What is Context-Free Grammar?

A **context-free grammar (CFG)** is a formal system used to describe a class of languages known as **context-free languages (CFLs)**. purpose of context-free grammar is:

- To list all strings in a language using a set of rules (production rules).
- It extends the capabilities of regular expressions and finite automata.

A grammar is said to be the Context-free grammar if every production is in the form of:

G -> (V∪T), where G ∈ V*

V (Variables/Non-terminals): *These are symbols that can be replaced using production rules. They help in defining the structure of the grammar. Typically, non-terminals are represented by uppercase letters (e.g., S, A, B).*

T (Terminals): *These are symbols that appear in the final strings of the language and cannot be replaced further. They are usually represented by lowercase letters (e.g., a, b, c) or specific symbols.*

The left-hand side can only be a Variable; it cannot be a terminal.

But on the right-hand side here it can be a Variable or Terminal or both combination of Variable and Terminal.

The above equation states that every production which contains any combination of the 'V' variable or 'T' terminal is said to be a context-free grammar.

Core Concepts of CFGs

A CFG is defined by:

1. **Nonterminal symbols (variables):** Represent abstract categories or placeholders (e.g., E,SE, SE,S).
2. **Terminal symbols (alphabet):** The actual characters or tokens in the language (e.g., a,b,+,∗,(,)a, b, +, *, (,)a,b,+,∗,(,)).
3. **Production rules:** Specify how non-terminals can be replaced with other non-terminals or terminals (e.g., E→E+EE → E + EE→E+E).
4. **Start symbol:** A special nonterminal from which derivations begin.

CFG vs. Other Models

Model	Description
Finite Automata	Accept strings via computation (accept/reject).
Regular Expressions	Match strings by describing their structure.
CFG	Generate strings via recursive replacement.

Example: Arithmetic Expressions
To describe arithmetic expressions with operators +,−,∗,/+, -, *, /+,−,∗,/, a CFG can be written as:

Production Rules:

E → int
E → E Op E
E → (E)
*Op → + | – | * | /*

Example Derivation:

E
⇒ E Op E
⇒ int Op E
*⇒ int * E*
*⇒ int * (E Op E)*
*⇒ int * (int + int)*
This derivation generates the string `int * (int + int)`.

Designing a CFG
When creating CFGs:
1. **Base case:** Define the simplest valid strings.
2. **Recursive rules:** Combine smaller components into larger ones.

Examples:
1. Palindromes over {a, b}:
S → ε | a | b | aSa | bSb
2. Balanced Parentheses:
S → ε | (S) | SS
Languages Defined by CFGs
The language L(G) generated by a CFG G is: $L(G)=\{\omega\in\Sigma*|S\Rightarrow*\omega\}L(G)=\{\omega\in\Sigma_*|S\Rightarrow_*\omega\}$

- **ω:** Strings made of terminals.
- **S⇒∗ω:** S derives ω via zero or more production applications.

Regular Languages vs. Context-Free Languages

Property	Regular Languages	Context-Free Languages
Power	Limited	More expressive

Property	Regular Languages	Context-Free Languages
Memory Requirements	Finite	Unbounded recursion
Definable Structures	Simple patterns (e.g., repetition)	Nested structures (e.g., palindromes, balanced parentheses)

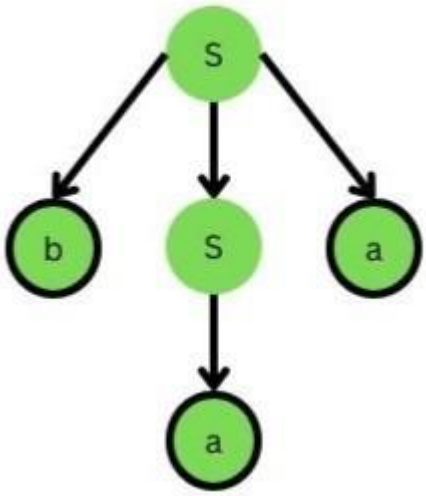
CFG for Regular Expressions

CFGs can model regular expressions:
1. Convert * (repetition):
$S \rightarrow Ab$
$A \rightarrow Aa \mid \varepsilon$
2. Convert a(b∪c∗):
$S \rightarrow aX$
$X \rightarrow b \mid C$
$C \rightarrow Cc \mid \varepsilon$

Example of CFG
For example, the grammar A = { S, a, b } having productions:
- Here S is the starting symbol.
- {a, b} are the terminals generally represented by small characters.
- S is the variable.

S-> aS
S-> bSa

Non-CFG Example
Productions such as:
a->bSa, or
a->ba is not a CFG as on the left-hand side there is a terminal which does not follow the CFGs rule.
Lets consider the string "**aba"** and and try to derive the given grammar from the productions given. we start with symbol **S,** apply production rule **S->bSa** and then S->aS **(S->a)** to get the string **"aba".**



Parse tree of string "aba"

In the computer science field, context-free grammars are frequently used, especially in the areas of formal language theory, compiler development, and natural language processing. It is also used for explaining the syntax of programming languages and other formal languages.

Additional Materials

In its most general form, a grammar is a set of rewriting rules. A rewriting rule specifies that a certain string of symbols can be substituted for all or part of another string. If ww and uu are strings, then w⟶uw⟶u is a rewriting rule that specifies that the string w can be replaced by the string u. The symbol "⟶"⟶ " is read "can be rewritten as." Rewriting rules are also called **production rules** or **productions**, and "⟶"⟶ can also be read as "produces." For example, if we consider strings over the alphabet {a, b, c}, then the production rule aba⟶ccaba⟶cc can be applied to the string abbabacto give the string abbccc. The substring aba in the string abbabac has been replaced with cc.

In a **context-free grammar**, every rewriting rule has the form A⟶A⟶w, where A is single symbol and w is a string of zero or more symbols. (The grammar is "context-free" in the sense that w can be substituted for A wher- ever A occurs in a string, regardless of the surrounding context in which Aoccurs.) The symbols that occur on the left-hand sides of production rules in a context-free grammar are called **non-terminal** symbols. By convention, the non-terminal symbols are usually uppercase letters. The strings on the right-hand sides of the production rules can include non-terminal symbols as well as other symbols, which are called **terminal symbols**. By convention, the terminal symbols are usually lowercase letters. Here are some typical production rules that might occur in context-free grammars:

$$A \longrightarrow aAbBA \longrightarrow aAbB$$
$$S \longrightarrow SSS \longrightarrow SS$$
$$C \longrightarrow AccC \longrightarrow Acc$$
$$B \longrightarrow bB \longrightarrow b$$
$$A \longrightarrow \varepsilon A \longrightarrow \varepsilon$$

In the last rule in this list, ε represents the empty string, as usual. For example, this rule could be applied to the string aBaAcA to produce the string aBacA. The first occurrence of the symbol A in aBaAcA has been replaced by the empty string—which is just another way of saying that the symbol has been dropped from the string.

In every context-free grammar, one of the non-terminal symbols is designated as the **start symbol** of the grammar. The start symbol is often, though not always, denoted by S. When the grammar is used to generate strings in a language, the idea is to start with a string consisting of nothing but the start symbol. Then a sequence of production rules is applied. Each application of a production rule to the string transforms the string to a new string. If and when this process produces a string that consists purely of terminal symbols, the process ends. That string of terminal symbols is one of the strings in the language generated by the grammar. In fact, the language consists precisely of all strings of terminal symbols that can be produced in this way.

As a simple example, consider a grammar that has three production rules: $S \longrightarrow aS, S \longrightarrow bS, S \longrightarrow aS, S \longrightarrow bS$, and $S \longrightarrow bS \longrightarrow b$. Inthisexample,Sistheonly non-terminal symbol, and the terminal symbols are a and b. Starting from the string S, we can apply any of the three rules of the grammar to produce either aS, bS, or b. Since the string b contains no non-terminals, we see that b is one of the strings in the language generated by this grammar. The strings aS and bS are not in that language, since they contain the non- terminal symbol S, but we can continue to apply production rules to these strings. From aS, for example, we can obtain aaS, abS, or ab. From abS, we go on to obtain abaS, abbS, or abb. The strings ab and abb are in the language generated by the grammar. It's not hard to see that any string of a's and b's that ends with a b can be generated by this grammar, and that these are the only strings that can be generated. That is, the language generated by this grammar is the regular language specified by the regular expression $(a|b)*b(a|b)*b$. It's time to give some formal definitions of the concepts which we have been discussing.

Definition 4.1.

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

1. V is a finite set of symbols. The elements of V are the non-terminal symbols of the grammar.

2.Σ is a finite set of symbols such that $V \cap \Sigma = \emptyset V \cap \Sigma = \emptyset$. The elements of Σ are the terminal symbols of the grammar.

3. P is a set of production rules. Each rule is of the form $A \longrightarrow wA \longrightarrow w$ where A is one of the symbols in V and w is a string in the language $(V \cup \Sigma)*(V \cup \Sigma)*$.

4. $S \in V.SS \in V.S$ is the start symbol of the grammar.

Even though this is the formal definition, grammars are often specified informally simply by listing the set of production rules. When this is done it is assumed, unless otherwise specified, that the non-terminal symbols are just the symbols that occur on the left-hand sides of production rules of the grammar. The terminal symbols are all the other symbols that occur on the right-hand sides of production rules. The start symbol is the symbol that occurs on the left-hand side of the first production rule in the list.Thus, the list of production rules

$$T \longrightarrow TTT \longrightarrow TT$$
$$T \longrightarrow AT \longrightarrow A$$
$$A \longrightarrow aAaA \longrightarrow aAa$$
$$A \longrightarrow bBA \longrightarrow bB$$
$$B \longrightarrow bBB \longrightarrow bB$$
$$B \longrightarrow \varepsilon B \longrightarrow \varepsilon$$

specifies a
grammar $G=(V,\Sigma,P,T)G=(V,\Sigma,P,T)$ where VV is $\{T,A,B\},\Sigma\{T,A,B\},\Sigma$ is $\{a,b\},\{a, b\}$, and T is the start symbol. P, of course, is a set containing the six production rules in the list.

Let $G=(V,\Sigma,P,S)G=(V,\Sigma,P,S)$ be a context-free grammar. Suppose
that xx and yy are strings in the language $(V \cup \Sigma)*.(V \cup \Sigma)*$. The
notation $x \Longrightarrow_G yx \Longrightarrow Gy$ is used to express the fact that y can be obtained from x by applying one of the production rules in P . To be more exact, we say
that $x \Longrightarrow Gx \Longrightarrow G$ y if and only if there is a production rule $A \longrightarrow wA \longrightarrow w$ in the grammar and two strings uand v in the language $(V \cup \Sigma)*(V \cup \Sigma)*$ such that x = uAv and y = uwv. The fact that x = uAv is just a way of saying that A occurs somewhere in x. When the production rule \(A \ is applied to substitute w for A in uAv, the result is uwv, which is y. Note that either u or v or both can be the empty string.

If a string y can be obtained from a string x by applying a sequence of zero or more production rules, we write $x \Longrightarrow *_G x \Longrightarrow G*$ y. In most cases, the "G" in the notations $\Longrightarrow G \Longrightarrow G$ and $\Longrightarrow *_G \Longrightarrow G*$ will be omitted, assuming that the grammar in question is understood. Note that $\Longrightarrow \Longrightarrow$ is a relation on the set $(V \cup \Sigma)*(V \cup \Sigma)*$. The relation $\Longrightarrow * \Longrightarrow *$ is the reflexive, transitive closure of that relation. (This explains the use of "$*$", which is usually used to denote the transitive, but not necessarily reflexive, closure of a relation. In this case, $\Longrightarrow * \Longrightarrow *$ is reflexive as well as transitive since $x = \Rightarrow * x$ is true for any string x.) For example, using the grammar that is defined by the above list of production rules, we have

$$aTB \Longrightarrow aTTB \Longrightarrow aTAB \Longrightarrow aTAbB \Longrightarrow aTbBbB \Longrightarrow aTbbBaTB \Longrightarrow aTTB \Longrightarrow aTAB \Longrightarrow aTAbB \Longrightarrow aTbBbB \Longrightarrow aTbbB$$

From this, it follows that $aTB \Longrightarrow_* aTbbB$.$aTB \Longrightarrow_* aTbbB$. The relation $\Longrightarrow\Longrightarrow$ is read "yields" or "produces" while $\Longrightarrow\Longrightarrow$ can be read "yields in zero or more steps" or "produces in zero or more steps." The following theorem states some simple facts about the relations $\Longrightarrow\Longrightarrow$ and $\Longrightarrow_*:\Longrightarrow_*$:

Theorem 4.1.

Let GG *be the context-free grammar* (V,Σ,P,S).(V,Σ,P,S). *Then:*

1. If xx *and* yy *are strings in* $(V \cup \Sigma)_*(V \cup \Sigma)_*$ *such that* $x \Longrightarrow y, x \Longrightarrow y$, *then* $x \Longrightarrow_* yx \Longrightarrow_* y$

2. If x,y,x,y, *and* zz *are strings in* $(V \cup \Sigma)_*(V \cup \Sigma)_*$ *such that* $x \Longrightarrow_* yx \Longrightarrow_* y$ *and* $y \Longrightarrow_* zy \Longrightarrow_* z$ *then* $x \Longrightarrow_* zx \Longrightarrow_* z$

3. If xx *and* yy *are strings in* $(V \cup \Sigma)_*(V \cup \Sigma)_*$ *such that* $x \Longrightarrow y, x \Longrightarrow y$, *and if* ss *and* tt *are any strings in* $(V \cup \Sigma)_*, (V \cup \Sigma)_*$, *then* $sxt \Longrightarrow sytsxt \Longrightarrow syt$

4. If xx *and* yy *are strings in* $(V \cup \Sigma)_*(V \cup \Sigma)_*$ *such that* $x \Longrightarrow_* y, x \Longrightarrow_* y$, *and if* ss *and* tt *are any strings in* $(V \cup \Sigma)_*, (V \cup \Sigma)_*$, *then sxt* $\Longrightarrow_* syt \Longrightarrow_* syt$

Proof. Parts 1 and 2 follow from the fact that $\Longrightarrow_* \Longrightarrow_*$ is the transitive closure of $\Longrightarrow\Longrightarrow$ Part 4 follows easily from Part 3.3. (I leave this as an exercise.) To prove Part 3, suppose that x, y, s, and t are strings such that $x \Longrightarrow yx \Longrightarrow y$. By definition, this means that there exist strings u and v and a production rule $A \longrightarrow wA \longrightarrow w$ such that $x = uAvx = uAv$ and $y = uwv.y = uwv$. But then we also have $sxt = sxt =$ suAvt and syt = suwvt. These two equations, along with the existence of the production rule $A \longrightarrow wA \longrightarrow w$ show, by definition, that $sxt \Longrightarrow sytsxt \Longrightarrow syt$.

We can use $\Longrightarrow_* \Longrightarrow_*$ to give a formal definition of the language generated by a context-free grammar:

Definition 4.2.

Suppose that $G=(V,\Sigma,P,S)G=(V,\Sigma,P,S)$ is a context-free grammar.

Then the language generated by G is the language L(G) over the alphabet Σ defined by

$$L(G)=\{w \in \Sigma_* | S \Longrightarrow_{*G} w\} L(G)=\{w \in \Sigma_* | S \Longrightarrow_G * w\}$$

That is, L(G) contains any string of terminal symbols that can be obtained by starting with the string consisting of the start symbol, S, and applying a sequence of production rules.

A language L is said to be a **context-free language** if there is a context-free grammar G such that L(G) is L. Note that there might be many different context-free grammars that generate the same context-free language. Two context-free grammars that generate the same language are said to be **equivalent**.

Suppose G is a context-free grammar with start symbol S and suppose $w \in L(G)$w∈L(G). By definition, this means that there is a sequence of one or more applications of production rules which produces the string w from S. This sequence has the form $S \Longrightarrow x_1 \Longrightarrow x_2 \Longrightarrow \cdots \Longrightarrow w$S⟹x1⟹x2⟹⋯⟹w. Such a sequence is called a **derivation** of w (in the grammar G). Note that w might have more than one derivation. That is, it might be possible to produce w in several different ways.

Consider the language $L = \{a^n b^n | n \in N\}$.L={anbn|n∈N}. We already know that LL is not a regular language. However, it is a context-free language. That is, there is a context-free grammar such that L is the language generated byG. This gives us our first theorem about grammars:

Theorem 4.2.

Let LL *be the language* $L = \{a^n b^n | n \in N\}$.L={anbn|n∈N}. *Let* GG *be the context-free grammar* (V, Σ, P, S)(V,Σ,P,S) *where* $V = \{S\}, \Sigma = \{a, b\}$V={S},Σ={a,b} *and* PP *consists of the productions*

$$S \longrightarrow aSbS \longrightarrow aSb$$
$$S \longrightarrow \varepsilon S \longrightarrow \varepsilon$$

Then L = L(G), so that L is a context-free language. In particular, there exist context-free languages which are not regular.

Proof. To show that $L = L(G)$,L=L(G), we must show both that $L \subseteq L(G)$L⊆L(G) and that $L(G) \subseteq L$.L(G)⊆L. To show that $L \subseteq L(G)$,L⊆L(G), let ww be an arbitrary element of L.L. By definition of $L, w = a^n b^n$L,w=anbn for some $n \in N$.n∈N. We show that In the case where n=0,we have w=ε. Now, ε ∈ L(G) since ε can be produced from the start symbol S by an application of the rule $S \longrightarrow \varepsilon$,S⟶ε, so our claim is true for $n = 0$n=0. Now, suppose that $k \in N$k∈N and that we already know that $a^k b^k \in L(G)$akbk∈L(G). We must show that $a^{k+1} b^{k+1} \in L(G)$.ak+1bk+1∈L(G). since $S \Longrightarrow^* a^k b^k S \Longrightarrow^* a^k b^k$S⟹∗akbkS⟹∗akbk, we also have, by Theorem 4.1, that $aSb \Longrightarrow^* aa^k b^k baSb \Longrightarrow^* aa^k b^k b$aSb⟹∗aakbkbaSb⟹∗aakbkb. That is, $aSb \Longrightarrow^* a^{k+1} b^{k+1} aSb \Longrightarrow^* a^{k+1} b^{k+1}$aSb⟹∗ak+1bk+1aSb⟹∗ak+1bk+1. Combining this with the production rule $S \longrightarrow aSb$,S⟶aSb, we see that $S \Longrightarrow^* a^{k+1} b^{k+1} S \Longrightarrow^* a^{k+1} b^{k+1}$S⟹∗ak+1bk+1S⟹∗ak+1bk+1. This means

that $a_{k+1}b_{k+1} \in L(G)$ak+1bk+1∈L(G)ak+1bk+1∈L(G), as we wanted to show. This completes the proof that $L \subseteq L(G)$L⊆L(G)L⊆L(G).

To show that $L(G) \subseteq L$,L(G)⊆L(G)⊆L, suppose that $w \in L(G)$.w∈L(G)w∈L(G). That is, $S \Rightarrow^* w$.S⇒∗wS⇒∗w. We must show that $w = a_n b_n$w=anbnw=anbn for some n.n. since $S \Rightarrow^* w$,S⇒∗wS⇒∗w, there is a derivation $S \Rightarrow x_0 \Rightarrow x_1 \Rightarrow \cdots \Rightarrow x_n$,S⇒x0⇒x1⇒⋯⇒xnS⇒x0⇒x1⇒⋯⇒xn, where $w = x_n$w=xnw=xn. We first prove by induction on n that in any derivation $S \Rightarrow x_0 \Rightarrow x_1 \Rightarrow \cdots \Rightarrow x_n$S⇒x0⇒x1⇒⋯⇒xnS⇒x0⇒x1⇒⋯⇒xnS⇒x0⇒x1⇒⋯⇒xn, we must have either $x_n = a_n b_n$xn=anbnxn=anbn or $x_n = a_{n+1} S b_{n+1}$xn=an+1Sbn+1xn=an+1Sbn+1. Consider the case n = 0. Suppose $S \Rightarrow x_0$S⇒x0S⇒x0. Then, we must have that $S \rightarrow x_0$S→x0S→x0 is a rule in the grammar, so x_0x0x0 must be either εεε or aSb.aSb. since $\varepsilon = a_0 b_0$ε=a0b0ε=a0b0 and $aSb = a_{0+1} S b_{0+1}$,x0aSb=a0+1Sb0+1aSb=a0+1Sb0+1,x0 is of the required form. Next, consider the inductive case. Suppose that k > 1 and we already know that in any derivation $S \Rightarrow x_0 \Rightarrow x_1 \Rightarrow \cdots \Rightarrow x_k$S⇒x0⇒x1⇒⋯⇒xkS⇒x0⇒x1⇒⋯⇒xkS⇒x0⇒x1⇒⋯⇒xk, we must have $x_k = a_k b_k$xk=akbkxk=akbk or $x = a_{k+1} S b_{k+1}$x=ak+1Sbk+1x=ak+1Sbk+1. Suppose that $S \Rightarrow x_0 \Rightarrow S \Rightarrow x_0 \Rightarrow x_1 \Rightarrow \cdots \Rightarrow x_k \Rightarrow x_{k+1} x_1 \Rightarrow \cdots \Rightarrow x_k \Rightarrow x_{k+1}$S⇒x0⇒S⇒x0⇒ x1⇒⋯⇒xk⇒xk+1x1⇒⋯⇒xk⇒xk+1. We know by induction that $x_k = a_k b_k$xk=akbkxk=akbk or $x = a_{k+1} S b_{k+1}$,x=ak+1Sbk+1,x=ak+1Sbk+1, but since $x_k \Rightarrow x_{k+1} x_k \Rightarrow x_{k+1}$ and $a_k b_k$akbkakbk contains no non-terminal symbols, we must have $x_k = a_{k+1} S b_{k+1}$xk=ak+1Sbk+1xk=ak+1Sbk+1. Since $x_{k+1} x_{k+1}$ is obtained by applying one of the production rules $S \rightarrow \varepsilon$S→εS→ε or $S \rightarrow aSbS \rightarrow aSb$ to x_k,xk+1xk,xk+1 is either $a_{k+1} \varepsilon b_{k+1}$ak+1εbk+1ak+1εbk+1 or $a_{k+1} aSbb_{k+1}$.ak+1aSbbk+1ak+1aSbbk+1. That is, $x_{k+1} x_{k+1}$ is either $a_{k+1} b_{k+2} S b_{k+2}$ak+1bk+2Sbk+2ak+1bk+2Sbk+2, as we wanted to show. This completes the induction. Turning back to w, we see that ww must be of the form $a_n b_n$anbnanbn or of the form $a_n S b_n$anSbnanSbn. But since $w \in L(G)$w∈L(G)w∈L(G), it can contain no non-terminal symbols, so w must be of the form $a_n b_n$,anbn,anbn, as we wanted to show. This completes the proof that $L(G) \subseteq L$L(G)⊆LL(G)⊆L.

I have given a very formal and detailed proof of this theorem, to show how it can be done and to show how induction plays a role in many proofs about grammars. However, a more informal proof of the theorem would probably be acceptable and might even be more convincing. To show that $L \subseteq L(G)$,L⊆L(G)L⊆L(G), we could just note that the derivation $S \Rightarrow aSb \Rightarrow a_2 S b_2 \Rightarrow S \Rightarrow aSb \Rightarrow a2Sb2 \Rightarrow \cdots \Rightarrow a_n S b_n \Rightarrow a_n b_n \cdots \Rightarrow anSbn \Rightarrow anbn$S⇒aSb⇒a2Sb2⇒S⇒aSb⇒a2Sb2⇒ ⋯⇒anSbn⇒anbn⋯⇒anSbn⇒anbn demonstrates that $a_n b_n \in L$.anbn∈L. On the other hand, it is clear that every derivation for this grammar must be of this form, so every string in $L(G)$L(G)L(G) is of the form $a_n b_n$anbnanbn.

For another example, consider the language $\{a_n b_m | n \geq m \geq 0\}$\{anbm|n≥m≥0\}{anbm|n≥m≥0}. Let's try to design a grammar that generates this language. This is similar to the previous example, but now we want to include strings that contain more a's than b's. The production rule $S \rightarrow aSbS \rightarrow aSb$S→aSbS→aSbS→aSb always produces the same number of a's and b's. Can we modify this idea to produce more a's than b's? One approach would be to

produce a string containing just as manya's as b's, and then to add some extra a's. A rule that can generate any number of a's is $A \longrightarrow aA$.A⟶aA. After applying the rule $S \longrightarrow aSbS$⟶aSb for a while, we want to move to a new state in which we apply the rule $A \longrightarrow aAA$⟶aA. We can get to the new state by applying a rule $S \longrightarrow AS$⟶A that changes the Sinto an A. We still need a way to finish the process, which means getting rid of all non-terminal symbols in the string. For this, we can use the rule $A \longrightarrow \varepsilon$.A⟶ε. Putting these rules together, we get the grammar.

$$S \longrightarrow aSbS \longrightarrow aSb$$
$$S \longrightarrow AS \longrightarrow A$$
$$A \longrightarrow aAA \longrightarrow aA$$
$$A \longrightarrow \varepsilon A \longrightarrow \varepsilon$$

This grammar does indeed generate the language $\{a^n b^m | n \geq m \geq 0\}${anbm|n≥m≥0} With slight variations on this grammar, we can produce other related lan- guages. For example, if we replace the rule $A \longrightarrow \varepsilon A \longrightarrow \varepsilon$ with $A \longrightarrow aA \longrightarrow a$, we get the language $\{a^n b^m | n > m \geq 0\}${anbm|n>m≥0}.

There are other ways to generate the language $\{a^n b^m | n \geq m \geq 0\}${anbm|n≥m≥0}. For example, the extra non-terminal symbol, A, is not really necessary, if we allow S to sometimes produce a single a without a b. This leads to the grammar.

$$S \longrightarrow aSbS \longrightarrow aSb$$
$$S \longrightarrow aSS \longrightarrow aS$$
$$S \longrightarrow \varepsilon S \longrightarrow \varepsilon$$

(But note that the rule $S \longrightarrow SaS \longrightarrow Sa$ would not work in place of $S \longrightarrow aSS \longrightarrow aS$ since it would allow the production of strings in which an a can follow a b, and there are no such strings in the language $\{a^n b^m | n \geq m \geq 0\}$.){anbm|n≥m≥0}.) And here are two more grammars that generate this language:

$$S \longrightarrow ABS \longrightarrow ASbS \longrightarrow ABS \longrightarrow ASb$$

$$A \longrightarrow aAA \longrightarrow aAA \longrightarrow aAA \longrightarrow aA$$

$$B \longrightarrow aBbS \longrightarrow \varepsilon B \longrightarrow aBbS \longrightarrow \varepsilon$$

$$A \longrightarrow \varepsilon A \longrightarrow \varepsilon A \longrightarrow \varepsilon A \longrightarrow \varepsilon$$

$$B \longrightarrow \varepsilon B \longrightarrow \varepsilon$$

Consider another variation on the language $\{a^n b^n | n \in N\}${anbn|n∈N}, in which thea's and b's can occur in any order, but the number of a's is still equal to the number of b's. This language can be defined

as $L=\{w\in\{a,b\}*|n_a(w)=L=\{w\in\{a,b\}*|na(w)=n_b(w)\}nb(w)\}$. This language includes strings such as abbaab, baab, and bbbaaa.

Let's start with the grammar containing the rules $S\longrightarrow aSbS\longrightarrow aSb$ and $S\longrightarrow S\longrightarrow \varepsilon.\varepsilon$. We can try adding the rule $S\longrightarrow bSaS\longrightarrow bSa$. Every string that can be generated using these three rules is in the language L. However, not every string in L can be generated. A derivation that starts with $S\Longrightarrow aSbS\Longrightarrow aSb$ can only produce strings that begin with a and end with b. A derivation that starts with $S\Longrightarrow bSaS\Longrightarrow bSa$ can only generate strings that begin with b and end with a. There is no way to generate the strings baab or abbbabaaba, which are in the language L. But we shall see that any string in L that begins and ends with the same letter can be written in the form xy where x and yare shorter strings in L. To produce strings of this form, we need one more rule, $S\longrightarrow SSS\longrightarrow SS$. The complete set of production rules for the language L is

$$S\longrightarrow aSbS\longrightarrow aSb$$
$$S\longrightarrow bSaS\longrightarrow bSa$$
$$S\longrightarrow SSS\longrightarrow SS$$
$$S\longrightarrow \varepsilon S\longrightarrow \varepsilon$$

It's easy to see that every string that can be generated using these rules is in L, since each rule introduces the same number of a's as b's. But we also need to check that every string w in L can be generated by these rules.

This can be done by induction on the length of w, using the second form of the principle of mathematical induction. In the base case, $|w|=0|w|=0$ and $w=\varepsilon.w=\varepsilon$. In this case, $w\in Lw\in L$ since $S\Longrightarrow \varepsilon S\Longrightarrow \varepsilon$ in one step. Suppose $|w|=k|w|=k$, where $k>0,k>0$, and suppose that we already know that for any $x\in Lx\in L$ with $|x|<k,S\Longrightarrow *x.|x|<k,S\Longrightarrow *x$. To finish the induction we must show, based on this induction hypothesis, that $S\Longrightarrow *wS\Longrightarrow *w$.

Suppose that the first and last characters of w are different. Then w is either of the form axb or of the form bxa, for some string x. Let's assume that w is of the form axb. (The case where w is of the form bxa is handled in a similar way.) Since w has the same number of a's and b's and sincex has one fewer a than w and one fewer b than w, x must also have the same number of a's as b's.That is $x\in L.x\in L$. But $|x|=|w|-2<k,|x|=|w|-2<k$, so by the induction hypothesis, $x\in L(G).x\in L(G)$. So we have $S\Longrightarrow *xS\Longrightarrow *x$. By Theorem 4.1, we get then $aSb\Longrightarrow *axb.aSb\Longrightarrow *axb$. Combining this with the fact that $S\Longrightarrow aSbS\Longrightarrow aSb$, we get that $S\Longrightarrow *axb,S\Longrightarrow *axb$, that is ,$S\Longrightarrow *w.,S\Longrightarrow *w$. This proves that $w\in L(G)w\in L(G)$.

Finally, suppose that the first and last characters of w are the same. Let's say that w begins and ends with a. (The case where w begins and ends with b is handled in a similar way.) I claim that w can be written in the form xy

where $x \in L(G)$x∈L(G) and $y \in L(G)$y∈L(G) and neither x nor y is the empty string. This will finish the induction, since we will then have by the induction hypothesis that $S \Longrightarrow * xS \Longrightarrow * x$ and $S \Longrightarrow * yS \Longrightarrow * y$, and we can derive xy fromS by first applying the rule $S \longrightarrow SSS \longrightarrow SS$ and then using the first SS on the right-hand side to derive x and the second to derive y.

It only remains to figure out how to divide w into two strings x and ywhich are both in L(G). The technique that is used is one that is more generally useful. Suppose that $w = c_1 c_2 \cdots c_k$,w=c1c2⋯ck, where each $c_i$ci is either aa or b.b. Consider the sequence of integers $r_1, r_2, \ldots, r_k$r1,r2,…,rk where for each $i = 1, 2, \ldots, k$,ri i=1,2,…,k,ri is the number of $a's in c_1 c_2 \cdots c_i$a′sinc1c2⋯cia′sin f_0c1c2⋯ci minus the number of $b's$b′s in $c_1 c_2 \cdots c_i$.c1c2⋯ci. since $c_1 = a, r_1 = 1$.c1=a,r1=1. since $w \in L, r_k = 0$.w∈L,rk =0. And since $c_k = a$ck=a, we must have $r_{k-1} = r_k - 1 = -1$.rk−1=rk−1=−1. Furthermore the difference between r_{i+1}ri+1 and $r_i$ri is either 1 or -1,−1, for $i = 1, 2, \ldots, k-1$i=1,2,…,k−1.

Since $r_1 = 1$r1=1 and $r_{k-1} = -1$rk−1=−1 and the value of $r_i$ri goes up or down by 1 when ii increases by $1, r_i$1,ri must be zero for some ii between 1 and $k-1$k−1. That is, $r_i$ri cannot get from 1 to -1−1 unless it passes through zero. Let ii be a number between 1 and $k-1$k−1 such that $r_i = 0$.ri=0. Let $x = c_1 c_2 \cdots c_i$x=c1c2⋯ci and let $y = c_{i+1} c_{i+2} \cdots c_k$.y=ci+1ci+2⋯ck. Note that $xy = w$.xy=w. The fact that $r_i = 0$ri=0 means that the string $c_1 c_2 \cdots c_i$c1c2⋯ci has the same number of $a's$a′s and $b's$b′s, so $x \in L(G)$.x∈L(G). It follows automatically that $y \in L(G)$y∈L(G) also. Since ii is strictly between 1 and k − 1, neither x nor y is the empty string. This is all that we needed to show to finish the proof that L = L(G).

The basic idea of this proof is that if w contains the same number of a's as b's, then an a at the beginning of w must have a "matching" b somewhere in w. This b matches the a in the sense that the corresponding ri is zero, and the b marks the end of a string x which contains the same number ofa's as b's. For example, in the string aababbabba, the a at the beginning of the string is matched by the third b, since aababb is the shortest prefix ofaababbabba that has an equal number of a's and b's.

Closely related to this idea of matching a's and b's is the idea of **balanced parentheses**. Consider a string made up of parentheses, such as(()(()))(()). The parentheses in this sample string are balanced because each left parenthesis has a matching right parenthesis, and the matching pairs are properly nested. A careful definition uses the sort of integer se- quence introduced in the above proof. Let w be a string of parentheses.Write $w = c_1 c_2 \cdots c_n$,w=c1c2⋯cn, where each $c_i$ci is either (or).(or). Define a sequence of integers $r_1, r_2, \ldots, r_n$,r1,r2,…,rn, where $r_i$ri is the number of left parentheses in $c_1 c_2 \cdots c_i$c1c2⋯ci minus the number of right parentheses. We say that the parentheses in w are balanced

if $r_n=0$rn=0 and $r_i \geq 0$ri≥0 for all $i=1,2,\ldots,n$i=1,2,…,n. The fact that $r_n=0$rn=0 says that ww contains the same number of left parentheses as right parentheses. The fact the $r_i \geq 0$ri≥0 means that the nesting of pairs of parentheses is correct: You can't have a right parenthesis unless it is balanced by a left parenthesis in the preceding part of the string. The language that consists of all balanced strings of parentheses is context-free. It is generated by the grammar

$$S \longrightarrow (S)S \longrightarrow (S)$$
$$S \longrightarrow SSS \longrightarrow SS$$
$$S \longrightarrow \varepsilon S \longrightarrow \varepsilon$$

The proof is similar to the preceding proof about strings of a's and b's. (It might seem that I've made an awfully big fuss about matching and balancing. The reason is that this is one of the few things that we can do with context-free languages that we can't do with regular languages.)

Before leaving this section, we should look at a few more general results. Since we know that most operations on regular languages produce languages that are also regular, we can ask whether a similar result holds for context-free languages. We will see later that the intersection of two context-free languages is not necessarily context-free. Also, the comple- ment of a context-free language is not necessarily context-free. However, some other operations on context-free languages do produce context-free languages.

Theorem 4.3.

Suppose that L and M are context-free languages. Then the languages $L \cup M, LM, L \cup M$L∪M,LM,L∪M, LM, *and* $L^* L^*$ *are also context-free.*

Proof. I will prove only the first claim of the theorem, that $L \cup M L \cup M$ is context-free. In the exercises for this section, you are asked to construct grammars for LM and $L^* L^*$ (without giving formal proofs that your answers are correct).

Let $G=(V,\Sigma,P,S)$G=(V,Σ,P,S) and $H=(W,\Gamma,Q,T)$H=(W,Γ,Q,T) be context-free grammars such that $L=L(G)$L=L(G) and $M=L(H)$.M=L(H). We can assume that $W \cap V=\emptyset$W∩V=∅, since otherwise we could simply rename the non-terminal symbols in W. The idea of the proof is that to generate a string in $L \cup M L \cup M$, we first decide whether we want a string in L or a string in M . Once that decision is made, to make a string in L, we use production rules from G, while to make a string in M, we use rules from H. We have to design a grammar, K, to represent this process.

Let RR be a symbol that is not in any of the alphabets $V,W,\Sigma,$V,W,Σ, or Γ.Γ. R will be the start symbol of K. The production rules for K consist of all the production rules from G and H together with two new rules:

$$R \longrightarrow SR \longrightarrow S$$
$$R \longrightarrow TR \longrightarrow T$$

Formally, K is defined to be the grammar

$$(V \cup W \cup \{R\}, P \cup Q \cup \{R \longrightarrow S, R \longrightarrow T\}, \Sigma \cup \Gamma, R)(V \cup W \cup \{R\}, P \cup Q \cup \{R \longrightarrow S, R \longrightarrow T\}, \Sigma \cup \Gamma, R)$$

Suppose that $w \in L.w \in L$. That is $w \in L(G), w \in L(G)$, so there is a derivation $S \Longrightarrow_{*G} wS \Longrightarrow G * w$. since every rule from GG is also a rule in K, K, if follows that $S \Longrightarrow_{*K} wS \Longrightarrow K * w$. Combining this with the fact that $R \Longrightarrow_{K} S, R \Longrightarrow KS$, we have that $R \Longrightarrow_{*K} w, R \Longrightarrow K * w$, and $w \in L(K).w \in L(K)$. This shows that $L \subseteq L(K).L \subseteq L(K)$. In an exactly similar way, we can show that $M \subseteq L(K).M \subseteq L(K)$. Thus, $L \cup M \subseteq L(K) L \cup M \subseteq L(K)$.

It remains to show that $L(K) \subseteq L \cup M.L(K) \subseteq L \cup M$. Suppose $w \in L(K).w \in L(K)$. Then there is a derivation $R \Longrightarrow_{*K} w.R \Longrightarrow K * w$. This derivation must begin with an application of one of the rules $R \longrightarrow SR \longrightarrow S$ or $R \longrightarrow T, R \longrightarrow T$, since these are the only rules in which RR appears. If the first rule applied in the derivation is $R \longrightarrow SR \longrightarrow S$, then the remainder of the derivation shows that $S \Longrightarrow_{*K} w.S \Longrightarrow K * w$. Starting from S, the only rules that can be applied are rules from G, so in fact we have $S \Longrightarrow_{*G} w.S \Longrightarrow G * w$. This shows that $w \in L.w \in L$. Similarly, if the first rule applied in the derivation $R \Longrightarrow_{*K} wR \Longrightarrow K * w$ is $R \longrightarrow T, R \longrightarrow T$, then $w \in M.w \in M$. In any case, $w \in L \cup Mw \in L \cup M$. This proves that $L(K) \subseteq L \cup ML(K) \subseteq L \cup M$.

Finally, we should clarify the relationship between context-free lan- guages and regular languages. We have already seen that there are context- free languages which are not regular. On the other hand, it turns out that every regular language is context-free. That is, given any regular language, there is a context-free grammar that generates that language. This means that any syntax that can be expressed by a regular expression, by a DFA, or by an NFA could also be expressed by a context-free grammar. In fact, we only need a certain restricted type of context-free grammar to duplicate the power of regular expressions.

Definition 4.3.

A right-regular grammar is a **context-free grammar** in which the right-hand side of every production rule has one of the following forms: the empty string; a string consisting of a single non-terminal symbol; or a string consisting of a single terminal symbol followed by a single non- terminal symbol.

Examples of the types of production rule that are allowed in a right- regular grammar are $A \longrightarrow \varepsilon, B \longrightarrow C, A \longrightarrow \varepsilon, B \longrightarrow C$, and $D \longrightarrow aED \longrightarrow aE$. Theideaof the proof is that given a right-regular grammar, we can build a corresponding NFA and vice-versa. The states of the NFA correspond to the non-terminal symbols of the

grammar. The start symbol of the grammar corresponds to the starting state of the NFA. A production rule of the form $A \longrightarrow bC$ corresponds to a transition in the NFA from state A to state C while reading the symbol b. A production rule of the form $A \longrightarrow B$ corresponds to an ε-transition from state A to state B in the NFA. And a production rule of the form $A \longrightarrow \varepsilon$ exists in the grammar if and only if A is a final state in the NFA. With this correspondence, a derivation of a string w in the grammar corresponds to an execution path through the NFA as it accepts the string w. I won't give a complete proof here. You are welcome to work through the details if you want. But the important fact is:

Theorem 4.4.

A *language* L is *regular* if and only if there is a right-regular grammar G such that L = L(G). In *particular, every regular language is context-free.*