Context-Free Languages

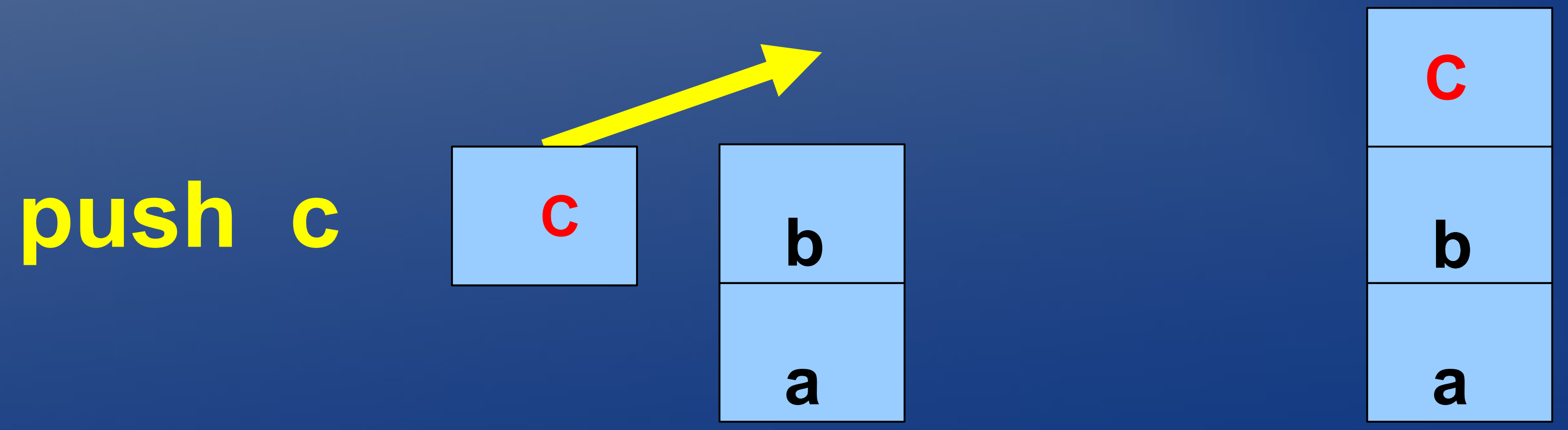
- Pushdown automata (PDA)
- Context-free grammars (CFG)
- Deterministic and nondeterministic PDA
- Equivalence of NPDA and CFGs
- Ambiguous grammars and inherently ambiguous languages
- Normal forms and cleaning up "dirty" grammars
- Closure properties and a new pumping lemma
- Other topics, e.g., parsing with lex & yacc, L-Systems, .

Not all languages are regular

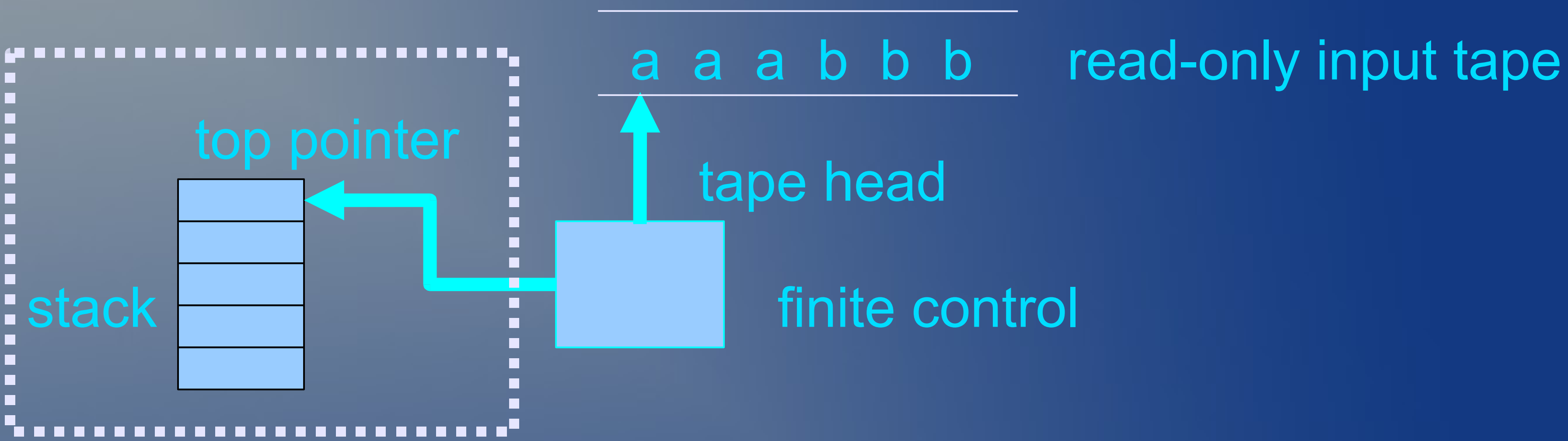
- For example, there is no DFA for { $a^n b^n$: n>0 }, proven using the Pumping Lemma for regular languages
- We will meet other non-regular languages, many of which are very useful
- We need something more powerful than DFA/NFA, something more expressive than regular expressions and regular grammars

What's the problem with DFAs?

- We can only store information on the states, hence, we only have a finite amount of memory
- If we want a more powerful machine, we have to add (infinite) memory
- One of the simplest storage devices we can use is a STACK, along with the stack operations PUSH and POP (as well as TOP = check the top without popping, and NOP = no operation)



PDA, or pushdown automata

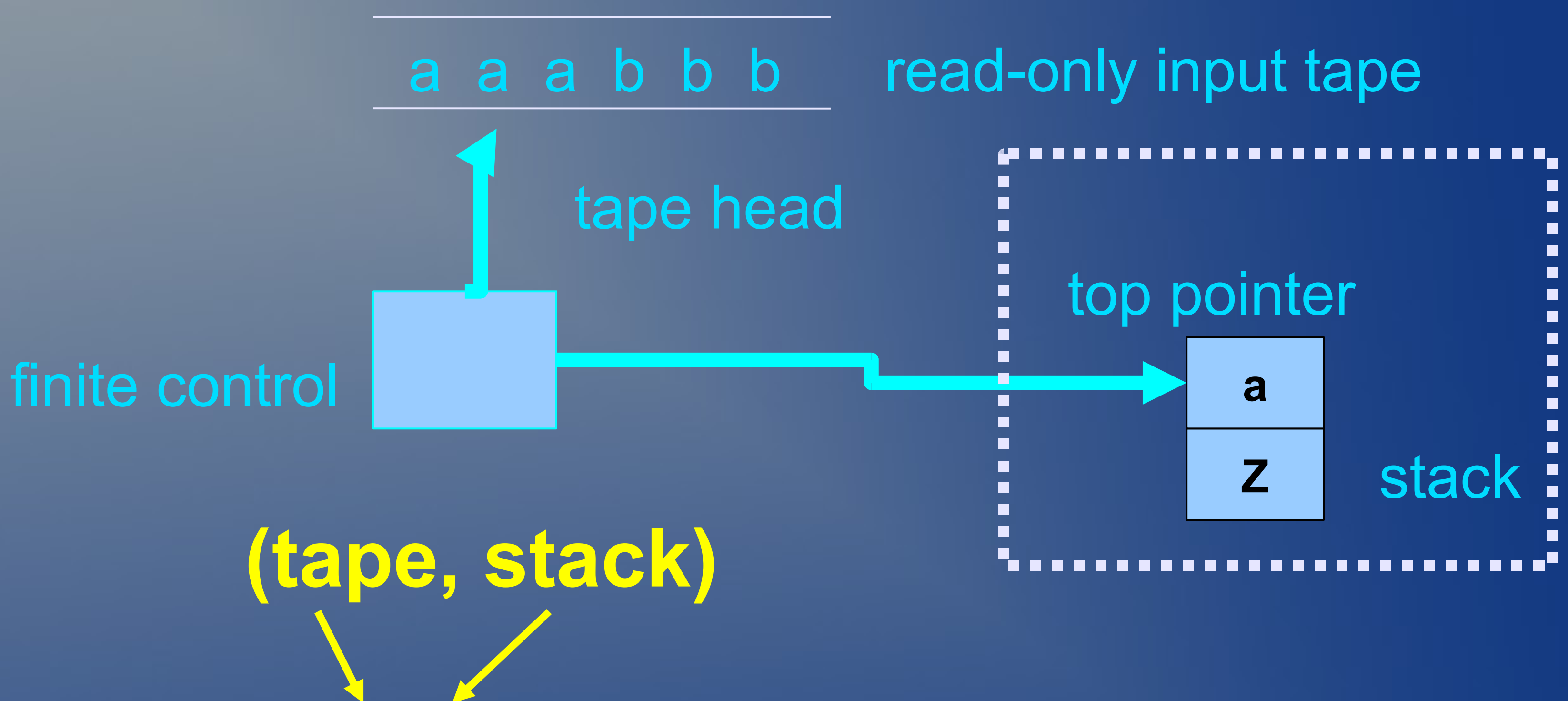


- Addition of a stack for storage significantly increases the power of the automaton
- We assume that the stack size is unbounded, it can never be full

PDA for { $a^n b^n$: n > 0 }

- We want a machine that will accept all strings in L = {ab, aabb, aaabbb, … }, and only these strings
- Idea is to **push the a's** onto the stack as we read them, and **pop them one by one for every matching b**
- If we initially **put a special symbol Z at the bottom of the stack**, we must again see Z if we have seen an equal number of a's and b's

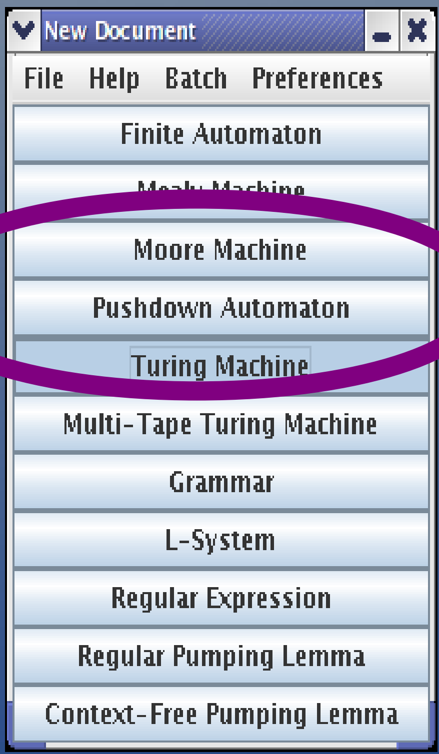
PDA for { a^nb^n : n > 0 }

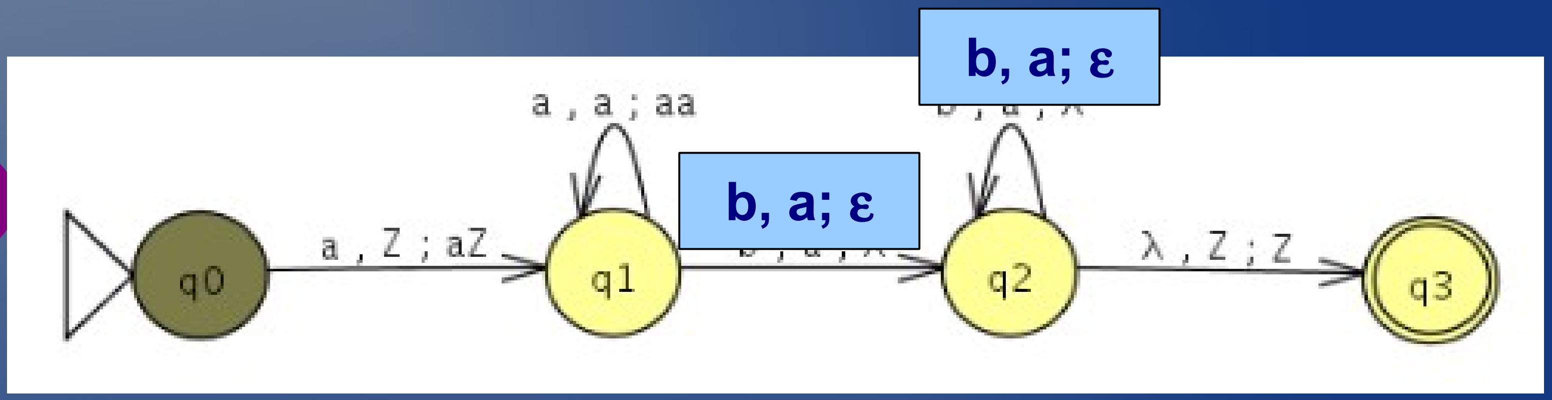


- if (a, Z) or (a, a) then "push a"
- if (b, a) then "pop"
- if (ε, Z) then "go accept the string"

PDA for $\{ a^n b^n : n > 0 \}$ on JFLAP

(current symbol on the tape,
symbol on the top of the stack;
replacement symbols for the top)





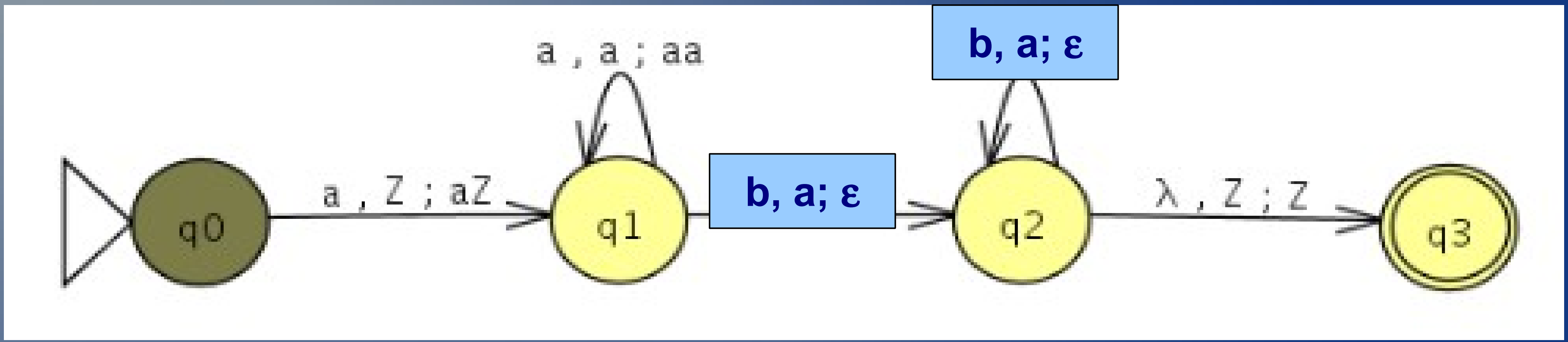
- if (a, Z) or (a, a) then “push a”
- if (b, a) then “pop”
- if (ε, Z) then “go accept the string”

Basic structure of a PDA

$M = (Q , \Sigma , \Gamma , \delta , q_o , Z_o , F)$

- Q = finite set of states
- Σ = input alphabet
- Γ = stack alphabet
- δ = transition function, $\delta : Q \times \Sigma \times \Gamma \rightarrow Q \times \Gamma^*$
- q_o = start/initial state
- Z_o = initial/bottom symbol for the stack
- F = set of final/accepting states

Tracing the execution: Instantaneous Descriptions for PDAs



(current state, remaining input, stack config):

$(q_0, aabb, Z)$ # (q_1, abb, aZ) # (q_1, bb, aaZ)

(q_2, b, aZ) # (q_2, ε, Z) # (q_3, ε, Z)

Two forms of acceptance

- A PDA accepts the string x **by final state** if (q_0, x, Z) eventually leads to ($\mathbf{p}$, ε, ?) for some final state p
- A PDA accepts the string x **by empty stack** if (q_0, x, Z) eventually leads to (p, ε, $\boldsymbol{\varepsilon}$)
- A PDA **accepts the language** L if every string in L is accepted (and every other string is rejected)
- These two forms of acceptance can be shown to be **equivalent**, that is, a PDA in one form can always be converted into the other form

Other non-regular languages which can be accepted by some PDA

Exercises: Construct PDAs for the ff. languages:

- $\{ a^n b^{2n} : n>0 \}$ = { abb, aabbbb, aaabbbbbb, … }
- $\{ a^n b^{n+1} : n>0 \}$ = { abb, aabbb, aaabbbb, … }
- **palindromes** = { a, b, aa, bb, aaa, aba, bab, … }
- an **equal number** of a's and b's (in any order)
  = { ab, ba, aabb, abab, baba, bbaa, ... }
- **balanced pairs** of parentheses
  = { ( ), (( )), ( ) ( ), ((( ))), (( )) ( ), … }
  = { ab, aabb, abab, aaabbb, aabbab, … }

Context-Free Grammars & Context-Free Languages

- A grammar is a set of **string-rewriting rules** for producing a set of strings
- Example: the context-free grammar below generates the language $\{ a^n b^n : n>0 \} = \{ ab, aabb, aaabbb, \ldots \}$

  S → ab (basis)

  S → aSb (recursive rule)
- Often abbreviated as **S → ab | aSb**
- If L is generated by a CFG, L is said to be a **Context-Free Language**

Derivations

- A string x can be derived from the start symbol S, if x can be generated by successive applications of the production rules of the grammar, for example:

  S → ab (rule 1) basis

  S → aSb (rule 2) recursive rule

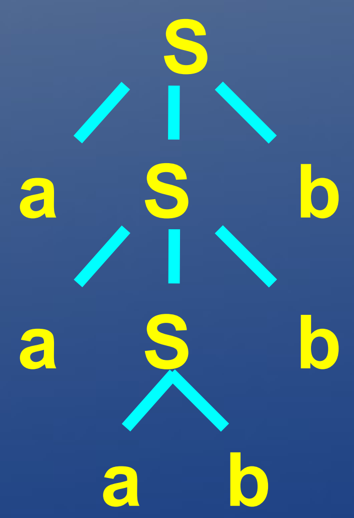
To derive the string "aaabbb":

$$\mathbf{S} \Rightarrow_2 a\mathbf{S}b \Rightarrow_2 a\,a\mathbf{S}b\,b \Rightarrow_1 aa\,ab\,bb$$

Parse trees (or Derivation trees)

- A parse tree (or derivation tree) is a tree with the **start symbol as the root**, and the target **string forming the leaves** of the tree

$S \rightarrow ab \mid aSb$



Non-leaf nodes are variables, children are the symbols on the right-hand-side of some valid production rule

a parse tree for “aaabbb”

CFGs, formal definition

- A context-free grammar is a structure

G = (V, T, P, S) where

  - V is a finite set of variables (or non-terminals)
  - T is a finite set of terminals (or the alphabet Σ)
  - P is a finite set of production rules
  - S is the start variable

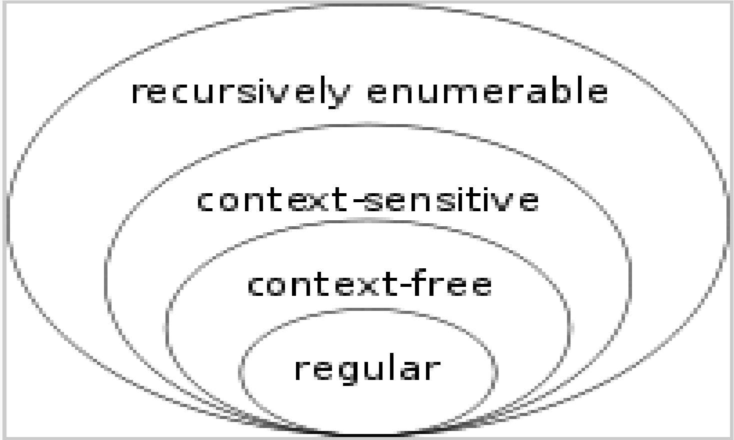
- Our previous grammar is more formally defined as

$$G = (\underbrace{\{ S \}}_{\textbf{Variables}}, \underbrace{\{ a, b \}}_{\textbf{Terminals}}, \underbrace{\{ \mathbf{S \rightarrow ab}, \mathbf{S \rightarrow aSb} \}}_{\textbf{Production Rules}}, \underbrace{S}_{\textbf{Start}})$$

Chomsky's hierarchy of grammars

- Regular grammars (simplest, weakest)
  - Right-hand side always has the form **$T^*(V+\varepsilon)$**, i.e., it contains at most one variable and if present this variable forms the suffix of the RHS
  - Example: $S \rightarrow abS \mid a \mid \varepsilon$ what is L(G)?
- **Context-free grammars**
  - LHS is still a single variable; RHS has the form **$(T+V)^*$**
  - Example: $S \rightarrow \varepsilon \mid a \mid b \mid aSa \mid bSb$ what is L(G)?
- Context-sensitive grammars
- Unrestricted grammars (most expressive)

Chomsky hierarchy of grammars

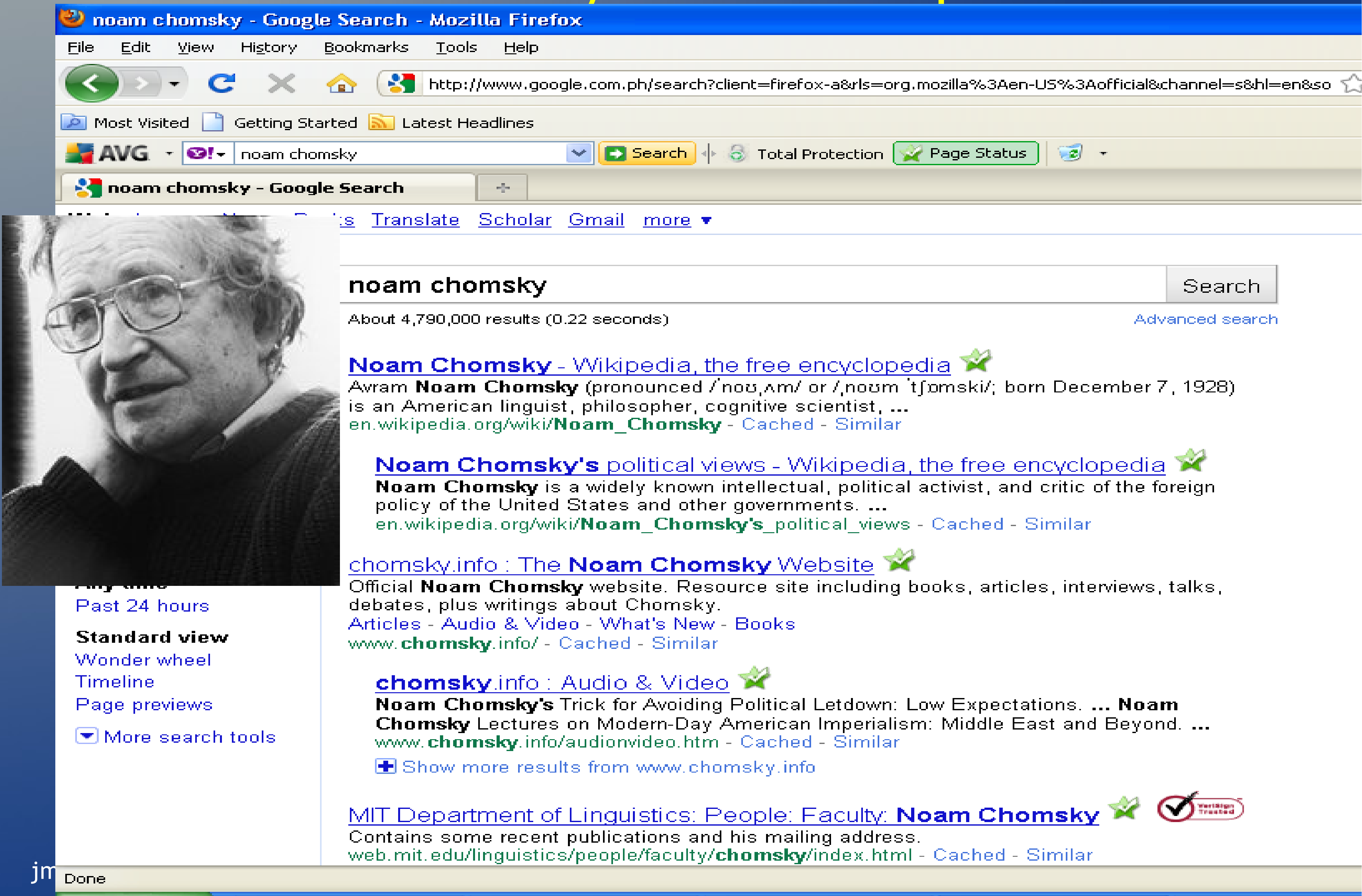


Set inclusions described by the Chomsky hierarchy.

Grammar	Languages	Automaton	Production rules (constraints)
Type-0	Recursively enumerable	Turing machine	$\alpha \rightarrow \beta$ (no restrictions)
Type-1	Context-sensitive	Linear-bounded non-deterministic Turing machine	$\alpha A \beta \rightarrow \alpha \gamma \beta$
Type-2	Context-free	Non-deterministic pushdown automaton	$A \rightarrow \gamma$
Type-3	Regular	Finite state automaton	$A \rightarrow a$ and $A \rightarrow aB$

Figure and table from Wikipedia

Noam Chomsky – the rebel professor



Exercises on grammar construction

- Design CFGs for:
  - Odd-length palindromes over { a, b }
  - Even-length palindromes over { a, b }
  - Arbitrary-length palindromes
  - Palindromes that begin and end with an 'a'
  - Equal number of a's and b's
  - Balanced parentheses over { (,) }
  - Palindromes with a double-b

Ambiguous grammars

- A grammar is **ambiguous** if there is more than one parse tree for any string x in the language
- A grammar is **non-ambiguous** if every string in the language has a unique parse tree

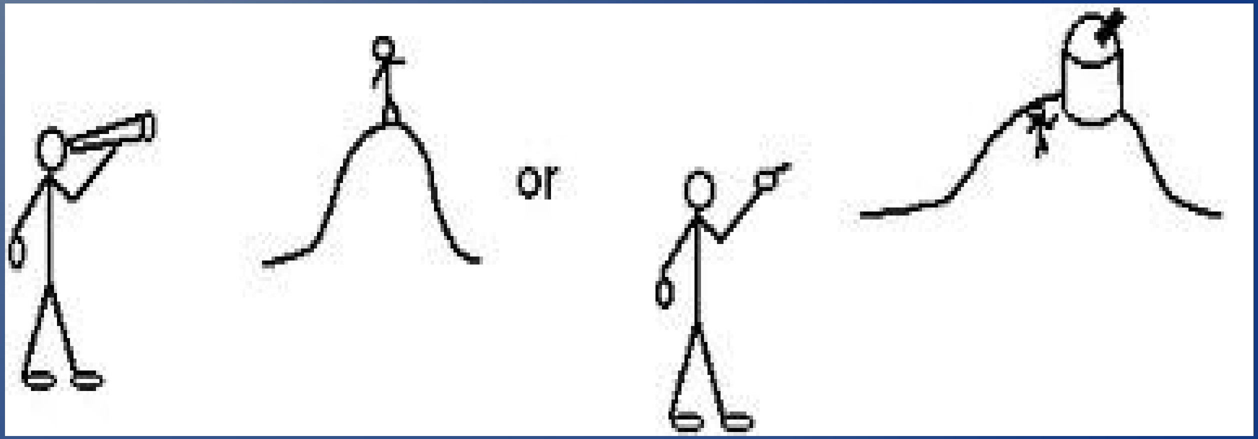
- **S → ab | aSb** is non-ambiguous
- S → **a | S+S** is ambiguous

  T = Σ = {a, +}; draw 2 parse trees for "a+a+a"

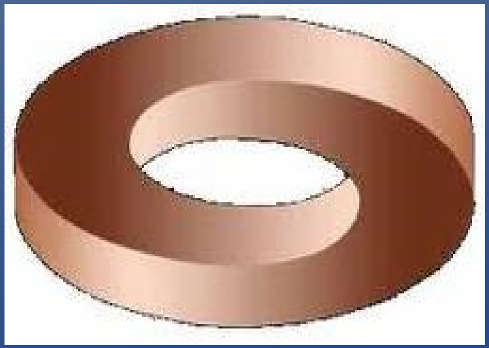
jmsamaniego@uplb, revised 2010

Ambiguity in natural languages

- Time flies like an arrow.
- Fruit flies like a banana.
- The man on the hill saw the boy with a telescope.



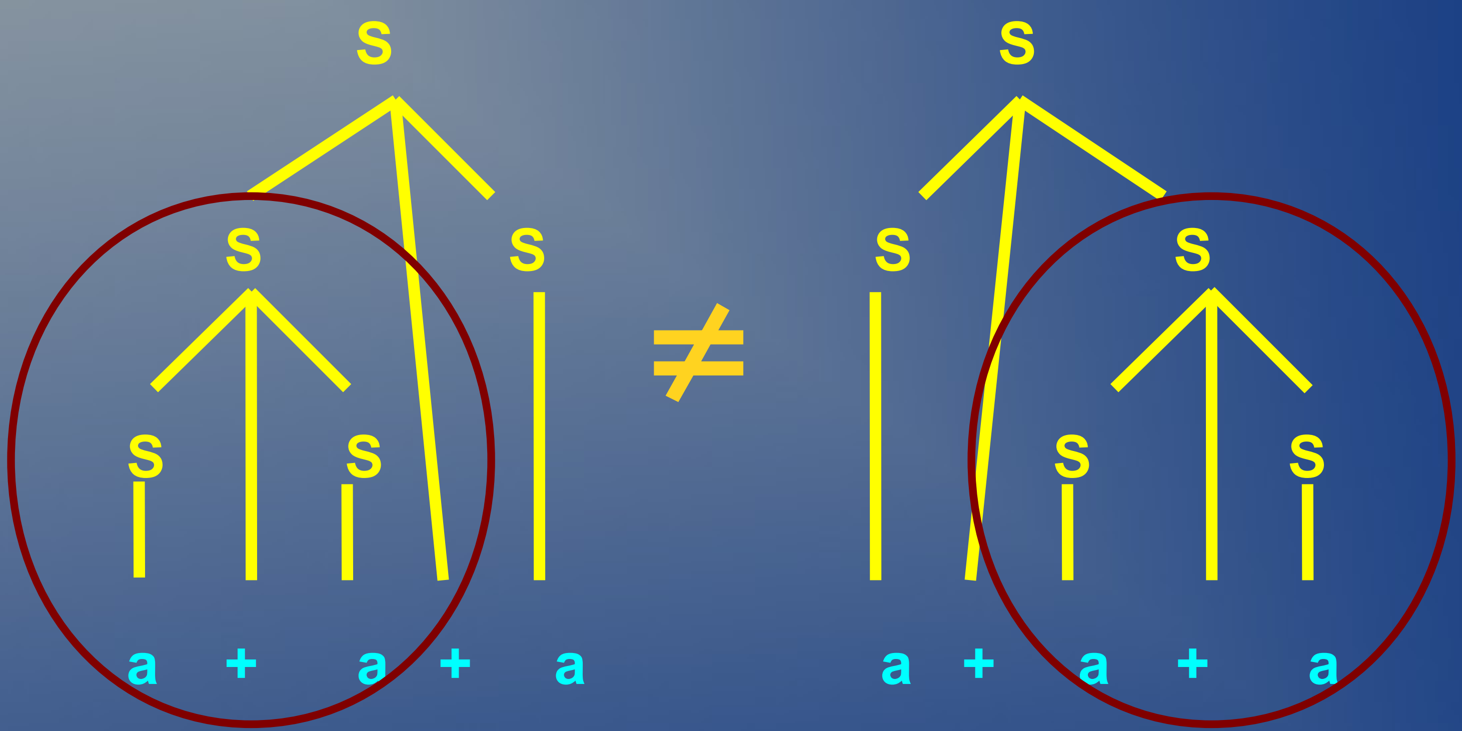
.



Most optical illusions are ambiguous images.

The grammar $S \rightarrow a \mid S+S$ is ambiguous

- The string "a+a+a" can be derived in at least 2 ways:



- The problem becomes an arithmetic evaluation problem if we consider the related grammar
  $S \rightarrow 1 \mid S - S$ and the string "1 – 1 – 1" using the alphabet $T = \Sigma = \{ 1, - \}$

Removing ambiguity

- G_1: $\mathbf{S \rightarrow a \mid S - S}$ is ambiguous
- G_2: $\mathbf{S \rightarrow a \mid S - a}$ is non-ambiguous
- The two grammars generate the same language, i.e., $L(G_1) = L(G_2)$
- G_2 is better, because it forces the rule on left-associativity, removing the ambiguity in G_1

Left-associative vs Right-associative operators

- Addition, subtraction, multiplication and division are commonly treated as left-associative by most programming languages
  - 8/2/2 is evaluated as (8/2)/2 and not as 8/(2/2)
- Most programming languages which support an exponentiation operator treat it as right-associative
  - 2^2^3 is evaluated as 2^(2^3) and not as (2^2)^3, e.g., try the python expression 2**2**3
  - Exercise: Construct a non-ambiguous grammar equivalent to $S \rightarrow 2 \mid 3 \mid S \wedge S$ that supports right-associativity (use $\Sigma = T = \{ 2, 3, \wedge \}$)

Enforcing precedence in expression grammars

- Consider the expression grammar

$$S \rightarrow 0 \mid 1 \mid S + S \mid S * S \mid (S)$$

- Another source of ambiguity is operator precedence
- In how many ways can “1+1*0” be parsed?
- Is there a non-ambiguous grammar that generates the same language?

A non-ambiguous expression grammar

$S \rightarrow E$

$E \rightarrow E + T \mid T$ (expressions)

$T \rightarrow T * F \mid F$ (terms)

$F \rightarrow 0 \mid 1 \mid (E)$ (factors)

- Every string in the language has a unique parse tree
- Construct the parse trees for “1+1*0” and “(1+1)*0”
- Exercise: Modify the grammar to allow binary-valued operands and a right-associative exponentiation operator with higher precedence than multiplication, e.g., 10+10^10^11*10 = ?

Dangling-else ambiguity

- $S \rightarrow B$ | **if** C **then** S | **if** C **then** S **else** S

  $B \rightarrow$ block; | { some other block of statements }

  $C \rightarrow$ (cond) | (some other condition)

if ($cond_1$) **then**
    if ($cond_2$) **then** $block_1$;
else $block_2$;
	if ($cond_1$) **then**
	    if ($cond_2$) **then** $block_1$;
	    else $block_2$;

Exercise: Design a non-ambiguous version of this grammar that associates an **else**-clause to the nearest **if**.

Dangling-else ambiguity

- Consider the grammar for the if-then-[else] construct found in many languages

S → B | **if** C **then** S | **if** C **then** S **else** S

B → block; | { some other block of statements }

C → (cond) | (some other condition)

- In how many ways can you parse the string below:
- **if** (cond) **then** **if** (cond) **then** block; **else** block;

Inherently-ambiguous languages

- A language L is inherently ambiguous if every grammar G_i for L is ambiguous
- Example: $L = \{ a^m b^m c^n \} \cup \{ a^m b^n c^n \}$, $m, n > 0$
- Sample strings are aabbcccc, abbbccc, abc
- A CFG for L is given by

  $S \rightarrow XC \mid AY$

  $X \rightarrow ab \mid aXb$ $\quad\quad$ $C \rightarrow c \mid cC$

  $A \rightarrow a \mid aA$ $\quad\quad$ $Y \rightarrow bc \mid bYc$
- Show that "aabbcc" has 2 different parse trees
- Explain why every possible grammar for L is ambiguous?

Grammars for language structures

- Nested tags in markup languages

```
\begin{enumerate}
  \item …
•     \item ...
  \begin{itemize}
    \item ....
    \item ....
  \end{itemize}
\end{enumerate}
```

$S \to L_1 \mid L_2$

$L_1 \to B_1\ L\ E_1$

$L_2 \to B_2\ L\ E_2$

$L \to \text{\item} \mid \text{\item}\ L \mid \text{\item}\ S$

$B_1 \to$ \begin{enumerate}

$E_1 \to$ \end{enumerate}

$B_2 \to$ \begin{itemize}

$E_2 \to$ \end{itemize}

Grammars for program structures

- Arithmetic expressions in assignment statements
- Boolean expressions in conditions
- Regular expressions (formal and egrep-style)
- Lambda expressions in LISP
- Nested control structures in block-structured languages

A sample LISP-like function and part of a LISP grammar

```
(define (factorial n)
    (if (= n 0)
        1
        (* n (factorial (- n 1)))))
```

S → (define Head Body)

Head → (FName ParameterList)

Expression → (Operator Operands)

Operator → if | = | + | - | * | … | FuncCall

Operands → Number | Expression

A typical block-structured language

Statement → Assignment **|** Block **|**
    If-statement **|** While-statement

Assignment → Var **=** Expression

Block → **{** Statement-list **}**

Statement-list → ε **|** Statement**;** Statement-list

If-statement → **if (** Condition **)** Statement **|**
    if (Condition **)** Statement **else** Statement

While-statement → **while (** Condition **)** Statement **|**
    do Statement **while (** Condition **)**

Simplifying grammars

- Chomsky Normal Form
  - Right-hand side is restricted to a single terminal or a pair of variables, i.e., T + VV
- Greibach Normal Form
  - Right-hand side is restricted to a terminal followed by zero or more variables, i.e., TV*
- Elimination of useless symbols, unit productions $V \rightarrow W$, and empty productions $V \rightarrow \varepsilon$ (for non-empty languages)

Chomsky Normal Form

- Right-hand side is restricted to a single terminal or a pair of variables, i.e., T + VV
- Example: Chomskyize the grammar: $S \rightarrow ab \mid aSb$
- Idea is to convert all into variables first and group by 2s
- Chomsky Normal Form:

$S \rightarrow AB$	$A \rightarrow a$
$S \rightarrow XB$	$B \rightarrow b$
$X \rightarrow AS$	

- Note that parse trees of grammars in CNF are always binary trees

Greibach Normal Form

- Right-hand side is restricted to a <u>terminal followed by zero or more variables, i.e., TV*</u>
- Example: Greibachize the grammar $S \rightarrow a \mid S+S$
- Greibach Normal Form:

  $S \rightarrow a$

  $S \rightarrow aPS$ (but this makes + right-associative)

  $P \rightarrow +$
- When in GNF, an input string of length n can always be derived in n steps; the grammar can also be converted into an NPDA with no ε-moves

Equivalence of PDAs and CFGs

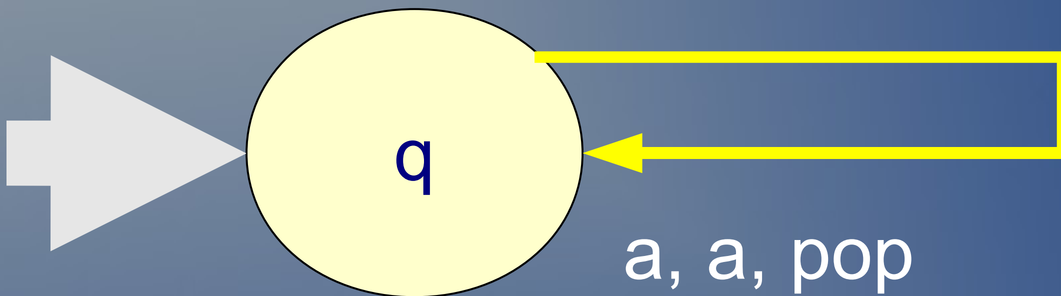
- Analogous to Kleene's theorem in regular languages
- Every NPDA can be converted into a CFG, every CFG can be converted into a NPDA
- We use NPDA (for non-deterministic PDA) because <u>deterministic PDA are weaker than the non-deterministic PDA</u>

CFG to NPDA

- Every CFG can be converted to a nondeterministic PDA
- Use a single state q; stack alphabet $\Gamma = V \cup T$; we accept by empty stack
- The initial stack symbol will be S, the start variable
- For every terminal symbol a in Σ,
  add the transition $\delta($ q, a, a $) = ($ q, pop $)$
- For every empty production $A \rightarrow \varepsilon$, add the transition
  $\delta($ q, ε, A $) = ($ q, pop $)$
- For every rule $A \rightarrow B_1B_2 \ldots B_n$, add the transition
  $\delta($ q, ε, A $) = ($ q, {pop; push B_n; push B_{n-1}; … push B_1} $)$

CFL to NPDA example

- $S \rightarrow \varepsilon \mid aSb$, $L(G) = \{ a^n b^n : n \geq 0 \}$

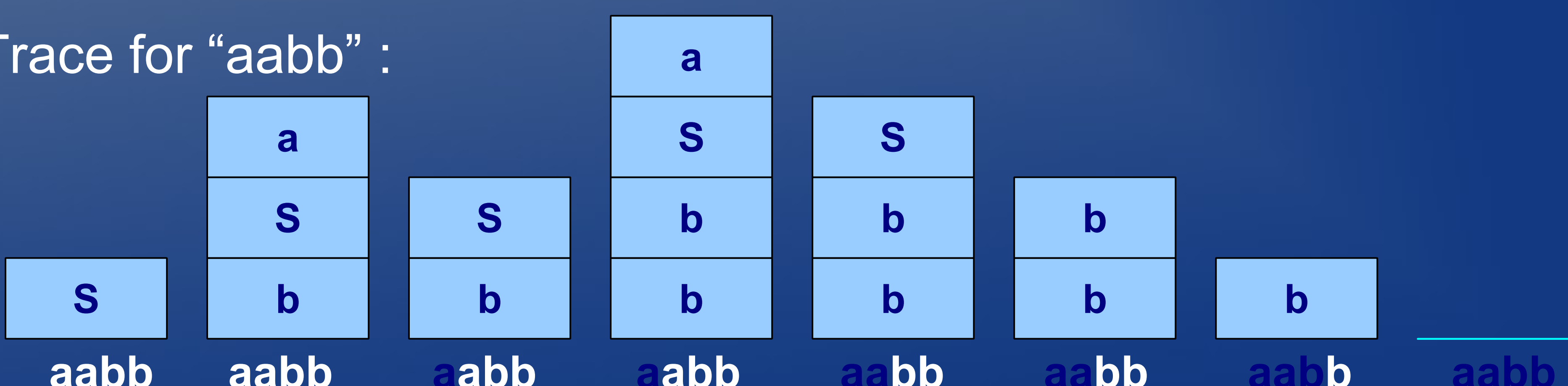


a, a, pop
b, b, pop
ε, S, pop
ε, S, { pop, push b, push S, push a }

In JFLAP, these would be
a, a, ε
b, b, ε
ε, S, ε
ε, S, aSb

Stack alphabet
$\Gamma = \{ S, a, b \}$
Initial stack symbol
$Z_0 = S$

Trace for "aabb" :



Closure Properties for Context-Free Languages

- CFLs are closed under union, concat and Kleene star

$S \rightarrow A \mid B$	union
$S \rightarrow AB$	concat
$S \rightarrow \varepsilon \mid AS$	Kleene star

- CFLs are not closed under complementation nor general intersection. Why?
- Intersection of a regular language with a CFL results in a CFL
- CFLs are closed under reversals, homomorphism (string substitutions), inverse homomorphisms

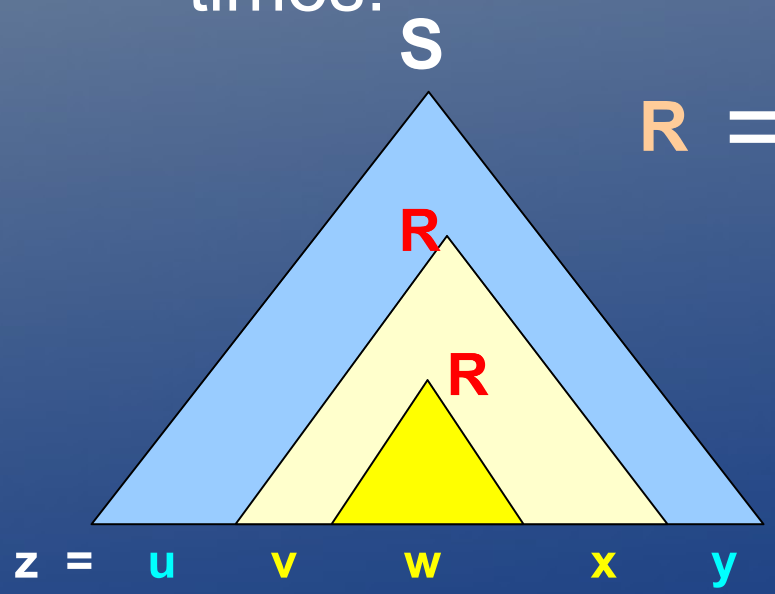
A pumping lemma for CFLs

Let L be an <u>infinite</u> context-free language. There is a positive integer n such that for all strings z in L, with $|z| \geq n$, z can be written in the form z = **uvwxy**, such that the following properties hold:

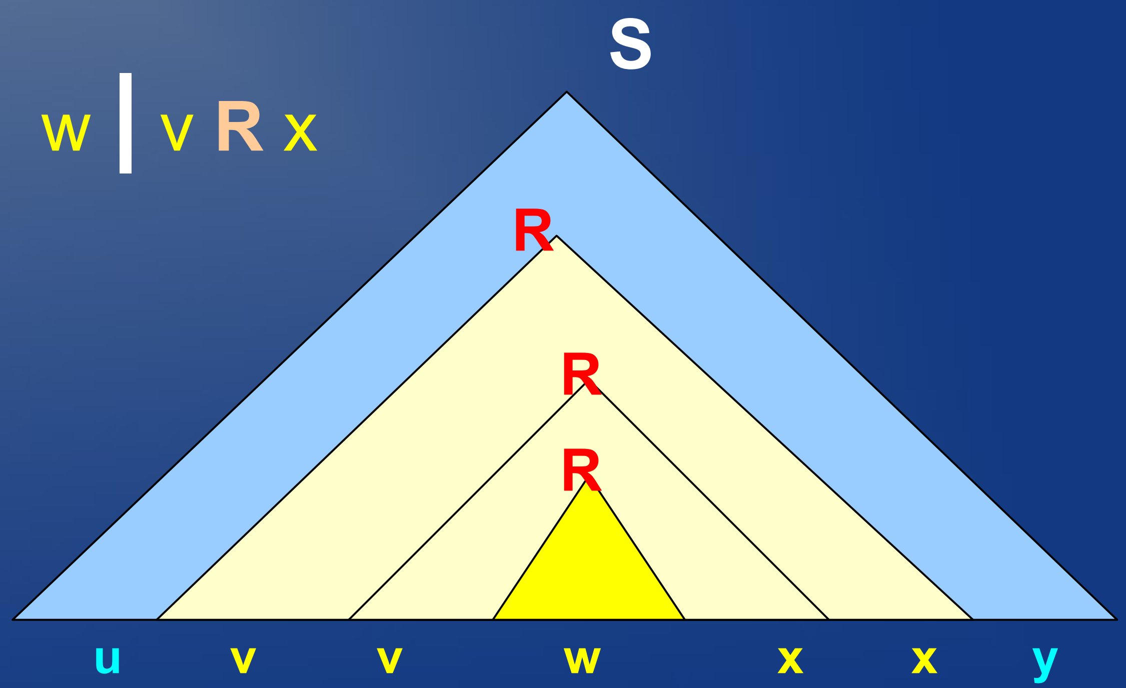
$|vx| \geq 1$, (v and x cannot be both empty)
$|vwx| \leq n$,
$u\, \mathbf{v}^k\, w\, \mathbf{x}^k\, y$ is in L, for all $k \geq 0$.

Main idea in proof of the pumping lemma

- We use the pigeonhole principle on the nodes of the parse tree.
- If the input string z is long enough, then some interior node (say, variable R) must be repeated.
- We can "pump" by expanding R any number of times.



$$R \Rightarrow^* w \mid v\,R\,x$$



Some languages are not context-free

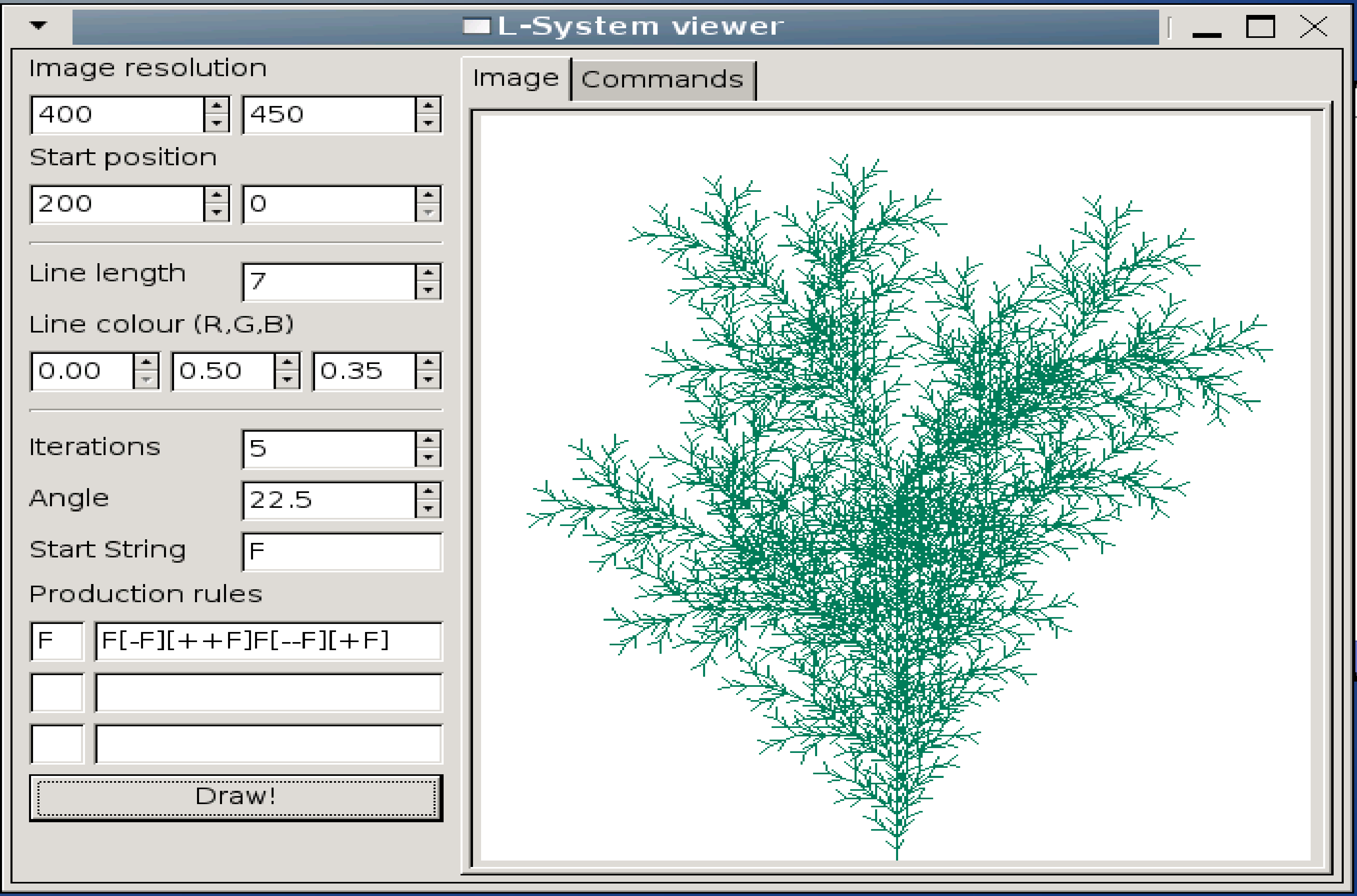
- Some languages cannot be recognized by a PDA or by a CFG – a single stack is insufficient and the memory model is still too weak
- One of the simplest non-CFL is
  $L = \{ a^n b^n c^n : n > 0 \} = \{ abc, aabbcc, \ldots \}$

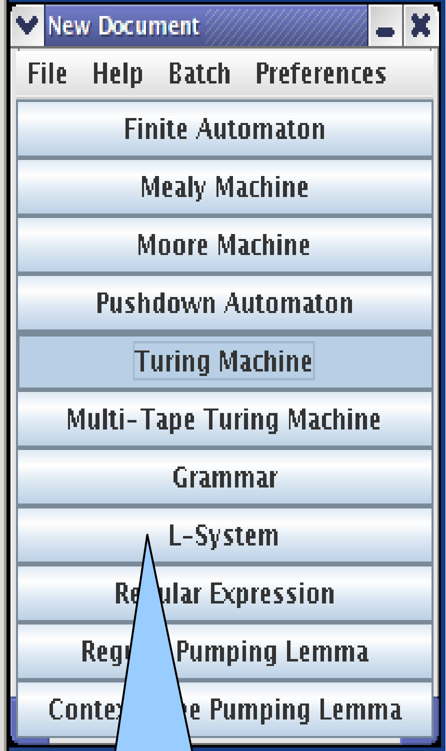
  (Proof is by contradiction using the pumping lemma.)
- What if we had access to two stacks? Can we now recognize L?

Lindenmayer systems

grammar-like structures for drawing fractals

Beyond Context-Free Grammars

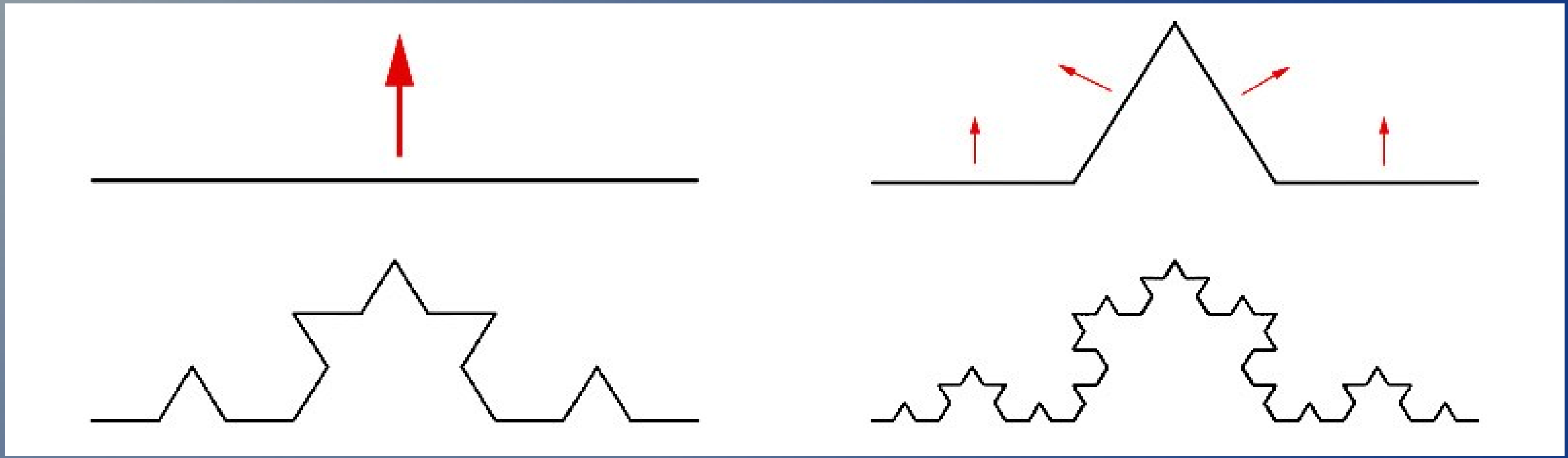




JFLAP has L-systems

http://www.haskell.org/gtk2hs/gallery/ox-practical/L_System_viewer

Grammars and Fractals



Representation as Lindenmayer system

The Koch Curve can be expressed by a rewrite system (Lindenmayer system).

Alphabet : F

Constants : +, −

Axiom : F++F++F

Production rules:

F → F−F++F−F

Here, *F* means "draw forward", + means "turn right 60°", and − means "turn left 60°" (see turtle graphics).

From Wikipedia