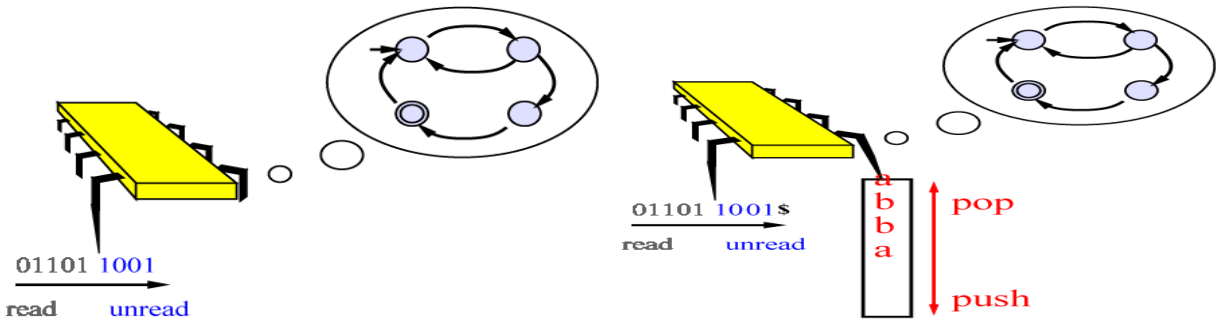
Computational Models - Lectures 4 and 5

- Context Free Grammars
- Pumping Lemma for context free languages
- Non context free languages
- Closure properties for CFL
- Push Down Automata (PDA)
- ~~Equivalence of CFGs and PDAs~~
- Algorithmic issues for CFL



- Sipser's book, 2.1, 2.2 & 2.3

Short Overview of the Course

So far we saw

- finite automata,
- regular languages,
- regular expressions,
- Myhill-Nerode theorem
- pumping lemma for regular languages.

We now introduce stronger machines and languages with more expressive power:

- pushdown automata,
- context-free languages,
- context-free grammars,
- pumping lemma for context-free languages.

Formal Definitions

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

- V is a finite set of variables,
- Σ is a finite set of terminals,
- R is a finite set of rules: each rule is a variable and a finite string of variables and terminals.
- S is the start symbol.

Formal Definitions

- If u and v are strings of variables and terminals,
- and $A \to w$ is a rule of the grammar, then
- we say uAv yields uwv, written $uAv \Rightarrow uwv$.

We write $u \overset{*}{\Rightarrow} v$ if $u = v$ or

$$u \Rightarrow u_1 \Rightarrow \ldots \Rightarrow u_k \Rightarrow v.$$

for some sequence $u_1, u_2, \ldots, u_k$.

Definition: The language of the grammar is

$$\left\{ w \in \Sigma^* \mid S \overset{*}{\Rightarrow} w \right\} .$$

Example 1

Consider $G_4 = (V, \{a, b\}, R, S)$.

R (Rules): $S \to aSb \mid SS \mid \varepsilon$.

Some words in the language: $aabb$, $aababb$.

Q.: But what is this language?

Hint: Think of parentheses.

Example 2

Consider $G_5 = (V, \{a, b\}, R, S)$.

R (Rules): $S \to aSb \mid bSa \mid \varepsilon$.

Some words in the language: $abba$, $aabaabaa$.

Q.: But what is this language?

$L(G_5) = \{ww^R | w \in \{a, b\}^*\}$

Example 3

Consider $G_6 = (V, \{a, b\}, R, S)$.

R (Rules):
$S \to aB \mid bA \mid \varepsilon$.
$A \to a \mid aS \mid bAA$.
$B \to b \mid bS \mid aBB$.

Some words in the language: $aababb$, $baabba$.

Q.: But what is this language?

$L(G_6) = \{w \in \{a, b\}^* | \#_a(w) = \#_b(w)\}$

Example $L(G_6) = \{w \in \{a,b\}^* | \#_a(w) = \#_b(w)\}$

- What do we need to show?!
- (1) If w generated by G_6 then $\#_a(w) = \#_b(w)$.
- (2) If $\#_a(w) = \#_b(w)$ then w generated by G_6.
- Claim: If $S \stackrel{*}{\Rightarrow} \alpha$ then $\#_a(\alpha) + \#_A(\alpha) = \#_b(\alpha) + \#_B(\alpha)$.
- Claim: For any w let $k = \#_a(w) - \#_b(w)$. Then:
  - (i) if $k = 0$ then $S \stackrel{*}{\Rightarrow} wS$,
  - (ii) if $k > 0$ then $S \stackrel{*}{\Rightarrow} wA^k$,
  - (iii) if $k < 0$ then $S \stackrel{*}{\Rightarrow} wB^{|k|}$
- Are we done?!

Designing Context-Free Grammars

No recipe in general, but few rules-of-thumb

- If CFG is the union of several CFGs, rename variables (not terminals) so they are disjoint, and add new rule $S \rightarrow S_1 \mid S_2 \mid \ldots \mid S_i$.
- For a regular language, grammar "follows" a DFA for the language. For initial state q_0, make R_0 the start variable. For state transition $\delta(q_i, a) = q_j$ add rule $R_i \rightarrow aR_j$ to grammar. For each final state q_f, add rule $R_f \rightarrow \varepsilon$ to grammar. This is called a linear grammar.
- For languages (like $\{0^n\#1^n | n \geq 0\}$), with linked substrings, a rule of form $R \rightarrow uRv$ is helpful to force desired relation between substrings.

Simple Closure Properties

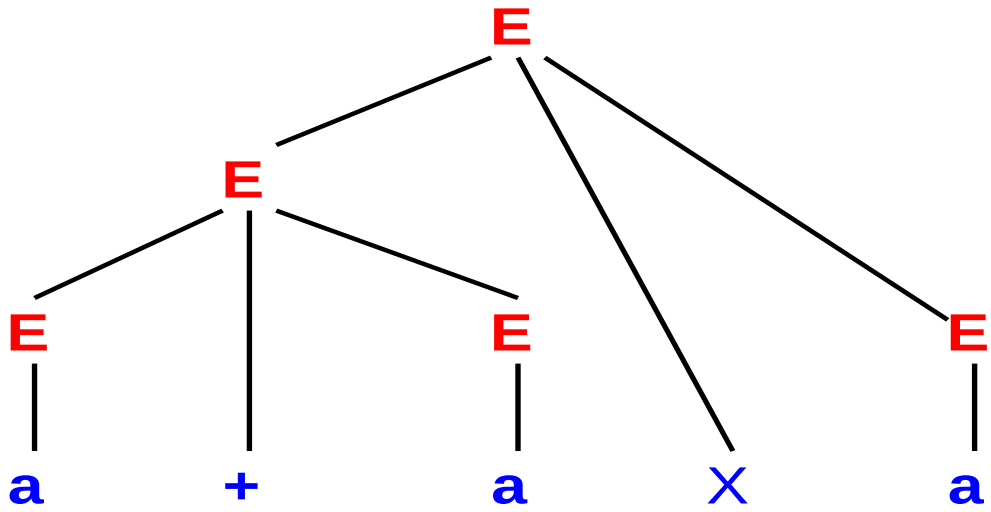
- Regular languages are closed under
  - union
  - concatenation
  - star
- Context-Free Languages are closed under
  - union : $S \rightarrow S_1 \mid S_2$
  - concatenation $S \rightarrow S_1 S_2$
  - star $S_{new} \rightarrow \varepsilon \mid S_{old} \mid S_{new} S_{new}$

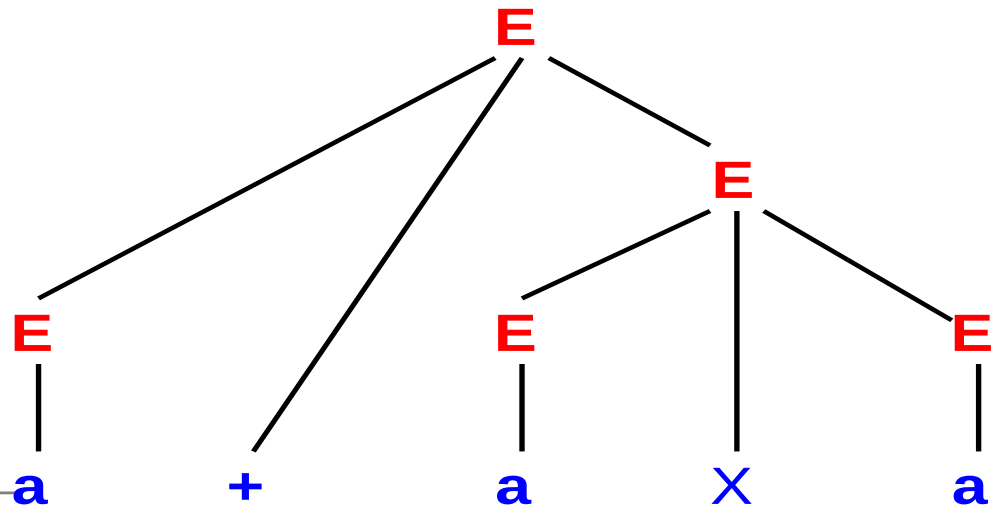
More Closure Properties

- Regular languages are also closed under
  - complement (replace accept/non-accept states of DFA)
  - intersection $\left(L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}} \right)$.
- What about complement and intersection of context-free languages?
  - Absolutely not clear at this point (but will become crystal clear soon) . . .

Ambiguity in CFLs

Grammar: $E \rightarrow E{+}E \mid E{\times}E \mid (E) \mid a$





Arithmetic Example

Consider (V, Σ, R, E) where

- $V = \{E, T, F\}$
- $\Sigma = \{a, +, \times, (,)\}$

Rules:

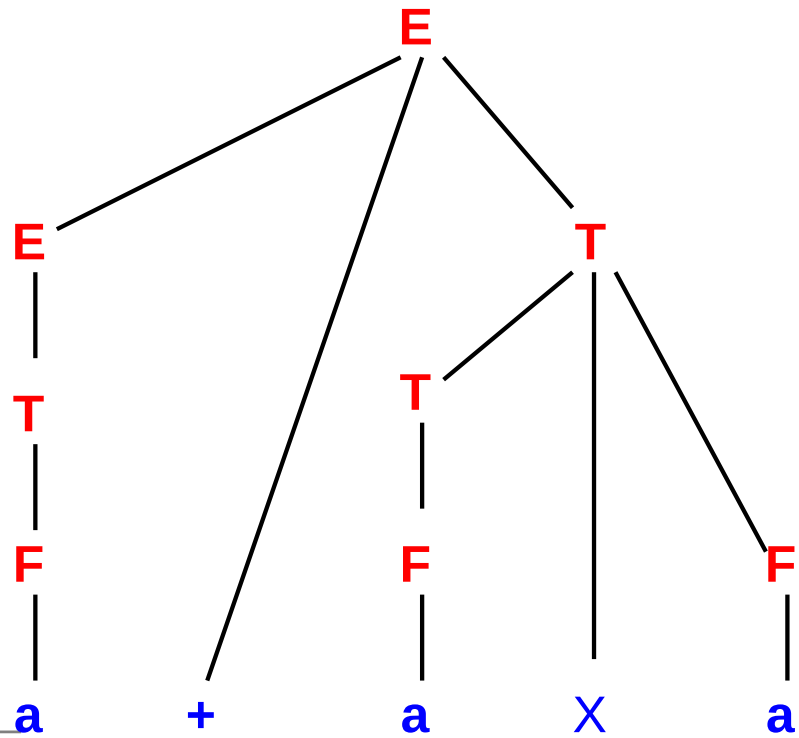
$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T \times F \mid F \\ F &\rightarrow (E) \mid a \end{aligned}$$

Some strings generated by the grammar:
$a + a \times a$ and $(a + a) \times a$.
Same arithmetic expressions, just not ambiguous.

$E =$ expression, $T =$ term, $F =$ factor.

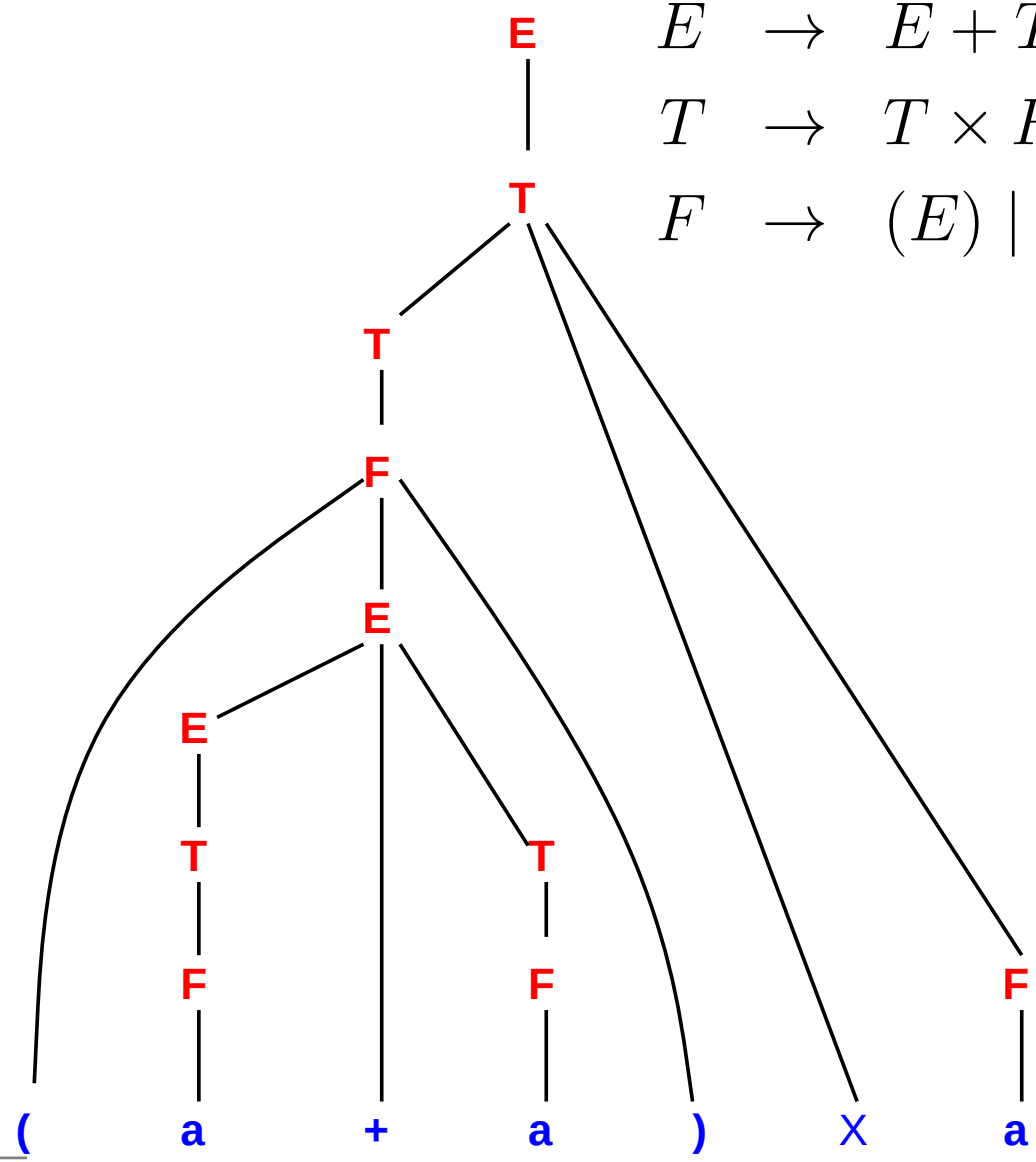
Parse Tree for $a + a \times a$

$$
\begin{aligned}
E &\rightarrow E + T \mid T \\
T &\rightarrow T \times F \mid F \\
F &\rightarrow (E) \mid a
\end{aligned}
$$



Parse Tree for $(a + a) \times a$

$$
\begin{aligned}
E &\rightarrow E + T \mid T \\
T &\rightarrow T \times F \mid F \\
F &\rightarrow (E) \mid a
\end{aligned}
$$



Ambiguity

We say that a string w is derived ambiguously from grammar G if w has two or more parse trees that generate it from G.

Ambiguity is usually not only a syntactic notion but also semantic, implying multiple meanings for the same string. Think of $a + a \times a$ from last grammar.

It is sometime possible to eliminate ambiguity by finding a different context free grammar generating the same language. This is true for the arithmetic expressions grammar.

Some languages, *e.g.* $\{1^i 2^j 3^k \mid i = j \text{ or } j = k\}$ are inherently ambiguous.

Non-Context-Free Languages

- The pumping lemma for finite automata and Myhill-Nerode theorem are our tools for showing that languages are not regular.
- We will now show a similar pumping lemma for context-free languages.
- It is slightly more complicated . . .

Pumping Lemma for CFL

Also known as the $uvxyz$ Theorem.

Theorem: If A is a CFL , there is an ℓ (critical length), such that if $s \in A$ and $|s| \geq \ell$, then $s = uvxyz$ where

- for every $i \geq 0$, $uv^i xy^i z \in A$
- $|vy| > 0$, (non-triviality)
- $|vxy| \leq \ell$.

Basic Intuition

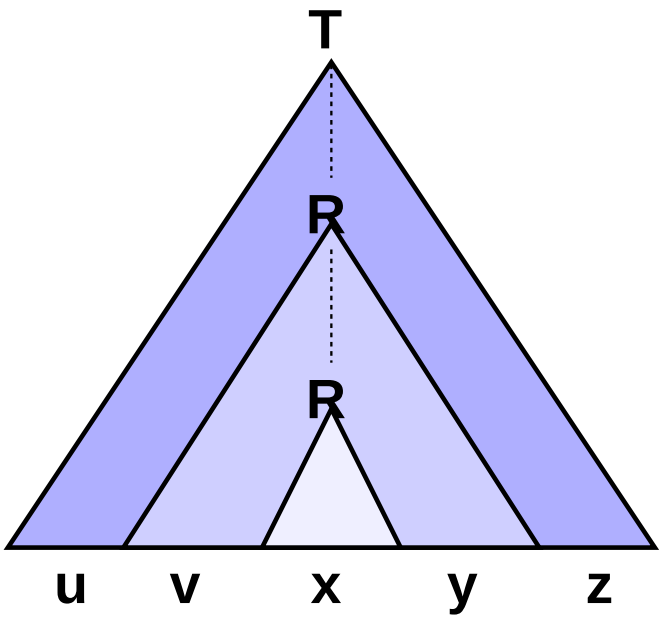
Let A be a CFL and G its CFG.

Let s be a "very long" string in A.

Then s must have a "tall" parse tree.

And some root-to-leaf path must repeat a symbol.

ahmmm,. . . why is that so?



Proof

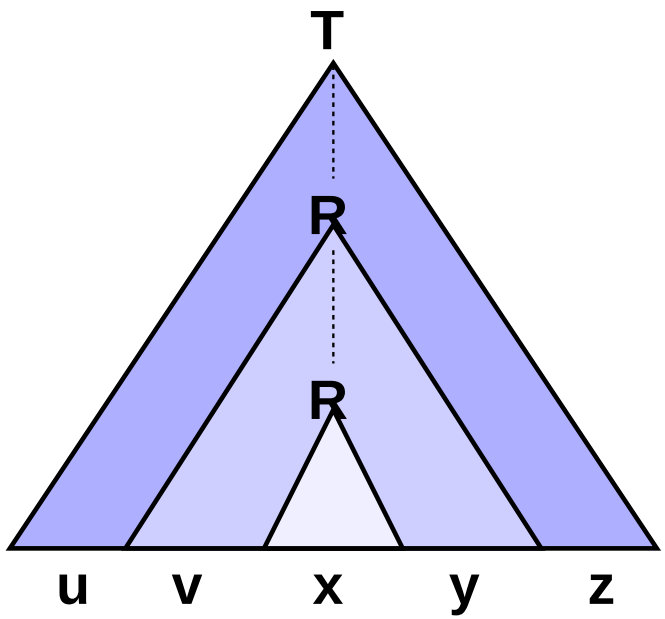
- let G be a CFG for CFL A.
- let b be the max number of symbols in right-hand-side of any rule.
- no node in parse tree has $> b$ children.
- at depth d, can have at most b^d leaves.
- let $|V|$ be the number of variables in G.
- set $\ell = b^{|V|+2}$.

Proof (2)

- let s be a string where $|s| \geq \ell$
- let T be parse tree for s with fewest nodes
- T has height $\geq |V| + 2$
- some path in T has length $\geq |V| + 2$
- that path repeats a variable R

Proof (3)

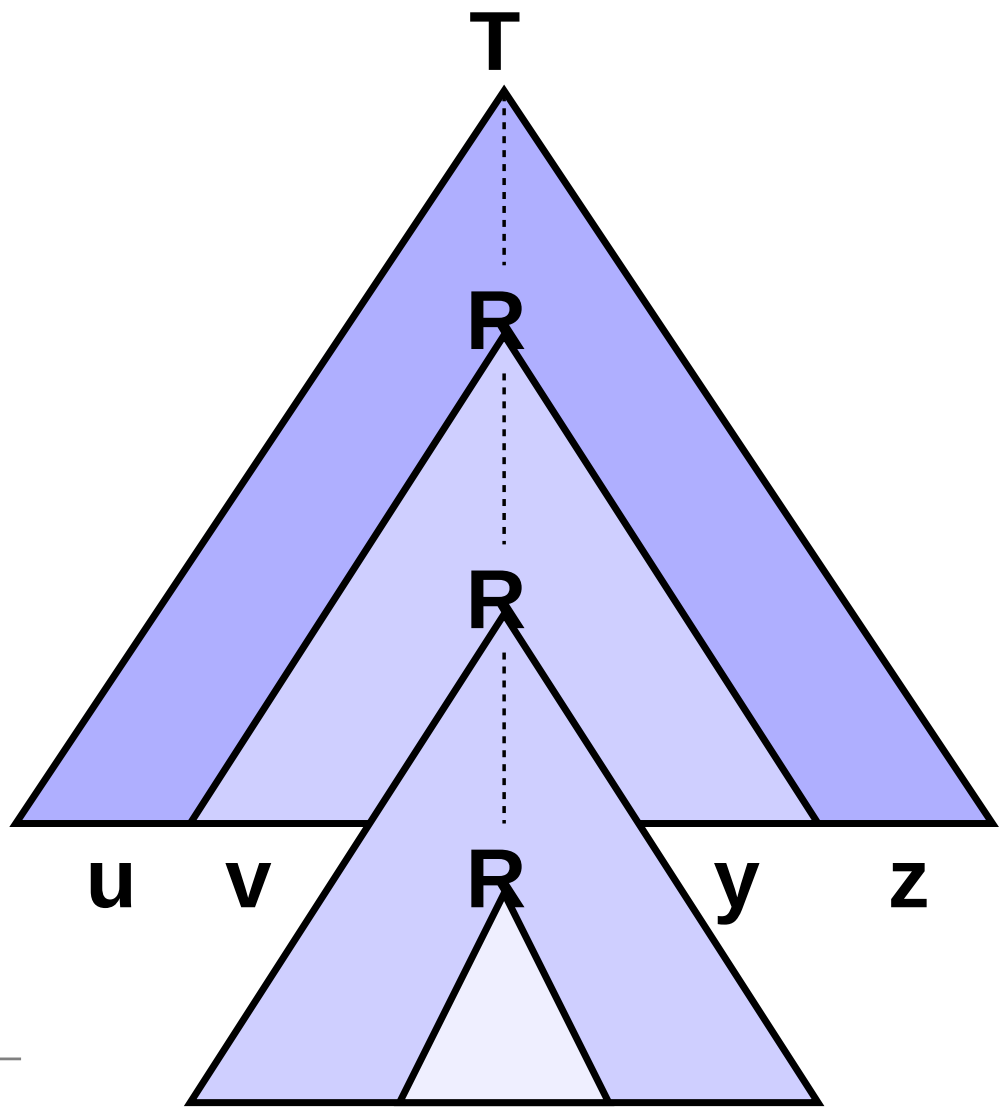
Split $s = uvxyz$



- each occurrence of R produces a string
- upper produces string vxy
- lower produces string x

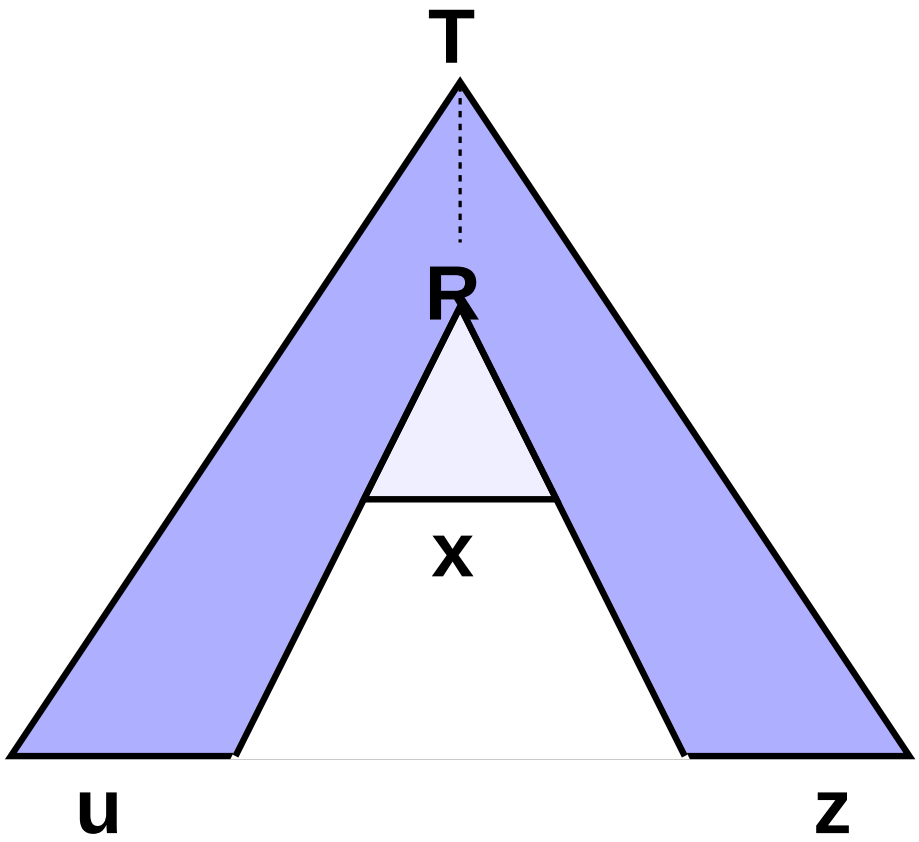
Proof

Replacing smaller by larger yields uv^ixy^iz, for $i > 0$.



Proof (4)

Replacing larger by smaller yields uxz.



Together, they establish:

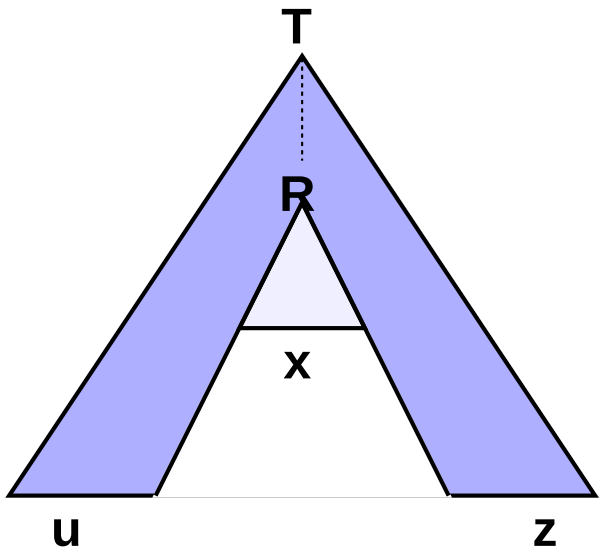
- for all $i \geq 0$, $uv^i xy^i z \in A$

Proof (5)

Next condition is:

- $|vy| > 0$ (out of $uvxyz$)

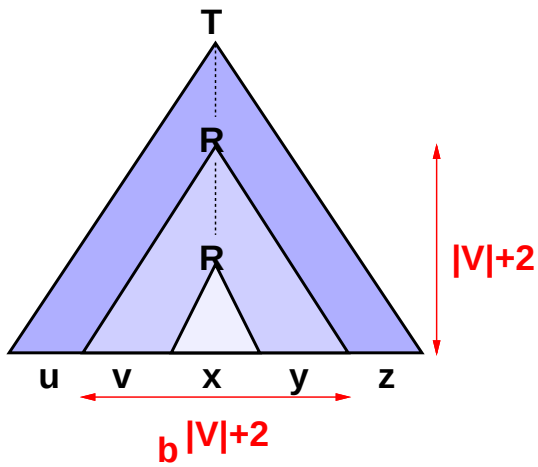
If v and y are both ε, then



is a parse tree for s with fewer nodes than T, contradiction.

Proof (6)

Final condition is $|vxy| \leq \ell$:



- the upper occurrence of R generates vxy.
- can choose symbols such that both occurrences of R lie in bottom $|V| + 1$ variables on path.
- subtree where R generates vxy is $\leq |V| + 2$ high.
- strings by subtree at most $b^{|V|+2} = \ell$ long. ♠

Non CFL Example

Theorem: $B = \{a^n b^n c^n\}$ is not a CFL.

Proof: By contradiction.

Let ℓ be the critical length.

Consider $s = a^\ell b^\ell c^\ell$.
If $s = uvxyz$, neither v nor y can contain

- both a's and b's, or
- both b's and c's,

because otherwise uv^2xy^2z would have out-of-order symbols.
But if v and y contain only one letter, then uv^2xy^2z has an imbalance!

Non CFL Example (2)

The language $C = \{a^i b^j c^k | 0 \leq i \leq j \leq k\}$ is not context-free.
Let ℓ be the critical length, and $s = a^\ell b^\ell c^\ell$.

Let $s = uvxyz$

- neither v nor y contains two distinct symbols, because otherwise uv^2xy^2z would have out-of-order symbols.
- vxy cannot be all the same letter (why?)
- $|vxy| \leq \ell$, so either
  - v contains only a's and y contains only b's, but then uv^2xy^2z has too few c's.
  - v contains only b's and y contains only c's. but then uv^0xy^0z has too many a's.

Non CFL Example (3)

The language $D = \{ww | w \in \{0,1\}^*\}$ is not context-free.
Let $s = 0^\ell 1^\ell 0^\ell 1^\ell$ As before, suppose $s = uvxyz$.
Recall that $|vxy| \leq \ell$.

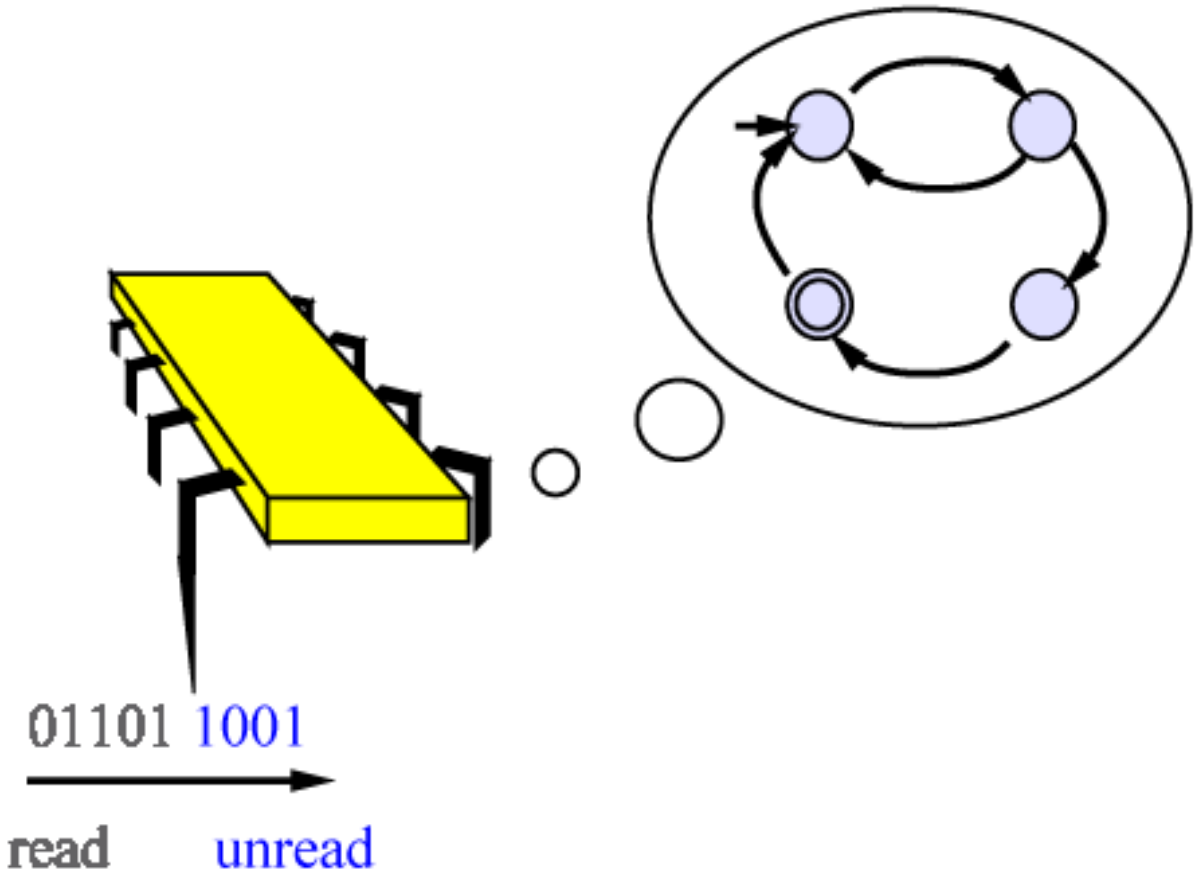
- if vxy is in the first half of s, uv^2xy^2z moves a 1 into the first position in second half.
- if vxy is in the second half, uv^2xy^2z moves a 0 into the last position in first half.
- if vxy straddles the midpoint, then pumping *down* to uxz yields $0^\ell 1^i 0^j 1^\ell$ where i and j cannot both be ℓ.

Note that $D = \{ww^R | w \in \{0,1\}^*\}$ is CFL.

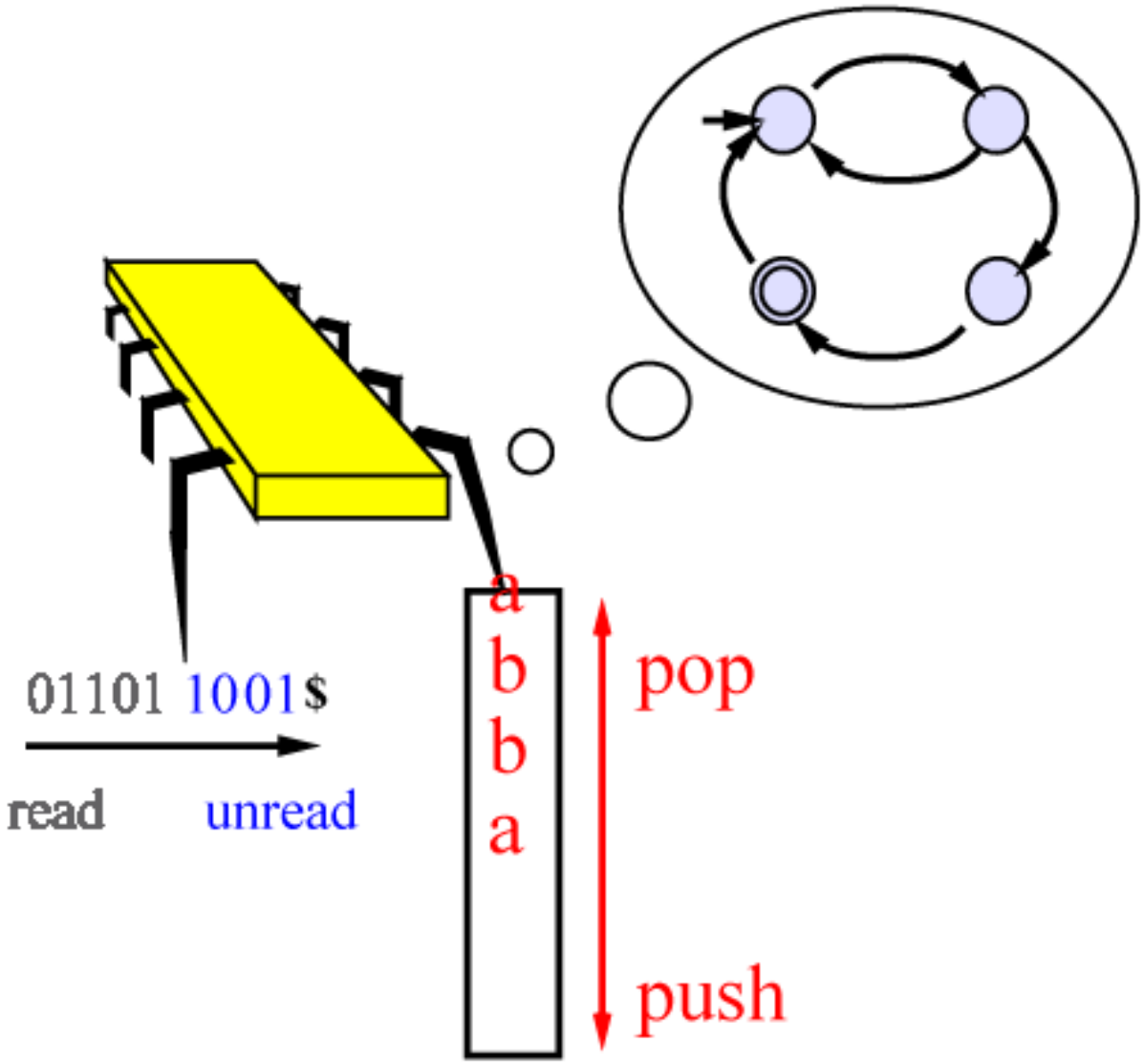
String Generators and String Acceptors

- Regular expressions are string generators – they tell us how to generate all strings in a language L
- Finite Automata (DFA, NFA) are string acceptors – they tell us if a specific string w is in L
- CFGs are string generators
- Are there string acceptors for Context-Free languages?
- YES! Push-down automata

A Finite Automaton



A PushDown Automaton



An Example

Recall that the language $\{0^n 1^n \mid n \geq 0\}$ is not regular.
Consider the following PDA:

- Read input symbols
- For each 0, push it on the stack
- As soon as a 1 is seen, pop a 0 for each 1 read
- Accept if stack is empty when last symbol read
- Reject if
  - Stack is non-empty when end of input symbol read
  - Stack is empty but input symbol(s) still exist,
  - 0 is read after 1.

PDA Configuration

- A Configuration of a Push-Down Automata is a triplet
- (<state>, <remaining input string>, <stack>).

- A configuration is like a snapshot of PDA progress.
- A PDA computation is a sequence of successive configurations, starting from start configuration.
- The string describing the stack has top of the stack on the left (technical item).

Comparing PDA and Finite Automata

- PDA may be deterministic or non-deterministic.
- Unlike finite automata, non-determinism adds power: There are some languages accepted only by non-deterministic PDAs.

Transition function δ looks different than DFA or NFA cases, reflecting stack functionality.

PDA Transition Function

Denote input alphabet by Σ and stack alphabet by Γ.

- the domain of the transition function δ is
  - current state: Q
  - next input, if any: Σ_ε (=$\Sigma \cup \{\varepsilon\}$)
  - stack symbol popped, if any: Γ_ε(=$\Gamma \cup \{\varepsilon\}$)
- and its range is
  - new state: Q
  - stack symbol pushed, if any: Γ_ε
  - non-determinism: $\mathcal{P}$(two components above)
- $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}(Q \times \Gamma_\varepsilon)$

Formal Definitions

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

- Q is a finite set called the *states*,
- Σ is a finite set called the *input alphabet*,
- Γ is a finite set called the *stack alphabet*,
- $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}(Q \times \Gamma_\varepsilon)$ is the *transition function*,
- $q_0 \in Q$ is the *start state*, and
- $F \subseteq Q$ is the set of *accept states*.

Conventions

- It will be convenient to be able to know when the stack is empty, but there is no built-in mechanism to do that.
  - Solution:
  - Start by pushing $ onto stack.
  - When you see it again, stack is empty.

Semi Formal Definitions

- A pushdown automaton (PDA) M accepts a string x if there is a computation of M on x (a sequence of state and stack transitions according to M's transition function and corresponding to x) that leads to an accepting state.
- The language accepted by M is the set of all strings x accepted by M.
- While a non-deterministic PDA, M, may have many computations on x, a deterministic PDA has just one.

Notation

- Transition $a, b \rightarrow c$ means
  - if read a from input
  - and pop b from stack
  - then push c onto stack
- Meaning of ε transitions:
  - if $a = \varepsilon$, don't read inputs
  - if $b = \varepsilon$, don't pop any symbols
  - if $c = \varepsilon$, don't push any symbols

PDA Languages

The Push-Down Automata Languages, L_{PDA}, is the set of all languages that can be described by some PDA:

- $L_{PDA} = \{L : \exists \text{ PDA } M \wedge L[M] = L\}$

We already know $L_{PDA} \supsetneq L_{DFA}$, since every DFA is just a PDA that ignores the stack.

- $L_{CFG} \subseteq L_{PDA}$?
- $L_{PDA} \subseteq L_{CFG}$?
- $L_{PDA} = L_{CFG}$!!!

PDA Transition Function – Reminder

Denote input alphabet by Σ and stack alphabet by Γ.

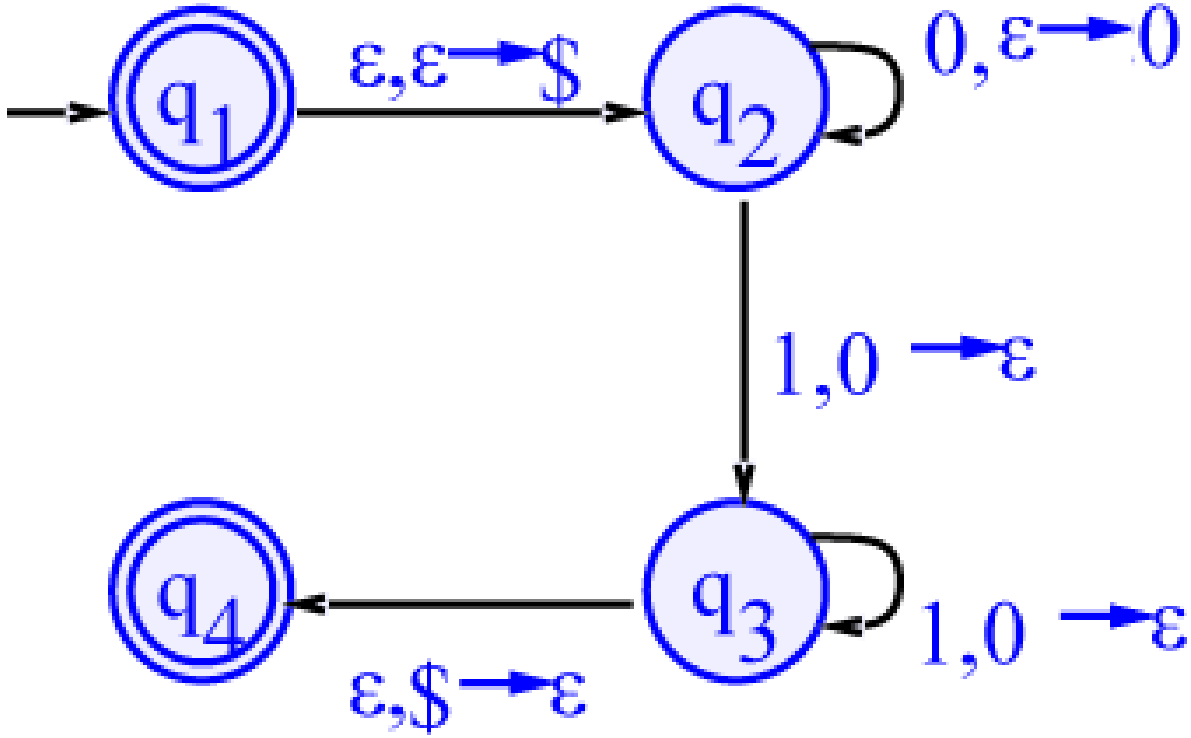
- the domain of the transition function δ is
  - current state: Q
  - next input, if any: Σ_ε (=$\Sigma \cup \{\varepsilon\}$)
  - stack symbol popped, if any: Γ_ε(=$\Gamma \cup \{\varepsilon\}$)
- and its range is
  - new state: Q
  - stack symbol pushed, if any: Γ_ε
  - non-determinism: $\mathcal{P}$(two components above)
- $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}(Q \times \Gamma_\varepsilon)$

The Transition Function for Deterministic PDAs

- the domain of the transition function δ is
  - current state: Q
  - next input, if any: Σ_ε (=$\Sigma \cup \{\varepsilon\}$)
  - stack symbol popped, if any: Γ_ε(=$\Gamma \cup \{\varepsilon\}$)
- and its range is
  - next state and stack symbol pushed, if any: $Q \times \Gamma_\varepsilon$
  - or nothing ($\emptyset$), meaning no continuation.
- $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \rightarrow (Q \times \Gamma_\varepsilon) \cup \{\emptyset\}$
- $\forall\, q \in Q, a \in \Sigma, \gamma \in \Gamma_\varepsilon$, if $\delta(q, \varepsilon, \gamma) \neq \emptyset$, then $\delta(q, a, \gamma) = \emptyset$ (epsilon input transition prevents "real" input transition).
- $\forall\, q \in Q, a \in \Sigma_\varepsilon, \gamma \in \Gamma$, if $\delta(q, a, \varepsilon) \neq \emptyset$, then $\delta(q, a, \gamma) = \emptyset$ (epsilon stack transition prevents "real" stack transition).

Example: Deterministic PDA

The PDA



accepts $\{0^n 1^n | n \geq 0\}$.

Another Example

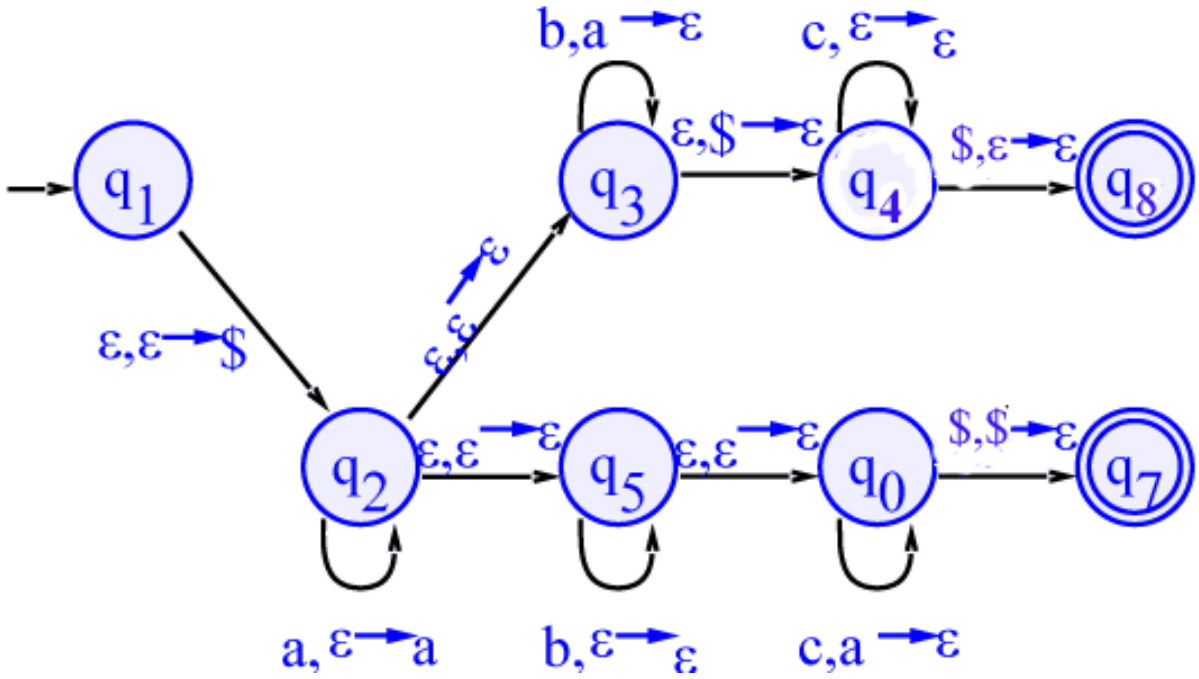
A PDA that accepts

$$\left\{a^i b^j c^k | i,j,k > 0 \text{ and } i = j \text{ or } i = k\right\}$$

Informally:

- read and push a's
- either pop and match with b's
- or else pop and match with c's
- a non-deterministic choice!

Another Example (cont.)



This PDA accepts

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

Another Example (cont.)

A PDA that accepts $\{a^i b^j c^k | i, j, k > 0$ and $i = j$ or $i = k\}$

- Note: non-determinism is essential here!
- Unlike finite automata, non-determinism does add power.
- Let us try to think how a proof could go,
- ⋮
- and realize it does not seem trivial or immediate.
- We will later (maybe) give a proof that the language $L = \{x^n y^n | n \geq 0\} \cup \{x^n y^{2n} \mid n \geq 0\}$ is accepted by a non-deterministic PDA but not by a deterministic one.

Another PDA Example

A palindrome is a string w satisfying $w = w^{\mathcal{R}}$.

- "Madam I'm Adam"
- "Dennis and Edna sinned"
- "Red rum, sir, is murder"
- "Able was I ere I saw Elba"
- "In girum imus nocte et consumimur igni" (Latin: "we go into the circle by night, we are consumed by fire".)
- "$\nu\iota\psi o\nu\ \alpha\nu o\mu\eta\mu\alpha\tau\alpha\ \mu\eta\ \mu o\nu\alpha\nu\ o\psi\iota\nu$"
- Palindromes also appear in nature. For example as DNA restriction sites – short genomic strings over $\{A, C, T, G\}$, being cut by (naturally occurring) restriction enzymes.

A PDA that Recognizes Palindromes

- On input x, the PDA starts pushing x into stack.
- At some point, PDA guesses that the mid point of x was reached.
- Pops and compares to input, letter by letter.
- If end of input occurs together with emptying of stack, accept.
- This PDA accepts palindromes of *even length* over the alphabet (all lengths is easy modification).
- Again, non-determinism (at which point to make the switch) seems necessary.

The CFG–PDA Equivalence Theorem

Theorem: A language is context free if and only if some pushdown automata accepts it.

This time (unlike the regular expression vs. regular languages theorem), the proofs of both the "if" part and the "only if" part are non trivial.

Proof sketch follows.

If Part

Theorem: If a language is context free, then some pushdown automaton accepts it.

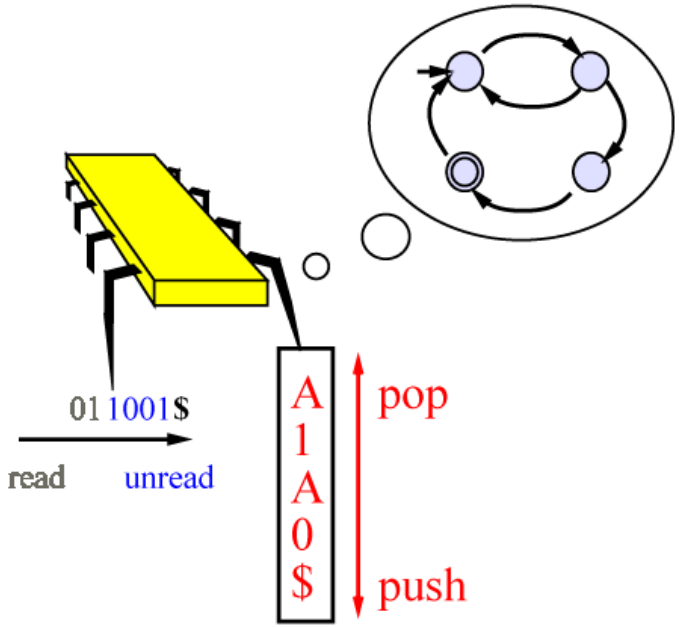
- Let A be a context-free language.
- By definition, A has a context-free grammar G generating it.
- On input w, the PDA P should figure out if there is a derivation of w using G.

Question: How does P figure out which substitution to make?

Answer: It guesses.

CFL Implies PDA (cont.)

Where do we keep the intermediate string ($01A1A0$ in the example below))?



intermediate string: 01A1A0

- can't put it all on the stack
- partial strings starting by a variable are kept on stack

CFL Implies PDA

Informally, on input string $w \in \Sigma^*$:

- P pushes start variable S on stack
- keeps making substitutions
- when popping a terminal, P checks equality with current input string
- rejects if not equal
- when popping a variable, P pushes to top of stack a right hand side of some rule corresponding to variable (zero, one, or more symbols).
- if stack is empty, allow to accept and terminate.

CFL Implies PDA (cont.)

Informal description:

- push $S\$$ on stack
- if top of stack is variable A, non-deterministically select rule $A \to \alpha$ and substitute.
- if top of stack is terminal a, read next input and compare. If they differ, reject.
- if top of stack is $\$$, allow to enter accept state.

CFL Implies PDA (cont.)

Need shorthand to push strings of finite length onto stack. For example, suppose

$$A \to BC$$

is a derivation of the CFG.
Then we add a "shorthand state", q_e, and the two transitions

$$(q_e, C) \in \delta(q_\ell, A, \varepsilon), \quad \delta(q_e, \varepsilon, \varepsilon) = \{(q_\ell, B)\}$$

Notice that the second transition is deterministic (the first one may or may not be). Also notice order: Push C first, then B.
These intermediate states are different for different derivations.

CFL Implies PDA (cont.)

States of P are

- start state q_s
- accept state q_a
- loop state q_ℓ
- q_e states, needed for shorthand of right hand sides of rules

Transition Function

Initialize stack

$$\delta(q_s, \varepsilon, \varepsilon) = \{q_\ell, S\$\}$$

Top of stack is variable (shorthand for multiple transitions)

$$\delta(q_\ell, \varepsilon, A) = \{(q_\ell, w) | \text{ where } A \to w \text{ is a rule } \}$$

Top of stack is terminal

$$\delta(q_\ell, a, a) = \{(q_\ell, \varepsilon)\}$$

End of Stack and End of Input

$$\delta(q_\ell, \varepsilon, \$) = \{(q_a, \varepsilon)\}$$

Example

$$
\begin{aligned}
S &\rightarrow AT|\varepsilon \\
A &\rightarrow AB|AA|a \\
B &\rightarrow b \\
T &\rightarrow TT|t
\end{aligned}
$$

Transition rules for PDA: On black/white board.

Only If Part

Theorem: If a PDA accepts a language, L, then L is context-free.

- For each pair of states p and q in P, we will have a variable A_{pq} in the grammar G.
- This variable, A_{pq}, generates all strings that take P from p with an empty stack to q with empty stack.
- Same string also takes p with any stack to q with same stack!
- Start variable is A_{q_0,q_a} (assuming a single accept state q_a).

PDA Implies CFL

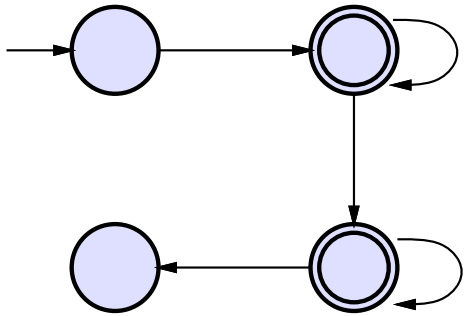
To make things easier, we slightly modify P

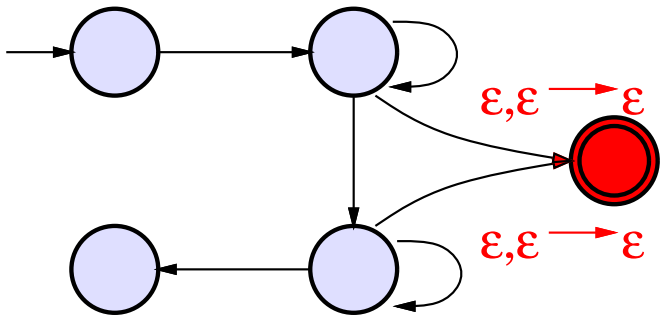
- Has single accept state q_a.
- It empties stack before accepting.
- Each transition either pushes a symbol on stack, or pops a symbol from stack, but not both.

PDA Implies CFL (2)

Modify P to make things easier

- single accept state q_a



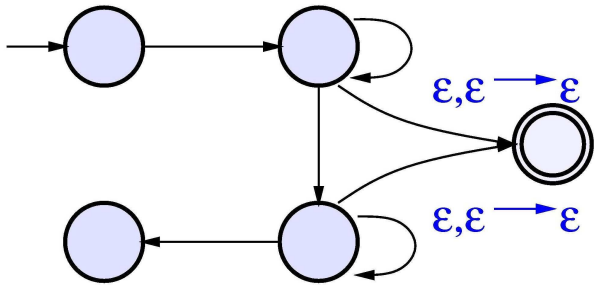


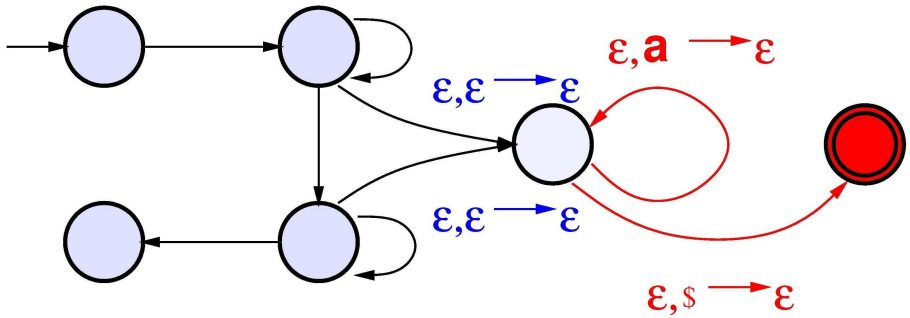
- empties stack before accepting
- each transition pushes or pops, but not both.

PDA Implies CFL (3)

Modify P to make things easier

- single accept state q_a $\surd$
- empties stack before accepting

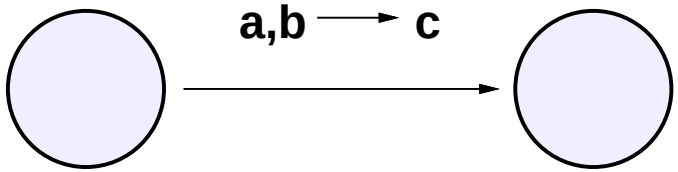


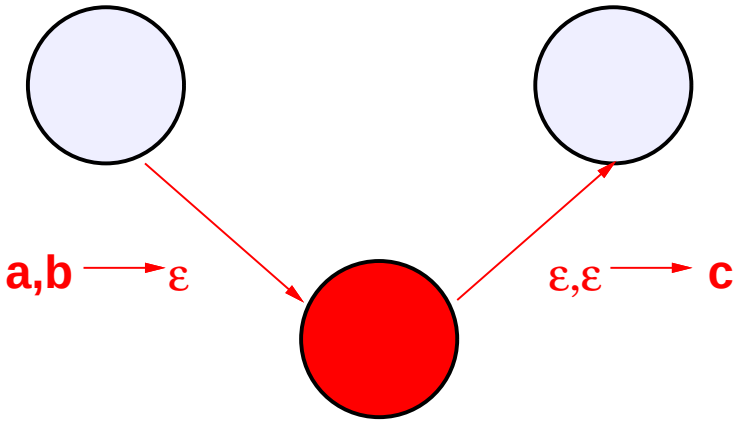


PDA Implies CFL (4)

Modify P to make things easier

- single accept state q_a $\surd$
- empties stack before accepting $\surd$
- transition either pushes or pops, but not both





Proof Idea

Suppose string x takes P from p with empty stack to q with empty stack.

First move that touches the stack must be a push, last must be a pop.

In between, two possibilities:

- Stack is empty only at start and finish, but not in middle.
- Stack was also empty at some point in between.

Proof Idea (2)

Suppose string x takes P from p with empty stack to q with empty stack.

First move that touches the stack must be a push, last must be a pop.

In between, two possibilities:

- Stack is empty only at start and finish, but not in middle.
  Simulate by: $A_{pq} \rightarrow aA_{rs}b$, where a, b are first and last symbols in x (shorter x will be taken care of too), r follows p, and s precedes q.
- Stack was also empty at some point in between.
  Simulate by: $A_{pq} \rightarrow A_{pr}A_{rq}$, r is intermediate state where P has empty stack.

Details of Simulating Grammar

Given PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, \{q_a\})$, construct grammar G.
Variables are $\{A_{pq} \mid p, q \in Q\}$.

Start variable is $A_{q_0 q_a}$.

Rules:

- For $p, q, r, s \in Q$, $t \in \Gamma$, and $a, b \in \Sigma$, if $(r, t) \in \delta(p, a, \varepsilon)$ and $(q, \varepsilon) \in \delta(s, b, t)$,
  add rule $A_{pq} \rightarrow a A_{rs} b$.
- for every $p, q, r \in Q$, add rule $A_{pq} \rightarrow A_{pr} A_{rq}$.
- for each $p \in Q$, add rule $A_{pp} \rightarrow \varepsilon$.

Overall Structure

Should now prove

Claim: A_{pq} generates x if and only if x brings P from p with empty stack to q with empty stack.

Only If Part

Theorem: If a PDA accepts a language, L, then L is context-free.

Proof After constructing the grammar G , should prove it generates exactly the same language accepted by the PDA. This is done by induction on the length of any computation of P on any input string x.

The induction argument is a bit lengthy and tedious, and we'll skip it.

Diehards are welcome to consult pp. 106–114 in Sipser's book.

CFL Closure Properties

- Are the context free languages closed under intersection?
- Suggested approach: Can we intersect two context free languages to get $0^n 1^n 2^n$?

CFL Closure Properties

- Are the context free languages closed under intersection?

$$S_1 \to A_1 B_1 \qquad S_2 \to A_2 B_2$$
$$A_1 \to 0A_1 1 | 01 \qquad A_2 \to 0A_2 | \varepsilon$$
$$B_1 \to 2B_1 | \varepsilon \qquad B_2 \to 1B_2 2 | 12$$

$$L_1 = 0^n 1^n 2^* \qquad L_2 = 0^* 1^n 2^n$$

- $L_1 \cap L_2 = 0^n 1^n 2^n$
- L_1 is a context free language, L_2 is a context free language, but $L_1 \cap L_2$ is not a context free language.

CFL Closure Properties

The fact that CFLs are not closed under intersection but are closed under union implies they are not closed under complementation, as $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$.

CFL Closure Properties

Can we give a simple, specific example, where L is not CFL but $\overline{L}$ is?

- Take $L = \{ww \mid w \in \{0,1\}^*\}$.
- For any $y \in \overline{L}$, either
  - y's length is odd.
  - y's length is even, 2ℓ, and there is an $i \geq 1$ such that $y_i \neq y_{\ell+i}$.
- PDA non-deterministically tries to verify one of the options. Employs stack for "matching locations". Accepts only on a successful branch (voluntary home assignment: fill in the details!).

CFL Closure Properties

- Are the context free languages context free languages closed under intersection with a regular language?
- That is, if L_1 is context free languages, and L_2 is regular, must $L_1 \cap L_2$ be context free languages?
- Run PDA L_1 and DFA L_2 "in parallel" (just like the intersection of two regular languages).
- Formal details omitted (but you should be able to figure them out).
- Why does intersection work here but does not work for two PDAs?
- Because the DFA does not try to access the stack, while the two PDAs may impose conflicting actions on the stack.

CFL Closure Properties: Example

Is $L = \{(0 \cup 1 \cup 2)^* : \#\text{ of } 0\text{'s} = \#\text{ of } 1\text{'s} = \#\text{ of } 2\text{'s}\ \}$ context free?

CFL Closure Properties

- $L \triangleq \{(0 \cup 1 \cup 2)^* : \#\ 0\text{'s} = \#\ 1\text{'s} = \#\ 2\text{'s}\ \}$
- Is L context free?
  - $L \cap 00^*11^*22^* = \{0^n 1^n 2^n : n > 0\}$ which is not context free.
  - Context free languages intersected with a regular languages are context free.
  - $00^*11^*22^*$ is regular.
  - So L is not a context free language!
- This could also be established using pumping lemma, but proof above is much more elegant.

More CFL Closure Properties

- Assignments
- Homomorphism
- Inverse Homomorphism

Algorithmic Questions Regarding CFGs

Given a CFG, G, and a string w, does G generate w?

Initial Idea: Design an algorithm that tries all derivations.

Problem: If G does not generate w, we'll never stop.

Chomsky Normal Form

A simplified, canonical form of context free grammers. Elegant by itself, useful (but not crucial) in proving equivalence theorem. Can also be used to slightly simplify proof of pumping lemma.
Every rule has the form

$$\begin{aligned} A &\rightarrow BC \\ A &\rightarrow a \\ S &\rightarrow \varepsilon \end{aligned}$$

where S is the start symbol, A, B and C are any variable, except B and C not the start symbol, and A can be the start symbol.

CFN: Theorem

Theorem: Any context-free language is generated by a context-free grammar in Chomsky normal form.
Basic idea:

- Add new start symbol S_0.
- Eliminate all ε rules of the form $A \to \varepsilon$.
- Eliminate all "unit" rules of the form $A \to B$.
- Patch up rules so that grammar generates the same language.
- Convert remaining "long rules" to proper form.

Algorithmic Questions for CFGs (2)

Lemma: If G is in Chomsky normal form, $|w| = n$, and w is generated by G, then w has a derivation of length $2n - 1$ or less.

Algorithm's idea:

- First, convert G to Chomsky normal form.
- Now need only consider a finite number of derivations – those of length $2n - 1$ or less.

Algorithmic Questions for CFGs: membership

- Build a function $derive(A, x)$ that returns TRUE if $A \stackrel{*}{\Rightarrow} x$
- PROCEDURE $derive(A, x)$.
  - If $|x| = 1$ then if $A \rightarrow x \in R$ return TRUE, otherwise return FALSE.
  - For each $A \rightarrow BC$ and each partition $x = x_1 x_2$,
    - Call $derive(B, x_1)$ and $derive(C, x_2)$.
    - If both return true, exit and return TRUE.
  - Return FALSE.
- Test membership by calling $derive(S, w)$.
- Why did we need CNF ?!

Algorithmic Questions for CFGs: membership

- What is the time complexity of $derive(S, w)$?
- Simple Recursive implementation analysis:
  - each time test $|R|$ rules and n partitions.
  - $T(n) \leq |R|n \cdot 2T(n-1)$
  - $T(n) = O((|R|n)^n)$.
- Still exponential

Dynamic Programming for CFG membership

- What is the time complexity of $derive(S, w)$?
- Better implementation: keep in memory the results of $derive(A, x)$.
  - Number of different inputs: $|V| \cdot n^2$.
  - Only $|V| \cdot n^2$ calls, each takes $O(n|R|)$.
  - $T(n) = O(|R|n^3|V|)$.
- Polynomial rime!

Emptiness of CFGs

Given a CFG, G, is $L(G) = \emptyset$?

In other words, is there any string w, such that G generate w?

Theorem: There is an algorithm that solves this problem (and always halts).

Possible approaches for a proof:

Bad Idea: We know how to test whether $w \in L(G)$ for any string w, so just try it for each w. (criticize this...)

Better Idea: Can the start variable generate a string of terminals?

Even Better Idea: Can a particular variable generate a string of terminals?

Removing redundant variables and terminals of a CFG

1. Mark all terminal symbols in G.
2. Repeat until no new variable become marked:
   Mark any A where $A \rightarrow U_1 U_2 \ldots U_k$ and all U_i have already been marked.
4. Remove all unmarked variables, and any rule they appear in.
5. If S is removed, then $L(G) = \emptyset$.
6. Remove any variable A not reachable from S .
7. Remove any terminal which does not appear in some rule.

CFG Emptiness (2)

Algorithm: On input G (a CFG),

1. Remove redundant variables and terminals, get G'.
2. $L(G') = \emptyset$ iff S is removed

Finiteness of CFGs

Given a CFG, G, is $|L(G)|$ finite?

1. Remove redundant variables and terminals.
2. Create a graph where nodes are variables and directed edges are derivations.
3. $L(G)$ is infinite iff the graph has a cycle.

CFGs "Fullness"

Given a CFG, G, is $L(G) = \Sigma^*$?

We just saw an algorithm to determine, given a CFG, G, if $L(G) = \emptyset$

$L(G) = \Sigma^*$ iff $\overline{L(G)} = \emptyset$. Why not modify the algorithm so it determines emptiness of the complement?

Unfortunately, CFGs are not closed under complement.

Fact: There is no algorithm to solve this problem.

We are not prepared to prove this remarkable fact (yet).

CFGs Inherent Ambiguity

Given a CFG, G, is $L(G)$ inherently ambiguous?

This means that for any CFG that generates $L(G)$, there is a word in the language with two different parse trees.

Fact: There is no algorithm to solve this problem.

We will not prove this fact, yet you want to know it to put things in context.

When Are Two CFGs equivalent?

Given two CFGs, G, H, is $L(G) = L(H)$?

Hey, we did this already for equivalence of DFAs!

We constructed C from A and B:

$$L(C) = \left(L(A) \cap \overline{L(B)}\right) \cup \left(\overline{L(A)} \cap L(B)\right).$$

and tested whether $L(C)$ is empty.

When Are Two CFGs equivalent?

This approach was fine for DFAs, but not for CFLs!

The class of context-free languages is not closed under complementation or intersection.

Fact: There is no algorithm to solve this problem.

We are not prepared to prove this remarkable fact (yet).

A Short Summary

- Regular Languages $\equiv$ Finite Automata.
- Context Free Languages $\equiv$ Push Down Automata.
- Closure properties of regular languages and of CFLs.
- Most algorithmic problems for finite automata are solvable.
- Some algorithmic problems for finite automata are not solvable.
- Pumping lemmata for both classes of languages.
- There are additional languages out there.

The View Over The Horizon