, A}, Σ = {a, b, c}, P consists of the following rules

  S → aS

  S → bA

  A → ε

  A → cA

  Where S is the start symbol. This grammar describes the same language as the regular expression a*bc*, viz. the set of all strings consisting of arbitrarily many "a"s, followed by a single "b", followed by arbitrarily many "c"s.

- A somewhat longer but more explicit extended right regular grammar G for the same regular expression is given by N = {S, A, B, C}, Σ = {a, b, c}, where P consists of the following rules:

  S → A

  A → aA

  A → B

  B → bC

C → ε

C → cC

Where each uppercase letter corresponds to phrases starting at the next position in the regular expression.

- As an example from the area of programming languages, the set of all strings denoting a floating point number can be described by a right regular grammar G with N = {S, A,B,C,D,E,F}, Σ = {0,1,2,3,4,5,6,7,8,9,+,-,.,e}, where S is the start symbol, and P consists of the following rules:

S → +A	A → 0A	B → 0C	C → 0C	D → +E	E → 0F	F → 0F
S → -A	A → 1A	B → 1C	C → 1C	D → -E	E → 1F	F → 1F
S → A	A → 2A	B → 2C	C → 2C	D → E	E → 2F	F → 2F
	A → 3A	B → 3C	C → 3C		E → 3F	F → 3F
	A → 4A	B → 4C	C → 4C		E → 4F	F → 4F
	A → 5A	B → 5C	C → 5C		E → 5F	F → 5F
	A → 6A	B → 6C	C → 6C		E → 6F	F → 6F
	A → 7A	B → 7C	C → 7C		E → 7F	F → 7F
	A → 8A	B → 8C	C → 8C		E → 8F	F → 8F
	A → 9A	B → 9C	C → 9C		E → 9F	F → 9F
	A → .B		C → eD			F → ε
	A → B		C → ε			

Linear Grammer:

- In computer science, a **linear grammar** is a context-free grammar that has at most one nonterminal in the right hand side of its productions.
- A **linear language** is a language generated by some linear grammar.

  Example:

  A simple linear grammar is G with N = {S}, Σ = {a, b}, P with start symbol S and rules

  S → aSb

  S → ε It generates the language .

Relationship with Regular Grammars:

- Two special types of linear grammars are the following:
  - The **left-linear** or left regular grammars, in which **all nonterminals** in right hand sides are **at the left ends**;
  - The **right-linear** or right regular grammars, in which **all nonterminals** in right hand sides are **at the right ends**.
- Collectively, these two special types of linear grammars are known as the regular grammars; both can describe exactly the regular languages.
- Another special type of linear grammar is the following:
  - Linear grammars in which all nonterminals in right hand sides are at the left or right ends, but not necessarily all at the same end.
  - By inserting new nonterminals, every linear grammar can be brought into this form without affecting the language generated. For instance, the rules of G above can be replaced with

    $S \rightarrow aA$

    $A \rightarrow Sb$

    $S \rightarrow \varepsilon$

    Hence, linear grammars of this special form can generate all linear languages.

Context free grammar

Context-free language

- The language of a grammar is the set. This can also be done in reverse to check if a sentence is grammatically correct.
- Languages generated by context-free grammars are known as Context-Free Languages (CFL). Different context-free grammars can generate the same context-free language.
- It is important to distinguish properties of the language (intrinsic properties) from properties of a particular grammar (extrinsic properties).
- The language equality question (do two given context-free grammars generate the same language?) is undecidable.
- Context-free grammars arise in linguistics where they are used to describe the structure of sentences and words in natural language, and they were in fact invented by the linguist Noam Chomsky for this purpose, but have not really lived up to their original expectation. By contrast, in computer science, as the use of recursively defined concepts increased, they were used more and more.

- In an early application, grammars are used to describe the structure of programming languages. In a newer application, they are used in an essential part of the Extensible Markup Language (XML) called the Document Type Definition.
- In linguistics, some authors use the term phrase structure grammar to refer to context-free grammars, whereby phrase structure grammars are distinct from dependency grammars. In computer science, a popular notation for context-free grammars is Backus–Naur Form, or BNF.
- Proper CFGs: A context-free grammar is said to be proper, if it has
  - No unreachable symbols
  - No unproductive symbols
  - No ε-productions
  - No cycles
- Every context-free grammar can be effectively transformed into a weakly equivalent one without unreachable symbols, a weakly equivalent one without unproductive symbols, and a weakly equivalent one without cycles.
- Every context-free grammar not producing ε can be effectively transformed into a weakly equivalent one without ε-productions altogether, every such grammar can be effectively transformed into a weakly equivalent proper CFG.

Examples

- The grammar, with productions as following is context-free. It is not proper since it includes an ε-production.

  $S \rightarrow aSa$,

  $S \rightarrow bSb$,

  $S \rightarrow \varepsilon$,

  A typical derivation in this grammar is

  $S \rightarrow aSa \rightarrow aaSaa \rightarrow aabSbaa \rightarrow aabbaa$

  This makes it clear that. The language is context-free, however it can be proved that it is not regular.
- **Well-formed Parentheses**
  - The canonical example of a context free grammar is parenthesis matching, which is representative of the general case. There are two terminal symbols "(" and ")" and one nonterminal symbol S. The production rules are

    $S \rightarrow SS$

    $S \rightarrow (S)$

S → ()

- The first rule allows the S symbol to multiply; the second rule allows the S symbol to become enclosed by matching parentheses; and the third rule terminates the recursion.

- **Well-formed Nested Parentheses and Square Brackets**
  - A second canonical example is two different kinds of matching nested parentheses, described by the productions:

    S → SS

    S → ()

    S → (S)

    S → []

    S → [S]

    with terminal symbols [ ] ( ) and nonterminal S.
  - The following sequence can be derived in that grammar:

    ([[[()() [ ][ ]]]([ ])])

    However, there is no context-free grammar for generating all sequences of two different types of parentheses, each separately balanced disregarding the other, but where the two types need not nest inside one another, for example:

    [( ])

or

[[[[(((( ]]]]))))(([ ))(([ ))([ )( ])( ])( ])

- **A regular grammar**
  - Every regular grammar is context-free, but not all context-free grammars are regular. The following context-free grammar, however, is also regular.

    S → a

    S → aS

    S → bS

    The terminals here are a and b, while the only non-terminal is S. The language described is all nonempty strings of s and s that end in.
  - This grammar is regular, no rule has more than one nonterminal in its right-hand side, and each of these nonterminals is at the same end of the right-hand side. Every regular grammar corresponds directly to a nondeterministic finite automaton, so we know that this is a regular language.

- Using pipe symbols, the grammar above can be described more tersely as follows:

  S → a | aS | bS

- **Matching pairs**
  - In a context-free grammar, we can pair up characters the way we do with brackets. The simplest example:

    S → aSb

    S → ab
  - This grammar generates the language, which is not regular (according to the pumping lemma for regular languages).
  - The special character ε stands for the empty string. By changing the above grammar to

    S → aSb | ε we obtain a grammar generating the language instead. This differs only in that it contains the empty string while the original grammar did not.

- **Algebraic expressions**
  - Here is a context-free grammar for syntactically correct infix algebraic expressions in the variables x, y and z:

1. S → x
2. S → y
3. S → z
4. S → S + S
5. S → S - S
6. S → S * S
7. S → S / S
8. S → (S)

- This grammar can, for example, generate the string

  (x + y) * x - z * y / (x + x) as follows:

  S (the start symbol)

  → S - S (by rule 5)

  → S * S - S (by rule 6, applied to the leftmost S)

  → S * S - S / S (by rule 7, applied to the rightmost S)

  → (S) * S - S / S (by rule 8, applied to the leftmost S)

  → (S) * S - S / (S) (by rule 8, applied to the rightmost S)

  → (S + S) * S - S / (S) (etc.)

→ (S + S) * S - S * S / (S)

→ (S + S) * S - S * S / (S + S)

→ (x + S) * S - S * S / (S + S)

→ (x + y) * S - S * S / (S + S)

→ (x + y) * x - S * y / (S + S)

→ (x + y) * x - S * y / (x + S)

→ (x + y) * x - z * y / (x + S)

→ (x + y) * x - z * y / (x + x)

- Note that many choices were made underway as to which rewrite was going to be performed next. These choices look quite arbitrary. As a matter of fact, they are, in the sense that the string finally generated is always the same. For example, the second and third rewrites

→ S * S - S (by rule 6, applied to the leftmost S)

→ S * S - S / S (by rule 7, applied to the rightmost S) could be done in the opposite order:

→ S - S / S (by rule 7, applied to the rightmost S)

→ S * S - S / S (by rule 6, applied to the leftmost S)

- Also, many choices were made on which rule to apply to each selected S. Changing the choices made and not only the order they were made in usually affects which terminal string comes out at the end.
- Let's look at this in more detail. Consider the parse tree of this derivation:

```
                S
                |
               /|\
              S - S
             /     \
           /|\     /|\
          S * S   S / S
         /   | |     \
       /|\   x /|\   /|\
      ( S )   S * S ( S )
      /       |  |    \
    /|\       z  y    /|\
   S + S             S + S
```

```
        |  |        |  |
        x  y        x  x
```

- Starting at the top, step by step, an S in the tree is expanded, until no more unexpanded Ses (non-terminals) remain. Picking a different order of expansion will produce a different derivation, but the same parse tree. The parse tree will only change if we pick a different rule to apply at some position in the tree.
- But can a different parse tree still produce the same terminal string, which is (x + y) * x - z * y / (x + x) in this case? Yes, for this particular grammar, this is possible. Grammars with this property are called ambiguous.
- For example, x + y * z can be produced with these two different parse trees:

```
         S          S
         |          |
        /|\        /|\
       S * S      S + S
       /   \      /   \
     /|\    z    x    /|\
    S + S            S * S
    |   |            |   |
    x   y            y   z
```

- However, the language described by this grammar is not inherently ambiguous: an alternative, unambiguous grammar can be given for the language, for example:

$$T \rightarrow x$$

$$T \rightarrow y$$

$$T \rightarrow z$$

$$S \rightarrow S + T$$

$$S \rightarrow S - T$$

$$S \rightarrow S * T$$

$$S \rightarrow S / T$$

$$T \rightarrow (S)$$

$$S \rightarrow T$$

- This alternative grammar will produce x + y * z with a parse tree similar to the left one above, i.e. implicitly assuming the association (x + y) * z, which is not according to standard operator precedence. More elaborate, unambiguous and context-free grammars can be constructed that produce parse trees that obey all desired operator precedence and associativity rules.

- A context-free grammar for the language consisting of all strings over {a,b} containing an unequal number of a's and b's:

  $S \rightarrow U \mid V$

  $U \rightarrow TaU \mid TaT \mid UaT$

  $V \rightarrow TbV \mid TbT \mid VbT$

  $T \rightarrow aTbT \mid bTaT \mid \varepsilon$

Here, the nonterminal T can generate all strings with the same number of a's as b's, the nonterminal U generates all strings with more a's than b's and the nonterminal V generates all strings with fewer a's than b's. Omitting the third alternative in the rule for U and V doesn't restrict the grammar's language.

- It is context-free as it can be generated by the following context-free grammar:

  $S \rightarrow bSbb \mid A$

  $A \rightarrow aA \mid \varepsilon$

- The formation rules for the terms and formulas of formal logic fit the definition of context-free grammar, except that the set of symbols may be infinite and there may be more than one start symbol.

Derivations and Syntax Trees

- A derivation of a string for a grammar is a sequence of grammar rule applications that transforms the start symbol into the string. A derivation proves that the string belongs to the grammar's language.
- A derivation is fully determined by giving, for each step:
  - The rule applied in that step
  - The occurrence of its left hand side to which it is applied
- For clarity, the intermediate string is usually given as well. For instance, with the grammar:

  (1) $S \rightarrow S + S$

  (2) $S \rightarrow 1$

  (3) $S \rightarrow a$

  the string 1 + 1 + a can be derived with the derivation:

  S

  → (rule 1 on first S)

  S+S

→ (rule 1 on second S)

S+S+S

→ (rule 2 on second S)

S+1+S

→ (rule 3 on third S)

S+1+a

→ (rule 2 on first S)

1+1+a

- Often, a strategy is followed that deterministically determines the next nonterminal to rewrite:
  - In a leftmost derivation, it is always the leftmost nonterminal;
  - In a rightmost derivation, it is always the rightmost nonterminal.
- Given such a strategy, a derivation is completely determined by the sequence of rules applied. For instance, the leftmost derivation

S

→ (rule 1 on first S)

S+S

→ (rule 2 on first S)

1+S

→ (rule 1 on first S)

1+S+S

→ (rule 2 on first S)

1+1+S

→ (rule 3 on first S)

1+1+a

can be summarized as rule 1, rule 2, rule 1, rule 2, rule 3

- The distinction between leftmost derivation and rightmost derivation is important because in most parsers the transformation of the input is defined by giving a piece of code for every grammar rule that is executed whenever the rule is applied. Therefore it is important to know whether the parser determines a leftmost or a rightmost derivation because this determines the order in which the pieces of code will be executed. See for an example LL parsers and LR parsers.
- A derivation also imposes in some sense a hierarchical structure on the string that is derived. For example, if the string "1 + 1 + a" is derived according to the leftmost derivation:

$S \to S + S$ (1)

$\to 1 + S$ (2)

$\to 1 + S + S$ (1)

$\to 1 + 1 + S$ (2)

$\to 1 + 1 + a$ (3)

the structure of the string would be:

$\{ \{ 1 \}_S + \{ \{ 1 \}_S + \{ a \}_S \}_S \}_S$

where $\{ \dots \}_S$ indicates a substring recognized as belonging to S. This hierarchy can also be seen as a tree:

```
     S
    /|\
   / | \
  /  |  \
 S  '+'  S
 |      /|\
 |     / | \
'1'   S '+' S
      |     |
     '1'   'a'
```

- This tree is called a parse tree or "concrete syntax tree" of the string, by contrast with the abstract syntax tree. In this case the presented leftmost and the rightmost derivations define the same parse tree; however, there is another (rightmost) derivation of the same string

$S \to S + S$ (1)

$\to S + a$ (3)

$\to S + S + a$ (1)

$\to S + 1 + a$ (2)

$\to 1 + 1 + a$ (2)

and this defines the following parse tree:

```
     S
    /|\
   / | \
  /  |  \
     S  '+'  S
    /|\      |
   / | \     |
  S '+' S   'a'
  |     |
 '1'   '1'
```

- If, for certain strings in the language of the grammar, there is more than one parsing tree, then the grammar is said to be an ambiguous grammar. Such grammars are usually hard to parse because the parser cannot always decide which grammar rule it has to apply. Usually, ambiguity is a feature of the grammar, not the language, and an unambiguous grammar can be found that generates the same context-free language. However, there are certain languages that can only be generated by ambiguous grammars; such languages are called inherently ambiguous languages.