CFGs, PDAs, and the Pumping Lemma

204213 Theory of Computation

Jittat Fakcharoenphol

Kasetsart University

December 6, 2008

Outline

Quiz

- Let $A = \{\mathrm{a}^{(n^2)} | n \geq 0\}$.

Quiz

- Let $A = \{\mathrm{a}^{(n^2)} | n \geq 0\}$.
- E.g., a, aaaa, aaaaaaaaa$\in A$.

Quiz

- Let $A = \{\mathrm{a}^{(n^2)} | n \geq 0\}$.
- E.g., a, aaaa, aaaaaaaaa$\in A$.
- Prove that A is not regular.
- **Hint:** using the pumping lemma (together with some calculations)

More Example

- Let $B = \{0^i 1^j | i > j \geq 0\}$
- Prove that B is not regular.
- **Hint:** try to pump down.

More Example

- Let $B = \{0^i 1^j | i > j \geq 0\}$
- Prove that B is not regular.
- **Hint:** try to pump down.
- **More hint:** Let p be the pumping length. Try $0^{p+1}1^p$.

Review: Chomsky normal form

CNF

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

$$A \longrightarrow BC$$

$$A \longrightarrow a$$

where a is any terminal and A, B, and C are any variables,

Review: Chomsky normal form

CNF

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

$$A \longrightarrow BC$$

$$A \longrightarrow a$$

where a is any terminal and A, B, and C are any variables, except that B and C cannot be the start variable.

Review: Chomsky normal form

CNF

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

$$A \longrightarrow BC$$

$$A \longrightarrow a$$

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 $S \longrightarrow \varepsilon$, 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: $S \longrightarrow aS$, or $A \longrightarrow BS$.

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

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

$$S_0 \longrightarrow S$$

Step 2: ε rules

- Sample rules:

$$B \longrightarrow aAb|bAcA$$

$$A \longrightarrow c|aA|\varepsilon$$

Step 2: ε rules

- Sample rules:

$$B \longrightarrow aAb|bAcA$$

$$A \longrightarrow c|aA|\varepsilon$$

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

$$B \longrightarrow aAb \Rightarrow$$

Step 2: ε rules

- Sample rules:

$$B \longrightarrow aAb|bAcA$$

$$A \longrightarrow c|aA|\varepsilon$$

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

$$B \longrightarrow aAb \Rightarrow B \longrightarrow aAb|ab$$

Step 2: ε rules

- Sample rules:

$$B \longrightarrow aAb|bAcA$$

$$A \longrightarrow c|aA|\varepsilon$$

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

$$B \longrightarrow aAb \Rightarrow B \longrightarrow aAb|ab$$

$$B \longrightarrow bAcA \Rightarrow$$

Step 2: ε rules

- Sample rules:

$$B \longrightarrow aAb|bAcA$$

$$A \longrightarrow c|aA|\varepsilon$$

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

$$B \longrightarrow aAb \Rightarrow B \longrightarrow aAb|ab$$

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

Step 2: ε rules

- Sample rules:

$$B \longrightarrow aAb|bAcA$$

$$A \longrightarrow c|aA|\varepsilon$$

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

$$B \longrightarrow aAb \Rightarrow B \longrightarrow aAb|ab$$

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

$$A \longrightarrow aA \Rightarrow$$

Step 2: ε rules

- Sample rules:

$$B \longrightarrow aAb|bAcA$$

$$A \longrightarrow c|aA|\varepsilon$$

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

$$B \longrightarrow aAb \Rightarrow B \longrightarrow aAb|ab$$

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

$$A \longrightarrow aA \Rightarrow A \longrightarrow aA|a$$

Step 3: unit rules

- Sample rules:

$$C \rightarrow Ba|Ac$$
$$B \rightarrow aAb|bAcA$$
$$A \rightarrow B|c$$

Step 3: unit rules

- Sample rules:

$$C \longrightarrow Ba|Ac$$
$$B \longrightarrow aAb|bAcA$$
$$A \longrightarrow B|c$$

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

$$C \longrightarrow Ba|Ac \Rightarrow$$

Step 3: unit rules

- Sample rules:

$$C \longrightarrow Ba|Ac$$

$$B \longrightarrow aAb|bAcA$$

$$A \longrightarrow B|c$$

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

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

Step 3: unit rules

- Sample rules:

$$C \longrightarrow Ba|Ac$$

$$B \longrightarrow aAb|bAcA$$

$$A \longrightarrow B|c$$

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

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

$$B \longrightarrow aAb|bAcA \Rightarrow$$

Step 3: unit rules

- Sample rules:

$$C \longrightarrow Ba|Ac$$

$$B \longrightarrow aAb|bAcA$$

$$A \longrightarrow B|c$$

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

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

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

Step 3: unit rules

- Sample rules:

$$C \longrightarrow Ba|Ac$$
$$B \longrightarrow aAb|bAcA$$
$$A \longrightarrow B|c$$

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

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

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

Step 3: unit rules

- Sample rules:

$$C \longrightarrow Ba|Ac$$

$$B \longrightarrow aAb|bAcA$$

$$A \longrightarrow B|c$$

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

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

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

$$A \longrightarrow c$$

Step 4: long rules

- Sample rules:

$$C \rightarrow abC|asbdB$$

Step 4: long rules

- Sample rules:

$$C \longrightarrow abC|asbdB$$

- 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:

$$C \longrightarrow abC|asbdB$$

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

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

Step 4: long rules

- Sample rules:

$$C \rightarrow abC|asbdB$$

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

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

$$C \rightarrow asbdB \Rightarrow$$

Step 5: remove rules with terminal

- Sample rules:

$$C \longrightarrow aC$$

$$D \longrightarrow ab|a$$

Step 5: remove rules with terminal

- Sample rules:

$$C \longrightarrow aC$$

$$D \longrightarrow ab|a$$

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

Step 5: remove rules with terminal

- Sample rules:

$$C \longrightarrow aC$$

$$D \longrightarrow ab|a$$

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

$$C \longrightarrow AC$$

$$A \longrightarrow a$$

$$D \longrightarrow AB|a$$

$$B \longrightarrow b$$

Context-free languages

CFL

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

Equivalence

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.

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:

$$S \longrightarrow AB$$

$$A \longrightarrow aAb|\varepsilon$$

$$B \longrightarrow cB|c$$

- 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

$$S \rightarrow AB$$

$$A \rightarrow aAb|\varepsilon$$

$$B \rightarrow cB|c$$

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:

$$\begin{aligned} S &\Rightarrow AB \\ &\Rightarrow \mathbf{aAb}B \\ &\Rightarrow a\mathbf{aAb}bB \\ &\Rightarrow aa\varepsilon bbB \\ &\Rightarrow aabb\mathbf{cB} \\ &\Rightarrow aabbc\mathbf{cB} \\ &\Rightarrow aabbcc\mathbf{c} \end{aligned}$$

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 `aAbB` $\Rightarrow$ `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 $\mathtt{aAbB} \Rightarrow \mathtt{aaAbbB}$.
  - 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 aAbB $\Rightarrow$ aaAbbB.
  - What do you want to do? aAbB $\Rightarrow$ ~~a~~AbB $\Rightarrow$ ~~a~~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 aAbB $\Rightarrow$ aaAbbB.
  - What do you want to do? aAbB $\Rightarrow$ ~~a~~AbB $\Rightarrow$ ~~a~~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 $\mathtt{aAbB} \Rightarrow \mathtt{aaAbbB}$.
  - 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?
  - 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 $\mathtt{aAbB} \Rightarrow \mathtt{aaAbbB}$.
    - 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?
    - 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

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	~~a~~AbB
~~a~~abbccc	~~a~~$aAbbB$
~~aa~~bbccc	~~aa~~$AbbB$
~~aa~~bbccc	~~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

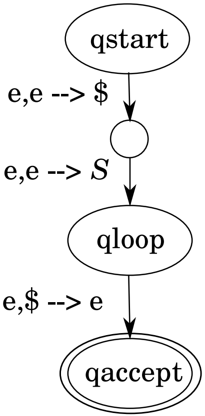
The algorithm for PDA

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

$$S \longrightarrow AB$$

$$A \longrightarrow aAb|\varepsilon$$

$$B \longrightarrow cB|c$$

Practice:

CFG G_2

$$S \longrightarrow aTb|b$$

$$T \longrightarrow Ta|\varepsilon$$

Formal proof

Non-context-free language

Can you find a CFG describing the language $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n | n \geq 0\}$?

Non-context-free language

Can you find a CFG describing the language $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n | n \geq 0\}$?
I bet you can't.

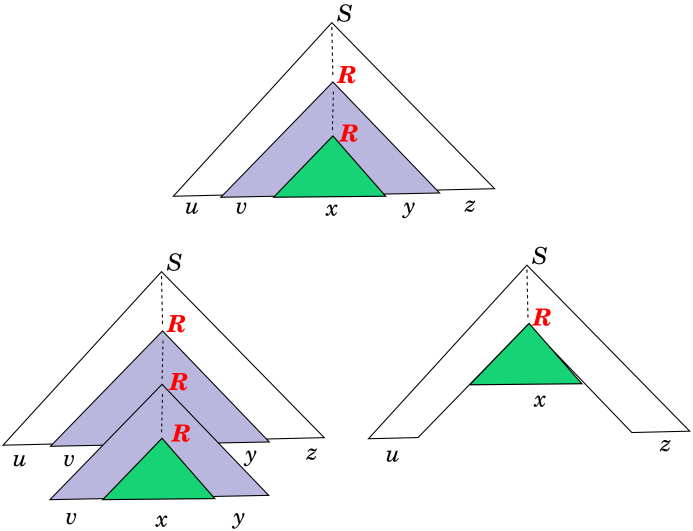
Pumping lemma for CFL

Theorem 3 (pumping lemma for CFL)

If A is a context-free language, then there is a pumping length p such that for any string $s \in A$ of length at least p, s can be divided into 5 pieces $s = uvxyz$ satisfying the following conditions

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

Parse tree for s



$C = \{a^n b^n c^n | n \geq 0\}$ is not context-free (1)

- We'll prove by contradiction. Assume that C is context-free.

$C = \{a^n b^n c^n | n \geq 0\}$ is not context-free (1)

- We'll prove by contradiction. Assume that C is context-free.
- Thus, there exists a pumping length p.

$C = \{a^n b^n c^n | n \geq 0\}$ is not context-free (1)

- We'll prove by contradiction. Assume that C is context-free.
- Thus, there exists a pumping length p.
- Consider $s = a^p b^p c^p \in C$. Note that $|s| \geq p$.

$C = \{a^n b^n c^n | n \geq 0\}$ is not context-free (1)

- We'll prove by contradiction. Assume that C is context-free.
- Thus, there exists a pumping length p.
- Consider $s = a^p b^p c^p \in C$. Note that $|s| \geq p$.
- The pumping lemma states that we can divide $s = uvxyz$, such that $uv^i xy^i z \in C$ for any $i \geq 0$.

$C = \{a^n b^n c^n | n \geq 0\}$ is not context-free (1)

- We'll prove by contradiction. Assume that C is context-free.
- Thus, there exists a pumping length p.
- Consider $s = a^p b^p c^p \in C$. Note that $|s| \geq p$.
- The pumping lemma states that we can divide $s = uvxyz$, such that $uv^i xy^i z \in C$ for any $i \geq 0$.
- We'll show that this leads to a contradiction.

First proof

There are two cases.

- **Case 1**, if each of v and y contains only one kind of alphabets.

First proof

There are two cases.

- **Case 1**, if each of v and y contains only one kind of alphabets. Then consider $s' = uv^2xy^2z$. Note that at least one alphabet appears in s' in fewer times than the others; thus, $s' \notin C$.

First proof

There are two cases.

- **Case 1**, if each of v and y contains only one kind of alphabets. Then consider $s' = uv^2xy^2z$. Note that at least one alphabet appears in s' in fewer times than the others; thus, $s' \notin C$.
- **Case 2**, if v or y contains two kinds of alphabets.

First proof

There are two cases.

- **Case 1**, if each of v and y contains only one kind of alphabets. Then consider $s' = uv^2xy^2z$. Note that at least one alphabet appears in s' in fewer times than the others; thus, $s' \notin C$.
- **Case 2**, if v or y contains two kinds of alphabets. Note that $s' = uv^2xy^2z$ contains alphabets in the wrong order. Again, $s' \notin C$.

First proof

There are two cases.

- **Case 1**, if each of v and y contains only one kind of alphabets. Then consider $s' = uv^2xy^2z$. Note that at least one alphabet appears in s' in fewer times than the others; thus, $s' \notin C$.
- **Case 2**, if v or y contains two kinds of alphabets. Note that $s' = uv^2xy^2z$ contains alphabets in the wrong order. Again, $s' \notin C$.

Note that in either case, s cannot be pumped, and this contradicts the assumption that C is context-free.

Second proof

Note that the first proof doesn't use the 3rd property, stating that $|vxy| \leq p$.

Second proof

Note that the first proof doesn't use the 3rd property, stating that $|vxy| \leq p$. Given that fact, we know that v and y cannot contain all three types of alphabets.

Second proof

Note that the first proof doesn't use the 3rd property, stating that $|vxy| \leq p$. Given that fact, we know that v and y cannot contain all three types of alphabets. Therefore, $s' = uv^2xy^2z$ contains different numbers of a's, b's, or c's, and $s' \notin C$.

Second proof

Note that the first proof doesn't use the 3rd property, stating that $|vxy| \leq p$. Given that fact, we know that v and y cannot contain all three types of alphabets. Therefore, $s' = uv^2xy^2z$ contains different numbers of a's, b's, or c's, and $s' \notin C$. This, again, leads to a contradiction.