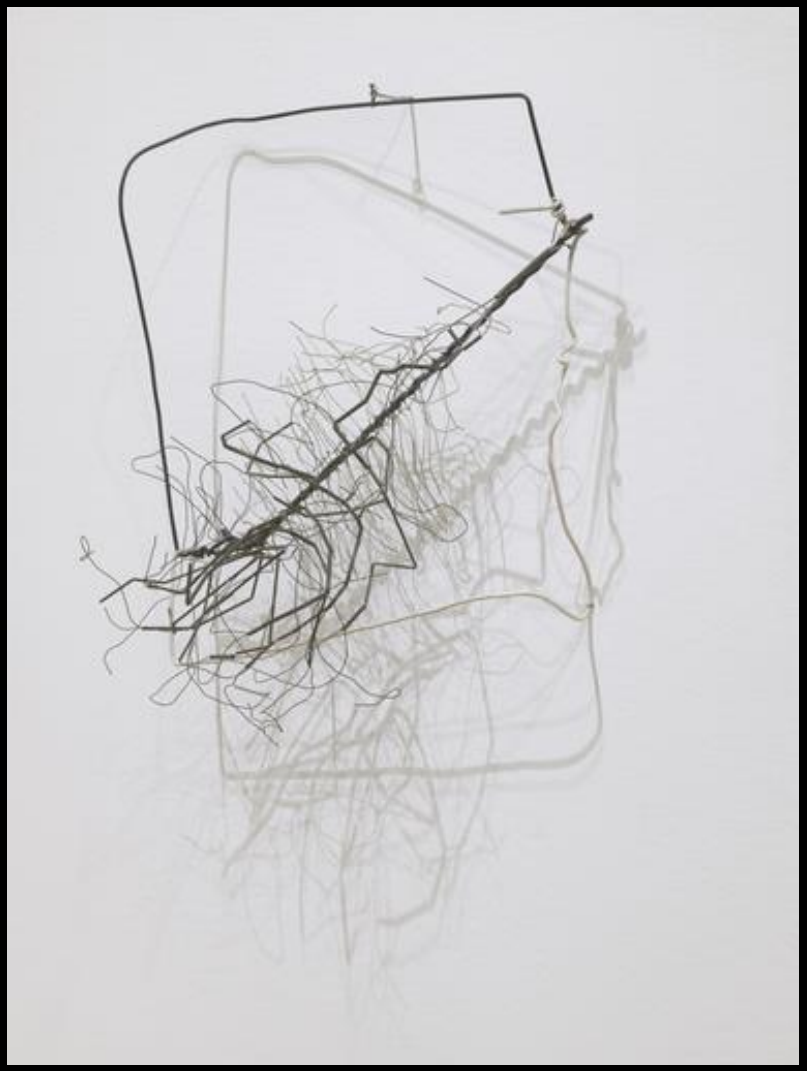


Context-Free Languages

Contents

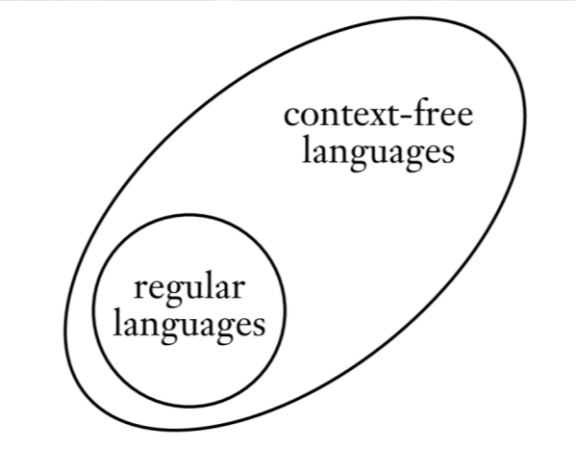
- Context-Free Grammars
- Pushdown Automata
- Non-Context-Free Languages

Context-Free Grammars

• To date we have seen two different, though equivalent, methods of describing languages: *finite automata* and *regular expressions*. In addition, we have seen several examples of non-regular languages, indicating the existence of more powerful abstract computational models.

• In this section we consider **context-free grammars**, a more powerful method of describing languages using a recursive structure.

• The collection of languages associated with context-free grammars are called **context-free languages**. They include all regular languages and many more.



Context-Free Grammars

• Importantly, context-free grammar applications occur in the specification and compilation of programming languages. Designers of **compilers** and **interpreters** for programming languages often start by obtaining a grammar for the language.

• A number of methodologies facilitate the construction of a parser using a context-free grammar; parser are commonly use in NLP (natural language processing) and related AI/ML domains.

Context-Free Grammars

• Context-free grammars are typically defined as a set **of substitution rules** (also called *productions*).

• Each **rule** appears as a line in the grammar, comprising a symbol and a string separated by an arrow. The symbol is called a **variable**; the string consist of variables and **terminals**; terminals are analogous to the input alphabet and may consist of letters, numbers or special symbols.

• One variable is designed as the **start variable**.

Here is a basic example of a context-free grammar *G*:

$$A \rightarrow 0A1$$
$$A \rightarrow B$$
$$B \rightarrow \#$$

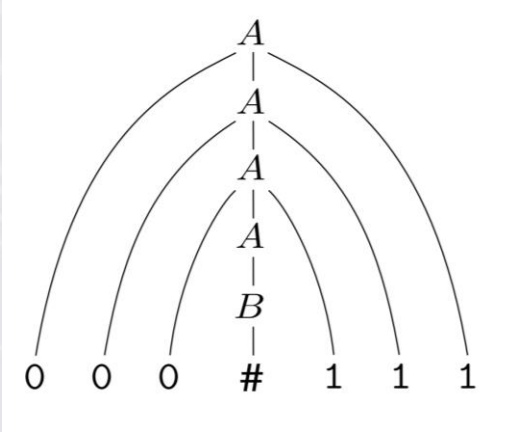
G contains the variables A (start) and B; its terminals are $0,1$ and $\#$.

Context-Free Grammars

• Grammars generate strings in a language: we start with the substitution rule pertaining to the start variable; next, we recursively and sequentially iterate the rules of the grammar until no variables remain.

• For example, *G* given previously can generate the string 000#111 via the following sequence of substitutions:

$$A \to 0A1$$
$$A \to B$$
$$B \to \#$$



$$A \to 0A1 \to 00A11 \to 000A111 \to 000B111 \to 000\#111$$

• We call this procedure a **derivation**. This derivation gives rise to a corresponding visualization called a **parse tree**, as shown.

Context-Free Grammars

• All strings generated in this way constitute the **language of the grammar**, denoted L(G). Any language that can be generated by some context-free grammar is called a **context-free language** (CFL).

We can generate the language of balanced parentheses as follows:

$$S \rightarrow (S)|SS|\varepsilon$$

where the symbol $'|'$ denotes "or" when we combine several grammar derivation rules in one line.

Context-Free Grammars

• Recall that we previously established that the language defined by: $B = \{0^n 1^n | n \geq 0\}$ is not regular. However, it is easy to see that B is a CFL, as the following grammar generates it:

$$S \rightarrow 0S1|\varepsilon$$

Context-Free Grammars

• Here is another example of a context-free grammar, called G_2 which describes a fragment of the English language:

$$
\begin{aligned}
\langle \text{SENTENCE} \rangle &\to \langle \text{NOUN-PHRASE} \rangle \langle \text{VERB-PHRASE} \rangle \\
\langle \text{NOUN-PHRASE} \rangle &\to \langle \text{CMPLX-NOUN} \rangle \mid \langle \text{CMPLX-NOUN} \rangle \langle \text{PREP-PHRASE} \rangle \\
\langle \text{VERB-PHRASE} \rangle &\to \langle \text{CMPLX-VERB} \rangle \mid \langle \text{CMPLX-VERB} \rangle \langle \text{PREP-PHRASE} \rangle \\
\langle \text{PREP-PHRASE} \rangle &\to \langle \text{PREP} \rangle \langle \text{CMPLX-NOUN} \rangle \\
\langle \text{CMPLX-NOUN} \rangle &\to \langle \text{ARTICLE} \rangle \langle \text{NOUN} \rangle \\
\langle \text{CMPLX-VERB} \rangle &\to \langle \text{VERB} \rangle \mid \langle \text{VERB} \rangle \langle \text{NOUN-PHRASE} \rangle \\
\langle \text{ARTICLE} \rangle &\to \texttt{a} \mid \texttt{the} \\
\langle \text{NOUN} \rangle &\to \texttt{boy} \mid \texttt{girl} \mid \texttt{flower} \\
\langle \text{VERB} \rangle &\to \texttt{touches} \mid \texttt{likes} \mid \texttt{sees} \\
\langle \text{PREP} \rangle &\to \texttt{with}
\end{aligned}
$$

Grammar G_2 has 10 variables (the capitalized grammatical terms written inside brackets); 27 terminals (the standard English alphabet plus a space character); and 18 rules. Strings in $L(G_2)$ include:

```
a boy sees
the boy sees a flower
a girl with a flower likes the boy
```

$$
\begin{aligned}
\langle \text{SENTENCE} \rangle &\Rightarrow \langle \text{NOUN-PHRASE} \rangle \langle \text{VERB-PHRASE} \rangle \\
&\Rightarrow \langle \text{CMPLX-NOUN} \rangle \langle \text{VERB-PHRASE} \rangle \\
&\Rightarrow \langle \text{ARTICLE} \rangle \langle \text{NOUN} \rangle \langle \text{VERB-PHRASE} \rangle \\
&\Rightarrow \texttt{a}\ \langle \text{NOUN} \rangle \langle \text{VERB-PHRASE} \rangle \\
&\Rightarrow \texttt{a boy}\ \langle \text{VERB-PHRASE} \rangle \\
&\Rightarrow \texttt{a boy}\ \langle \text{CMPLX-VERB} \rangle \\
&\Rightarrow \texttt{a boy}\ \langle \text{VERB} \rangle \\
&\Rightarrow \texttt{a boy sees}
\end{aligned}
$$

Context-Free Languages

- We now give the formal definition of a CFG:

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

1. V is a finite set called the ***variables***,
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 string of variables and terminals, and $A \to w$ is a rule of the grammar, we say that **uAv yields uwv**, written uAv → uwv; the **language of the grammar** is $\{w \in \Sigma^* | S \to w\}$.

Context-Free Languages

Consider grammar $G_3 = (\{S\}, \{\mathtt{a}, \mathtt{b}\}, R, S)$. The set of rules, R, is

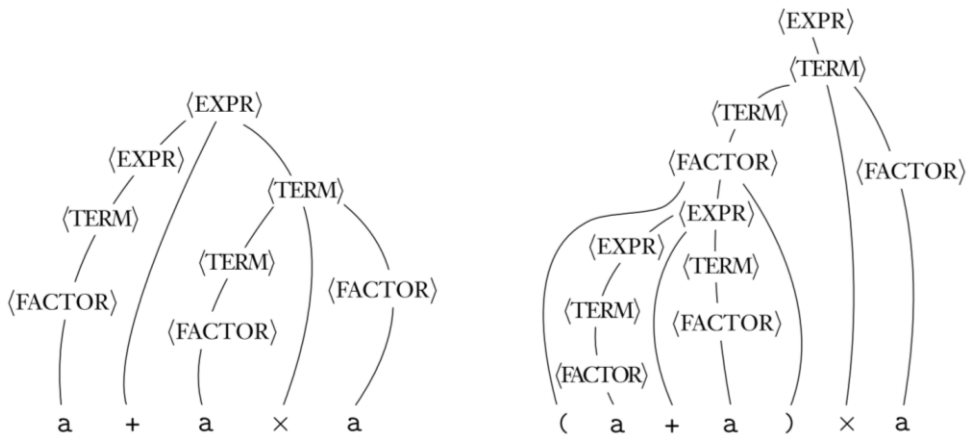
$$S \rightarrow \mathtt{a}S\mathtt{b} \mid SS \mid \varepsilon.$$

This grammar generates strings such as abab, aaabbb, and aababb. You can see more easily what this language is if you think of a as a left parenthesis "(" and b as a right parenthesis ")". Viewed in this way, $L(G_3)$ is the language of all strings of properly nested parentheses. Observe that the right-hand side of a rule may be the empty string ε.

Consider grammar $G_4 = (V, \Sigma, R, \langle \text{EXPR} \rangle)$.
V is $\{\langle \text{EXPR} \rangle, \langle \text{TERM} \rangle, \langle \text{FACTOR} \rangle\}$ and Σ is $\{\mathtt{a}, \mathtt{+}, \mathtt{x}, \mathtt{(}, \mathtt{)}\}$. The rules are

$$\begin{aligned} \langle \text{EXPR} \rangle &\rightarrow \langle \text{EXPR} \rangle \mathtt{+} \langle \text{TERM} \rangle \mid \langle \text{TERM} \rangle \\ \langle \text{TERM} \rangle &\rightarrow \langle \text{TERM} \rangle \mathtt{x} \langle \text{FACTOR} \rangle \mid \langle \text{FACTOR} \rangle \\ \langle \text{FACTOR} \rangle &\rightarrow \mathtt{(} \langle \text{EXPR} \rangle \mathtt{)} \mid \mathtt{a} \end{aligned}$$

The two strings a+axa and (a+a)xa can be generated with grammar G_4. The parse trees are shown in the following figure.



Context-Free Languages

- We now consider some general techniques for constructing CFLs.

- First, note that many CFLs can be defined as the union of simpler CFLs. When possible, consider breaking a complex CFL into discrete "parts", and then take the union of these parts.

As a straightforward example, consider the CFL: $\{0^n 1^n | n \geq 0\} \cup \{1^n 0^n | n \geq 0\}$; an obvious construction is given by:

$$\begin{aligned} S &\rightarrow S_1 \mid S_2 \\ S_1 &\rightarrow 0S_1 1 \mid \varepsilon \\ S_2 &\rightarrow 1S_2 0 \mid \varepsilon \end{aligned}$$

Context-Free Languages

• Second, constructing a CFG for a regular language is generally easy, as one can first construct a DFA, and then convert the DFA into a CFL.

• To do this, make a variable R_i for each state q_i in the DFA; next, add the rules $R_i \rightarrow aR_j$ if $\delta(q_i, a) = q_j$ is a transition in the DFA. Add the rule $R_i \rightarrow \varepsilon$ if q_i is an accept state; make R_0 the start variable where q_0 is the start state of the machine.

• Third, oftentimes CFLs contain certain recursive structures (e.g. requiring the same number of 0s and 1s); in this case, place the variable symbol generating the structure in the location of the rules corresponding to where the recursion occurs, e.g. $\mathrm{R} \rightarrow aRb$.

Context-Free Languages

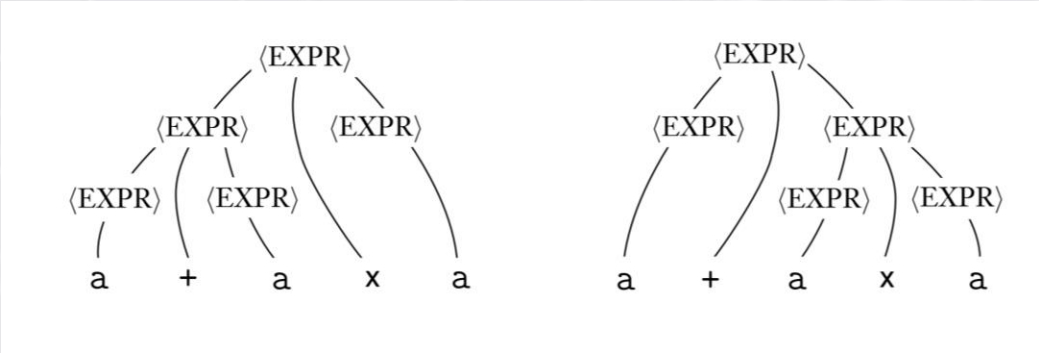
Ambiguity

• Sometimes a grammar can generate the same string in several different ways. Such a string will have several different parse trees (and thus potentially different meanings)

If a grammar generates some string in several different ways, we say (informally) the grammar is **ambiguous**. For example:

⟨EXPR⟩ → ⟨EXPR⟩+⟨EXPR⟩ | ⟨EXPR⟩×⟨EXPR⟩ | (⟨EXPR⟩) | a

This grammar generates the string a+a×a ambiguously. The following figure shows the two different parse trees.



• Notice that the previous example doesn't capture the usual operation precedence rules.

Context-Free Languages

Ambiguity

• We say that a derivation of a string w in a grammar G is a **leftmost derivation** if at every step the leftmost remaining variable is the one replaced. For example, consider the grammar: $\mathrm{X} \to X + X | X * X | X | a$; then the leftmost derivation of the string $\mathrm{a} + \mathrm{a} * \mathrm{a}$ is given by:

$$\mathrm{X} \to X + X \to a + X \to a + X * X \to a + a * X \to a + a * a$$

Context-Free Languages

Ambiguity

• We say that a derivation of a string w in a grammar G is a **leftmost derivation** if at every step the leftmost remaining variable is the one replaced. For example, consider the grammar: $X \to X + X | X * X | X | a$; then the leftmost derivation of the string a + a * a is given by:

$$X \to X + X \to a + X \to a + X * X \to a + a * X \to a + a * a$$

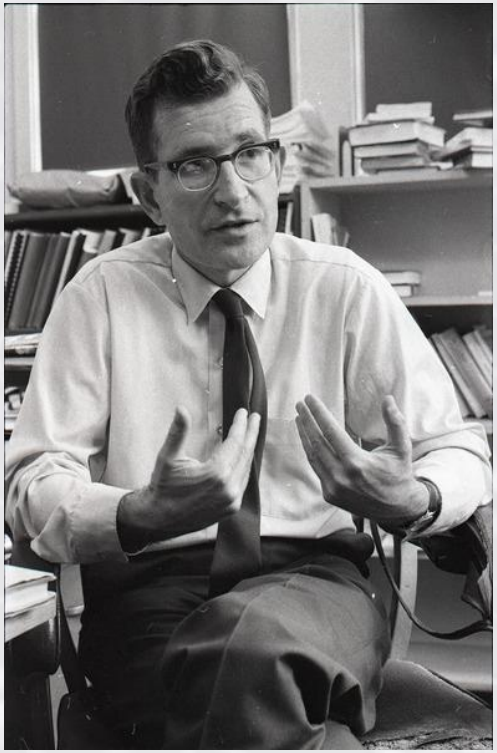
With this in mind, the formal definition of an ambiguous grammar is given by:

> A string w is derived ***ambiguously*** in context-free grammar G if it has two or more different leftmost derivations. Grammar G is ***ambiguous*** if it generates some string ambiguously.

• Some languages can be generated by both ambiguous and unambiguous grammars; however, some languages can only be generated by ambiguous grammars; we call these **inherently ambiguous languages**.

Context-Free Languages

• It is commonly useful to express a CFG in "simplified form"; a common, simple form for CFGs is **Chomsky normal form**. Note that all CFGs admit of a Chomsky normal form (we show this shortly):



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

$$A \to BC$$
$$A \to a$$

where a is any terminal and A, B, and C are any variables—except that B and C may not be the start variable. In addition, we permit the rule $S \to \varepsilon$, where S is the start variable.

Context-Free Languages

Theorem. Any CFL is generated by a CFG in Chomsky normal form.

• Proof idea: We show how to convert any grammar *G* into Chomsky normal form. (1) We add a new start variable; (2) we eliminate all epsilon rules of the form $A \rightarrow \varepsilon$; (3) we eliminate all "unit rules" of the form $A \rightarrow B$; finally, we convert the remaining rules into the proper form.

Context-Free Languages

Theorem. Any CFL is generated by a CFG in Chomsky normal form.

• Proof idea: We show how to convert any grammar *G* into Chomsky normal form. (1) We add a new start variable; (2) we eliminate all epsilon rules of the form $A \to \varepsilon$; (3) we eliminate all "unit rules" of the form $A \to B$; finally, we convert the remaining rules into the proper form.

(1) We add a new start variable S_0 and corresponding rule $S_0 \to S$.

(2) Next, we handle all of the ε rules. To do this we first delete a rule of the form $A \to \varepsilon$ for the variable A; next, for each occurrence of this variable on the right-hand side of a rule, we add a new rule with that occurrence deleted, for instance, the rule $R \to uAvA$ would be replaced by $R \to uAvA$, $R \to uvA$, $R \to uAv$, $R \to uv$.

Context-Free Languages

Theorem. Any CFL is generated by a CFG in Chomsky normal form.

(1) We add a new start variable S_0 and corresponding rule $S_0 \to S$.

(2) Next, we handle all of the ε rules. To do this we first delete a rule of the form $\mathrm{A} \to \varepsilon$ for the variable A; next, for each occurrence of this variable on the right-hand side of a rule, we add a new rule with that occurrence deleted, for instance, the rule $\mathrm{R} \to uAvA$ would be replaced by $\mathrm{R} \to uAvA$, $\mathrm{R} \to uvA$, $\mathrm{R} \to uAv$, $\mathrm{R} \to uv$.

(3) Eliminate all unit rules of the form $\mathrm{A} \to B$, then, whenever a rule $\mathrm{B} \to u$ appears, we add the rule $\mathrm{A} \to u$, for example.

(4) Lastly, we convert all remaining rules into the proper form. For example, the rule $\mathrm{A} \to u_1u_2u_3$ would be replaced with $\mathrm{A} \to u_1A_1$, $A_1 \to u_2A_2$, $A_2 \to u_3$.

Context-Free Languages

Theorem. Any CFL is generated by a CFG in Chomsky normal form.

1. The original CFG G_6 is shown on the left. The result of applying the first step to make a new start variable appears on the right.

$$S \to ASA \mid \mathtt{a}B$$
$$A \to B \mid S$$
$$B \to \mathtt{b} \mid \varepsilon$$

$$\boldsymbol{S_0 \to S}$$
$$S \to ASA \mid \mathtt{a}B$$
$$A \to B \mid S$$
$$B \to \mathtt{b} \mid \varepsilon$$

2. Remove ε-rules $B \to \varepsilon$, shown on the left, and $A \to \varepsilon$, shown on the right.

$$S_0 \to S$$
$$S \to ASA \mid \mathtt{a}B \mid \mathtt{a}$$
$$A \to B \mid S \mid \boldsymbol{\varepsilon}$$
$$B \to \mathtt{b} \mid \varepsilon$$

$$S_0 \to S$$
$$S \to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid \boldsymbol{SA} \mid \boldsymbol{AS} \mid \boldsymbol{S}$$
$$A \to B \mid S \mid \varepsilon$$
$$B \to \mathtt{b}$$

3a. Remove unit rules $S \to S$, shown on the left, and $S_0 \to S$, shown on the right.

$$S_0 \to S$$
$$S \to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS \mid S$$
$$A \to B \mid S$$
$$B \to \mathtt{b}$$

$$S_0 \to S \mid \boldsymbol{ASA} \mid \boldsymbol{\mathtt{a}B} \mid \mathtt{a} \mid \boldsymbol{SA} \mid \boldsymbol{AS}$$
$$S \to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS$$
$$A \to B \mid S$$
$$B \to \mathtt{b}$$

3b. Remove unit rules $A \to B$ and $A \to S$.

$$S_0 \to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS$$
$$S \to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS$$
$$A \to B \mid S \mid \mathtt{b}$$
$$B \to \mathtt{b}$$

$$S_0 \to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS$$
$$S \to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS$$
$$A \to S \mid \mathtt{b} \mid \boldsymbol{ASA} \mid \boldsymbol{\mathtt{a}B} \mid \mathtt{a} \mid \boldsymbol{SA} \mid \boldsymbol{AS}$$
$$B \to \mathtt{b}$$

4. Convert the remaining rules into the proper form by adding additional variables and rules. The final grammar in Chomsky normal form is equivalent to G_6. (Actually the procedure given in Theorem 2.9 produces several variables U_i and several rules $U_i \to \mathtt{a}$. We simplified the resulting grammar by using a single variable U and rule $U \to \mathtt{a}$.)

$$S_0 \to AA_1 \mid UB \mid \mathtt{a} \mid SA \mid AS$$
$$S \to AA_1 \mid UB \mid \mathtt{a} \mid SA \mid AS$$
$$A \to \mathtt{b} \mid AA_1 \mid UB \mid \mathtt{a} \mid SA \mid AS$$
$$A_1 \to SA$$
$$U \to \mathtt{a}$$
$$B \to \mathtt{b}$$

Context-Free Languages

- Give CFGs generating the following languages:

(i) the set of strings over the alphabet $\{a, b\}$ with more a's than b's.

Context-Free Languages

•Give CFGs generating the following languages:

(i) the set of strings over the alphabet $\{a, b\}$ with more a's than b's.

$$S \rightarrow TaT$$

$$T \rightarrow TT|aTb|bTa|a|\varepsilon$$

Context-Free Languages

•Give CFGs generating the following languages:

(i) the set of strings over the alphabet $\{a, b\}$ with more a's than b's.

$$S \rightarrow TaT$$

$$T \rightarrow TT|aTb|bTa|a|\varepsilon$$

(ii) $\{w\#x|w^R \text{ is a substring of } x \text{ for } w, x \in \{0,1\}^*\}$

Context-Free Languages

•Give CFGs generating the following languages:

(i) the set of strings over the alphabet $\{a, b\}$ with more a's than b's.

$$S \rightarrow TaT$$

$$T \rightarrow TT|aTb|bTa|a|\varepsilon$$

(ii) $\{w\#x|w^R \text{ is a substring of } x \text{ for } w, x \in \{0,1\}^*\}$

$$S \rightarrow TX$$

$$T \rightarrow 0T0|1T1|\#X$$

$$X \rightarrow 0X|1X|\varepsilon$$

Context-Free Languages

- Give CFGs generating the following languages:

(iii) The complement of $\{0^i, 1^j | i, j > 0\}$

Context-Free Languages

- Give CFGs generating the following languages:

(iii) The complement of $\{0^i, 1^j | i, j > 0\}$

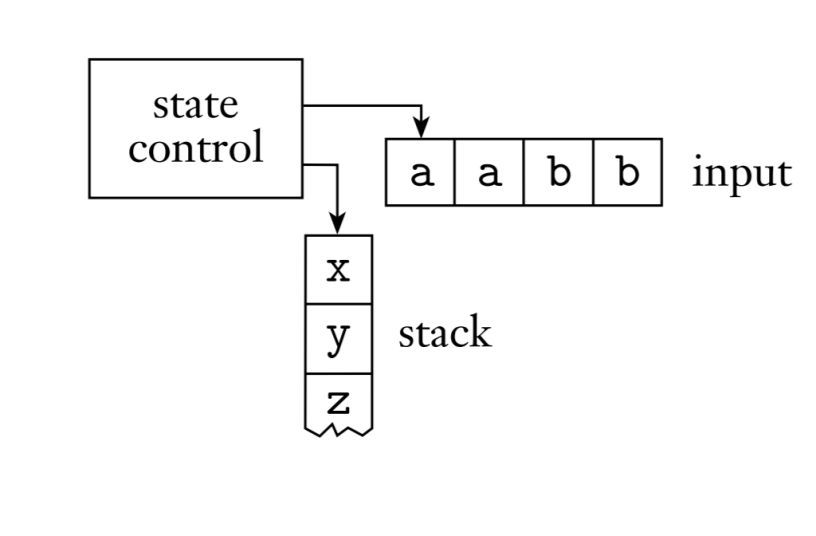
$$S \rightarrow A|B|C$$
$$A \rightarrow D10D$$
$$D \rightarrow 0D|1D|\varepsilon$$
$$B \rightarrow 0B|0, C \rightarrow 1C|1$$

- Notice that A will produce all strings with a 0 and 1 out of order, variable B will produce strings of zeros, variable C will produce strings of ones, and variable D will produce all strings.

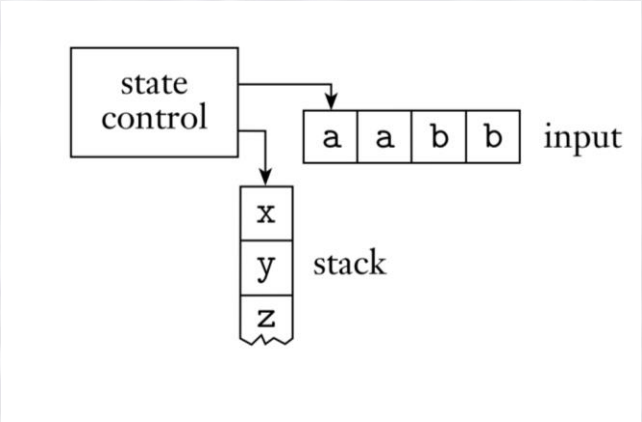
Pushdown Automata

• Finite state automata were severely limited in their computational capacity due to their lack of memory. We now introduce a new type of computational model called a **pushdown automaton (PDA)** which includes an extra memory component called a **stack**.

• PDA can write symbols on the stack and read them back later. Writing a symbol "pushes down" all the other symbols on the stack (think LIFO queue); at any time, the symbol on the top of the stack can be read and removed (i.e. we can "pop" the stack)



Pushdown Automata



•The stack is important structurally as it can hold an unlimited amount of information; this unlimited structure allows the PDA to store numbers of an unbounded size, e.g., $\{0^n 1^n | n \geq 0\}$.

• Additionally, **PDA may be non-deterministic**; unlike FA, non-deterministic PDA can recognize certain languages that no deterministic PDA can recognize. We **assume a referenced PDA is non-deterministic** unless explicitly stated otherwise.

• (non-deterministic) **PDA are equivalent in power to context-free grammars**.

Pushdown Automata

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

1. Q is the set of states,
2. Σ is the input alphabet,
3. Γ is the stack alphabet,
4. $\delta: Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$ is the transition function,
5. $q_0 \in Q$ is the start state, and
6. $F \subseteq Q$ is the set of accept states.

A few notes regarding the formal definition of a PDA:

• The alphabet for the stack Γ may use different (but not necessarily) symbols than those from the input alphabet Σ.

• The domain of the transition function is $Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon$; the non-determinism of the PDA entails that the transition function maps to a set, namely $\mathrm{P}(Q \times \Gamma_\varepsilon)$, so the machine transitions to a new state while writing (possible the empty string) to the top of the stack.

Pushdown Automata

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

1. Q is the set of states,
2. Σ is the input alphabet,
3. Γ is the stack alphabet,
4. $\delta\colon Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$ is the transition function,
5. $q_0 \in Q$ is the start state, and
6. $F \subseteq Q$ is the set of accept states.

A pushdown automaton $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ computes as follows. It accepts input w if w can be written as $w = w_1 w_2 \cdots w_m$, 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 three 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 out properly, in the start state and with an empty stack.
2. For $i = 0, \ldots, m-1$, 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.

Pushdown Automata

• Here is a formal descript of the PDA that recognizes $\{0^n1^n|n \geq 0\}$. Let M_1 be $(Q, \Sigma, \Gamma, \delta, q_1, F)$, where:

$Q = \{q_1, q_2, q_3, q_4\}$,

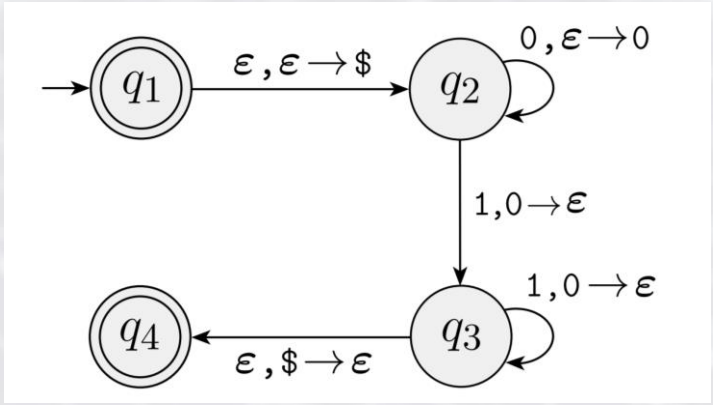
$\Sigma = \{0,1\}$,

$\Gamma = \{0, \$\}$,

$F = \{q_1, q_4\}$, and

δ is given by the following table, wherein blank entries signify $\emptyset$.

Input:	0			1			ε		
Stack:	0	$	ε	0	$	ε	0	$	ε
q_1									$\{(q_2, \$)\}$
q_2			$\{(q_2, 0)\}$	$\{(q_3, \varepsilon)\}$					
q_3				$\{(q_3, \varepsilon)\}$				$\{(q_4, \varepsilon)\}$	
q_4									



Notice that transitions are written in the form: $a, b \rightarrow c$ to signify when a PDA is treading an a from the input, it may replace the symbol b on the stop of the stack with a c. The special symbol **$** is typically used to signal when the stack is empty.

Pushdown Automata

- We now consider the construction of a PDA recognizing the language:

$$\{a^i b^j c^k | i, j, k \geq 0 \; and \; i = j \; or \; i = k\}$$

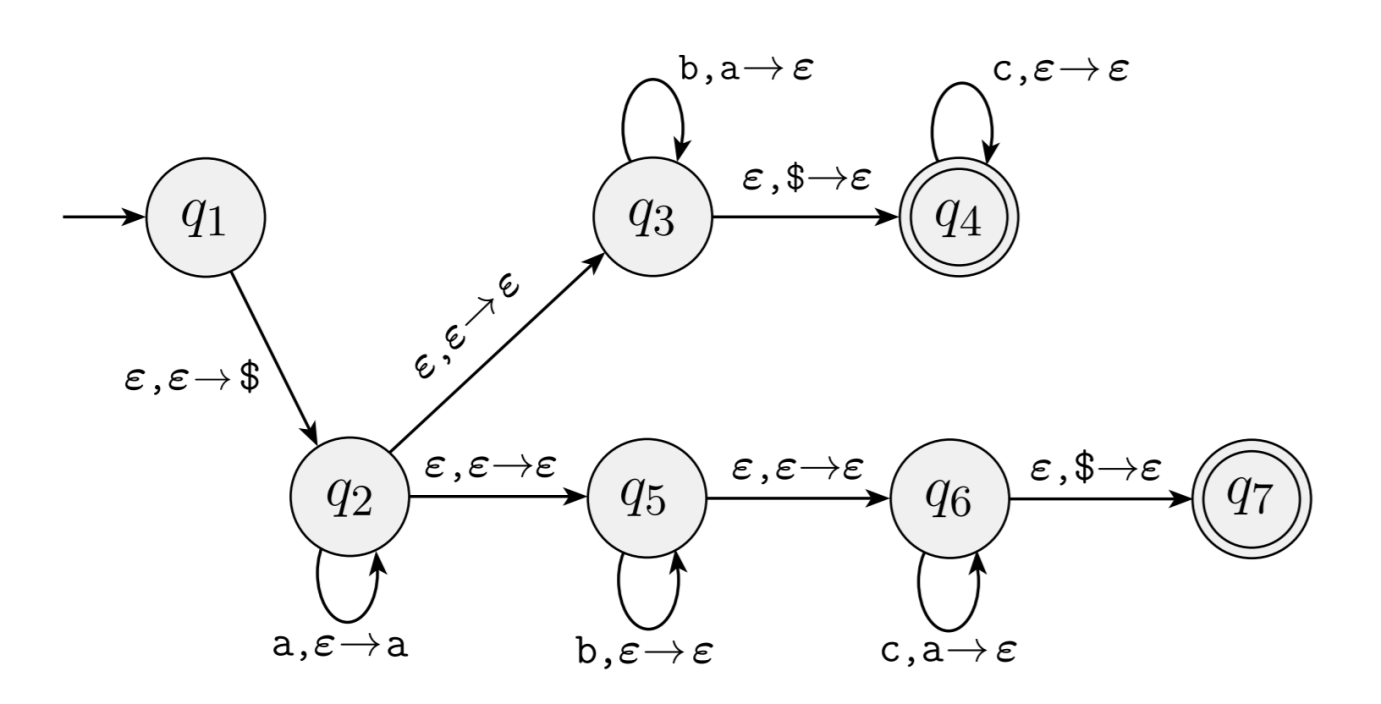
Our general strategy is as follows: read and "push" all of the a's; next match them with either the b's or c's. Because we don't know in advance whether to match the a's with the b's or the c's, we appeal to non-determinism.

Pushdown Automata

• We now consider the construction of a PDA recognizing the language:

$$\{a^i b^j c^k | i, j, k \geq 0 \; and \; i = j \; or \; i = k\}$$

Our general strategy is as follows: read and "push" all of the a's; next match them with either the b's or c's. Because we don't know in advance whether to match the a's with the b's or the c's, we appeal to non-determinism.



Pushdown Automata

- Consider another example, where we construct a PDA for the given language:

$$\{ww^R | w \in \{0,1\}^*\}$$

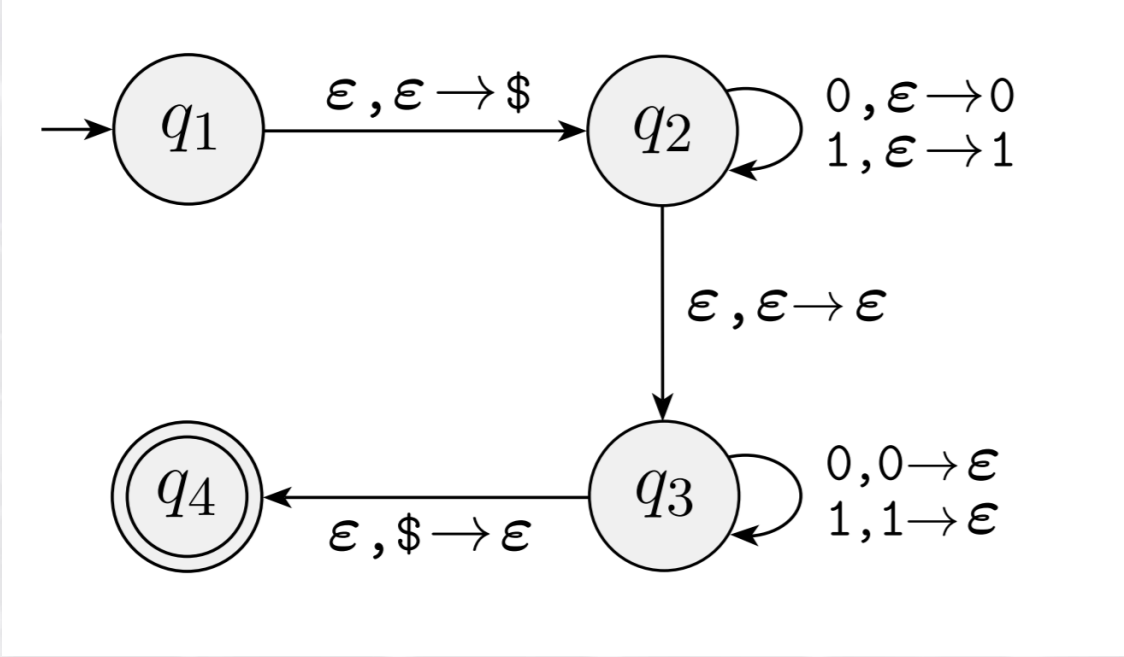
Our general strategy is as follows: begin by pushing the symbols that are read onto the stack; at each point non-deterministically guess that the middle of the string has been reached and then revert to popping off the stack for each symbol read, check to see if they are the same. **Note that this PDA requires non-determinism**.

Pushdown Automata

- Consider another example, where we construct a PDA for the given language:

$$\{ww^R | w \in \{0,1\}^*\}$$

Our general strategy is as follows: begin by pushing the symbols that are read onto the stack; at each point non-deterministically guess that the middle of the string has been reached and then revert to popping off the stack for each symbol read, check to see if they are the same.



Pushdown Automata

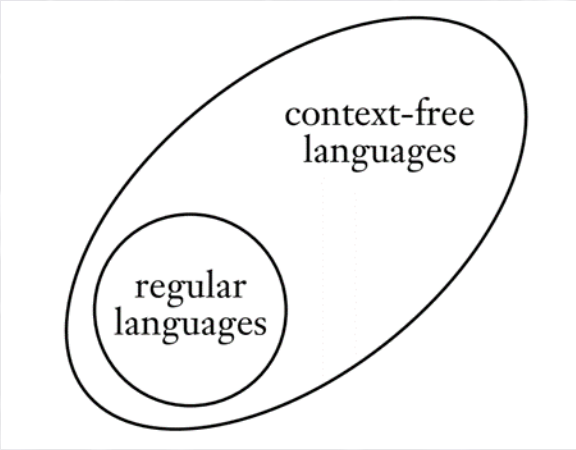
• As mentioned, **PDA and context-free grammars are equivalent in power** (recall we assume the PDA is non-deterministic unless stated otherwise).

The major theorem of this section relates to proving this equivalence:

Theorem. A language is context-free if and only if some PDA recognizes it.

• As usual with any *iff* statement, it is necessary to prove both conditional implications; for brevity, we prove one direction (→)in class (the remaining proof can be found in Sipser).

• A basic consequence of this theorem is that **every regular language is context-free**.



Pushdown Automata

Lemma. If a language is context-free, then some PDA recognizes it.

Proof Idea: Let A be a CFL, so A has a corresponding CFG, G, generating it. We show how to convert G into an equivalent PDA, which we call P.

• The PDA P will work by accepting its input w if G generate that input, by determining whether there is a derivation for w.

Pushdown Automata

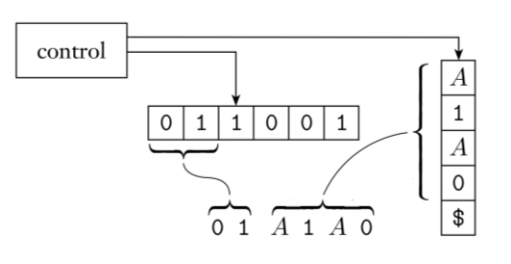
Lemma. If a language is context-free, then some PDA recognizes it.

Proof Idea: Let A be a CFL, so A has a corresponding CFG, G, generating it. We show how to convert G into an equivalent PDA, which we call P.

• The PDA P will work by accepting its input w if G generate that input, by determining whether there is a derivation for w.

• One of the difficulties in testing whether there is a derivation for w is in figuring out which substitutions to make – **we utilize the PDA's non-determinism** to guess the sequence of correct substitutions.

• The remaining complication stems from the issue of the PDA storing the intermediate strings – in short, we store only part of the intermediate string on the stack : the symbols starting with the first variable in the intermediate string. Any terminal symbols appearing before the first variable are matched immediately with symbols in the input string.



Pushdown Automata

Lemma. If a language is context-free, then some PDA recognizes it.

Proof Idea: Let A be a CFL, so A has a corresponding CFG, G, generating it. We show how to convert G into an equivalent PDA, which we call P.

- With those things in mind, here is an informal description of P.

1. Place the marker symbol $ and the start variable on the stack.

2. Repeat the following steps:

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

c. If the top of the stack is the symbol $, enter the accept state.

Pushdown Automata

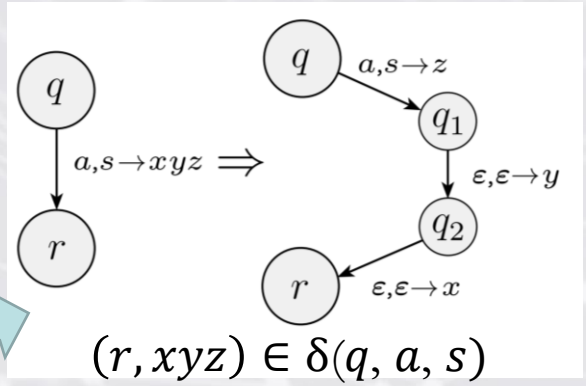
Lemma. If a language is context-free, then some PDA recognizes it.

Proof. We give the construction of the PDA, $P = (Q, \Sigma, \Gamma, \delta, q_{start}, F)$.

Let q and r be states of the PDA and let a be in Σ_ε and s be in Γ_ε. Suppose that we want the PDA to go from q to r when it reads a and pop s. Furthermore, we want it to push the entire string $u = q_1, \ldots, q_l$ on the stack at the same time.

We can implement this action by introducing new states $q_1, \ldots, q_{l-1}$ and setting the transition function as follows:

$\delta(q, a, s)$ to contain (q_1, u_l),
$\delta(q_1, \varepsilon, \varepsilon) = \{(q_2, u_{l-1})\}$,
$\delta(q_2, \varepsilon, \varepsilon) = \{(q_3, u_{l-2})\}$,
$\ldots$
$\delta(q_{l-1}, \varepsilon, \varepsilon) = \{(r, u_1)\}$.

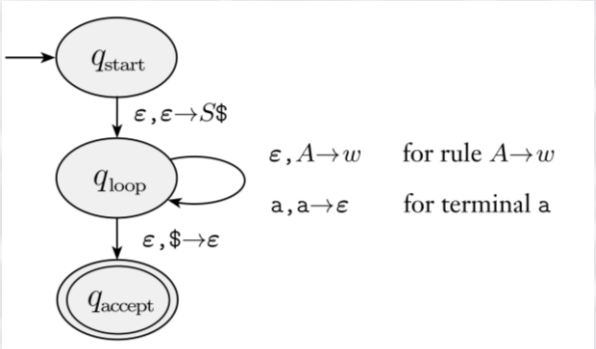


We use the notation $(r, u_1) \in \delta(q, a, s)$ to mean that when q is the state of the automaton, a is the next input symbol, and s is the symbol on the top of the stack, the PDA may read the a and pop the s, then push the string s onto the stack and go on to state r (see figure).

Pushdown Automata

Lemma. If a language is context-free, then some PDA recognizes it.

Proof. We give the construction of the PDA, $P = (Q, \Sigma, \Gamma, \delta, q_{start}, F)$.



The states of P are $Q = \{q_{\text{start}}, q_{\text{loop}}, q_{\text{accept}}\} \cup E$, where E is the set of states we need for implementing the shorthand just described. The start state is q_{start}. The only accept state is q_{accept}.

The transition function is defined as follows. We begin by initializing the stack to contain the symbols \$ and S, implementing step 1 in the informal description: $\delta(q_{\text{start}}, \varepsilon, \varepsilon) = \{(q_{\text{loop}}, S\$)\}$. Then we put in transitions for the main loop of step 2.

First, we handle case (a) wherein the top of the stack contains a variable. Let $\delta(q_{\text{loop}}, \varepsilon, A) = \{(q_{\text{loop}}, w) | \text{ where } A \to w \text{ is a rule in } R\}$.

Second, we handle case (b) wherein the top of the stack contains a terminal. Let $\delta(q_{\text{loop}}, a, a) = \{(q_{\text{loop}}, \varepsilon)\}$.

Finally, we handle case (c) wherein the empty stack marker \$ is on the top of the stack. Let $\delta(q_{\text{loop}}, \varepsilon, \$) = \{(q_{\text{accept}}, \varepsilon)\}$.

Non-Context-Free Languages

• Just as some languages are non-regular (i.e. no FA accepts them), so too there exist languages unrecognized by PDA; we call these **non-context-free languages**.

• Recall that, previously, the *pumping lemma* provided a practical means to prove that a language was non-regular.

• Now we develop an analogous *pumping lemma for context-free languages*.

Non-Context-Free Languages

Pumping lemma for context-free languages If A is a context-free 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. $|vxy| \leq p$.

• It is important to understand the strong parallel between the CFL pumping lemma (above) and the pumping lemma for regular languages (below).

Pumping lemma If A is a regular 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 three pieces, $s = xyz$, satisfying the following conditions:

1. for each $i \geq 0$, $xy^iz \in A$,
2. $|y| > 0$, and
3. $|xy| \leq p$.

* The key difference between the CFL pumping lemma and the version for regular languages is the fact that in the former, **two substrings are "pumped"** (as opposed to one) – this structure is due to the recursive nature of CFLs.

Non-Context-Free Languages

Pumping lemma for context-free languages If A is a context-free 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. $|vxy| \leq p$.

• We omit the full, formal proof of the pumping lemma for CFLs, as it is very similar in flavor to the pumping lemma for regular languages. The basic idea is as follows.

Proof Sketch: Let *A* be a CFL and let *G* be a CFG that generates it.

Let *s* be a string in *A*; because *s* is in *A*, it is derivable from *G* and so has a parse tree. When *s* is "sufficiently" long, this parse tree will contain a path from the start variable to the root with some variable *R* repeated – by the **pigeonhole principle**.

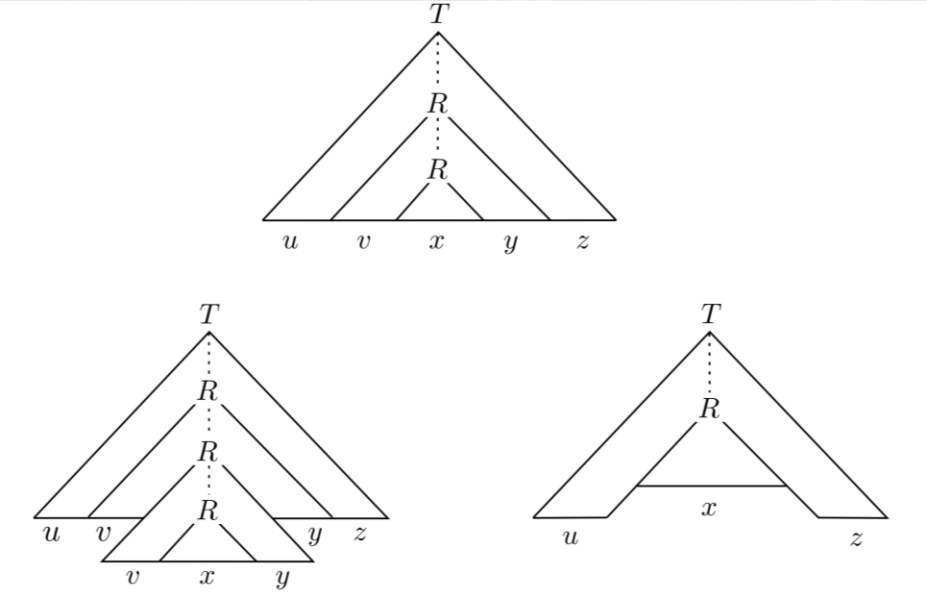
Non-Context-Free Languages

Pumping lemma for context-free languages If A is a context-free 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. $|vxy| \leq p$.

Proof Sketch: Let A be a CFL and let G be a CFG that generates it.

The repetition of R in the parse tree allow us to replace the subtree under the second occurrence of R with the subtree under the first occurrence of R, and we still render a legal parse tree. Thus, we can decompose s into five pieces, i.e., $s = uvxyz$, with the second and fourth pieces repeated and still obtain a string in the language. Hence, $uv^ixy^iz \in A \; \forall i \geq 0$.



Non-Context-Free Languages

Pumping lemma for context-free languages If A is a context-free 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. $|vxy| \leq p$.

Use the pumping lemma to show that the language $B = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\}$ is not context free.

We assume that B is a CFL and obtain a contradiction. Let p be the pumping length for B that is guaranteed to exist by the pumping lemma. Select the string $s = \mathtt{a}^p\mathtt{b}^p\mathtt{c}^p$. Clearly s is a member of B and of length at least p. The pumping lemma states that s can be pumped, but we show that it cannot. In other words, we show that no matter how we divide s into $uvxyz$, one of the three conditions of the lemma is violated.

First, condition 2 stipulates that either v or y is nonempty. Then we consider one of two cases, depending on whether substrings v and y contain more than one type of alphabet symbol.

1. When both v and y contain only one type of alphabet symbol, v does not contain both a's and b's or both b's and c's, and the same holds for y. In this case, the string uv^2xy^2z cannot contain equal numbers of a's, b's, and c's. Therefore, it cannot be a member of B. That violates condition 1 of the lemma and is thus a contradiction.
2. When either v or y contains more than one type of symbol, uv^2xy^2z may contain equal numbers of the three alphabet symbols but not in the correct order. Hence it cannot be a member of B and a contradiction occurs.

One of these cases must occur. Because both cases result in a contradiction, a contradiction is unavoidable. So the assumption that B is a CFL must be false. Thus we have proved that B is not a CFL. ■

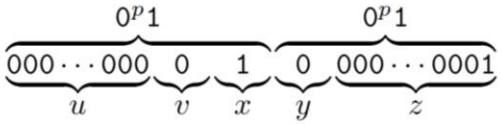
Non-Context-Free Languages

Pumping lemma for context-free languages If A is a context-free 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. $|vxy| \leq p$.

Let $D = \{ww \mid w \in \{0,1\}^*\}$. Use the pumping lemma to show that D is not a CFL. Assume that D is a CFL and obtain a contradiction. Let p be the pumping length given by the pumping lemma.

This time choosing string s is less obvious. One possibility is the string 0^p10^p1. It is a member of D and has length greater than p, so it appears to be a good candidate. But this string *can* be pumped by dividing it as follows, so it is not adequate for our purposes.



Let's try another candidate for s. Intuitively, the string $0^p1^p0^p1^p$ seems to capture more of the "essence" of the language D than the previous candidate did. In fact, we can show that this string does work, as follows.

We show that the string $s = 0^p1^p0^p1^p$ cannot be pumped. This time we use condition 3 of the pumping lemma to restrict the way that s can be divided. It says that we can pump s by dividing $s = uvxyz$, where $|vxy| \leq p$.

First, we show that the substring vxy must straddle the midpoint of s. Otherwise, if the substring occurs only in the first half of s, pumping s up to uv^2xy^2z moves a 1 into the first position of the second half, and so it cannot be of the form ww. Similarly, if vxy occurs in the second half of s, pumping s up to uv^2xy^2z moves a 0 into the last position of the first half, and so it cannot be of the form ww.

But if the substring vxy straddles the midpoint of s, when we try to pump s down to uxz it has the form $0^p1^i0^j1^p$, where i and j cannot both be p. This string is not of the form ww. Thus s cannot be pumped, and D is not a CFL. ■

Fin