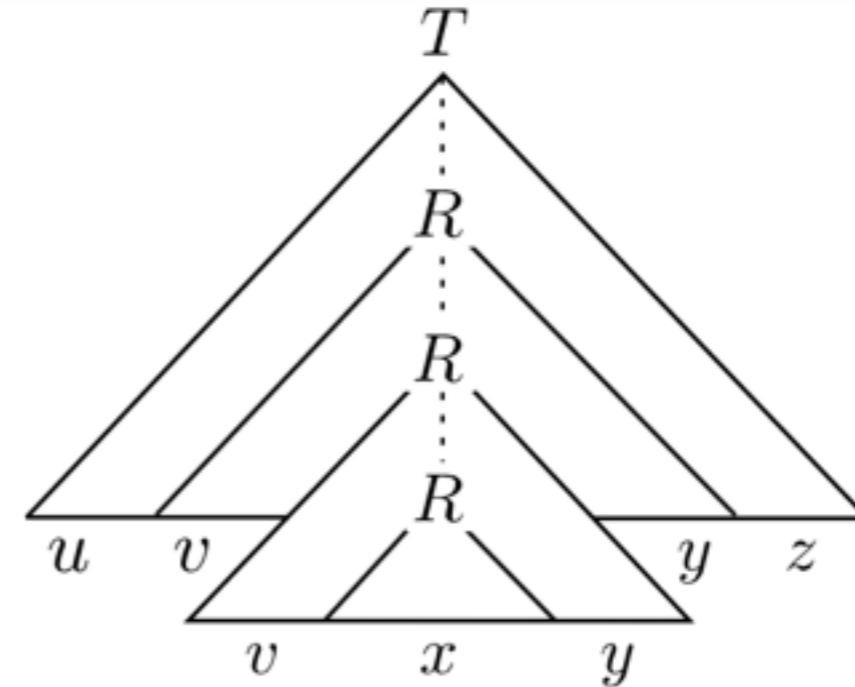
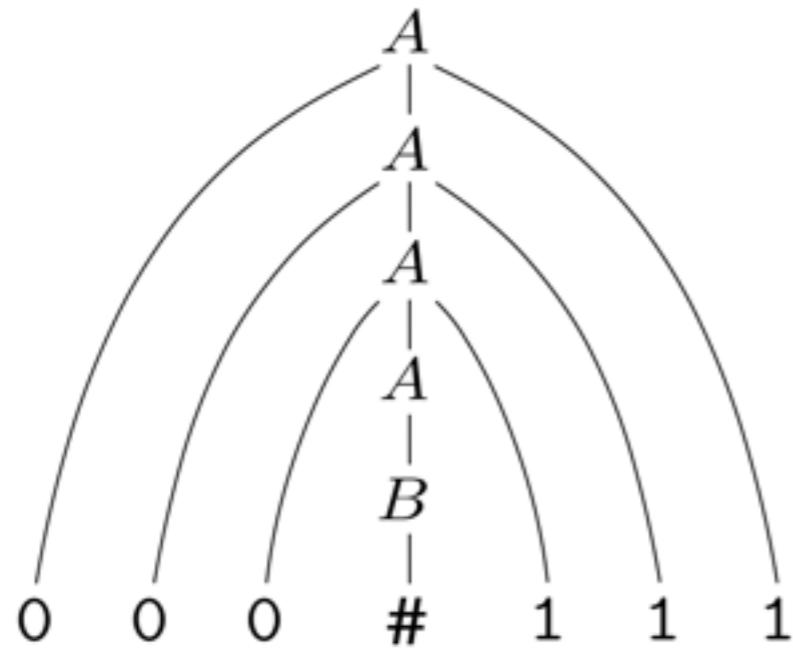


CS 321

Theory of Computation

Part II: Context-Free Languages

Sipser 2; Linz 5,6,7,8



Instructor: Liang Huang
(some slides courtesy of H. Potter)

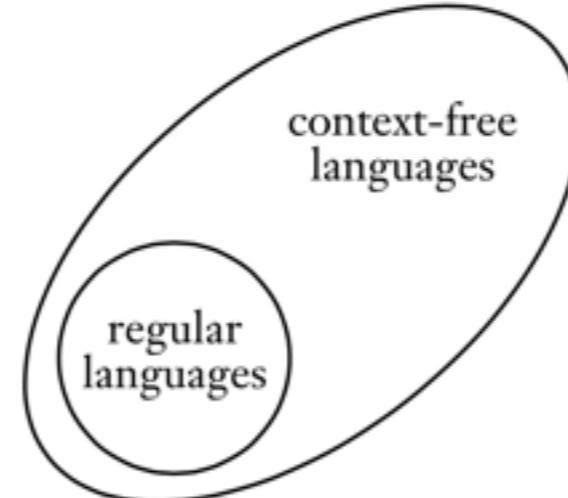
Roadmap

- 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

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



EXAMPLE ~~CFG~~

$$S \rightarrow (S) \mid SS \mid \epsilon$$

Language = $\{\epsilon, (), ()(), ((())(), (((()))((())()), ((())), ((()())((())(), \dots\}$

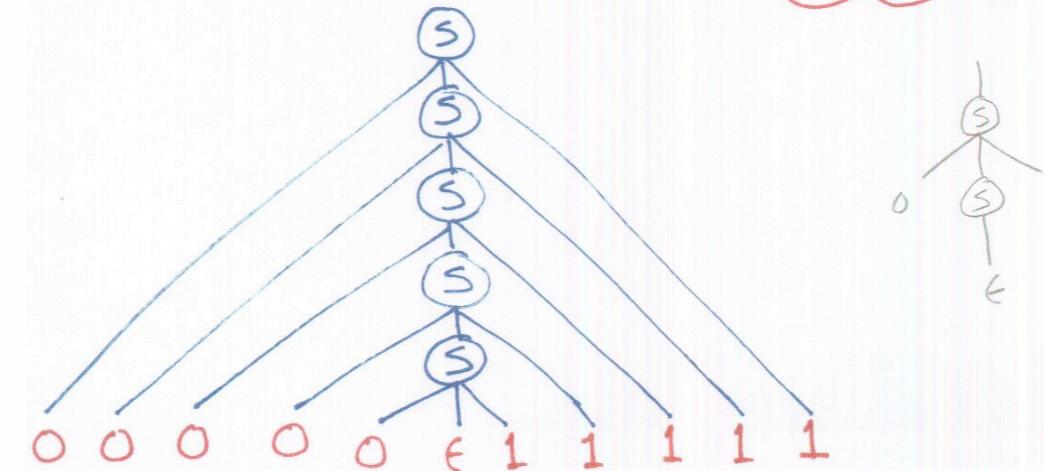
EXAMPLE

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

$$S \rightarrow \epsilon$$

$$S \rightarrow 0 S 1$$

We used the PUMPING LEMMA to show this is NOT REGULAR.
So:
REG \subset CFL [PROPER SUBSET].



Derivations and Parse Trees

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

PARSE TREES

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

DERIVATION

$$\begin{aligned}
 E &\Rightarrow E + T \Rightarrow I + T \Rightarrow E + T \Rightarrow a + I \\
 &\Rightarrow a + F \Rightarrow a + (E) \Rightarrow a + (I) \\
 &\Rightarrow a + (I \times F) \Rightarrow a + (F \times F) \\
 &\Rightarrow a + (a \times F) = a + (a \times a)
 \end{aligned}$$

WE WRITE:

$$\begin{aligned}
 E &\xrightarrow{*} a + (a \times a) \\
 E &\xrightarrow{*} a + (E) \Rightarrow a + (T) \xrightarrow{*} a + (a \times a)
 \end{aligned}$$

LEFT-MOST DERIVATION:

Always choose Left-most Variable

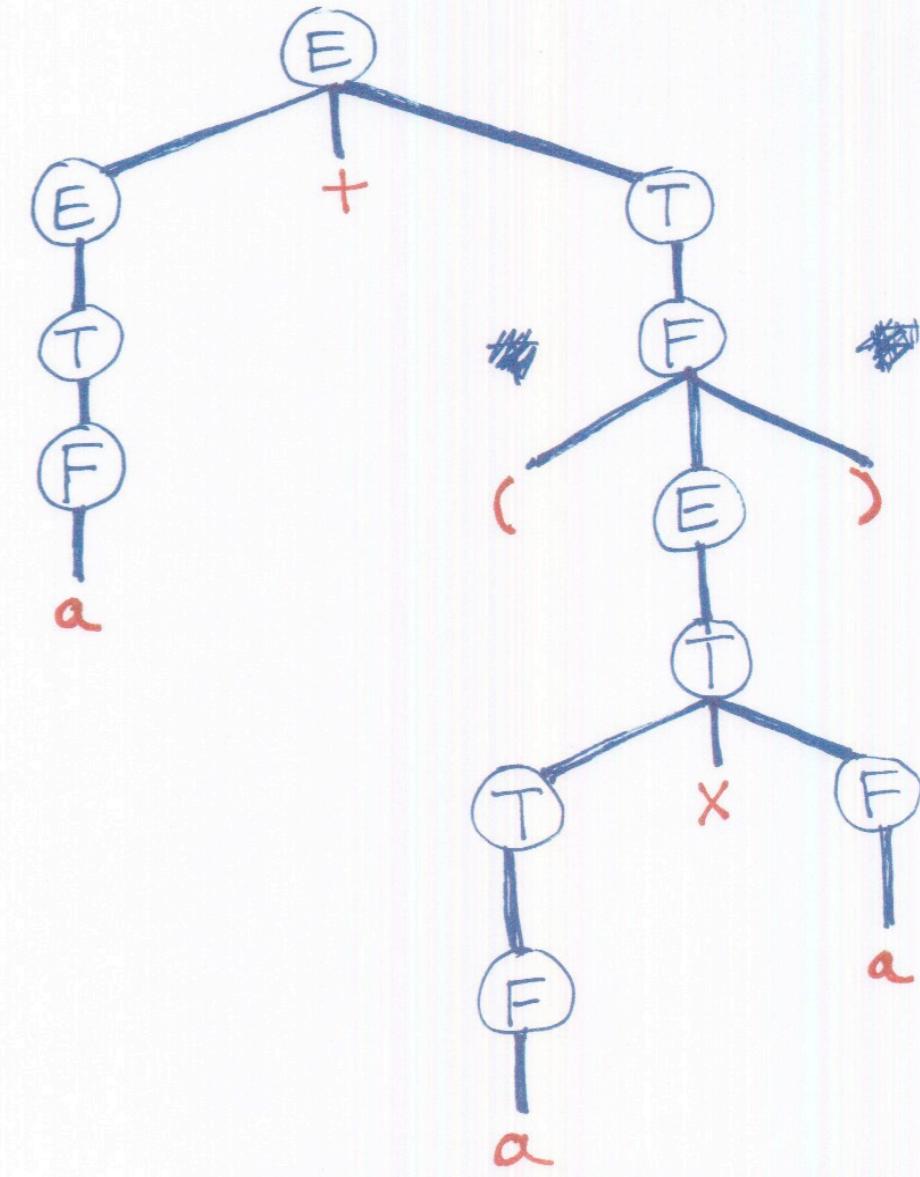
$$\dots \Rightarrow F + T \Rightarrow a + T \Rightarrow \dots a + (a \times a)$$

RIGHT-MOST DERIVATION:

Always choose Right-most Variable

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

$$E \xrightarrow{*} a + (a \times a)$$



CFGs for Natural Language

- $S \rightarrow NP\ VP$ I saw a boy with a telescope
 - $N \rightarrow \text{boy } | \text{ girl } | \dots$ I eat sushi with tuna
 - $P \rightarrow \text{with } | \text{ in } | \text{ on } | \dots$
 - $V \rightarrow \text{eat } | \text{saw } | \dots$
 - $NP \rightarrow DN$
 - $NP \rightarrow NP\ PP$
 - $PP \rightarrow P\ NP$
 - $VP \rightarrow V\ NP$
 - $VP \rightarrow VP\ PP$

CFGs for Natural Language

- $S \rightarrow NP\ VP$ I saw a boy with a telescope
- $N \rightarrow boy\ | \ girl\ | \ ...$ I eat sushi with tuna
- $P \rightarrow with\ | \ in\ | \ on\ | \ ...$
- $V \rightarrow eat\ | \ saw\ | \ ...$
- $NP \rightarrow D\ N$
- $NP \rightarrow NP\ PP$
- $PP \rightarrow P\ NP$
- $VP \rightarrow V\ NP$
- $VP \rightarrow VP\ PP$



Context-Free Grammars: Definition

A CFG is a 4-tuple $\langle V, \Sigma, R, S \rangle$

A set of nonterminals or variables V (often “N” for nonterminals)

(e.g. $V = \{S, NP, VP, PP, Noun, Verb, \dots\}$)

A set of terminals Σ (often “T” for terminals)

(e.g. $\Sigma = \{I, you, he, eat, drink, sushi, ball, \dots\}$)

A set of rules R (often “P” for productions)

$R \subseteq \{A \rightarrow \beta \text{ with left-hand-side (LHS) } A \in V$
 $\text{and right-hand-side (RHS) } \beta \in (V \cup \Sigma)^*\}$ -- RHS could be ϵ

A start symbol S (sentence)

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

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

Yields, Derives, Sentential Form

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

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

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

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

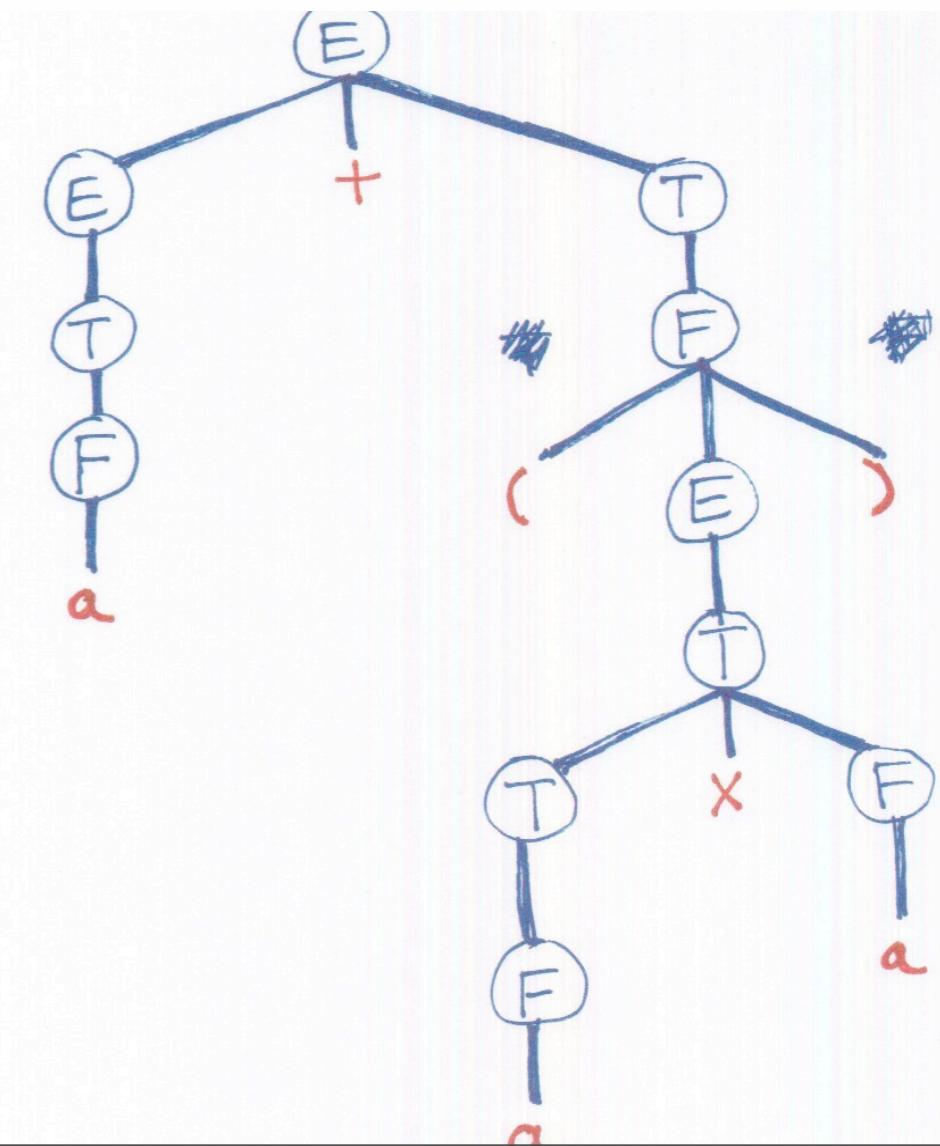
$$\begin{array}{l} E \rightarrow E + T \mid T \\ T \rightarrow T \times F \mid F \\ F \rightarrow (E) \mid a \end{array}$$

DERIVATION

$$\begin{aligned} E &\xrightarrow{*} E + T \rightarrow I + T \rightarrow E + T \rightarrow a + T \\ &\Rightarrow a + E \Rightarrow a + (E) \Rightarrow a + (I) \\ &\Rightarrow a + (I \times F) \Rightarrow a + (E \times F) \\ &\Rightarrow a + (a \times F) = a + (a \times a) \end{aligned}$$

WE WRITE:

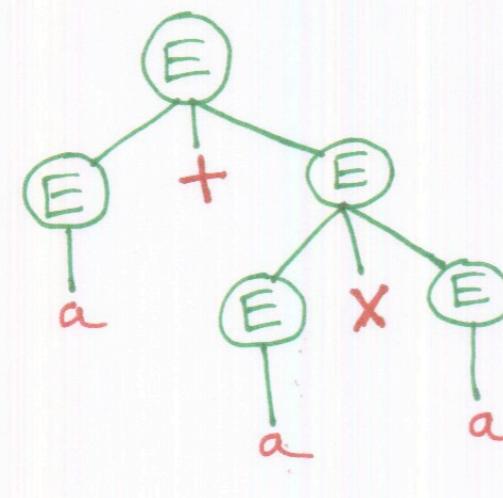
$$\begin{aligned} E &\xrightarrow{*} a + (a \times a) \\ E &\xrightarrow{*} a + (E) \Rightarrow a + (T) \xrightarrow{*} a + (a \times a) \end{aligned}$$



Ambiguous vs. Unambiguous Grammar

```
E → E + E  
→ E × E  
→ ( E )  
→ a
```

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

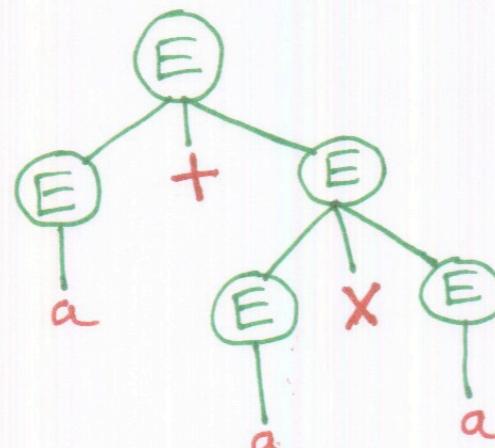
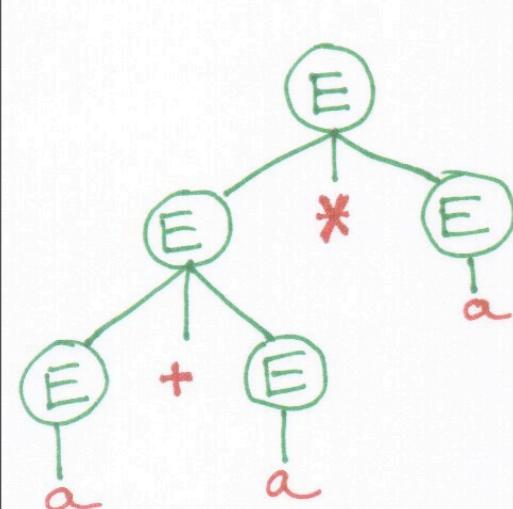
A grammar is "AMBIGUOUS" if some string can be derived in more than one way.
I.e., Some string has MULTIPLE PARSE TREES.

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

Ambiguous vs. Unambiguous Grammar

```
E → E + E  
→ E × E  
→ ( E )  
→ a
```

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 MULTIPLE PARSE TREES.

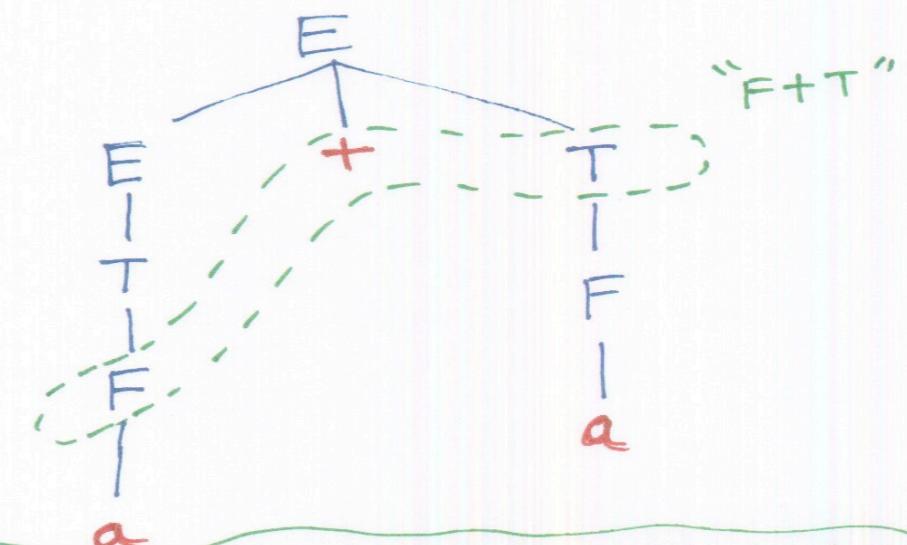
- ambiguous grammar: some string can have two parse trees
- sometimes there is an equivalent unambiguous grammar
- but not always! => inherently ambiguous languages

You GET THE SAME RESULT:

```
E → E + T | T  
T → T × F | F  
F → ( E ) | a
```

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

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

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

EQUIVALENT UNAMBIGUOUS GRAMMAR

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

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

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

EQUIVALENT UNAMBIGUOUS GRAMMAR

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

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
 \equiv 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

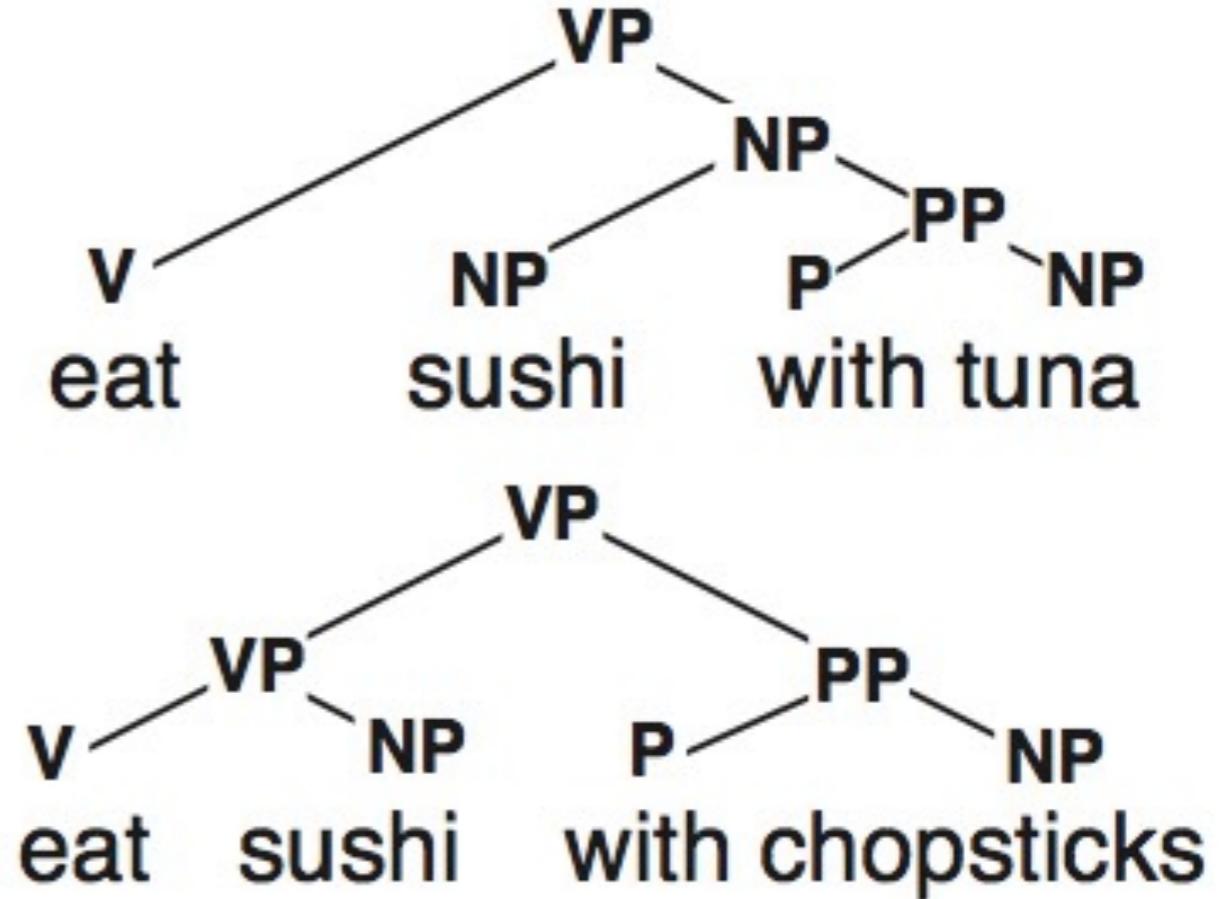
ALL GRAMMARS FOR THE LANGUAGE
ARE AMBIGUOUS.

THERE IS NO UNAMBIGUOUS GRAMMAR
THAT ACCEPTS THE LANGUAGE

THE LANGUAGE IS "INHERENTLY AMBIGUOUS."

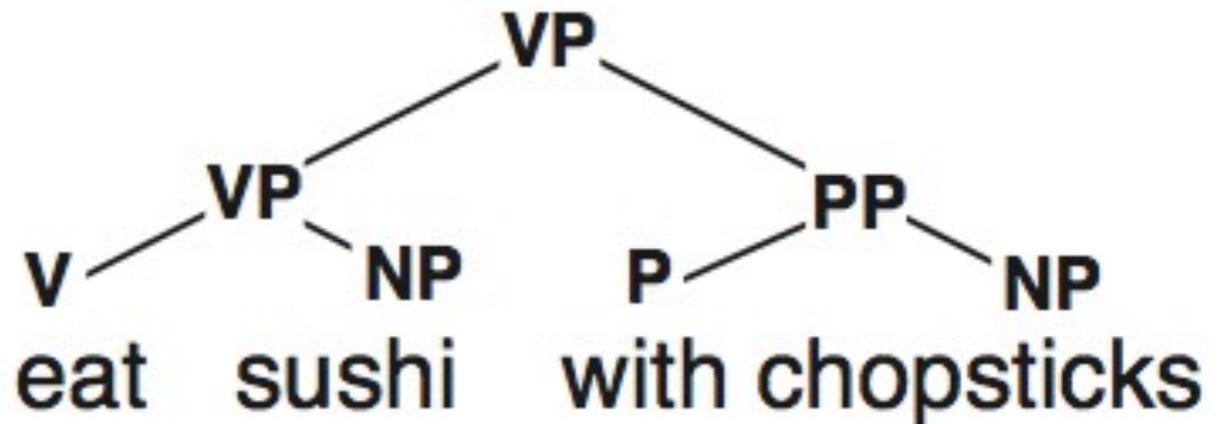
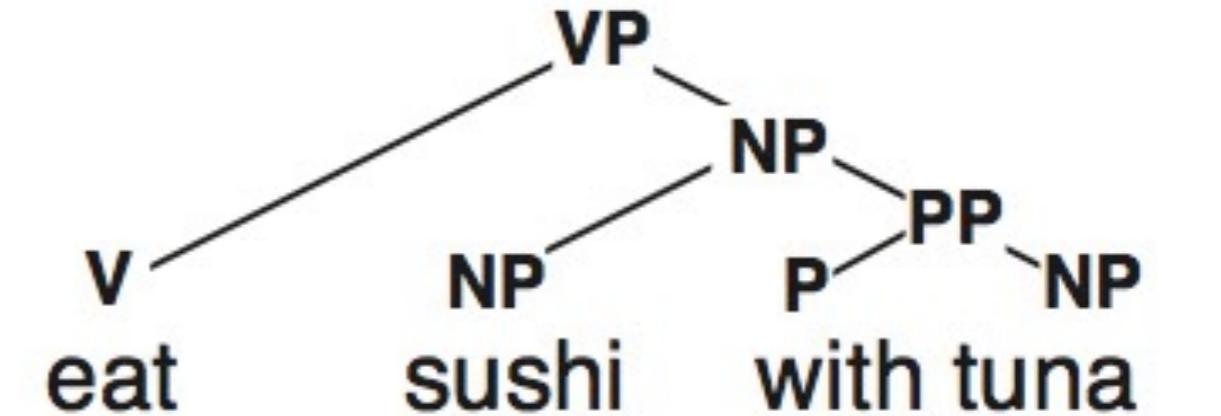
Parse Trees and Ambiguity in NL

- $N \rightarrow \{sushi, tuna\}$
- $P \rightarrow \{with\}$
- $V \rightarrow \{eat\}$
- $NP \rightarrow N$
- $NP \rightarrow NP\ PP$
- $PP \rightarrow P\ NP$
- $VP \rightarrow V\ NP$
- $VP \rightarrow VP\ PP$



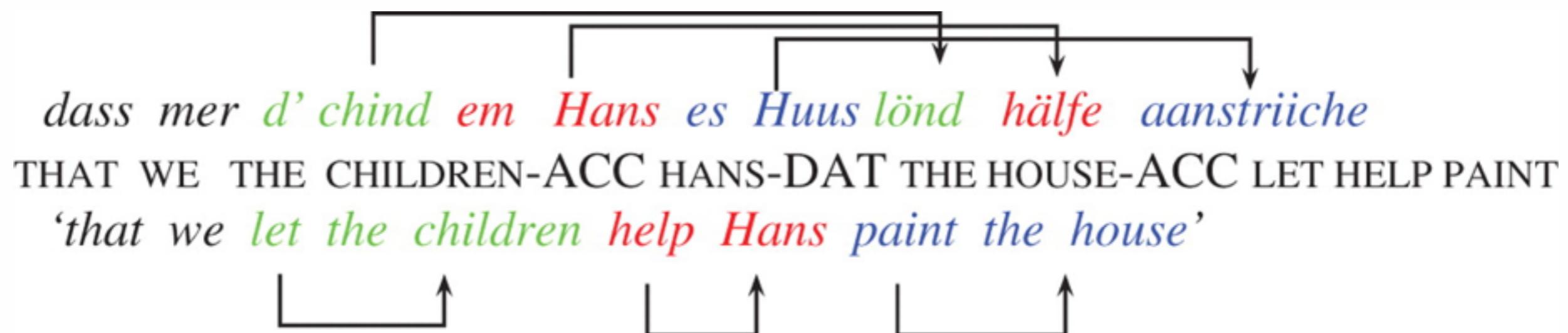
Parse Trees and Ambiguity in NL

- $N \rightarrow \{sushi, tuna\}$
- $P \rightarrow \{with\}$
- $V \rightarrow \{eat\}$
- $NP \rightarrow N$
- $NP \rightarrow NP PP$
- $PP \rightarrow P NP$
- $VP \rightarrow V NP$
- $VP \rightarrow VP PP$



Natural Languages Beyond Context-Free

- Dutch and Swiss German (Shieber, 1985)



STUART M. SHIEBER*

EVIDENCE AGAINST THE CONTEXT-FREENESS
OF NATURAL LANGUAGE**

Chomsky Hierarchy

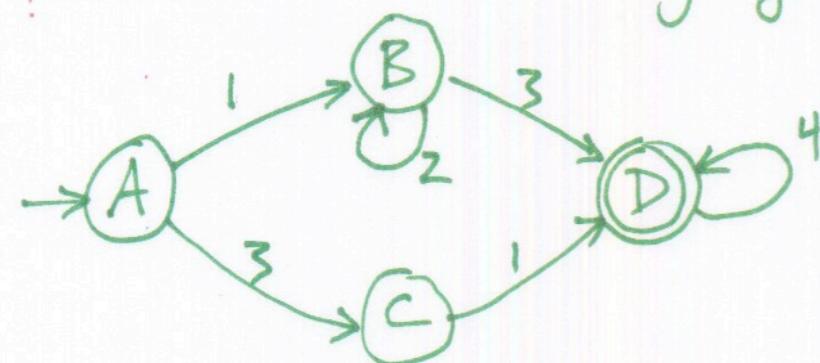
	Language	Automata	Parsing complexity	Dependencies
Type 3	Regular	Finite-state	linear	adjacent words
Type 2	Context-Free	Pushdown	cubic	nested
Type 1	Context-sensitive	Linear Bounded	exponential	
Type 0	Recursively Enumerable	Turing machine		

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.

$$A \rightarrow 1B$$

$$C \rightarrow 1D$$

$$A \rightarrow 3C$$

$$D \rightarrow 4D$$

$$B \rightarrow zB$$

$$D \rightarrow \epsilon$$

$$B \rightarrow 3D$$

closed under union but not intersection

ARE CONTEXT-FREE LANGUAGES
CLOSED UNDER UNION?

GRAMMAR 1:

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

GRAMMAR 2:

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

UNION:

$S \rightarrow S_1 \mid S_2$

$S_1 \rightarrow \dots$

$S_2 \rightarrow \dots$

YES!

ARE CONTEXT-FREE LANGUAGES
CLOSED UNDER CONCATENATION?

$S \rightarrow S_1 S_2$

$S_1 \rightarrow \dots$

$S_2 \rightarrow \dots$

YES!

ARE CONTEXT-FREE LANGUAGES
CLOSED UNDER INTERSECTION?

CONSIDER $L_1 = \{0^N 1^N 2^N\}$

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.

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

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.

HOWEVER

\overline{L} IS CONTEXT-FREE.

CFLs are closed under:
union, concatenation, star

"PALINDROMES
 $\subseteq \{wwr^n \mid n \in \mathbb{N}\}$ "

Complement: No

ARE CONTEXT-FREE LANGUAGES
CLOSED UNDER COMPLEMENT?

THESE ARE SETS.

DEMORGAN'S LAWS APPLY.

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

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.

HOWEVER

\overline{L} IS CONTEXT-FREE.

" $\{w \mid w \text{ is a palindrome}\}$ "

CFLs are closed under:
union, concatenation, star

$$\begin{aligned} L_1 &= \{x1y \mid |x| = |y|\} \\ L_2 &= \{x0y \mid |x| = |y|\} \\ \overline{L} &= L_1 \cup L_2 \cup (L_1 L_2) \cup (L_2 L_1) \end{aligned}$$

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

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

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

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

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

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

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

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

Same # of 0s and 1s

GRAMMAR DESIGN CHALLENGE

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

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:

$S \rightarrow OA \mid 1B \mid \epsilon$

$A \rightarrow 1S \mid OAA$

$B \rightarrow OS \mid 1BB$

SOLUTION #2:

$S \rightarrow SAB \mid \epsilon$

$A \rightarrow 0S1 \mid \epsilon$

$B \rightarrow 1S0 \mid \epsilon$

NOTE:

$C \rightarrow Cx \mid \epsilon$

GENERATES: x^*

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

$S \rightarrow SAB \mid \epsilon$

GENERATES:

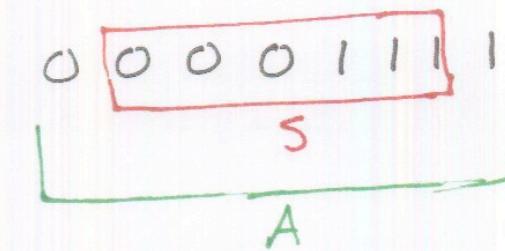
$ABABABAB\dots$

EVERY A CAN GO TO ϵ .

EVERY B CAN GO TO ϵ .

\Rightarrow ANY STRING OF A'S AND B'S!

0 1 0 1 0 1 0 1
A A A A



Inductive Proof part I

- $L_1 = \text{same number of } a\text{'s as } b\text{'s}$ $S \rightarrow aSbS \mid bSaS \mid \epsilon$

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 ϵ ($S \rightarrow \epsilon$): it trivially holds that the number of a 's and b 's in ϵ 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

- $L_1 = \text{same number of } a\text{'s as } b\text{'s}$ $S \rightarrow aSbS \mid bSaS \mid \epsilon$

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 $S \Rightarrow aSbS \Rightarrow aSb \Rightarrow aaSbSb \Rightarrow aaSbb \Rightarrow \dots \Rightarrow a^N b^N$.

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

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

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 $S \rightarrow \epsilon$, 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 RIGHHAND SIDE.

STEP 2

REMOVE RULES LIKE $A \rightarrow \epsilon$

STEP 3

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

STEP 4

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

$A \rightarrow BCDE$

$A \rightarrow Bcde$

STEP 5

MAKE SURE
 $A \rightarrow BC$ ← ONLY
ONLY 2.
MUST BE
VARIABLES.
 $A \rightarrow a$ ← ONLY
ONLY 1.
MUST BE A
TERMINAL
SYMBOL.

STEP 1:

MAKE SURE START SYMBOL
DOESN'T APPEAR ON RIGHHAND SIDE.
ADD NEW START SYMBOL.

EXAMPLE

$S \rightarrow ASA | aB$

$A \rightarrow B | S$

$B \rightarrow b | \epsilon$

$S_0 \rightarrow S$

$S \rightarrow ASA | aB$

$A \rightarrow B | S$

$B \rightarrow b | \epsilon$

→ Add new
Start
Symbol

Step 2: Remove Unary: $A \rightarrow \epsilon$

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.

$$\begin{array}{l} S \rightarrow ASA \mid aB \\ A \rightarrow B \mid S \\ B \rightarrow b \mid \epsilon \end{array}$$

$$\begin{array}{l} S_0 \rightarrow S \\ S \rightarrow ASA \mid aB \\ A \rightarrow B \mid S \\ B \rightarrow b \mid \epsilon \end{array}$$

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

$$\begin{array}{l} S_0 \rightarrow S \\ S \rightarrow ASA \mid aB \mid a \\ A \rightarrow B \mid S \mid \epsilon \\ B \rightarrow b \mid \epsilon \end{array}$$

$$\begin{array}{l} S_0 \rightarrow S \\ S \rightarrow ASA \mid aB \mid a \mid SA \mid AS \mid S \\ A \rightarrow B \mid S \mid \epsilon \\ B \rightarrow b \end{array}$$

Step 3: Remove Unary: A → B

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

$$\begin{array}{l} S_0 \rightarrow S \\ S \rightarrow ASA | aB | \mathbf{a} \\ A \rightarrow B | S | \mathbf{\epsilon} \\ B \rightarrow b | \epsilon \end{array}$$
$$\begin{array}{l} S_0 \rightarrow S \\ S \rightarrow ASA | aB | a | SA | AS | S \\ A \rightarrow B | S | \mathbf{\epsilon} \\ B \rightarrow b \end{array}$$

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

$$\begin{array}{l} S_0 \rightarrow S \\ S \rightarrow ASA | aB | a | SA | AS | S \\ A \rightarrow B | S \\ B \rightarrow b \end{array}$$
$$\begin{array}{l} S_0 \rightarrow S | ASA | aB | a | SA | AS \\ S \rightarrow ASA | aB | a | SA | AS \\ A \rightarrow B | S \\ B \rightarrow b \end{array}$$

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

$$\begin{array}{l} S_0 \rightarrow ASA | aB | a | SA | AS \\ S \rightarrow ASA | aB | a | SA | AS \\ A \rightarrow B | S | \mathbf{b} \\ B \rightarrow b \end{array}$$
$$\begin{array}{l} S_0 \rightarrow ASA | aB | a | SA | AS \\ S \rightarrow ASA | aB | a | SA | AS \\ A \rightarrow S | b | ASA | aB | a | SA | AS \\ B \rightarrow b \end{array}$$

Step 4: Binarize and Step 5: Embellish

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

$$\begin{array}{l} S_0 \rightarrow ASA \mid aB \mid a \mid SA \mid AS \\ S \rightarrow ASA \mid aB \mid a \mid SA \mid AS \\ A \rightarrow B \mid S \mid \mathbf{b} \\ B \rightarrow \mathbf{b} \end{array}$$

$$\begin{array}{l} S_0 \rightarrow ASA \mid aB \mid a \mid SA \mid AS \\ S \rightarrow ASA \mid aB \mid a \mid SA \mid AS \\ A \rightarrow S \mid \mathbf{b} \mid ASA \mid aB \mid \mathbf{a} \mid SA \mid AS \\ B \rightarrow \mathbf{b} \end{array}$$

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 $U_i \rightarrow a$. We simplified the resulting grammar by using a single variable U and rule $U \rightarrow a$.)

$$\begin{array}{l} S_0 \rightarrow AA_1 \mid UB \mid a \mid SA \mid AS \\ S \rightarrow AA_1 \mid UB \mid a \mid SA \mid AS \\ A \rightarrow b \mid AA_1 \mid UB \mid a \mid SA \mid AS \\ A_1 \rightarrow SA \\ U \rightarrow a \\ B \rightarrow b \end{array}$$

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

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

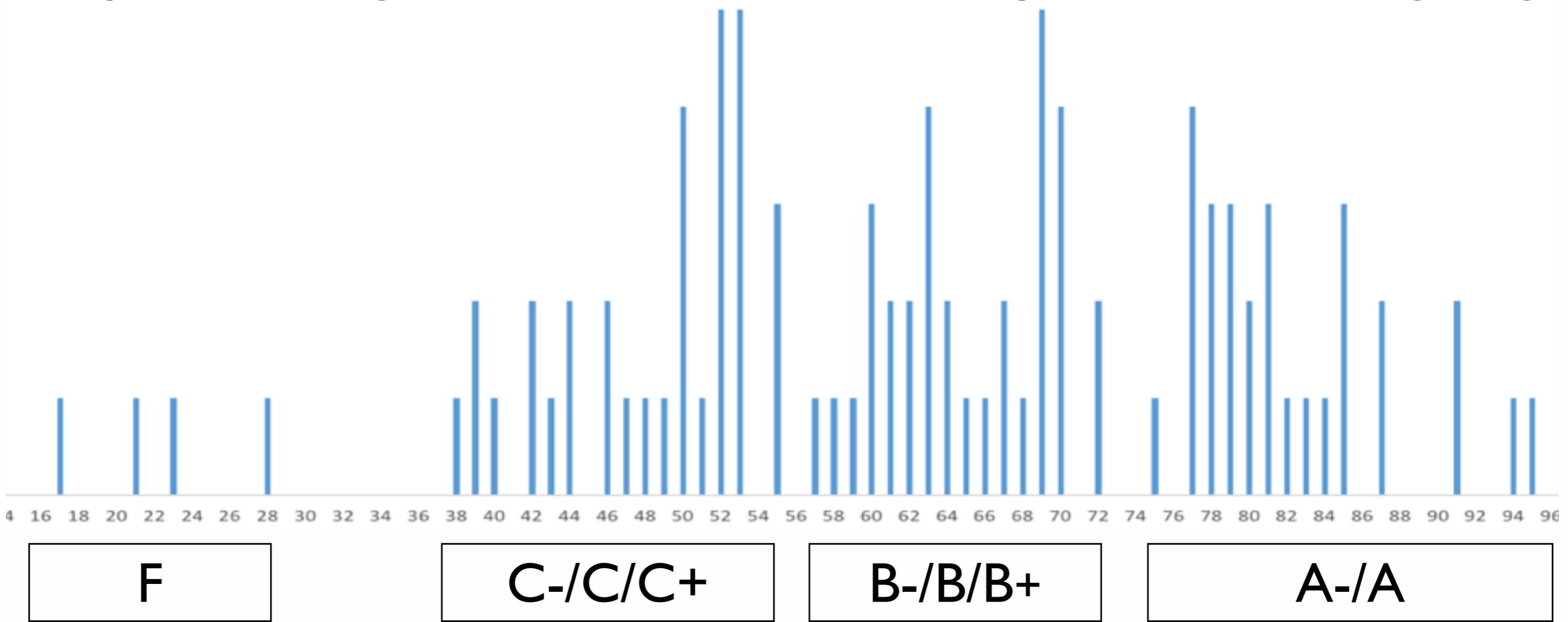
which is in Chomsky Normal Form

Infix-Postfix (reverse Polish); SVO-SOV

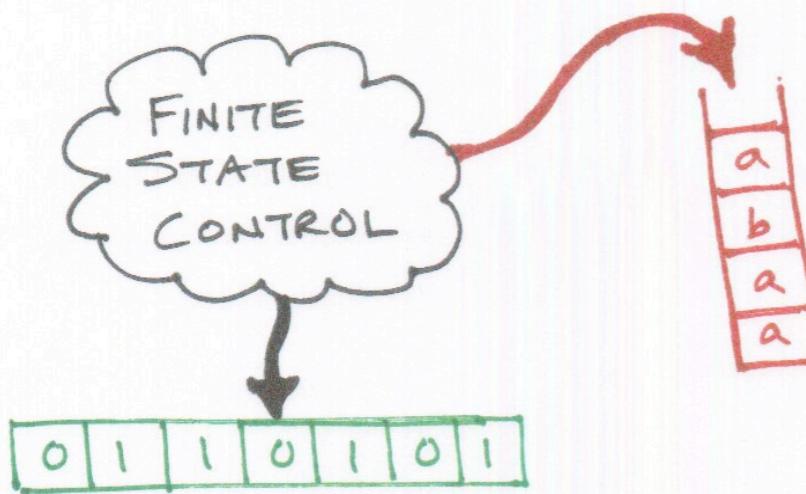
- ambiguous infix expr grammar $E \rightarrow E+E | E^*E | a$
 - two kinds of ambiguity: precedence and associativity
- unambiguous infix grammar (eliminates both ambiguities)
 - $E \rightarrow E+T|T, T \rightarrow T^*F|F, F \rightarrow a$ (see Sipser)
- postfix grammar (unambiguous) -- $E \rightarrow EE+ | EE^* | a$
- in natural language, infix \sim SVO, postfix \sim 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 \sqcup 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: $5\times 6 = 30$, homework: $1\times 7 = 7$
- quiz 8 will be the last quiz
 - the other 5% will be “corrections to your worst 5 quizzes”
- 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

Γ GAMMA,

MAY BE DIFFERENT FROM INPUT
ALPHABET, Σ

STATE TRANSITIONS:

MAY DEPEND ON STACK TOP.

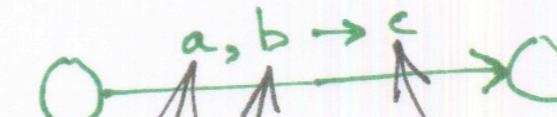
MAY PUSH ONTO THE STACK.

NON-DETERMINISTIC!

FINITE STATE MACHINE



PUSHDOWN AUTOMATON



Input symbol

May "e" be

Symbol on top of
the stack.
This symbol is
popped.

"e" means the
stack is neither
read nor popped

This symbol
is pushed
onto the stack.

"e" means
nothing
is pushed.

NONDETERMINISM

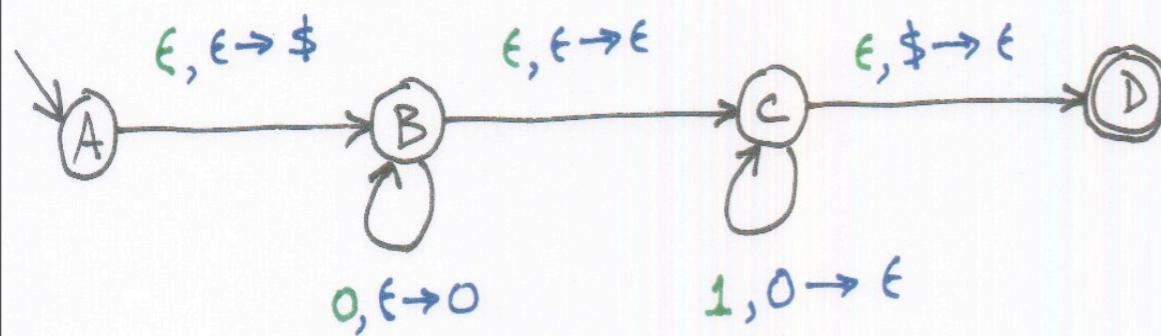
Example: $0^n \mid n$

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

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

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

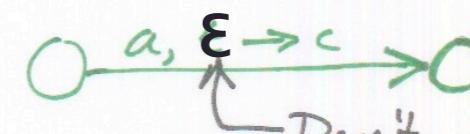
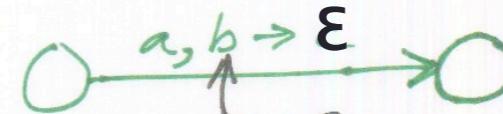
To detect bottom of stack.
 (We could use other symbols,
 but these seem clear enough.)



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

Σ = Input alphabet

Γ = Stack alphabet

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

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

$$\delta: Q \times \Sigma_\epsilon \times \Gamma_\epsilon \rightarrow P(Q \times \Gamma_\epsilon)$$

q_0 : Starting State $q_0 \in Q$

F : Accept States $F \subseteq Q$

EXAMPLE

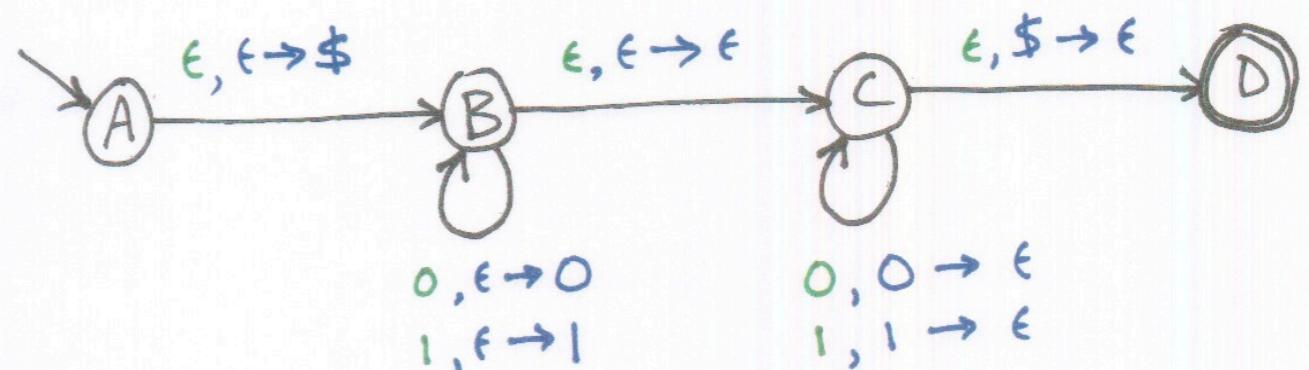
Palindrome

MADAM I'M ADAM
WAS IT A CAT I SAW
NO LEMON, NO MELON

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

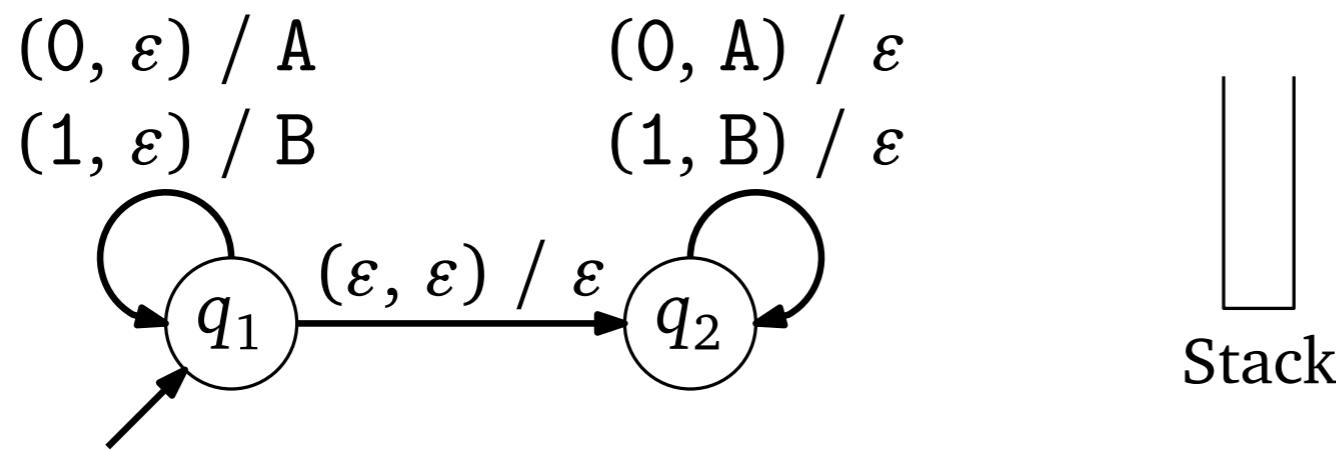
$$\begin{aligned} S &\rightarrow 0 S 0 \\ &\rightarrow 1 S 1 \\ &\rightarrow \epsilon \end{aligned}$$

wrong! this is ww^R
need to add $S \rightarrow 0 \mid 1$



Push-Down Automata: Example

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



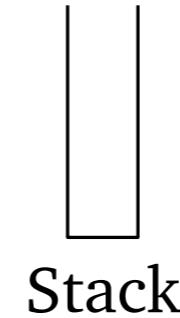
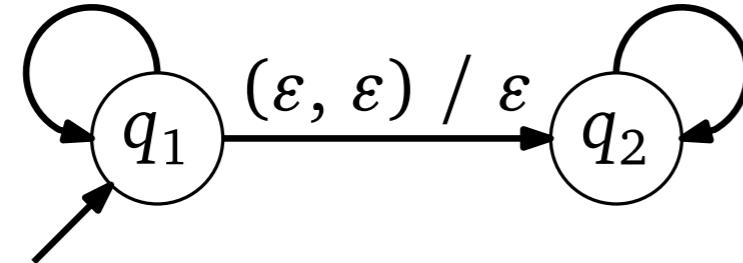
Push-Down Automata: Example

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

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

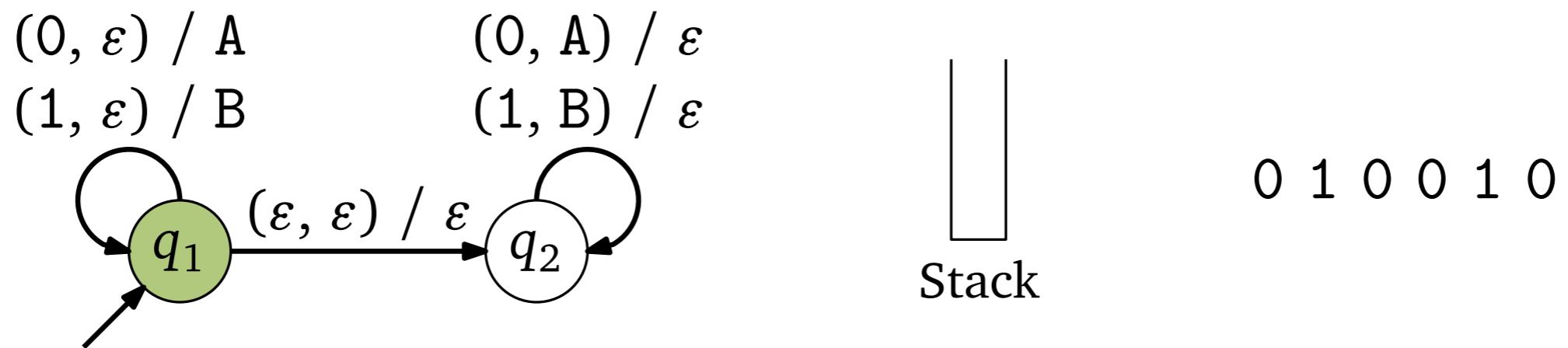
If the next input symbol is a 0 and the top of the stack is A,
then pop the A and push nothing onto the stack.

$(0, \varepsilon) / A$
 $(1, \varepsilon) / B$



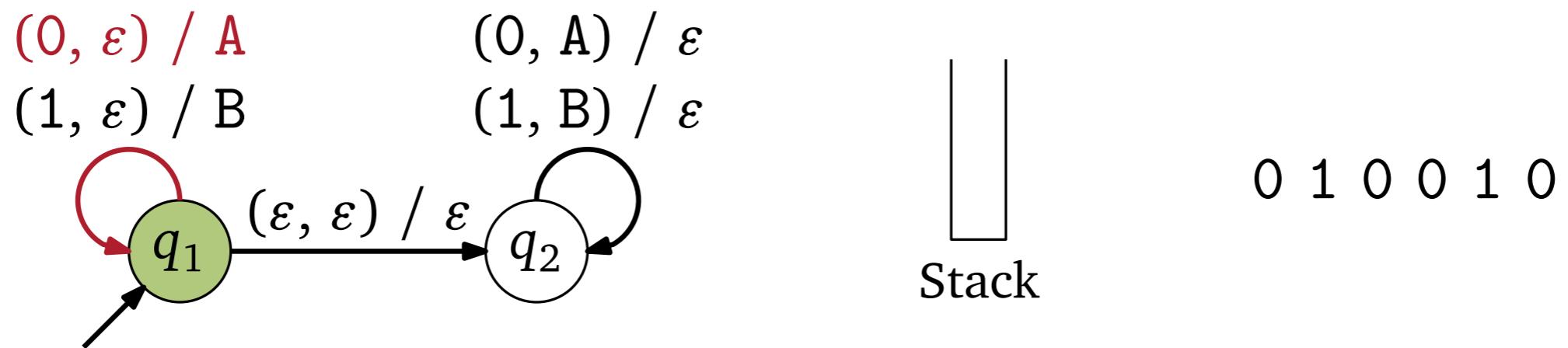
Push-Down Automata: Example

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



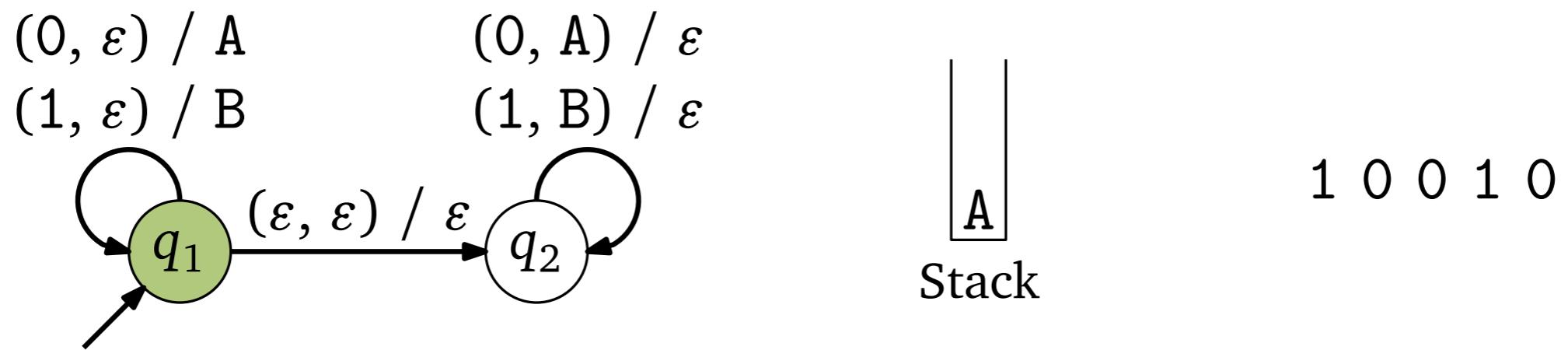
Push-Down Automata: Example

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



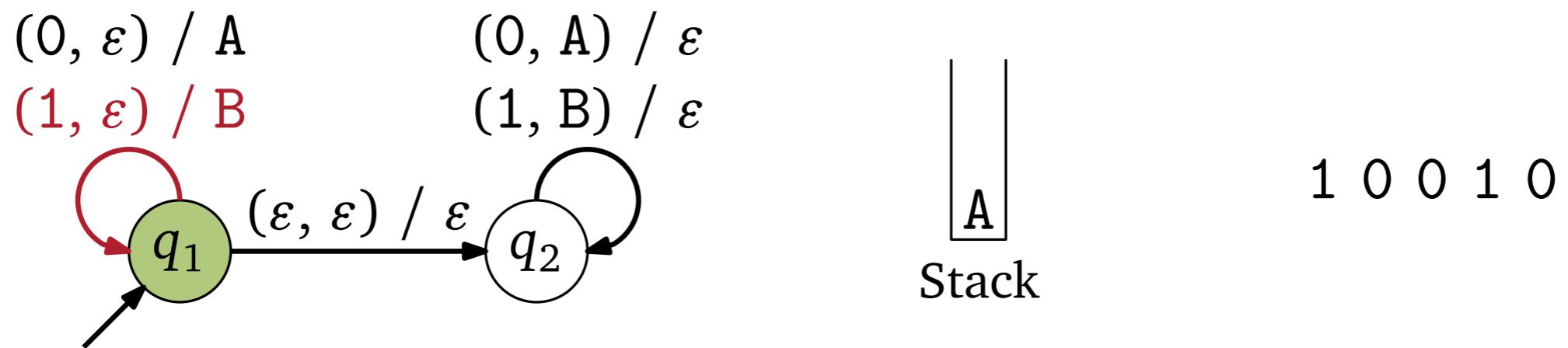
Push-Down Automata: Example

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



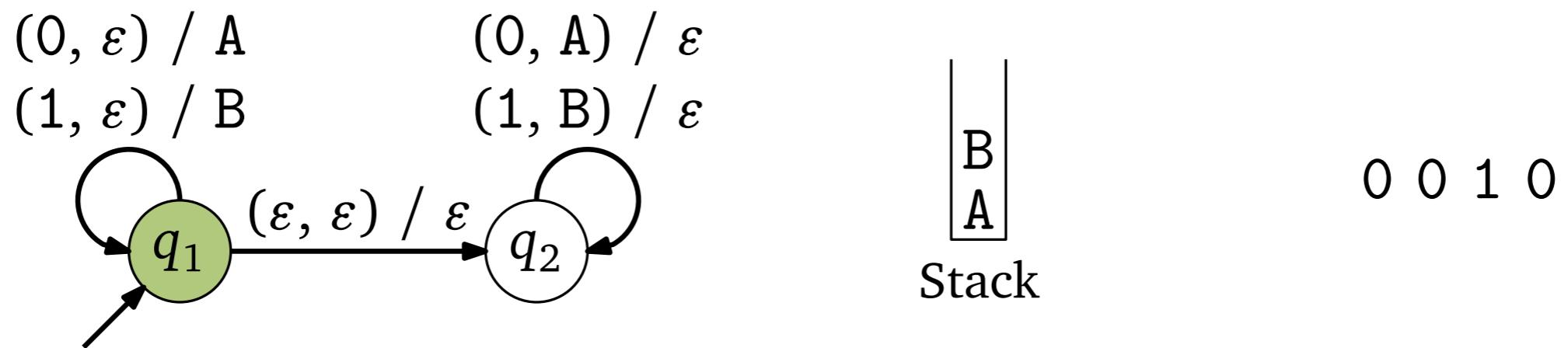
Push-Down Automata: Example

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



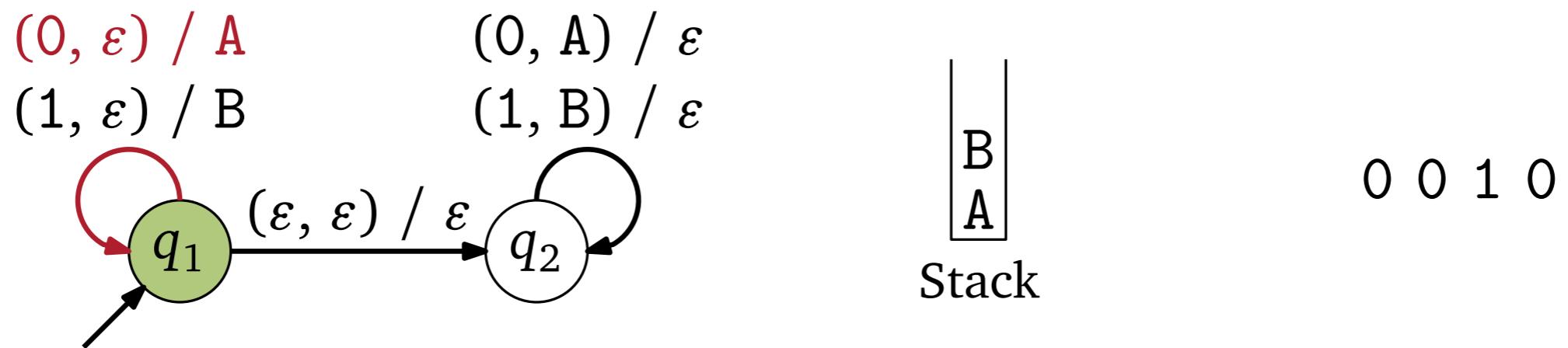
Push-Down Automata: Example

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



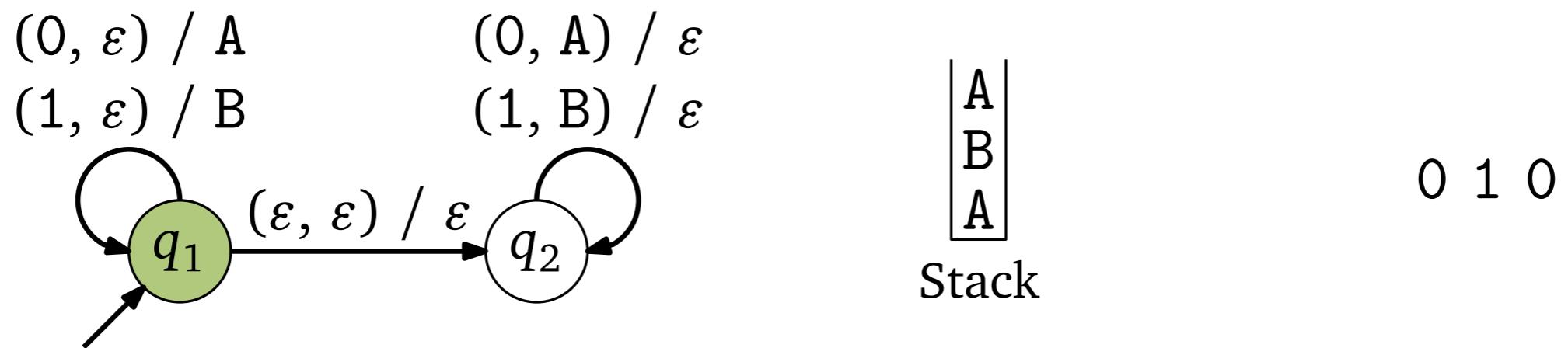
Push-Down Automata: Example

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



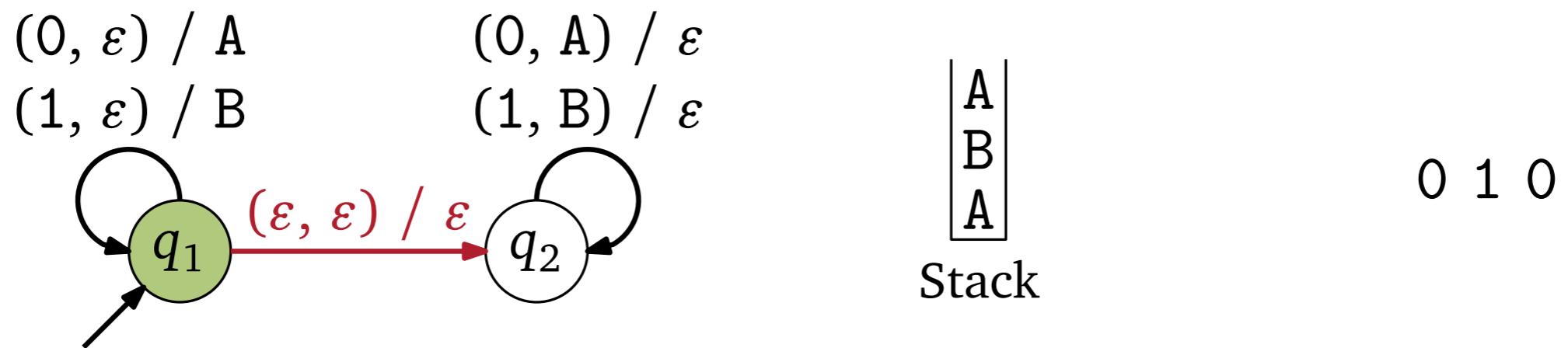
Push-Down Automata: Example

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



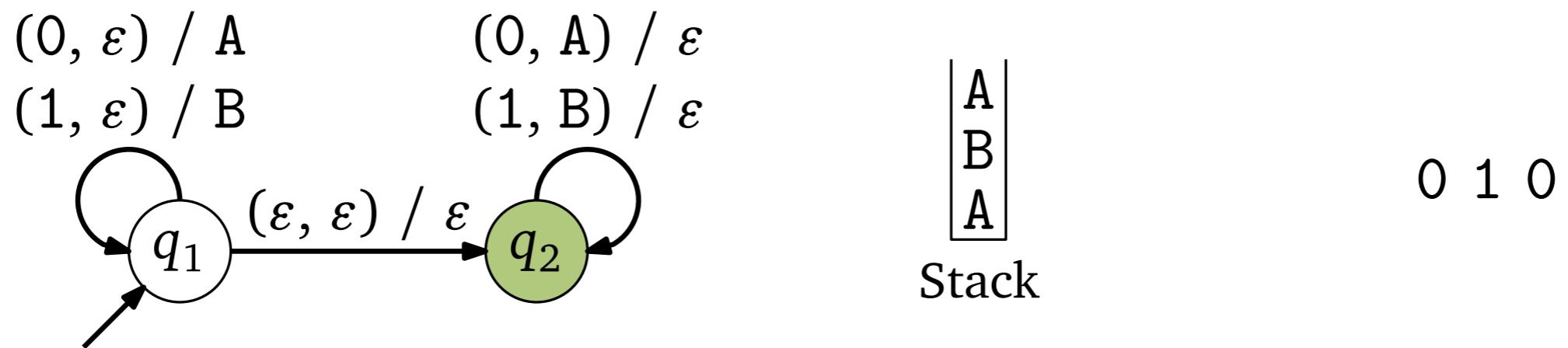
Push-Down Automata: Example

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



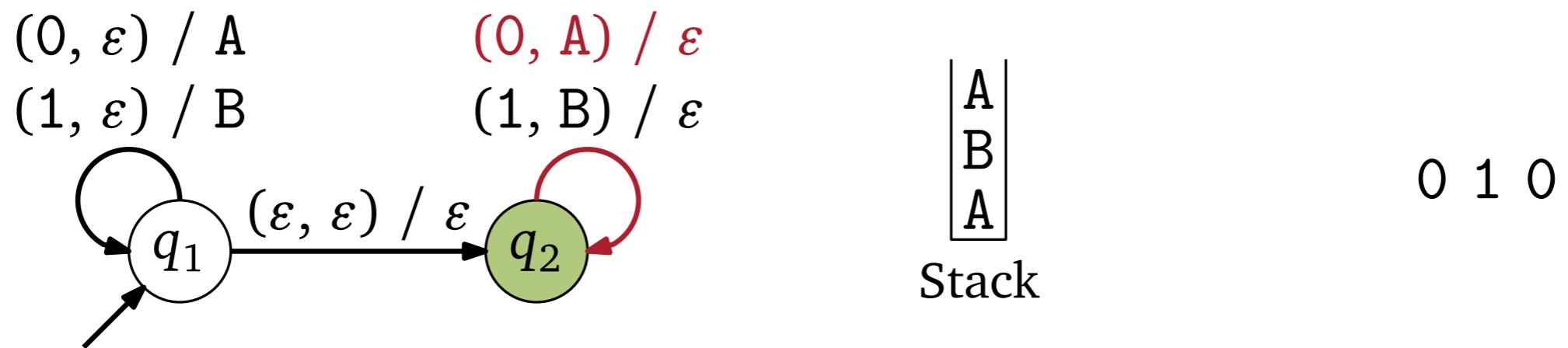
Push-Down Automata: Example

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



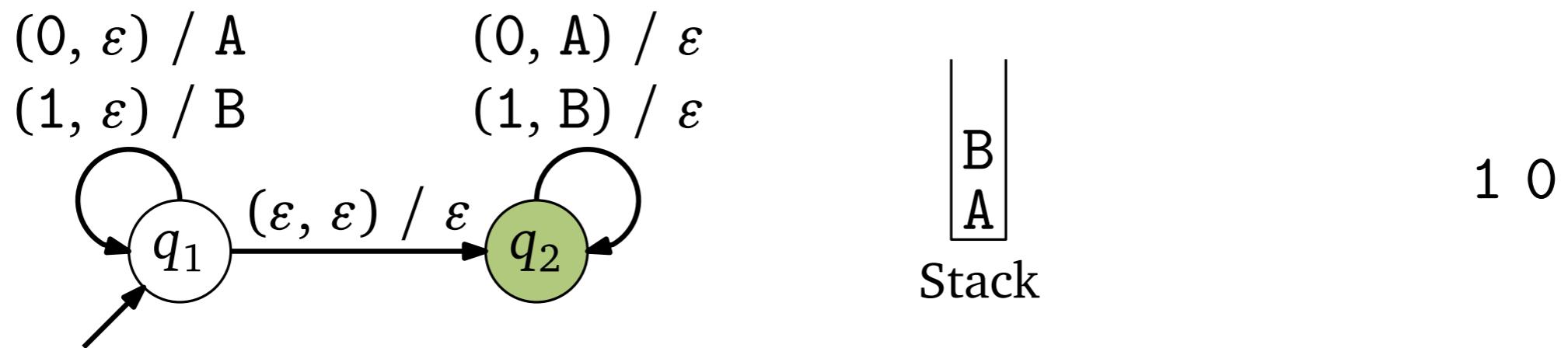
Push-Down Automata: Example

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



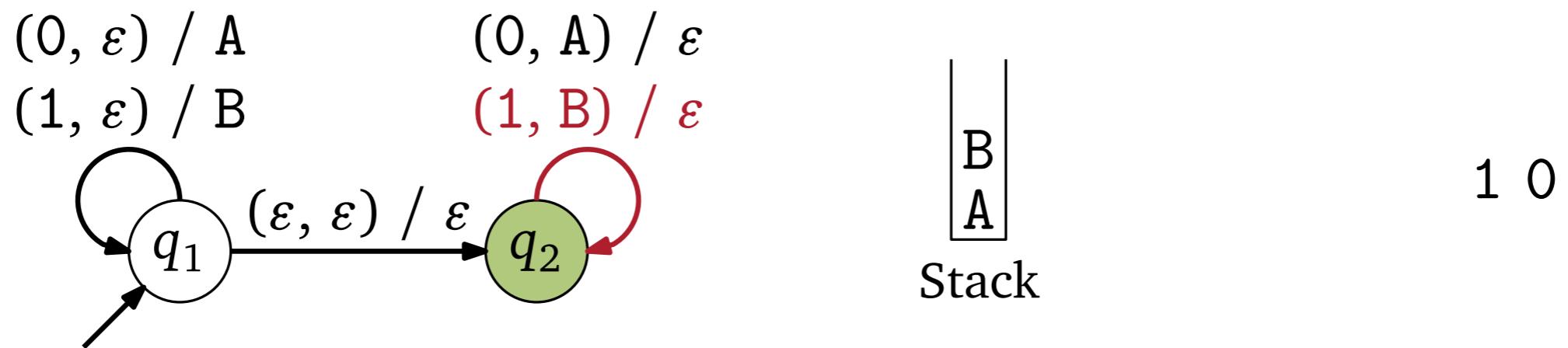
Push-Down Automata: Example

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



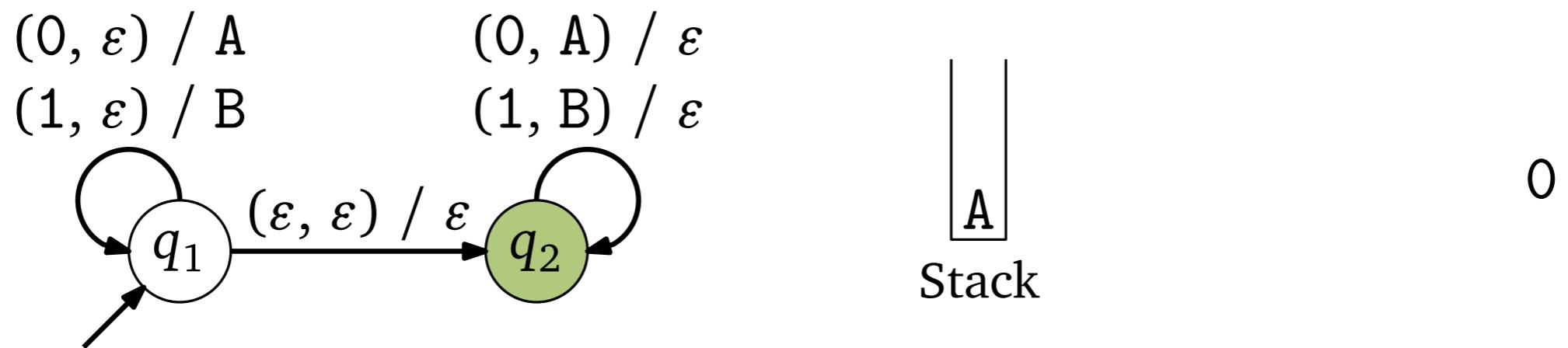
Push-Down Automata: Example

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



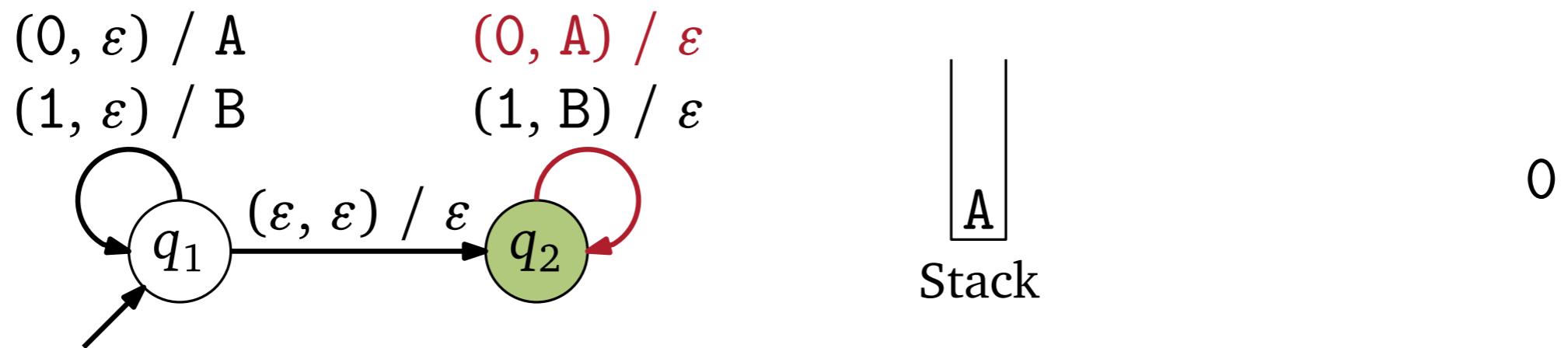
Push-Down Automata: Example

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



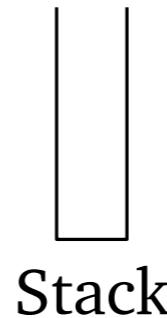
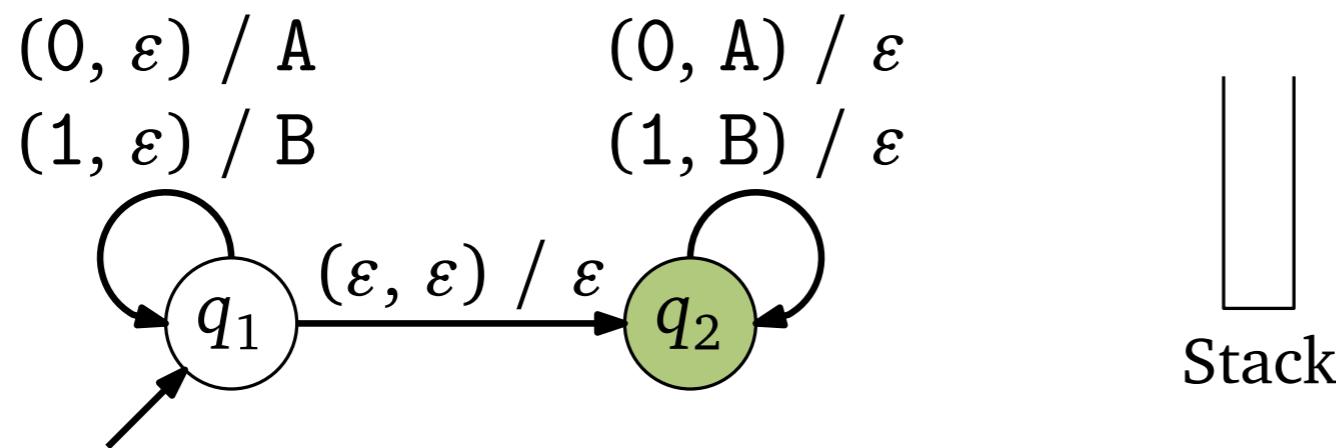
Push-Down Automata: Example

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



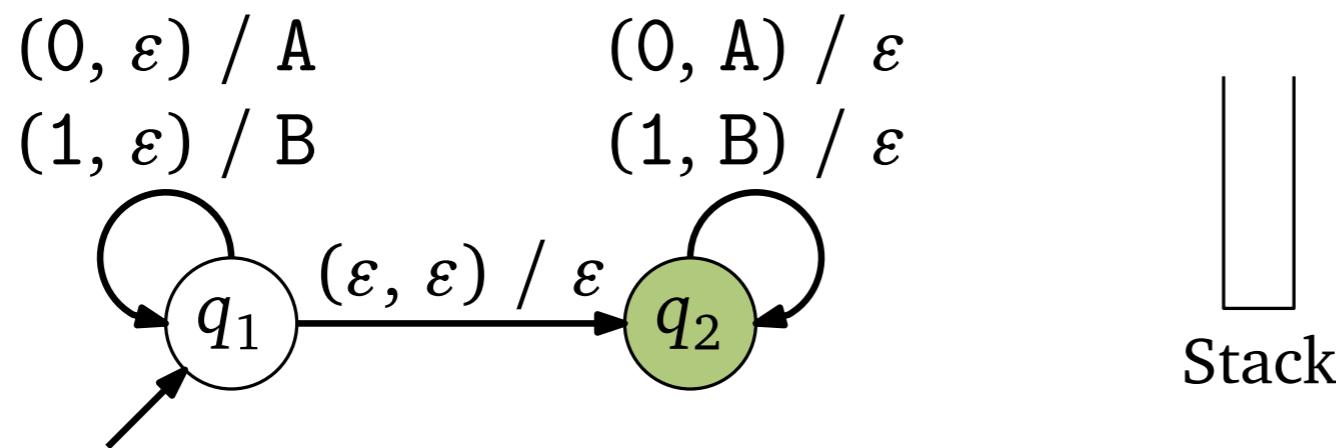
Push-Down Automata: Example

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



Push-Down Automata: Example

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

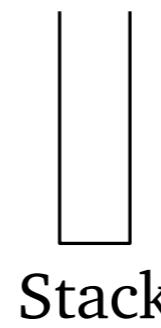
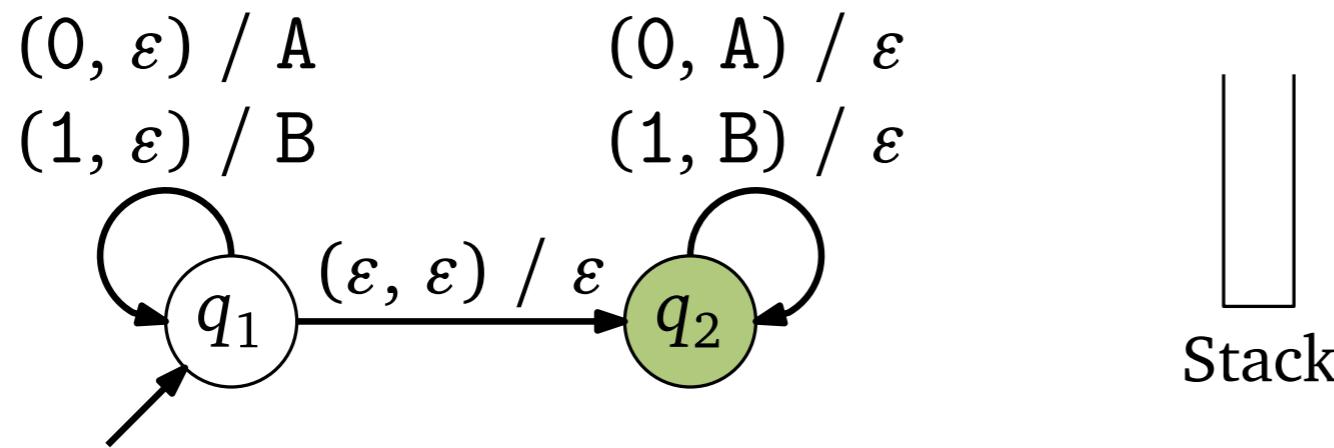


Accept by empty stack

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

Push-Down Automata: Example

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

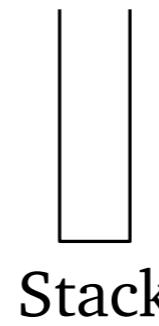
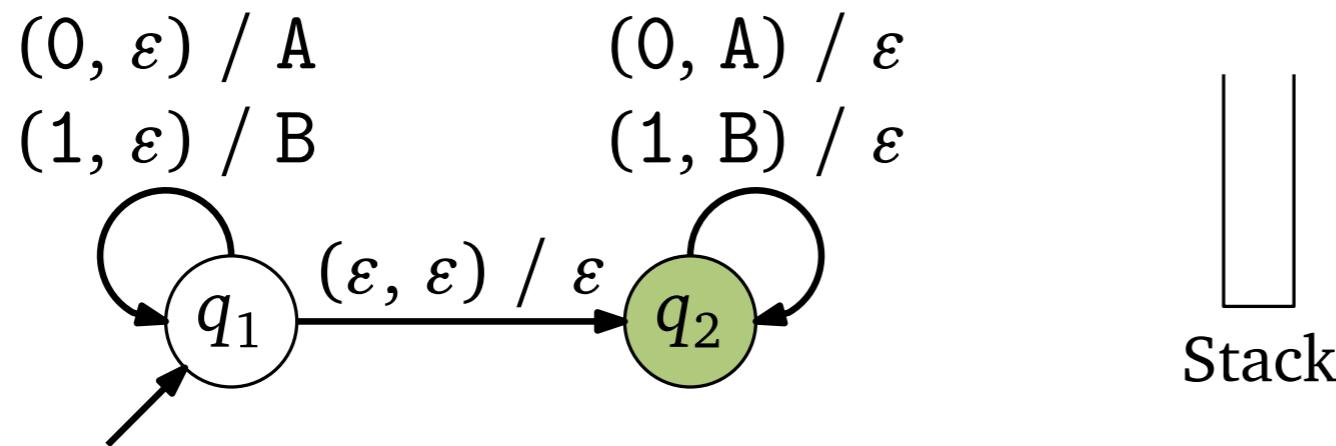


Accept by empty stack
PDA has consumed all
input and stack is empty
→ accept.

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***
- Σ is the ***input alphabet***
- Γ 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**
- Σ is the **input alphabet**
- Γ 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**

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 \mathcal{L} by empty stack if and only if there exists a PDA recognizing \mathcal{L} by final state.

Some Facts About PDA (1)

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

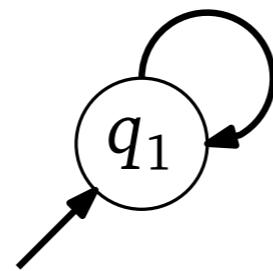
A language is context-free if and only if it can be recognized using a PDA.

“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:

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

Undecidable Problems in CFG

ARE TWO GRAMMARS EQUIVALENT?

"EQUIVALENT" = GENERATE THE SAME LANGUAGE

UNDECIDABLE!

CANNOT WRITE A COMPUTER PROGRAM.
[PROGRAM MAY NOT HALT!]

APPROACH:

- GENERATE EVERY STRING IN TURN. (INFINITELY MANY).
- TEST EACH STRING.

ACCEPTED BY GRAMMAR #1?

FIND A PARSE TREE.

(THIS IS DECIDABLE.)

ACCEPTED BY GRAMMAR #2?

- FIND A COUNTER-EXAMPLE?

HALT; PRINT "NOT EQUIVALENT!"

- OTHERWISE, KEEP LOOKING.

MAY NOT HALT... OR MAY...?

- NO WAY TO KNOW WHEN TO STOP LOOKING!!!

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

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

$\underbrace{00000}_5 \underbrace{11111}_5 \underbrace{00000}_5$

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

$$A \rightarrow \gamma$$

TYPE-1 LANGUAGES

CONTEXT-SENSITIVE

$$\alpha A \beta \rightarrow \alpha \gamma \beta$$

TYPE-0 LANGUAGES

RECURSIVELY ENUMERABLE
TURING ENUMERABLE.
TURING RECOGNIZABLE

$$\text{ex: } ABx \rightarrow ADEy$$

TERMINAL AND NON-TERMINAL

NOTE
... CB...
CAN NEVER BE REDUCED.
"C" CAN ONLY TURN INTO
A "3".
AND "3B" CAN NEVER
BE REDUCED.
 \Rightarrow B's must precede C's.

$$L = \{1^n 2^n 3^n \mid n \geq 0\}$$

APPROACH:

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

$$S \xrightarrow{*} \dots AAA BBB CCC \xrightarrow{*} \dots 111 222 333$$

INITIALLY THE STRING WILL START
WITH "1"s

$$111 \cancel{BBB} BBB CCC$$

Will never use "A" actually.

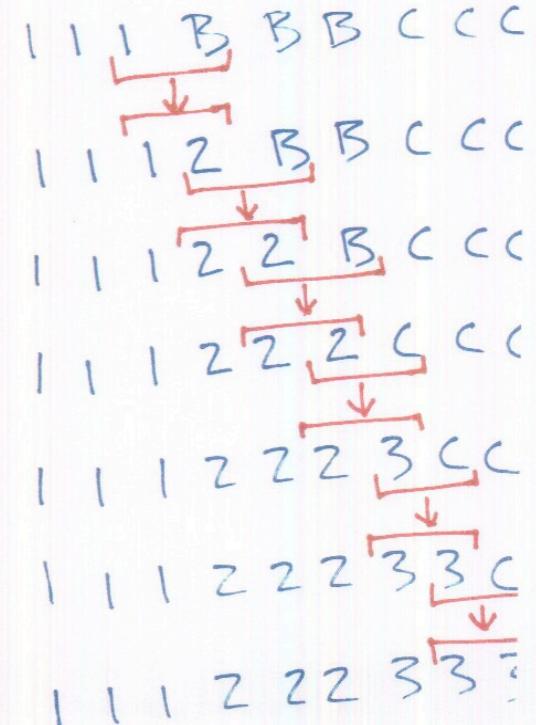
RULES TO FINISH IT UP:

$$1B \rightarrow 12$$

$$2B \rightarrow 22$$

$$2C \rightarrow 23$$

$$3C \rightarrow 33$$



Context-Sensitive: $1^n 2^n 3^n$

STEP 1: GENERATE THE
CORRECT NUMBER OF 1's, 2's, and 3's:

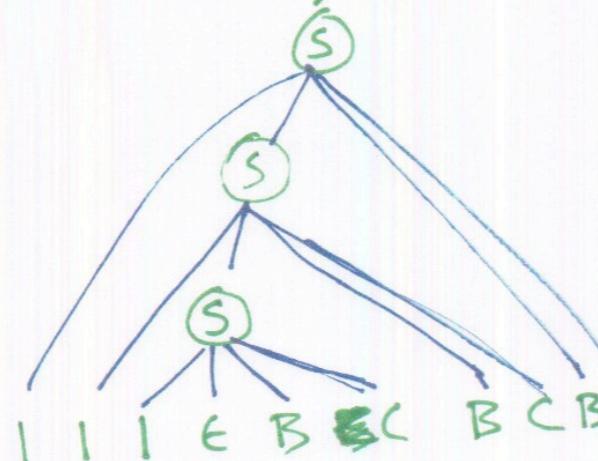
(JUST NOT THE RIGHT ORDER.)

$$S \rightarrow 1SBC$$

~~$$\cancel{S} \rightarrow \cancel{1} \cancel{S} B C$$~~

$$S \rightarrow \epsilon$$

1 1 1 1 BC BC BC BC



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

$$CB \rightarrow \bullet BC$$

STEP 3:

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

(Rules shown above.)

CONTEXT-SENSITIVE GRAMMAR

$$L = \{1^n 2^n 3^n \mid n \geq 0\}$$

$$\underline{S} \rightarrow \underline{1} \underline{S} BC$$

$$\underline{S} \rightarrow \underline{\epsilon}$$

FROM
BEFORE

$$\underline{CB} \rightarrow \underline{H} \underline{B}$$

$$\underline{HB} \rightarrow \underline{H} \underline{C}$$

$$\underline{HC} \rightarrow \underline{B} \underline{C}$$

REVISED

$$\underline{1B} \rightarrow \underline{12}$$

$$\underline{2B} \rightarrow \underline{22}$$

$$\underline{2C} \rightarrow \underline{23}$$

$$\underline{3C} \rightarrow \underline{33}$$

FROM
BEFORE

CFG == PDA

THEOREM

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

PROOF

PART 1:

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

PART 2:

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

Proof: Part I: CFG \Rightarrow PDA

GIVEN: A GRAMMAR

$$S \rightarrow BS \mid A$$

$$A \rightarrow \epsilon \mid a$$

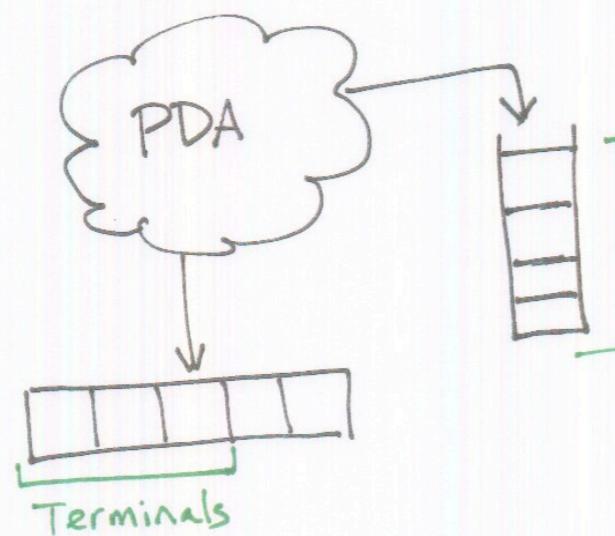
$$B \rightarrow BB1 \mid 2$$

FIND: (OR BUILD) A PDA.

PROOF,
PART 1

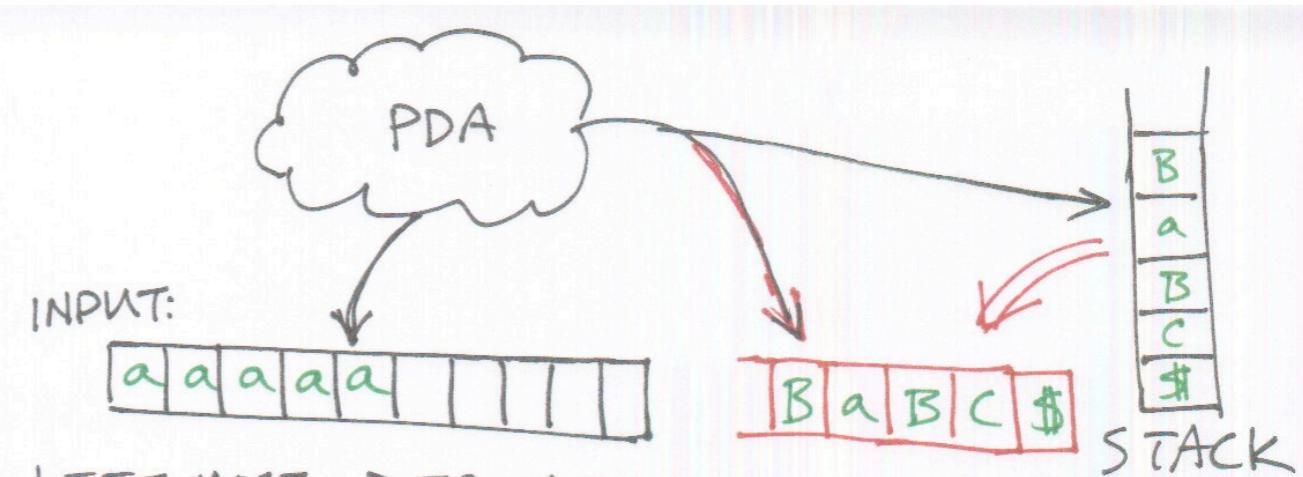
CONSIDER A DERIVATION: (LEFT-MOST)

$$\begin{aligned} S &\xrightarrow{\quad} S \\ &\Rightarrow \overline{BS} \\ &\Rightarrow \overline{BB1S} \\ &\Rightarrow \overline{2B1S} \\ &\Rightarrow \overline{221S} \\ &\Rightarrow \overline{221A} \\ &\Rightarrow \overline{221\epsilon} \end{aligned}$$



GENERAL FORM:

$$\begin{array}{c} \text{aaaaaa } \overline{\text{BaBBaCa}} \\ \text{Terminals} \qquad \text{Rest (both)} \end{array}$$



LEFT-MOST DERIVATION:

$$S^* \Rightarrow \dots \text{ aaaa } \boxed{BaBC} \Rightarrow \dots$$

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

RULE:
 $B \rightarrow ASA \times BA$

$$\dots \Rightarrow \text{aaaaa } \boxed{ASA \times BA} \text{ a } \boxed{BC}$$

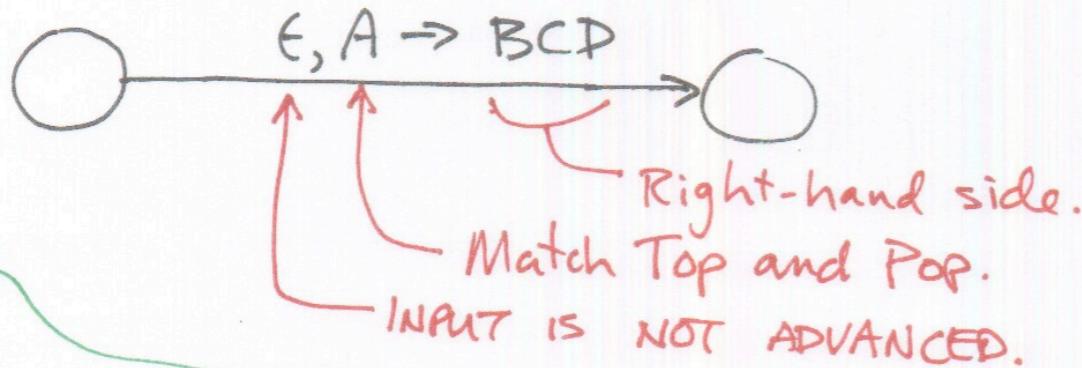
So:

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

Proof: Part I: CFG=>PDA

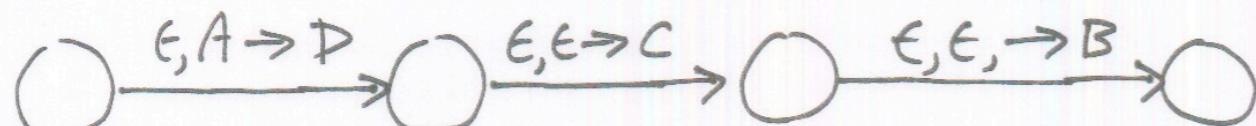
RULE: $A \rightarrow BCD$

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.)

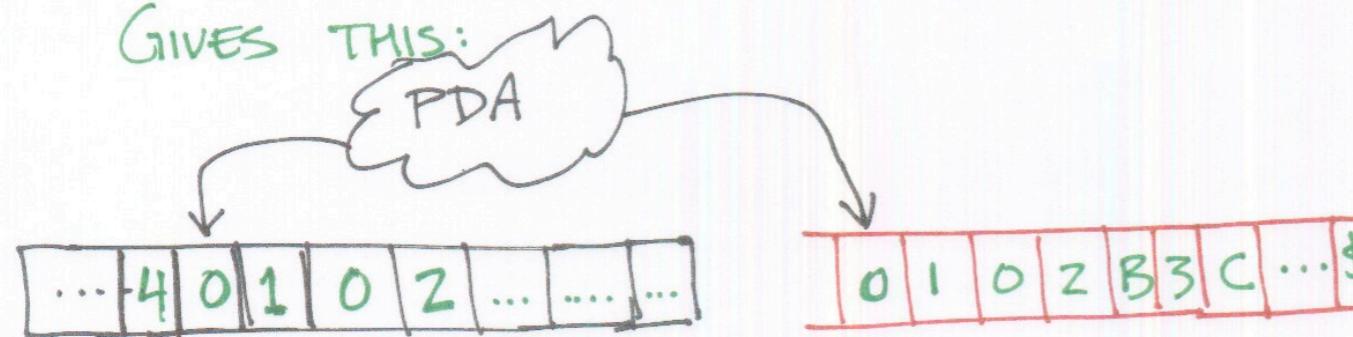
(JUST CHOOSE THE "RIGHT" RULE.)

Proof: Part I: CFG=>PDA

RULE :

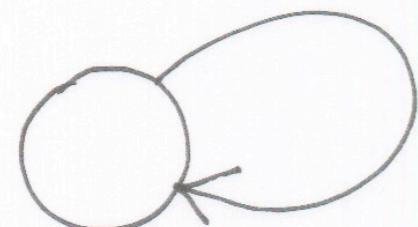
$$A \rightarrow 0102B3C$$

GIVES THIS:



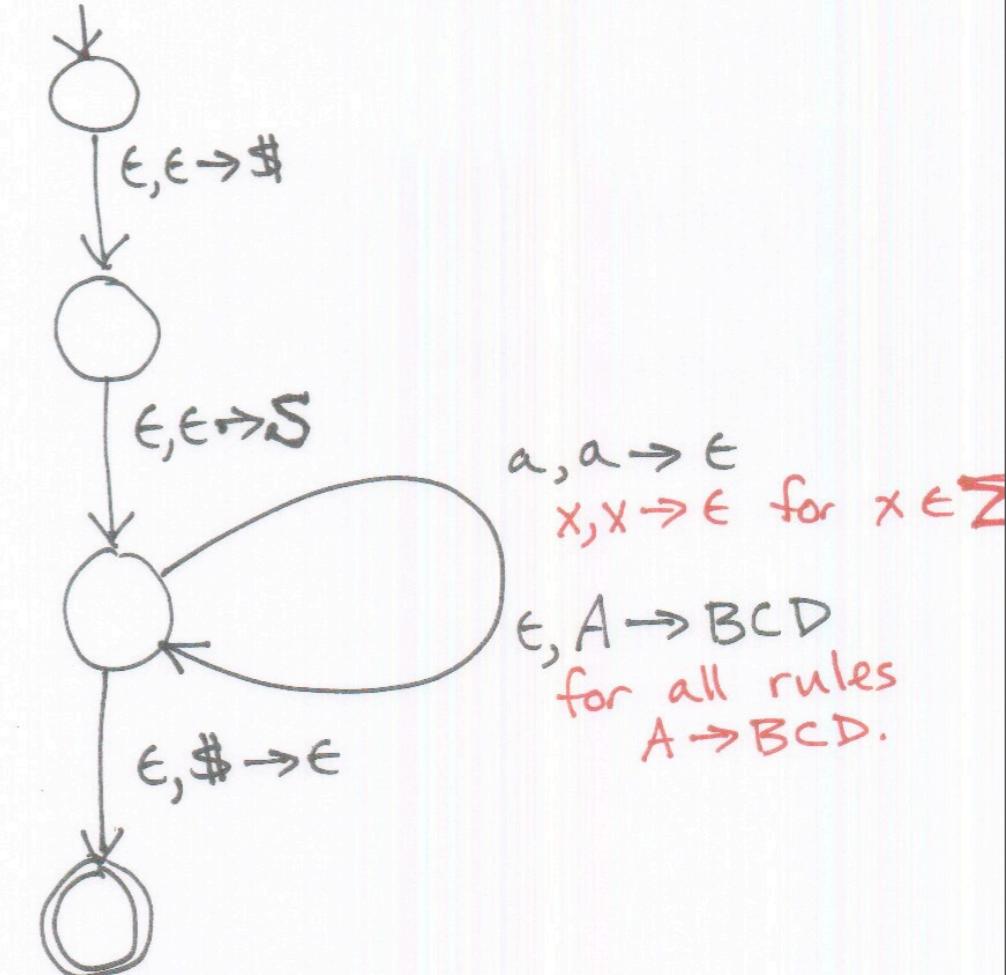
INPUT

So:
MATCH TERMINAL SYMBOLS TO
THE STACK TOP.



$$\begin{aligned} 0, 0 &\rightarrow \epsilon \\ 1, 1 &\rightarrow \epsilon \\ z, z &\rightarrow \epsilon \\ \text{etc. for all } x \in \Sigma. & \end{aligned}$$

THE FINAL MACHINE:



$a, a \rightarrow \epsilon$
 $x, x \rightarrow \epsilon$ for $x \in \Sigma$
 $\epsilon, A \rightarrow BCD$
 for all rules
 $A \rightarrow BCD$.

Proof: Part 2: CFG \leq PDA

PROOF, PART 2

WE ARE GIVEN: A P.D.A.

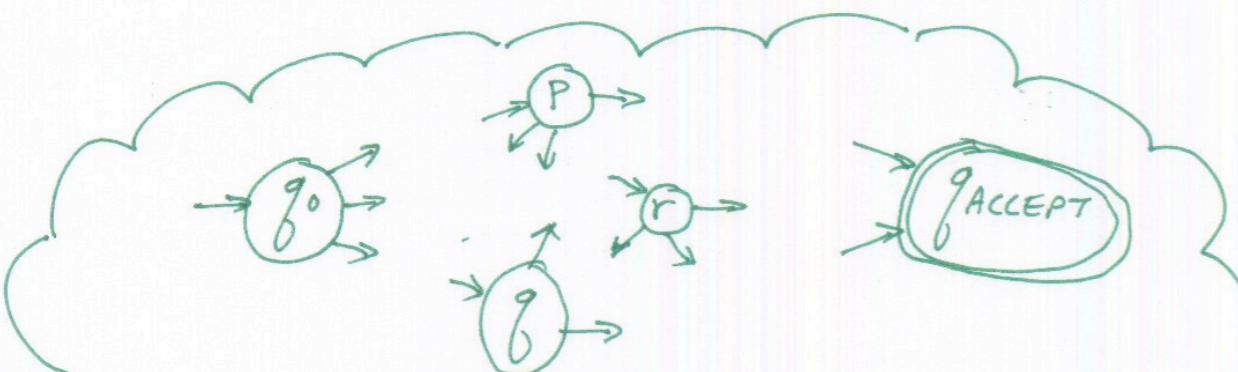
MUST BUILD: A CFG FROM IT.

STEP 1

SIMPLIFY THE PDA.

STEP 2

BUILD THE CFG.



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

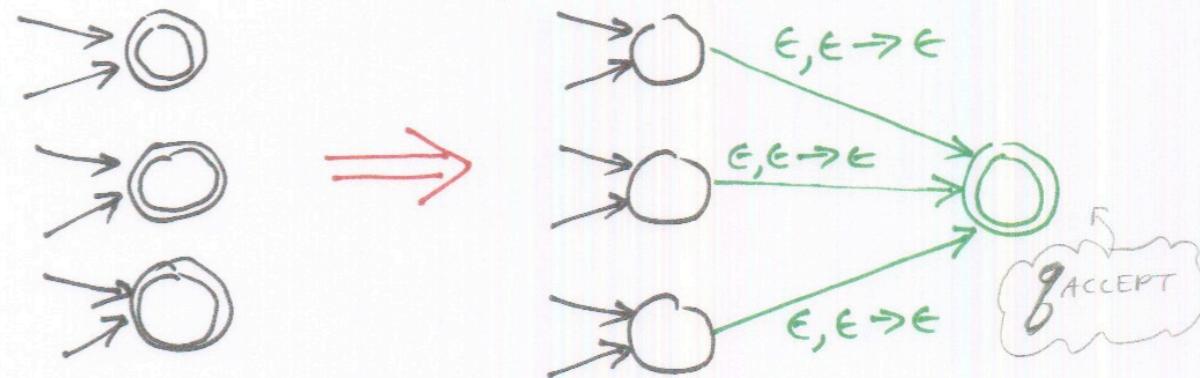
A_{Pq} A_{qP} A_{qg} A_{gq} ...

The starting nonterminal will be

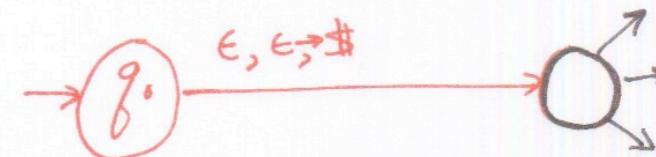
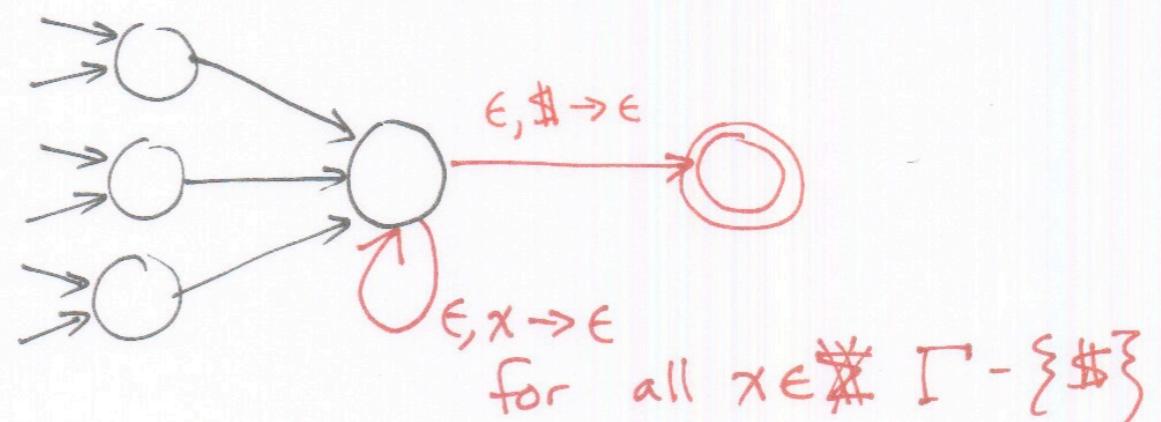
$A_{q_0 q_{ACCEPT}}$

SIMPLIFY THE PDA

① The PDA has only one ACCEPT state

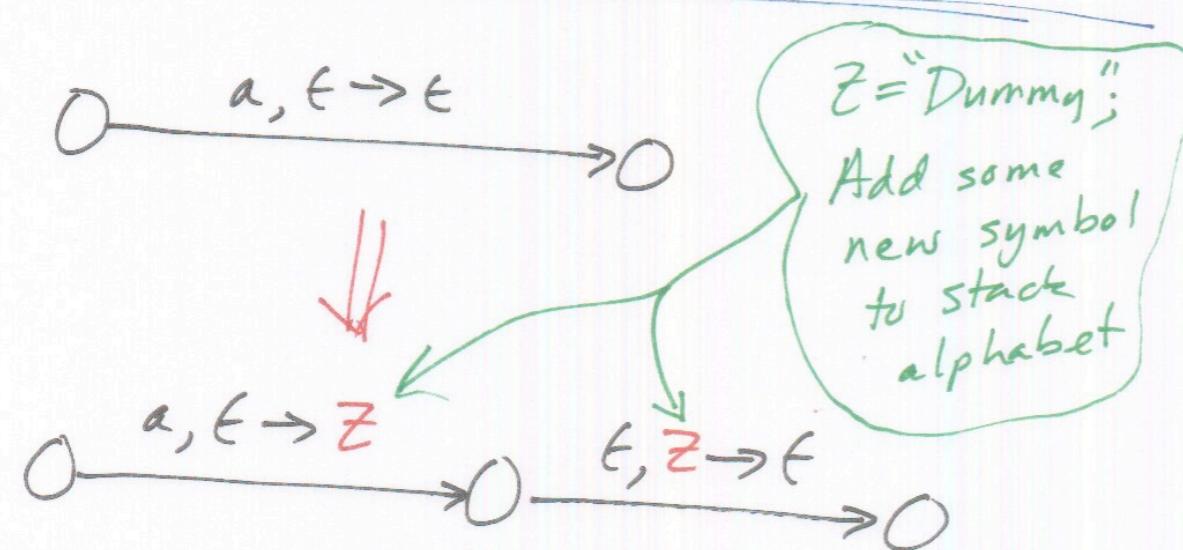
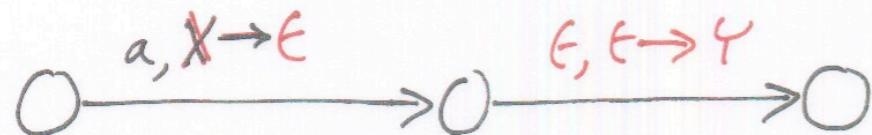
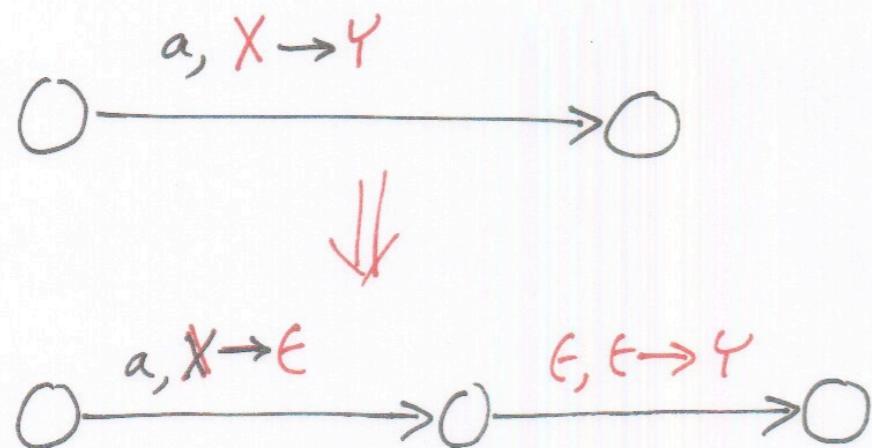


② The PDA empties its stack
before ACCEPTING.

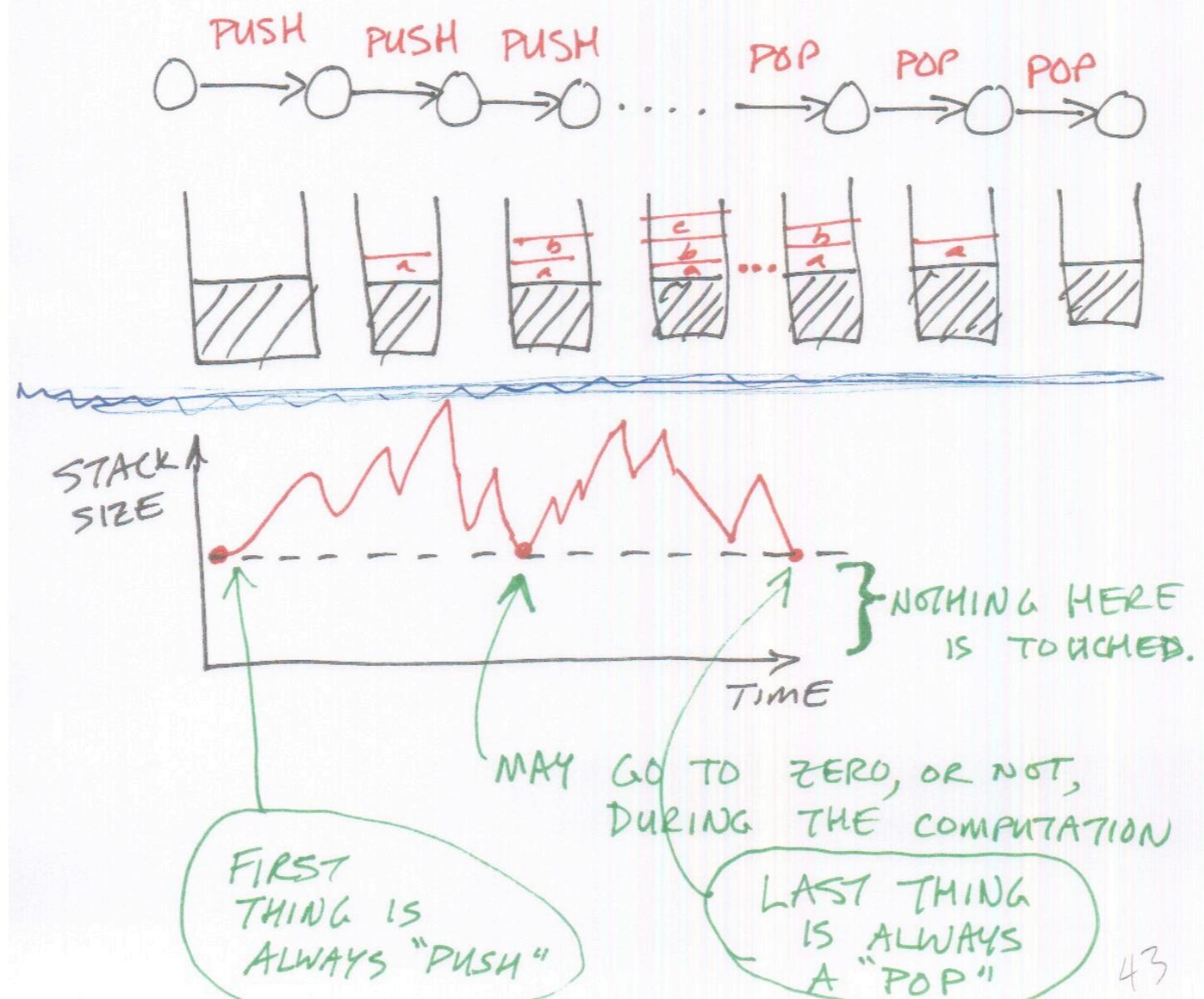


Proof: Part 2: CFG \leq PDA

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



"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 g
in the PDA.

Could we go from p to g
without touching the stack?

What strings would do that?

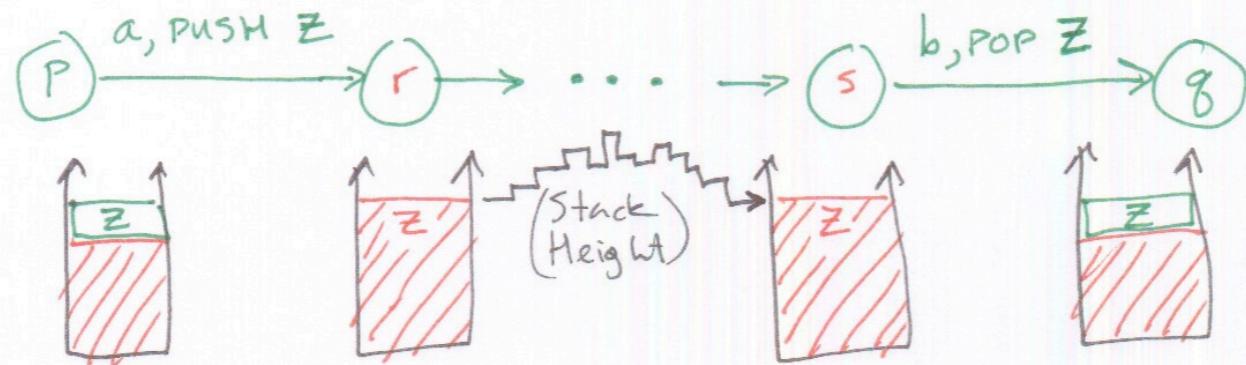
That is:

Starting w/ an empty stack,
we could go from p to g
and end up with an
empty stack.

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

The grammar we build will have
a non-terminal

→ We'll call it A_{pq}
that will generate exactly these
strings!

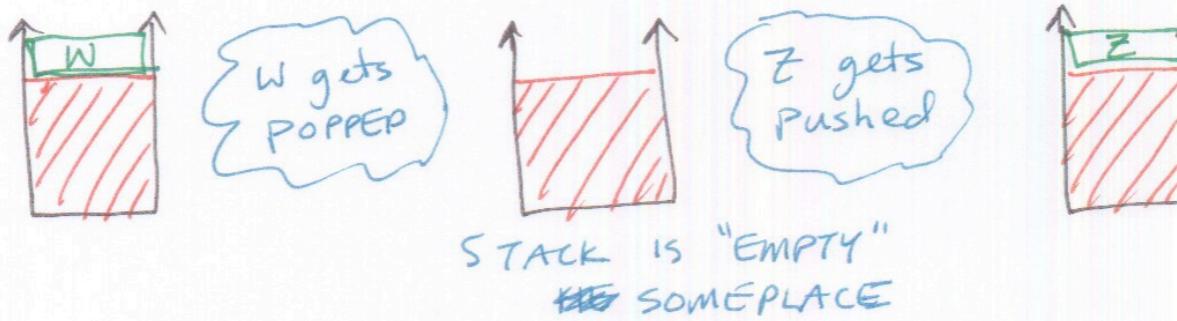
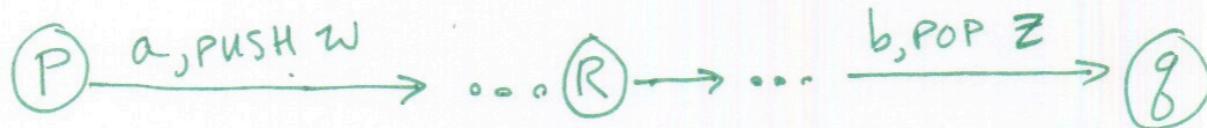


What strings can be generated/accepted by following this path?

"a... b"

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

This rule will generate exactly those strings!



What strings can be generated by following this path?

"aaa... $\overset{a}{\underset{\text{From } P \text{ to } R}{\mid}}$ b...bb" $\underset{\text{From } R \text{ to } q}{\mid}$
 From P to R From R to q

$A_{Pg} \rightarrow A_{Pr} A_{Rg}$
 This rule will generate exactly those strings!

If we have these edges



And we could get from R to S
without touching the stack,

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

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

FOR EACH $p, q, r, s \in EQ$ in the PDA

such that $\delta(p, a, \epsilon) = \text{contains } (r, t)$

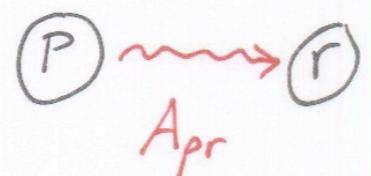
and $\delta(s, b, t) = \text{contains } (q, \epsilon)$

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

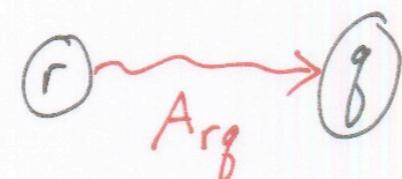
THEN ADD THIS RULE TO THE CFG:

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

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:

$$A_{pq} \rightarrow Apr Arg$$

There is a trivial way to get from state P to itself without touching the stack: The string ϵ .

So add

$$A_{pp} \rightarrow \epsilon \text{ for every state } P.$$

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.

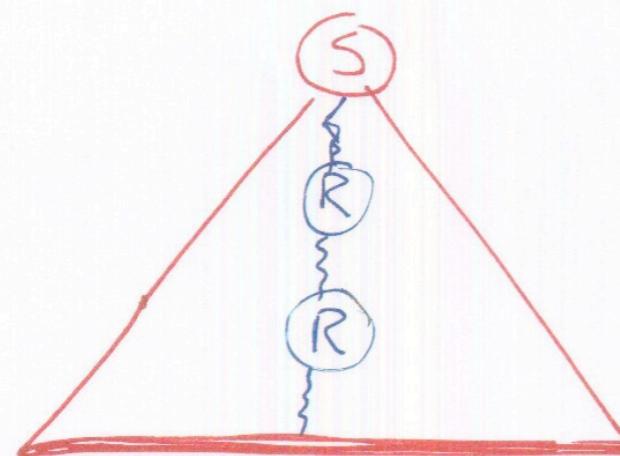
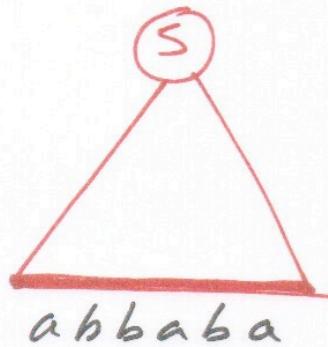
The grammar we seek should accept generate exactly these strings.

Our START NON-TERMINAL IS:

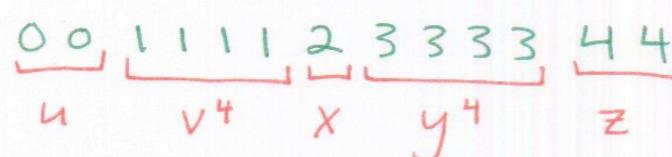
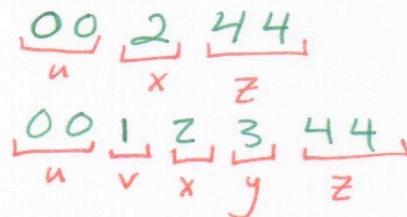
$$A_{q_0 q_{\text{ACCEPT}}}$$

Pumping Lemma for CFGs

PUMPING LEMMA FOR CFG's



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



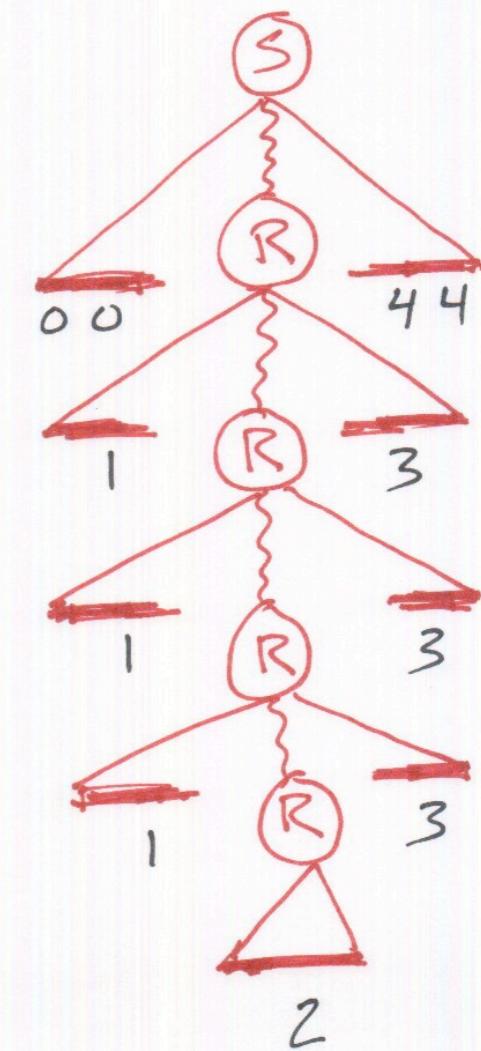
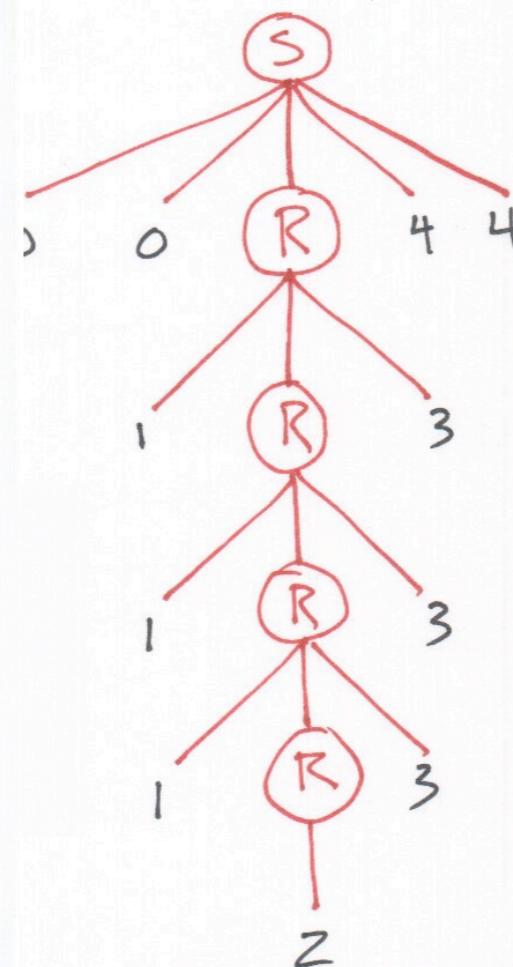
uvⁱxy^jz
is also in
the Language!

CONSIDER:

$$L = \{001^N 23^N 44 \mid N \geq 0\}$$

$$\begin{array}{l} S \rightarrow 00R44 \\ R \rightarrow 1R_3 | 2 \end{array}$$

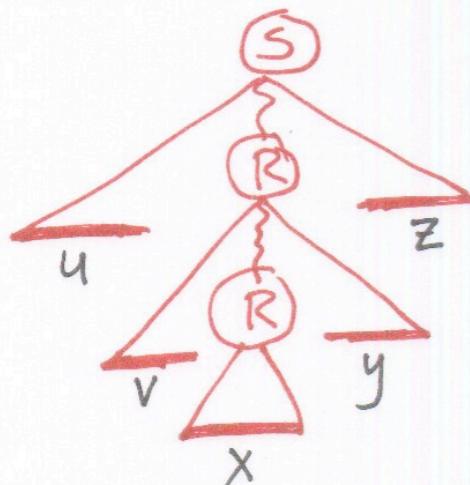
HOW CAN WE GENERATE "REALLY LONG" STRINGS?



Harder Case

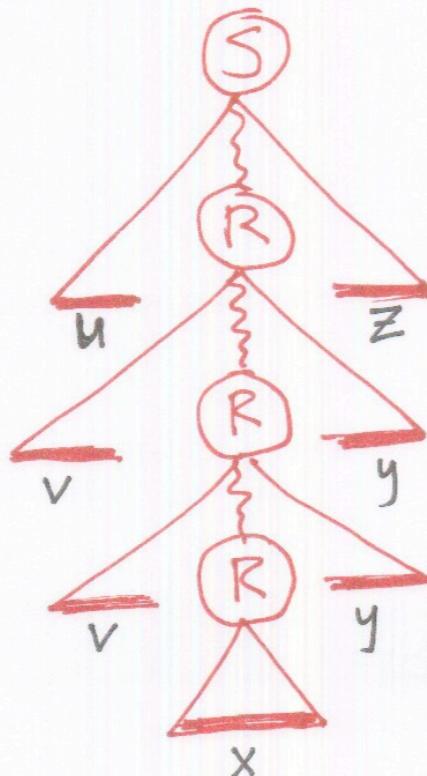
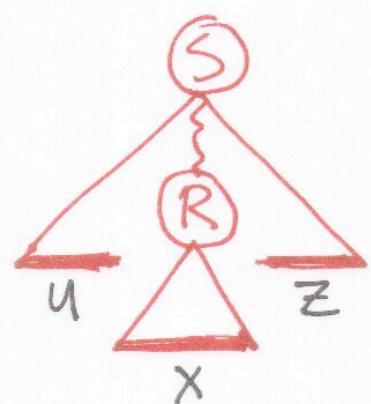
FOR ALL STRINGS THAT ARE "LONG ENOUGH,"

SOME NON-TERMINAL HAS TO BE
REPEATED IN THE PARSE TREE.



$$R \xrightarrow{*} v R y$$

THEREFORE, THESE ARE ALSO LEGAL
PARSE TREES:



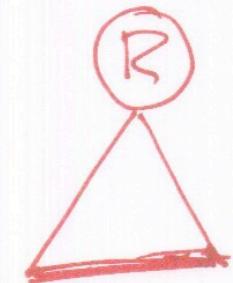
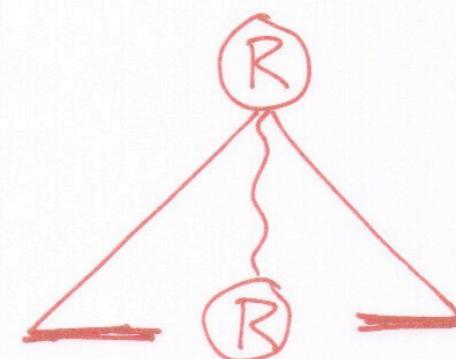
UVⁱX^jY^kZ^l
is also in
the language!

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

$$R \xrightarrow{*} \dots R \dots$$

AND FINALLY TO FINISH:

$$R \xrightarrow{*} x$$



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)

$$\cancel{s} = uvxyz$$

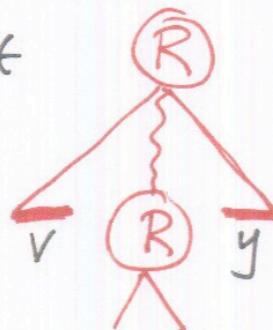
such that all strings of the form
 uv^ixy^iz
are also in the language.

NOTES:

- v and y cannot both be ϵ
 $|vy| > 0$

- 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:

$uv^i xy^i z$ 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"

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

"THERE EXISTS"

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

"AND"

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

"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."

57

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

PUMPING PROPERTY

$\exists p$

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

$$\exists u v x y z = s$$

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 \sim (PUMPING PROPERTY) holds.

NOT-PUMPING PROPERTY

$\forall p$

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

$$\forall u v x y z = s$$

such that

- ① $uv^ixy^iz \notin L, \forall i \geq 0$ OR
- ② $|vy| > 0$ OR
- ③ $|vxy| \neq p$

78

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 \sim (PUMPING PROPERTY)

Let p be the pumping length.

(No constraints on p . We'll show it $\nexists p$.)

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

We'll use: $a^p b^p c^p$ [$\exists s \dots$]

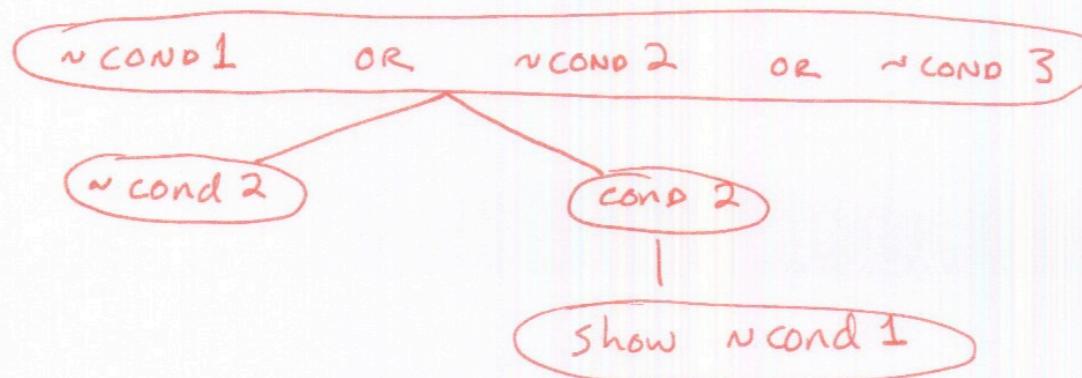
Now look at all ways to divide
it up. [$\forall u v w x y z = s \dots$]

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

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

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.

$a \underbrace{aaa} \quad b \underbrace{bbb} \quad c \underbrace{ccc}$
 $v \qquad \qquad \qquad y$

$a \underbrace{aaa} \quad b \underbrace{bbb} \quad c \underbrace{ccc}$
 $v \qquad \qquad \qquad y = \epsilon$

One symbol will always be left out.

Pump s to uv^2xy^2z

$a \underbrace{aaaaaa} \quad b \underbrace{bbb} \quad c \underbrace{ccc}$
 $v \qquad v \qquad y \qquad y$

At least one symbol will increase in ~~not~~ 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:

$a \underbrace{aaa} \quad b \underbrace{bb} \quad b \underbrace{ccc}$
 $v \qquad \qquad \qquad y$

$a \underbrace{aa} \quad a \quad b \underbrace{bb} \quad b \underbrace{ccc}$
 $v \qquad \qquad \qquad y$

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

$a \underbrace{aaa} \quad a \quad b \underbrace{bb} \quad b \underbrace{c} \quad b \underbrace{cc}$
 $v \qquad v \qquad \qquad y \qquad y$

Example 2: copy language

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

Assume it is a CFL.

Show $\sim(\text{PUMPING PROPERTY})$

Let p be the pumping length.

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

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

[To show $\exists s$, just provide an example; just give ~~a~~ ^{an} s that exists]

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

Look at all ways to divide s into parts.

[Show $\nexists uvxyz = s$]

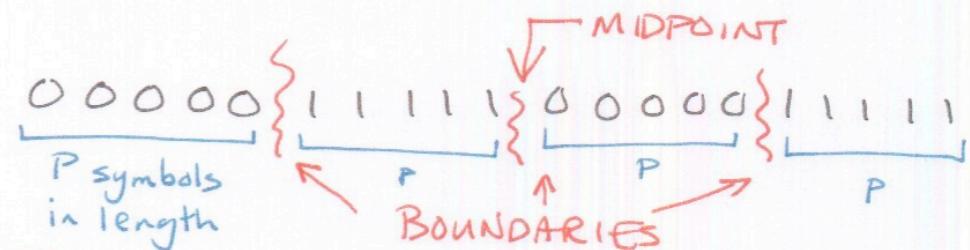
Must show that some condition is not satisfied.

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".

00000 11... 11 00000 11111
P MORE THAN P P P

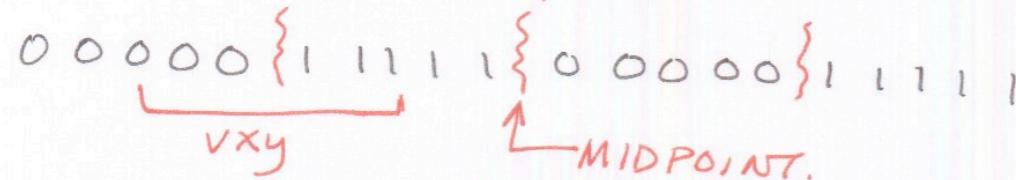
THE STRING NOW HAS THE FORM
 $\overbrace{0-----}^{\text{NEW MIDPOINT.}} \overbrace{1-----}^{\text{NEW MIDPOINT.}}$

This string is not of the form ww .

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

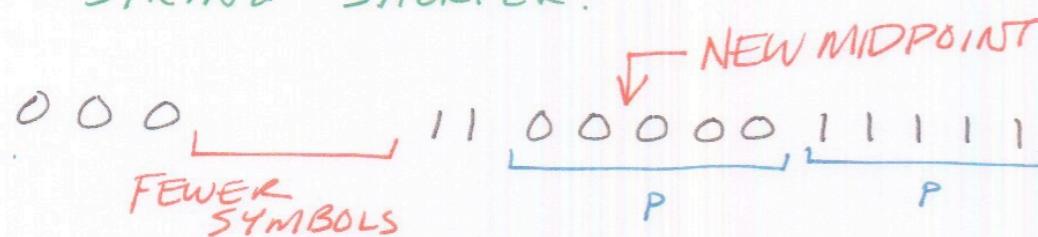
Example 2: copy language

CASE 2: vxy straddles the first boundary.

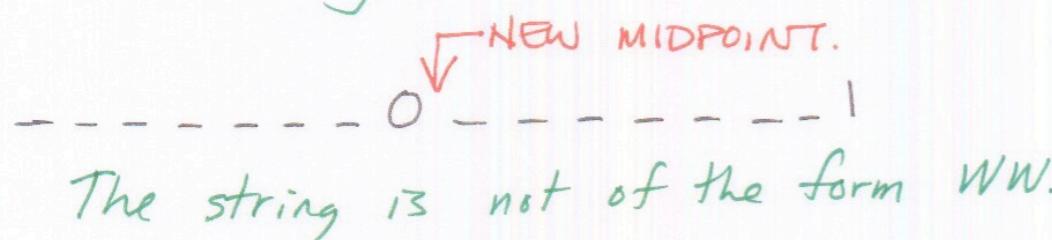


SINCE $|vxy| \leq p$ it cannot straddle the midpoint.

PUMPING DOWN WILL MAKE THE STRING SHORTER.

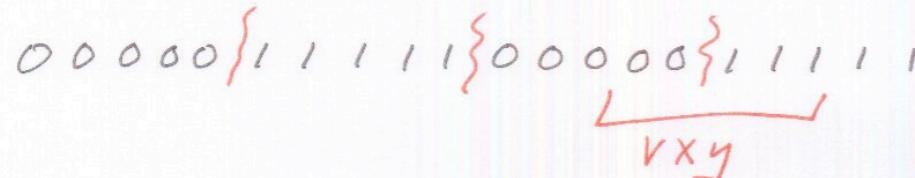


NOTE: The string now has the form:



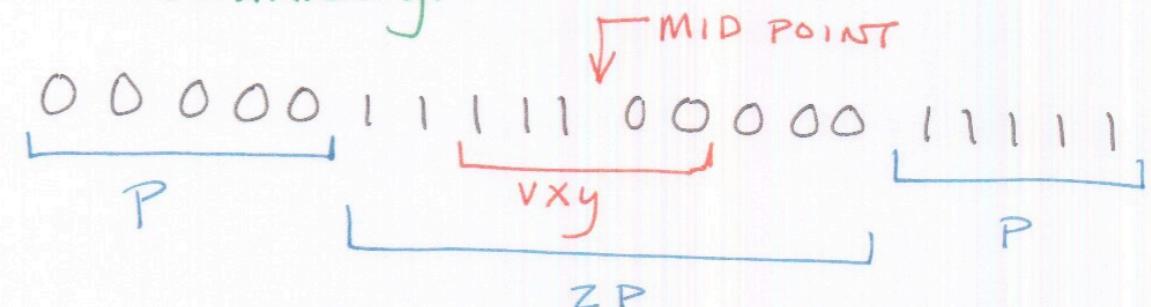
The string is not of the form WW .

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

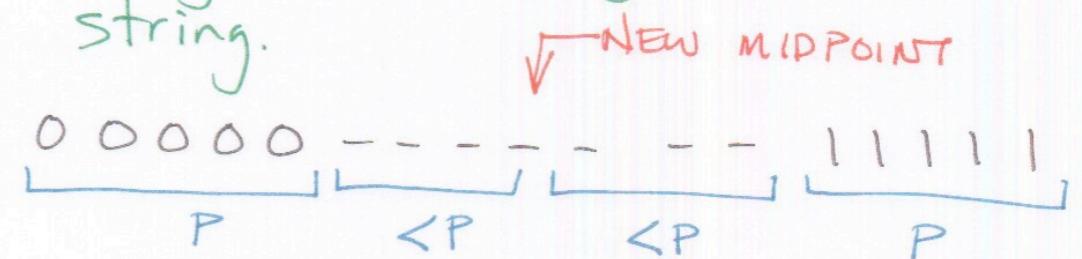


CASE 3: vxy straddles the midpoint.

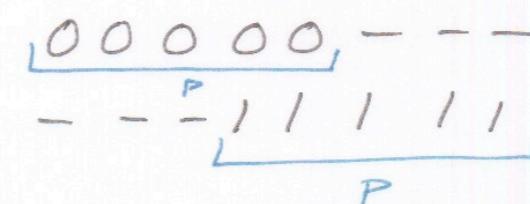
Since $|vxy| \leq p$, it cannot also straddle the first or third boundary.



Pumping down will give us a shorter string.



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



They cannot be equal.
This string fails condition 1.