Introduction to the Theory of Computation

Set 6 — Context-Free Languages

Context-Free Languages

The shortcoming of finite automata is that each state has very limited meaning

- **FA have no memory of where they've been – only knowledge of where they are**
- **Example: $\{0^n1^n \mid n \geq 0\}$**

Context-free grammars are a more powerful method of describing languages

Example Grammar

Grammars use substitution to maintain knowledge

$\Sigma = \{(,)\}$

$S \rightarrow (S)$

$S \rightarrow SS$

$S \rightarrow ()$

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

All possible legal parenthesis pairings can be expressed by consecutive applications of these rules

Is this a regular language?

Example Context Free Grammar

$$S \rightarrow (\,S\,)\ |\ SS\ |\ (\,)$$

(()())(())

$S \rightarrow SS$

$\rightarrow (S)S$

$\rightarrow (S)(S)$

$\rightarrow (SS)(S)$

$\rightarrow (SS)(())$

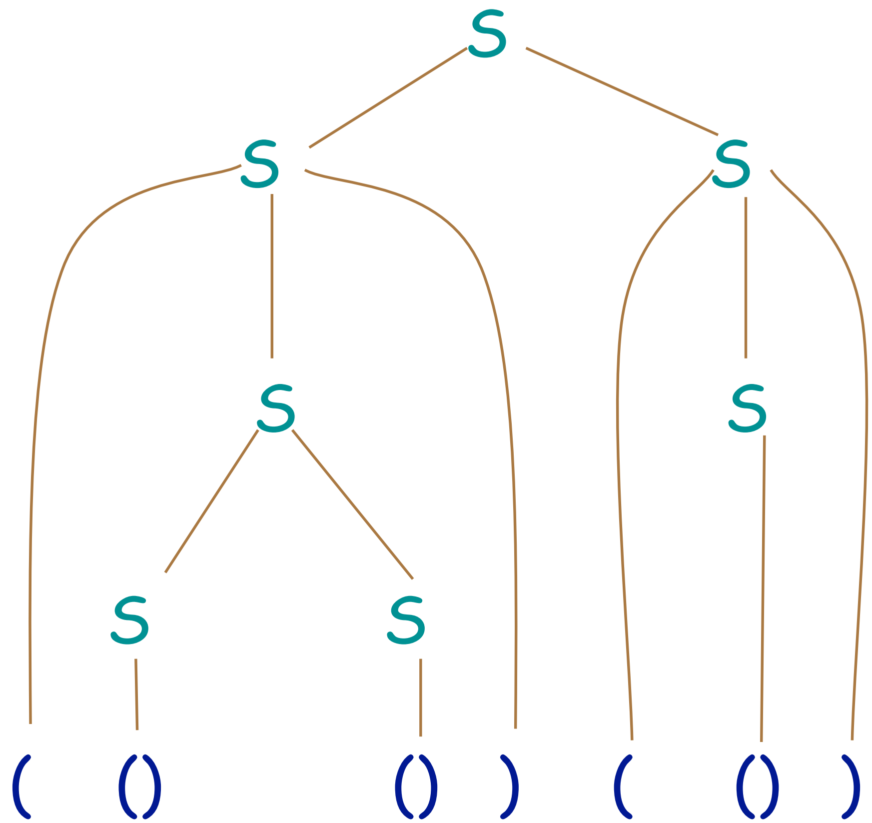
$\rightarrow (()S)(())$

$\rightarrow (()())(())$

The sequence of substitutions is called a derivation

Example CFG Parse Tree

S → (S) | SS | ()



Example 2

S → Sb | Bb

B → aBb | aCb

C → ε

Derivation for aaabbbbb

S → Sb

→ Bbb

→ aBbbb

→ aaBbbbb

→ aaaCbbbbb

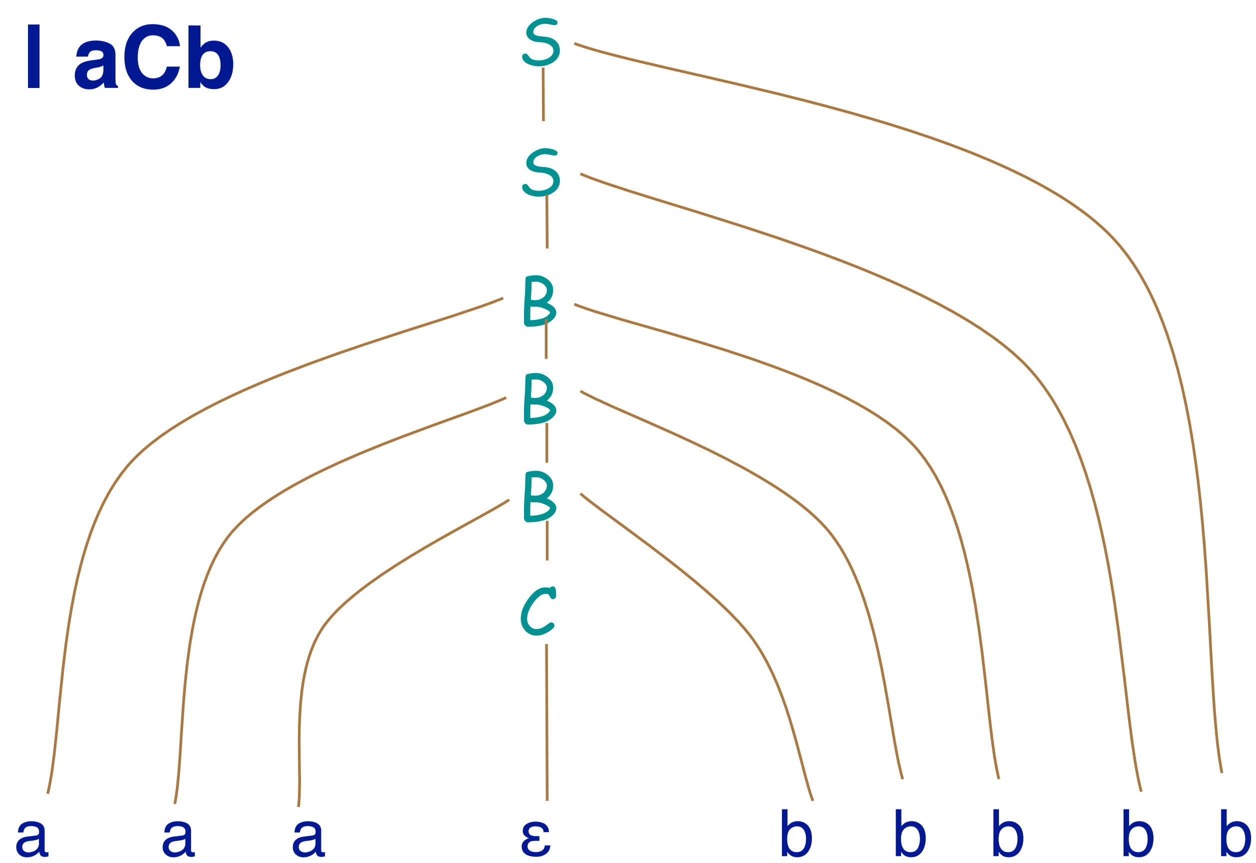
→ aaaεbbbbb = aaabbbbb

Example 2 Parse Tree

S → Sb | Bb

B → aBb | aCb

C → ε



aaabbbbb

Example 2

$S \rightarrow Sb \mid Bb$

$B \rightarrow aBb \mid aCb$

$C \rightarrow \varepsilon$

What language does this grammar accept?

$\{a^n b^m \mid m > n > 0\}$

Can this CFG be simplified?

Yes.

Replace $B \rightarrow aCb$ with $B \rightarrow ab$ and remove $C \rightarrow \varepsilon$

Context-Free Grammar Definition

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

1. **V is a finite set called the variables**
2. **Σ is a finite set, disjoint from V, called the terminals**
3. **R is a finite set of rules, with each rule being a variable and *a string of variables and terminals***
4. **$S \in V$ is the start variable**

$(A,w) \equiv A \rightarrow w$

Definitions

If u, v, and x are strings of variables and terminals, and $A \rightarrow x$ is a rule of the grammar, we say uAv yields uxv

Denoted $uAv \Rightarrow uxv$

If a sequence of rules leads from u to v, $u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \ldots \Rightarrow v$, we denote this

$$u \stackrel{*}{\Rightarrow} v$$

The language of the grammar is

$$\{w \in \Sigma^* \mid S \stackrel{*}{\Rightarrow} w\}$$

Example CFG

$A \rightarrow Ab \mid Bb$

$B \rightarrow aBb \mid ab$

$V = \{A,B\}$

$\Sigma = \{a,b\}$

R is the set of rules listed above

$S = A$

The language of this grammar is

$\{w \in \{a,b\}^* \mid w = a^n b^m,\ m > n > 0\}$

Designing CFG’s

Requires creativity

There are some guidelines to help

- Union of two CFG’s
- Converting a DFA to a CFG
- Linked terminals
- Recursive behavior

Designing the Union of CFGs

For the union of k CFGs, design each CFG separately with starting variables $S_1, S_2, \ldots, S_k$ and combine using the rule

$$S \rightarrow S_1 \mid S_2 \mid \ldots \mid S_k$$

What is a CFG for the following language?

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

$$\{a^i b^j c^k \mid i,j,k \geq 0 \text{ and } i = j\} \cup \{a^i b^j c^k \mid i,j,k \geq 0 \text{ and } j = k\}$$

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

Example

First design $\{a^i b^j c^k \mid i,j,k \geq 0 \text{ and } i = j\}$

Then design $\{a^i b^j c^k \mid i,j,k \geq 0 \text{ and } j = k\}$

Finally, add the "unifying" rule

Converting DFA's into CFG's

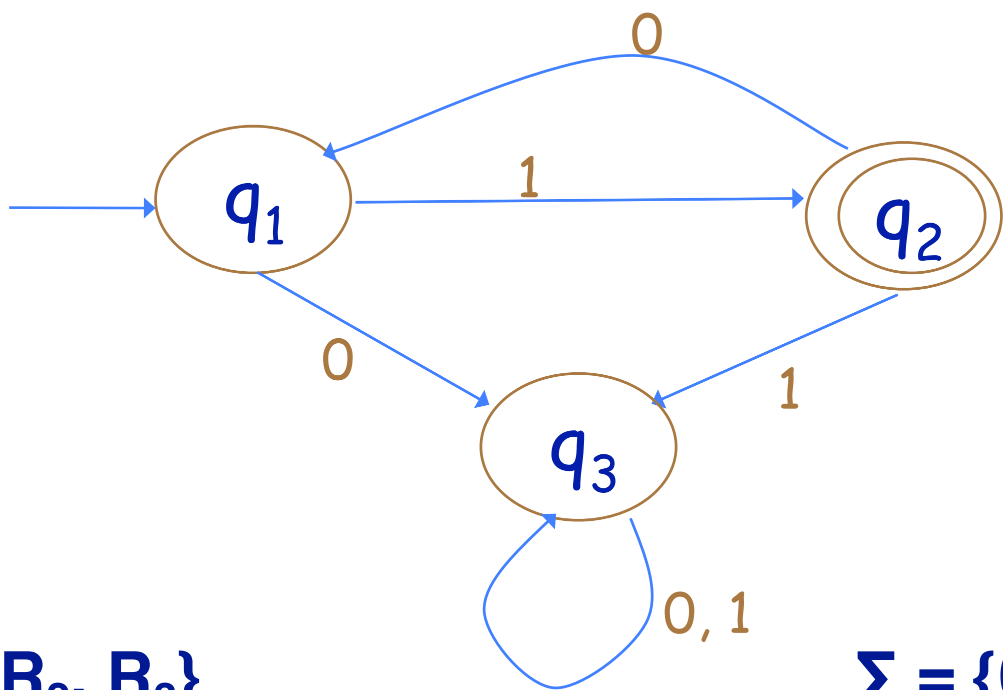
For each state q_i in the DFA,
make a variable R_i for the CFG.

For each transition rule $\delta(q_i,a)=q_k$ in the DFA,
add the rule $R_i \rightarrow aR_k$ to the CFG

For each accept state q_a in the DFA,
add the rule $R_a \rightarrow \varepsilon$

If q_0 is the start state in the DFA,
then R_0 is the starting variable in the CFG

DFA to CFG Example



$V = \{R_1, R_2, R_3\}$ $\Sigma = \{0,1\}$

$R_1 \rightarrow 0R_3 \mid 1R_2$ $R_2 \rightarrow 0R_1 \mid 1R_3$ $R_3 \rightarrow 0R_3 \mid 1R_3$

$R_2 \rightarrow \varepsilon$

R_1 is the start symbol

Linked Terminals

Terminals may be "linked" to one another in that they have the same (or related) number of occurrences

$\{0^n1^n \mid n \geq 0\}$

$\{x^ny^{2n} \mid n > 0\}$

Add terminals simultaneously

$S \rightarrow 0S1 \mid \varepsilon$

$S \rightarrow xSyy \mid xyy$

Recursive Behavior

Some languages may be built of pieces that are within the language

For example, legal pairing of parentheses

For these languages, you will want a recursive rule

For example, $S \rightarrow SS$

Not all recursive rules will be that easy!

Example of Recursive Rules

Construct a CFG accepting all strings in $\{0,1\}^*$ that have equal numbers of 0's and 1's

$$S \rightarrow S0S1S \mid S1S0S \mid \varepsilon$$

$$S \rightarrow A0A1A \mid A1A0A \mid \varepsilon$$
$$A \rightarrow S1S0S \mid S0S1S \mid \varepsilon$$

"*mutual recursion*"

Consider the CFG $(\{S\},\{0,1,+,\times\},R,S)$, where the rules of R are

$$S \rightarrow 0 \mid 1 \mid S + S \mid S \times S$$

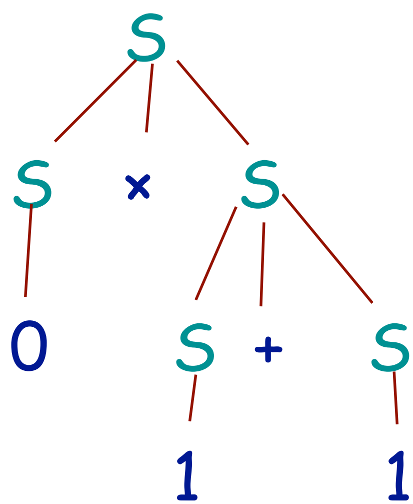
Derive the string $0 \times 1 + 1$

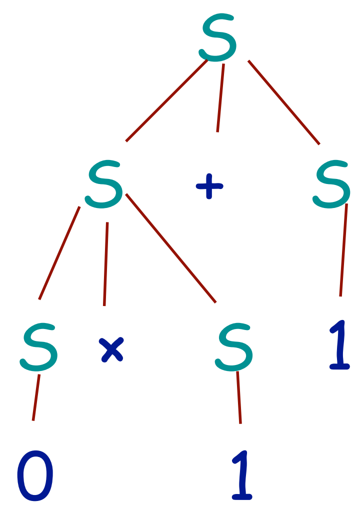
Draw the associated parse tree

Ambiguity

$$S \rightarrow 0 \mid 1 \mid S + S \mid S \times S$$

$$0 \times 1 + 1$$





Different parse trees!

$(0x(1+1)) = 0$ $((0x1)+1) = 1$

Definition of Ambiguity

Ambiguity exists when a context-free grammar G generates a string w and there are two *different* parse trees that generate w

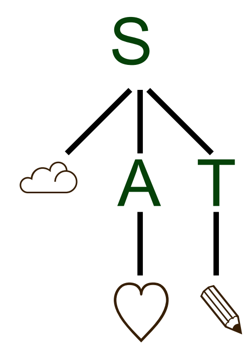
- **Different derivations that differ only in order do not indicate ambiguity**

({A,S,T}, {♡,✎,☁}, {S→☁AT, A→♡, T→✎,}, S)

Derivations of ☁♡✎

S→☁AT →☁♡T →☁♡✎	S→☁AT →☁A✎ →☁♡✎

Parse Tree



Derivation & Ambiguity

A derivation of a string w in a grammar G is a leftmost derivation if every step of the derivation replaced the leftmost variable

A string is derived ambiguously in CFG G if it has two or more different <u>leftmost</u> derivations

<u>leftmost</u>	¬leftmost
S→☁AT	S→☁AT
→☁♡T	→☁A✎
→☁♡✎	→☁♡✎

Derivation & Ambiguity

A derivation of a string w in a grammar G is a leftmost derivation if every step of the derivation replaced the leftmost variable

A string is derived ambiguously in CFG G if it has two or more different <u>leftmost</u> derivations

The grammar G is ambiguous if it generates some string ambiguously

- **Some grammars are inherently ambiguous**

Chomsky Normal Form

Method of simplifying a CFG

Definition: **A context-free grammar is in Chomsky normal form if *every* rule is of one of the following forms**

$$A \rightarrow BC$$

$$A \rightarrow a$$

where a is any terminal, A is *any* variable, and B and C are any variables *other than* the start variable.

If S is the start variable then the rule $S \rightarrow \varepsilon$ is the *only* permitted ε rule

(Note that some CNF formalisms allow B & C to be terminals or variables.)

CFG and Chomsky Normal Form

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

Proof idea: Convert any CFG to one in Chomsky normal form by removing or replacing all rules in the wrong form

1. Add a new start symbol
2. Eliminate ε rules of the form $A \rightarrow \varepsilon$
3. Eliminate unit rules of the form $A \rightarrow B$
4. Convert remaining rules into proper form

Convert a CFG to Chomsky Normal Form

1. Add a new start symbol

☞ Create the following new rule

$$S_0 \rightarrow S$$

where S is the start symbol and S_0 is not used in the CFG

Convert a CFG to Chomsky Normal Form

2. Eliminate all ε rules A → ε, where A is <u>not</u> the start variable

☞ For each rule with an occurrence of A on the right-hand side, add a new rule with the A deleted

R → uAv becomes R → uAv | uv

R → uAvAw becomes R → uAvAw | uvAw | uAvw | uvw

☞ If we have R → A, add R → ε unless we had already removed R → ε

Convert a CFG to Chomsky Normal Form

3. Eliminate all unit rules of the form A → B

☞ For each rule B → *u*, add a new rule A → *u*, where *u* is a string of terminals and variables, unless this rule had already been removed

☞ Repeat until all unit rules have been replaced

Convert a CFG to Chomsky Normal Form

4. Convert remaining rules into proper form

What's left?

☞ **Replace each rule $A \rightarrow u_1u_2\ldots u_k$, where $k \geq 3$ and u_i is a variable or a terminal, with $k-1$ rules**

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

Convert a CFG to Chomsky Normal Form

4. Convert remaining rules into proper form

What's left?

☞ **The formalism requires B and C to be variables in A → BC, so must move all terminals to unit productions**

For every terminal on the right of a nonunit production, add a substitute variable

A → bC *becomes* **A → BC & B → b**

Example

$S \rightarrow S_1 \mid S_2$

$S_1 \rightarrow S_1 b \mid Xb$

$X \rightarrow aXb \mid ab \mid \varepsilon$

$S_2 \rightarrow S_2 a \mid Ya$

$Y \rightarrow bYa \mid ba \mid \varepsilon$

Step 1: **Add a new start symbol**

Example

$S_0 \rightarrow S$

$S \rightarrow S_1 \mid S_2$

$S_1 \rightarrow S_1b \mid Xb$

$X \rightarrow aXb \mid ab \mid \varepsilon$

$S_2 \rightarrow S_2a \mid Ya$

$Y \rightarrow bYa \mid ba \mid \varepsilon$

Step 2: **Eliminate ε rules**

Example

$S_0 \rightarrow S$

$S \rightarrow S_1 \mid S_2$

$S_1 \rightarrow S_1b \mid Xb \mid b$

$X \rightarrow aXb \mid ab$

$S_2 \rightarrow S_2a \mid Ya \mid a$

$Y \rightarrow bYa \mid ba$

Step 3: Eliminate all unit variable rules

Example

$S_0 \rightarrow S_1b \mid Xb \mid b \mid S_2a \mid Ya \mid a$

$S \rightarrow S_1b \mid Xb \mid b \mid S_2a \mid Ya \mid a$

$S_1 \rightarrow S_1b \mid Xb \mid b$

$X \rightarrow aXb \mid ab$

$S_2 \rightarrow S_2a \mid Ya \mid a$

$Y \rightarrow bYa \mid ba$

Step 4: Convert remaining rules to proper form

Example

$S_0 \rightarrow S_1B \mid XB \mid b \mid S_2A \mid YA \mid a$

$S \rightarrow S_1B \mid XB \mid b \mid S_2A \mid YA \mid a$

$S_1 \rightarrow S_1B \mid XB \mid b$

$X \rightarrow AX_1 \mid AB$

$X_1 \rightarrow XB$

$S_2 \rightarrow S_2A \mid YA \mid a$

$Y \rightarrow BY_1 \mid BA$

$Y_1 \rightarrow YA$

$A \rightarrow a \quad B \rightarrow b$

PushDown Automata (PDA)

Similar to finite automata, but for CFL's

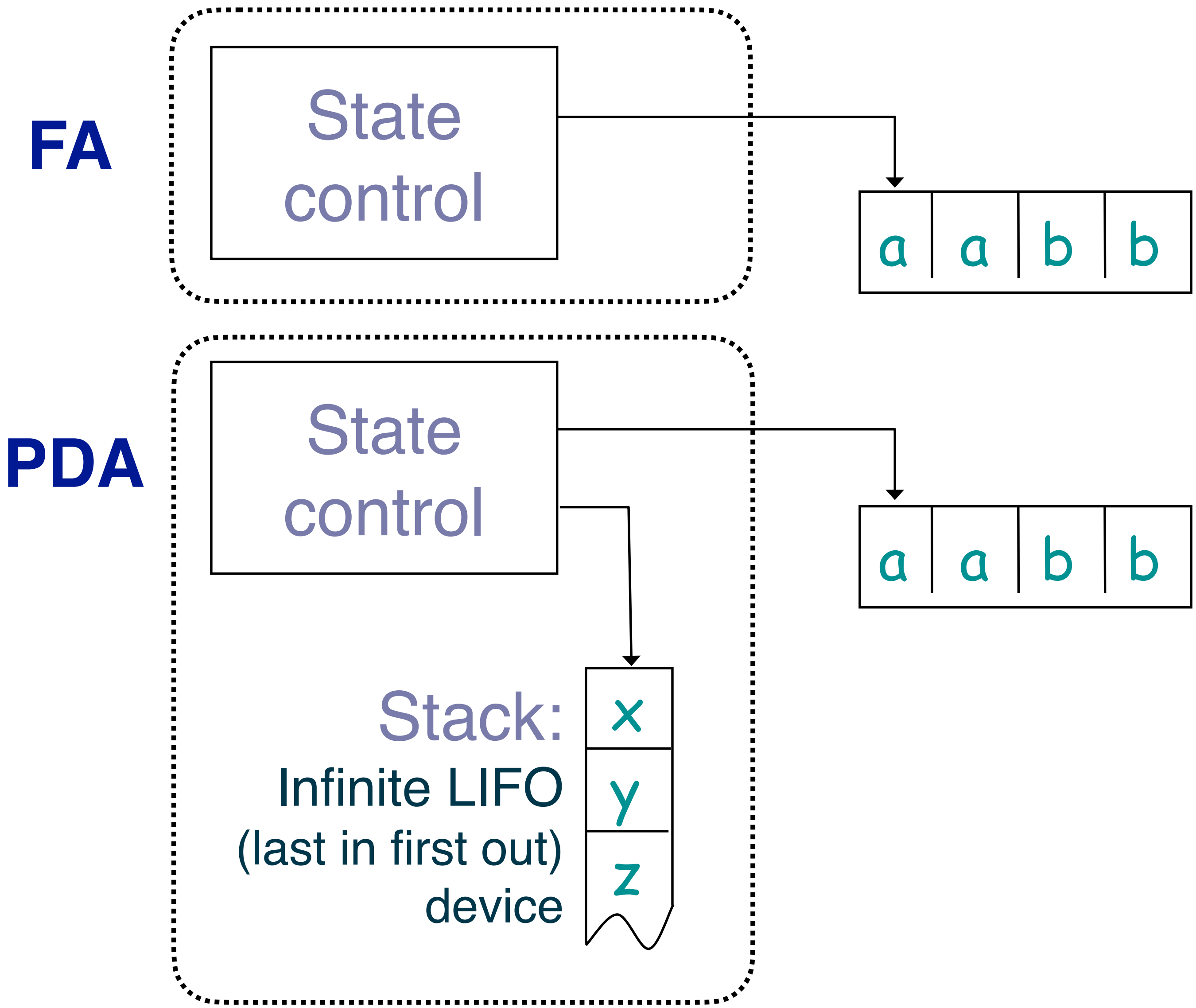
Finite automata are not adequate for CFL's because they cannot keep track of what what's previously been done

- At any point, we only know the current state, not previous states

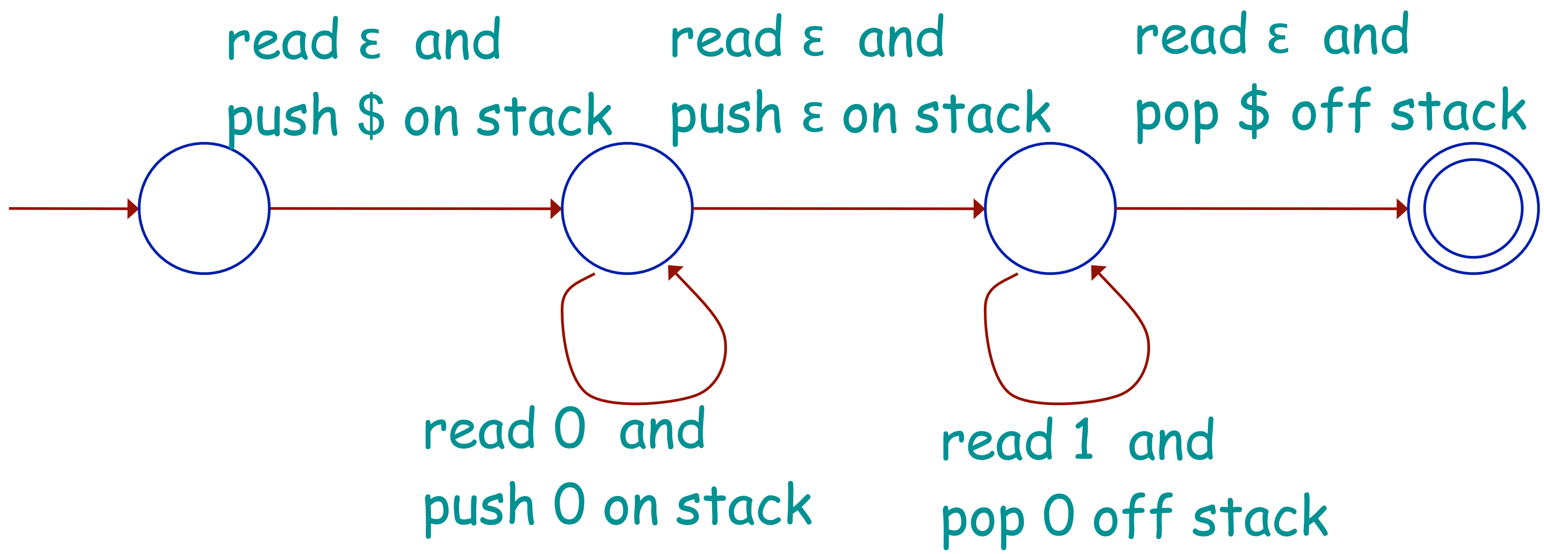
Need memory

- PDA are finite automata with a stack

Finite Automata and PDA Schematics



Example



Language accepted: $\{0^n1^n \mid n \geq 0\}$

Differences Between PDA's and NFA's

Transitions read a symbol of the string and push a symbol onto or pop a symbol off of the stack

Stack alphabet is not necessarily the same as the alphabet for the language

- e.g., $ marks bottom of stack in previous (0^n1^n) example

Definition of Pushdown Automaton

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

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

Strings Accepted by a PDA

Let *w* be a string in Σ^* and M be a PDA.

w is in L(M) $\Leftrightarrow$ w can be written $w = w_1 w_2 \ldots w_n$, where each $w_i \in \Sigma_\varepsilon$, and there exist $r_0, r_1, \ldots, r_n \in Q$ and $s_0, s_1, \ldots, s_n \in \Gamma^*$ satisfying the following:

- $r_0 = q_0$ and $\underline{s_0 = \varepsilon}$

  M starts in the start state with an empty stack

- $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$, where $\underline{s_i = at}$ and $\underline{s_{i+1} = bt}$ for some $a, b \in \Gamma_\varepsilon$ and $t \in \Gamma^*$

  M moves according to transition rules for the state, input, and stack

- $r_n \in F$

  Accept state occurs at input end

The Transition Rule

$(r_{i+1},b)\in\delta(r_i,w_{i+1},a)$, where $s_i = at$ and $s_{i+1} = bt$ for some $a,b \in\Gamma_\varepsilon$ and $t\in\Gamma^*$

The top symbol is

- Pushed if $a=\varepsilon$ and $b\neq\varepsilon$
- Popped if $a\neq\varepsilon$ and $b=\varepsilon$
- Changed if $a\neq\varepsilon$ and $b\neq\varepsilon$
- Unchanged if $a=\varepsilon$ and $b=\varepsilon$

Symbols below the top of the stack may be considered, but not changed

That is *t*'s role

Example

Find δ for the PDA that accepts all strings in $\{0,1\}^*$ with the same number of 0's and 1's

- Need to keep track of "equilibrium point" so use a $ on the stack
- If stack top is not $, it contains the symbol currently dominating in the string

Example

Find δ for the PDA that accepts all strings in $\{0,1\}^*$ with the same number of 0's and 1's

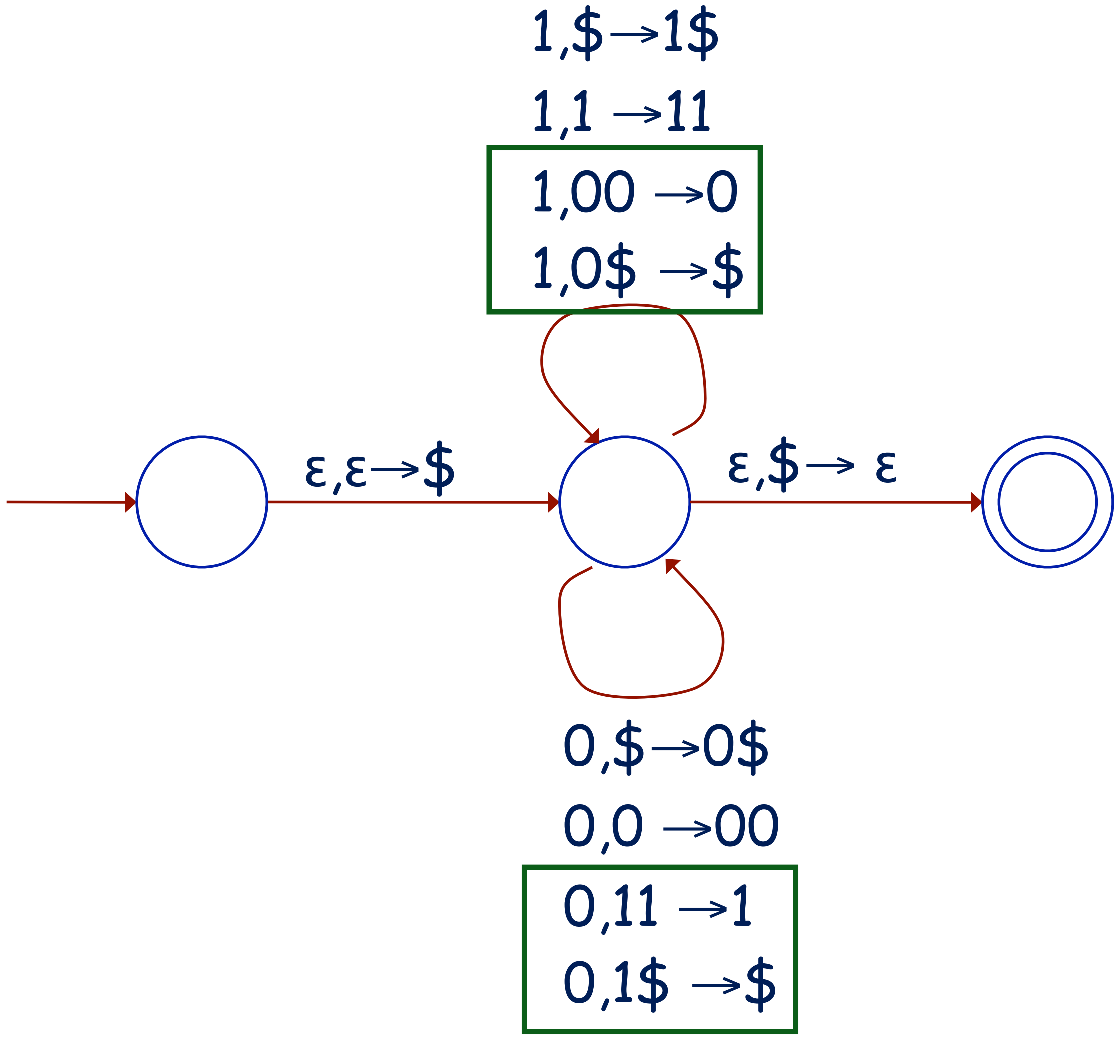
- Push a symbol on the stack as it is read if

  It matches the top of the stack, or

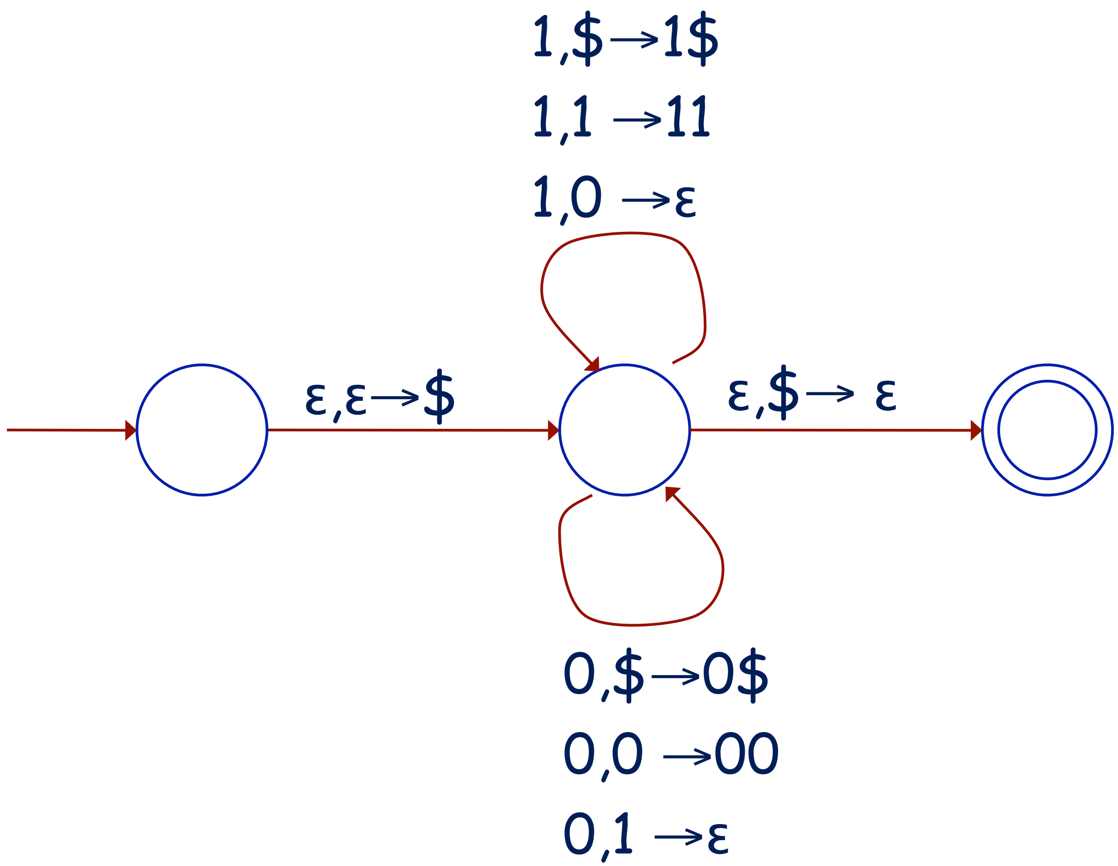
  The top of stack is $

- Pop the symbol off the top of the stack if it reads a 0 and the top of stack is 1 or it reads a 1 and the top of stack is 0.

Example

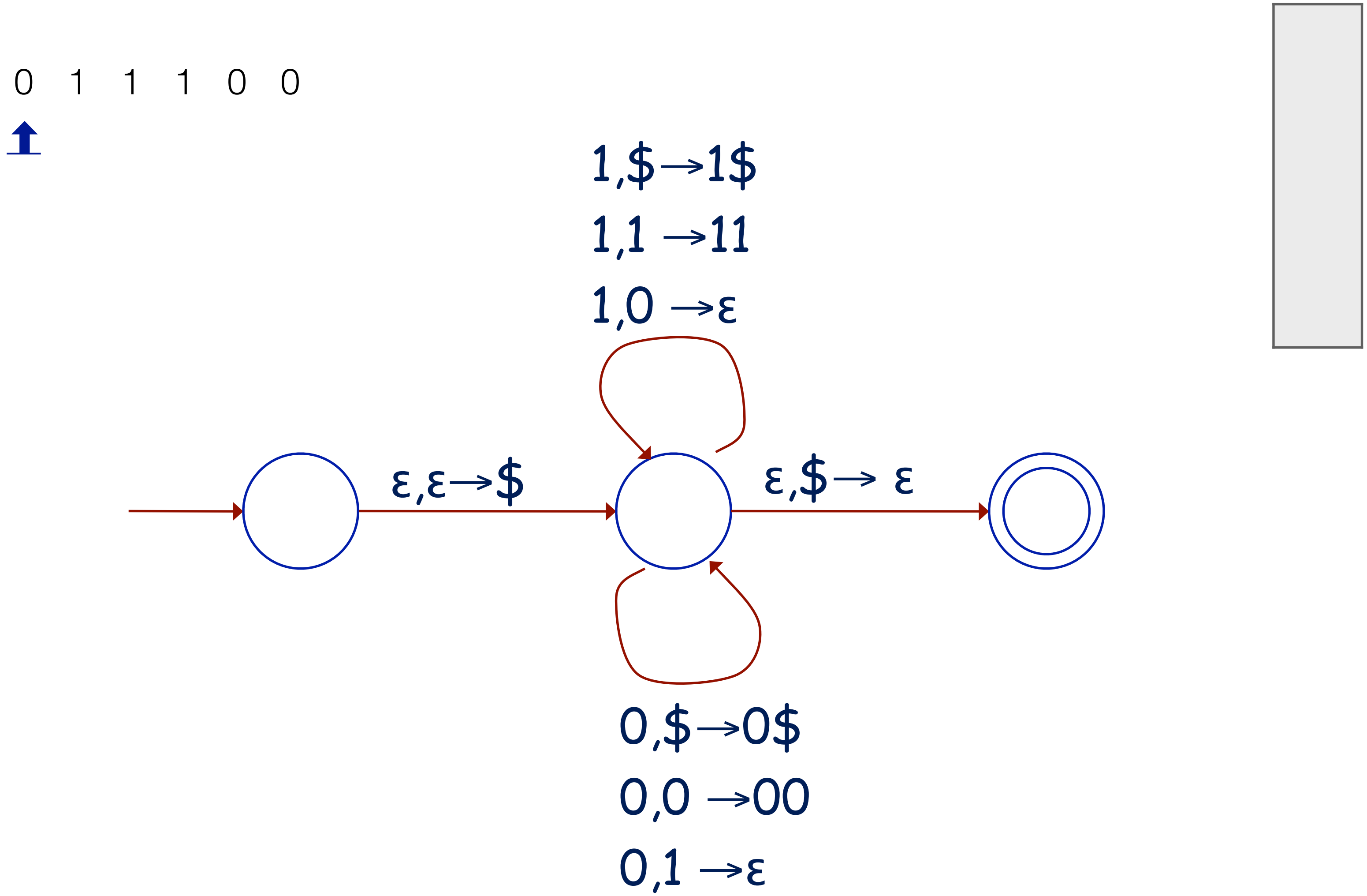


Example



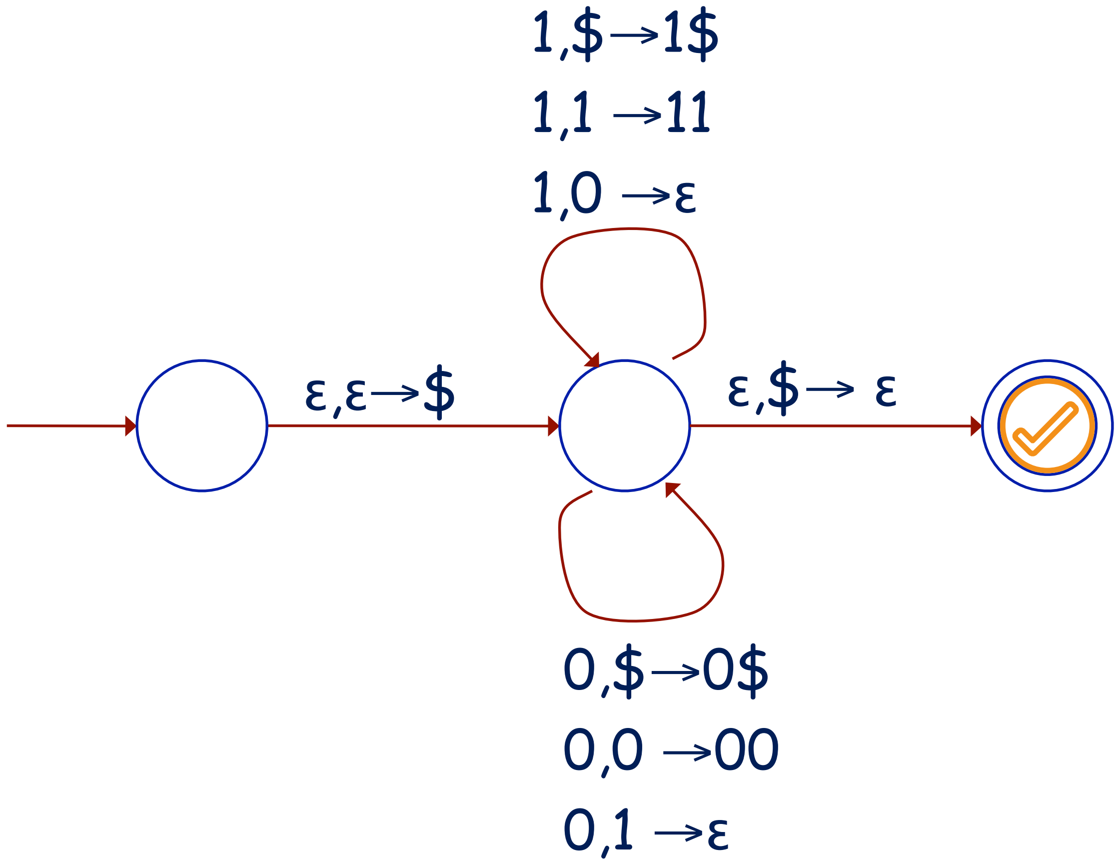
This PDA is equivalent to the one on the previous slide

Example



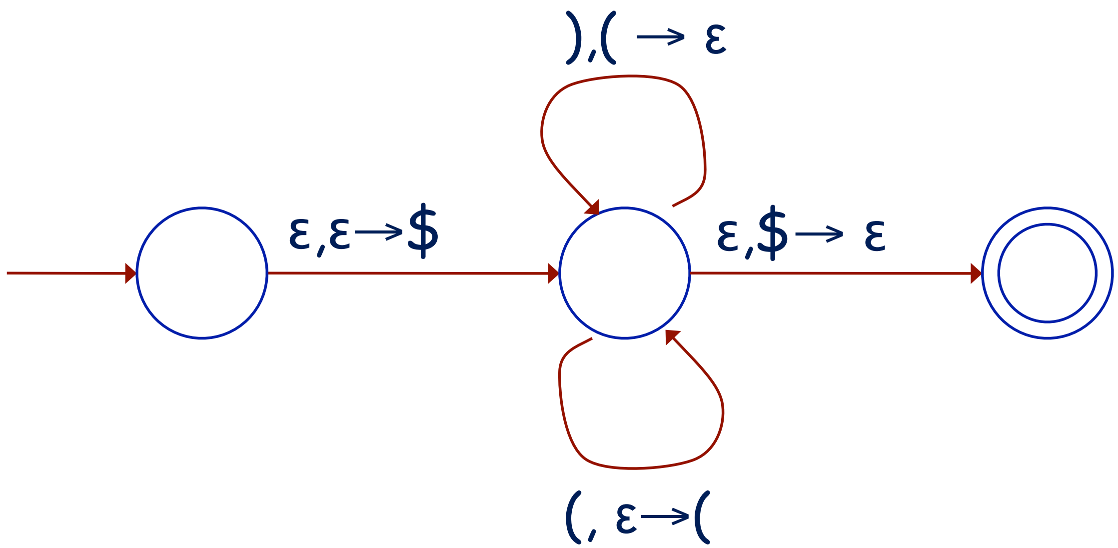
Example

0 1 1 1 0 0 ✔



Example

Nested parentheses



Equivalence of PDAs and CFLs

Theorem: A language is context free if and only if some pushdown automaton recognizes it

Proved in two lemmas –
one for the "if" direction and
one for the "only if" direction

CFLs Are Recognized by PDAs

Lemma: **If a language is context free, then some pushdown automaton recognizes it**

Proof idea:

Construct a PDA following CFG rules

Constructing the PDA

You can read any symbol in Σ when that symbol is at the top of the stack

- Transitions of the form **a,a$\rightarrow\varepsilon$**

The rules indicate what is pushed onto the stack: when a variable A is on top of the stack and there is a rule A$\rightarrow$w, you pop A and push w

You go to the accept state only if the stack is empty

Informal Description of the PDA

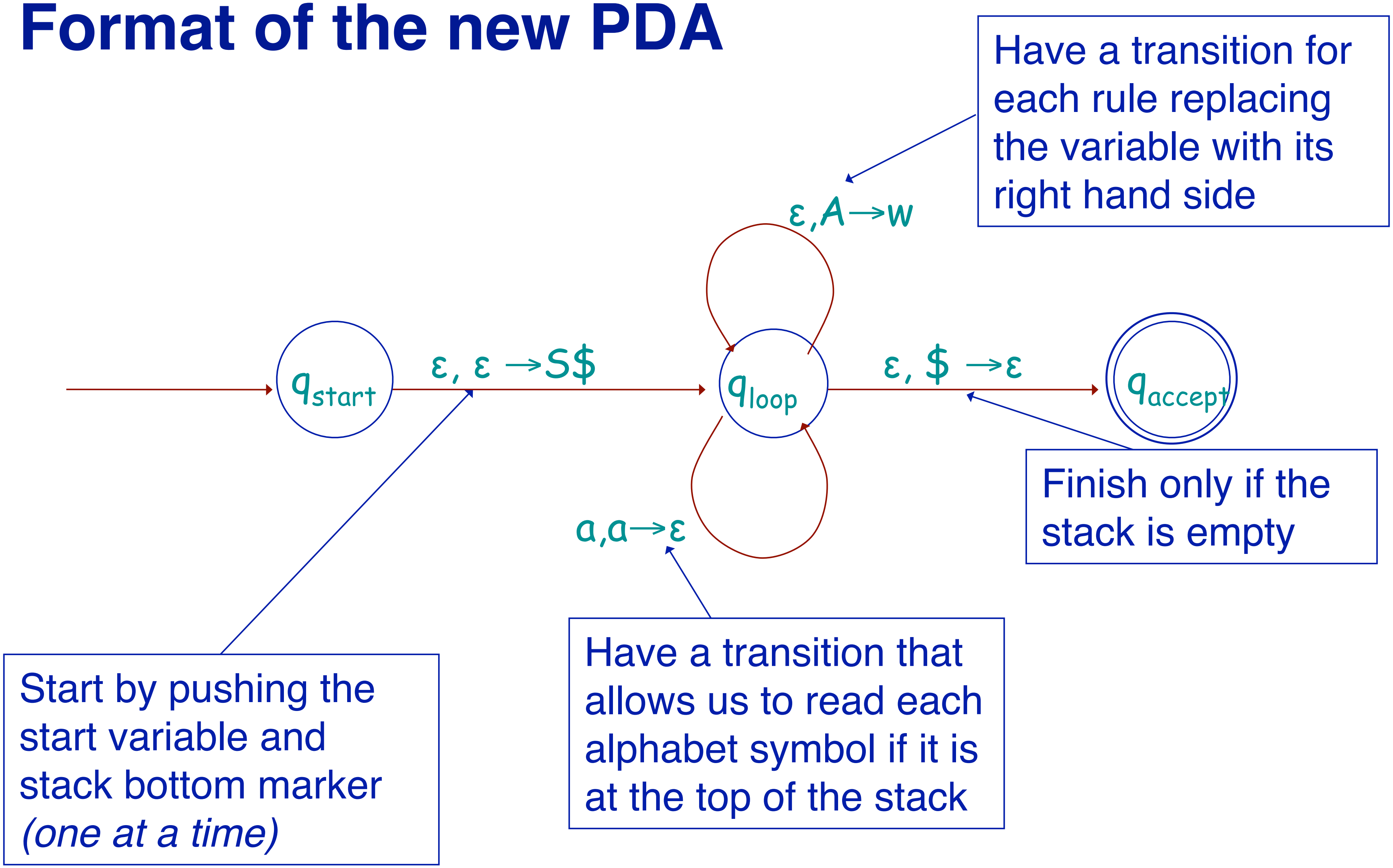
Place $ and start variable on stack

Repeat forever…

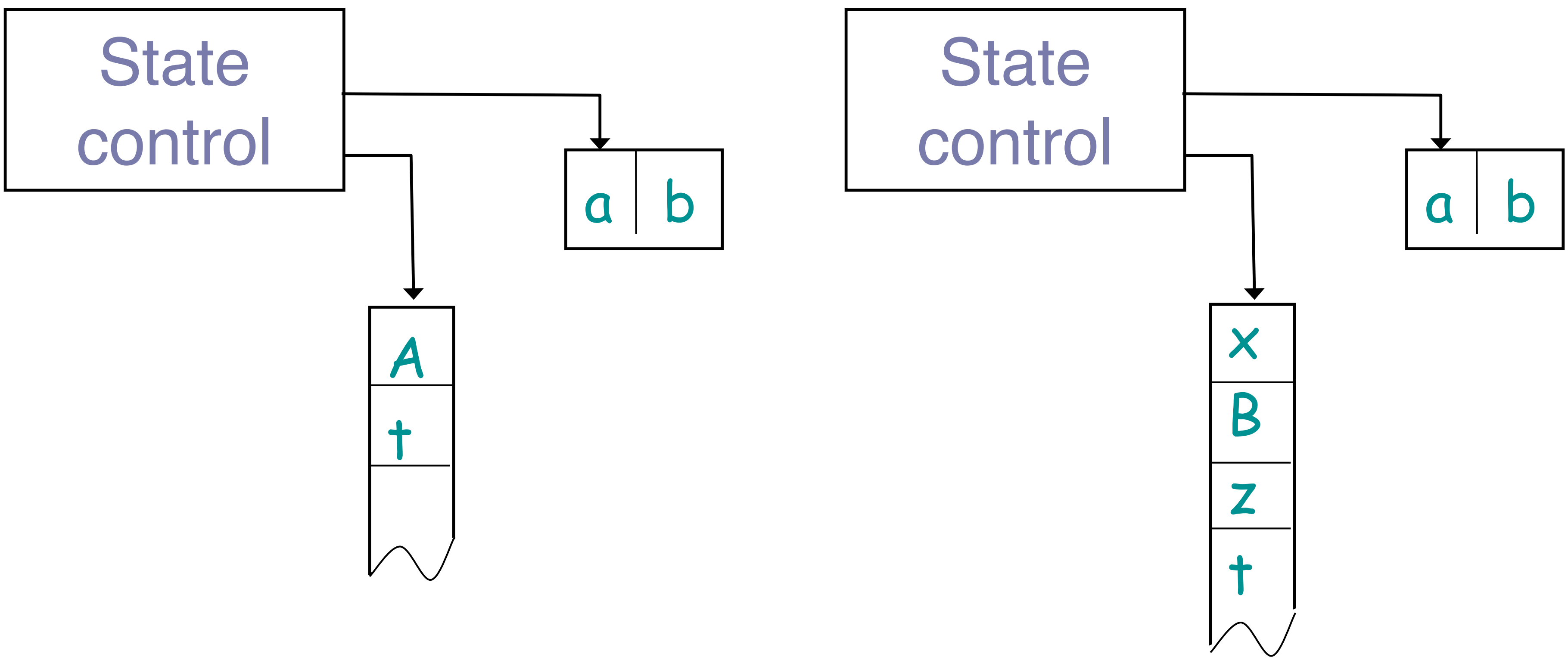
1. If stack top is variable A, nondeterministically select an A rule and substitute the string on the RHS for A
2. If stack top is terminal a, read next symbol from input and compare to a. If match, repeat. If no match, reject this branch.
3. If stack top is $, enter accept state. Accept input if no more input remains.

CFG's are recognized by PDA's

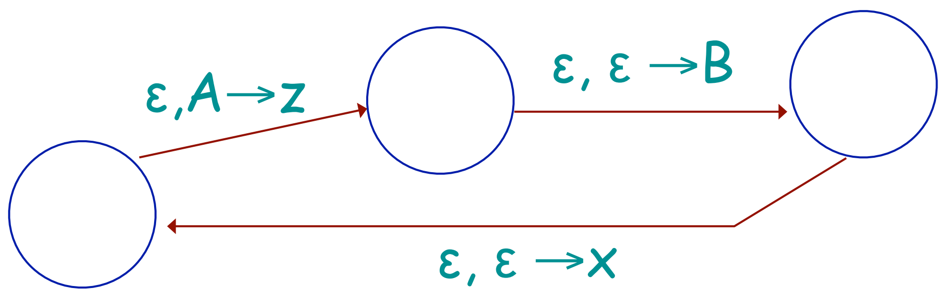
Format of the new PDA



Idea of PDA construction for A→xBz



Actual construction for A→xBz



Notationally, we say δ**(q,ε,A)=(q,xBz)**

Constructing the PDA

$Q = \{q_{start}, q_{loop}, q_{accept}\} \cup E$, where E is the set of states used for replacement rules onto the stack

Σ (the PDA alphabet) is the set of terminals in the CFG

Γ (the stack alphabet) is the <u>union</u> of the *terminals* and the *variables* and {$} (or some other suitable *placeholder*)

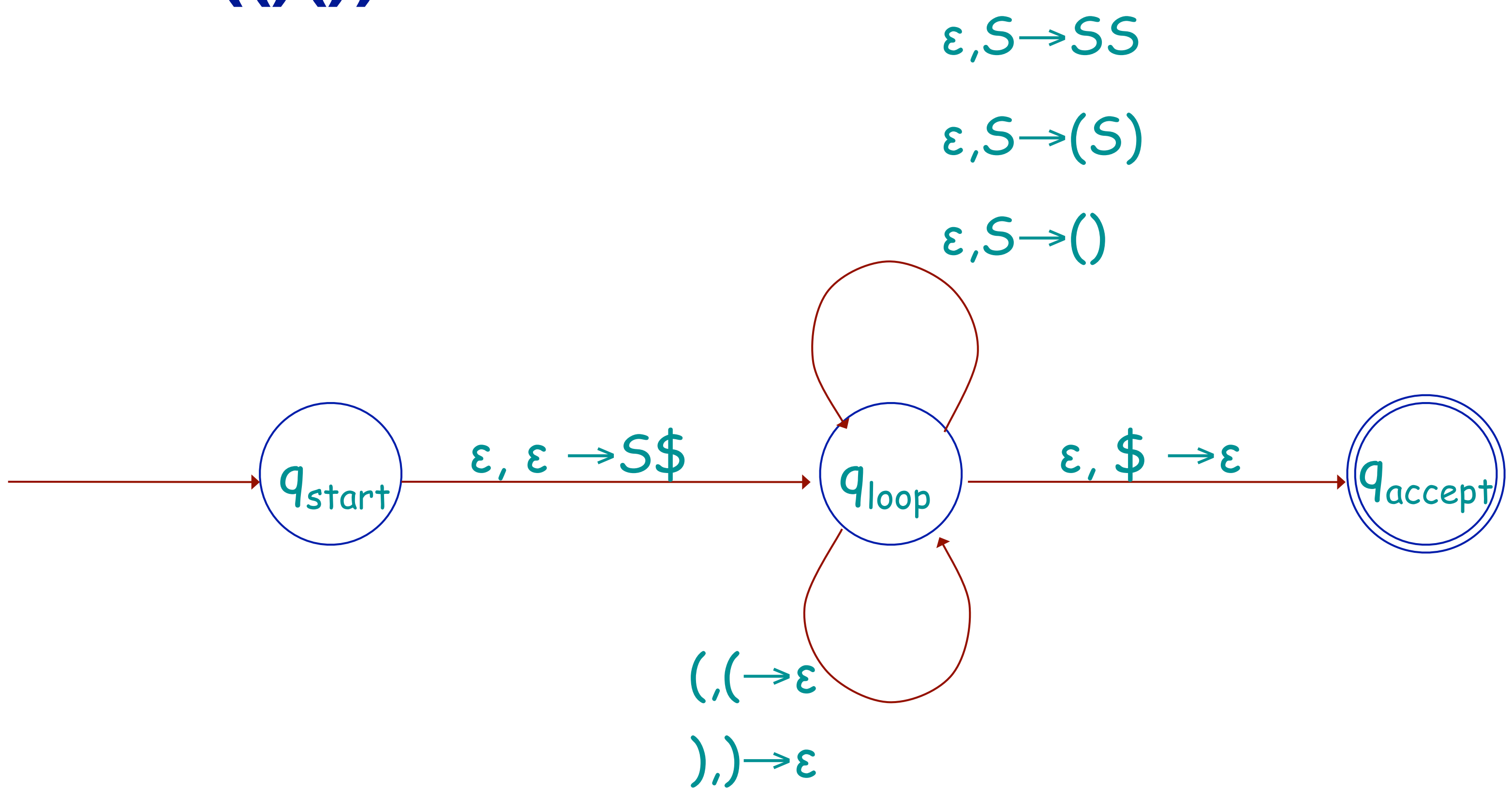
Constructing the PDA

- δ is comprised of several rules
  - $\delta(q_{start},\varepsilon,\varepsilon)=(q_{loop},S\$)$
    - Start with placeholder on the stack and with the start variable
  - $\delta(q_{loop},a,a)=(q_{loop},\varepsilon)$ for every $a\in\Sigma$
    - Terminals may be read off the top of the stack
  - $\delta(q_{loop},\varepsilon,A)=(q_{loop},w)$ for every rule $A\rightarrow w$
    - Implement replacement rules
  - $\delta(q_{loop},\varepsilon,\$)=(q_{accept},\varepsilon)$
    - Accept when the stack is empty

Example

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

Read (()())



Recap

Finite automata (both deterministic and nondeterministic) accept regular languages

- **Weakness:** no memory

Pushdown automata accept context-free languages

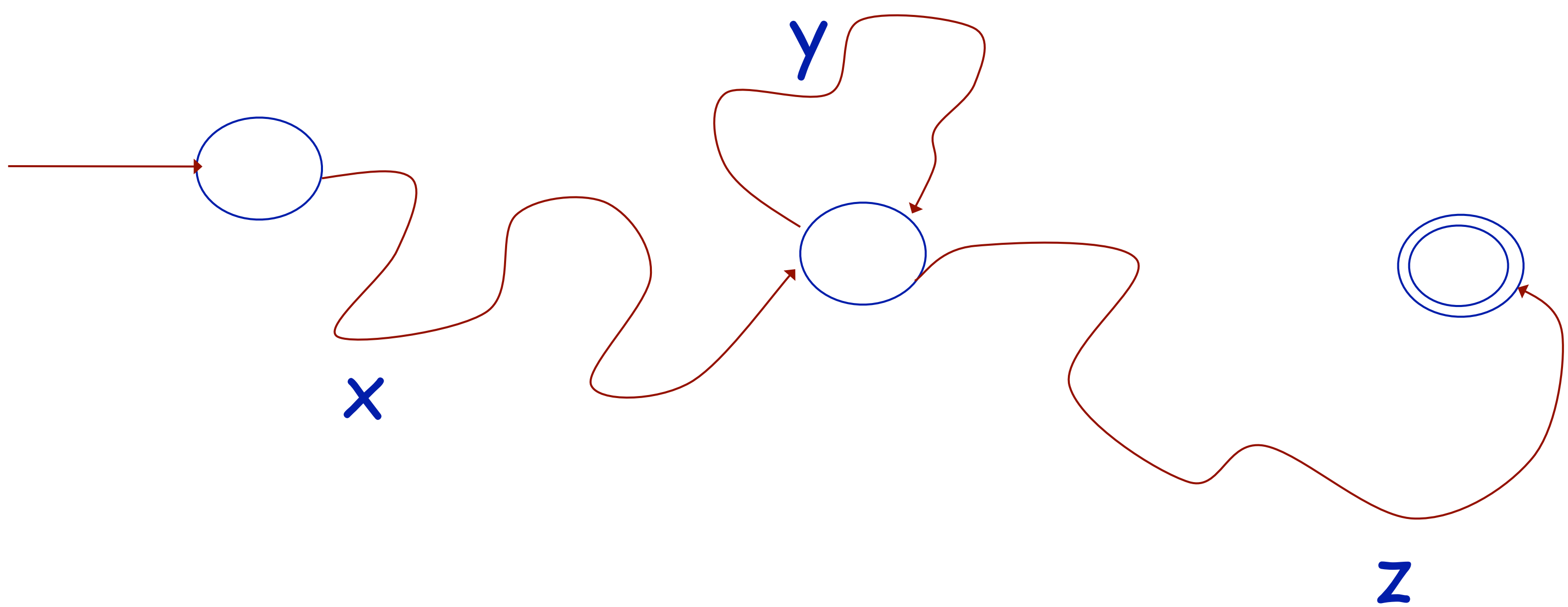
- Add memory in the form of a stack
  - **Potential Weakness:** stack is restrictive

HOW CAN WE TELL THAT A LANGUAGE IS NOT CF?

The pumping lemma for *regular* languages

The pumping lemma for *regular* languages depends on the structure of the *DFA* and the fact that a *state* must be *revisited*

- Only a finite number of states

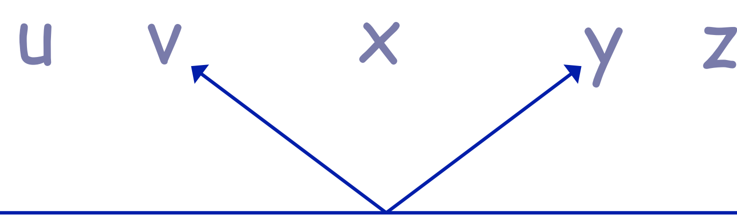


The pumping lemma for CFG's

What might be repeated in a CFG?

- **The variables**

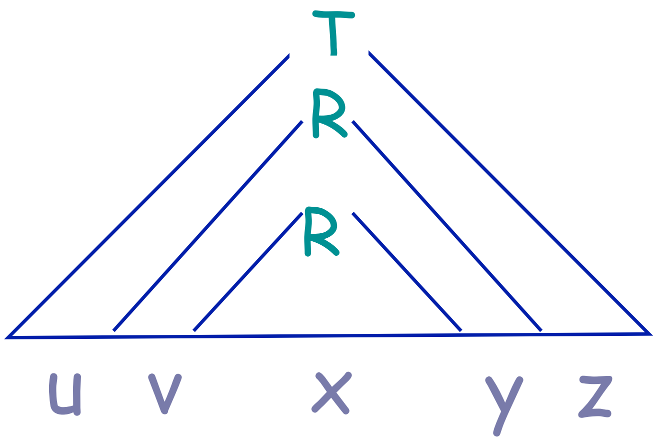
T → uRz
R → vRy | x

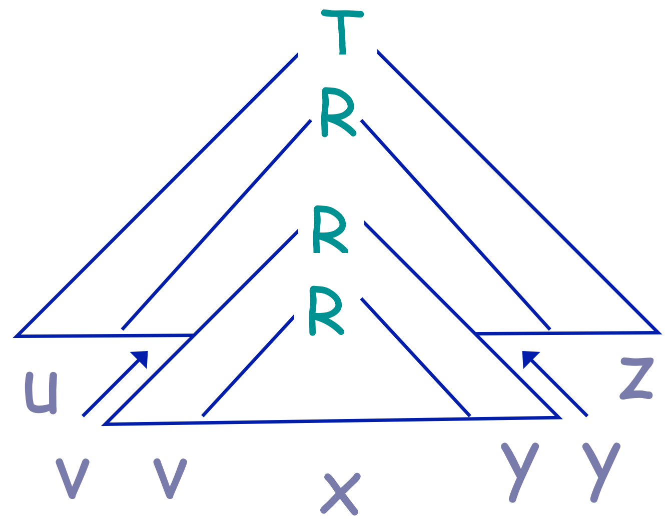


v & y will be repeated simultaneously

The pumping lemma for CFG's

T → uRz
R → vRy | x

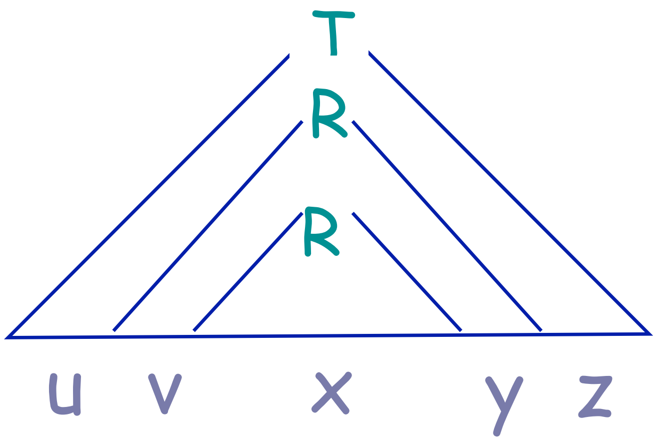


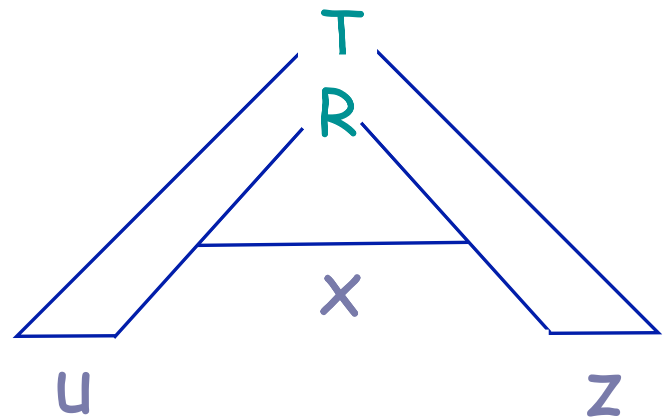


The pumping lemma for CFG's

T → uRz

R → vRy | x





The pumping lemma for CFL's

Theorem: If A is a context-free language, then there is a number p (the pumping length) where, if *s* is any string in A of length at least p, then *s* may be divided into five pieces *s=uvxyz* satisfying the conditions:

1. For each $i \geq 0$, $uv^ixy^iz \in A$
2. $|vy| > 0$
3. $|vxy| \leq p$

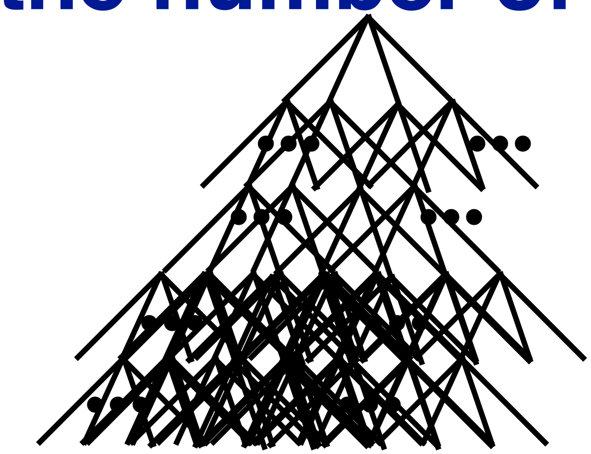
Finding the pumping length of a CFL

Let b equal the longest right-hand side of any rule (assume b > 1)

- Each node in the parse tree has at most b children
- At most b^h nodes are h steps from the start node

Let p equal $b^{|V|+2}$, where |V| is the number of variables

- Tree height is at least |V|+2



Example

Show A is not context free, where

$A = \{a^n \mid n \text{ is prime}\}$

Proof:

Assume A is context-free and let p be the pumping length of A.

Let $w = a^n$ for any $n \geq p$.

By the pumping lemma, $w = uvxyz$ such that $|vxy| \leq p$, $|vy| > 0$, and $uv^ixy^iz \in A$ for all $i = 0, 1, 2, \ldots$

Example (cont.)

Show A is not context free, where

$A = \{a^n \mid n \text{ is prime}\}$

Clearly, *vy*=a^k for some k

Consider the string $uv^{n+1}xy^{n+1}z$

This string adds n copies of a^k to *w*

– i.e., this is a^{n+nk}

Since the exponent is n(1+k), the length of the string is not prime, thus the string is not in A, which contradicts the pumping lemma. Therefore, A is not context free.

Closure Properties of CFLs

If A and B are context free languages then:

A^R is a context-free language ✔

A^* is a context-free language ✔

$A \cup B$ is a context-free language ✔

Is $\bar{A}$ *(complement)* a context-free language?

Is $A \cap B$ a context-free language?

Closure Properties of CFLs

If A and B are context free languages then:

Is $A \cap B$ a context-free language?

Consider $A = \{ a^i b^j c^k \mid i = j \}$ and $B = \{ a^i b^j c^k \mid j = k \}$

A: $S_A \rightarrow XC$, $X \rightarrow aXb \mid \varepsilon$, $C \rightarrow cC \mid \varepsilon$

B: $S_B \rightarrow AY$, $A \rightarrow aA \mid \varepsilon$, $Y \rightarrow bYc \mid \varepsilon$

$$A \cap B = \{ a^i b^j c^k \mid i = j = k \}$$

Does this language satisfy the pumping lemma?

$s \in L$, $|s| \geq p \Rightarrow s = uvxyz$, $uv^ixy^iz \in L \ \forall i \geq 0$

$|vy| > 0$

$|vxy| \leq p$

Closure Properties of CFLs

Consider $A = \{ a^i b^j c^k \mid i = j \}$ and $B = \{ a^i b^j c^k \mid j = k \}$

$$A \cap B = \{ a^i b^j c^k \mid i = j = k \}$$

Does this language satisfy the pumping lemma?

$s \in L, |s| \geq p \Rightarrow s = uvxyz$, $uv^ixy^iz \in L \ \forall i \geq 0$
$|vy| > 0$
$|vxy| \leq p$

Try $s = a^p b^p c^p$

$|vy| > 0 \Rightarrow$ vy contains at least one symbol

$|vxy| \leq p \Rightarrow$ vxy contains at most 2 different symbols

$uv^2xy^2z \notin A \cap B$ so $A \cap B$ is not a CFL

Closure Properties of CFLs

If A and B are context free languages then:

A^R is a context-free language ✔

A^* is a context-free language ✔

$A \cup B$ is a context-free language ✔

$\overline{A}$ is *not necessarily* a context-free language

$A \cap B$ is *not necessarily* a context-free language