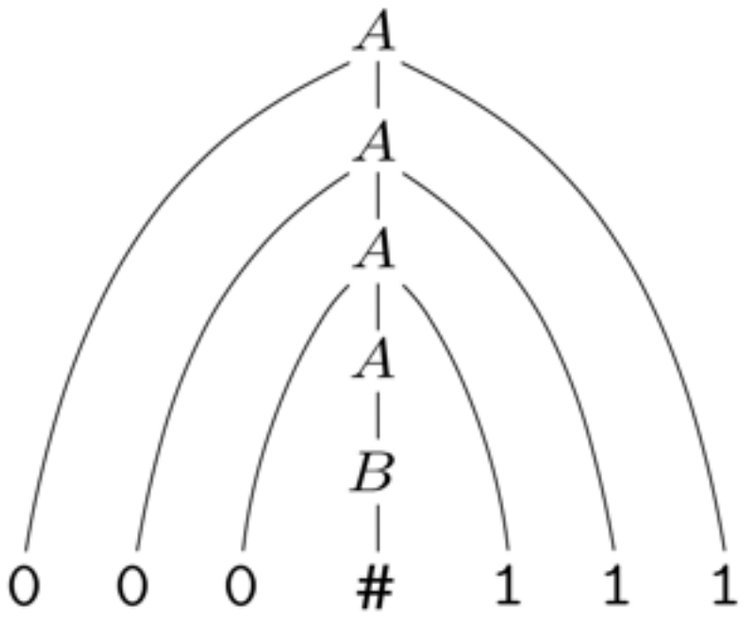
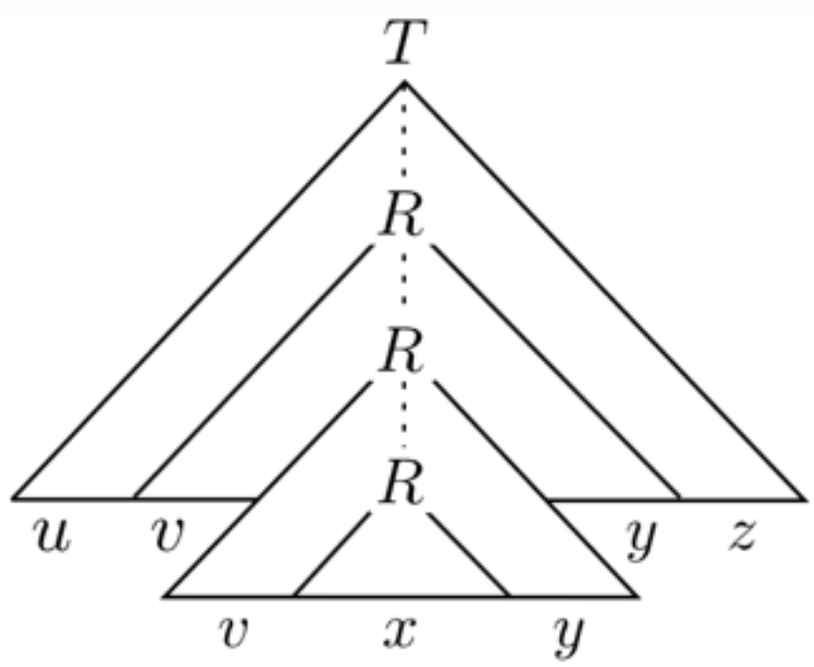
# CS 321

# Theory of Computation

## Part II: Context-Free Languages

Sipser 2; Linz 5,6,7,8




**Instructor: Liang Huang**

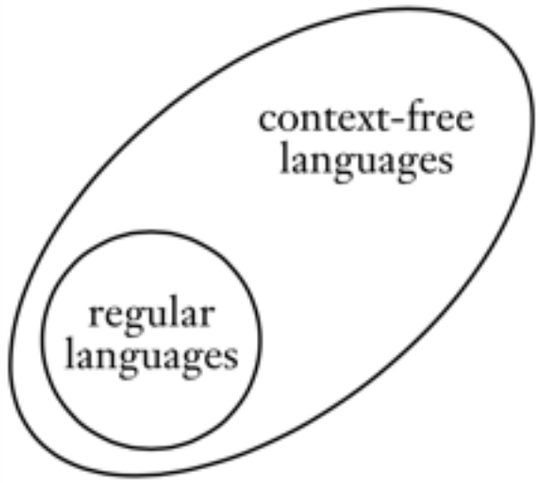
(some slides courtesy of H. Potter)

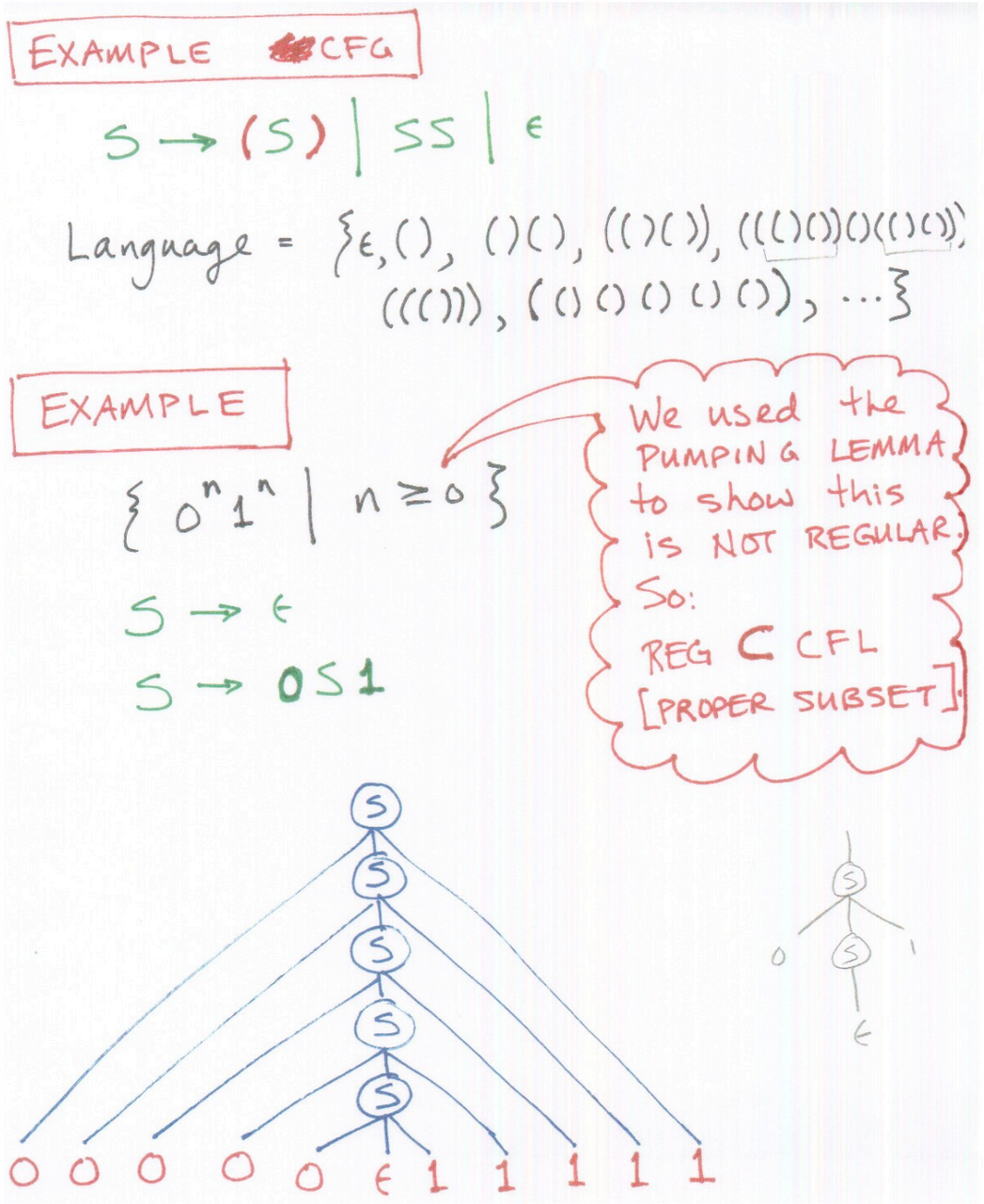
# Roadmap

- Weeks 1-5: Regular (finite-state automata)
  - DFA, RL, NFA, RE
  - regular operations and closure properties
  - pumping lemma to prove non-regularity
- Weeks 6-8: Context-Free (pushdown automata)
  - context-free grammars & languages such as $a^n b^n$, $ww^R$
  - pushdown automata
  - regular operations and closure properties
- Week 9: Turing Machines
- Week 10: Decidability

# Context-Free Languages

- write a CFG for
  - $a^n\ b^n$
  - $w\ w^R$
  - $a^m\ b^n\ c^n\ d^m$
  - $a^m\ b^n\ c^{3m+2n}$
  - $a^m\ b^n\ c^m\ d^n$ ?
  - $a^n\ b^n\ c^n$ ?
  - $a^n\ b^m$
  - (0 (0|1)* 0) | (1 (0|1)* 1) | 0 | 1 | \epsilon




- CFG can only do "nested"; Regular Grammar ⊆ CFG

# Derivations and Parse Trees

$$E \to E+T \mid T$$
$$T \to T \times F \mid F$$
$$F \to (E) \mid a$$

DERIVATION

$E \Rightarrow E+T \Rightarrow T+T \Rightarrow F+T \Rightarrow a+T$
$\Rightarrow a+F \Rightarrow a+(E) \Rightarrow a+(T)$
$\Rightarrow a+(T \times F) \Rightarrow a+(F \times F)$
$\Rightarrow a+(a \times F) = a+(a \times a)$

WE WRITE:

$E \overset{*}{\Rightarrow} a+(a \times a)$

$E \overset{*}{\Rightarrow} a+(E) \Rightarrow a+(T) \overset{*}{\Rightarrow} a+(a \times a)$

LEFT-MOST DERIVATION:
Always choose Left-most Variable

$\ldots \Rightarrow F+T \Rightarrow a+T \Rightarrow \ldots a+(a \times a)$

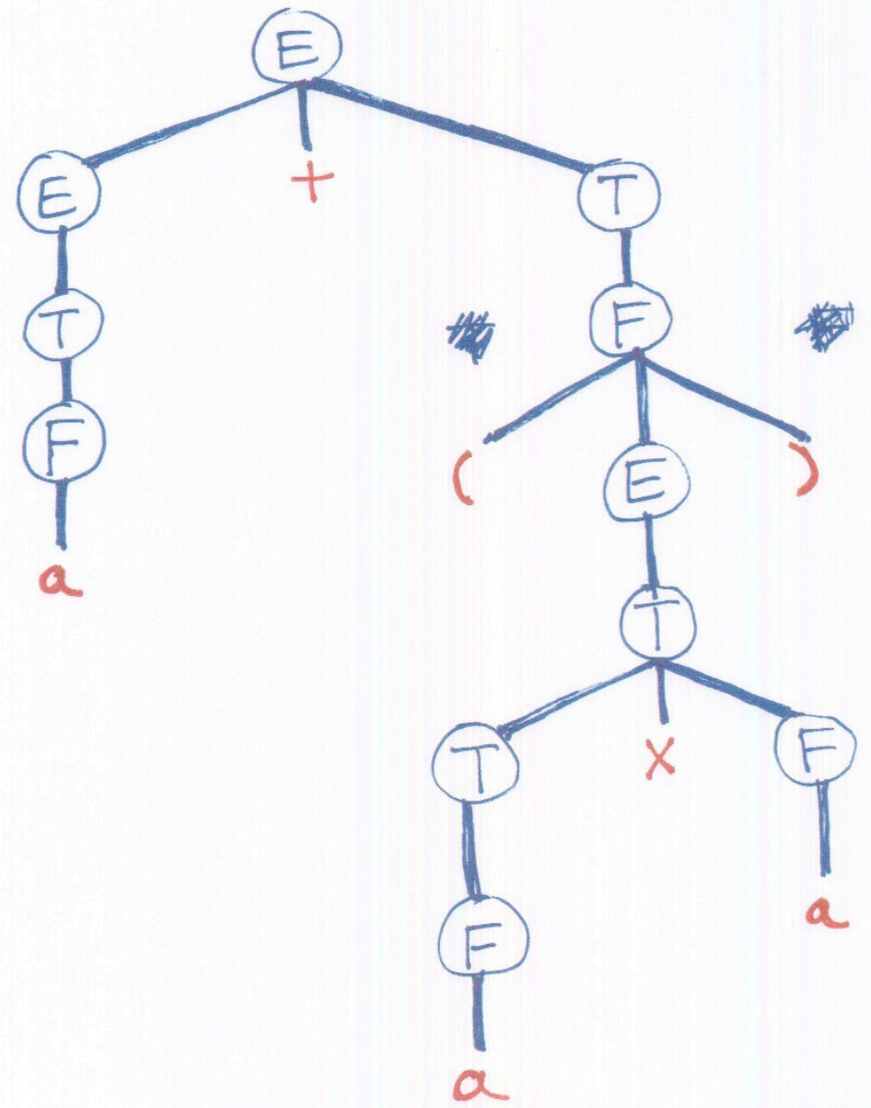
RIGHT-MOST DERIVATION:
Always choose Right-most Variable

$\Rightarrow F+T \Rightarrow F+F \Rightarrow \ldots a+(a \times a)$

PARSE TREES

$$E \to E+T \mid T$$
$$T \to T \times F \mid F$$
$$F \to (E) \mid a$$

$E \overset{*}{\Rightarrow} a+(a \times a)$



# CFGs for Natural Language

- S → NP VP
- N → boy | girl | ...
- P → with | in | on | ...
- V → eat | saw | ...

I saw a boy with a telescope

I eat sushi with tuna

- NP → D N
- NP → NP PP
- PP→P NP
- VP→V NP
- VP→VP PP

# CFGs for Natural Language

- S → NP VP
- N → boy | girl | ...
- P → with | in | on | ...
- V → eat | saw | ...

- NP → D N
- NP → NP PP
- PP→P NP
- VP→V NP
- VP→VP PP

I saw a boy with a telescope

I eat sushi with tuna

# Context-Free Grammars: Definition

**A CFG is a 4-tuple ⟨V,Σ,R,S⟩**

**A set of nonterminals or variables V (often "N" for nonterminals)**
(e.g. V = {S, NP, VP, PP, Noun, Verb, ....})

**A set of terminals Σ (often "T" for terminals)**
(e.g. Σ = {*I, you, he, eat, drink, sushi, ball,* })

**A set of rules R (often "P" for productions)**
**R ⊆ {A → β with left-hand-side (LHS)** $A \in V$
**and right-hand-side (RHS)** $\beta \in (V \cup \Sigma)^*$ **}** **-- RHS could be ε**

**A start symbol S (sentence)**

$$G = (\{S\}, \{0, 1\}, \{S \to 0S1, S \to \epsilon\}, S)$$

$$L(G) = \{0^n 1^n \mid n \geq 0\}$$

# Yields, Derives, Sentential Form

yields: $uAv \Rightarrow uwv$ if $A \to w \in R$

derives: $u \overset{*}{\Rightarrow} v$ if $u = v$ or $\exists w$ s.t. $u \overset{*}{\Rightarrow} w$ and $w \Rightarrow v$.

language: $L(G) = \{w \in \Sigma^* \mid S \overset{*}{\Rightarrow} w\}$

sentential form: $v \in (\Sigma \cup V)^*$ if $S \overset{*}{\Rightarrow} v$

$E \to E + T \mid T$
$T \to T \times F \mid F$
$F \to (E) \mid a$

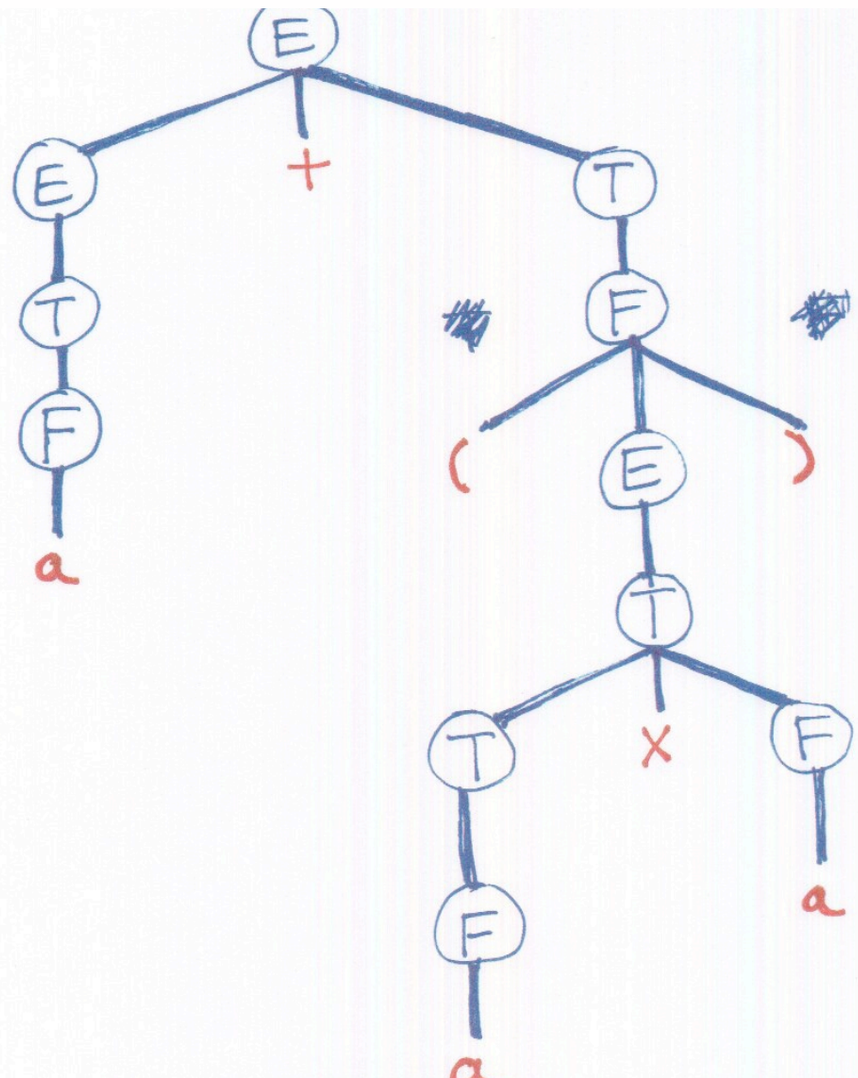
DERIVATION

$\underline{E} \Rightarrow \underline{E}+T \Rightarrow \underline{T}+T \Rightarrow \underline{F}+T \Rightarrow a+\underline{T}$
$\Rightarrow a+\underline{F} \Rightarrow a+(\underline{E}) \Rightarrow a+(\underline{T})$
$\Rightarrow a+(\underline{T} \times F) \Rightarrow a+(\underline{F} \times F)$
$\Rightarrow a+(a \times \underline{F}) = a+(a \times a)$

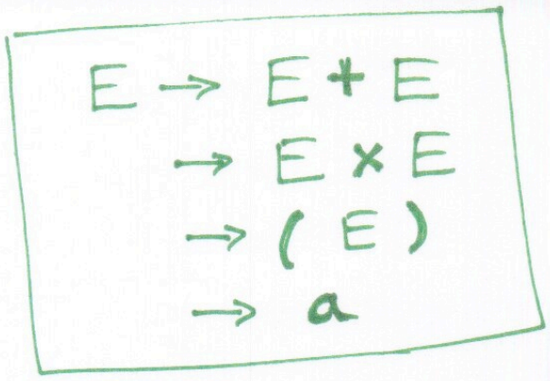
WE WRITE:

$E \overset{*}{\Rightarrow} a+(a \times a)$

$E \overset{*}{\Rightarrow} a+(E) \Rightarrow a+(T) \overset{*}{\Rightarrow} a+(a \times a)$

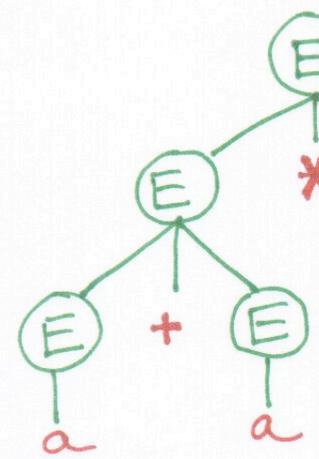


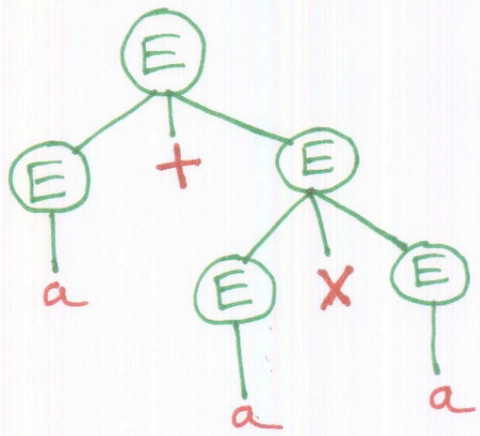
# Ambiguous vs. Unambiguous Grammar




- ambiguous grammar: some string can have two parse trees
- <u>sometimes</u> there is an equivalent unambiguous grammar
- but not always! => inherently ambiguous languages

a + a × a




- FOR EACH PARSE TREE, THERE IS EXACTLY ONE LEFT-MOST DERIVATION. (AND ONE RIGHT-MOST DERIVATION.)

**AMBIGUOUS STRINGS**

- MORE THAN ONE PARSE TREE.
- FUNDAMENTALLY DIFFERENT WAYS TO DERIVE THIS STRING.

**AMBIGUOUS GRAMMAR**

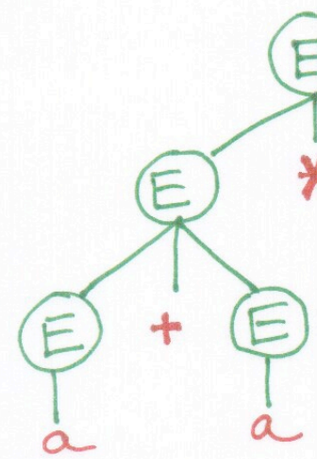
A grammar is "AMBIGUOUS" if some string can be derived in more than one way.
I.e., Some string has MULTIBLE PARSE TREES.

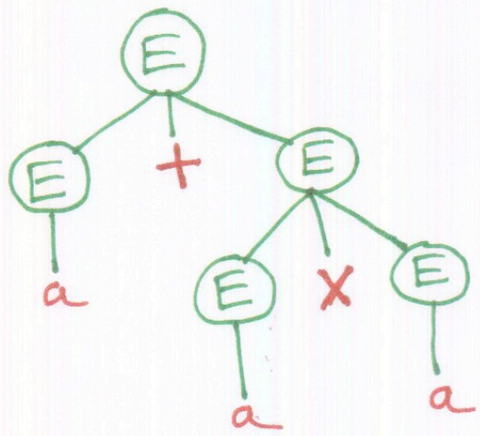
# Ambiguous vs. Unambiguous Grammar

- ambiguous grammar: some string can have two parse trees
- <u>sometimes</u> there is an equivalent unambiguous grammar
- but not always! => inherently ambiguous languages

$E \to E+E \mid E \times E \mid (E) \mid a$

$a+a \times a$




- FOR EACH PARSE TREE, THERE IS EXACTLY ONE LEFT-MOST DERIVATION. (AND ONE RIGHT-MOST DERIVATION.)

**AMBIGUOUS STRINGS**

- MORE THAN ONE PARSE TREE.
- FUNDAMENTALLY DIFFERENT WAYS TO DERIVE THIS STRING.

**AMBIGUOUS GRAMMAR**

A grammar is "AMBIGUOUS" if some string can be derived in more than one way.
I.e., some string has MULTIPLE PARSE TREES.

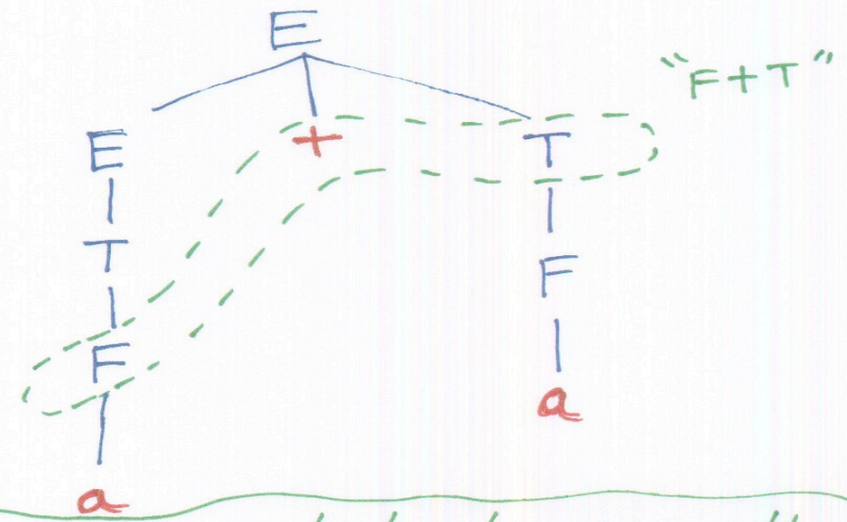
YOU GET THE SAME RESULT:

$E \to E+T \mid T$
$T \to T \times F \mid F$
$F \to (E) \mid a$

$\ldots \Rightarrow \underline{F}+T \Rightarrow a+\underline{T} \Rightarrow a+\underline{F} \Rightarrow a+a$

$\ldots \Rightarrow F+\underline{T} \Rightarrow F+\underline{F} \Rightarrow \underline{F}+a \Rightarrow a+a$

PARSE TREE:




The parse tree abstracts away the actual order in which the rules are used. It only remembers which rules were used

# Ambiguous String, Grammar, Language

## Ambiguous Grammar

$E \to E+E$

$\to E \times E$

$\to (E)$

$\to a$

## Equivalent Unambiguous Grammar

$E \to E+T$

$\to T$

$T \to T \times F$

$\to F$

$F \to (E)$

$\to a$

## Ambiguous Strings

Multiple Left-most Derivations
≡ Multiple Parse Trees.

## Ambiguous Grammar

If there exists some string that can be derived ambiguously.

If you have an ambiguous grammar, you might be able to find an equivalent grammar that is unambiguous.

## Ambiguous Language

All grammars for the language are ambiguous.
There is no unambiguous grammar that accepts the language
The language is "inherently ambiguous."

# Ambiguous String, Grammar, Language

## AMBIGUOUS GRAMMAR

$E \to E+E$
$\to E \times E$
$\to (E)$
$\to a$

## EQUIVALENT UNAMBIGUOUS GRAMMAR

$E \to E+T$
$\to T$
$T \to T \times F$
$\to F$
$F \to (E)$
$\to a$

inherently ambiguous language:
$\{a^i b^j c^k \mid i=j \text{ or } j=k\}$
there is no algorithm to decide whether a context-free language is inherently ambiguous
(*undecidable problem!*)

## AMBIGUOUS STRINGS

MULTIPLE LEFT-MOST DERIVATIONS
≡ MULTIPLE PARSE TREES.

## AMBIGUOUS GRAMMAR

IF THERE EXISTS SOME STRING THAT CAN BE DERIVED AMBIGUOUSLY.

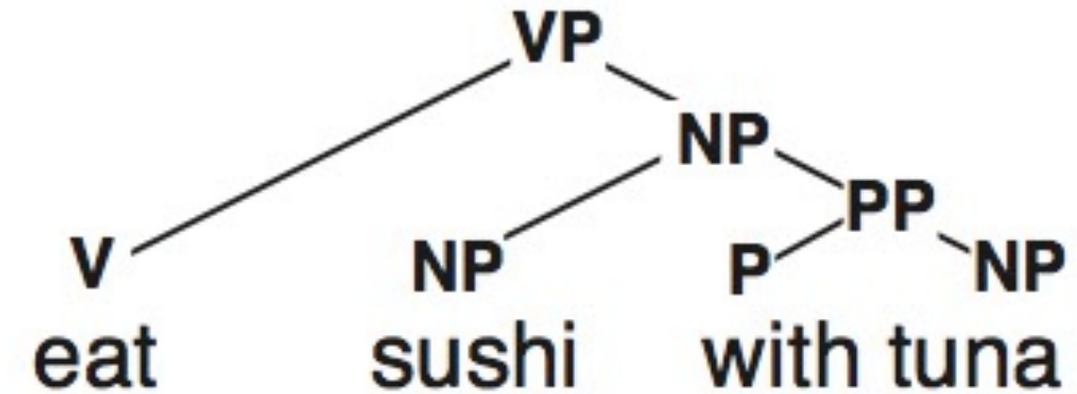
If you have an ambiguous grammar, you might be able to find an equivalent grammar that is unambiguous.

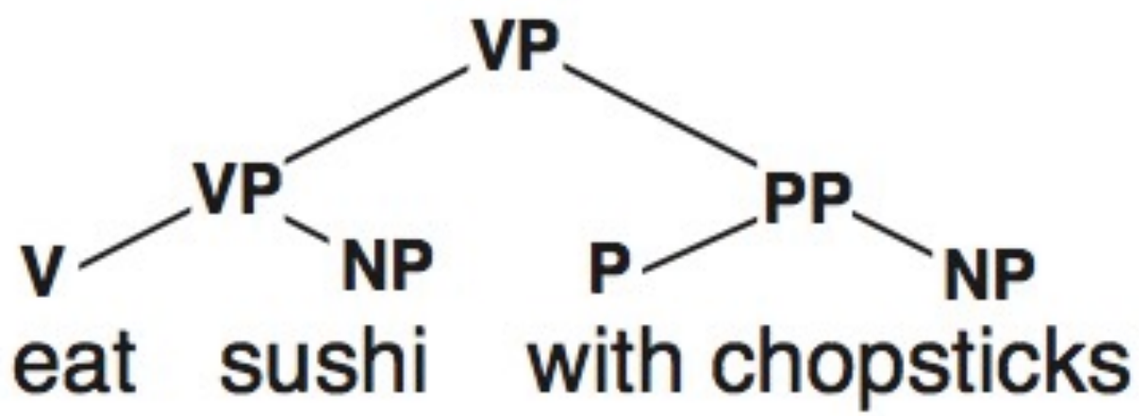
## AMBIGUOUS LANGUAGE

ALL GRAMMARS FOR THE LANGUAGE ARE AMBIGUOUS.
THERE IS NO UNAMBIGUOUS GRAMMAR THAT ACCEPTS THE LANGUAGE
THE LANGUAGE IS "INHERENTLY AMBIGUOUS."

# Parse Trees and Ambiguity in NL

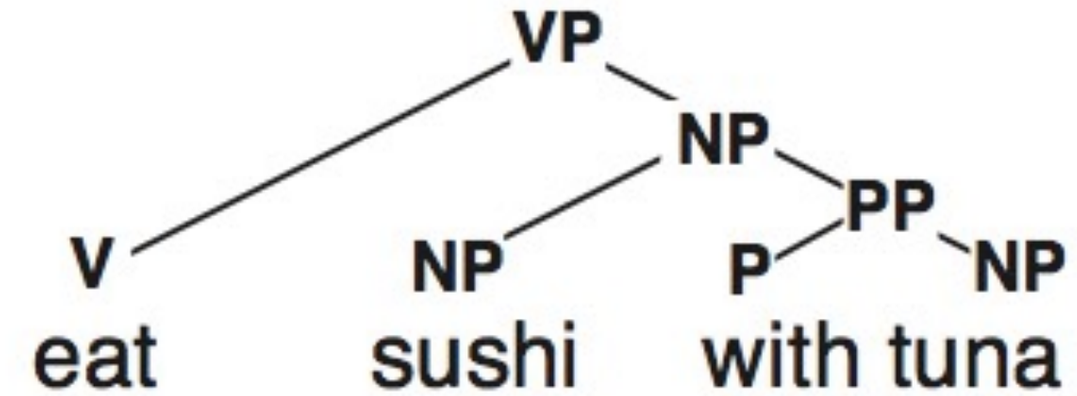
- **N** → *{sushi, tuna}*
- **P** → *{with}*
- **V** → *{eat}*
- **NP** → **N**
- NP → NP PP
- PP→P NP
- VP→V NP
- VP→VP PP

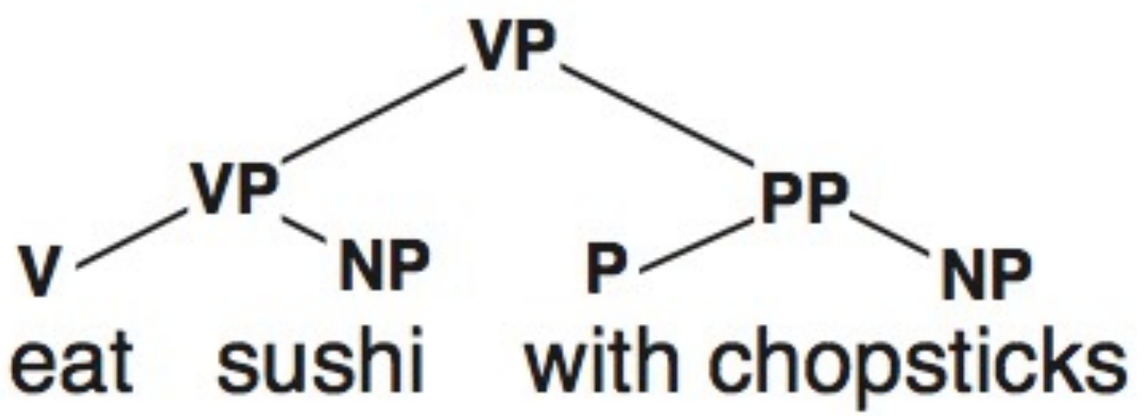


# Parse Trees and Ambiguity in NL

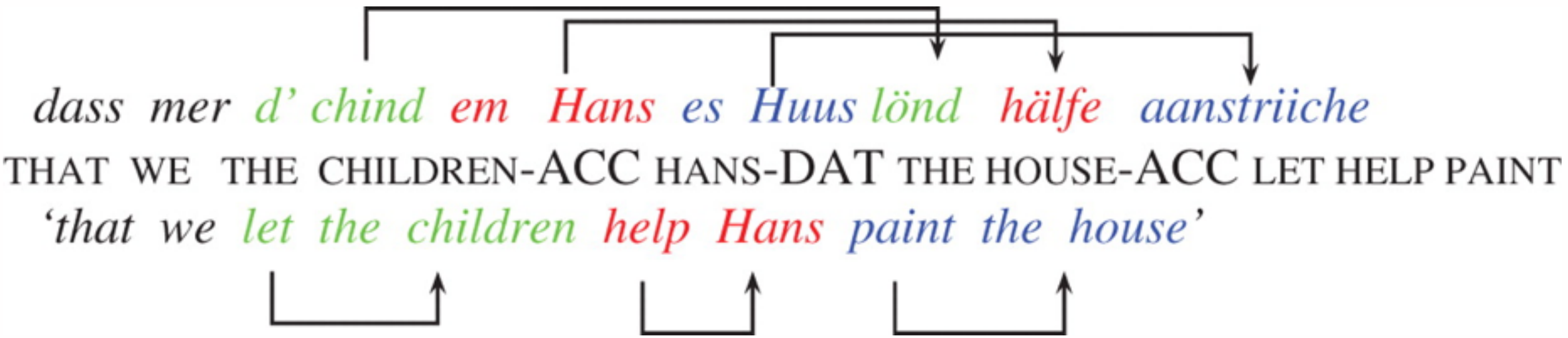
- **N** → *{sushi, tuna}*
- **P** → *{with}*
- **V** → *{eat}*
- **NP** → **N**
- NP → NP PP
- PP→P NP
- VP→V NP
- VP→VP PP

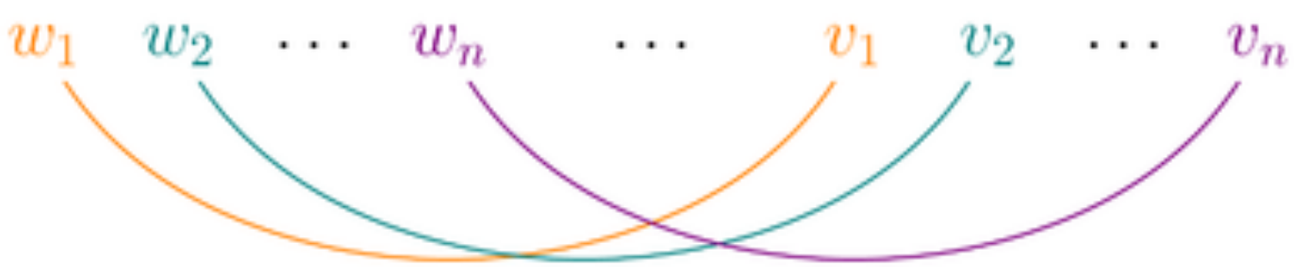



# Natural Languages Beyond Context-Free

- Dutch and Swiss German (Shieber, 1985)




STUART M. SHIEBER*

EVIDENCE AGAINST THE CONTEXT-FREENESS OF NATURAL LANGUAGE**

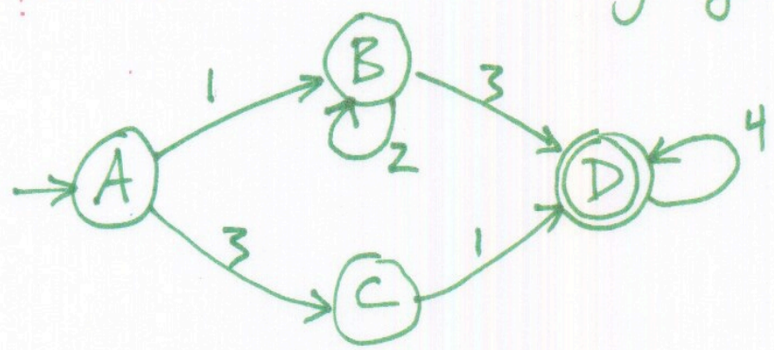
# Chomsky Hierarchy

| | Language | Automata | Parsing complexity | Dependencies |
|---|---|---|---|---|
| Type 3 | Regular | Finite-state | linear | adjacent words |
| Type 2 | Context-Free | Pushdown | cubic | nested |
| Type 1 | Context-sensitive | Linear Bounded | exponential | |
| Type 0 | Recursively Enumerable | Turing machine | | |

# Every regular language is context-free

**PROOF:**

Given a DFA for the Language, Construct a grammar that generates the same Language.




- Make a variable for each state.
- Make the variable for the starting state the starting variable
- Make a rule for each edge
- Add an EPSILON Rule for each accept state.

$A \to 1B$
$A \to 3C$
$B \to 2B$
$B \to 3D$
$C \to 1D$
$D \to 4D$
$D \to \epsilon$

# closed under union but not intersection

## Are context-free languages closed under union?

Grammar 1:

$S_1 \rightarrow$ ... lots of rules...

Grammar 2:

$S_2 \rightarrow$ ... lots of rules...

Union:

$S \rightarrow S_1 \mid S_2$

$S_1 \rightarrow \ldots$

$S_2 \rightarrow \ldots$

Yes!

## Are context-free languages closed under concatenation?

$S \rightarrow S_1 \; S_2$

$S_1 \rightarrow \ldots$

$S_2 \rightarrow \ldots$

Yes!

## Are context-free languages closed under intersection?

Consider $L_1 = \{ 0^N 1^N 2^i \}$

It is a "CFL":

$S \rightarrow AB$

$A \rightarrow 0A1 \mid \epsilon$

$B \rightarrow 2B \mid \epsilon$

Consider $L_2 = \{ 0^k 1^N 2^N \}$

It is a "CFL":

$S \rightarrow AB$

$A \rightarrow 0A \mid \epsilon$

$B \rightarrow 1B2 \mid \epsilon$

Consider $L = L_1 \cap L_2$

$\{ 0^N 1^N 2^N \}$

This is NOT a context-free language.

No!

Not in general,
but for some languages
the intersection
may also be
context-free...

# Complement: No

ARE CONTEXT-FREE LANGUAGES CLOSED UNDER COMPLEMENT?

THESE ARE SETS.

DEMORGAN'S LAWS APPLY.

$$A \cap B = \overline{\bar{A} \cup \bar{B}}$$

ASSUME: CLOSED UNDER COMPLEMENT.

THEN RIGHT-HAND SIDE IS CLOSED.

THEN LEFT-HAND SIDE IS A C.F.L.

CONTRADICTION.

**No!** (NOT IN GENERAL, BUT YES FOR SOME LANGUAGES)

EXAMPLE

$$L = \{ww \mid w \in \{0,1\}^*\}$$

The first half of the string is equal to the second half.

L is not context-free.

PALINDROMES $= \{ww^R \mid w\}$

HOWEVER

$\overline{L}$ IS CONTEXT-FREE.

CFLs are closed under: union, concatenation, star

# Complement: No

ARE CONTEXT-FREE LANGUAGES CLOSED UNDER COMPLEMENT?

THESE ARE SETS.

DEMORGAN'S LAWS APPLY.

$$A \cap B = \overline{\bar{A} \cup \bar{B}}$$

ASSUME: CLOSED UNDER COMPLEMENT.

THEN RIGHT-HAND SIDE IS CLOSED.

THEN LEFT-HAND SIDE IS A C.F.L.

CONTRADICTION.

**No!** (NOT IN GENERAL, BUT YES FOR SOME LANGUAGES)

EXAMPLE

$L = \{ww \mid w \in \{0,1\}^*\}$

The first half of the string is equal to the second half.

L is not context-free.

PALINDROMES = $\{ww^R \mid w\}$

HOWEVER

$\overline{L}$ IS CONTEXT-FREE.

CFLs are closed under: union, concatenation, star

$$L_1 = \{x1y \mid |x| = |y|\}$$

$$L_2 = \{x0y \mid |x| = |y|\}$$

$$\bar{L} = L_1 \cup L_2 \cup (L_1 L_2) \cup (L_2 L_1)$$

# Same number of a's as b's

- $L_1$ = same number of a's as b's
- $L_2$ = twice as many a's as b's

$L_1$ $S \rightarrow aSb \mid bSa \mid SS \mid \epsilon$

or $S \rightarrow aSbS \mid bSaS \mid \varepsilon$

# Same number of a's as b's

- $L_1$ = same number of a's as b's
- $L_2$ = twice as many a's as b's

$L_1$ $S \rightarrow aSb \mid bSa \mid SS \mid \epsilon$

or $S \rightarrow aSbS \mid bSaS \mid \varepsilon$

$L_2$ $S \rightarrow SS \mid aaSb \mid bSaa \mid aSbSa \mid \epsilon$

or $S \rightarrow SaSaSbS \mid SaSbSaS \mid SbSaSaS \mid \varepsilon$ (Miles)

# Same # of 0s and 1s

GRAMMAR DESIGN CHALLENGE

$L = \{ w \mid w \in \{0,1\}^* \text{ and the number of 0's equals the number of 1's} \}$

APPROACH: TRY TO THINK OF "MEANINGS" FOR THE NON-TERMINALS.

S = EQUAL # of 0's and 1's.

A = ONE MORE "1" THAN "0"'s

B = ONE MORE "0" THAN "1"'s

SOLUTION:

$S \to 0A \mid 1B \mid \epsilon$

$A \to 1S \mid 0AA$

$B \to 0S \mid 1BB$

SOLUTION #2:

$S \to SAB \mid \epsilon$

$A \to 0S1 \mid \epsilon$

$B \to 1S0 \mid \epsilon$

NOTE:

$C \to Cx \mid \epsilon$

GENERATES: $x^*$

$\{ \epsilon, x, xx, xxx, xxxx, \ldots \}$

$S \to SAB \mid \epsilon$

GENERATES:

AB AB AB AB ...

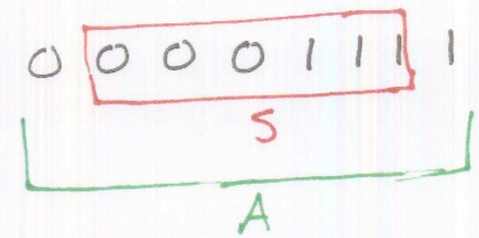
EVERY A CAN GO TO $\epsilon$.

EVERY B CAN GO TO $\epsilon$.

$\Rightarrow$ ANY STRING OF A's AND B's!

0101 0101

A A A A



# Inductive Proof part I

- $L_1$ = same number of a's as b's  $S \rightarrow aSbS \mid bSaS \mid \varepsilon$

We first show that each string generated by this grammar has the same number of $a$'s and $b$'s. Proceed by induction over the length $N$ of the derivation.

- **Case base:** $N = 1$. The only string derivable with a derivation of length $N = 1$ is $\epsilon$ ($S \rightarrow \epsilon$): it trivially holds that the number of $a$'s and $b$'s in $\epsilon$ is the same.

- **Inductive step:** Assume that all the strings $w$ derivable from $S$ with a derivation of length at most $N$ have an equal number of $a$'s and $b$'s. Consider now a string $w'$ derivable from $S$ with a derivation of length $N + 1$. Clearly, the first production used in the derivation that yields $w'$ is either $S \rightarrow aSbS$ or $S \rightarrow bSaS$. In either cases, the remaining $N$ productions can be ideally split into two derivations: one that generates a string $w_1$ from the first $S$ and another that generates a string $w_2$ from the second $S$. By induction, it clearly holds that in both $w_1$ and $w_2$ the number of $a$'s is equal to the number of $b$'s. Moreover, since $w$ is either $aw_1bw_2$ or $bw_1aw_2$, we get that the number of $a$'s in $w$ is equal to the number of $b$'s.

# Inductive Proof part 2

- $L_1$ = same number of a's as b's  $S \rightarrow aSbS \mid bSaS \mid \varepsilon$

To complete the proof, we still need to show that any string $w$ with an equal number of $a$'s and $b$'s can be derived from $S$. Proceed by induction on the length of $w$.

- **Case base:** $|w| = 0$, i.e. $w = \epsilon$. It is clear that $w$ can be derived from $s$ ($S \rightarrow \epsilon$).
- **Inductive step:** Assume that all the strings $w$ with an equal number of $a$'s and $b$'s and of length up to $2N$ are derivable from $S$. Consider now a string $w'$ with an equal number of $a$'s and $b$'s of length $2(N+1)$. W.l.o.g. assume that the first symbol in $w'$ is an $a$. Let $2 \leq j \leq 2N+2$ be the first index such that the substring of $w'$ from position 1 to position $j$ has an equal number of $a$'s and $b$'s. Notice that such $j$ always exists, since in the worst case $j = 2N+2$ will do it.

  If $j = 2N+2$, then this means that $w'$ is of the form $a^N b^N$ and thus can be derived from $S$ by the derivation $S \Rightarrow aSbS \Rightarrow aSb \Rightarrow aaSbSb \Rightarrow aaSbb \Rightarrow \ldots \Rightarrow a^N b^N$.

  Otherwise (i.e. $j < 2N+2$), notice that, since the symbol at position $j$ must be a $b$, $w'$ can be written as $w' = aw_1bw_2$, where $w_1$ and $w_2$ are both strings of length at most $N$, each having an equal number of $a$'s and $b$'s. Thus, by induction, both $w_1$ and $w_2$ can be derived from $S$ from which it follows that $w'$ can also be derived from $S$.

# Chomsky Normal Form (CNF)

**DEFINITION 2.8**

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

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

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

- Theorem: any CFG can be converted to CNF
  - how about epsilon-removal?

# Convert to Chomsky Normal Form

ALGORITHM TO CONVERT ANY CFG INTO CHOMSKY NORMAL FORM

**STEP 1**

MAKE SURE START SYMBOL DOES NOT APPEAR ON RIGHTHAND SIDE.

**STEP 2**

REMOVE RULES LIKE $A \to \epsilon$

**STEP 3**

GET RID OF ALL UNIT RULES $A \to B$

**STEP 4**

GET RID OF RULES WITH MORE THAN 2 SYMBOLS ON RIGHTHAND SIDE

$A \to BCDE$

$A \to Bcde$

**STEP 5**

MAKE SURE

$A \to BC$ ← ONLY 2. MUST BE VARIABLES.

$A \to a$ ← ONLY 1. MUST BE A TERMINAL SYMBOL.

**STEP 1:** MAKE SURE START SYMBOL DOESN'T APPEAR ON RIGHTHAND SIDE. ADD NEW START SYMBOL.

EXAMPLE

$S \to ASA \mid aB$

$A \to B \mid S$

$B \to b \mid \epsilon$

---

$S_0 \to S$

$S \to ASA \mid aB$

$A \to B \mid S$

$B \to b \mid \epsilon$

Add new start symbol

# Step 2: Remove Unary: A→ε

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

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

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

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

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

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

# Step 3: Remove Unary: A→B

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

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

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

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

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

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

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

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

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

# Step 4: Binarize and Step 5: Embellish

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

$$
\begin{aligned}
S_0 &\to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS \\
S &\to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS \\
A &\to B \mid S \mid \mathbf{b} \\
B &\to \mathtt{b}
\end{aligned}
\qquad\qquad
\begin{aligned}
S_0 &\to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS \\
S &\to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS \\
A &\to S \mid \mathtt{b} \mid \boldsymbol{ASA} \mid \mathbf{a}\boldsymbol{B} \mid \mathbf{a} \mid \boldsymbol{SA} \mid \boldsymbol{AS} \\
B &\to \mathtt{b}
\end{aligned}
$$

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

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

# Another Example of CFG=>CNF

- is this language regular?
- is this grammar ambiguous? an algorithm decide this?
- will CNF change the ambiguity of the original CFG?

**Example:** For the grammar

$$S \rightarrow aSb \mid SS \mid ab$$

we get first

$$S \rightarrow ASB \mid SS \mid AB \quad A \rightarrow a \quad B \rightarrow b$$

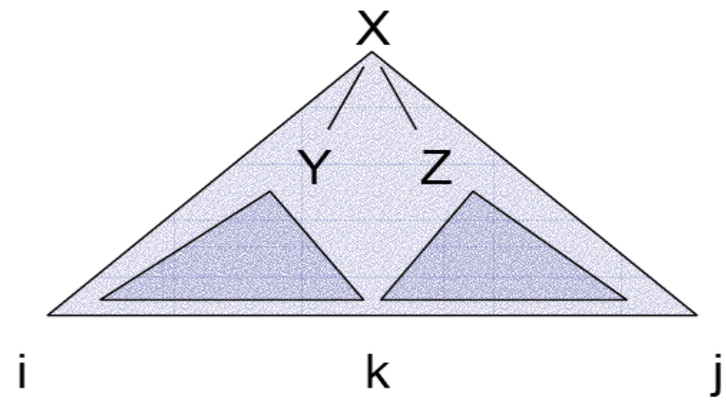
and then

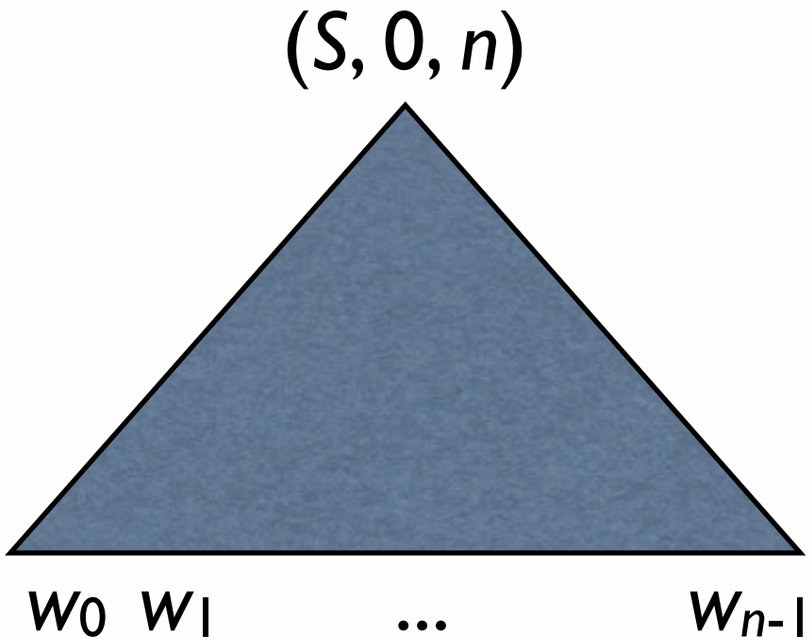
$$S \rightarrow AC \mid SS \mid AB \quad A \rightarrow a \quad B \rightarrow b \quad C \rightarrow SB$$

which is in Chomsky Normal Form

# CKY Parsing (for membership & ambiguity)

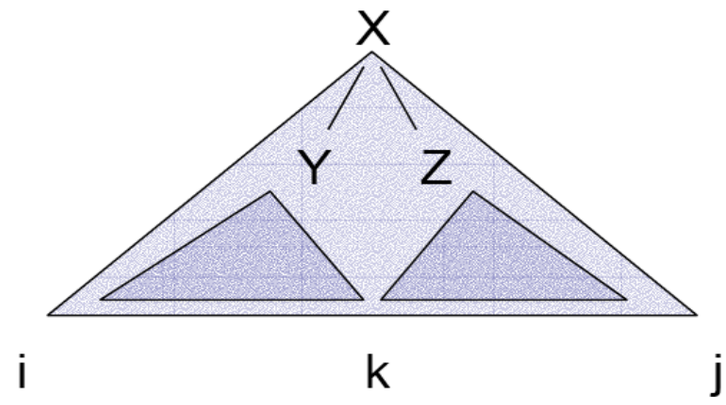
- given *any* grammar *G* and string *s*, decide membership: *s* in *L*(*G*)?
  - if *yes*, output one parse tree or *all* parse trees
  - this can decide whether *s* is ambiguous in *G*
- bottom-up dynamic programming: $O(n^3)$
  - assume *G* already in CNF (what if not?)

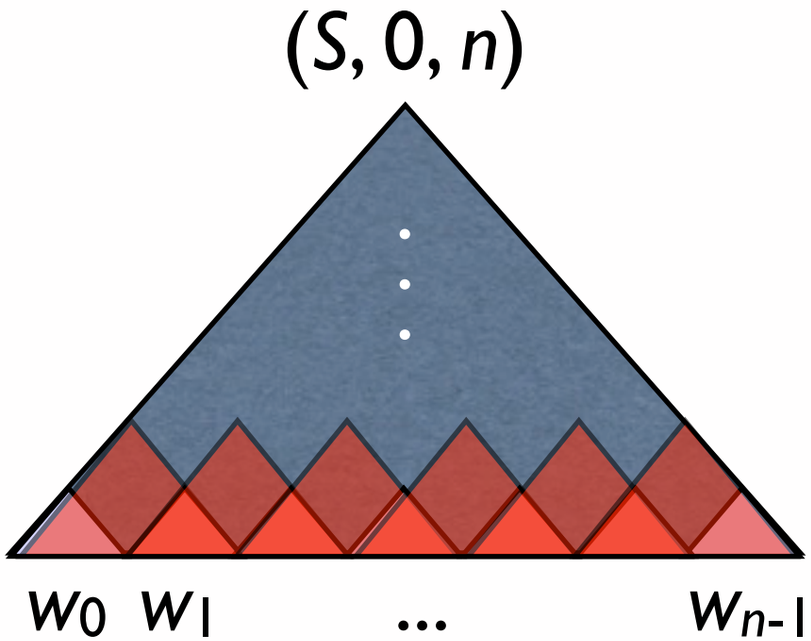


# CKY Parsing (for membership & ambiguity)

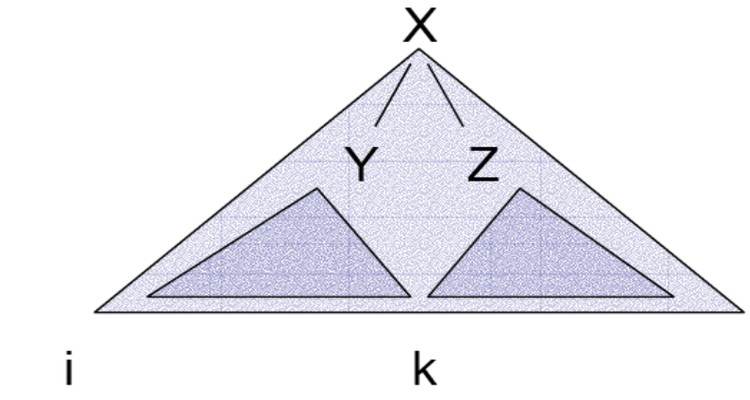
- given *any* grammar *G* and string *s*, decide membership: *s* in *L*(*G*)?
  - if *yes*, output one parse tree or *all* parse trees
  - this can decide whether *s* is ambiguous in *G*
- bottom-up dynamic programming: $O(n^3)$
  - assume *G* already in CNF (what if not?)



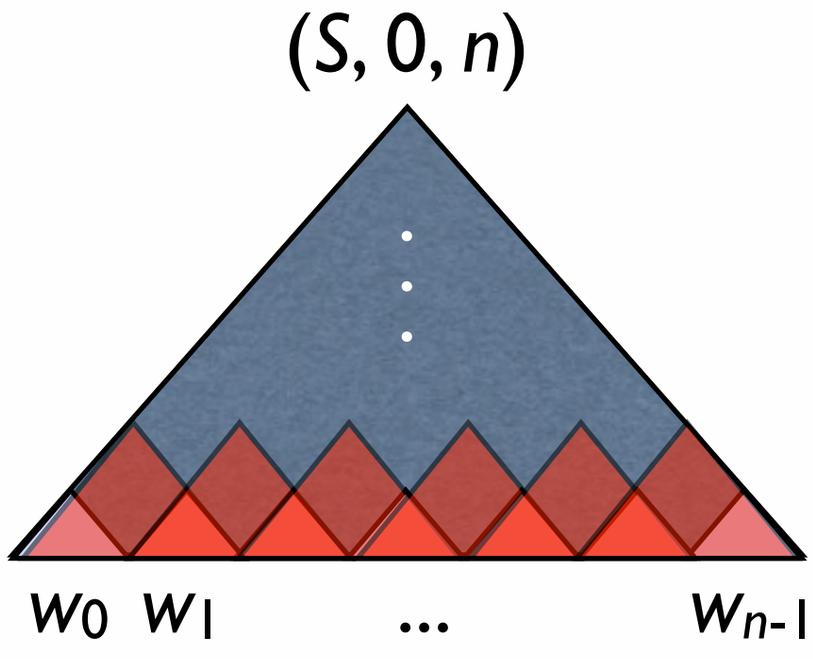



# CKY Parsing (for membership & ambiguity)

- given *any* grammar *G* and string *s*, decide membership: *s* in *L*(*G*)?
  - if *yes*, output one parse tree or *all* parse trees
  - this can decide whether *s* is ambiguous in *G*
- bottom-up dynamic programming: $O(n^3)$
  - assume *G* already in CNF (what if not?)



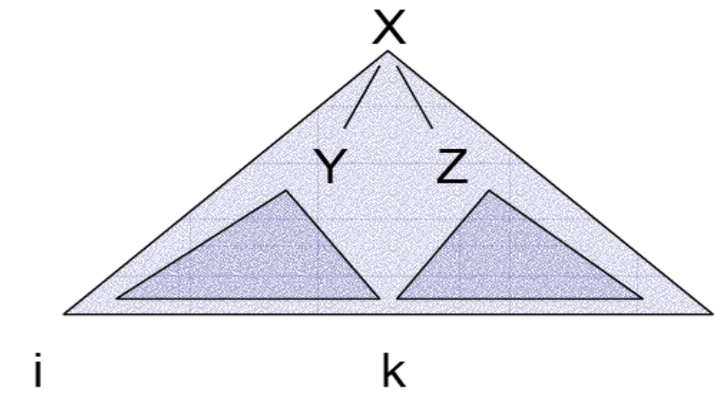

$$S \rightarrow AC \mid SS \mid AB$$
$$A \rightarrow a \quad B \rightarrow b \quad C \rightarrow SB$$



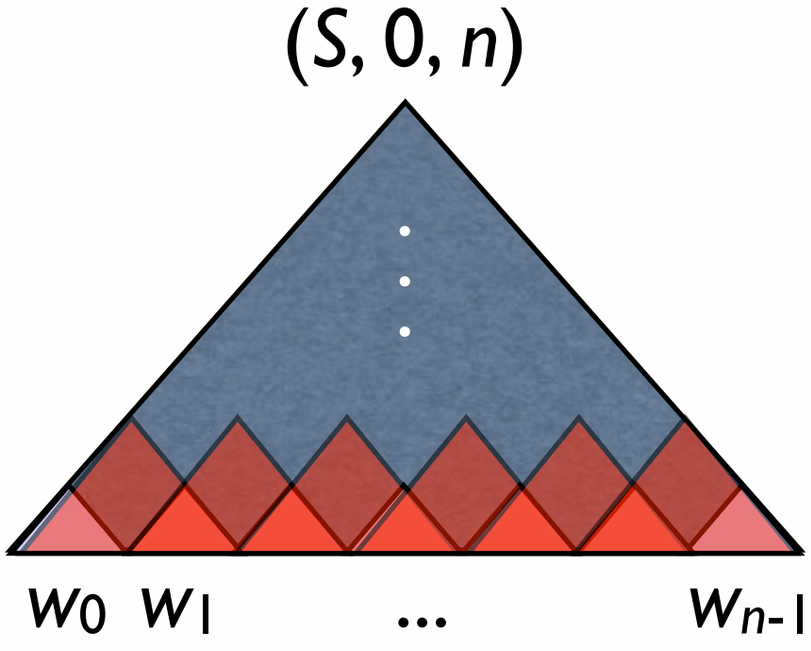

a b a b a b

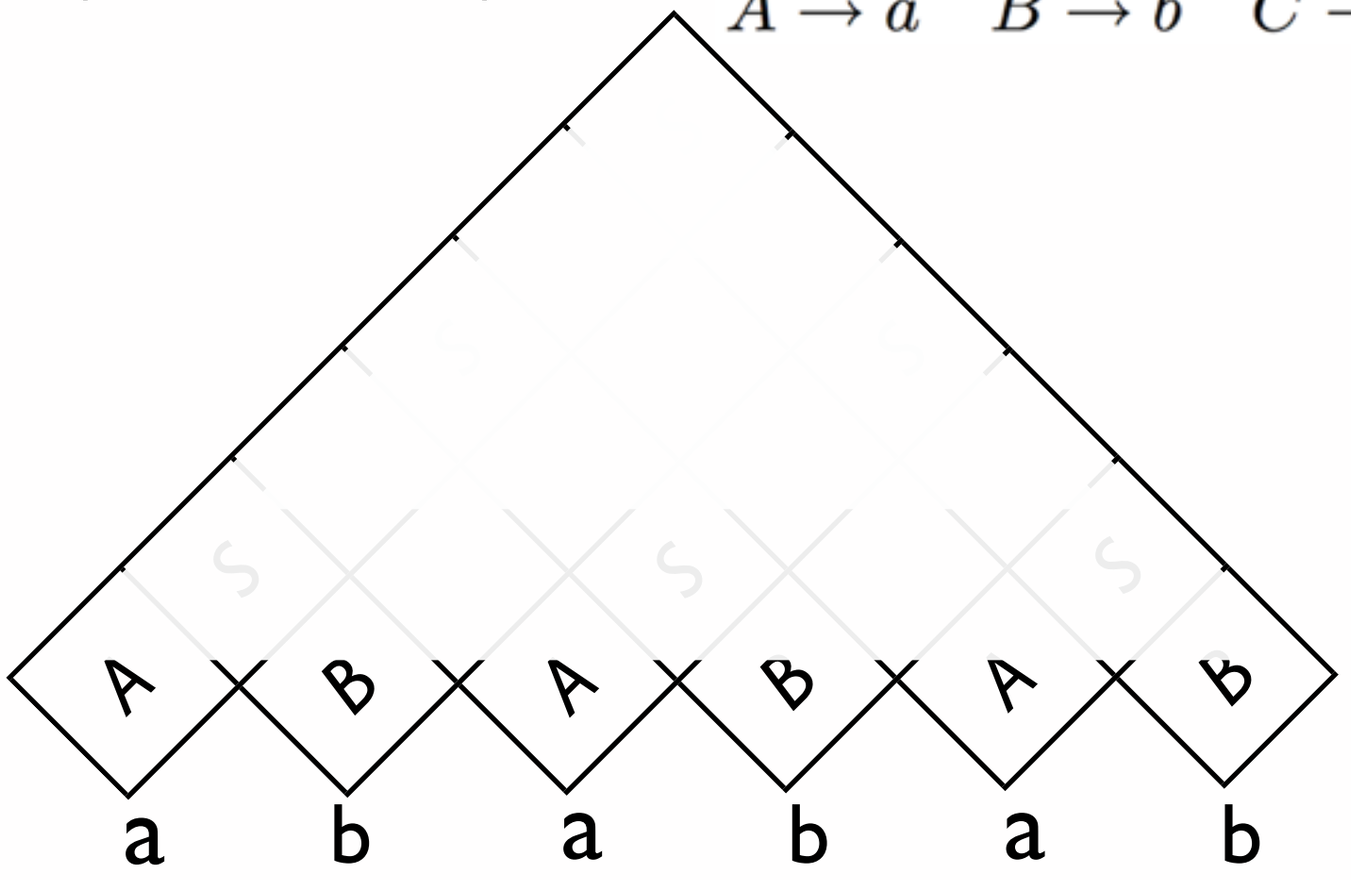
# CKY Parsing (for membership & ambiguity)

- given *any* grammar *G* and string *s*, decide membership: *s* in *L*(*G*)?
  - if *yes*, output one parse tree or *all* parse trees
  - this can decide whether *s* is ambiguous in *G*
- bottom-up dynamic programming: $O(n^3)$
  - assume *G* already in CNF (what if not?)



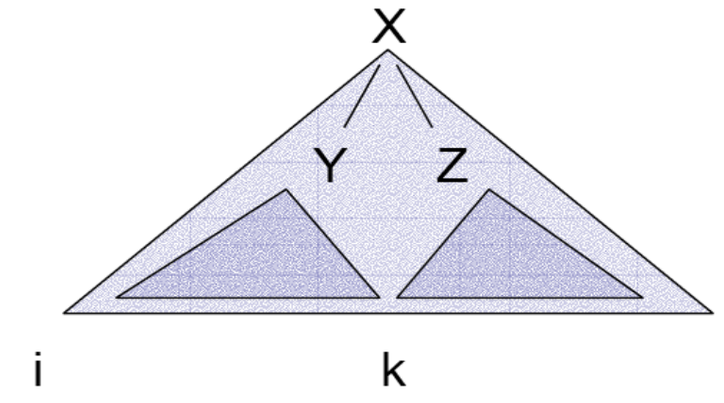

$$S \rightarrow AC \mid SS \mid AB$$
$$A \rightarrow a \quad B \rightarrow b \quad C \rightarrow SB$$



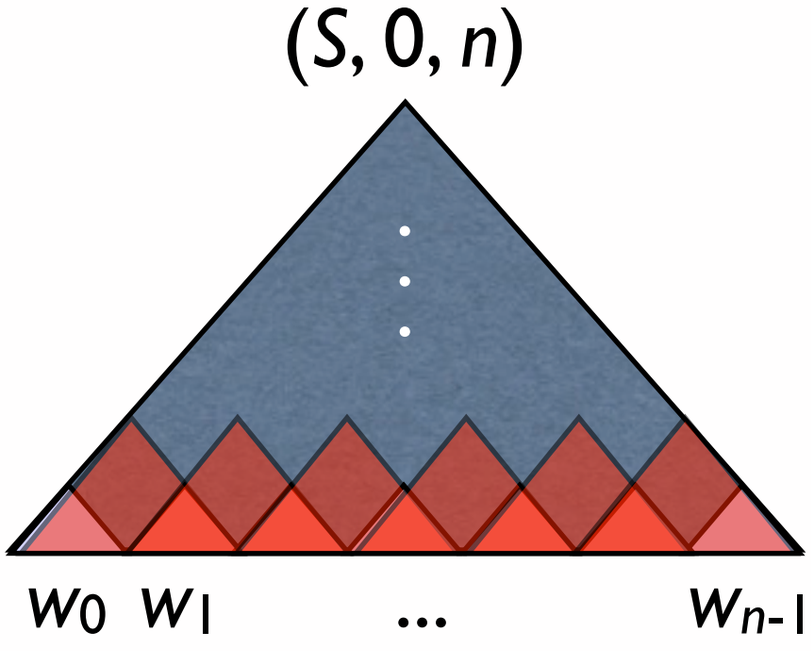



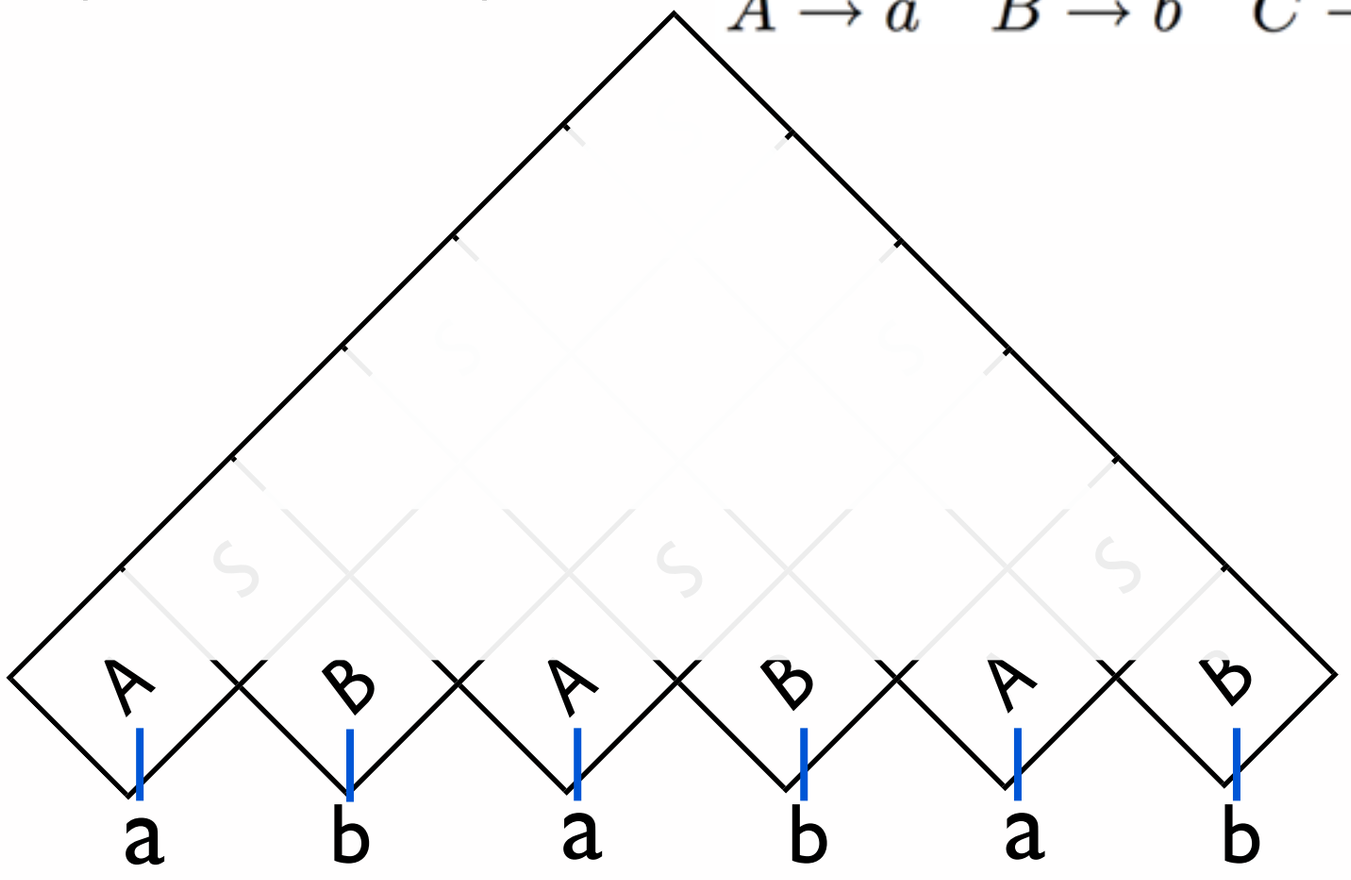
# CKY Parsing (for membership & ambiguity)

- given *any* grammar *G* and string *s*, decide membership: *s* in *L*(*G*)?
  - if *yes*, output one parse tree or *all* parse trees
  - this can decide whether *s* is ambiguous in *G*
- bottom-up dynamic programming: $O(n^3)$
  - assume *G* already in CNF (what if not?)



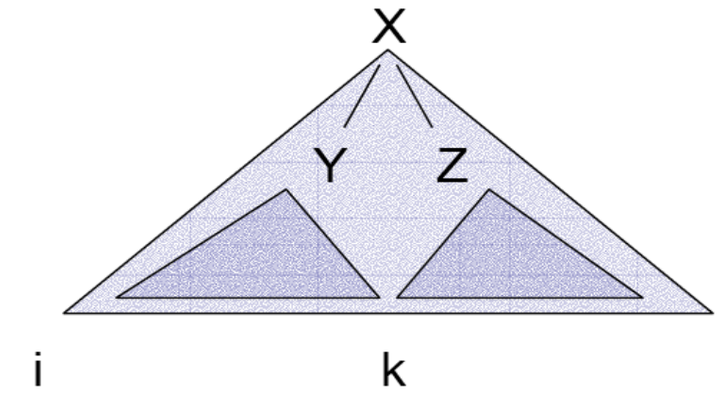

$$S \rightarrow AC \mid SS \mid AB$$
$$A \rightarrow a \quad B \rightarrow b \quad C \rightarrow SB$$



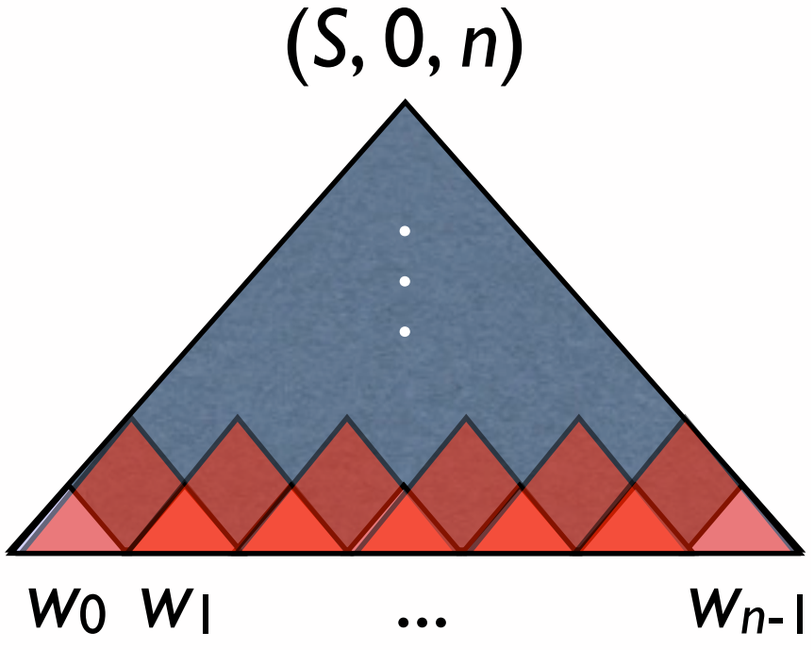



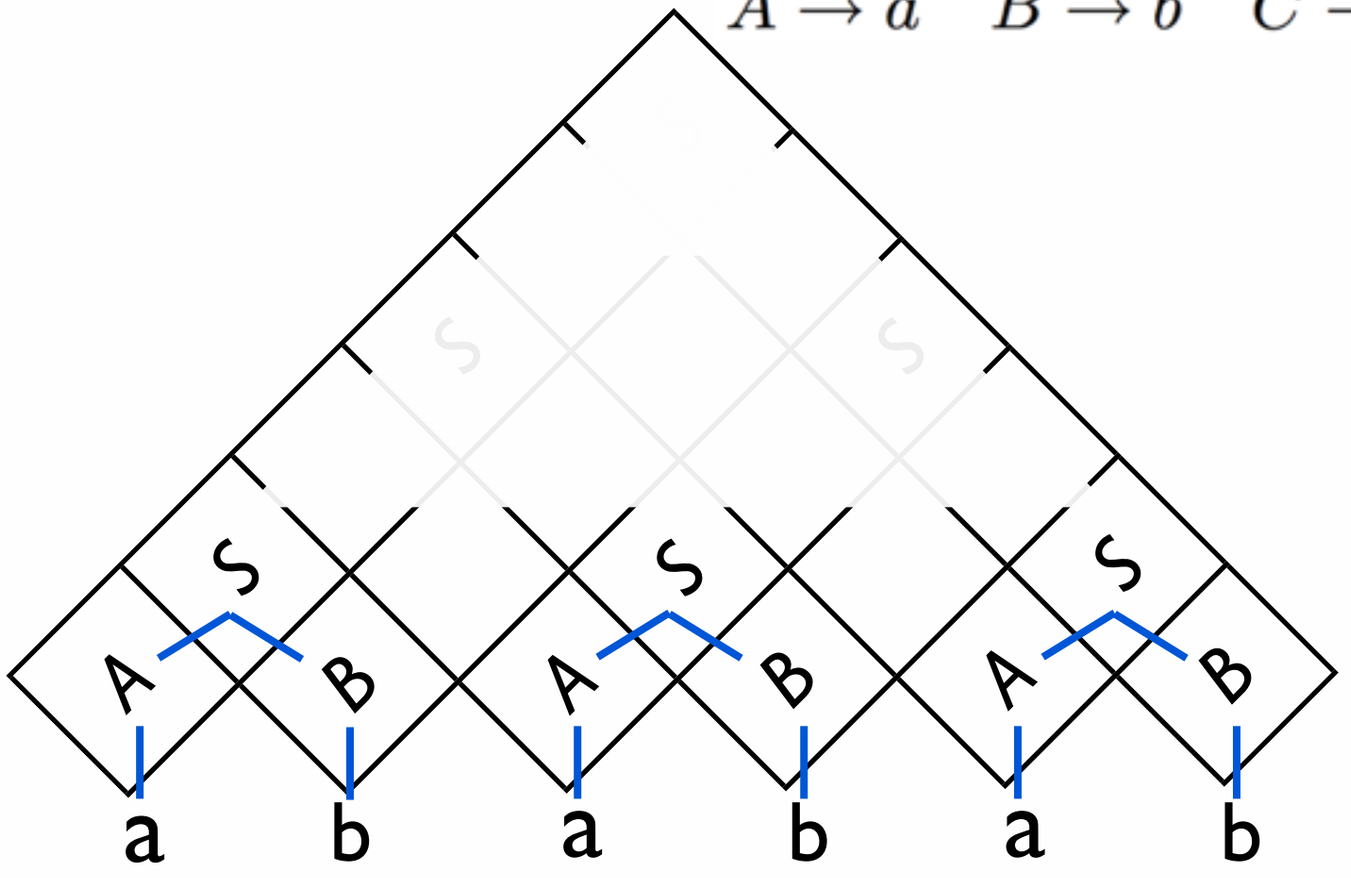
# CKY Parsing (for membership & ambiguity)

- given *any* grammar *G* and string *s*, decide membership: *s* in *L*(*G*)?
  - if *yes*, output one parse tree or *all* parse trees
  - this can decide whether *s* is ambiguous in *G*
- bottom-up dynamic programming: $O(n^3)$
  - assume *G* already in CNF (what if not?)



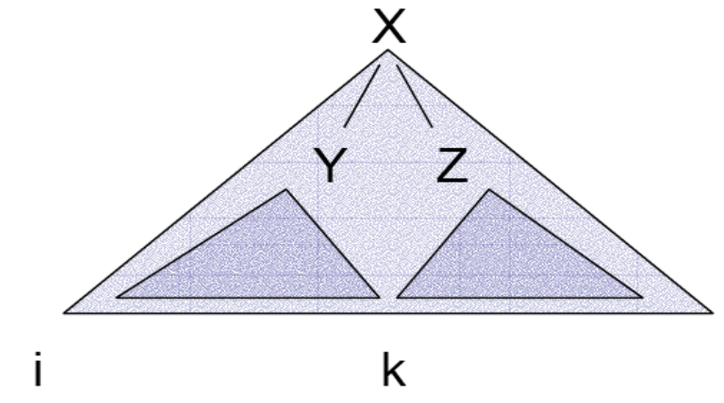

$$S \rightarrow AC \mid SS \mid AB$$
$$A \rightarrow a \quad B \rightarrow b \quad C \rightarrow SB$$



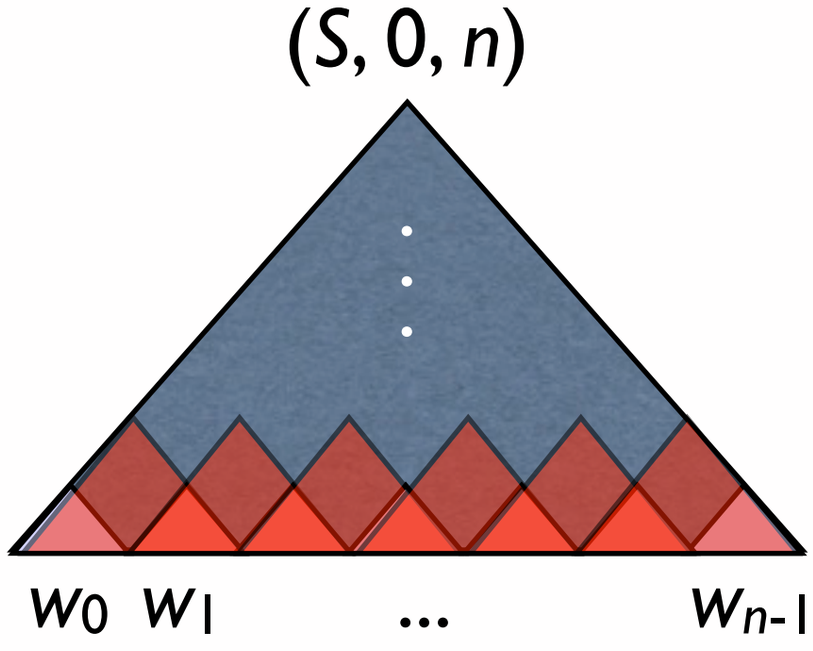



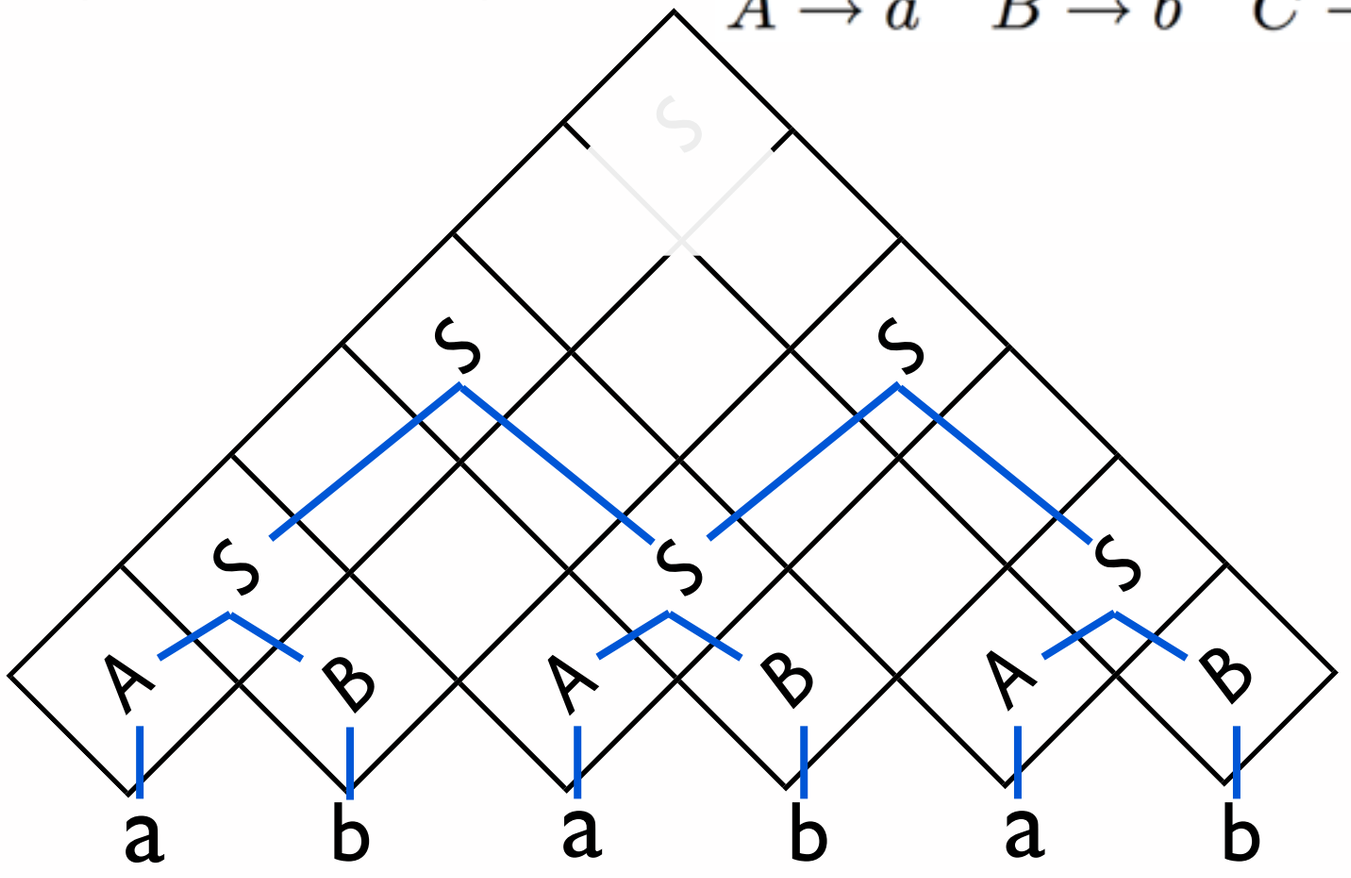
# CKY Parsing (for membership & ambiguity)

- given *any* grammar *G* and string *s*, decide membership: *s* in *L*(*G*)?
  - if *yes*, output one parse tree or *all* parse trees
  - this can decide whether *s* is ambiguous in *G*
- bottom-up dynamic programming: $O(n^3)$
  - assume *G* already in CNF (what if not?)



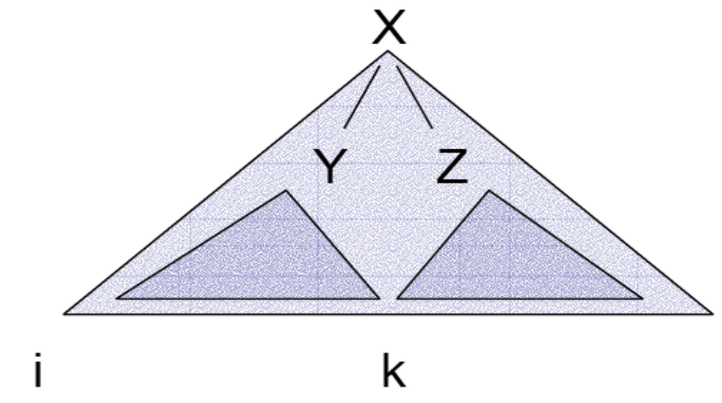

$$S \to AC \mid SS \mid AB$$
$$A \to a \quad B \to b \quad C \to SB$$



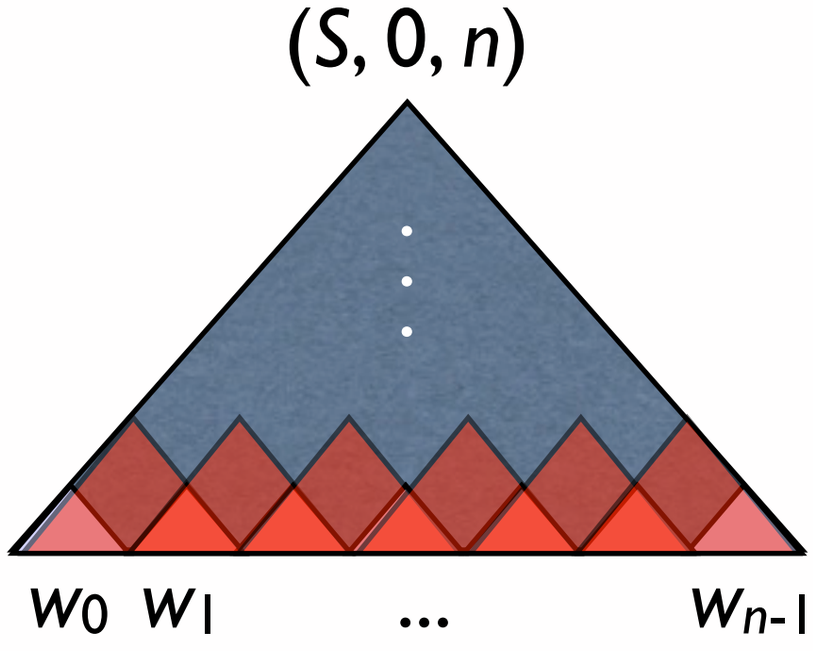



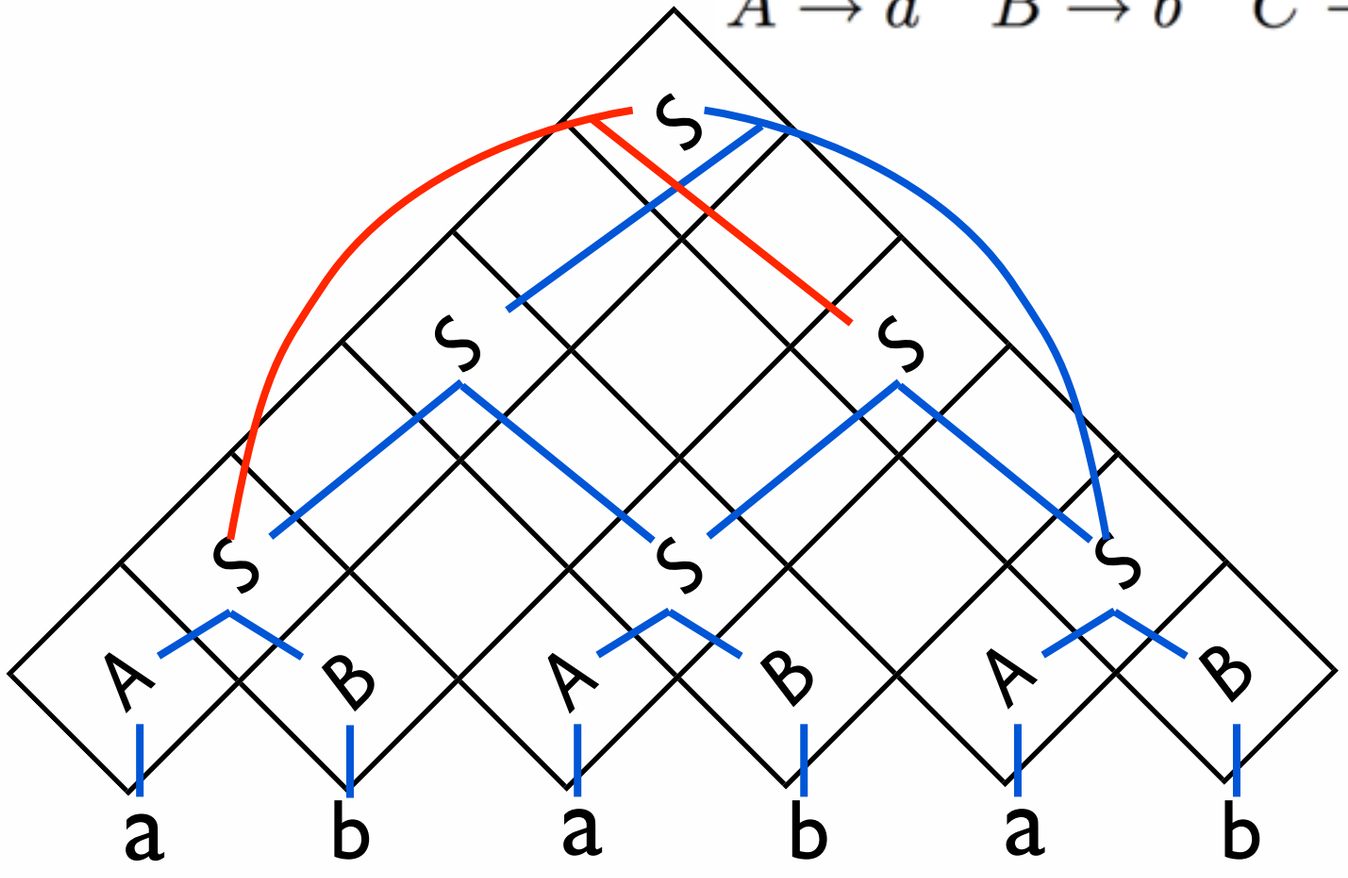
# CKY Parsing (for membership & ambiguity)

- given *any* grammar *G* and string *s*, decide membership: *s* in *L*(*G*)?
  - if *yes*, output one parse tree or *all* parse trees
  - this can decide whether *s* is ambiguous in *G*
- bottom-up dynamic programming: $O(n^3)$
  - assume *G* already in CNF (what if not?)




$$S \to AC \mid SS \mid AB$$
$$A \to a \quad B \to b \quad C \to SB$$



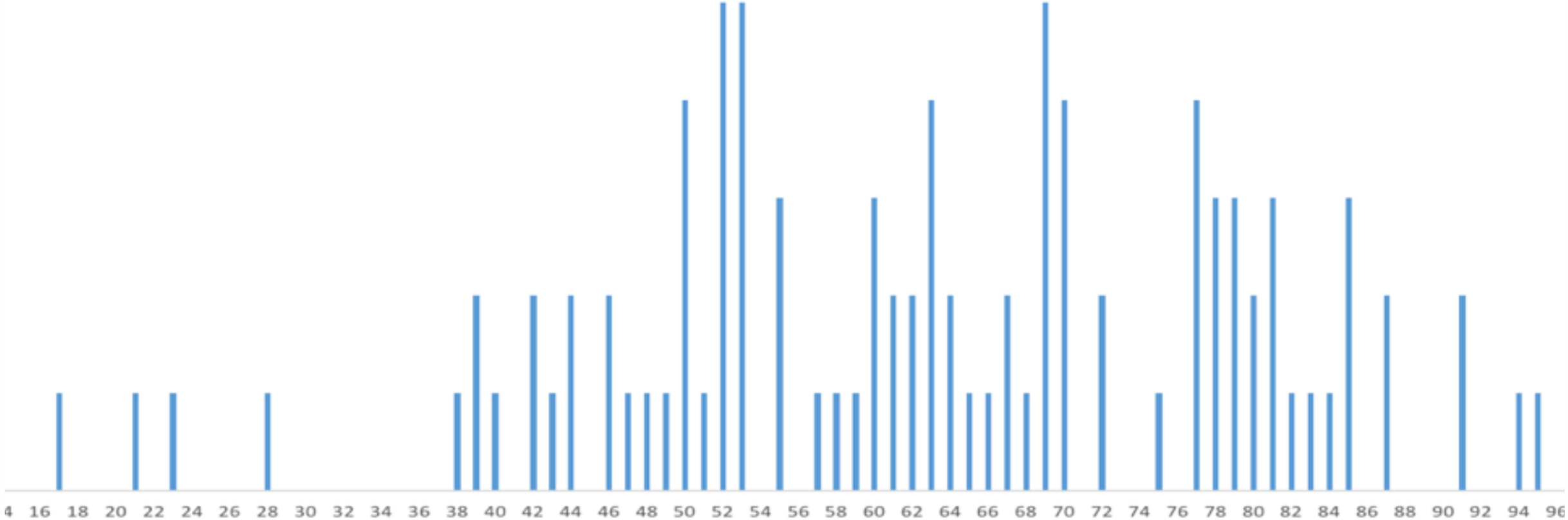
# Infix-Postfix (reverse Polish); SVO-SOV

- ambiguous infix expr grammar E-> E+E | E*E | a
  - two kinds of ambiguity: precedence and associativity
- unambiguous infix grammar (eliminates both ambiguities)
  - E -> E + T | T,  T -> T * F | F,  F-> a  (see Sipser)
- postfix grammar (unambiguous) -- E -> E E + | E E * | a

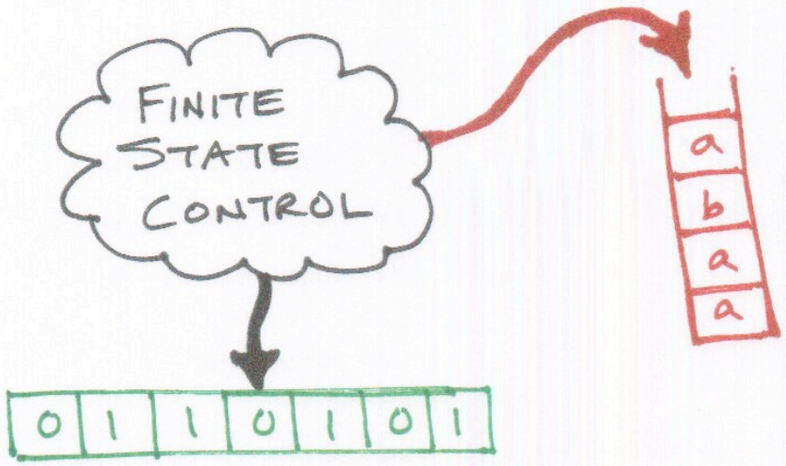
- in natural language, infix ~ SVO, postfix ~ SOV
  - SVO: I eat sushi with tuna = I + s * t (attachment ambiguity)
  - SOV: no attachment ambiguity, but need case marker
    - to separate two nouns (5␣6 +); infix separates them by verb/prep (5+6*7)
    - I*subj* sushi*obj* eat
      Watashi-*wa* sushi-*o* tabemasu
      私 は 寿司 を 食べます

# Grades so far (62%) - last day to withdraw

- midterm: 20+5=25, quizzes: 5x6=30, homework: 1x7=7
- quiz 7 will be the last quiz
  - the other 5% is "corrections to your worst 5 quizzes/exams"
- please bring *clickers* on the Wednesday before Thanksgiving



# Pushdown Automata (FSA w/ stack)

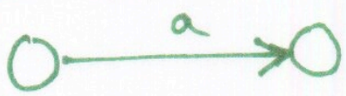



Input String:
- Same as F.S.M.
- (cannot back up)

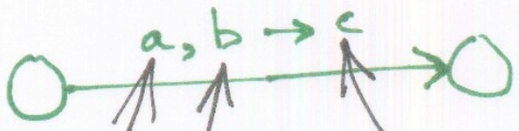
Stack
- Operations:
  - Read+pop / Ignore
  - push / Ignore
- Stack Alphabet
  - $\Gamma$ Gamma,
    May be different from Input Alphabet, $\Sigma$

State Transitions:
- May depend on stack top.
- May push onto the stack.
- **Non-Deterministic!**

## Finite State Machine




## Pushdown Automaton




Input symbol

May "ϵ" be

Symbol on top of the stack.

This symbol is popped.

"ϵ" means the stack is neither read nor popped

This symbol is pushed onto the stack.

"ϵ" means nothing is pushed.

## Nondeterminism

# Example: $0^n 1^n$

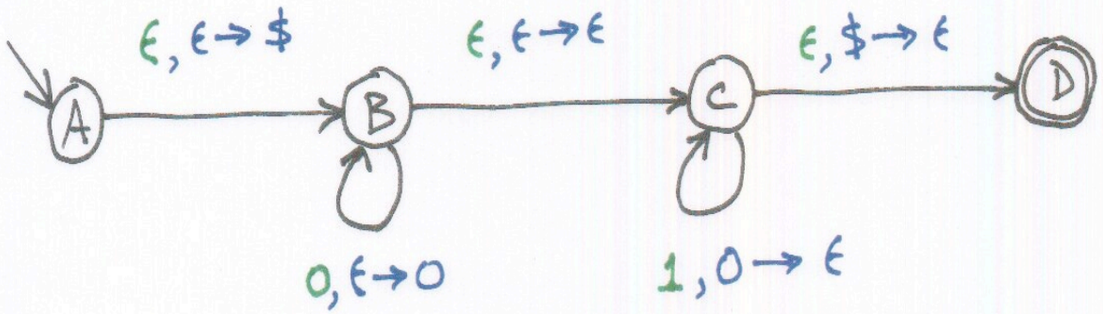
$\{0^n 1^n \mid n \geq 0\}$

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

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

↑ To detect bottom of stack.

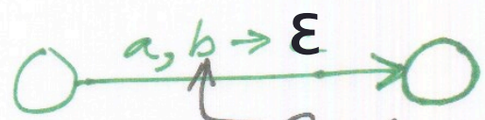
(We could use other symbols, but these seem clear enough.)

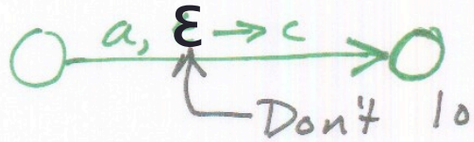



## When is a string accepted?

- Begin in the start state.
- End in an Accept state.
- Consume all the input symbols.
  (Okay to leave stuff on the stack.)
- There is a path thru the Finite State Control.

Note: It is not possible to pop an empty stack.




- Non-Deterministic:
  You just have to find one path to a Accept state.

# Definition of PDA

## Formal Definition

$(Q, \Sigma, \Gamma, \delta, q_0, F)$

$Q$ = Set of states

$\Sigma$ = Input alphabet

$\Gamma$ = Stack alphabet

$\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$

$\Gamma_\epsilon = \Gamma \cup \{\epsilon\}$

$\delta: Q \times \Sigma_\epsilon \times \Gamma_\epsilon \to \mathcal{P}(Q \times \Gamma_\epsilon)$

$q_0$: Starting State $q_0 \in Q$

$F$: Accept States $F \subseteq Q$

## EXAMPLE

Palindrome

MADAM IM ADAM

WAS IT A CAT I SAW

NO LEMON, NO MELON

$\{w \mid w \text{ is a palindrome AND } w \in \{0,1\}^*\}$

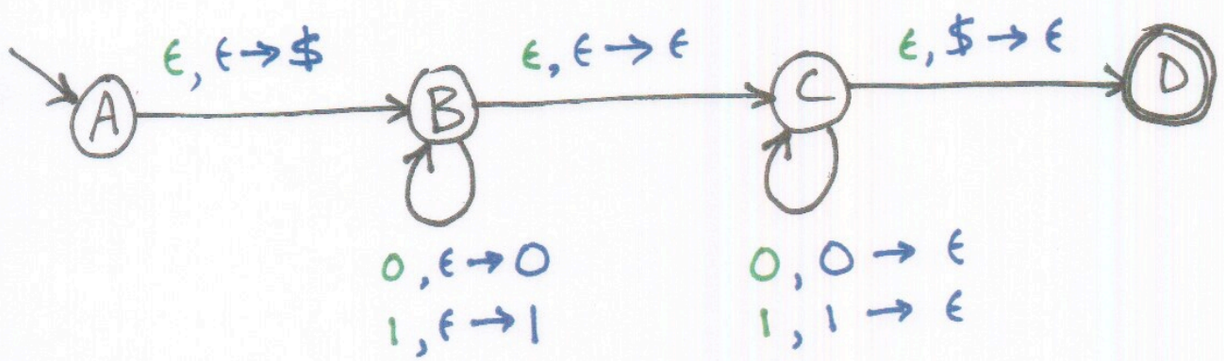
$S \to 0S0$

$\to 1S1$

$\to \epsilon$

wrong! this is $ww^R$

add S-> 0 | 1 to be palindrome



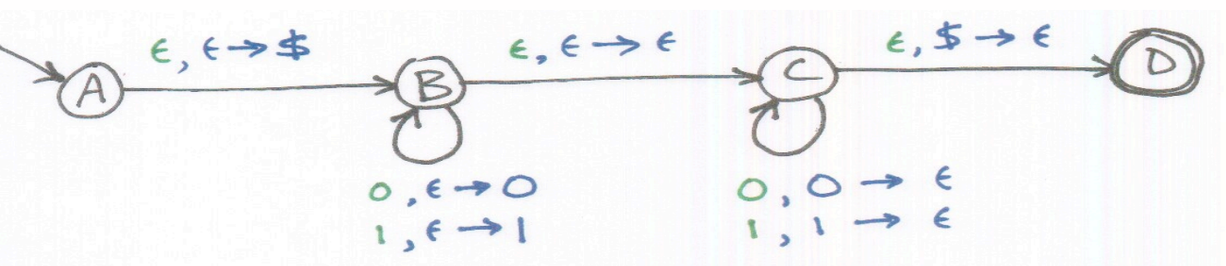
# Deterministic PDA (DPDA)

- by default PDA is non-deterministic; DPDA: one branch at each state
- DPDA is strictly less powerful than PDA: can't do $ww^R$, *palindrome*, etc.

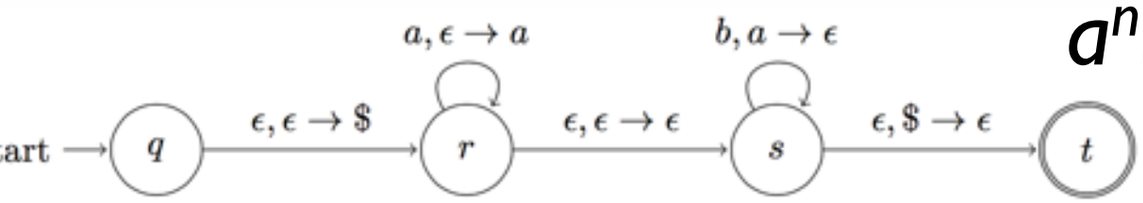
**DEFINITION 2.13**

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

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




$a^n b^n$




$ww^R$ only by PDA

**DEFINITION 2.39**

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

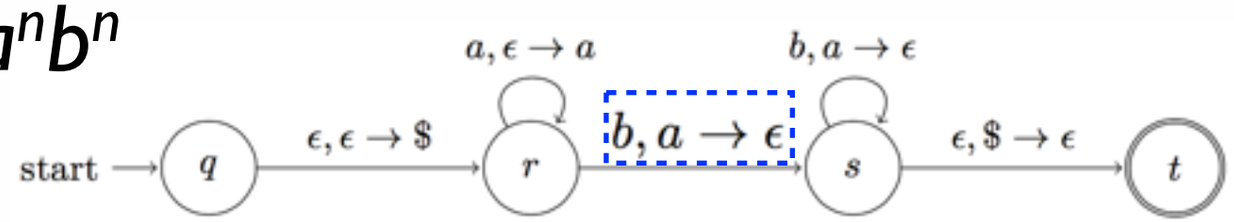
1. $Q$ is the set of states,
2. $\Sigma$ is the input alphabet,
3. $\Gamma$ is the stack alphabet,
4. $\delta: Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow (Q \times \Gamma_\varepsilon) \cup \{\emptyset\}$ is the transition function,
5. $q_0 \in Q$ is the start state, and
6.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

The transition function $\delta$ must satisfy the following condition. For every $q \in Q$, $a \in \Sigma$, and $x \in \Gamma$, exactly one of the values

$$\delta(q, a, x), \delta(q, a, \varepsilon), \delta(q, \varepsilon, x), \text{ and } \delta(q, \varepsilon, \varepsilon)$$

is not $\emptyset$.

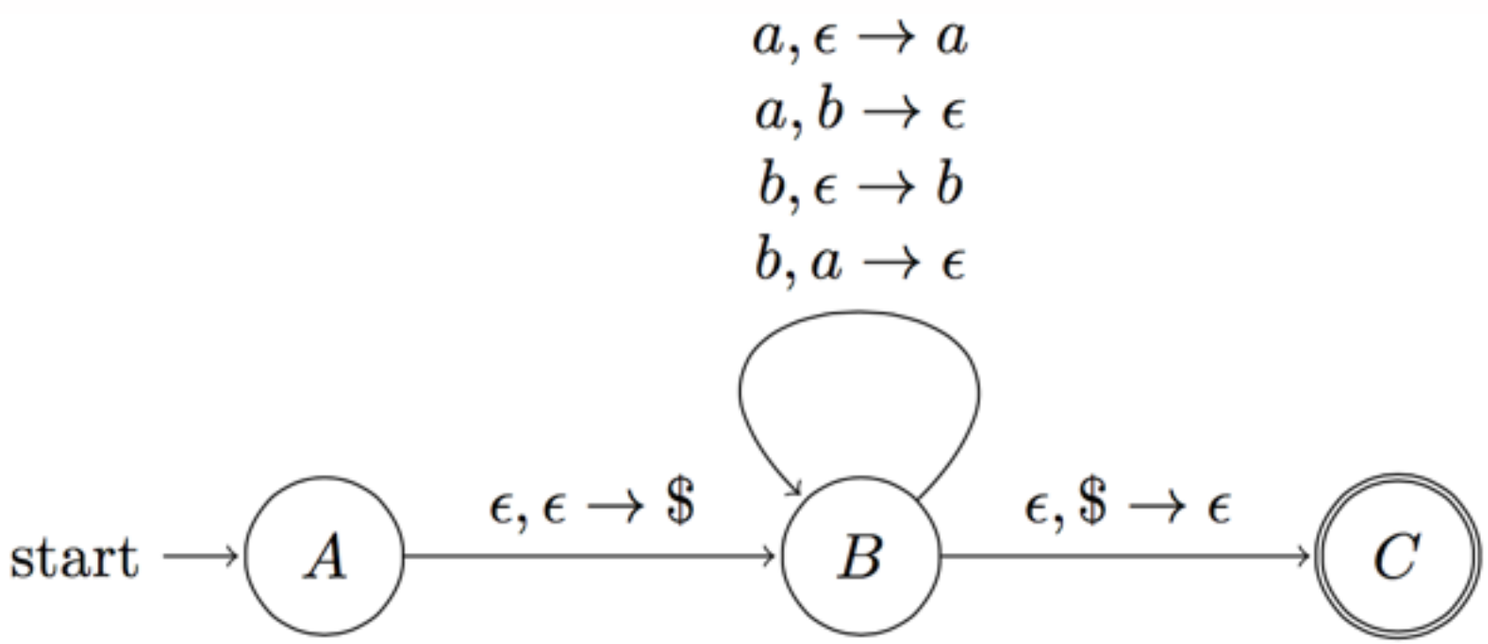
$a^n b^n$




w#w$^R$ by DPDA

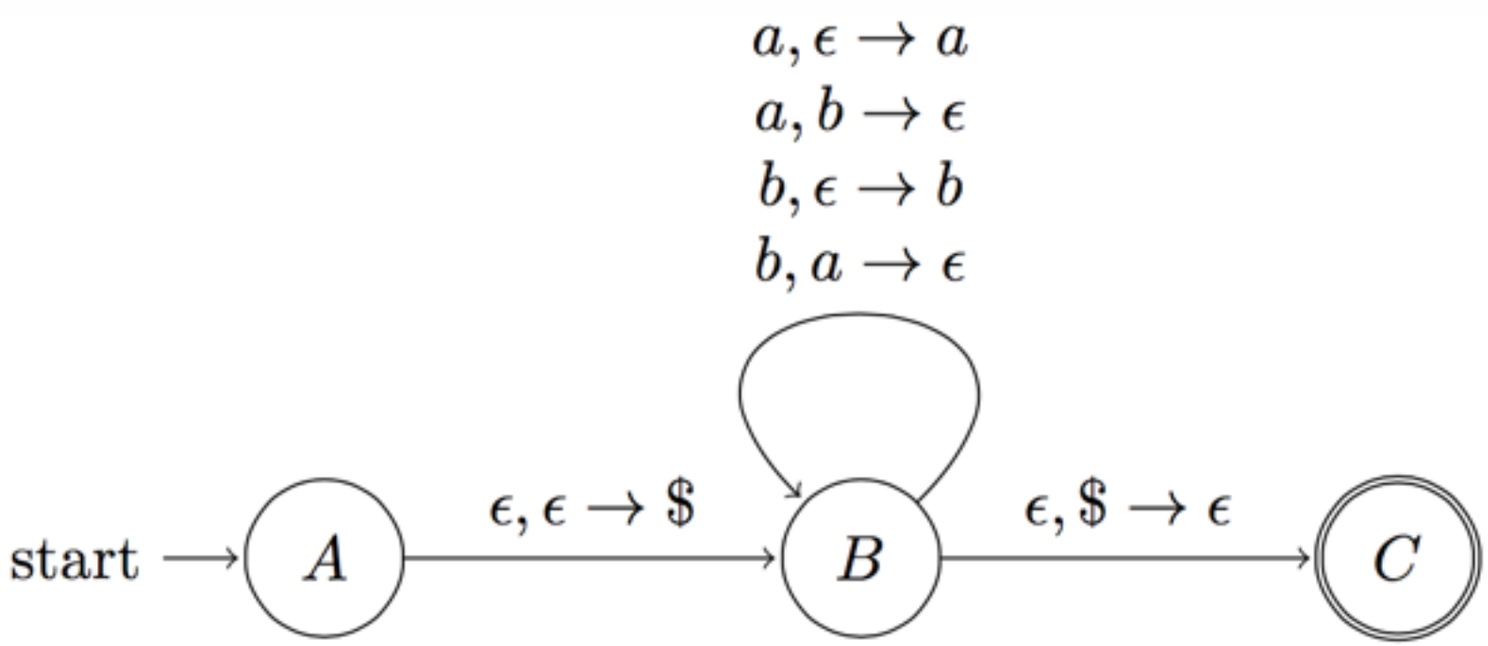
# Same number of a's and b's

- and how about twice many a's as b's?



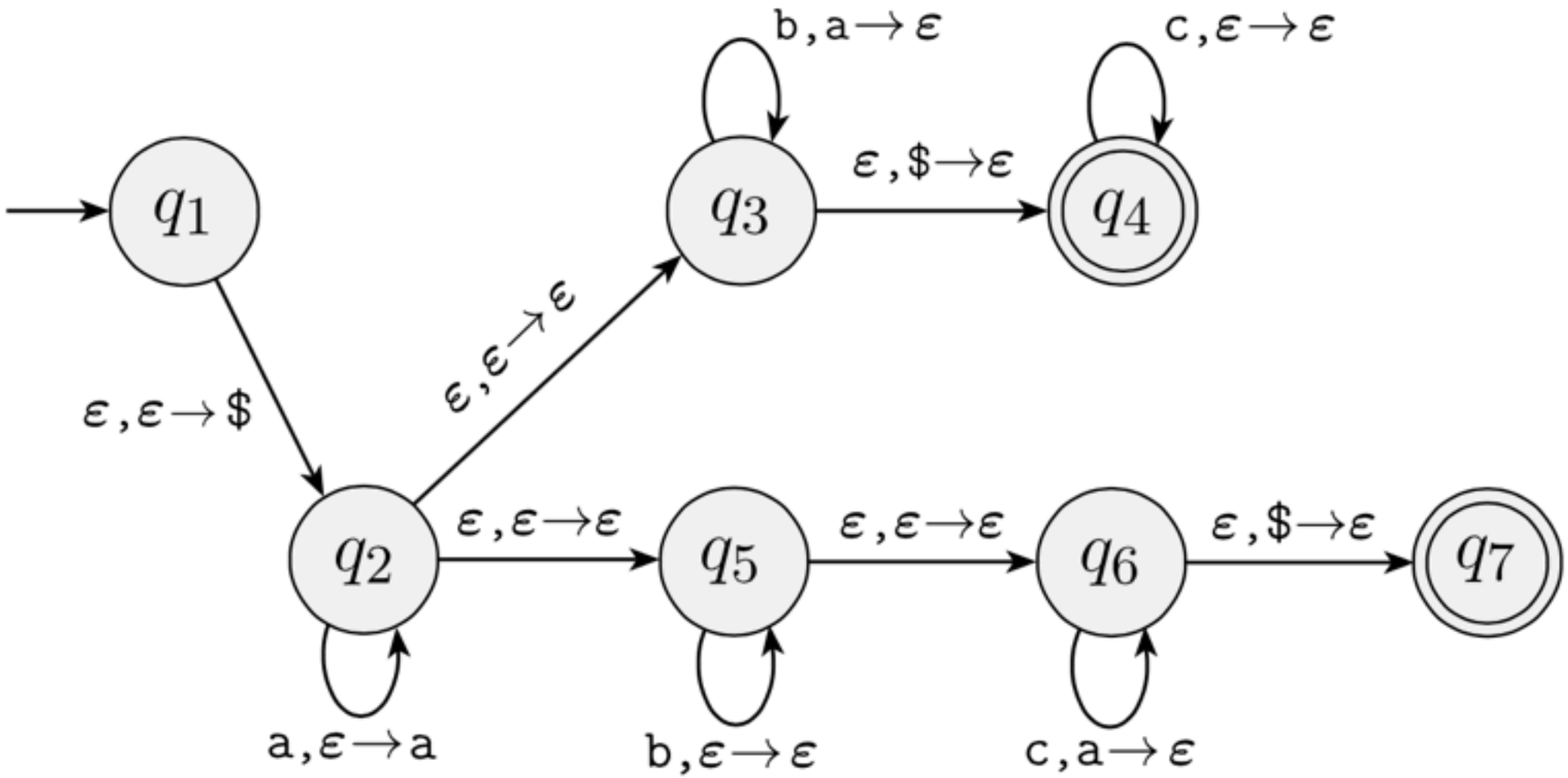
# Same number of a's and b's

- and how about twice many a's as b's?



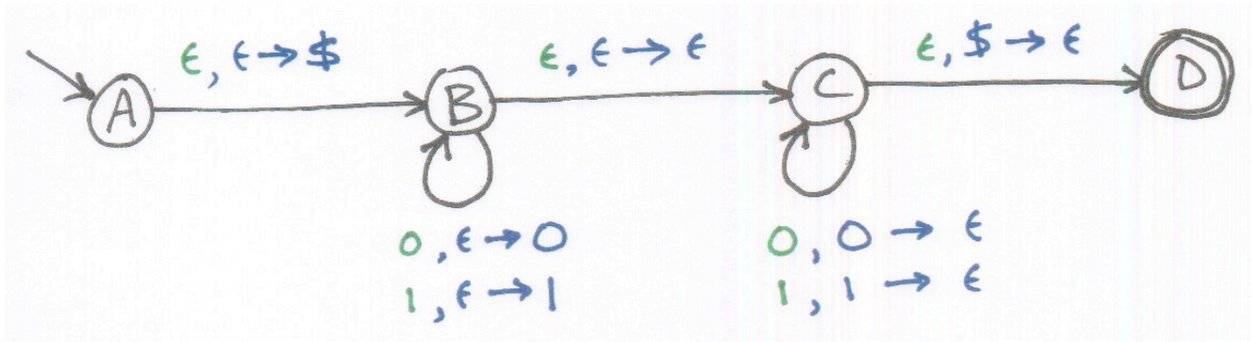

(a,ε,A)
(a,B,ε)
(b,AA,ε)
(b,ε,BB)
(b,A,B)

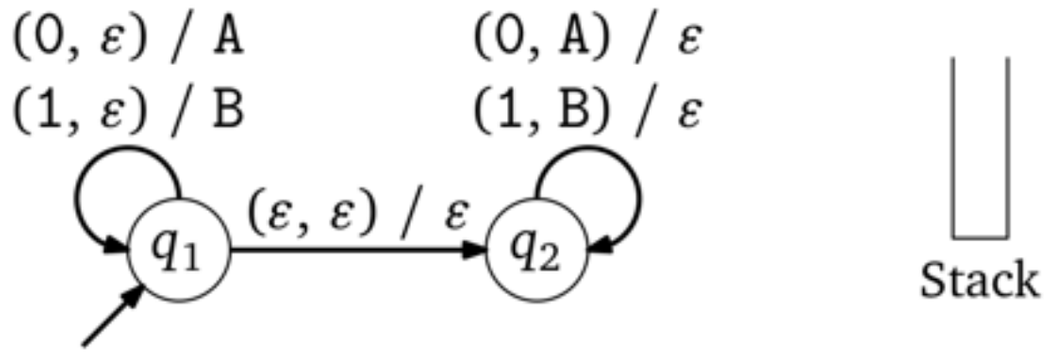
# $\{ a^i b^j c^k \mid i=j \text{ or } j=k \}$



# Acceptance by Empty Stack

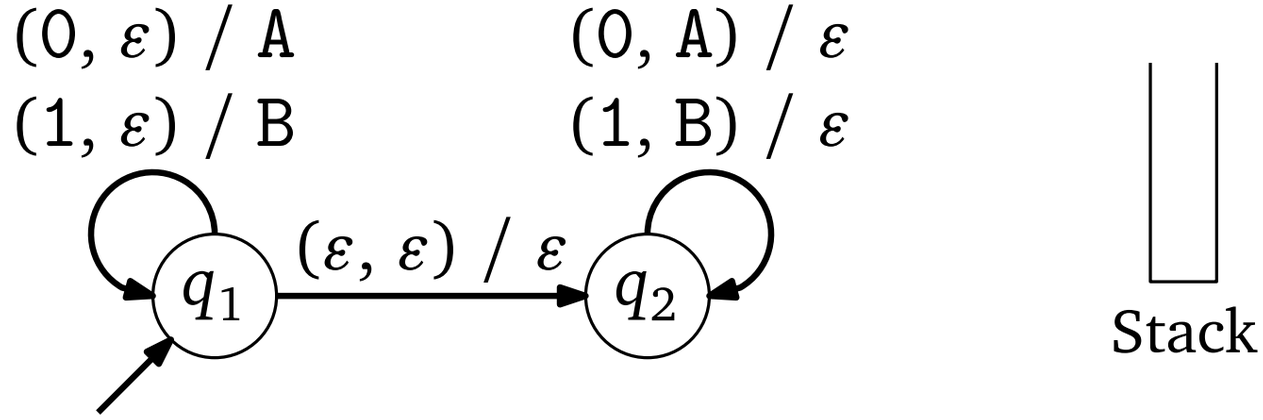
- by default, PDA can't detect end of stack (so we add $)
- if PDA can accept by empty stack it could be simpler



# Push-Down Automata: Example

A PDA for the language $\{\sigma\sigma^R \mid \sigma \in \{0,1\}^*\}$.

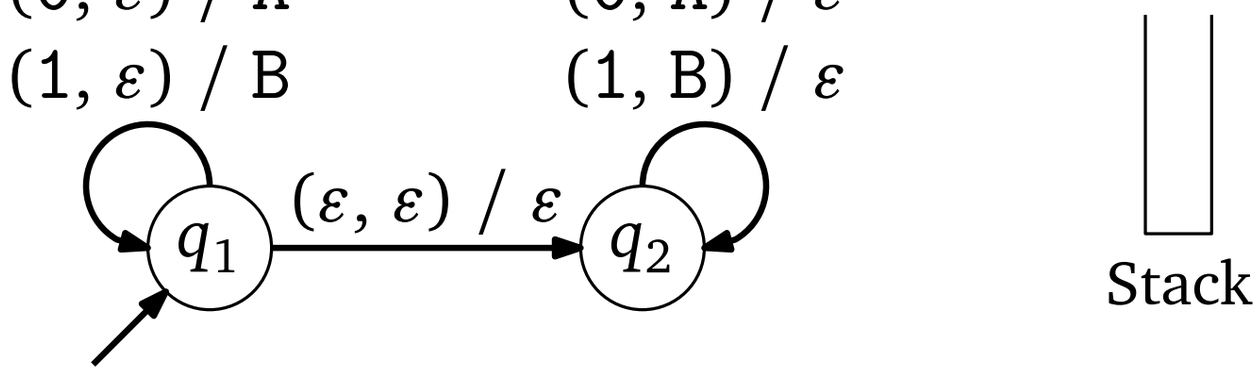


# Push-Down Automata: Example

A PDA for the language $\{\sigma\sigma^R \mid \sigma \in \{0,1\}^*\}$.

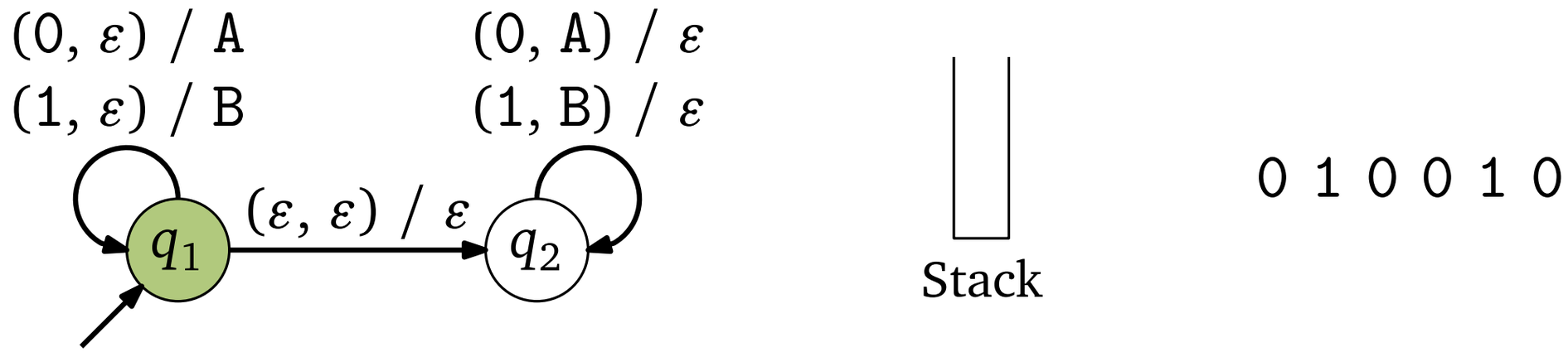
If the next input symbol is a 0, then push an A onto the stack.

If the next input symbol is a 0 and the top of the stack is A, then pop the A and push nothing onto the stack.



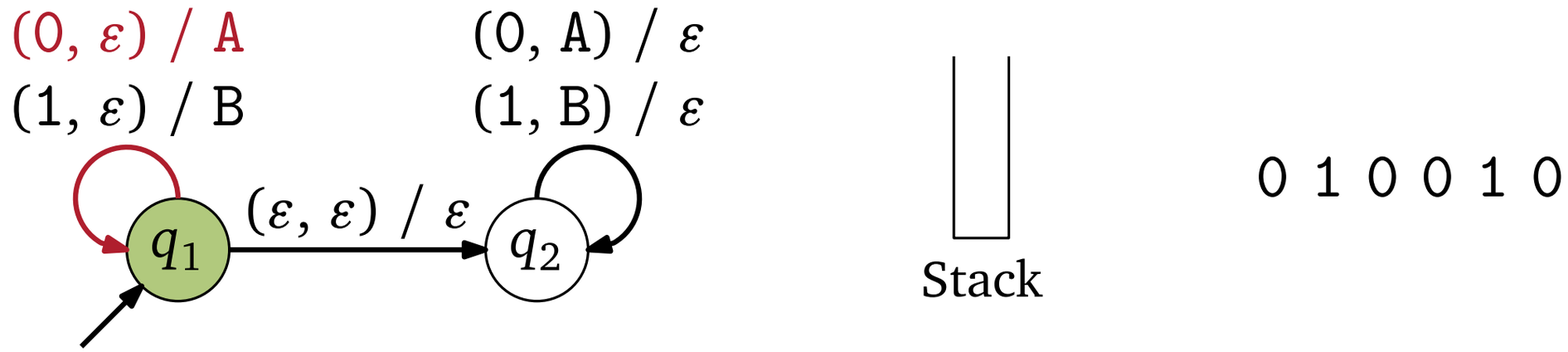
# Push-Down Automata: Example

A PDA for the language $\{\sigma\sigma^R \mid \sigma \in \{0,1\}^*\}$.



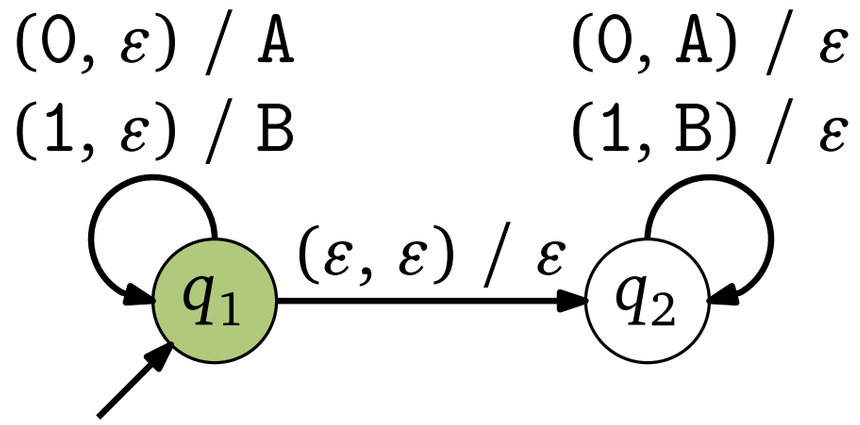
# Push-Down Automata: Example

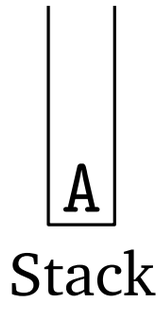
A PDA for the language $\{\sigma\sigma^R \mid \sigma \in \{0,1\}^*\}$.



# Push-Down Automata: Example

A PDA for the language $\{\sigma\sigma^R \mid \sigma \in \{0,1\}^*\}$.

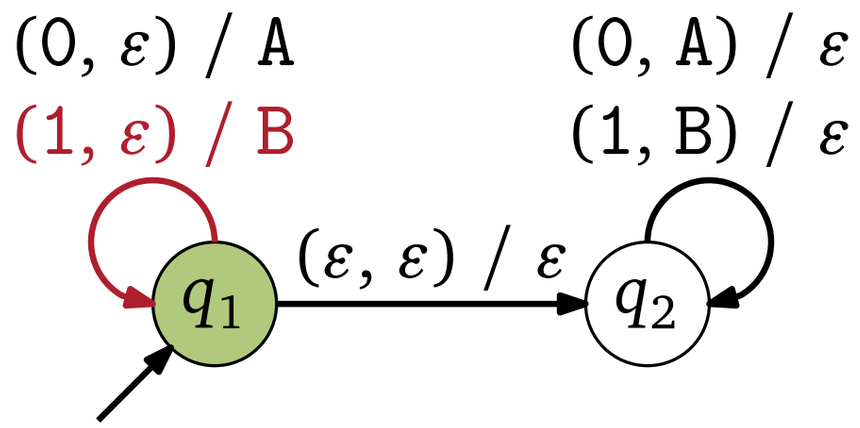


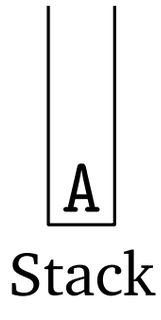




1 0 0 1 0

# Push-Down Automata: Example

A PDA for the language $\{\sigma\sigma^R \mid \sigma \in \{0,1\}^*\}$.

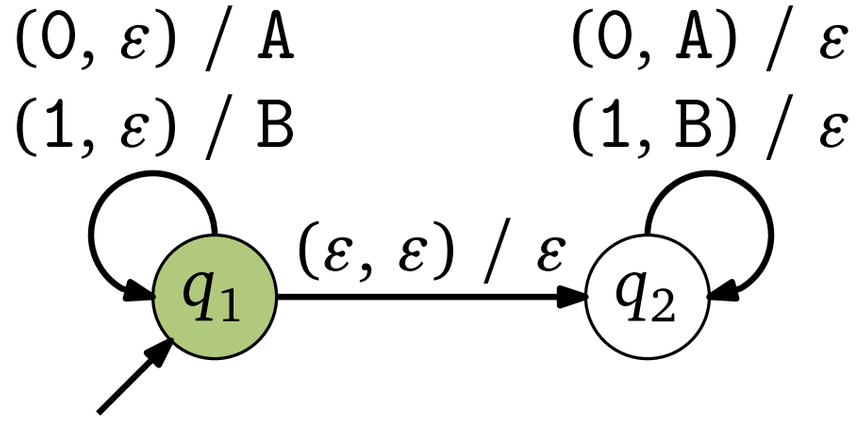


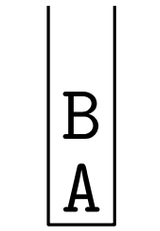




1 0 0 1 0

# Push-Down Automata: Example

A PDA for the language $\{\sigma\sigma^R \mid \sigma \in \{0,1\}^*\}$.

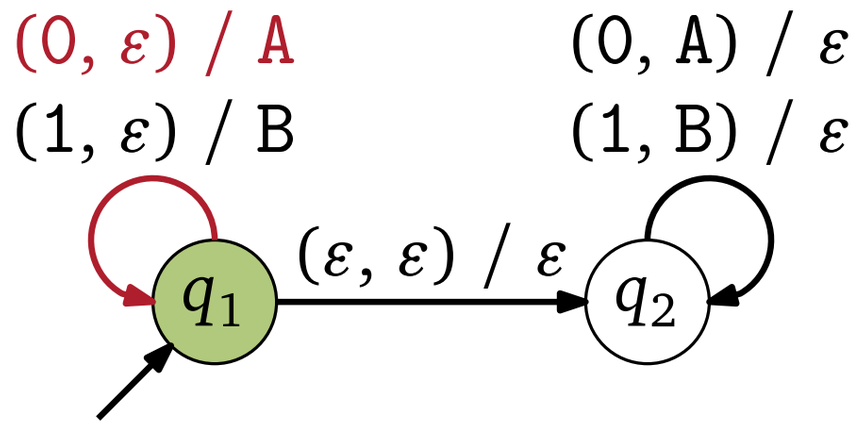


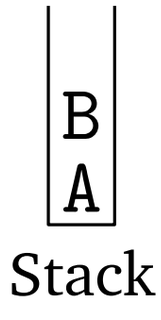




0 0 1 0

# Push-Down Automata: Example

A PDA for the language $\{\sigma\sigma^R \mid \sigma \in \{0,1\}^*\}$.

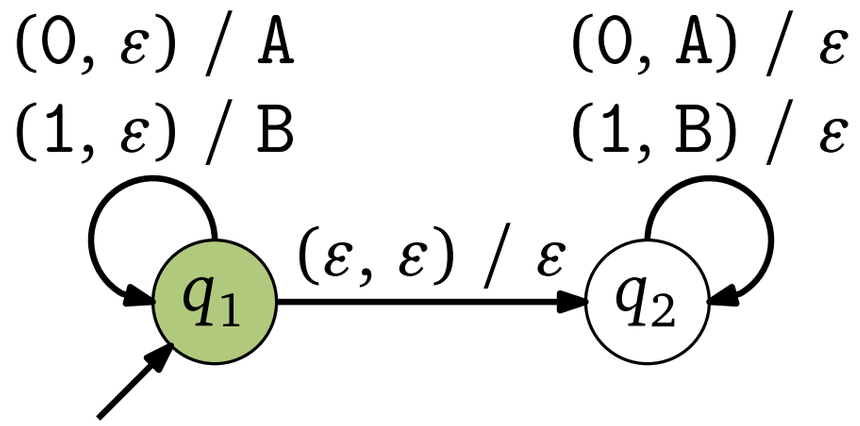


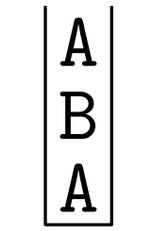




0 0 1 0

# Push-Down Automata: Example

A PDA for the language $\{\sigma\sigma^R \mid \sigma \in \{0,1\}^*\}$.

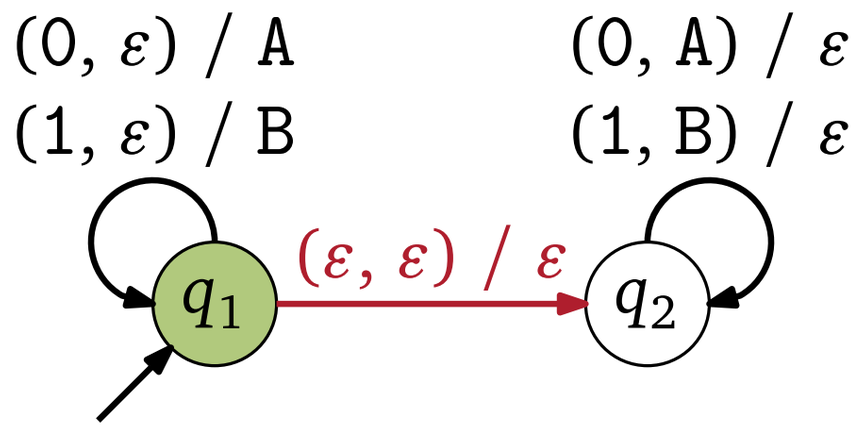


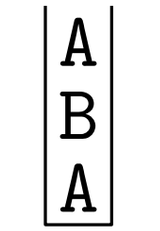




0 1 0

# Push-Down Automata: Example

A PDA for the language $\{\sigma\sigma^R \mid \sigma \in \{0,1\}^*\}$.



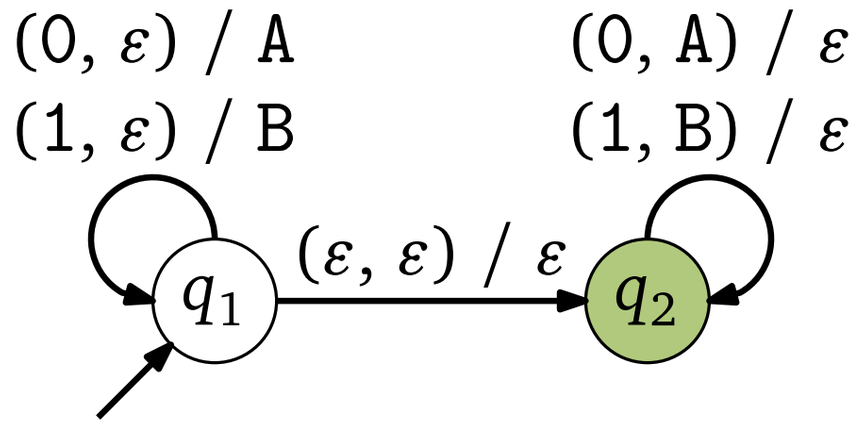



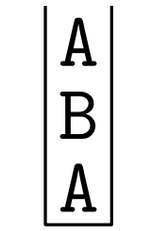

Stack

0 1 0

# Push-Down Automata: Example

A PDA for the language $\{\sigma\sigma^R \mid \sigma \in \{0,1\}^*\}$.



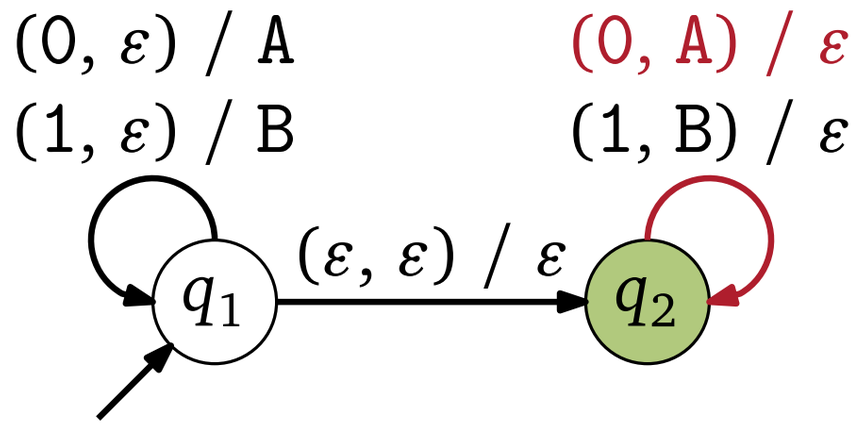



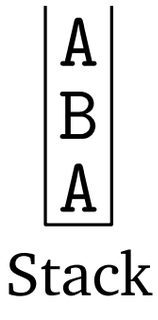

Stack

0 1 0

# Push-Down Automata: Example

A PDA for the language $\{\sigma\sigma^R \mid \sigma \in \{0,1\}^*\}$.

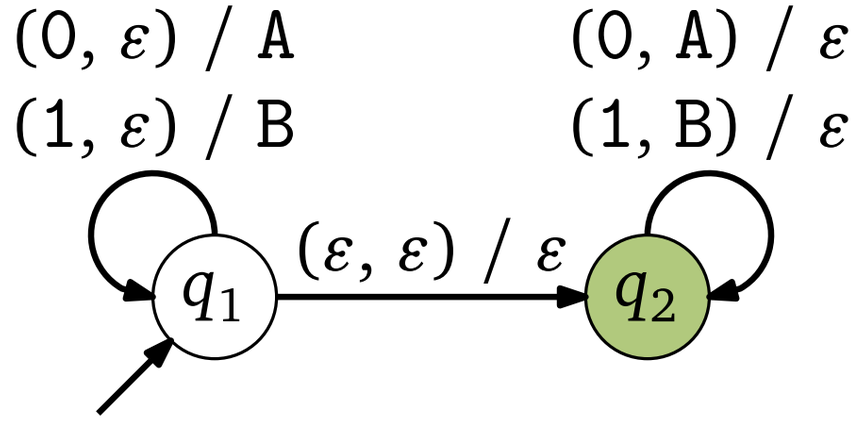


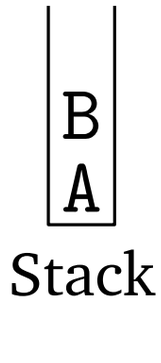




0 1 0

# Push-Down Automata: Example

A PDA for the language $\{\sigma\sigma^R \mid \sigma \in \{0,1\}^*\}$.

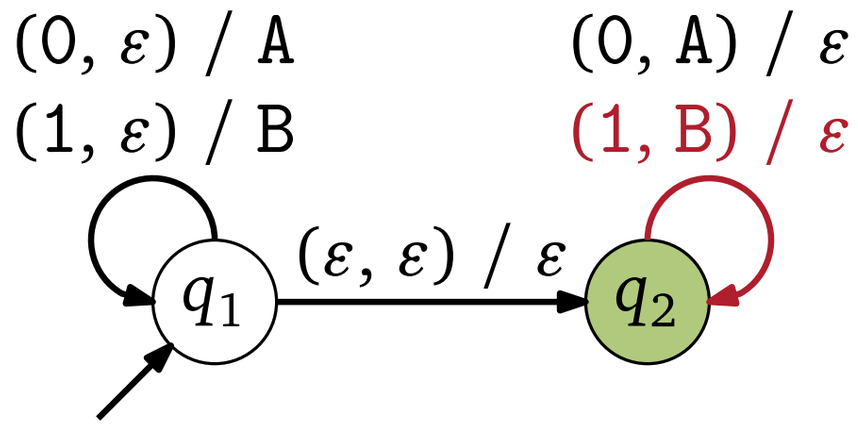


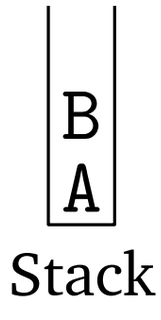




1 0

# Push-Down Automata: Example

A PDA for the language $\{\sigma\sigma^R \mid \sigma \in \{0,1\}^*\}$.

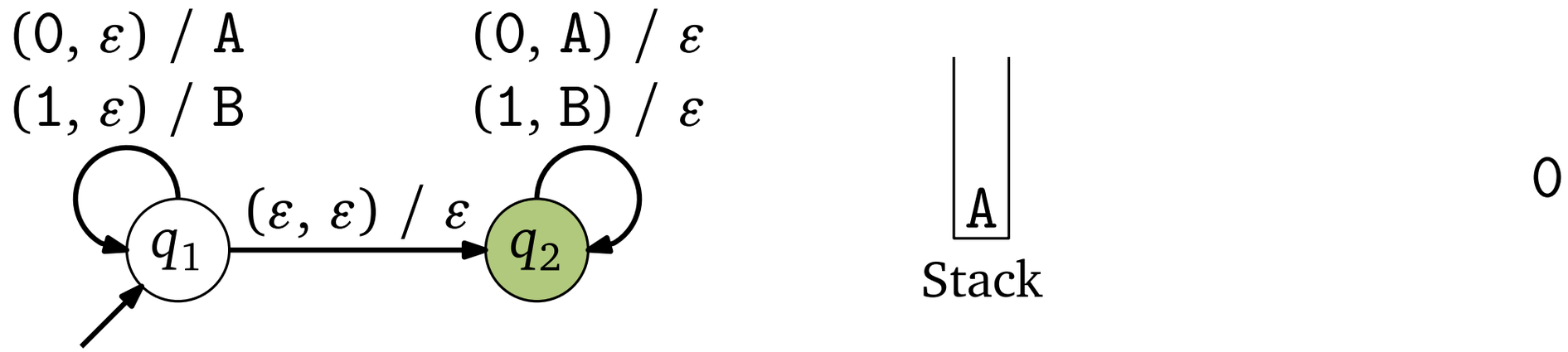



1 0

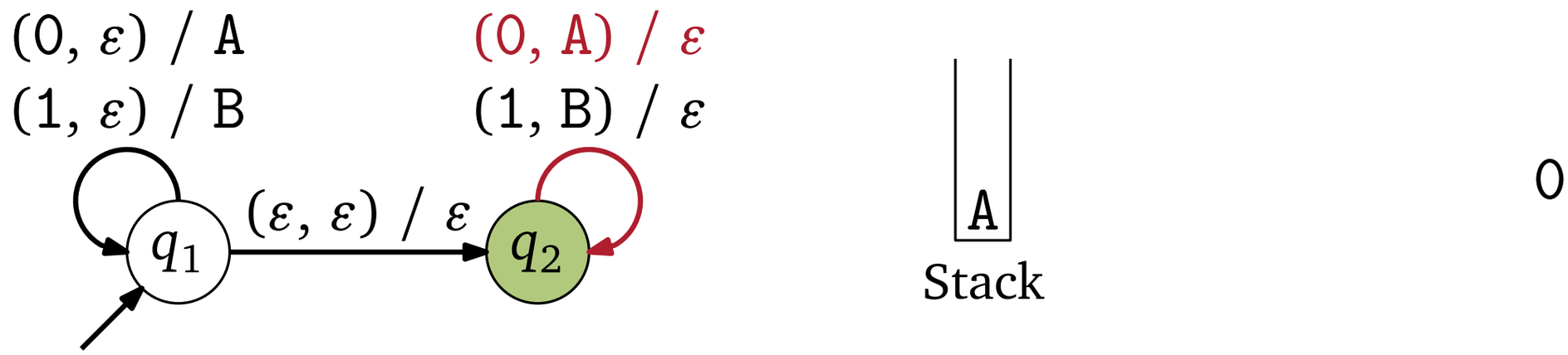
# Push-Down Automata: Example

A PDA for the language $\{\sigma\sigma^R \mid \sigma \in \{0,1\}^*\}$.



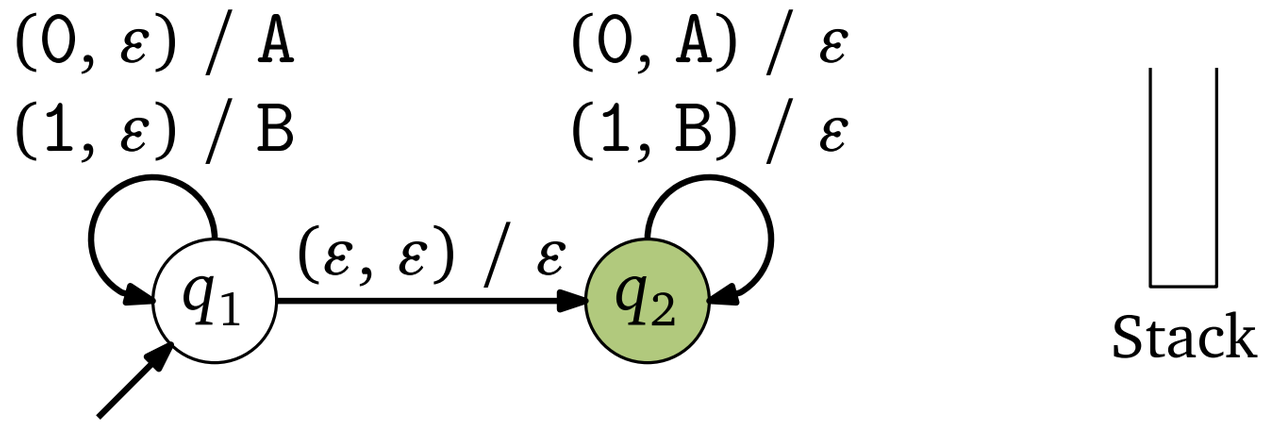
# Push-Down Automata: Example

A PDA for the language $\{\sigma\sigma^R \mid \sigma \in \{0,1\}^*\}$.



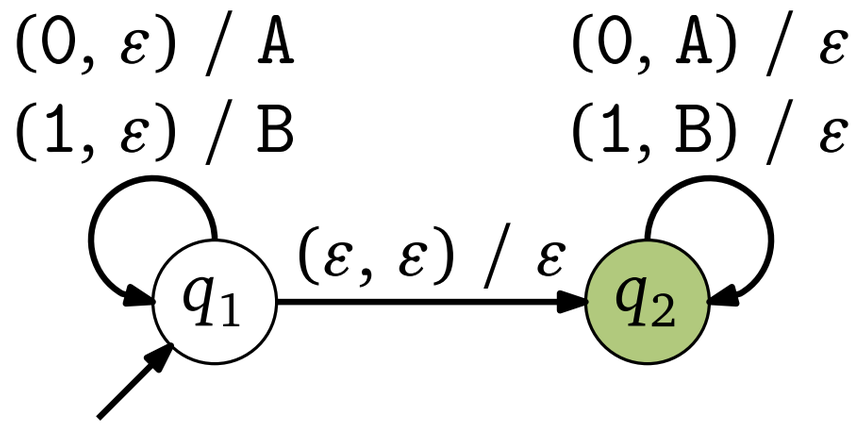
# Push-Down Automata: Example

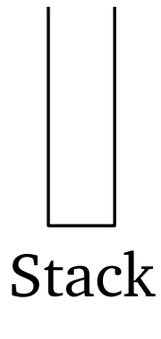
A PDA for the language $\{\sigma\sigma^R \mid \sigma \in \{0,1\}^*\}$.



# Push-Down Automata: Example

A PDA for the language $\{\sigma\sigma^R \mid \sigma \in \{0,1\}^*\}$.



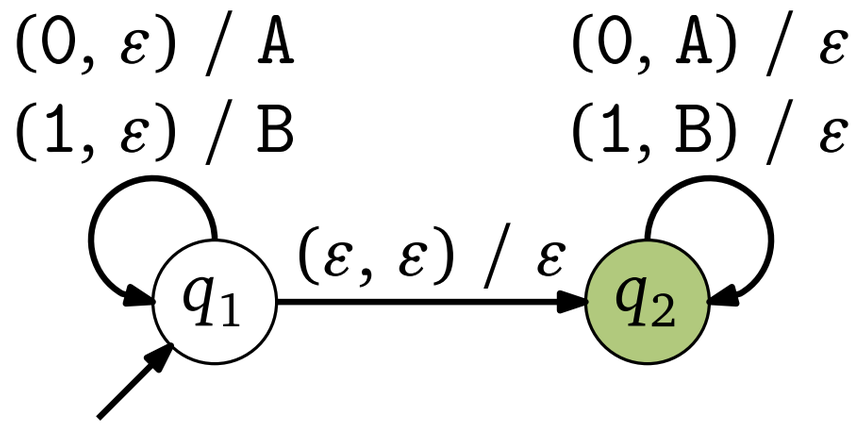



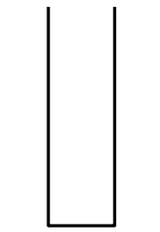

**Accept by empty stack**

PDA has consumed all input and stack is empty → accept.

# Push-Down Automata: Example

A PDA for the language $\{\sigma\sigma^R \mid \sigma \in \{0,1\}^*\}$.



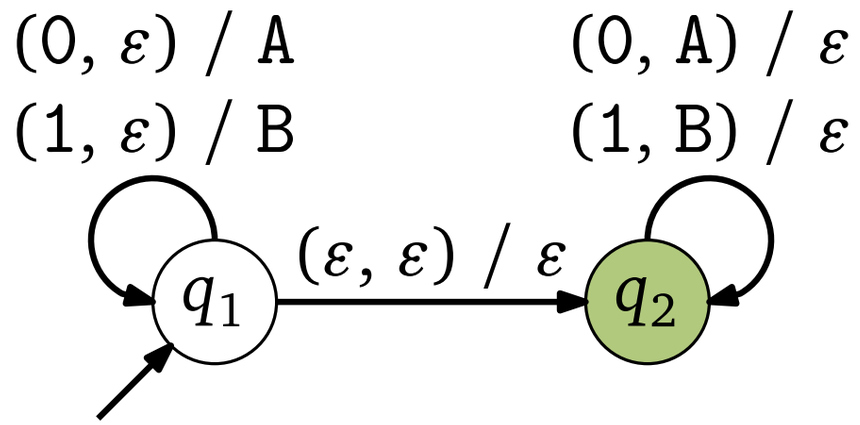

**Accept by empty stack**

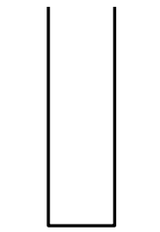
PDA has consumed all input and stack is empty → accept.

**Problem:** We need to guess where the left half ends and the right half starts.

# Push-Down Automata: Example

A PDA for the language $\{\sigma\sigma^R \mid \sigma \in \{0,1\}^*\}$.




**Accept by empty stack**

PDA has consumed all input and stack is empty → accept.

**Problem:** We need to guess where the left half ends and the right half starts.

**This language cannot be parsed deterministically!**

# Push-Down Automata: Formal Definition

A ***push-down automaton*** (PDA) is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, S, F)$

- $Q$ is a finite set of ***states***
- $\Sigma$ is the ***input alphabet***
- $\Gamma$ is the ***stack alphabet***
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\}) \rightarrow 2^{Q \times \Gamma^*}$ is the ***transition function***
- $q_0$ is the ***start state***
- $S$ is the ***start symbol*** of the stack
- $F$ is the set of ***accepting states***

# Push-Down Automata: Formal Definition

A ***push-down automaton*** (PDA) is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, S, F)$

- $Q$ is a finite set of ***states***
- $\Sigma$ is the ***input alphabet***
- $\Gamma$ is the ***stack alphabet***
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\}) \to 2^{Q \times \Gamma^*}$ is the ***transition function***
- $q_0$ is the ***start state***
- $S$ is the ***start symbol*** of the stack
- $F$ is the set of ***accepting states***

## Two modes of acceptance

- **Accept by empty stack**: Accept if and only if it is possible to reach a configuration where the input has been consumed completely and the stack is empty. The state the PDA is in at this point does not matter.
- **Accept by final state**: Accept if and only if it is possible to reach a configuration where the input has been consumed completely and the current state is an accepting state. The stack contents do not matter.

# Some Facts About PDA (1)

The two modes of acceptance are equivalent: there exists a PDA recognizing a language $\mathscr{L}$ by empty stack if and only if there exists a PDA recognizing $\mathscr{L}$ by final state.

# Some Facts About PDA (1)

The two modes of acceptance are equivalent: there exists a PDA recognizing a language $\mathscr{L}$ by empty stack if and only if there exists a PDA recognizing $\mathscr{L}$ by final state.

A language is context-free if and only if it can be recognized using a PDA.

# Some Facts About PDA (1)

The two modes of acceptance are equivalent: there exists a PDA recognizing a language $\mathscr{L}$ by empty stack if and only if there exists a PDA recognizing $\mathscr{L}$ by final state.

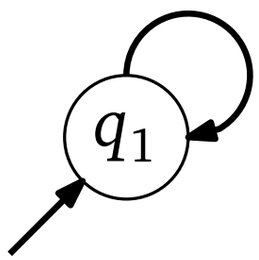
A language is context-free if and only if it can be recognized using a PDA.

**"Proof" by example**

$S \rightarrow \varepsilon$

$S \rightarrow [S]S$

$S \rightarrow (S)S$




**Terminals**

- $\delta(q_1, [, [) = \varepsilon$
- $\delta(q_1, ], ]) = \varepsilon$
- $\delta(q_1, (, () = \varepsilon$
- $\delta(q_1, ), )) = \varepsilon$

**Rules**

- $\delta(q_1, \varepsilon, S) = \{\varepsilon, [S]S, (S)S\}$

**Start symbol:** $S$

# Some Facts About PDA (2)

By default, a PDA is non-deterministic: multiple transitions are possible for a given combination of state, input symbol, and symbol on the top of the stack.

If there is only one possible transition, for any combination of state, input symbol, and stack symbol, we call the PDA a ***deterministic PDA*** (DPDA).

# Some Facts About PDA (2)

By default, a PDA is non-deterministic: multiple transitions are possible for a given combination of state, input symbol, and symbol on the top of the stack.

If there is only one possible transition, for any combination of state, input symbol, and stack symbol, we call the PDA a ***deterministic PDA*** (DPDA).

**A language can be recognized by a DPDA if and only if it is LL($k$) or LR($k$).**

In particular, there are context-free languages that cannot be recognized by a DPDA:

$$\{\sigma\sigma^R \mid \sigma \in \{0,1\}^*\}$$

# CFG == PDA

**THEOREM**

A LANGUAGE IS CONTEXT-FREE IFF SOME PUSHDOWN AUTOMATON RECOGNIZES IT.

**PROOF**

PART 1:

GIVEN A CFG, SHOW HOW TO CONSTRUC A PUSHDOWN AUTOMATON THAT RECOGNIZES IT.

PART 2:

GIVEN A PUSHDOWN AUTOMATON, SHOL HOW TO CONSTRUCT A CONTEXT-FREE GRAMMAR THAT RECOGNIZES THE SAME STRINGS.

# Proof: Part 1: CFG=>PDA

GIVEN: A GRAMMAR

$S \to BS \mid A$

$A \to 0A \mid \epsilon$

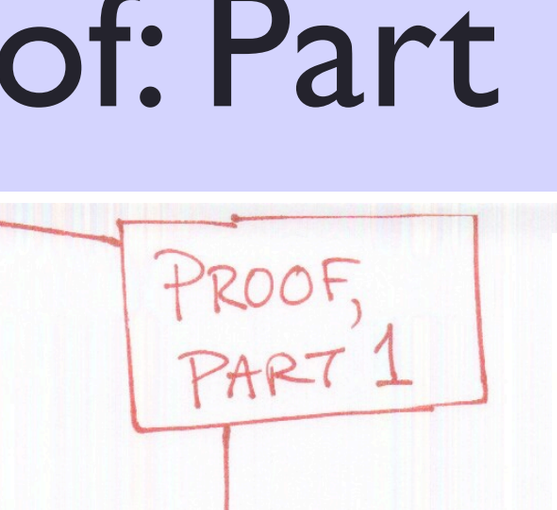
$B \to BB1 \mid 2$

FIND: (OR BUILD) A PDA.

PROOF, PART 1

CONSIDER A DERIVATION: (LEFT-MOST)

$S \Rightarrow BS \Rightarrow BB1S \Rightarrow 2B1S \Rightarrow 22 1S \Rightarrow 221A \Rightarrow 221\epsilon$




GENERAL FORM:

aaaaaa BaBBaCa

Terminals — Rest (both)


LEFT-MOST DERIVATION:

$S \overset{*}{\Rightarrow} \ldots \Rightarrow aaaaa\,BaBC \Rightarrow \ldots$

AT EACH STEP, EXPAND LEFT-MOST NON-TERMINAL.

RULE: $B \to ASAxBA$

$\ldots \Rightarrow aaaaa\,ASAxBAaBC$

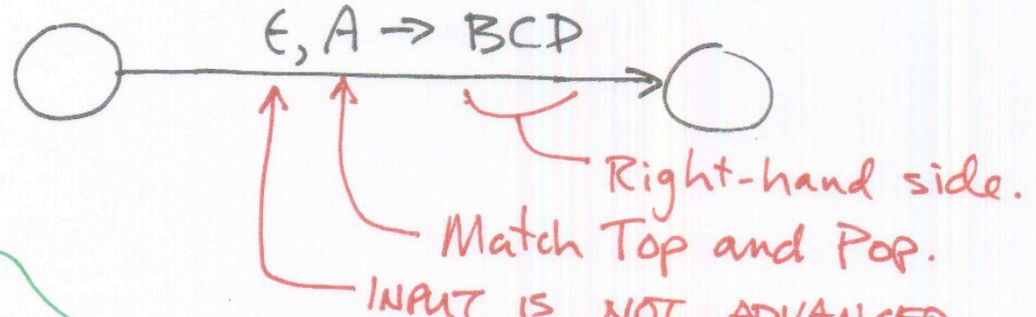
SO:

- MATCH STACK-TOP TO A RULE
- POP STACK
- PUSH RIGHT-HAND SIDE OF RULE ONTO STACK.

# Proof: Part 1: CFG=>PDA

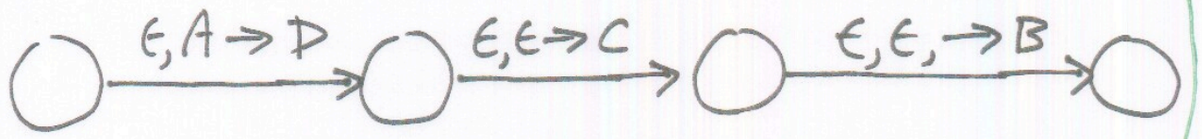
RULE: $A \to BCD$

Add this to PDA




NOTE:

TO PUSH MULTIPLE ITEMS, YOU'LL NEED TO ADD SOME EXTRA STATES.




WHICH RULE TO USE?

P.D.A.'S ARE NON-DETERMINISTIC!

(TRY THEM ALL IN PARALLEL.)

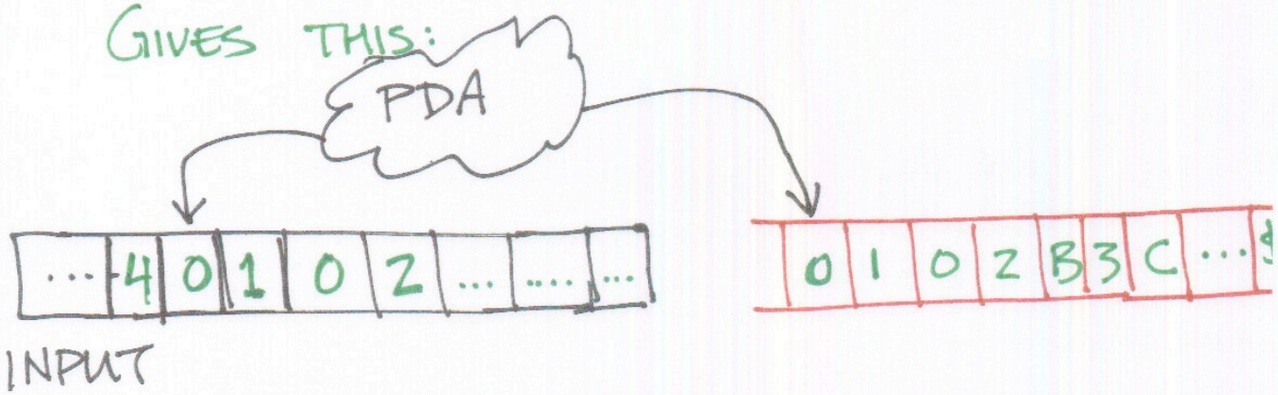
(JUST CHOOSE THE "RIGHT" RULE.)

# Proof: Part 1: CFG=>PDA

RULE:

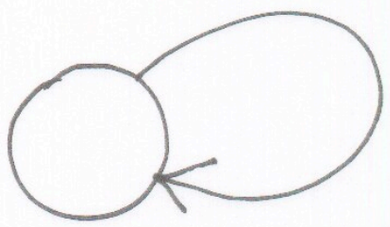
$A \to 0102B3C$

GIVES THIS:




So:

MATCH TERMINAL SYMBOLS TO THE STACK TOP.



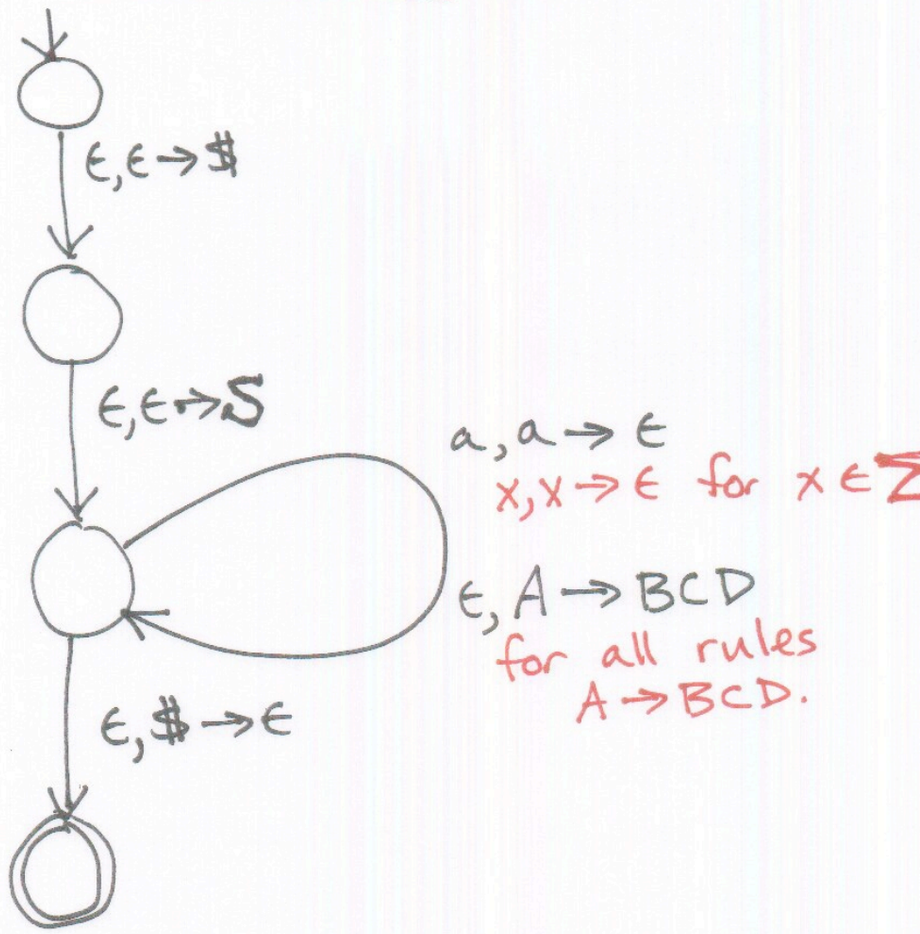
$0,0 \to \epsilon$

$1,1 \to \epsilon$

$2,2 \to \epsilon$

etc. for all $x \in \Sigma$.

THE FINAL MACHINE:



# Proof: Part 2: CFG<=PDA

PROOF, PART 2

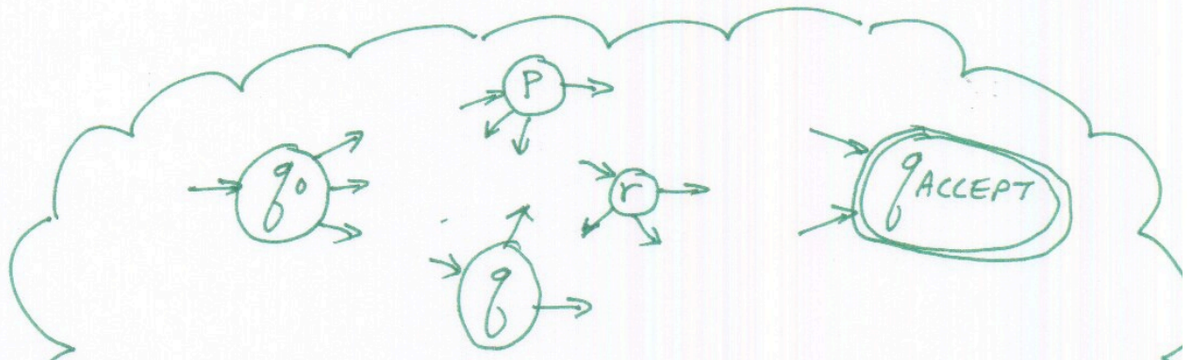
WE ARE GIVEN: A P.D.A.
MUST BUILD: A CFG FROM IT.

STEP 1

SIMPLIFY THE PDA.

STEP 2

BUILD THE CFG.




There will be a non-terminal for ever PAIR of states.

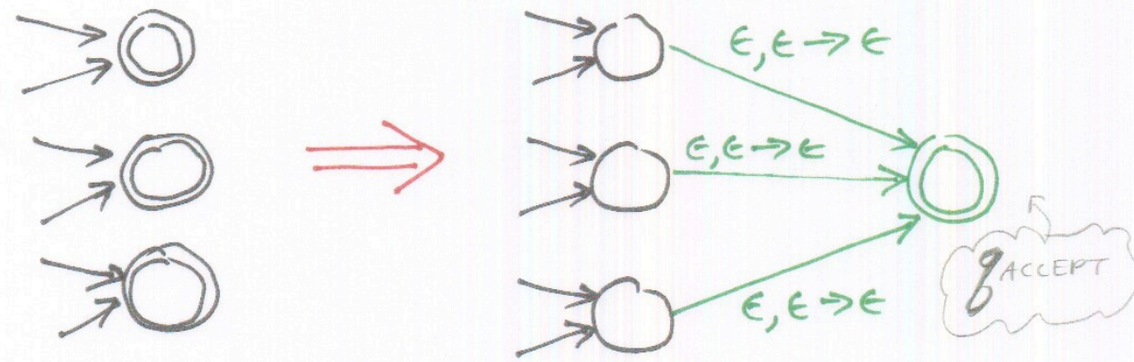
$A_{pq}$ $A_{qr}$ $A_{rq_0}$ ...

The starting nonterminal will be

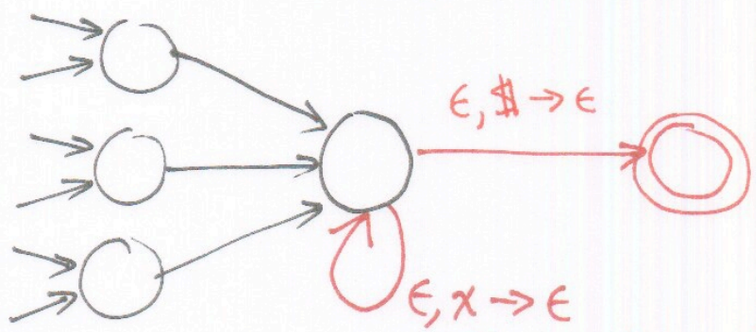
$A_{q_0 q_{ACCEPT}}$

SIMPLIFY THE PDA

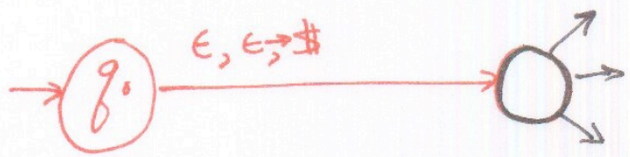
(1) The PDA has only one ACCEPT state




(2) The PDA empties its stack before ACCEPTING.

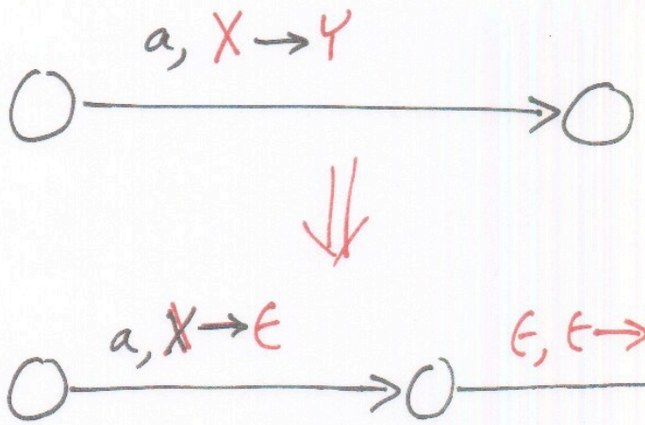


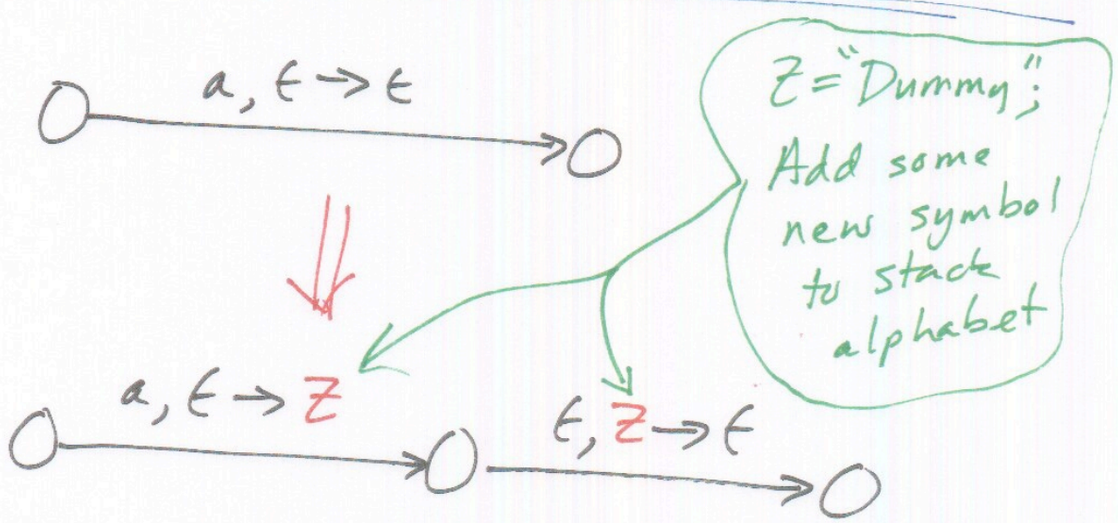

for all $x \in$ ✱ $\Gamma - \{\$\}$



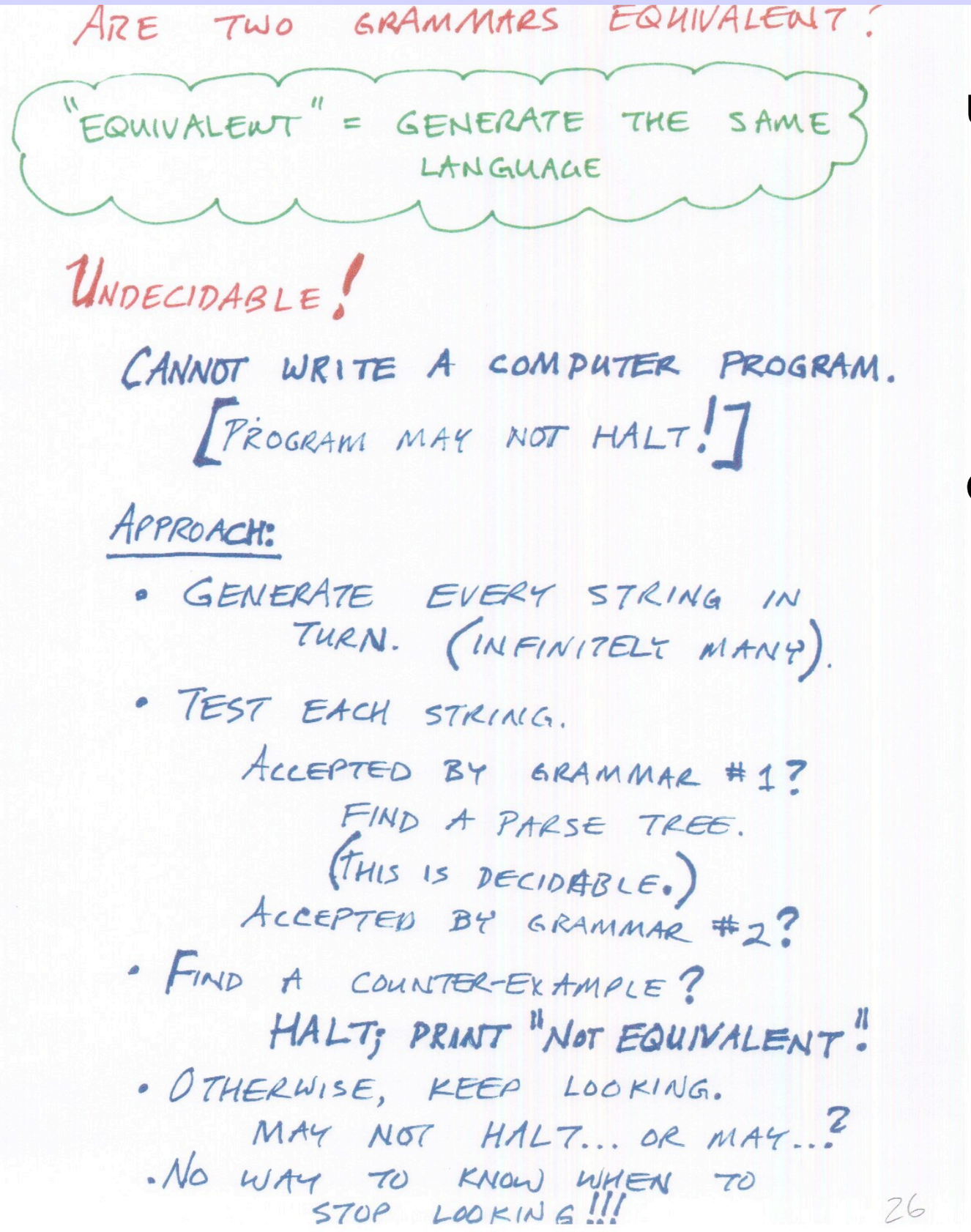
# Proof: Part 2: CFG<=PDA

(3) Each transition either PUSHES or POPS, but does not do both.



# Undecidable Problems in CFG




undecidable problems:

*ambiguity of CFG*
*ambiguity of CFL*
*CFG equivalence*

decidable problems:

*ambiguity of string*
*string in grammar* (is s in *L*(*G*)?)

both decidable problems need CNF+CKY

# Context-Sensitive Grammar

## NON-CONTEXT-FREE GRAMMARS

$L = \{ 0^N 1^N 0^N \mid N \geq 0 \}$

00000 11111 00000
5 5 5

IS THIS LANGUAGE CONTEXT-FREE?

NO!

(Use the PUMPING LEMMA FOR CFG's TO PROVE IT.)

## CHOMSKY HIERARCHY

**TYPE-3 LANGUAGES**

REGULAR

**TYPE-2 LANGUAGES**

CONTEXT-FREE $A \to \gamma$

**TYPE-1 LANGUAGES**

CONTEXT-SENSITIVE

$\alpha A \beta \to \alpha \gamma B$

**TYPE-0 LANGUAGES**

RECURSIVELY-ENUMERABLE
TURING-ENUMERABLE.
TURING-RECOGNIZABLE

ex: $AB x \to ADEy$



$L = \{ 1^N 2^N 3^N \mid N \geq 1 \}$

APPROACH:

EACH "A" will turn into a "1".
EACH "B" will turn into a "2".
EACH "C" will turn into a "3".

$S \overset{*}{\Rightarrow} \ldots \quad AAABBBCCC \overset{*}{\Rightarrow} \ldots 111\,222\,333$

INITIALLY THE STRING WILL START WITH "1"s

111[redacted] BBB CCC

Will never use "A" actually.

RULES TO FINISH IT UP:

$1B \to 12$
$2B \to 22$
$2C \to 23$
$3C \to 33$

NOTE
... CB ...
CAN NEVER BE REDUCED.
"C" CAN ONLY TURN[redacted] INTO A "3".
AND "3B" CAN NEVER BE REDUCED.
⇒ B's MUST PRECEED "C"s.

1 1 1 B B B C C C
1 1 1 2 B B C C C
1 1 1 2 2 B C C C
1 1 1 2 2 2 C C C
1 1 1 2 2 2 3 C C
1 1 1 2 2 2 3 3 C
1 1 1 2 2 2 3 3 3

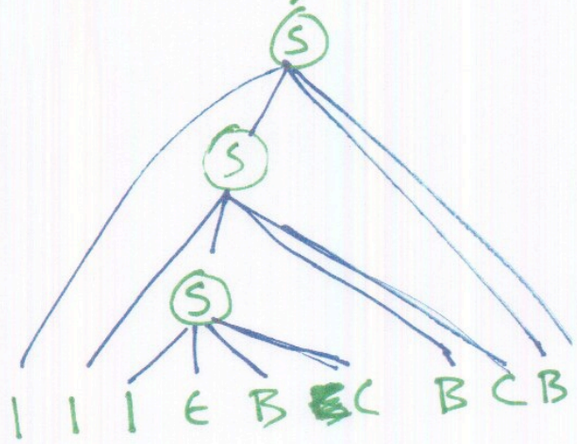
# Context-Sensitive: $1^n2^n3^n$

STEP 1: GENERATE THE CORRECT NUMBER OF 1's, B's, and C's (JUST NOT THE RIGHT ORDER.).

$S \rightarrow 1SBC$

~~[illegible]~~

$S \rightarrow \epsilon$




1 1 1 1 BC BC BC BC

STEP 2: GET THE "B"s IN FRONT OF THE "C"s.

$CB \rightarrow BC$

STEP 3:

REDUCE "B" TO "2" AND "C" TO "3."

(Rules shown above.)

CONTEXT-SENSITIVE GRAMMAR

$L = \{1^N 2^N 3^N / N \geq 1\}$

FROM BEFORE

$S \rightarrow 1SBC$

$S \rightarrow \epsilon$

---

REVISED

$CB \rightarrow HB$

$HB \rightarrow HC$

$HC \rightarrow BC$

---

FROM BEFORE

$1B \rightarrow 12$

$2B \rightarrow 22$

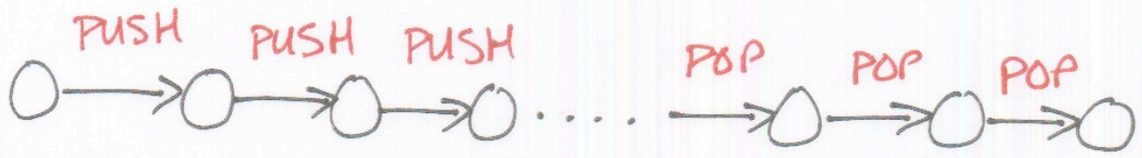
$2C \rightarrow 23$

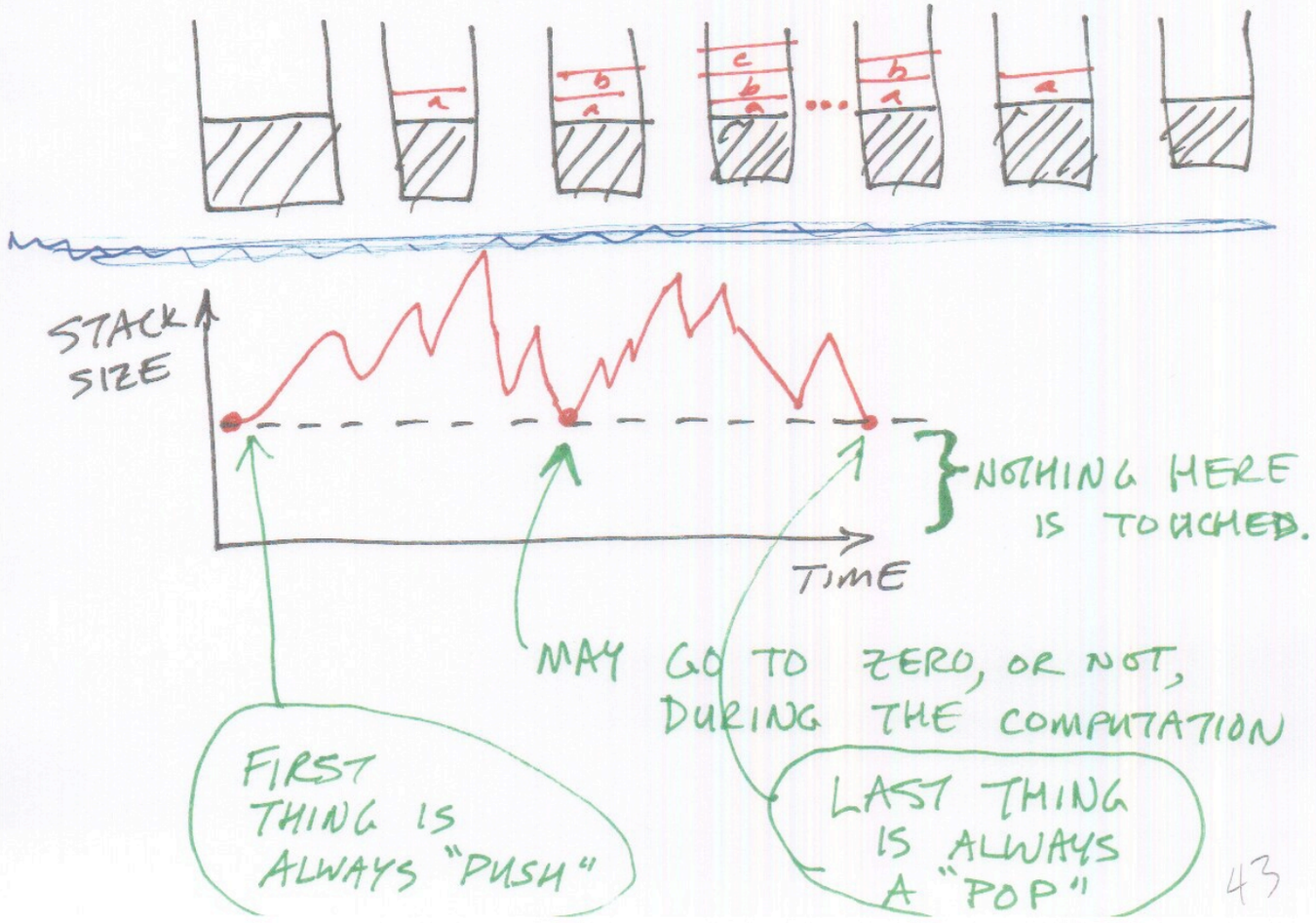
$3C \rightarrow 33$

"DON'T MODIFY THE STACK"

= "START W/ AN EMPTY STACK AND
FINISH WITH AN EMPTY STACK"

= "DON'T TOUCH THE STACK"!



## MAIN IDEA.

Consider two states $p$ and $q$ in the PDA.

Could we go from $p$ to $q$ without touching the stack?

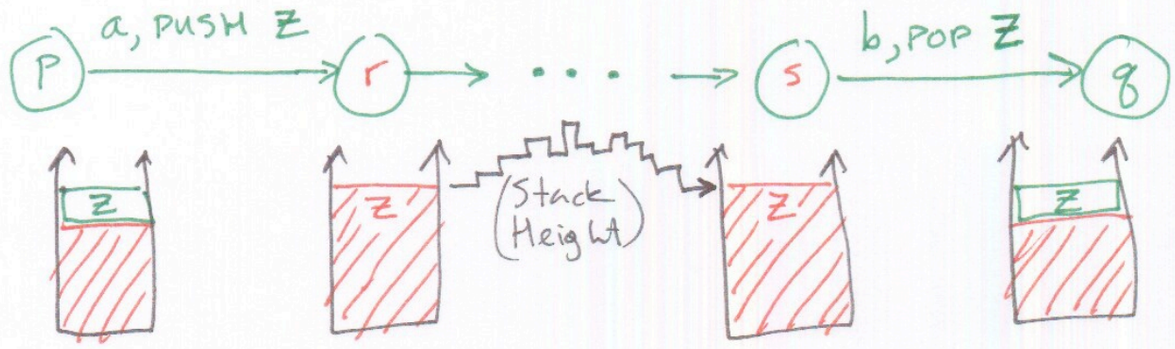
What strings would do that?

That is:
Starting w/ an empty stack, we could go from $p$ to $q$ and end up with an empty stack.

Or if somethings were on the stack they would never be touched.

The grammar we build will have a non-terminal
⇒ we'll call it $A_{pq}$
that will generate exactly these strings!

44



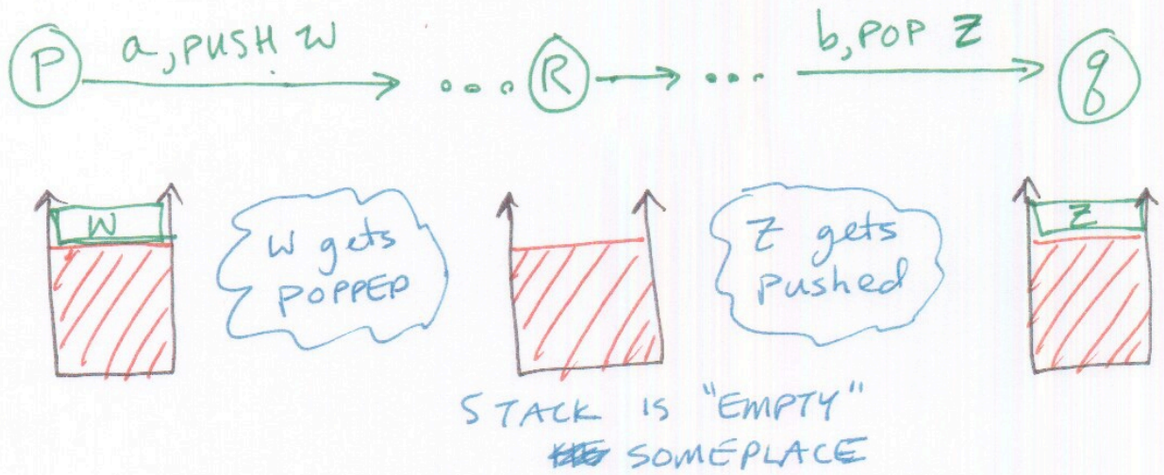

What strings can be generated/accepted by following this path?

"a ... b"

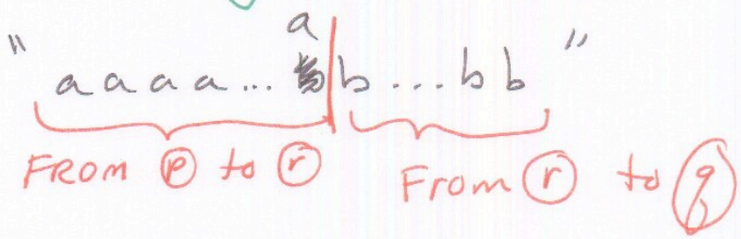
$A_{pq} \rightarrow a\ A_{rs}\ b$

This rule will generate exactly those strings!!

45




What strings can be generated by following this path?

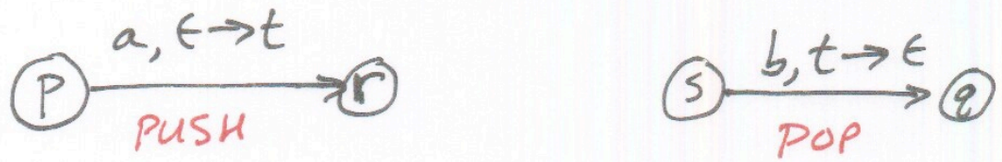



$A_{pq} \rightarrow A_{pr} A_{rq}$

This rule will generate exactly those strings!

46

If we have these edges




And we could get from (r) to (s) without touching the stack,

Then we need a ~~mess~~ grammar rule:

$$A_{pq} \rightarrow a A_{rs} b$$

---

FOR EACH $p, q, r, s \in Q$ in the PDA such that $\delta(p, a, \epsilon)$ contains $(r, t)$ and $\delta(s, b, t)$ contains $(q, \epsilon)$

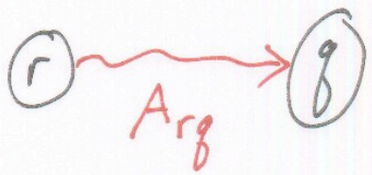
[i.e., "and the edges are labelled as above, for any $a, b \in \Sigma$ and $t \in \Gamma$..."]

THEN ADD THIS RULE TO THE CFG:

$$A_{pq} \rightarrow a A_{rs} b$$

If we have a way to get from state (p) to state (r) that doesn't touch the stack


And a way to get from (r) to (q) that doesn't touch the stack.




THEN We have a new way to get from (p) to (q) without touching the stack.

---

For every state $p, r, q \in Q$
Add this rule to the grammar:

$$A_{pq} \rightarrow A_{pr} A_{rq}$$

There is a trivial way to get from state (p) to itself without touching the stack: The string $\epsilon$.

So add

$$A_{pp} \rightarrow \epsilon \quad \text{for every state } (p).$$

---

If the PDA accepts some string then there is a way to go from $(q_0)$ to $(q_{ACCEPT})$ that does not & modify the stack.

The grammar we seek should ~~accept~~ generate exactly these strings.

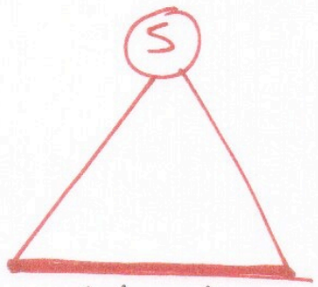
~~The~~ OUR START NON-TERMINAL IS:

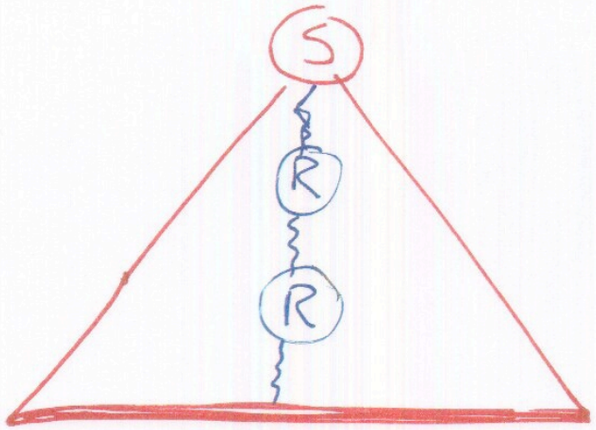
$$A_{q_0 q_{ACCEPT}}$$

49

# Pumping Lemma for CFGs

PUMPING LEMMA FOR CFG's




really long string

SOME NON-TERMINAL "R" MUST BE USED MORE THAN ONCE.

00 2 44
u x z

00 1 2 3 44
u v x y z

00 1111 2 3333 44
u $v^4$ x $y^4$ z

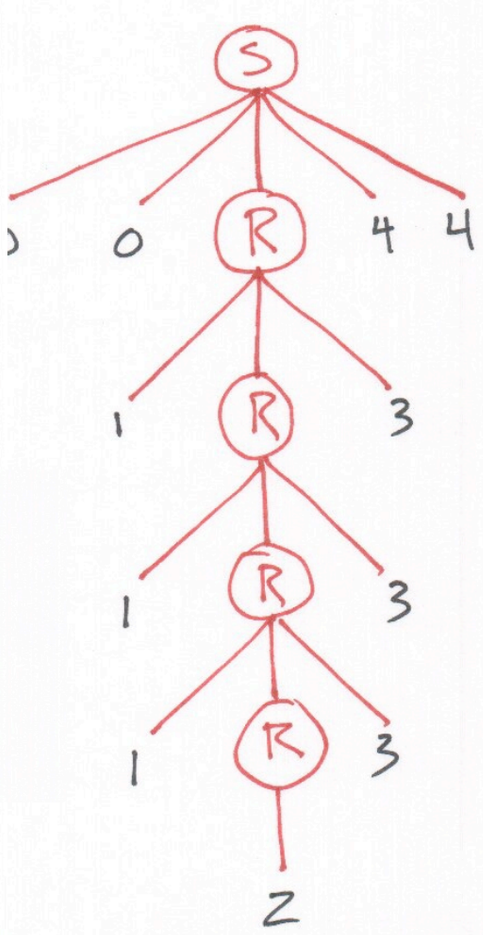
$uv^ixy^iz$ is also in the Language!

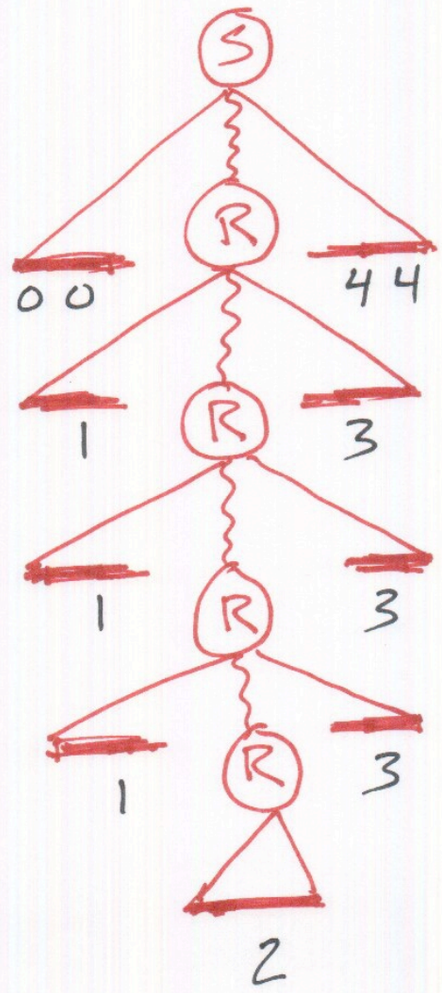
CONSIDER: $L = \{ 00 1^N 2 3^N 44 \mid N \geq 0 \}$

$S \rightarrow 00R44$

$R \rightarrow 1R3 \mid 2$

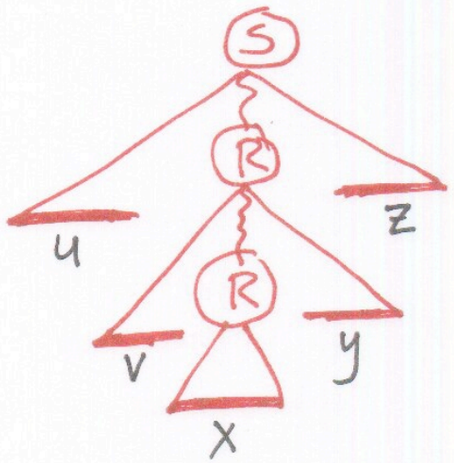
HOW CAN WE GENERATE "REALLY LONG" STRINGS?



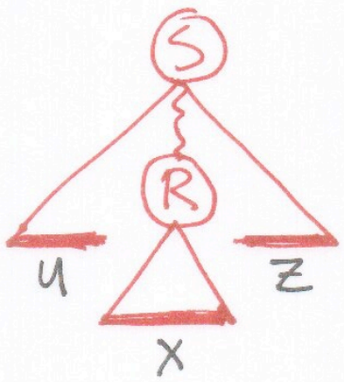
# Harder Case

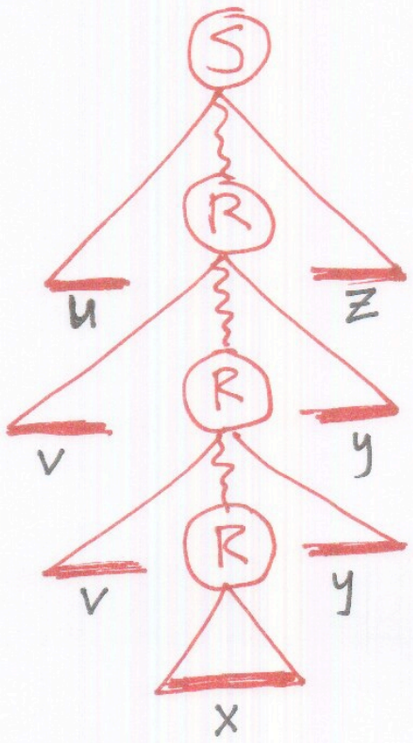
FOR ALL STRINGS THAT ARE "LONG ENOUGH", SOME NON-TERMINAL HAS TO BE REPEATED IN THE PARSE TREE.




$R \overset{*}{\Rightarrow} vRy$

THEREFORE, THESE ARE ALSO LEGAL PARSE TREES:



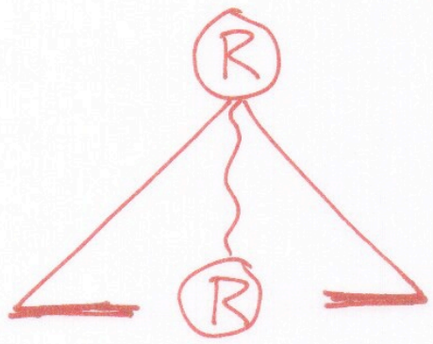

$uv^ixy^iz$ is also in the language!

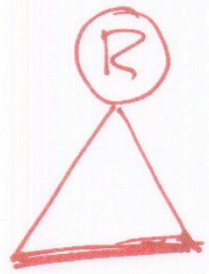
IN OTHER WORDS, TO GET LONG STRINGS WE MUST USE RECURSION IN THE GRAMMAR.

$R \overset{*}{\Rightarrow} \ldots R \ldots$

AND FINALLY TO FINISH:

$R \overset{*}{\Rightarrow} x$



# Pumping Lemma for CFGs

**PUMPING LEMMA FOR CFG's**

If a string is sufficiently long, $|s| \geq p$
then it can be pumped.
That is, the string can be
broken into parts (someway)

$s = uvxyz$

such that all strings of the form

$uv^ixy^iz$

are also in the language.

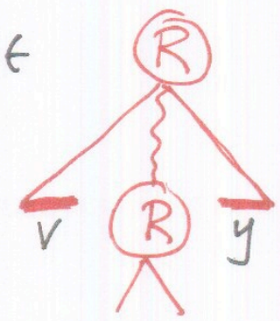
NOTES:

- v and y cannot both be $\epsilon$

  $|vy| > 0$

- The beginning of v and the end of y can't be too far apart

  $|vxy| \leq p$



# Pumping Lemma for CFGs

PUMPING LEMMA FOR CFG's
(REWORDING IT.)

IF $A$ IS A CONTEXT-FREE LANGUAGE, THEN THERE IS A PUMPING LENGTH $p$ SUCH THAT, FOR ANY STRING IN $A$ WHOSE LENGTH IS LONG ENOUGH, $|s| \geq p$, THAT STRING CAN BE BROKEN INTO PIECES $s = uvxyz$

IN A WAY THAT SATISFIES ALL THREE OF THESE CONDITIONS:

CONDITION 1:

$uv^ixy^iz$ is in $A$, for all $i \geq 0$

CONDITION 2:

$|vy| > 0$

CONDITION 3:

$|vxy| \leq p$

# Pumping Lemma Logic

## Logic Refresher

How can we negate

"For all"

$$\sim \forall x.\ P(\ldots) \equiv \exists x.\sim P(\ldots)$$

"There Exists"

$$\sim \exists x.P(\ldots) \equiv \forall x.\sim P(\ldots)$$

"And"

$$\sim(\ldots P\ldots \wedge \ldots Q\ldots) \equiv (\sim(\ldots P\ldots) \vee \sim(\ldots Q\ldots))$$

---

"It's not the case that all numbers are even."

⇕

"There exists a number that is not even."

---

"It's not the case that there exists a green number."

⇕

"All numbers have the "NOT-GREEN" property."



If $L$ is a context-free language....

**PUMPING PROPERTY**

$\exists p$

$\forall s$ in $L$ where $|s| \geq p$

$\exists uvwxyz = s$

Such that

① $uv^ixy^iz \in L$, $\forall i \geq 0$ AND

② $|vy| > 0$ AND

③ $|vxy| \leq p$

To show $L$ is not context-free, we must show that ~(PUMPING PROPERTY) holds.

**NOT-PUMPING PROPERTY**

$\forall p$

$\exists s$ in $L$ where $|s| \geq p$

$\forall uvxyz = s$

Such That

① $uv^ixy^iz \notin L$, $\forall i \geq 0$ OR

② $|vy| \neq 0$ OR

③ $|vxy| \not\leq p$

# Example 1: $a^n b^n c^n$

SHOW $B = \{a^N b^N c^N \mid N \geq 0\}$ is NOT CONTEXT-FREE.

Assume it is CFL. Show ~(PUMPING PROPERTY)

Let p be the pumping length.
(No constraints on P. We'll show it $\forall p$.)

There exists a string.... $|s| \geq p$
We'll use: $a^p b^p c^p$ [$\exists s \ldots$]

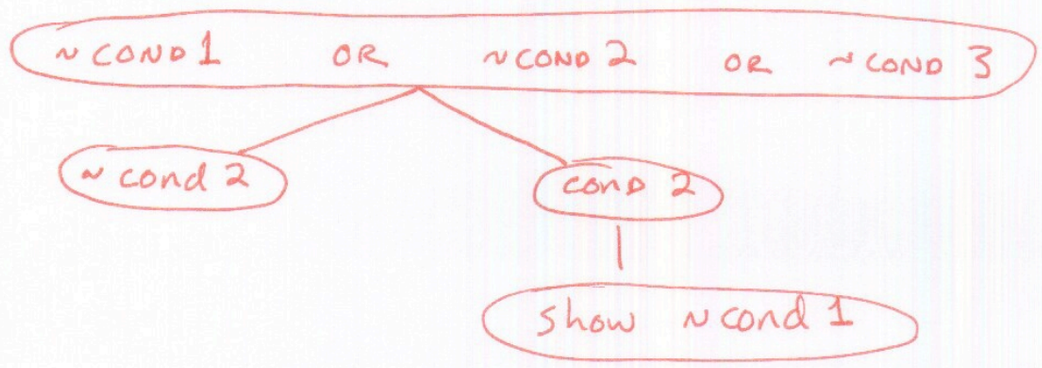
Now look at all ways to divide it up. [$\forall uvxyz = s \ldots$]

~~Show~~ We'll show some condition will always be violated.

Condition (2) says $|vy| > 0$

Assume this condition ~~is~~ holds.

Now look at two cases.



# Example 1: $a^n b^n c^n$

**CASE 1** v and y each contain only one type of symbol.

a a a a b b b b c c c c

v y

a a a a b b b b c c c c

v $y = \epsilon$

One symbol will always be left out.

Pump s to $uv^2xy^2z$

a a a a a b b b b c c c c c

v v y y

At least one symbol will increase in ~~number~~ number.

At least one symbol will not increase in number.

The string cannot still be in the form

$a^N b^N c^N$

**CASE 2** Either v or y has more than one kind of symbol:

a a a b b b c c c

v y

a a a b b b c c c

v y

Pump to $uv^2xy^2z$. We might have right number of symbols, but the order will be wrong.

a a a a b b b c b c c c

v v y y

# Example 2: copy language

Show $D = \{ww \mid w \in \{0,1\}^*\}$ is not context-free.

Assume it is a CFL.

Show ~(PUMPING PROPERTY)

Let $p$ be the pumping length.

[No constraints on $p$. Show $\forall p$]

There exists a string $s$, $|s| \geq p$...

[To show $\exists s$, just provide an example; just give an $s$ that exists]

$0^p 1^p 0^p 1^p$

Look at all ways to divide $s$ into parts.

[Show $\forall uvxyz = s$]

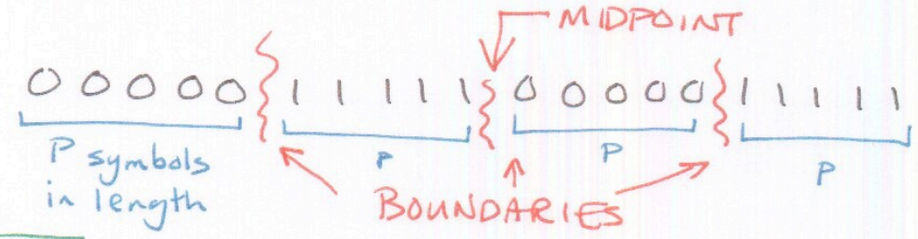
Must show that some condition is not satisfied.

We'll assume condition 3 holds

$|vxy| \leq p$

and show that the other conditions must fail.

CONSIDER THE BOUNDARIES BETWEEN 0's AND 1's IN OUR STRING.




CASE 1: $vxy$ does not straddle a boundary.

00000 11111 00000 11111

$vxy$

PUMPING UP WILL YIELD A STRING WITH MORE 1's AND AN "IMBALANCE"

NEW MIDPOINT.

00000 11.... 11 00000 11111

p MORE THAN p p

THE STRING NOW HAS THE FORM

NEW MIDPOINT.

0------1-------

This string is not of the form $ww$.

Since $uv^2xy^2z$ is not in the language, CONDITION 1 is violated.

# Example 2: copy language

**CASE 2:** vxy straddles the first boundary.

0 0 0 0 0 { 1 1 1 1 { 0 0 0 0 0 } 1 1 1 1 1

vxy — MIDPOINT.

SINCE $|vxy| \leq p$ it cannot straddle the midpoint.

PUMPING DOWN WILL MAKE THE STRING SHORTER.

NEW MIDPOINT

0 0 0 _____ 1 1 0 0 0 0 0 1 1 1 1 1

FEWER SYMBOLS — p — p

NOTE: The string now has the form:

NEW MIDPOINT.

- - - - - - - - 0 - - - - - - - - - 1

The string is not of the form WW.

**CASE 2b:** vxy straddles the third boundary: It is similar.

0 0 0 0 0 { 1 1 1 1 1 { 0 0 0 0 0 } 1 1 1 1 1

vxy

**CASE 3:** vxy straddles the midpoint.

Since $|vxy| \leq p$, it cannot also straddle the first or third boundary.

MID POINT

0 0 0 0 0 1 1 1 1 0 0 0 0 0 1 1 1 1 1

p — vxy — p

2p

Pumping down will give us a shorter string.

NEW MIDPOINT

0 0 0 0 0 - - - - - - - - - 1 1 1 1 1

p — <p — <p — p

Look at the first half of the string and the second half.

0 0 0 0 0 - - -

- - - 1 1 1 1 1

p — p

They cannot be equal.
This string fails condition 1.