มี stack

Context-Free Grammar and Pushdown Automata

01204213 Theory of Computation

Jittat Fakcharoenphol

Kasetsart University

July 27, 2021

Outline

Review: An example

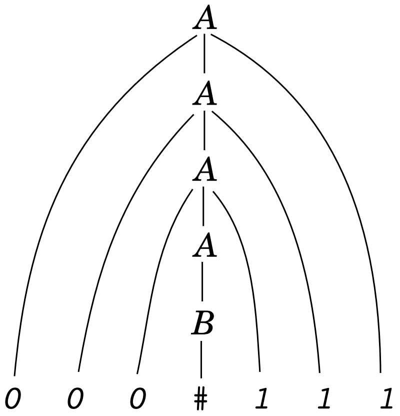
Grammar G_1

$$\begin{array}{rcl} A & \rightarrow & 0A1 \\ A & \rightarrow & B \\ B & \rightarrow & \# \end{array}$$

Start with $A \Rightarrow 0A1$ (rule 1) $\Rightarrow 00A11$ (rule 1) $\Rightarrow 00B11$ (rule 2) $\Rightarrow 00\#11$ (rule 3).

This sequence of substitution is called a **derivation**.

A parse tree



A grammar

From previous example, you may notice that the grammar has

- a set of substitution rules (or production rules),
- variables (symbols appearing on the left-hand side of the arrow), and
- terminals (other symbols).

To obtain a derivation, we also need a start variable. (If not specified otherwise, it is the left-hand side of the top rule.)

Language of the grammar

- A grammar **describes** a language by generating each string of the language.
- For a grammar G, let $L(G)$ denote the language of G.
- $L(G_1) = \{0^n\#1^n | n \geq 0\}$

A context-free language

A language described by some context-free grammar is called a context-free language.

Definition [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**, (alphabet)
3. R is a finite set of **rules**, with each rule being a variable and a string of variables and terminals, and
4. $S \in V$ is the **start variable**.

More definitions

- Let u, v, and w be strings of variables and terminals, and $A \rightarrow w$ be a rule of the grammar.
- We say that uAv **yields** uwv, denoted by $uAv \Rightarrow uwv$.
- We say that u **derives** v, written as $u \overset{*}{\Rightarrow} v$,
  - if $u = v$, or
  - if a sequence $u_1, u_2, \ldots, u_k$ exists for $k \geq 0$ and

$$u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \cdots \Rightarrow u_k \Rightarrow v.$$

Example: G_3

$G_3 = (\{S\}, \{a, b\}, R, S)$, where R is

$$S \rightarrow aSb|SS|\varepsilon.$$

↳ $n(a) = n(b)$ และ a ต้องมาก่อน b

Practice

Find a CFG that describes the following language

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

$S \to S_1 X_1 \mid X_2 S_2$ $X_1 \to \varepsilon \mid c X_1$

$(i=j)$ $S_1 \to a S_1 b \mid \varepsilon$ $X_2 \to \varepsilon \mid a X_2$

$(j=k)$ $S_2 \to b S_2 c \mid \varepsilon$

Example: G_4'

$G_4' = (V, \Sigma, R, EXPR)$, where

- $V = \{EXPR\}$,

Example: G_4'

$G_4' = (V, \Sigma, R, EXPR)$, where

- $V = \{EXPR\}$,
- $\Sigma = \{a, +, \times, (,)\}$,

Example: G_4'

$G_4' = (V, \Sigma, R, EXPR)$, where

- $V = \{EXPR\}$,
- $\Sigma = \{a, +, \times, (,)\}$,
- the rules are

$$EXPR \quad \rightarrow \quad EXPR + EXPR \mid EXPR \times EXPR \mid (EXPR) \mid a$$

Generate some string from G_4'.

$(a + (a \times a) + a)$ $\quad E \Rightarrow (E) \Rightarrow (E+E) \Rightarrow (E+E+E) \Rightarrow \ldots$

Ambiguity

Find a parse tree for $a + a \times a$ in grammar G_4'.

Ambiguity and leftmost derivation

- A grammar generates a string ambiguously when there exist two parse trees for the string. (Not two derivations)

Ambiguity and leftmost derivation

- A grammar generates a string ambiguously when there exist two parse trees for the string. (Not two derivations)
- A derivation of string w in a grammar G is a leftmost derivation if at every step the leftmost remaining variable is the one replaced.

Ambiguity and leftmost derivation

- A grammar generates a string ambiguously when there exist two parse trees for the string. (Not two derivations)
- A derivation of string w in a grammar G is a leftmost derivation if at every step the leftmost remaining variable is the one replaced.

ตีความได้สองแบบ

Definition

A string w is derived ambiguously in context-free grammar G if it has two or more leftmost derivations. Grammar G is ambiguous if it generates some string ambiguously.

Example: G_4

$G_4 = (V, \Sigma, R, EXPR)$, where

- $V = \{EXPR, TERM, FACTOR\}$,

Example: G_4

$G_4 = (V, \Sigma, R, EXPR)$, where

- $V = \{EXPR, TERM, FACTOR\}$,
- $\Sigma = \{a, +, \times, (,)\}$,

Example: G_4

$G_4 = (V, \Sigma, R, EXPR)$, where

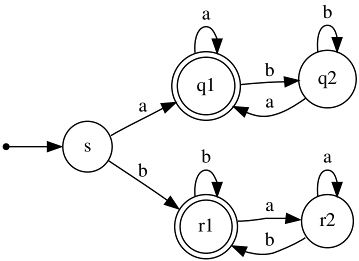
- $V = \{EXPR, TERM, FACTOR\}$,
- $\Sigma = \{a, +, \times, (,)\}$,
- the rules are

$$\begin{aligned} EXPR &\rightarrow EXPR + TERM | TERM \\ TERM &\rightarrow TERM \times FACTOR | FACTOR \\ FACTOR &\rightarrow (EXPR) | a \end{aligned}$$

CFGs and regular languages (1)

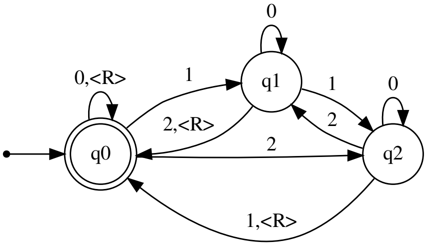
CFG → DFA

Can you find a context-free grammar that describes the language recognized by the following DFA?



CFGs and regular languages (2)

Can you find a context-free grammar that describes the language recognized by the following DFA?



Again, think about a "mechanical" procedure for constructing a CFG.

CFGs and regular languages (3)

Any general procedure?

Simpler forms

- Since we know that DFAs and NFAs are equivalent, DFA = NFA

Simpler forms

- Since we know that DFAs and NFAs are equivalent, we can pick one that allow us to prove the property that we want.

Simpler forms

- Since we know that DFAs and NFAs are equivalent, we can pick one that allow us to prove the property that we want.
- Again, CFGs is quite general and sometimes we want them to be in a simpler form.

Simpler forms

- Since we know that DFAs and NFAs are equivalent, we can pick one that allow us to prove the property that we want.
- Again, CFGs is quite general and sometimes we want them to be in a simpler form.
- One of the forms is called Chomsky normal form.

Noam Chomsky

From wikipedia. URL:
`http://en.wikipedia.org/wiki/Image:Noam_chomsky_cropped.jpg`

Avram Noam Chomsky is an American linguist, philosopher, cognitive scientist, political activist, author, and lecturer. [from wikipedia]

Chomsky normal form

CNF

A context-free grammar is in **Chomsky normal form** is 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,

ตัวสุดท้าย ตัวแปร

Chomsky normal form

CNF

A context-free grammar is in **Chomsky normal form** is 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 cannot be the start variable.

Chomsky normal form

CNF

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

$$A \rightarrow BC$$

$$A \rightarrow a$$

where a is any terminal and A, B, and C are any variables, except that B and C cannot be the start variable.
We also permit the rule $S \rightarrow \varepsilon$, where S is the start variable.

Any CFGs can be converted into CNF

Theorem 1

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

Any CFGs can be converted into CNF

Theorem 1

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

We shall not do the full proof, but will show how to do so by example. (See also Example 2.10 on the book.)

Step 1: The start variable cannot be on the right-hand side

- Suppose that S is the start variable.
- An example of violated rules: $S \to aS$, or $A \to BS$.

Step 1: The start variable cannot be on the right-hand side

- Suppose that S is the start variable.
- An example of violated rules: $S \to aS$, or $A \to BS$.
- We introduce a new start variable S_0 and add rule

$$S_0 \to S$$

Step 2: ε rules

- Sample rules:

$$B \to aAb|bAcA$$

$$A \to c|aA|\varepsilon$$

Step 2: ε rules

- Sample rules:

$$B \to aAb|bAcA$$

$$A \to c|aA|\varepsilon$$

- Remove $A \to \varepsilon$ and on any occurrence of A add new rules where A replaced by ε.
- Resulting rules:

$$B \to aAb \Rightarrow \quad B \to aAb|ab$$

Step 2: ε rules

- Sample rules:

$$B \to aAb|bAcA$$

$$A \to c|aA|\varepsilon$$

- Remove $A \to \varepsilon$ and on any occurrence of A add new rules where A replaced by ε.
- Resulting rules:

ε

$$B \to aAb \Rightarrow B \to aAb|ab$$

Step 2: ε rules

- Sample rules:

$$B \to aAb|bAcA$$

$$A \to c|aA|\varepsilon$$

- Remove $A \to \varepsilon$ and on any occurrence of A add new rules where A replaced by ε.
- Resulting rules:

$$B \to aAb \Rightarrow B \to aAb|ab$$

$$B \to bAcA \Rightarrow$$ B → bcA | bAc | bc | bAcA

Step 2: ε rules

- Sample rules:

$$B \to aAb|bAcA$$

$$A \to c|aA|\varepsilon$$

- Remove $A \to \varepsilon$ and on any occurrence of A add new rules where A replaced by ε.
- Resulting rules:

$$B \to aAb \Rightarrow B \to aAb|ab$$

$$B \to bAcA \Rightarrow B \to bAcA|bcA|bAc|bc$$

Step 2: ε rules

- Sample rules:

$$B \to aAb|bAcA$$

$$A \to c|aA|\varepsilon$$

- Remove $A \to \varepsilon$ and on any occurrence of A add new rules where A replaced by ε.
- Resulting rules:

$$B \to aAb \Rightarrow B \to aAb|ab$$

$$B \to bAcA \Rightarrow B \to bAcA|bcA|bAc|bc \checkmark$$

$$A \to aA \Rightarrow A \to a|aA$$

Step 2: ε rules

- Sample rules:

$$B \rightarrow aAb|bAcA$$

$$A \rightarrow c|aA|\varepsilon$$

- Remove $A \rightarrow \varepsilon$ and on any occurrence of A add new rules where A replaced by ε.
- Resulting rules:

$$B \rightarrow aAb \Rightarrow B \rightarrow aAb|ab$$

$$B \rightarrow bAcA \Rightarrow B \rightarrow bAcA|bcA|bAc|bc$$

$$A \rightarrow aA \Rightarrow A \rightarrow aA|a$$

Step 3: unit rules

- Sample rules:

$$C \to Ba|Ac$$

$$B \to aAb|bAcA$$

$$A \to B|c$$

Step 3: unit rules

- Sample rules:

$$C \to Ba|Ac$$

$$B \to aAb|bAcA$$

$$A \to B|c$$

- Remove $A \to B$ and on any occurrence of A add new rules where A replaced by B.
- Resulting rules:

$$C \to Ba|Ac \Rightarrow$$

Step 3: unit rules

- Sample rules:

$$C \to Ba|Ac$$

$$B \to aAb|bAcA$$

$$A \to B|c$$

- Remove $A \to B$ and on any occurrence of A add new rules where A replaced by B.
- Resulting rules:

$$C \to Ba|Ac \Rightarrow C \to Ba|Ac|Bc$$

Step 3: unit rules

- Sample rules:

$$C \to Ba|Ac$$

$$B \to aAb|bAcA$$

$$A \to B|c$$

- Remove $A \to B$ and on any occurrence of A add new rules where A replaced by B.
- Resulting rules:

$$C \to Ba|Ac \Rightarrow C \to Ba|Ac|Bc$$

$$B \to aAb|bAcA \Rightarrow B \to$$

Step 3: unit rules

- Sample rules:

$$C \to Ba|Ac$$

$$B \to aAb|bAcA$$

$$A \to B|c$$

- Remove $A \to B$ and on any occurrence of A add new rules where A replaced by B.
- Resulting rules:

$$C \to Ba|Ac \Rightarrow C \to Ba|Ac|Bc$$

$$B \to aAb|bAcA \Rightarrow B \to aAb|aBb|$$

Step 3: unit rules

- Sample rules:

$$C \rightarrow Ba|Ac$$

$$B \rightarrow aAb|bAcA$$

$$A \rightarrow B|c$$

- Remove $A \rightarrow B$ and on any occurrence of A add new rules where A replaced by B.
- Resulting rules:

$$C \rightarrow Ba|Ac \Rightarrow C \rightarrow Ba|Ac|Bc$$

$$B \rightarrow aAb|bAcA \Rightarrow B \rightarrow aAb|aBb|bAcA|bBcA|bAcB|bBcB$$

Step 3: unit rules

- Sample rules:

$$C \to Ba|Ac$$

$$B \to aAb|bAcA$$

$$A \to B|c$$

- Remove $A \to B$ and on any occurrence of A add new rules where A replaced by B.
- Resulting rules:

$$C \to Ba|Ac \Rightarrow C \to Ba|Ac|Bc$$

$$B \to aAb|bAcA \Rightarrow B \to aAb|aBb|bAcA|bBcA|bAcB|bBcB$$

$$A \to c$$

Step 4: long rules

- Sample rules:

$$C \to abC | asbdB$$

Step 4: long rules

- Sample rules:

$$C \to abC|asbdB$$

- Split rules into short rules and add more variables to connect them.

Step 4: long rules

- Sample rules:

$$C \rightarrow abC|asbdB$$

- Split rules into short rules and add more variables to connect them.
- Resulting rules:

$$C \rightarrow abC \Rightarrow$$

Step 4: long rules

- Sample rules:

$$C \to abC | asbdB$$

- Split rules into short rules and add more variables to connect them.
- Resulting rules:

$$C \to abC \Rightarrow C \to aC_1, C_1 \to bC$$

Step 4: long rules

- Sample rules:

$$C \to abC|asbdB$$

- Split rules into short rules and add more variables to connect them.
- Resulting rules:

$$C \to abC \Rightarrow C \to aC_1, C_1 \to bC$$

$$C \to asbdB \Rightarrow$$

Step 5: remove rules with terminal

- Sample rules:

$$C \to aC$$

$$D \to ab|a$$

Step 5: remove rules with terminal

- Sample rules:

$$C \to aC$$

$$D \to ab|a$$

- Replace terminals with new variables and add rules that the new variables derive to that terminals.

Step 5: remove rules with terminal

- Sample rules:

$$C \to aC$$

$$D \to ab|a$$

- Replace terminals with new variables and add rules that the new variables derive to that terminals.
- Resulting rules:

$$C \to AC$$

$$A \to a$$

$$D \to AB|a$$

$$B \to b$$

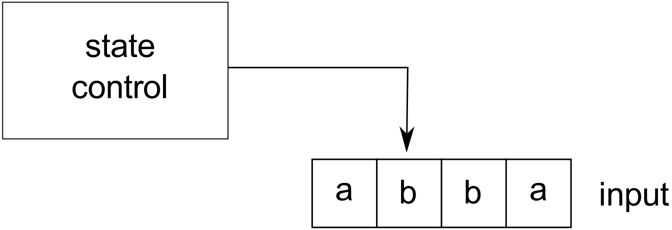
Pushdown automata

- NFAs power-up

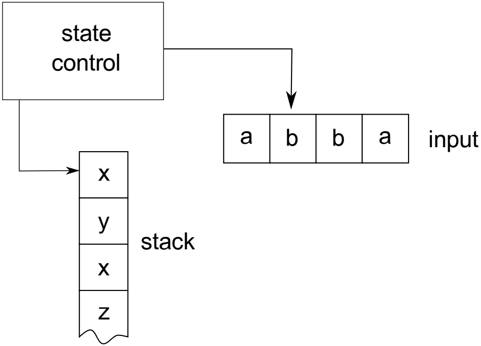
Pushdown automata

- NFAs power-up
- Think of them as NFAs with extra memory, called **stack**.

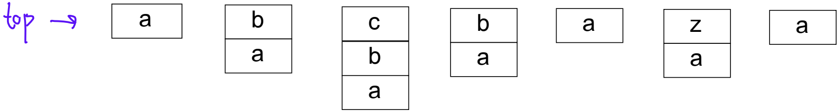
NFAs



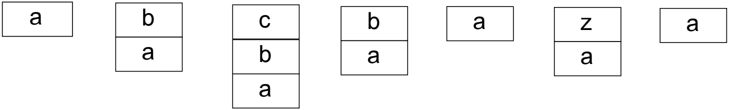
PDAs



Stacks

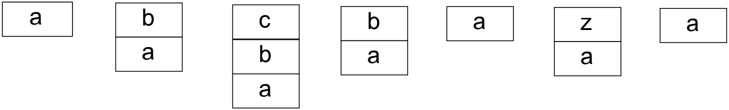


Stacks



A stack is an infinite memory but you can only access the topmost element.

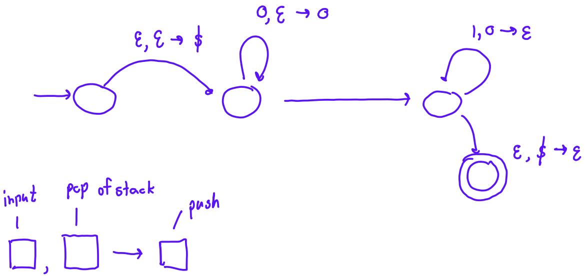
Stacks



A stack is an infinite memory but you can only access the topmost element.
You can pop (put something on top) and push (remove the topmost).

Informally

Can you find an NFA with a stack that recognizes $\{0^n1^n | n \geq 0\}$?



Informally

Can you find an NFA with a stack that recognizes $\{0^n1^n | n \geq 0\}$?

$NFA = (Q, \Sigma, \delta, q_0, F)$

$\delta : Q \times \Sigma_\varepsilon \rightarrow P(Q)$ set of next state

now state

input + ε

Transition function with stack (1)

- A stack keeps some data. Let Γ be a stack alphabet.

Transition function with stack (1)

- A stack keeps some data. Let Γ be a stack alphabet.
- How does a PDA move?

Transition function with stack (1)

แกมมา

- A stack keeps some data. Let Γ be a stack alphabet.
- How does a PDA move?
    - It reads some input (can be ε).

Transition function with stack (1)

- A stack keeps some data. Let Γ be a stack alphabet.
- How does a PDA move?
  - It reads some input (can be ε).
  - It reads the top of the stack (can be ε as well).

Transition function with stack (1)

- A stack keeps some data. Let Γ be a stack alphabet.
- How does a PDA move?
  - It reads some input (can be ε).
  - It reads the top of the stack (can be ε as well).
  - It changes the state and writes something to the top of the stack.
- Thus, the transition function accepts (q, x, s) where q is a state, x is an input symbol, and s is the top of the stack.

Transition function with stack (1)

- A stack keeps some data. Let Γ be a stack alphabet.
- How does a PDA move?
  - It reads some input (can be ε).
  - It reads the top of the stack (can be ε as well).
  - It changes the state and writes something to the top of the stack.
- Thus, the transition function accepts (q, x, s) where q is a state, x is an input symbol, and s is the top of the stack.
- The transition function returns a set of pairs (q', s') where q' is a new state and s' is the stack symbol written to the stack.

Transition function with stack (2)

- Transition function δ:
  - Domain: $Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon$
  - Range: $\mathcal{P}(Q \times \Gamma_\varepsilon)$

Definition [pushdown automaton]

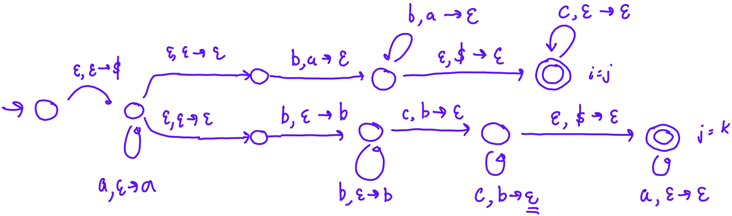
A **pushdown automaton** is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where Q, Σ, Γ, and F are finite sets, and

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

Practice 1

Find a pushdown automaton that recognizes the language

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



Practice 1

Find a pushdown automaton that recognizes the language

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

Test cases:

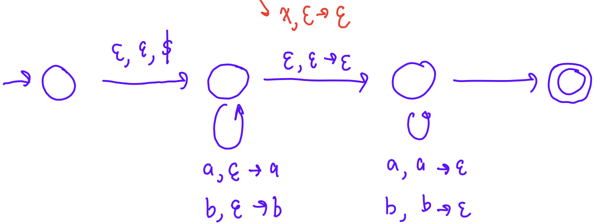
Practice 2

Find a pushdown automaton that recognizes the language

① $\{wxw^R \mid w \in \{0,1\}^*\}$

$$\{ww^{\mathcal{R}} \mid w \in \{0,1\}^*\}$$

in $\in \{0,1\}$



Practice 2

Find a pushdown automaton that recognizes the language

$$\{ww^{\mathcal{R}} \mid w \in \{0,1\}^*\}$$

Test cases:

Context-free languages

context-free language

regular language

CFL

A language described by some context-free grammar is called a **context-free language**.

language

$01(10)^* \cup \Sigma^*$

$S \to 01A \mid B$

$A \to 10A \mid \varepsilon$

$B \to 0B \mid 1B \mid \varepsilon$

Equivalence

$\longleftrightarrow$

Theorem 2

A language is context-free if and only if some pushdown automaton recognizes it.

Again, two directions to prove the equivalence

- **Only-if:** If a language is context-free, it is recognized by some pushdown automaton.

Again, two directions to prove the equivalence

- **Only-if:** If a language is context-free, it is recognized by some pushdown automaton.
  - Given a CFG G, construct a PDA P that recognizes the language generated by G.

Again, two directions to prove the equivalence

- **Only-if:** If a language is context-free, it is recognized by some pushdown automaton.
  - Given a CFG G, construct a PDA P that recognizes the language generated by G.
- **If:** A language is context-free if it is recognized by some pushdown automaton.

Again, two directions to prove the equivalence

- **Only-if:** If a language is context-free, it is recognized by some pushdown automaton.
  - Given a CFG G, construct a PDA P that recognizes the language generated by G.
- **If:** A language is context-free if it is recognized by some pushdown automaton.
  - Given a PDA P, construct a CFG G that generates a language recognized by P.

Again, two directions to prove the equivalence

- **Only-if:** If a language is context-free, it is recognized by some pushdown automaton.
  - Given a CFG G, construct a PDA P that recognizes the language generated by G.
- **If:** A language is context-free if it is recognized by some pushdown automaton.
  - Given a PDA P, construct a CFG G that generates a language recognized by P. SKIPPED.

Plan for today

Today we'll cover only the only-if part, i.e., given a CFL described by CFG G, we'll construct a PDA P that recognizes G.

Any CFLs can be recognized by PDAs

- Take an example CFG G:

$$S \to AB$$

$$A \to aAb|\varepsilon$$

$$B \to cB|c$$

- How can we recognize string generated by G?

Any CFLs can be recognized by PDAs

- Take an example CFG G:

$$S \rightarrow AB$$

$$A \rightarrow aAb|\varepsilon$$

$$B \rightarrow cB|c$$

- How can we recognize string generated by G?
- Consider aabbccc.

Generating: aabbccc

Maybe we can try to generate it using a PDA:

CFG G

$$S \to AB$$

$$A \to aAb|\varepsilon$$

$$B \to cB|c$$

and aabbccc.

Generating: aabbccc

CFG G

$$S \to AB$$

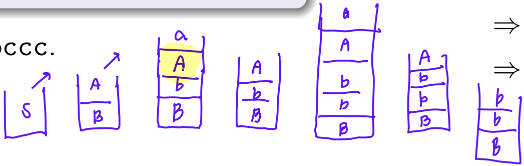
$$A \to aAb|\varepsilon$$

$$B \to cB|c$$

and aabbccc.

Maybe we can try to generate it using a PDA:

$$\begin{aligned} S &\Rightarrow AB \\ &\Rightarrow \mathrm{aAb}B \\ &\Rightarrow aaAbbB \\ &\Rightarrow aa\varepsilon bbB \\ &\Rightarrow aabbc\mathrm{B} \\ &\Rightarrow aabbcc\mathrm{B} \\ &\Rightarrow aabbccc \end{aligned}$$



Generating a string

So, we want to generate a string using a PDA.

- How can we generate the correct derivation?

Generating a string

So, we want to generate a string using a PDA.

- How can we generate the correct derivation?
    - We guess the rule.

Generating a string

So, we want to generate a string using a PDA.

- How can we generate the correct derivation?
  - We guess the rule. We always make a correct guess, because PDAs are nondeterministic machines.

Generating a string

So, we want to generate a string using a PDA.

- How can we generate the correct derivation?
  - We guess the rule. We always make a correct guess, because PDAs are nondeterministic machines.
- Where should we put the string (and its intermediate derivations)? How can we remember it?

Generating a string

So, we want to generate a string using a PDA.

- How can we generate the correct derivation?
  - We guess the rule. We always make a correct guess, because PDAs are nondeterministic machines.
- Where should we put the string (and its intermediate derivations)? How can we remember it?
  - A memory.

Generating a string

So, we want to generate a string using a PDA.

- How can we generate the correct derivation?
  - We guess the rule. We always make a correct guess, because PDAs are nondeterministic machines.
- Where should we put the string (and its intermediate derivations)? How can we remember it?
  - A memory. Yes, we have a memory: a stack

Generating a string

So, we want to generate a string using a PDA.

- How can we generate the correct derivation?
  - We guess the rule. We always make a correct guess, because PDAs are nondeterministic machines.
- Where should we put the string (and its intermediate derivations)? How can we remember it?
  - A memory. Yes, we have a memory: a stack
- But a stack has a very limited access rule. How can I do the derivation from $\mathtt{aAbB} \Rightarrow \mathtt{aaAbbB}$.

Generating a string

So, we want to generate a string using a PDA.

- How can we generate the correct derivation?
  - We guess the rule. We always make a correct guess, because PDAs are nondeterministic machines.
- Where should we put the string (and its intermediate derivations)? How can we remember it?
  - A memory. Yes, we have a memory: a stack
- But a stack has a very limited access rule. How can I do the derivation from aAbB $\Rightarrow$ aaAbbB.
  - What do you want to do?

Generating a string

So, we want to generate a string using a PDA.

- How can we generate the correct derivation?
  - We guess the rule. We always make a correct guess, because PDAs are nondeterministic machines.
- Where should we put the string (and its intermediate derivations)? How can we remember it?
  - A memory. Yes, we have a memory: a stack
- But a stack has a very limited access rule. How can I do the derivation from $\mathtt{aAbB} \Rightarrow \mathtt{aaAbbB}$.
  - What do you want to do? $\mathtt{aAbB} \Rightarrow$

Generating a string

So, we want to generate a string using a PDA.

- How can we generate the correct derivation?
  - We guess the rule. We always make a correct guess, because PDAs are nondeterministic machines.
- Where should we put the string (and its intermediate derivations)? How can we remember it?
  - A memory. Yes, we have a memory: a stack
- But a stack has a very limited access rule. How can I do the derivation from $\mathtt{aAbB} \Rightarrow \mathtt{aaAbbB}$.
  - What do you want to do? $\mathtt{aAbB} \Rightarrow \cancel{\mathtt{a}}\mathtt{AbB} \Rightarrow$

Generating a string

So, we want to generate a string using a PDA.

- How can we generate the correct derivation?
  - We guess the rule. We always make a correct guess, because PDAs are nondeterministic machines.
- Where should we put the string (and its intermediate derivations)? How can we remember it?
  - A memory. Yes, we have a memory: a stack
- But a stack has a very limited access rule. How can I do the derivation from $\mathtt{aAbB} \Rightarrow \mathtt{aaAbbB}$.
  - What do you want to do? $\mathtt{aAbB} \Rightarrow \cancel{\mathtt{a}}\mathtt{AbB} \Rightarrow \cancel{\mathtt{a}}\mathtt{aAbbB}$

Generating a string

So, we want to generate a string using a PDA.

- How can we generate the correct derivation?
  - We guess the rule. We always make a correct guess, because PDAs are nondeterministic machines.
- Where should we put the string (and its intermediate derivations)? How can we remember it?
  - A memory. Yes, we have a memory: a stack
- But a stack has a very limited access rule. How can I do the derivation from $\mathtt{aAbB} \Rightarrow \mathtt{aaAbbB}$.
  - What do you want to do? $\mathtt{aAbB} \Rightarrow \cancel{\mathtt{a}}\mathtt{AbB} \Rightarrow \cancel{\mathtt{a}}\mathtt{aAbbB}$
  - Okay, why are you stuck at a?

Generating a string

So, we want to generate a string using a PDA.

- How can we generate the correct derivation?
    - We guess the rule. We always make a correct guess, because PDAs are nondeterministic machines.
- Where should we put the string (and its intermediate derivations)? How can we remember it?
    - A memory. Yes, we have a memory: a stack
- But a stack has a very limited access rule. How can I do the derivation from `aAbB` $\Rightarrow$ `aaAbbB`.
    - What do you want to do? `aAbB` $\Rightarrow$ ~~a~~`AbB` $\Rightarrow$ ~~a~~`aAbbB`
    - Okay, why are you stuck at a?
    - Because it's not a variable.

Generating a string

So, we want to generate a string using a PDA.

- How can we generate the correct derivation?
  - We guess the rule. We always make a correct guess, because PDAs are nondeterministic machines.
- Where should we put the string (and its intermediate derivations)? How can we remember it?
  - A memory. Yes, we have a memory: a stack
- But a stack has a very limited access rule. How can I do the derivation from $\texttt{aAbB} \Rightarrow \texttt{aaAbbB}$.
  - What do you want to do? $\texttt{aAbB} \Rightarrow \cancel{\texttt{a}}\texttt{AbB} \Rightarrow \cancel{\texttt{a}}\texttt{aAbbB}$
  - Okay, why are you stuck at a?
  - Because it's not a variable.
  - So, anything we can do to **get rid of it**?

Generate and match

aabbccc | S

Generate and match

`aabbccc`	S
`aabbccc`	AB

Generate and match

left most

aabbccc	S
aabbccc	AB
aabbccc	$aAbB$

Generate and match

aabbccc	S
aabbccc	AB
aabbccc	$aAbB$
~~a~~abbccc	$\cancel{a}AbB$

Generate and match

aabbccc	S
aabbccc	AB
aabbccc	$aAbB$
~~a~~abbccc	~~a~~AbB
~~a~~abbccc	~~a~~$aAbbB$

Generate and match

aabbccc	S
aabbccc	AB
aabbccc	$aAbB$
~~a~~abbccc	~~a~~AbB
~~a~~abbccc	~~a~~$aAbbB$
~~aa~~bbccc	~~aa~~$AbbB$

Generate and match

aabbccc	S
aabbccc	AB
aabbccc	$aAbB$
~~a~~abbccc	$\cancel{a}AbB$
~~a~~abbccc	$\cancel{a}aAbbB$
~~aa~~bbccc	$\cancel{aa}AbbB$
~~aa~~bbccc	$\cancel{aa}bbB$

Generate and match

aabbccc	S
aabbccc	AB
aabbccc	$aAbB$
~~a~~abbccc	~~a~~AbB
~~a~~abbccc	~~a~~$aAbbB$
~~aa~~bbccc	~~aa~~$AbbB$
~~aa~~bbccc	~~aa~~bbB
~~aab~~bccc	~~aab~~bB

Generate and match

aabbccc	S
aabbccc	AB
aabbccc	$aAbB$
~~a~~abbccc	~~a~~AbB
~~a~~abbccc	~~a~~$aAbbB$
~~aa~~bbccc	~~aa~~$AbbB$
~~aa~~bbccc	~~aa~~bbB
~~aab~~bccc	~~aab~~bB
~~aabb~~ccc	~~aabb~~B

Generate and match

aabbccc	S
aabbccc	AB
aabbccc	$aAbB$
~~a~~abbccc	~~a~~AbB
~~a~~abbccc	~~a~~$aAbbB$
~~aa~~bbccc	~~aa~~$AbbB$
~~aa~~bbccc	~~aa~~bbB
~~aab~~bccc	~~aab~~bB
~~aabb~~ccc	~~aabb~~B
~~aabb~~ccc	~~aabb~~cB

Generate and match

aabbccc	S
aabbccc	AB
aabbccc	$aAbB$
~~a~~abbccc	~~a~~AbB
~~a~~abbccc	~~a~~$aAbbB$
~~aa~~bbccc	~~aa~~$AbbB$
~~aa~~bbccc	~~aa~~bbB
~~aab~~bccc	~~aab~~bB
~~aabb~~ccc	~~aabb~~B
~~aabb~~ccc	~~aabb~~cB
~~aabbc~~cc	~~aabbc~~B

Generate and match

aabbccc	S
aabbccc	AB
aabbccc	$aAbB$
~~a~~abbccc	~~a~~AbB
~~a~~abbccc	~~a~~$aAbbB$
~~aa~~bbccc	~~aa~~$AbbB$
~~aa~~bbccc	~~aa~~bbB
~~aab~~bccc	~~aab~~bB
~~aabb~~ccc	~~aabb~~B
~~aabb~~ccc	~~aabb~~cB
~~aabbc~~cc	~~aabbc~~B
~~aabbc~~cc	~~aabbc~~cB

Generate and match

aabbccc	S
aabbccc	AB
aabbccc	$aAbB$
~~a~~abbccc	~~a~~AbB
~~a~~abbccc	~~a~~$aAbbB$
~~aa~~bbccc	~~aa~~$AbbB$
~~aa~~bbccc	~~aa~~bbB
~~aab~~bccc	~~aab~~bB
~~aabb~~ccc	~~aabb~~B
~~aabb~~ccc	~~aabb~~cB
~~aabbc~~cc	~~aabbc~~B
~~aabbc~~cc	~~aabbc~~cB
~~aabbcc~~c	~~aabbcc~~B

Generate and match

aabbccc	S
aabbccc	AB
aabbccc	$aAbB$
~~a~~abbccc	~~a~~AbB
~~a~~abbccc	~~a~~$aAbbB$
~~aa~~bbccc	~~aa~~$AbbB$
~~aa~~bbccc	~~aa~~bbB
~~aab~~bccc	~~aab~~bB
~~aabb~~ccc	~~aabb~~B
~~aabb~~ccc	~~aabb~~cB
~~aabbc~~cc	~~aabbc~~B
~~aabbc~~cc	~~aabbc~~cB
~~aabbcc~~c	~~aabbcc~~B
~~aabbcc~~c	~~aabbcc~~c

Generate and match

aabbccc	S
aabbccc	AB
aabbccc	$aAbB$
~~a~~abbccc	~~a~~AbB
~~a~~abbccc	~~a~~$aAbbB$
~~aa~~bbccc	~~aa~~$AbbB$
~~aa~~bbccc	~~aa~~bbB
~~aab~~bccc	~~aab~~bB
~~aabb~~ccc	~~aabb~~B
~~aabb~~ccc	~~aabb~~cB
~~aabbc~~cc	~~aabbc~~B
~~aabbc~~cc	~~aabbc~~cB
~~aabbcc~~c	~~aabbcc~~B
~~aabbcc~~c	~~aabbcc~~c
~~aabbccc~~	~~aabbccc~~

The algorithm for PDA

1. Push empty stack symbol \$ on the stack
2. Push start variable on the stack

The algorithm for PDA

1. Push empty stack symbol \$ on the stack
2. Push start variable on the stack
3. **Repeat**

The algorithm for PDA

1. Push empty stack symbol \$ on the stack
2. Push start variable on the stack
3. **Repeat**
4. — Depending on the top of stack:

The algorithm for PDA

1. Push empty stack symbol $ on the stack
2. Push start variable on the stack
3. **Repeat**
4. — Depending on the top of stack:
5. — — **If it's a terminal,**
   — — — match with the same terminal on the input

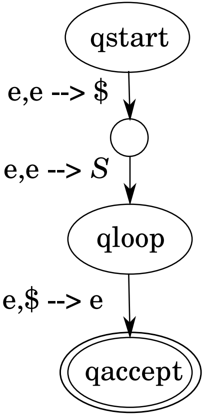
The algorithm for PDA

1. Push empty stack symbol \$ on the stack
2. Push start variable on the stack
3. **Repeat**
4. — Depending on the top of stack:
5. — — **If it's a terminal,**
   — — — match with the same terminal on the input
6. — — **If it's a variable,**
   — — — pick some substitution rule and put that on the stack

The algorithm for PDA

1. Push empty stack symbol \$ on the stack
2. Push start variable on the stack
3. **Repeat**
4. — Depending on the top of stack:
5. — — **If it's a terminal,**
   — — — match with the same terminal on the input
6. — — **If it's a variable,**
   — — — pick some substitution rule and put that on the stack
7. **Until** nothing's left on the stack (you'll see \$).
8. Accept if \$ is on top of the stack.

Overall structure



e, A --> w for rule A --> w

a, **a** --> e for each terminal **a**

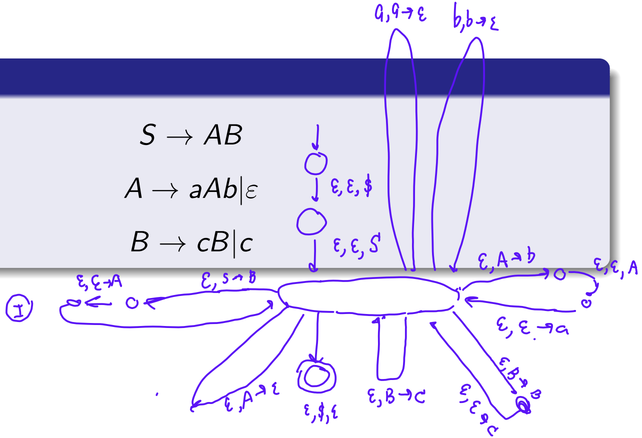
Practice:

CFG G_1

$$S \to AB$$

$$A \to aAb \mid \varepsilon$$

$$B \to cB \mid c$$



Practice:

CFG G_2

$$S \to aTb|b$$

$$T \to Ta|\varepsilon$$

Formal proof

$S \rightarrow aTb|b$

$T \rightarrow Ta|\varepsilon$