Context free languages

People intuitively recognize when a sentence spoken in the native language is correct or not. We act as language recognizers. A language recognizer is a device that accepts valid strings produced in a given language. The finite state automata of the last chapter are formalized types of language recognizers.

We are also capable of producing legal sentences, at least in our mother tongue. So we occasionally speak and write legal sentences (even if there are lies). In this respect, we act as language generators.
A language generator begins, when given a start signal, to construct a string. Its operation is not completely determined from the beginning but is limited by a set of rules.

Regular expressions can be viewed as language generators. For example, consider the regular expression $a(a^* \cup b^*)b$.
A verbal description of how to generate a string in accordance with this expression would be:

First output an *a*. Then do one of the following two things:

> **Either output a number of *a*'s or output a number of *b*'s**
>
> **Finally output a *b*.**

In this chapter, we will study certain more complex sorts of language generators, called context-free grammars, which are based on a more complete understanding of the structure of the strings belonging to the language.

To take the example of : $a(a^* \cup b^*)b$, we can describe the strings generated by this language as consisting of a leading *a*, followed by a middle part M, to be made explicit later on, followed by a trailing *b*. Let S be a new symbol interpreted as " a string in the language", we can express this observation by writing:

$S \rightarrow aMb$

→ is read *can be*. We call such an expression a **rule.**

What can M be? either a string of a's or a string of b's. We express this by adding:

$M \rightarrow A$
$M \rightarrow B$

Where A and B are new symbols that stand for strings of a's and b's respectively.
Now, let's define a string of a's.
First, it can be empty, so we add the rule:

$A \rightarrow \varepsilon$

or it may consist of a leading a followed by a string of a's'

$A \rightarrow aA.$

Similarly for B:

$B \rightarrow \varepsilon$ and
$B \rightarrow bB$

The language denoted by the regular expression $a(a^* \cup b^*)b$ can be defined by the following rules:

$S \rightarrow aMb$
$M \rightarrow A$
$M \rightarrow B$
$A \rightarrow \varepsilon$
$A \rightarrow aA.$
$B \rightarrow \varepsilon$
$B \rightarrow bB$

For example to generate the string *aaab*:

1. we start with the symbol S, as specified;

2. we replace S by aMb according to the first rule, S$\rightarrow$ aMb.
3. To aMb we apply the rule M$\rightarrow$A and obtain aAb.
4. We then twice apply the rule A$\rightarrow$ aA to get the string aaaAb.
5. Finally we apply the rule A$\rightarrow\varepsilon$
6. In the resulting string aaab we cannot identify any symbol that appears to the left of $\rightarrow$ in some rule, thus the operation of our generator has ended and aaab was produced.

A context-free grammar is a language generator that operates with a set of rules. Context-free grammars are a more powerful method of representing languages.

Such grammars can describe features that have a recursive structure.

In a context free grammar, some symbols appear to the left of $\rightarrow$ in rules -S, M, A, B- in our example,
and some do not -a and b- Symbols of the latter kind are called terminals, since the string is made up of these symbols. The other symbols are called non-terminal symbols.
The rule of the form X$\rightarrow$ Y is called a production rule or a substitution rule.

An important application of context-free grammars occurs in the specification and compilation of programming languages.

<u>Definition of Context-Free Grammars:</u>

Four important components in a grammatical description of a language:

1. There is a finite set of symbols that forms strings of the language being defined. We call this set the terminals or terminal symbols.

2. There is a finite set of variables, also called nonterminals or syntactic categories. Each variable represents a language; i.e. a set of strings/
3. One of the variables represents the language being defined; it is called the start symbol. Other variables represent auxiliary classes of strings that are used to help define the language of the start symbol.
4. There is a finite set of productions or rules that represent the recursive definition of a language. Each production consists of:
    a. A variable that is being (partially) defined by the production. This variable is often called the head of the production.
    b. The production symbol $\rightarrow$
    c. A string of zero or more terminals and variables. This string called the body of the production, represents one way to form strings in the language of the variable of the head. In so doing, we leave terminals unchanged and substitute for each variable of the body any string that is known to be in the language.

The four components just described form a context free grammar.

Formal Definition of a Context Free Grammar:

A context free grammar is a 4-tuple (V, Σ, R, S) where:

1. V is a finite set called the variables, or nonterminals
2. Σ is a finite set, disjoint from V, called the terminals
3. R is a finite set of rules, with each rule being a variable and a string of variables and terminals and
4. $S \in V$ is the start variable

If u, v and w are strings of variables and terminals, and

$A \rightarrow w$ is a rule of the grammar, we say that uAv yields uwv.

We say that u derives v, written $u \overset{*}{\Rightarrow} v$, if

1. u =v or
2. if a sequence $u_1, u_2, \ldots u_k$ exists for $k \geq 0$, such that $u \Rightarrow u_1 \Rightarrow u_2 \ldots.. \Rightarrow u_k \Rightarrow v$

Consider grammar G4= (V, ∑, R, <EXPR>)
V ={ <EXPR>, <TERM>, <FACTOR>} and
∑={a, +, x, (,)}.
The rules are:
<EXPR> → <EXPR> + <TERM> | <TERM>
<TERM>→ <TERM> x <FACTOR> | <FACTOR>
<FACTOR> → (<EXPR>) | a

The two strings a+a x a and (a+ a) x a can be generated with grammar G.

Parsing

A compiler translates code written in a programming language into another form, usually more suitable for execution. This process is called *parsing.*

Parse Trees

A parse tree for a grammar G is a tree where:

- the root is the start symbol for G
- the interior nodes are the nonterminals of G
- the leaf nodes are the terminal symbols of G.

- the children of a node T (from left to right) correspond to the symbols on the right hand side of some production for T in G.

Every terminal string generated by a grammar has a corresponding parse tree; every valid parse tree represents a string generated by the grammar (called the *yield* of the parse tree).

Example: Given the following grammar, find a parse tree for the string

1 + 2 * 3:

1. <E> --> <D>
2. <E> --> (<E>)
3. <E> --> <E> + <E>
4. <E> --> <E> - <E>
5. <E> --> <E> * <E>
6. <E> --> <E> / <E>
7. <D> --> 0 | 1 | 2 | ... 9

The parse tree is:

```
E --> E --> D --> 1
  +
   E --> E --> D --> 2
        *
        E --> D --> 3
```

Ambiguous Grammars

A grammar for which there are two different parse trees for the same terminal string is said to be *ambiguous*.

The grammar for balanced parentheses given earlier is an example of an ambiguous grammar:

```
P --> ( P ) | P P | epsilon
```

We can prove this grammar is ambiguous by demonstrating two parse trees for the same terminal string.

Here are two parse trees for the empty string:

```
1)  P -->   P --> epsilon
        P --> epsilon

 2) P --> epsilon
```

Here are two parse trees for ():

```
 P --> P --> (
             P --> epsilon
             )
       P --> epsilon

P --> P --> epsilon
       P --> (
   P --> epsilon
   )
```

To prove that a grammar is ambiguous, it is sufficient to demonstrate that the grammar generates two distinct parse trees for the same terminal string.

An unambiguous grammar for the same language (that is, the set of strings consisting of balanced parentheses) is:

```
        P --> ( P ) P | epsilon
```

The Problem of Ambiguous Grammars

A parse tree is supposed to display the structure used by a grammar to generate an input string. This structure is not unique if the grammar is ambiguous. A problem arises if we attempt to impart meaning to an input string using a parse tree; if the parse tree is not unique, then the string has multiple meanings.

We typically use a grammar to define the syntax of a programming language. The structure of the parse tree produced by the grammar imparts some meaning on the strings of the language.

If the grammar is ambiguous, the compiler has no way to determine which of two meanings to use. Thus, the code produced by the compiler is not fully determined by the program input to the compiler.

Ambiguous Precedence

The grammar for expressions given earlier:

1. <E> --> number
2. <E> --> (<E>)
3. <E> --> <E> + <E>
4. <E> --> <E> - <E>
5. <E> --> <E> * <E>

6. <E> --> <E> / <E>

This grammar is ambiguous as shown by the two parse trees for the input string: “number + number * number”:

```
 E --> E --> number
+
E --> E --> number
   *
   E --> number

 E --> E --> E --> number
   +
   E --> number
*
E --> number
```

The first parse tree gives precedence to multiplication over addition; the second parse tree gives precedence to addition over multiplication. In most programming languages, only the former meaning is correct. As written, this grammar is ambiguous with respect to the precedence of the arithmetic operators.

Ambiguous Associativity

Consider again the same grammar for expressions:

1. <E> --> number
2. <E> --> (<E>)
3. <E> --> <E> + <E>
4. <E> --> <E> - <E>
5. <E> --> <E> * <E>
6. <E> --> <E> / <E>

This grammar is ambiguous even if we only consider operators at the same precedence level, as in the input string “number - number + number”:

```
  E --> E --> number
-
E --> E --> number
   +
   E --> number

  E --> E --> E --> number
   -
   E --> number
+
E --> number
```

The first parse tree (incorrectly) gives precedence to the addition operator; the second parse tree gives precedence to the subtraction operator. Since we normally group operators left to right within a precedence level, only the latter interpretation is correct.

<u>An Unambiguous Grammar for Expressions</u>

It is possible to write a grammar for arithmetic expressions that

- is unambiguous
- enforces the precedence of * and / over + and -
- enforces left associativity

Here is one such grammar:

1. <E> --> <E> + <T> | <E> - <T> | <T>
2. <T> --> <T> * <F> | <T> / <F> | <F>
3. <F> --> (<E>) | number

If we attempt to build a parse tree for number - number + number, we see there is only one such tree:

```
 E --> E --> E --> T --> F --> number
   -
   T --> F --> number
+
T --> F→ number
```

This parse tree correctly represents left associativity by using recursion on the left. If we rewrote the grammar to use recursion on the right, we would represent right associativity:

1. <E> --> <T> + <E> | <T> - <E> | <T>
2. <T> --> <F> * <T> | <F> / <T> | <F>
3. <F> --> (<E>) | number

Our grammar also correctly represents precedence levels by introducing a new non-terminal symbol for each precedence level. According to our grammar, expressions consist of the sum or difference of terms (or a single terms), where a term consists of the product or division of factors (or a single factor), and a factor is a nested expression or a number.

Chomsky Normal Form:

When working with CFG it is often convenient to have them in simplified form. One of the simplest and most useful forms is called the Chomsky normal form. Chomsky normal form or CNF is useful when giving algorithms for working with CFG.

Definition:

A context free grammar is in Chomsky normal form if every rule is of the form:

$A \rightarrow BC$

$A \rightarrow a$

Where a is any terminal and A, B, and C are non terminals different from the start symbol. In addition, we permit the rule:

$S \rightarrow$ epsilon, where S is the start symbol.

Theorem:

Any context free language is generated by a context free grammar in Chomsky normal form.

Proof by construction:

1. Add a new start variable $S0 \rightarrow S$, where S was the original start nonterminal. This will guarantee that the new start variable does not occur on the right-hand side of a rule.
2. We take care of all epsilon rules. We remove ε-rule of the form $A \rightarrow \varepsilon$, where A is not the start symbol.
   This is how to do it:
   For each occurrence of an A on the right-hand side of a rule, we add a new rule with that occurrence deleted.
   For example: if $R \rightarrow uAv$ is a rule, we add the rule $R \rightarrow uv$. We do so for each occurrence of the non-terminal A .
   So the rule $R \rightarrow uAvAw$ causes us to generate:

$R \rightarrow uvAw$
$R \rightarrow uAvw$
$R \rightarrow uvw$
We repeat these steps for all rules of the form non-terminal $\rightarrow \varepsilon$

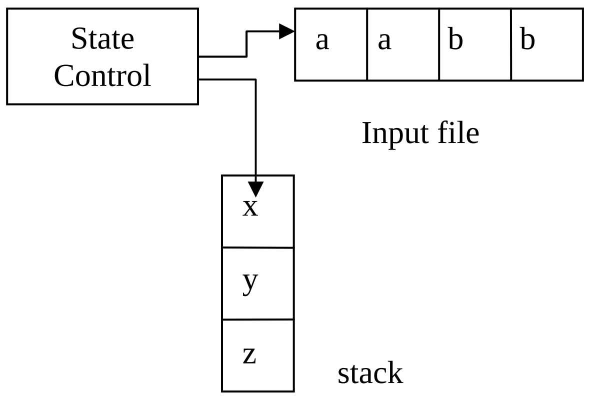
3. We handle unit rules, or rules of the form $A \rightarrow B$. We remove the preceding rule. Then, whenever a rule $B \rightarrow u$ appears, we add a rule $A \rightarrow u$. u is a string of variables and terminals. We repeat these steps until we eliminate all unit rules.

4. We convert all remaining rules into the proper form. We replace each rule of the form $A \rightarrow u_1u_2\ldots u_k$, where $k \geq 3$ and each u_i is a variable or terminal symbol, with the rules $A \rightarrow u_1A_1$, $A_1 \rightarrow u_2A_2, \ldots$, and $A_{k-2} \rightarrow u_{k-1}u_k$

Pushdown Automata:

The description of context-free languages by means of context free grammars is convenient. The next question is whether there is a class of automata that can be associated with context-free languages.
If such a machine exist, it requires an unbounded memory (a^nb^n), it requires a counting ability. It requires the ability to store and match a sequence of symbols. This gives us a class of machines called pushdown automata (pda)

A stack provides additional memory beyond the finite amount available. The stack allows pushdown automata to recognize some nonregular languages.



A pushdown automaton (PDA) can write symbol on the stack and read them back later.
-Writing a symbol "pushes down" all the other symbols on the stack. At any time, the symbol on the top can be read and removed. The remaining symbols then move back up.
-Writing a symbol on the stack is referred to as *pushing* the symbol,
-Removing the symbol is referred to as *popping* it.

A stack is a "last in, first out" storage device and read/write access can only be done at the top.

Formal definition of a pushdown automaton:

The formal definition of a pushdown automaton is similar to that of a finite automaton, except for the stack. The stack is a device containing symbols drawn from some alphabet. The machine may use different alphabets for its input and its stack. So we specify both an input alphabet $\sum$ and a stack alphabet Γ. The transition function δ which describes the automaton behavior is defined as $\delta \rightarrow Q \text{ x } \sum_\varepsilon \text{ x } \Gamma_\varepsilon \rightarrow P(Q \text{ x } \Gamma_\varepsilon)$

A pushdown automaton is a 6-tuple$(Q, \sum, \Gamma, \delta, q_0, F)$, where Q, $\sum$, Γ, F are all finite sets:

1. Q is the set of states.
2. Σ is the set of input symbols
3. Γ is the stack alphabet
4. δ: Q x Σ_ε x Γ_ε $\rightarrow$ *P*(Q x Γ_ε) is the transition function
5. $q_0 \in$ Q is the start state and
6. F $\subseteq$ Q is the set of accept states.

Computation:

A pushdown automaton M =(Q, Σ, Γ, δ, q_0, F) computes as follows:

It accepts input w if w can be written as w= $w_1w_2\ldots w_m$, where each $w_i \in \Sigma_\varepsilon$ and sequences of states $r_0, r_1, \ldots r_m \in$ Q and strings $s_0, s_1, \ldots, s_m \in \Gamma^*$ exist that satisfy the following 3 conditions: The strings s_i represent the sequence of stack contents that M has on the accepting branch of the computation:

1. $r_0=q_0$ and $s_0= \varepsilon$. This condition signifies that M starts on the start state and that the stack is empty.
2. for i=0,…., m-1, we have $(r_{i+1},b) \in \delta(r_i, w_{i+1},a)$, where s_i=at and s_{i+1} = bt for some a, b $\in \Gamma_\varepsilon$ and t $\in \Gamma^*$. This condition states that M moves properly according to the state stack, and next input symbol.
3. $r_m \in$F. This condition states that an accept state occurs at the input end.

The following is the formal description of the PDA that recognizes the language $\{0^n1^n \mid n \geq 0\}$.

Let M_1= (Q, Σ, Γ, δ, q1, F) where
Q={q1, q2, q3, q4},
Σ = {0, 1},
Γ = {0, \$},
F= {q1, q4}, and

δ is given by the following table, wherein blank entries signify
δ can also be defined as such:

$\delta(q1, 0, \varepsilon) = (q2, \$)$
$\delta(q2, 0, \$) = (q2, 0\$)$
$\delta(q2, 0, 0) = (q2, 00)$
$\delta(q2, 1, 0) = (q3, \varepsilon)$
$\delta(q3, 1, 0) = (q3, \varepsilon)$
$\delta(q3, \varepsilon, \$) = (q4, \varepsilon)$

Input	0			1			ε		
Stack	0	$	ε	0	$	ε	0	$	ε
q1									(q2,$)
q2	(q2,0)	(q2,0)		(q3,ε)					
q3				(q3,ε)				(q4,ε)	
q4									

Language accepted by a Pushdown Automaton:

Let M be a non-deterministic pushdown automaton, the language accepted by M is the set of all strings that can put M into a final state at the end of the string. The final stack content is irrelevant to this definition of acceptance

Construct a PDA that accepts the following language:
L1= L(aaa*b)

Pushdown Automata and Context-Free Languages:
Theorem:

A language is context free if and only if some pushdown automaton recognizes it.

Lemma:
If a language is context free, then some pushdown automaton recognizes it.

Greibach Normal Form:
A context free grammar is said to be in Greibach normal form if all productions have the form:
A→ ax where a is terminal symbol and x is one or more nonterminal symbols preceded by a terminal symbol:
S→ AB
A→ aA|bB|b
B→ b

Greibach Normal Form:

S→ aAB| bBB|bB
A→ aA|bB|b
B→ b
This concept is important for what follows:

Let A be a Context free language; then there is a CFG G that generates it.

<u>Procedure on how to convert G into an equivalent PDA, called P:</u>

1. Place the marker symbol $ and the start variable on the stack
2. Repeat the following steps forever:

    a. if the top of stack is a variable (non-terminal) symbol A, non deterministically select one of the rules for A and substitute A by the string on the right-hand side of the rule.

b. If the top of stack is a terminal symbol a, read the next symbol from the input and compare it to a. If they match, pop it repeat step 2. If they do not match, reject on this branch of the non-determinism.

c. if the top of stack is the symbol $, enter the accept state. Doing so accepts the input if it has all been read.

The corresponding automaton will have three states {q0, q1, q2}, with initial state q0 and final state q2

1. The start symbol S is put on the stack by
   $\delta(q0, \varepsilon, \varepsilon) = \{(q1, S\$)\}$
2. The production S→ aSA will be simulated in the pda by removing S from the stack and replacing it with SA, while reading a from the input.
3. The rule S→a should cause the pda to read a while removing S.

Thus the two productions are represented in the pda by:
$\delta(q1, a, S) = \{(q1, SA), (q1, \varepsilon)\}$
Rules 2 and 3

4. $\delta(q1, b, A) = \{(q1, B)\}$
5. $\delta(q1, b, B) = \{(q1, \varepsilon)\}$

Acceptance transition:

6. $\delta(q1, \varepsilon, \$) = \{(q2, \varepsilon)\}$

The underlying idea is to construct a pda that can carry out a leftmost derivation of any string in the language.

<u>Deterministic Pushdown Automata and Deterministic Context Free Languages:</u>

A deterministic pushdown accepter(dpda) is a pushdown automaton that never has a choice in its move

Definition:

A pushdown automaton $M = (Q, \Sigma, \Gamma, \delta, q0, z, F)$ is said to be deterministic if it is an automaton as described above with the added restriction that for every $q \in Q$ and $a \in \Sigma \cup \{\varepsilon\}$ and $b \in \Gamma$

1. $\delta(q, a, b)$ contains at most one element
2. $\delta(q, \varepsilon, b)$ is not empty, then $\delta(q, c, b)$ must be empty for every $c \in \Sigma$

The first of these conditions requires that for any given input symbol and any stack top, at most one move can be made.

The second condition is when an ε-move is possible for some configuration, no input-consuming alternative is available.

Note that we want to retain ε-moves; since the top of the stack plays a role in determining the next move, the presence of ε-move does not imply non determinism.

A language L is said to be a deterministic context free language if and only if there exists a dpda M such that $L = L(M)$.

The importance of deterministic context free languages lies in the fact that they can be parsed efficiently. If we view the pushdown automaton as a parsing device, there will be no backtracking, so we can easily write a computer program for it.

The pumping lemma is an effective tool for showing that certain languages are not regular. Similar pumping lemmas are used for other language families.

A pumping lemma for Context-Free Languages:

The pumping lemma for context free languages states that every context free language has a special value called the pumping length such that all longer strings in the language can be "pumped". This means that the string can be divided into five parts so that the second and the fourth parts may be repeated together.

Theorem:

If A is a context language, then there is a number p(the pumping length) where, if s is any string in A of length at least p, then s may be divided into five pieces s= uvxyz satisfying the conditions:

1. for each $i \geq 0$, $uv^ixy^iz \in A$
2. $|vy| > 0$ and
3. $|vxy| \leq p$

When s is being divided into uvxyz, condition 2 says that either v or y is not the empty string.

The pumping lemma is useful in showing that a language does not belong to the family of context-free languages.

Show that the language $L = \{a^nb^nc^n: n \geq 0\}$ is not context free.
Let's pick m, we pick the string $a^mb^mc^m$ which is in L.

If we choose vxy to contain only a’s, then the pumped string is $b^m c^m$ which is not in L.
If we choose a decomposition so that v and y are composed of an equal number of a’s and b’s then the pumped string $a^k b^k c^m$ with $k \neq m$ is not in the language.
So $L = \{a^n b^n c^n : n \geq 0\}$ is not context free.