Computing is **recognizing/describing/generating/accepting** a language.

1 Context-free grammars: introduction

Two functions associated with languages:

reading: for **language recognizers**; **writing:** for **language generators**

One way to generate a language is to use "expressions," for example, regular expressions. A different way is to use "substitution rules."

Generating $L = \{a\}^*$**:**

rules		application
R1:	$S \rightarrow aS$	
R2:	$S \rightarrow e$	
	rule #1	$S \Rightarrow aS$
	rule #1	$\Rightarrow a^2S$
	rule #1	$\Rightarrow a^3S$
	rule #1	$\Rightarrow a^4S$
	rule #2	$\Rightarrow a^6$

Generating $L = \{a\}^* \cup \{b\}^*$

rules		application
R1:	$S \rightarrow M$	
R2:	$S \rightarrow N$	
R3:	$M \rightarrow aM$	
R4:	$N \rightarrow bN$	
R5:	$M \rightarrow e$	
R6:	$N \rightarrow e$	
	rule #1	$S \Rightarrow M$
	rule #3	$\Rightarrow aM$
	...	....
	rule #5	$\Rightarrow a^m$
	rule #2	$S \Rightarrow N$
	...	....
	rule #6	$\Rightarrow b^n$

Problem 1 *What does this grammar generate?*

	rules	application
R1:	$S \to aSb$	
R2:	$S \to e$	
	rule #1	$S \Rightarrow aSb$
	rule #1	$\Rightarrow aaSbb$
	rule #1	$\Rightarrow aaaSbbb$
	rule #2	$\Rightarrow aaabbb$

Problem 2 *Consider the grammar given in the table below. Show the derivation of the string*

aboywithaflowerseesthegirl

How to improve the grammar so that it would generate words separated by blanks?

1	$\langle SENTENCE\rangle$	$\to$	$\langle NOUN\text{-}PHRASE\rangle\langle VERB\text{-}PHRASE\rangle$
2	$\langle NOUN\text{-}PHRASE\rangle$	$\to$	$\langle CMPLX\text{-}NOUN\rangle$ |
3			$\langle CMPLX\text{-}NOUN\rangle\langle PREP\text{-}PHRASE\rangle$
4	$\langle VERB\text{-}PHRASE\rangle$	$\to$	$\langle CMPLX\text{-}VERB\rangle$ |
5			$\langle CMPLX\text{-}VERB\rangle\langle PREP\text{-}PHRASE\rangle$
6	$\langle PREP\text{-}PHRASE\rangle$	$\to$	$\langle PREP\rangle\langle CMPLX\text{-}NOUN\rangle$
7	$\langle CMPLX\text{-}NOUN\rangle$	$\to$	$\langle ARTICLE\rangle\langle NOUN\rangle$
8/9	$\langle CMPLX\text{-}VERB\rangle$	$\to$	$\langle VERB\rangle$ |$\langle VERB\rangle$ $\langle NOUN\rangle$-$\langle PHRASE\rangle$
10/11	$\langle ARTICLE\rangle$	$\to$	`a`|`the`
12/13/14	$\langle NOUN\rangle$	$\to$	`boy`|`girl`|`flower`
15/16	$\langle VERB\rangle$	$\to$	`likes`|`sees`
17	$\langle PREP\rangle$	$\to$	`with`

Definition 3 *A* **context-free grammar** *G is a quadruple (V, Σ, R, S) where*

V	set of **variables** (nonterminals)
Σ	set of **terminals**; $\Sigma \cap V = \emptyset$
R	set of **rules**; a finite subset of $V \times (V \cup \Sigma)^*$
S	**start** nonterminal.

Notations:

$A \rightarrow_G u$	or $A \rightarrow u$: $(A, u) \in R$
$u \Rightarrow v$:	u **yields** v; $\exists x, y, w \in (V \cup \Sigma)^*, A \in V$ such that $u = xAy;\ v = xwy;\ A \rightarrow w$
$\Rightarrow_G^*$:	the reflexive, transitive closure of $\Rightarrow_G$
$L(G)$:	the language generated by G; $L(G) = \{w : w \in \Sigma^* \& S \Rightarrow_G^* w\}$

derivation $\quad v_0 \Rightarrow_G v_1 \Rightarrow_G v_2 \ldots \Rightarrow_G v_p$

p the length of the derivation

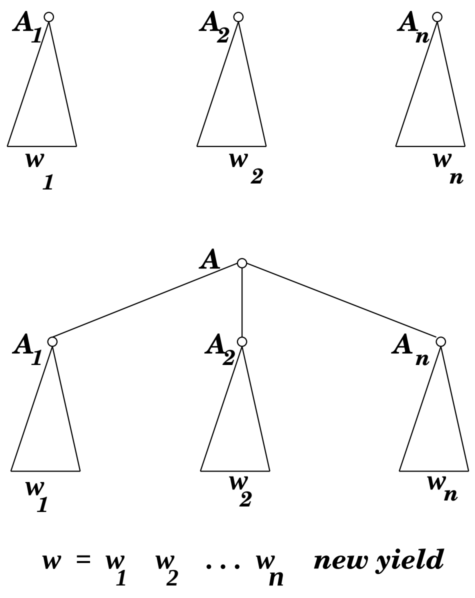
Problem 4 *Consider the grammar $G = (V, \Sigma, R, S)$ where $V = \{S, A\}$; $\Sigma = \{a, b\}$; and*

$$R = \ \{S \rightarrow AA;\ A \rightarrow AAA \mid bA \mid Ab \mid a\}.$$

1. Which strings of $L(G)$ can be produced by derivations of four or fewer steps?
2. Give at least three distinct derivations for the string $babbab$;
3. For any $m, n, p \geq 0$, describe a derivation in G of the string $b^m ab^n ab^p$;
4. Prove that $L(G)$ comprises the set of all strings in Σ^* that contain a positive even number of a's.

Definition 5 *A* **parse (derivation) tree** *of a grammar $G = (V, \Sigma, R, S)$ is a labeled rooted tree obtained through the use of the following operations:*

- *A one node tree labeled A where $A \in V$; the single node of this tree is the root and the leaf; the* **yield** *is A.*
- *If $A \rightarrow w$ is a rule in G, then the root (resp. leaf) of this tree is the node labeled A (resp. w); the yield is w.*
- *If $T_1 \ldots, T_n$ are parse trees with roots labeled $A_1, \ldots, A_n$ and with yields $w_1, \ldots, w_n$ respectively and $A \rightarrow A_1 \ldots A_n$ is a rule in* $\mathbf{R}$*, then the tree T with a root label A whose n subtrees are $T_1, \ldots, T_n$ is also a parse tree. The leaves of T are the leaves of all T_i's and the yield of T is $w_1 \ldots w_n$.*



2 Leftmost derivations

Definition 6 *A derivation is called* **leftmost** *(resp.* **rightmost***) if the leftmost (resp. rightmost) nonterminal is always expanded.*

Proposition 7 *For every for $w \in L(G)$, where G is a context-free grammar, there is a leftmost derivation of w.*

Proof: Induction on the location of a non-leftmost step in a given derivation.

Given a non-leftmost derivation of a word w, we construct a new derivation of w for which a non-leftmost step, if any, is farther to the right than it was in the original derivation. We do it without increasing the total length of the derivation.

We start by locating the 1st occurrence of a non-leftmost derivation.

$$\begin{array}{ccccccccccccc} S & = & w_0 & \Rightarrow & w_1 & \Rightarrow & \dots & w_k & \Rightarrow & w_{k+1} & \Rightarrow & \dots & w_n \in \Sigma^* \\ &&&&&&&& \Uparrow &&&& \end{array}$$

the first occurrence of a non-leftmost derivation

Initial derivation:

$$\begin{array}{ccccccc} \dots \Rightarrow w_k = \alpha A \beta B \gamma & \Rightarrow & \alpha A \beta \delta \gamma & \Rightarrow \dots \Rightarrow & \alpha A \epsilon & \Rightarrow & \alpha \zeta \epsilon \Rightarrow \dots \\ & B \to \delta & & \beta\delta\gamma \Rightarrow \epsilon & & A \to \zeta & \end{array}$$

New derivation:

$$\begin{array}{ccccccc} \ldots \Rightarrow w_k = \alpha A \beta B \gamma & \Rightarrow & \alpha \zeta \beta B \gamma & \Rightarrow & \alpha \zeta \beta \delta \gamma & \Rightarrow \ldots \Rightarrow & \alpha \zeta \epsilon \Rightarrow \ldots \\ & A \rightarrow \zeta & & B \rightarrow \delta & & \beta\delta\gamma \Rightarrow \epsilon & \end{array}$$

To construct a leftmost derivation, we repeat the transformation above as long as there are occurrences of non-leftmost derivations.

Problem 8 *Given a grammar*

$$G: \quad S \rightarrow SS|bA|Ba|e; \quad A \rightarrow aa; \quad B \rightarrow bb$$

and a derivation

$$S \Rightarrow SS \Rightarrow SBa \Rightarrow SSBa \Rightarrow SSbba \Rightarrow SbAbba \Rightarrow Sbaabba \Rightarrow baabba,$$

construct the leftmost derivation of $baabba$.

Solution. The 1st occurrence of the non-leftmost step is in step 2:

$$\ldots \Rightarrow SS \Rightarrow SBa \Rightarrow \ldots.$$

Instead of expanding the 1st S we expanded the 2nd one. According to the algorithm from the proof, we find the step which expands the 1st S. In our example, this is step #3. Thus, by the algorithm, we switch these two steps (#2 and #3). *Remember, if the steps were not neighbors, but were separated by other steps, we would in addition, have placed the separating steps after these two.* The resulting derivation is as follows:

$$S \Rightarrow SS \Rightarrow SSS \Rightarrow SSBa \Rightarrow SSbba \Rightarrow SbAbba \Rightarrow Sbaabba \Rightarrow baabba.$$

The new derivation is *not* leftmost, but the first occurrence of a non-leftmost step is now *farther* from the beginning although the total number of steps didn't change. The 1st non-leftmost step is now #3: $\ldots \Rightarrow SSS \Rightarrow SSBa \Rightarrow \ldots$. Again, we find the step which expands the 1st S *(as you can see, the nonterminal which was supposed to be expanded, but wasn't, happened to be S again).* This time, the step dealing with the missed non-terminal is #7; it uses the rule $S \rightarrow e$. Thus, according to the algorithm, in the new derivation, we preserve the first two steps, then execute step #7, followed by step #3, followed by the steps #4, 5, and 6. The result is:

$$S \Rightarrow SS \Rightarrow SSS \Rightarrow SS \Rightarrow SBa \Rightarrow Sbba \Rightarrow bAbba \Rightarrow baabba.$$

One can continue applying the algorithm pushing the non-leftmost steps to the right, until all of them are removed. On a side, one can also see that the final derivation can be simplified, giving rise to a shorter left-most derivation.

3 Transforming CFG's.

Theorem 9

Let a context-free grammar $G(V, \Sigma, R, S)$ contain a production of the form

$$A \to uBv, \quad \text{where } A, B \in V \text{ and } u, v \in (V \cup \Sigma)^*$$

and let the set of all productions for substituting $B \neq A$ is

$$B \to w_1 | w_2 | \dots | w_n.$$

Construct a new grammar H by removing $A \to uBv$ and adding $A \to uw_iv$ $(i = 1, \dots, n)$. Then

$$L(G) = L(H).$$

Definition 10

A variable A is **useful** for a grammar G, if at least one derivation of a string in $L(G)$ contains A. Else, A is called **useless**. A production is **useless** if it contains any useless variable.

Theorem 11

For every grammar G, there is an equivalent grammar H $(L(H) = L(G))$, without useless variables and useless productions.

Step 1.

```
Procedure Eliminate ($G(V, \Sigma, R, S)$)
 $V_1 = \emptyset$;
 while (not_done)
 $V_1 = V_1 \cup$ {all productions in $G$ of the form $A \to w$ where $w \in (V_1 \cup \Sigma)^*$}
```

Step 2.

Based on G_1, construct a dependency graph, defined as follows:

the set of vertices are variables of the grammar;
there is a directed edge (A, B) iff there is a production $A \to uBv$.

Remove all variables that cannot be reached from S. Remove all productions containing removed variables. The resulting grammar H has no useless variables or productions.

Definition 12

A production of the form $A \to \epsilon$ is called a null-production (λ-production). A variable A for which the derivation $A \Rightarrow^* \epsilon$, is called **nullable**.

Theorem 13

If $\epsilon \notin L(G)$, then there is a grammar H such that

(1) $L(G) = L(H)$, and
(2) H has no null productions.

Definition 14

A production of the form $A \rightarrow B$, where $A, B \in V$, is called a unit-production.

Theorem 15

If $\epsilon \notin L(G)$, then there is a grammar H such that

(1) $L(G) = L(H)$, and
(2) H has no unit productions.

Proof (*idea*). Find, for each variable A, all variables $B \neq A$, such that $A \Rightarrow^* B$. Include in H all non-unit productions. Then for every pair A, B with $A \Rightarrow^* B$, add to H all rules $A \rightarrow w$, for every rule $B \rightarrow w$ currently in H.

4 Regular CFG's.

Definition 16 *A context free grammar is called* **regular** *if for every production $T \rightarrow w$ of G, all letters of w, with a possible exception for the last one, are terminals.*

$$R \subseteq V \times \Sigma^*(V \cup \{e\}).$$

Example:
$V = \{S, T\}; \Sigma = \{a, b\}$

$$\begin{array}{lcl} S & \rightarrow & abaT \\ T & \rightarrow & bbaS|aa \end{array}$$

Theorem 17 *A language is regular iff it is generated by a regular grammar.*

Proof. We prove the statement by presenting two constructions. The first one constructs, for a given DFA, an equivalent regular grammar. The second one constructs, for a given regular grammar, an equivalent NFA (non-deterministic finite automaton).

$\Longrightarrow$ Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA, then the regular grammar $G = (V, \Sigma, R, S)$ is defined as follows:

- set V of variables is the set Q of states in M;

- alphabet Σ is the same as for the DFA;
- the start variable S of G is q_0;
- for every transaction q, p labeled with $a \in \Sigma$, *i.e.* $q = \delta(p, a)$, there is a rule $p \to aq$ in the grammar;
- for every final state $f \in F$, there is a rule $f \to \epsilon$.

One can easily prove that the grammar generates the language $L(M)$ which is accepted by M. End of $\Rightarrow$

$\Leftarrow$ Let $G = (V, \Sigma, R, S)$ be a regular grammar. Before constructing an equivalent finite automaton, modify G as follows:

- replace every rule $A \to a_1 a_2 \cdots a_k B$, where $k \geq 2$ and B is a variable, with the rules $A \to a_1 A_1$, $A_1 \to a_2 A_2$, $\ldots$, $A_{k-1} \to a_k B$. Here $A_1, \ldots, A_k$ are new variables.
- replace every rule $A \to a_1 a_2 \cdots a_k$, where $k \geq 2$, with the rules $A \to a_1 A_1$, $A_1 \to a_2 A_2$, $\ldots$, $A_{k-1} \to a_k$. Here $A_1, \ldots, A_{k-1}$ are new variables.

Obviously, the new grammar is equivalent to the initial one. All rules in the new grammar G' are of the form

$$A \to aB \text{ or } A \to a,$$

where a is a terminal or the empty string ϵ, and B is a variable.

To construct an equivalent NFA,

- define every variable in V as a state of the new NFA;
- for every rule $A \to aB$, introduce a transaction (A, B) labeled with a letter a;
- for every rule $A \to a$ (here a is either a letter or ϵ, the empty string), introduce a new state F and a transaction (A, F) labeled a; make F a final state;
- define the start variable to be the start state of the automaton.

One easily proves that the new NFA accepts the language generated by the grammar. End of $\Leftarrow$

5 Chomsky Normal Form.

Definition 18 *A context-free grammar is in* **Chomsky normal form** *if every rule is of the form*

$$\begin{array}{lcl||l} A & \rightarrow & BC & \textit{A, B, C are variables; } B, C \neq S \\ A & \rightarrow & a & \textit{a is a terminal;} \\ S & \rightarrow & \epsilon & \textit{S is the start variable.} \end{array}$$

Theorem 19 *If L is context-free, then there is a context-free grammar in Chomsky normal form H which generates this language.*

Step 1. Add a new start variable S_0 and a new rule $S_0 \rightarrow S$, where S is the old start variable.

Step 2. For every $A \rightarrow \epsilon$, where $A \neq S$,

- remove it;
- $\forall B \rightarrow uAv$, replace it with $B \rightarrow uv$;
  /*If $B \rightarrow uAvAw$, then we first add $B \rightarrow uvAw$, then $B \rightarrow uAvw$ and then $B \rightarrow uvw$. */
- If $A \rightarrow B$, add $B \rightarrow \epsilon$, unless this rule was previously removed.

Repeat step 2 until all ϵ-rules not involving S are eliminated.

Step 3. For every rule $A \rightarrow B$ and for every rule $B \rightarrow w$,

- remove $A \rightarrow B$;
- add $A \rightarrow w$, unless that was previously removed.

Repeat step 3 until all unit rules are eliminated.

Step 4. For every rule $A \rightarrow u_1u_2 \cdots u_k$, where $k \geq 3$, replace it with the rules

$$A \rightarrow u_1A_1;\ A_1 \rightarrow u_2A_2;\ \ldots,\ A_{k-2} \rightarrow u_{k-1}u_k,$$

where A_i's are new variables.

For $k > 1$, replace any terminal u_i in the preceding rules with the new variable U_i and add the rule $U_i \rightarrow u_i$. ▌▌

6 Some problems for CFG's and some solutions.

Problem 20 *Construct four parse trees for the four derivations of the string babbab in the grammar G of problem 4 (section 1).*

Problem 21 *Consider the grammar $G = (V, \Sigma, R, S)$ where*
$V = \{S, A\}$; $\Sigma = \{a, b\}$; and

$$R = \begin{array}{lcl} S & \rightarrow & aAa \mid bAb \mid \epsilon \\ A & \rightarrow & SS. \end{array}$$

1. Which strings of $L(G)$ can be produced by derivations of four or fewer steps?
2. Give a derivation of *baabbb* in G;
3. Construct the parse tree of the derivation from #2;
4. Describe $L(G)$ in English?

Problem 22 *Construct context-free grammars that generate each of the following languages:*

1. $L = \{wcw^R : w \in (a \cup b)^*\}$;
2. $L = \{ww^R : w \in (a \cup b)^*\}$;
3. $L = \{w \in (a \cup b)^*\} : w = w^R\}$;
4. $L = \{a^m b^n : m \geq n\}$;
5. $L = \{a^m b^n c^p d^q : m + n = p + q\}$;
6. $L = \{w \in \{a, b\}^* : n_a(w) = n_b(w)\}$;
   ($n_a(w)$ denotes the number of occurrences of letter a in the string w)
7. $L = \{w \in \{a, b\}^* : n_a(w) = 2n_b(w)\}$;
8. $L = \{uawb : u, w \in \{a, b\}^* \& |w| = |u|\}$;

Solutions to some problems:

• Construct a CFG that generates

$$L = \{w \in \{a, b\}^* : n_a(w) = n_b(w)\}.$$

Solution. The grammar which generates the language is as follows:

$$G: \quad S \rightarrow aSb|bSa|SS|e$$

Call property A of a string w to have $n_a(w) = n_b(w)$. It is easy to see that every string $w \in L(G)$ has property A. The reverse statement—every string with property A is in L—is also true but the proof is not evident.

We prove it by induction on the length of $|w|$. Obviously, every string with property A must have an even length. Thus, the base of the induction is two strings of length 2. It is easy to see that they are in L.

Assume that the reverse statement is true for all strings of length $n \geq 2$, and let $|w| > n$. If the first and the last letters of w are distinct, $w = \sigma_1 w_1 \sigma_2, \sigma_1 \neq \sigma_2$, then using the appropriate production, we reduce the statement to that on w_1 which is correct by the induction hypothesis.

Let us now consider the case of the first and the last letters of w being equal, say $w = aw_1a$. Moving from left to right, we add 1 for every a and subtract 1 for every b. After the first a, the counter is 1; since at the end it must be 0, its value just before the last a must be -1.
Since every step changes the counter by 1, there must be a moment at which the value of the counter is 0. Therefore, w can be presented as w_1w_2 so that both w_1w_2 where w_1 and w_2 have property A.
To generate w using G, we use $S \to SS$ and derivations of w_1 and w_2 (the latter exist by induction hypothesis).

• Construct a CFG that generates

$$L = \{w \in \{a, b\}^* : n_a(w) = 2n_b(w)\}.$$

Solution. The grammar which generates the language is as follows:

$$G : \quad S \to aaSb|aSab|abSa|aSba|bSaa|baSa|SS|aSbSa|e$$

Call B the property $w_a = 2w_b$ of a word w. We use again induction on $|w|$. It is easy see that the only case that is not trivial is the case of w of the type aaw_1aa or bbw_1bb. Consider the former one. Starting with a 0, move from left to right adding 1 for each a and subtracting 2 after each b. After the second letter, the counter equals 2, and before the last two a's it is -2.
If for some intermediate position, the counter = 0, we follow the proof to the previous problem.
Since for our case the counter changes by 1 and 2, if it is never 0, we prove (make it exact !) that two consecutive values of the counter are 1 and -1, implying that $w = aw_1bw_1a$ where both w_1 and w_2 have property B. Then we use $S \to aSbSa$ to finish the proof.

7 Parsing: algorithmic aspects

Two algorithmic problems related to representing a language by a grammar:

Problem #1: (*the parsing problem*):
given a grammar G and a string w, is $w \in L(G)$?

Problem #2: given a grammar G, is $L(G) = \emptyset$?

Parsing is the process of determining if a string of terminals (*in the Compiler Theory, they are called tokens*) can be generated by a given grammar. Most parsing methods fall into one of the two classes: **top-down** and **down-up** methods. Both construct the parse tree of the input string, but do it in the **opposite** directions. The former starts at the root and proceeds towards leaves, while the latter starts at the leaves and proceeds to the root.

An exhaustive top-down parsing: systematically construct all possible leftmost derivations and see if any of them match w (*the breadth-first search* algorithm).

Example 23

Grammar $G: S \to SS|aSb|bSa|\epsilon$; string $w = aabb$

Round 1: $S \Rightarrow SS$; $S \Rightarrow aSb$; $S \Rightarrow bSa$; $S \Rightarrow \epsilon$
the last two productions can be removed

Round 2: $S \Rightarrow SS \Rightarrow SSS; S \Rightarrow SS \Rightarrow aSbS$;
$S \Rightarrow SS \Rightarrow bSaS; S \Rightarrow SS \Rightarrow S$;
$S \Rightarrow aSb \Rightarrow aSSb; S \Rightarrow aSb \Rightarrow aaSbb$;
$S \Rightarrow aSb \Rightarrow abSab; S \Rightarrow aSb \Rightarrow ab$;

....

Round 3: $S \Rightarrow aSb \Rightarrow aaSbb; S \Rightarrow aabb.$

Idea for parsing: list out all derivations (= parse trees) that produce strings of length $1, 2, \dots, |w|$.

Caveat: some of the rules are of the form $A \to \alpha$, where $|\alpha| \leq 1$, which may create infinite number of derivations even though they produce a finite number of strings; thus, the algorithm must understand how to eliminate repetitions.

New idea: modify G so that the resulting grammar G' generates the same language but does not have rules of the type $A \to \epsilon$ and $A \to B$.

Caveat: It is not always possible, since if $\epsilon \in L(G)$, then G must have a rule $A \to \epsilon$ for some $A \in V$.

Final idea: construct a new grammar G' such that

- $L(G) = L(G')$;
- G' does not have any rule $A \to \epsilon$ and $A \to B$, where $A, B \in V$;
- if $\epsilon \in L(G)$ then $S \to \epsilon$ is allowed.

Question: why does the final idea work?

Answer: Let every rule of grammar L be of the form $A \to u$ where u is either a terminal symbol, or $|u| > 1$, with only exception for $S \to \epsilon$. Then after each derivation step, different from $S \to \epsilon$, either the derived string becomes longer, or the number of its terminals becomes bigger. Thus, any derivation of $w \neq \epsilon$ has at most $2|w| - 1$ steps. Consequently, we only need to examine parse trees whose size is bounded by $2|w| - 1$. Therefore, for every w, the algorithm determines in a finite number of steps whether $w \in L(G)$.

The implementation of the idea for parsing.

Eliminate all rules $A \to \epsilon$ except for $S \to \epsilon$.
For every rule $A \to \epsilon$, if there is a rule $B \to uAv$, add $B \to uv$ to the set of rules. Repeat until no new rule can be added. Then remove all rules of the form $A \to \epsilon$, except $S \to \epsilon$, if it is in the grammar.

For the resulting grammar G', $L(G) = L(G')$. (*Why ?*)

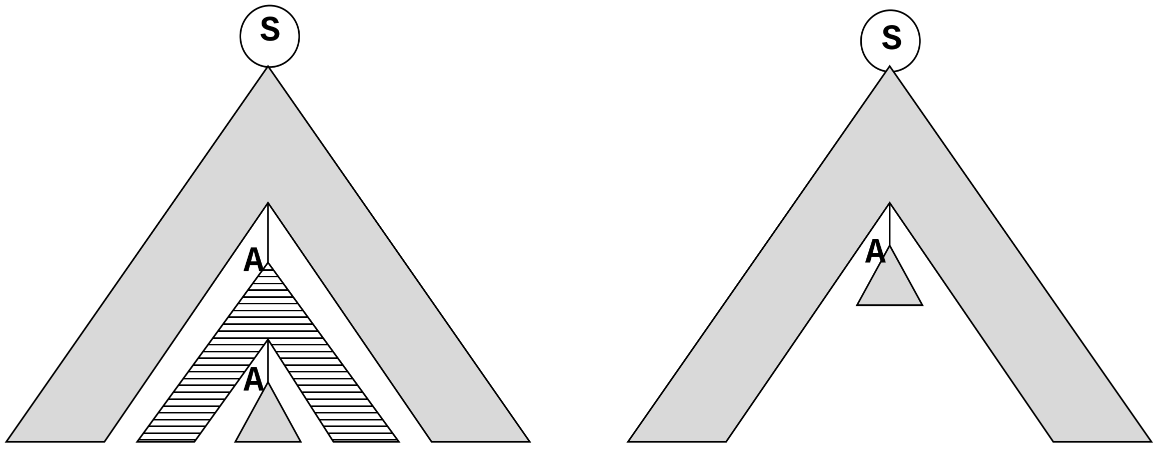
Eliminate all rules $A \to B$.

- For every two variables A and B, determine if $A \Rightarrow^* B$; (*How?*).
- Whenever $B \to u$ is a rule in G and $A \Rightarrow^* B$, add $A \to u$. Repeat until no new rule can be added. Eliminate all rules $A \to B$.
  For the resulting grammar G'', $L(G) = L(G'')$. (*Why ?*)

Problem #2: Given a grammar G, is $L(G) = \emptyset$?

Answer. If grammar G generates a nonempty language, then there is a parse tree whose yield is a terminal string and whose height $\leq |V|$. This is because every parse tree of height $> |V|$ has a path with two nodes labeled with the same variable. Then, a smaller parse tree can be constructed by excising the part of the tree between the nodes with the same label. The process can be repeated until a parse tree of height $\leq |V|$ is constructed with the terminal yield.

Thus, to answer the question, it is sufficient to search through all parse trees of height $\leq |V|$; if none of them yields a terminal string, the language is empty. ||



The solutions above to both problems are *only of a theoretical* value; a *practical* solution (when it is available) is based on the notion of a *pushdown automaton.*

8 Ambiguity

Steve saw a man with a telescope

Two ways to parse (to understand) the sentence:

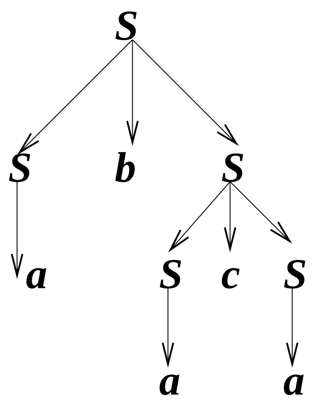
Steve used a telescope to see the man | The man was with the telescope

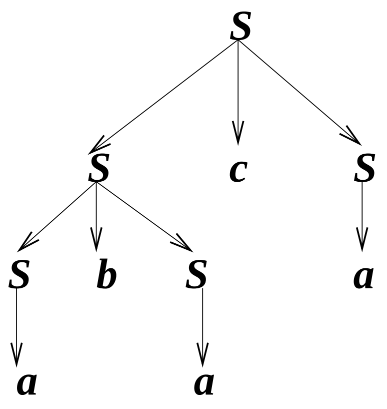
A grammar is said to be **ambiguous** if two or more parse (derivation) trees exist for the same string. A grammar which is not ambiguous is called **unambiguous**.

Example 24 $S \to SbS \mid ScS \mid a$; $S \Rightarrow_G^* abaca.$

derivation #1: $S \Rightarrow SbS \Rightarrow abS \Rightarrow abScS \Rightarrow abacS \Rightarrow abaca$

derivation #2: $S \Rightarrow ScS \Rightarrow SbScS \Rightarrow abScS \Rightarrow abacS \Rightarrow abaca$





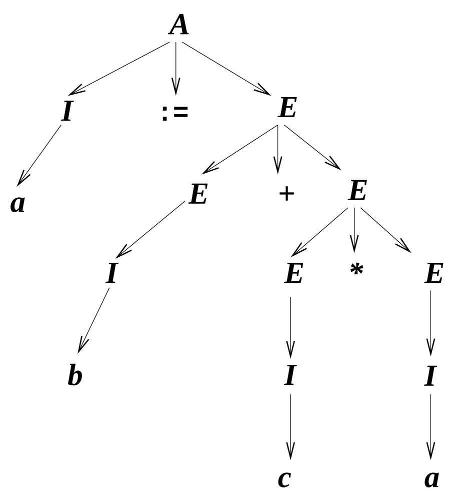
Example 25 $S \to aSb \mid SS \mid \epsilon$; $S \Rightarrow_G^* abaca.$

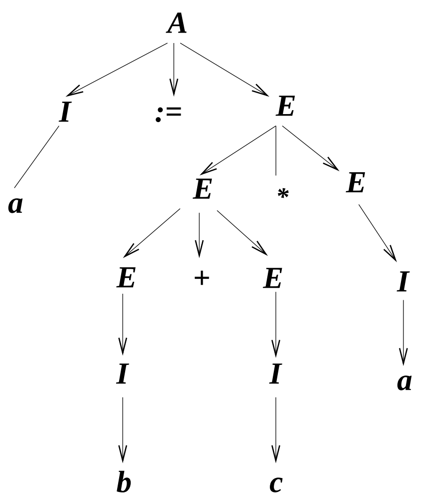
An unambiguous equivalent of the grammar: $S \to aTb \mid SS$; $T \to S \mid \epsilon$;

Example 26 *The grammar below describes a fragment of a programming language concerned with arithmetic expressions.*

$A \to I := E;\ I \to a|b|c;\ E \to E + E|E * E|(\ E\ )|I.$

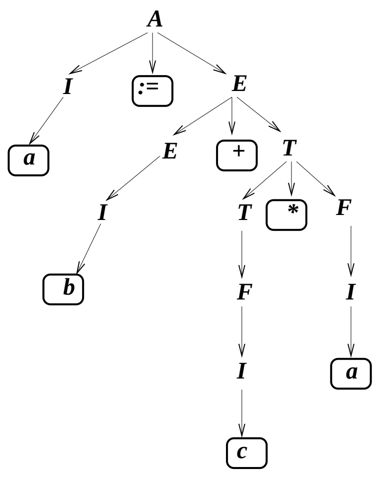
$A \Rightarrow_G a := b + c * a$





An unbiguous version of the grammar for arithmetic expressions.

$A \to I := E; I \to a|b|c;\quad E \to T \mid E + T;\quad T \to F \mid T * F;\quad F \to I \mid (\ E\ ).$



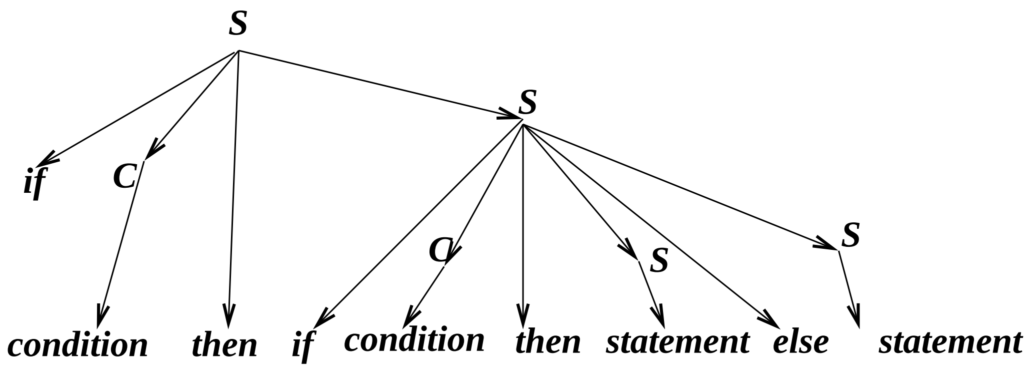
Example 27

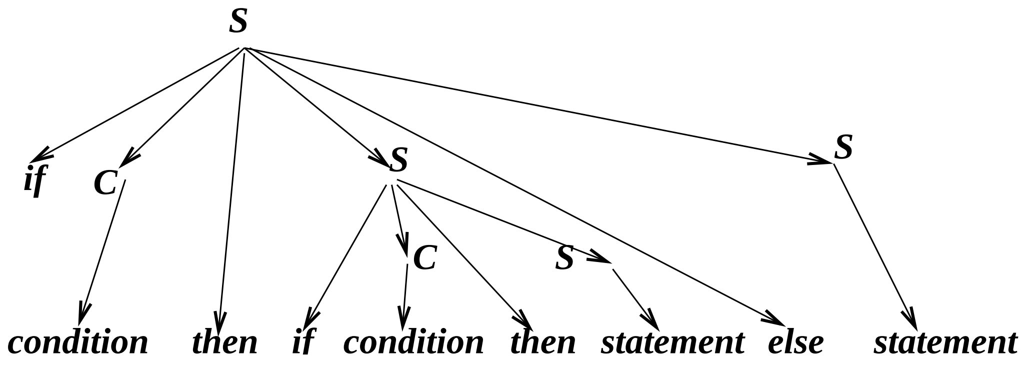
$S \quad \to$ `if` C `then` S `|` `if` C `then` S `else` S `|` `statement;` $C \to$ `condition`

The grammar is ambiguous; consider the string

if C_1 **then if** C_2 **then** S_1 **else** S_2.

(if $a = b$ **then if** $b = c$ **then** $x = 1$ **else** $x = 2$**)**





An unambiguous equivalent grammar (*idea*):

$$\begin{array}{lcl} S & \rightarrow & M \mid U \\ M & \rightarrow & \textbf{if } C \textbf{ then } M \textbf{ else } M \mid \textbf{ other} \\ U & \rightarrow & \textbf{if } C \textbf{ then } S \mid \textbf{if } C \textbf{ then } M \textbf{ else } U \end{array}$$

9 Pushdown Automata: finite automata + memory

Definition 28 *A pushdown automaton M is a 6-tuple* $(Q,\ \Sigma,\ \Gamma,\ \delta,\ q_0,\ F)$*, where*

$Q:$	*a finite set of* **states;**
$\Sigma:$	*a finite alphabet of* **input** *symbols;*
$\Gamma:$	*a finite alphabet of* **stack** *symbols;*
$q_0 \in Q:$	*the* **initial** *state;*
$F \subseteq Q:$	*a set of* **final** *states;*
$\delta:$	*the* **transition** *function;*
	$Q \times (\Sigma \cup \epsilon) \times (\Gamma \cup \epsilon) \to \mathcal{P}(Q \times (\Gamma \cup \epsilon))$.

An input string w to a PDA M is considered to be written on the **input tape** as

$$w = w_1 w_2 \cdots w_m,$$

where $w_i \in \Sigma \cup \epsilon \equiv \Sigma_\epsilon$.

Given the current state $q \in Q$, the current letter $u \in \Sigma \cup \epsilon$ of the input, and the current letter $a \in \Gamma \cup \epsilon \equiv \Gamma_\epsilon$ at the top of the stack, if

$$(q', b) \in \delta(q, u, a),$$

then M **may** select q as its next state and replace the top letter in the stack with b. If $\delta(q, u, a) = \emptyset$, no action of M is possible.

A computation by M begins in the start state with an empty stack.

If the input string w is written as

$$w = w_1 w_2 \cdots w_m,$$

then for every $i \in [1, m-1]$,

$$(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a),$$

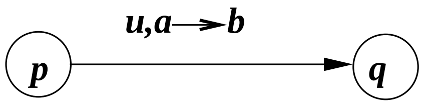
where $\{r_i\}$ is the sequences of states; a, b are stack symbols.

A string w is **accepted** by M if at the end of w M enters a final state.

The definition of a PDA does not include a requirement of an empty stack at the end of computation neither does it contain a formal mechanism to allow the PDA to test for an empty stack. However, it is always possible to achieve that the stack is empty at the end by introducing a special symbol \$ which is initially placed on the stack. If the PDA ever sees the \$ again, it "knows" that the stack is effectively empty. Thus, we will always design pushdown automata that empty the stack by the time they accept the string.

Definition 29 *The language $L(M)$ accepted by M is the set of all strings accepted by M.* ∎

If $(q, b) \in \delta(p, u, a)$, we say that $(p, u, a), (q, b)$ is a **transition** of M. A transition $(p, u, a), (q, b)$ is represented by a diagram composed of two nodes labeled p and q, respectively, and an arrow (p, q) labeled with $u, a \to b$.

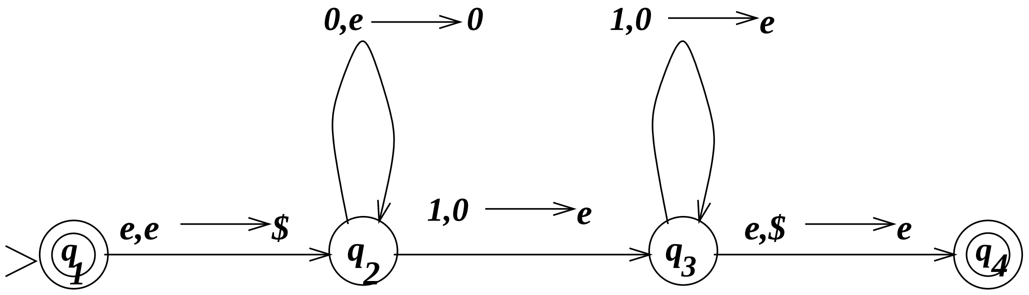


Since $\delta(q, u, a) \subseteq Q \times \Gamma \cup \epsilon$, at every particular step, there can be more than one applicable transitions; if $\delta(q, u, a) = \emptyset$, then no transition is applicable.

The transition function δ is completely described by the set Δ of its applicable transitions.

If M performs transition $(q, u, a)(p, \epsilon)$, we say that M **pops** a from the stack. The transition is called a pop. If M performs transition $(q, u, \epsilon)(q, b)$, we say that M **pushes** b onto the stack; the transition is called a push.

Example 30 $G = (Q, \Sigma, \Gamma, \delta, q_1, F)$, *where* $Q = \{q_1, q_2, q_3, q_4\}$; $\Sigma = \{0, 1\}$*;* $\Gamma = \{0, \$\}$; *and* $F = \{q_1, q_4\}$. *What is the language recognized by this pushdown automaton?*



Example 31 *What is the language recognized by the pushdown automaton described below?*

Q	$= \{q_0, q_1, q_2, f\}$	
Σ	$= \{a, b, c\}$	The transition function δ is represented by the
Γ	$= \{a, b, \$\}$	diagram below. For all triples (q, u, a) that are not
F	$= \{f\}$	shown on the diagram, $\delta(q, u, a) = \emptyset$.

10 Pushdown Automata and Context-free Grammars

Theorem 32 *The class of languages accepted by pushdown automata is exactly the class of context-free languages.*

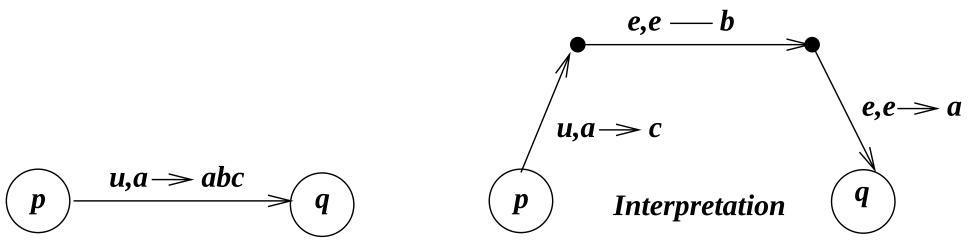
$\Longrightarrow$: Each context-free language is accepted by a PDA.

$\Longleftarrow$: Each language accepted by a PDA is context-free.

Proving $\Longrightarrow$:

$\forall G = (V, \Sigma, R, S)$,	$\Delta :$	$(p, \epsilon, \epsilon)(q, S)$	
$\exists M = (\{p, q, \ldots\}, \Sigma, V \cup \Sigma, \Delta, p, \{q\})$		$(q, \epsilon, A)(q, w)$,	$\forall A \to w$
such that $L(G) = L(M)$.		$(q, a, a)(q, \epsilon)$,	$\forall a \in \Sigma$.

How to interpret a transition $(q, u, a)(q, w)$, where w is a string?



For every string that can be derived by a grammar G, the new PDA accepts the string by "mimicking" its leftmost derivation. The end of the derivation corresponds to the moment the PDA finished reading the input-string and its stack is empty.

Example 33

$G :\ V = \{S, T\}; \Sigma = \{a, b\}$	
$R : \{S \to abTa; T \to ba : \ldots\}$	
Derivations in G	Transitions in M
$S \Rightarrow abTa$	$(p, \epsilon, \epsilon)(q, S)$; $(q, \epsilon, S)(q, abTa)$;
	$(q, a, a)(q, \epsilon)$; $(q, b, b)(q, \epsilon)$;
$\Rightarrow abbaa$	$(q, \epsilon, T)(q, ba)$;
	$(q, b, b)(q, \epsilon)$; $(q, a, a)(q, \epsilon)$; $(q, a, a)(q, \epsilon)$.

Proving $\Longleftarrow$:

Given PDA M, construct a grammar G, such that $L(G) = L(M)$.

First, we modify M so that it satisfies the following conditions:

1. the final state of M is unique; (*easy*)
2. when M accepts a string, its stack is empty; (*easy*)
3. each transition is either a push or a pop.

   Let $(p, u, a)(q, b)$ be a transition with $a \neq \epsilon$ and $b \neq \epsilon$. Replace it with $(p, u, a)(q_{new}, \epsilon)\&(q_{new}, \epsilon, \epsilon)(q, b)$, where q_{new} is a new state.

Design $G = (V, \Sigma, R, q_0)$ by:

$$\begin{array}{ll||lll} V = & \{A_{p,q}\}\ (\forall p, q \in Q); & A_{pq} \to aA_{rs}b & \forall (p,a,\epsilon)(r,t)\ \&\ (s,b,t)(q,\epsilon) \\ \Sigma & \text{set of terminals;} & A_{pq} \to A_{pr}A_{rq}, & \forall p,q,r \in Q \\ A_{q_0,f} & \text{the start variable;} & A_{pp} \to \epsilon. & \forall p \in Q. \end{array}$$

Proof. We prove a more general statement:

> $\forall p, q \in Q(M)$, the set $L(A_{p,q})$ of strings generated from A_{pq} is the set Σ_{pq} of strings that take M from state p to state q, starting and ending with the empty stack.

Claim 1. We prove that every $w \in L(A_{p,q})$ is also a string in Σ_{pq} by induction on the length of the derivation of w.

If the number of steps in the derivation of w is 1, then the rule used can only be $A_{pp} \to \epsilon$. Obviously, $\epsilon \in \Sigma_{p,p}$, since it takes M from p to p with the stack empty at the beginning and the end.

Assuming the statement is correct for all derivations with the number of steps $\leq k \geq 0$, and let the number of steps in the derivation of w_{new} is $k + 1$.

Case 1: the first step in the derivation of w_{new} is $A_{pq} \Rightarrow aA_{r,s}b$. Then $w_{new} = aw'b$ and the following two transitions must exist in M: $(p, a, \epsilon)(r, t)$ and $(s, b, t)(q, \epsilon)$.

Case 2: the first step in the derivation of w_{new} is $A_{pq} \Rightarrow A_{pr}A_{rq}$. Then $w_{new} = w'w''$, where w' and w'' are two strings generated in G from A_{pr} and Ar, q, respectively.

In both case, the statement about w_{new} follows from that for w' (case 1) and for $w'\&w''$ (Case 2).

Claim 2. We prove that $\Sigma_{pq} \subseteq L(A_{pq})$ by induction on the length of w.

If $w \in \Sigma_{pq}$, that is w can bring M from p to q with the empty stack at the beginning and the end, then the first (resp. last) transition of w is a push (resp. pop).

Let $|w| = 0$. Then $w = \epsilon$ and the path starts and ends at p. The derivation of w is $A_{pp} \to \epsilon$.

Let our statement be correct for strings of length $k \geq 0$ and let string w of length $k+1$ can bring M from p to q with the empty stack at the beginning and the end. If moving M from p to q contains a moment at which the stack is empty, the first step of a derivation of w in G is $A_{pq} \to A_{pr}A_{r,q}$.

Else, let $(p, a, \epsilon)(r, t)$ and $(q, b, s)(q, \epsilon)$ be the first and the last steps, respectively, in the path from p to q according to w. Since during no intermediate step the stack becomes empty, the symbol t pushed into the stack the first step is the symbol which is popped during the last step. Thus, $s = t$, and $w_{new} = aw'b$. Furthermore, w' takes M from r to s with the stack being empty initially and at the end. Since $|w'| = k - 1$, it can be derived from $L_{r,s}$. Then the first step of the derivation for w is

$$A_{pq} \Rightarrow aA_{r,s}b.$$

The derivations that follow are those used for w'. ▮

11 Languages that are not context-free.

The pumping lemma for context-free languages is a technique for proving that certain languages are not context-free. It states that every context-free language has a special value called **pumping length** such that all sufficiently long strings in the languages can by "pumped". The pumping theorem is typically used to prove that some languages are not CFL: for this purpose, one proves that a given language has no finite pumping length.

Theorem 34 *(The Pumping Theorem) Let G be a context-free grammar which describes an infinite language. Then there is a number p depending on G only such that* **any** *string $w \in L(G)$ of length $> p$ can be rewritten as $uvxyz$ so that*

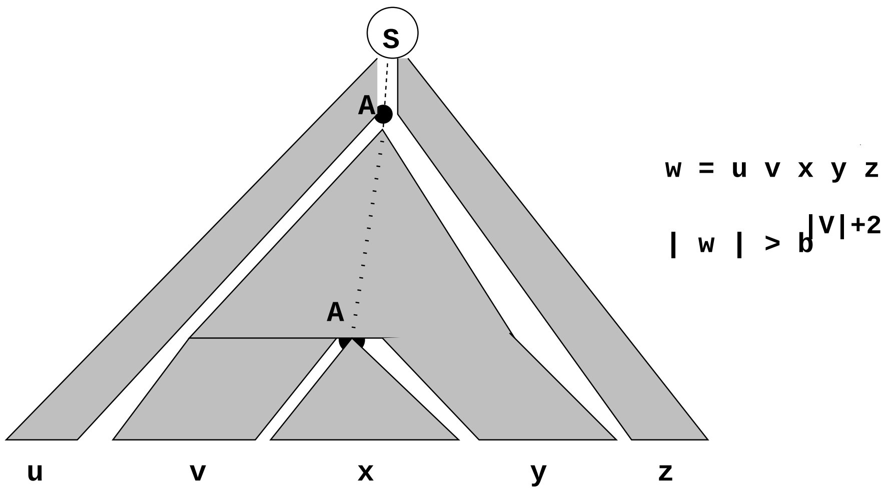
- $|v| + |y| > 0$;
- $|vxy| \leq p$;
- $uv^ixy^iz \in L(G) \quad \forall i \geq 0$.

Important comment: *the theorem asserts the* **existence** *of a sub-division into five substrings that satisfy the three conditions above; it does not indicate what they are, or how many exist. Not every subdivision would satisfy the three conditions.*

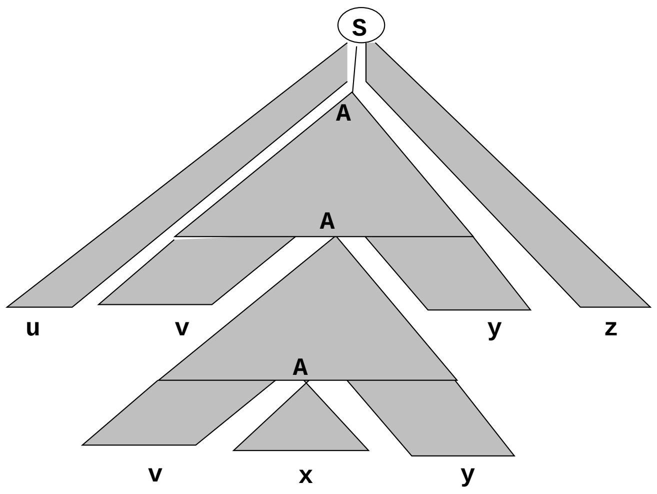
Proof. Let an infinite CFL L be generated by a CFG $G(V, \Sigma, R, S)$ and let b be the maximum length of the string in the right-hand side of any rule. Since L is infinite, $b \geq 2$.

> **Claim.** In any parse tree of a string $w \in L$, the number of nodes on the k-th level ($k \geq 0$) of the tree does not exceed b^k.
>
> This claim is easily proved by induction of k. Indeed, for $k = 0$, it is trivially correct. If the claim is correct for some level k, then the size of the level $k+1$ can increase by at most b times, since every node may have at most b children (for some nodes the number of children is smaller than b, if the the corresponding rule has fewer than b letters in its right-hand side.).

Set $p = b^{|V|+2}$ and let $w \in L$ with $|w| > p$. Take the parse tree for w with the **smallest** number of nodes. The height of the tree is at least $|V| + 2$, and so is the length of the longest path of this tree. The labels of the nodes of this path are variables, except for the leaf which is labeled with a terminal. Since there are $|V| + 2$ labels of non-leaf nodes and V variables, by pigeonhole principle, some variable A appear more than once on the path. Select A to be the variable that repeats among the **lowest** $|V| + 1$ variables on the path. Split w into five parts u, v, x, y, and z according to the figure below:



Both subtrees vxy and x are generated by the some variable A, so we may substitute one for the other and still obtain a valid tree. Repeating the substitution i times, we get i "pumped" parsed trees of strings in L. This satisfies the first condition of the lemma.



To satisfy the second condition, we must check, that v and y are not both empty strings. Indeed, if they were, the subtree generated by the upper A would be a path and a tree generated from the last vertex of the path; the labels of the first and the last vertices of the path would be A. But then, we would be able to construct a smaller parse tree for w by simply removing the edges of the path. This would contradict the minimality of the initial parse tree for w.

To show that the third condition of the theorem holds, notice that in the parse tree, the upper occurrence of A generates vxy. By our choice, both occurrences of A are among the bottom $|V|+2$ nodes of the longest path of the tree. So the height of the subtree generated by A is at most $|V|+2$. This implies that the length of the yield of this subtree is at most $b^{|V|+2} = p$. This completes the proof of the pumping theorem.

Problem 35

- Prove that $L = \{a^n b^n c^n : n \geq 0\}$ is not context-free.
- Prove that $L = \{ww : w \in (a \cup b)^*\}$ is not context-free.

12 Stronger implementations of the pumping idea

Here we consider strings in a CFL language L and *mark* some letters of the string. We show that there is a positive integer p such that if $w \in L$ and p of its characters are marked, then w is "pumpable" in such a way that one of the pumped strings contains a marked character.

Definition 36 *If some letters of a string w are marked and u is a substring of w, then the* **weight** *of u (w.r.t. the marking of w) is the number of marked symbols in u. The weight is denoted $weight(u)$.*

Lemma 37 *(Ogden's Derivation-Pumping Lemma.) Let G be a context-free grammar. Then there is a positive integer p such that for any $w \in L(G)$, if at least p characters of w are marked, then there is a variable A and splitting $w = uvxyz$ of w such that*

1. $weight(v) + weight(y) \geq 1$;
2. $weight(vxy) \leq p$;
3. $S \Rightarrow^* uAz$;
4. $A \Rightarrow^* vAy$;
5. $A \Rightarrow^* x$;
6. $\forall i \geq 0,\ A \Rightarrow^* v^i x y^i$.

Theorem 38 *(Ogden's Pumping Theorem.) Let L be a CFL. Then there is a positive integer p (depending on L) such that for any $w \in L$, if at least p characters of w are marked, then there exist splitting $w = uvxyz$ such that*

1. $weight(v) + weight(y) \geq 1$;
2. $weight(vxy) \leq p$;
3. $\forall i \geq 0,\ uv^i xy^i z \in L$.

Example 39 *Let $L = \{a^i b^j c^k : i \neq j$ and $j = k\}$. Then L is not context-free.*

Solution. Assume that L is a CFL and let p be the pumping number as in Ogden's theorem. Consider

$$w = a^{p!+p}b^pc^p,$$

where $p! = 1 \times 2 \times \ldots \times p$. Mark all b's. Then $w = uvxyz$ for some u, v, x, y, z satisfying Ogden's theorem, in particular, one of v or y overlaps with the b-substring of w. (*Notice, that this overlap is not guaranteed by the original pumping theorem.*)

Claim: For some $m \leq p$, $v = b^m$ and $y = c^m$. Indeed

1. v and y must belong to either a^*, or b^*, or c^*, since otherwise pumping would create strings not even in $a^*b^*c^*$;
2. if $v = a^r$, for some r, then y would be the one to contain marked symbols (they are b's only), and pumping would increase the number of b's without increasing the number of c's; this would yield a string not in L;
3. if $v = c^r$, then y would contain only c's and pumping would increase the number of c's without increasing the number of b's;
4. the remaining option for v is to be b^m; in this case y must be c^m so as to guarantee that the increase of b's equals that of c's.

Now let us pump i times for $i = p!/m + 1$. Then

$$uv^ixy^iz = uvv^{p!/m}xy^{p!/m}yz = uvb^{p!}xc^{p!}yz = a^{p+p!}b^{p+p!}c^{p+p!}.$$

The resulting string is not in L, thus the initial assumption that L is a CFL is incorrect. ▮

Theorem 40 *There exists an inherently ambiguous context-free language.*

Proof. We will show that for any grammar generating $L = \{a^ib^jc^k : i = j \text{ or } j = k\}$, there is a string of the form $a^mb^mc^m$ that can be parsed in two different ways.

Let G be a grammar for L and let $p > 1$ be the pumping number for G as in Ogden's lemma. Let $w_0 = a^pb^pc^{p+p!}$. Mark all b's. By Ogden's lemma, G contains a variable A and a splitting $w_0 = uvxyz$ such that

1. $S \Rightarrow^* uAz$; and
2. $\forall i \geq 0,\ A \Rightarrow^* v^ixy^i$.

The only way to pump without getting strings out of L is if v and y are of the form $v = a^k$ and $y = b^k$ for $1 \leq k \leq p$ (*if v or y contained more than one distinct character, pumping would yield a string not even of the form $a^*b^*c^*$; furthermore, $|v|$ and $|y|$ must be equal, for otherwise pumping down produces a string with all different numbers of a's, b's, and c's*).

Letting $i = p!/k + 1$, we see that $A \Rightarrow a^{p!+k}xy^{p!+k}$, so

$S \Rightarrow^* u\ A\ z \Rightarrow^* u\ a^{p!+k}\ x\ b^{p!+k}\ z \Rightarrow^* u\ a^{p!}\ v\ x\ b^{p!}y\ z \Rightarrow^* a^{p+p!}\ b^{p+p!}\ c^{p+p!}$

Thus, there is a derivation of $a^{p+p!}b^{p+p!}c^{p+p!}$ in which $a^{p!+k}xb^{p!+k}$ is a phrase; (*a phrase is defined to be the yield of a subtree of a parse tree provided the subtree's root is a variable.*)

If we apply the same argument to $w' = a^{p+p!}b^p c^p$ to derive $a^{p+p!}b^{p+p!}c^{p+p!}$, we discover that this derivation contains a phrase $b^{p!+l}x'c^{p!+l}$ for some $l \geq 0$.

Each of the two phrases, $a^{p!+k}xb^{p!+k}$ and $b^{p!+l}x'c^{p!+l}$, contains at least $p! + 1$ b's, and the strings derived contain just $p! + p$ b's; thus each of the phrases contain more than a half of the b's that are derived. Hence, the two phrases overlap. The 1st string contain some a's but no c's; the 2nd string contains some c's but no a's. Therefore neither phrase is a substring of the other, Thus, the parse trees are different, and the grammar is ambiguous. ||

Problem 41 *Prove that the following languages are not CFLs:*

1. $\{a^i b^j c^k : \quad i < j < k\}$; $(\{a^i b^j c^k : \quad j < i = k\})$;
2. $\{a^{i^2}\}$;
3. $\{a^i : i \text{ is composite}\}$;
   Hint: let q be the least prime greater than $(p!+1)!+1$, and show that in fact $q > (p!+1)!+p!+1$. Then let $i = q - p!$ and apply the pumping theorem.
4. $L = \{a^i b^j c^i d^j : i, j \geq 0\}$.

Problem 42 *Prove that* $L = \{a^i b^j c^k : i \neq j$ and $j = k\}$ *satisfies the conclusions of the original pumping theorem (we already know that* L *is not CFL which was proved using Ogden's pumping theorem.)*