

1 Context-Free Grammars

1.1 Motivation

Regular expressions are **insufficient** in describing the structure of complicated languages (e.g. programming languages). We recall our previously-used example: *simple arithmetic expressions with parentheses*.

$\langle \text{expr} \rangle ::= \langle \text{atom} \rangle \mid \langle \text{expr} \rangle + \langle \text{expr} \rangle \mid (\langle \text{expr} \rangle)$

This language is **not regular**, however it can be described via a *generator object* called (context-free) **grammar** (short CFG).

Definition (CFG). A **CFG** is a 4-tuple: $G = (V, \Sigma, R, S)$ where:

- V is a finite set whose elements are called **non-terminals and terminals**
- $\Sigma \subseteq V$ is the set of **terminals**
- R is a **relation** over $(V \setminus \Sigma) \times V^*$. Here $V \setminus \Sigma$ is the set of **non-terminals** and V^* is the set of **words** over V . An element of R is called **production rule**. We explain productions below.
- $S \in V \setminus \Sigma$ is the **start symbol**.

As an example, consider the following CFG G , where:

- $V = \{S\}$
- $\Sigma = \{a, b\}$
- $R = \{S \rightarrow aSb, S \rightarrow \epsilon\}$

This grammar contains a **single** non-terminal S , which is also the start symbol. **Production rules** are written as follows:

$X \rightarrow Y$

where X is a **non-terminal** and Y is a string containing terminal and non-terminal symbols. Our grammar has two production rules:

- $S \rightarrow aSb$
- $S \rightarrow \epsilon$

We can also write production rules of the form:

$$X \rightarrow Y_1, \dots X \rightarrow Y_n$$

in the more compact form:

$$X \rightarrow Y_1 \mid \dots \mid Y_n$$

In our example, we can write:

$$S \rightarrow aSb \mid \epsilon.$$

As a general **convention**, we use **italic uppercase symbols** to designate **non-terminals**, and **lowercase symbols** (or occasionally, typewriter symbols e.g. **A**) - for **terminals**. At the same time, *S* is always used to designate the **start-symbol**. Under this convention, we can completely define a grammar by giving the set of productions only.

For instance, we can define a CFG for expressions as follows:

$$S \rightarrow S + S \mid (S) \mid A$$

$$A \rightarrow UVT$$

$$U \rightarrow \mathbf{A} \mid \dots \mid \mathbf{Z}$$

$$V \rightarrow LV \mid \epsilon$$

$$L \rightarrow \mathbf{a} \mid \dots \mid \mathbf{z}$$

$$T \rightarrow DT \mid \epsilon$$

$$D \rightarrow 0 \mid \dots \mid 1$$

We have preserved the same convention for atoms: they must start with an uppercase, followed by zero-or-more lowercase symbols, and then zero-or-more digits.

(Context-Free) Grammars are the **corner-stone** for writing parsers.

1.2 The language of a CFG

Let $\alpha A \beta$ and $\alpha \gamma \beta$ be strings from V^* , where *A* is a **non-terminal**. Also, suppose we have a production $A \rightarrow \gamma$ in a CFG *G*. Then we say:

$$\alpha A \beta \Rightarrow_G \alpha \gamma \beta$$

and read that $\alpha \gamma \beta$ is a **one-step derivation** of $\alpha A \beta$. The relation over strings \Rightarrow_G is very similar in spirit to \vdash_M . We omit the subscript when the grammar *G* is understood from context, and write \Rightarrow^* to refer to the **reflexive and transitive closure of \Rightarrow** . \Rightarrow^* is the **zero-or-more steps derivation** relation.

As an example, consider the grammar for arithmetic expressions, and the following derivation:

$$S \Rightarrow (S) \Rightarrow (S + S) \Rightarrow (A + S) \Rightarrow (A + (S + S)) \Rightarrow (A + (A + S)) \Rightarrow (A + (A + A))$$

Hence, we have $S \Rightarrow^* (A + (A + A))$.

Notice that the string $(A + (A + A))$ contains **non-terminals**. One possible derivation for *A* is:

$$A \Rightarrow UVT \Rightarrow \mathbf{x}VT \Rightarrow \mathbf{x}T \rightarrow \mathbf{x}DT \rightarrow \mathbf{x}0T \rightarrow \mathbf{x}0$$

Similarly, we may write derivations that witness: $A \Rightarrow^* \mathbf{Y}$ and $A \Rightarrow^* \mathbf{Z}$.

and finally: $S \Rightarrow^* (A + (A + A)) \Rightarrow^* (X0 + (Y + Z))$. Notice that $(X0 + (Y + Z))$ contains only **terminal symbols**.

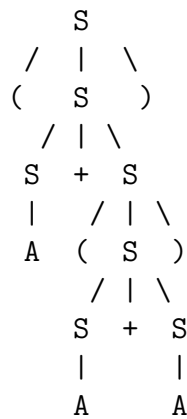
Definition (Language of a grammar). *For a CFG G , the **language generated by G** is defined as: $L(G) = \{w \in \Sigma^* \mid S \Rightarrow_G^* w\}$*

Informally, $L(G)$ is the set of words that be obtained via **zero-or-more** derivations from G .

If a language is generated by a CFG, then it is called a **context-free language**.

1.3 Parse trees

Informally, a parse tree is an *illustration of sequences of derivations*. We illustrate a parse tree for $(A + (A + A))$ below:

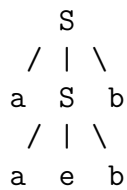


Notice that there is not a **one-to-one** correspondence between a sequence of derivations and a parse-tree. For instance, we may first derive the left-hand side of $+$, or the right-hand side. However, a parse-tree uniquely identifies the set of productions used in the derivation and how they are applied.

The construction rules for parse trees are as follows:

- the **root** of the tree is the **start symbol**
- each **interior node** X having as children nodes Y_1, \dots, Y_n corresponds to a **production rule** $X \rightarrow Y_1 \dots Y_n$
- if **each leaf** is a **terminal**, then the parse-tree **yields** a word of $L(G)$.

For instance, the following parse-tree:

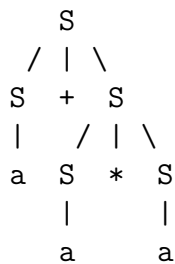


yields the word *aabb*, which is obtained by concatenating each terminal leaf from left to right.

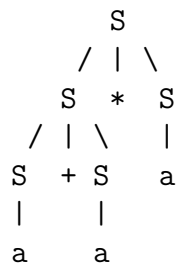
Parse-trees are especially useful for parsing, because they reveal **the structure** of a parsed program (or word in general).

It is only natural that we require **the program structure to be //unique//**. However, it is quite easy to find grammars where **the same word** has **different** parse trees as yield.

Consider the following CFG: $S \rightarrow S + S \mid S * S \mid a$
and the word: $a + a * a$ has different parse trees:



and



Incidentally, these two different *structures* reflect different interpretations of our arithmetic expression. Thus, our grammar is **ambiguous**. In general, a grammar is ambiguous if there **exist two different parse trees for the same word**.

To remove ambiguity in our example, it is sufficient to:

1. include **precedence-rules** in the grammar;
2. enforce parsing to proceed *left-to-right*;

The result is:

$$S \rightarrow M + S \mid M$$

$$M \rightarrow T * M \mid T$$

$$T \rightarrow a$$

The first production rule enforces *left-to-right* parsing. Consider the alternative production: $S \rightarrow S + S \mid M$

Via this production, a parse tree might unfold *to the left* ad-infinitum, **depending on how the parser implementation works**:

$$\begin{array}{c} S \\ / \mid \backslash \\ S \quad + \quad S \\ / \mid \backslash \\ S \quad + \quad S \\ \dots \end{array}$$

By instead using:

$S \rightarrow M + S \mid M$, we are requiring that a suitable **multiplication term** be found at the left of $+$, while any expression may occur at it's right.

The second production describes **multiplication terms**. Note that, under this grammar, addition cannot appear within a multiplication term. If this is the case, we need a new production rule which includes parentheses. Can you figure how this modification should be done?

1.3.1 Solving ambiguity in general

Consider another example:

$$L = \{a^n b^n c^m d^m \mid n, m \geq 1\} \cup \{a^n b^m c^m d^n \mid n, m \geq 1\}$$

This language contains strings in $L(aa^*bb^*cc^*dd^*)$ where ($\text{number}(a)=\text{number}(b)$ and $\text{number}(c)=\text{number}(d)$) or ($\text{number}(a)=\text{number}(d)$ and $\text{number}(b)=\text{number}(c)$).

One possible CFG is:

$$S \rightarrow AB \mid C$$

$$A \rightarrow aAb \mid ab$$

$$B \rightarrow cBd \mid cd$$

$$C \rightarrow aCd \mid aDc$$

$$D \rightarrow bDc \mid bc$$

This CFG is **ambiguous**: the word $aabbccdd$ has two different parse trees:

$$\begin{array}{c} S \\ / \quad \backslash \\ A \quad B \\ \dots \quad \dots \end{array}$$

and

```

      S
      |
      C
    / | \
...  ...

```

The reason for ambiguity is that in *aabbccdd* both conditions of the grammar hold ($\text{number}(a)=\text{number}(b)=\text{number}(c)=\text{number}(d)$).

It is not straightforward how ambiguity can be lifted from this grammar. This particular example raises two interesting questions:

- can we **automatically** lift ambiguity from any CFG?
- how to find an unambiguous grammar for a Context-Free Language?

We cannot provide a general answer for any of the above questions. In fact:

The problem of establishing if a CFG is ambiguous is not decidable.

Also **there exist context-free languages for which no unambiguous grammar exists.** Our above example is such a language.

1.4 Regular grammars

Definition (Regular grammars). *A grammar is called **regular** iff all its production rules have one of the following forms:*

$X \rightarrow aA$

$X \rightarrow A$

$X \rightarrow a$

$X \rightarrow \epsilon$

where A, X are nonterminals and a is a terminal.

Formally:

$R \subseteq (V \setminus \Sigma) \times (\Sigma^*((V \setminus \Sigma) \cup \{\epsilon\}))$

Thus:

- each production rule contains **at most one non-terminal**
- each non-terminal appears as *the last symbol in the production body*

As it turns out, regular grammars **precisely capture regular languages:**

*theorem[Regular grammars capture regular languages] A language is **regular** iff it is generated by*

Proof. Direction \implies . Suppose L is a regular language, i.e. it is accepted by a DFA $M = (K, \Sigma, \delta, q_0, F)$. We build a regular grammar G from M . Informally, each production of G mimics some transition of M . Formally, $G = (V, \Sigma, R, S)$ where:

- $V = K \cup \Sigma$ - the set of non-terminals is the set of states, and the set of terminals is the set of symbols;
- $S = q_0$ - the start-symbol corresponds to the start state;
- for each transition $\delta(q, c) = p$, we build a production rule $q \rightarrow cp$. For each final state $q \in F$, we build a production rule $q \rightarrow \epsilon$

The grammar is obviously regular. To prove $L(M) = L(G)$, we must show: for all $w = c_1 \dots c_n \in \Sigma^*$, $(q_0, c_1 \dots c_n) \vdash_M^* (p, \epsilon)$ with $p \in F$, **iff** $p_0 \Rightarrow_G^* c_1 \dots c_n p$, where we recall that p is a non-terminal for which the production rule $p \rightarrow \epsilon$ exists. The above proposition can be easily proven by induction over the length of the word w .

Direction \Leftarrow . Suppose $G = (V, \Sigma, R, S)$ is a regular grammar. We build an NFA $M = (K, \Sigma, \Delta, q_0, F)$ whose transitions *mimic* production rules:

- $K = (V \setminus \Sigma) \cup \{p\}$: for each non-terminal in G , we build a state in M . Additionally, we build a final state.
- $q_0 = S$
- $F = \{p\}$
- for each production rule $A \rightarrow cB$ in G , where B is a non-terminal and $c \in \Sigma^*$, we build a transition $(A, c, B) \in \Delta$. Also, for each transition $A \rightarrow c$, with $c \in \Sigma^*$, we build a transition $(A, c, p) \in \Delta$.

We must prove that:

for all $w = c_1 \dots c_n \in \Sigma^*$, we have $S \Rightarrow_G^* c_1 \dots c_n$ **iff** $(q_0, c_1 \dots c_n) \vdash_M^* (p, \epsilon)$.

The proof is similar to the above one.

□

The theorem also shows that Context-Free Languages are a proper **superset** of Regular Languages.