FORMAL LANGUAGES AND AUTOMATA THEORY

UNIT 3

Ambiguity in CFGs and CFLs

Ambiguity in CFGs

- A CFG is said to be *ambiguous* if there exists a string which has more than one left-most derivation

Example:

S ==> AS | ε

A ==> A1 | 0A1 | 01

LM derivation #1:

S => AS
=> 0A1S
=>0A11S
=> 00111S
=> 00111

LM derivation #2:

S => AS
=> A1S
=> 0A11S
=> 00111S
=> 00111

Input string: 00111

Can be derived in two ways

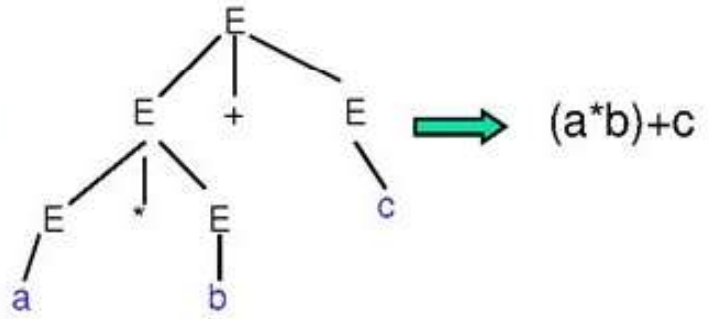
Why does ambiguity matter?

E ==> E + E | E * E | (E) | a | b | c | 0 | 1

Values are different !!!

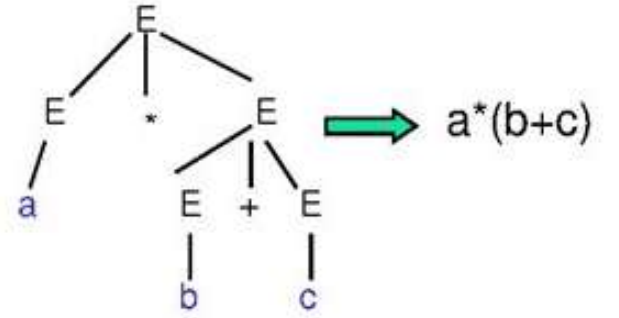
*string = a * b + c*

- LM derivation #1:
  - E => E + E => E * E + E
    ==>* a * b + c



(a*b)+c

- LM derivation #2
  - E => E * E => a * E =>
    a * E + E ==>* a * b + c



a*(b+c)

The calculated value depends on which of the two parse trees is actually used.

Removing Ambiguity in Expression Evaluations

- It MAY be possible to remove ambiguity for some CFLs
  - E.g.,, in a CFG for expression evaluation by imposing rules & restrictions such as precedence
  - This would imply rewrite of the grammar
- Precedence: (), * , +

Modified unambiguous version:

E => E + T | T
T => T * F | F
F => I | (E)
I => a | b | c | 0 | 1

How will this avoid ambiguity?

Ambiguous version:

E ==> E + E | E * E | (E) | a | b | c | 0 | 1

Inherently Ambiguous CFLs

- However, for some languages, it may not be possible to remove ambiguity

- A CFL is said to be *inherently ambiguous* if every CFG that describes it is ambiguous

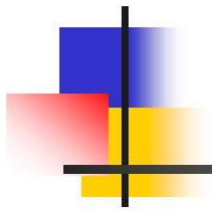
Example:

- L = { $a^n b^n c^m d^m$ | n,m≥ 1} U {$a^n b^m c^m d^n$ | n,m≥ 1}
- L is inherently ambiguous
- Why?

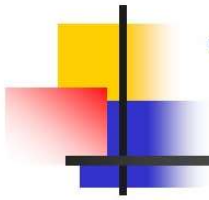
Input string: $a^n b^n c^n d^n$

Summary

- Ambiguous grammars
- Removing ambiguity

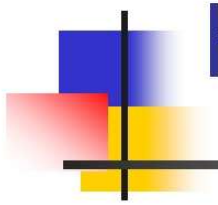


Properties of Context-free Languages

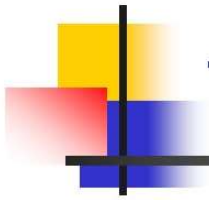


Topics

1) Simplifying CFGs, Normal forms
2) Pumping lemma for CFLs
3) Closure and decision properties of CFLs



How to “simplify” CFGs?



Three ways to simplify/clean a CFG

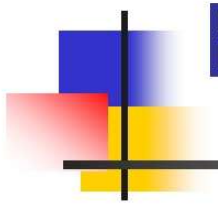
(clean)

1. Eliminate *useless symbols*

(simplify)

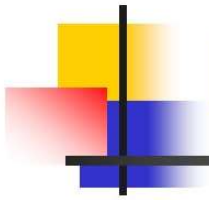
2. Eliminate ε-productions $A \not\Rightarrow \varepsilon$

3. Eliminate *unit productions* $A \not\Rightarrow B$



Eliminating useless symbols

Grammar cleanup



Eliminating *useless symbols*

A symbol X is *reachable* if there exists:

- $S \rightarrow^* \alpha X \beta$

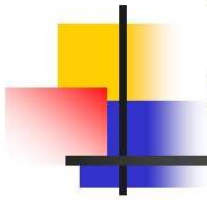
A symbol X is *generating* if there exists:

- $X \rightarrow^* w$,
  - for some $w \in T^*$

For a symbol X to be "useful", it has to be both reachable *and* generating

- $S \rightarrow^* \alpha X \beta \rightarrow^* w'$, for some $w' \in T^*$

reachable generating



Algorithm to detect useless symbols

1. First, eliminate all symbols that are *not* generating

2. Next, eliminate all symbols that are *not* reachable

Is the order of these steps important, or can we switch?

Example: Useless symbols

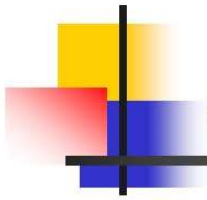
- S➔AB | a
- A➔ b

1. *A, S* are generating
2. *B* is *not generating* (and therefore B is useless)
3. ==> Eliminating B… (i.e., remove all productions that involve B)
   1. S➔ a
   2. A ➔ b
4. Now, A is *not reachable* and therefore is useless
5. Simplified G:
   1. S ➔ a

What would happen if you reverse the order:
i.e., test reachability before generating?

Will fail to remove:
A ➔ b

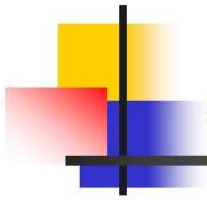
$X \Rightarrow^* w$



Algorithm to find all generating symbols

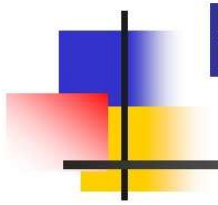
- Given: G=(V,T,P,S)
- Basis:
  - Every symbol in T is obviously generating.
- Induction:
  - Suppose for a production A→ α, where α is generating
  - Then, A is also generating

$S \rightarrow^{*} \alpha X \beta$



Algorithm to find all reachable symbols

- Given: G=(V,T,P,S)
- Basis:
  - S is obviously reachable (from itself)
- Induction:
  - Suppose for a production A→ $\alpha_1 \alpha_2 \ldots \alpha_k$, where A is reachable
  - Then, all symbols on the right hand side, $\{\alpha_1, \alpha_2, \ldots \alpha_k\}$ are also reachable.



Eliminating ε-productions

A => ε

What's the point of removing ε-productions?

$A \rightarrow \varepsilon$

Eliminating ε-productions

Caveat: It is *not* possible to eliminate ε-productions for languages which include ε in their word set

So we will target the grammar for the *rest of the language*

Theorem: If G=(V,T,P,S) is a CFG for a language L, then L-{ε} has a CFG without ε-productions

Definition: A is "nullable" if $A \rightarrow^* \varepsilon$

- If A is nullable, then any production of the form "B→ CAD" can be simulated by:
  - B → CD | CAD
    - This can allow us to remove ε transitions for A

Algorithm to detect all nullable variables

- Basis:
  - If A➔ ε is a production in G, then A is nullable
    (note: A can still have other productions)
- Induction:
  - If there is a production B➔ $C_1C_2\ldots C_k$, where *every* C_i is nullable, then B is also nullable

Eliminating ε-productions

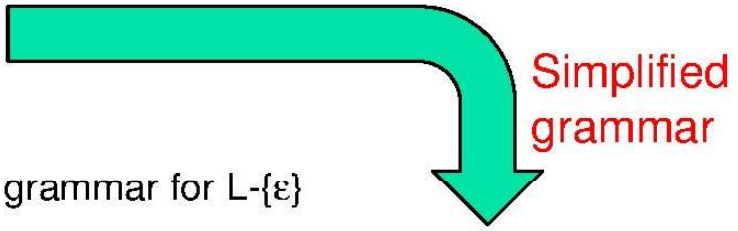
Given: G=(V,T,P,S)

Algorithm:

1. Detect all nullable variables in G
2. Then construct $G_1=(V,T,P_1,S)$ as follows:
   i. For each production of the form: $A \rightarrow X_1X_2\ldots X_k$, where $k \geq 1$, suppose ***m*** out of the ***k*** X_i's are nullable symbols
   ii. Then G_1 will have 2^m versions for this production
      i. i.e, all combinations where each X_i is either present or absent
   iii. Alternatively, if a production is of the form: $A \rightarrow \varepsilon$, then remove it

Example: Eliminating ε-productions

- Let L be the language represented by the following CFG G:
  i. S → AB
  ii. A → aAA | ε
  iii. B → bBB | ε



Goal: To construct G1, which is the grammar for L-{ε}

- Nullable symbols: {A, B}
- G_1 can be constructed from G as follows:
  - B → b | bB | bB | bBB
    - ==> B → b | bB | bBB
  - Similarly, A → a | aA | aAA
  - Similarly, S → A | B | AB
- Note: $L(G) = L(G_1) \cup \{\varepsilon\}$

G_1:
- S → A | B | AB
- A → a | aA | aAA
- B → b | bB | bBB

+

- S → ε

Eliminating unit productions

A => B ← B has to be a variable

X

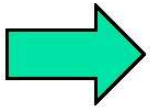
What's the point of removing unit transitions ?

Will save #substitutions

E.g.,

A=>B | …
B=>C | …
C=>D | …
D=>xxx | yyy | zzz

before



A=>xxx | yyy | zzz | …
B=> xxx | yyy | zzz | …
C=> xxx | yyy | zzz | …
D=>xxx | yyy | zzz

after

$A \rightarrow B$

Eliminating unit productions

- Unit production is one which is of the form A➔ B, where both A & B are variables
- E.g.,
  1. E ➔ T | E+T
  2. T ➔ F | T*F
  3. F ➔ I | (E)
  4. I ➔ a | b | Ia | Ib | I0 | I1
  - How to eliminate unit productions?
    - Replace E➔ T with E ➔ F | T*F
    - Then, upon recursive application wherever there is a unit production:
      - E➔ F | T*F | E+T (substituting for T)
      - E➔ I | (E) | T*F| E+T (substituting for F)
      - E➔ a | b | Ia | Ib | I0 | I1 | (E) | T*F | E+T (substituting for I)
      - Now, E has no unit productions
    - Similarly, eliminate for the remainder of the unit productions

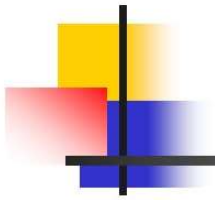
The **Unit Pair Algorithm**: to remove unit productions

- Suppose $A \rightarrow B_1 \rightarrow B_2 \rightarrow \ldots \rightarrow B_n \rightarrow \alpha$
- Action: Replace all intermediate productions to produce α directly
  - i.e., $A \rightarrow \alpha$; $B_1 \rightarrow \alpha$; ... $B_n \rightarrow \alpha$;

Definition: (A,B) to be a ***"unit pair"*** if $A \rightarrow^* B$

- We can find all unit pairs inductively:
  - Basis: Every pair (A,A) is a unit pair (by definition). Similarly, if $A \rightarrow B$ is a production, then (A,B) is a unit pair.
  - Induction: If (A,B) and (B,C) are unit pairs, and $A \rightarrow C$ is also a unit pair.

The Unit Pair Algorithm: to remove unit productions



Input: G=(V,T,P,S)

Goal: to build G_1=(V,T,P_1,S) devoid of unit productions

Algorithm:

1. Find all unit pairs in G
2. For each unit pair (A,B) in G:
    1. Add to P_1 a new production A➔α, for every B➔α which is a *non-unit* production
    2. If a resulting production is already there in P, then there is no need to add it.

Example: eliminating unit productions

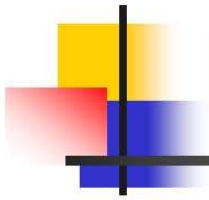
G:

1. E → T | E+T
2. T → F | T*F
3. F → I | (E)
4. I → a | b | Ia | Ib | I0 | I1

Unit pairs	*Only non-unit productions to be added to P_1*
(E,E)	E → E+T
(E,T)	E → T*F
(E,F)	E → (E)
(E,I)	E → a|b|Ia | Ib | I0 | I1
(T,T)	T → T*F
(T,F)	T → (E)
(T,I)	T → a|b| Ia | Ib | I0 | I1
(F,F)	F → (E)
(F,I)	F → a| b| Ia | Ib | I0 | I1
(I,I)	I → a| b | Ia | Ib | I0 | I1

G_1:

1. E → E+T | T*F | (E) | a| b | Ia | Ib | I0 | I1
2. T → T*F | (E) | a| b | Ia | Ib | I0 | I1
3. F → (E) | a| b | Ia | Ib | I0 | I1
4. I → a | b | Ia | Ib | I0 | I1

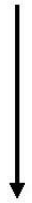


Putting all this together…

- Theorem: If G is a CFG for a language that contains at least one string other than ε, then there is another CFG G_1, such that $L(G_1)=L(G) - \varepsilon$, *and* G_1 has:
  - no ε -productions
  - no unit productions
  - no useless symbols

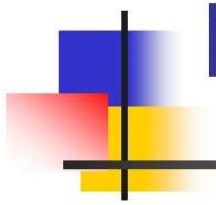
- Algorithm:

Step 1) eliminate ε -productions
Step 2) eliminate unit productions
Step 3) eliminate useless symbols

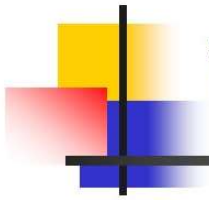


Again,
the order is important!

Why?



Normal Forms



Why normal forms?

- If all productions of the grammar could be expressed in the same form(s), then:

  a. It becomes easy to design algorithms that use the grammar

  b. It becomes easy to show proofs and properties

Chomsky Normal Form (CNF)

Let G be a CFG for some L-{ε}

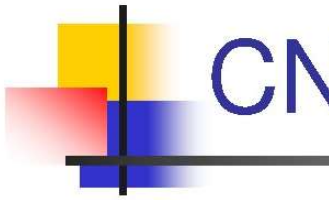
Definition:

*G is said to be in **Chomsky Normal Form** if all its productions are in one of the following two forms:*

i. ***A → BC*** *where A,B,C are variables, or*

ii. ***A → a*** *where a is a terminal*

- *G has no useless symbols*
- *G has no unit productions*
- *G has no ε-productions*



CNF checklist

Is this grammar in CNF?

G_1:
1. E → E+T | T*F | (E) | Ia | Ib | I0 | I1
2. T → T*F | (E) | Ia | Ib | I0 | I1
3. F → (E) | Ia | Ib | I0 | I1
4. I → a | b | Ia | Ib | I0 | I1

Checklist:
- G has no ε-productions ✓
- G has no unit productions ✓
- G has no useless symbols ✓
- But…
  - the normal form for productions is violated

⟹ So, the grammar is not in CNF

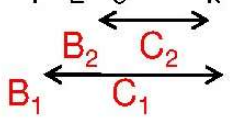
How to convert a G into CNF?

- Assumption: G has no ε-productions, unit productions or useless symbols

1) For every terminal ***a*** that appears in the body of a production:
   i. create a unique variable, say X_a, with a production $X_a \rightarrow a$, and
   ii. replace all other instances of a in G by X_a

2) Now, all productions will be in one of the following two forms:
   - $A \rightarrow B_1B_2\ldots B_k$ $(k\geq 3)$ or $A \rightarrow a$

3) Replace each production of the form $A \rightarrow B_1B_2B_3\ldots B_k$ by:



and so on…

   - $A \rightarrow B_1C_1$ $C_1 \rightarrow B_2C_2$ … $C_{k-3} \rightarrow B_{k-2}C_{k-2}$ $C_{k-2} \rightarrow B_{k-1}B_k$

Example #1

G:

S => AS | BABC

A => A1 | 0A1 | 01

B => 0B | 0

C => 1C | 1

G in CNF:

X_0 => 0

X_1 => 1

S => AS | BY_1

Y_1 => AY_2

Y_2 => BC

A => AX_1 | X_0Y_3 | X_0X_1

Y_3 => AX_1

B => X_0B | 0

C => X_1C | 1

All productions are of the form: A=>BC or A=>a

Example #2

G:

1. E → E+T | T*F | (E) | Ia | Ib | I0 | I1
2. T → T*F | (E) | Ia | Ib | I0 | I1
3. F → (E) | Ia | Ib | I0 | I1
4. I → a | b | Ia | Ib | I0 | I1

Step (1)

1. E → $EX_{+}T$ | $TX_{*}F$ | $X_{(}EX_{)}$ | IX_a | IX_b | IX_0 | IX_1
2. T → $TX_{*}F$ | $X_{(}EX_{)}$ | IX_a | IX_b | IX_0 | IX_1
3. F → $X_{(}EX_{)}$ | IX_a | IX_b | IX_0 | IX_1
4. I → X_a | X_b | IX_a | IX_b | IX_0 | IX_1
5. X_{+} → +
6. X_{*} → *
7. X_{+} → +
8. $X_{(}$ → (
9. .....

Step (2)

1. E → EC_1 | TC_2 | $X_{(}C_3$ | IX_a | IX_b | IX_0 | IX_1
2. C_1 → $X_{+}T$
3. C_2 → $X_{*}F$
4. C_3 → $EX_{)}$
5. T → .....
6. ....

Languages with ε

- For languages that include ε,
  - Write down the rest of grammar in CNF
  - Then add production "S => ε" at the end

E.g., consider:

G:

S => AS | BABC

A => A1 | 0A1 | 01 | ε

B => 0B | 0 | ε

C => 1C | 1 | ε

G in CNF:

$X_0 \Rightarrow 0$

$X_1 \Rightarrow 1$

$S \Rightarrow AS \mid BY_1 \mid \varepsilon$

$Y_1 \Rightarrow AY_2$

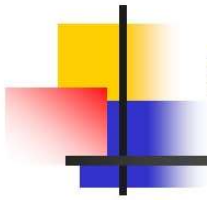
$Y_2 \Rightarrow BC$

$A \Rightarrow AX_1 \mid X_0Y_3 \mid X_0X_1$

$Y_3 \Rightarrow AX_1$

$B \Rightarrow X_0B \mid 0$

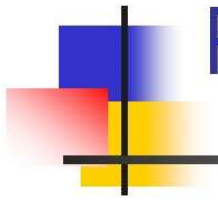
$C \Rightarrow X_1C \mid 1$



Other Normal Forms

- Griebach Normal Form (GNF)
  - All productions of the form

$$A ==> a\ \alpha$$

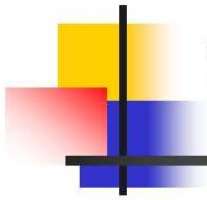


Return of the Pumping Lemma !!

Think of languages that cannot be CFL

== think of languages for which a stack will not be enough

e.g., the language of strings of the form *ww*



Why pumping lemma?

- A result that will be useful in proving languages that *are not* CFLs
  - (just like we did for regular languages)

- But before we prove the pumping lemma for CFLs ….
  - Let us first prove an important property about parse trees

Observe that any parse tree generated by a CNF will be a binary tree, where all internal nodes have exactly two children (except those nodes connected to the leaves).

The "parse tree theorem"

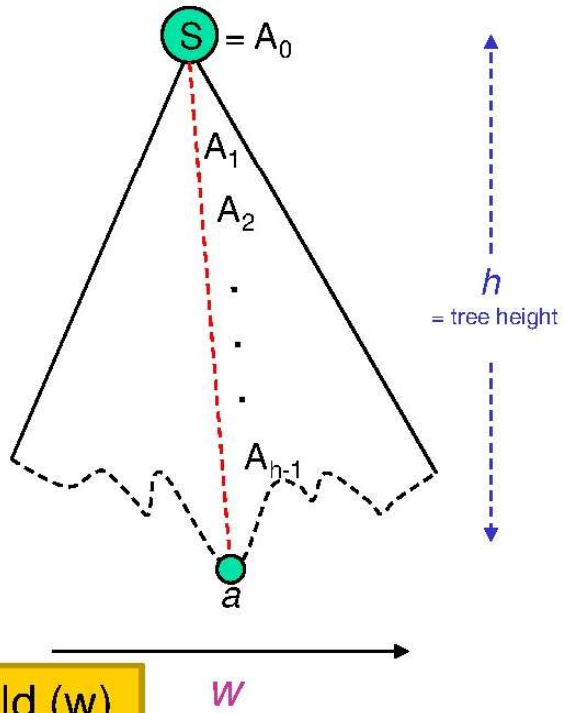
Given:

- Suppose we have a parse tree for a string ***w***, according to a CNF grammar, G=(V,T,P,S)
- Let *h* be the height of the parse tree

Implies:

- $|w| \leq 2^{h-1}$

Parse tree for w



In other words, a CNF parse tree's string yield (w) can no longer be 2^{h-1}

To show: $|w| \leq 2^{h-1}$

Proof…The size of parse trees

Proof: (using induction on h)

Basis: h = 1

- Derivation will have to be "S→a"
- $|w| = 1 = 2^{1-1}$.

Ind. Hyp: h = k-1

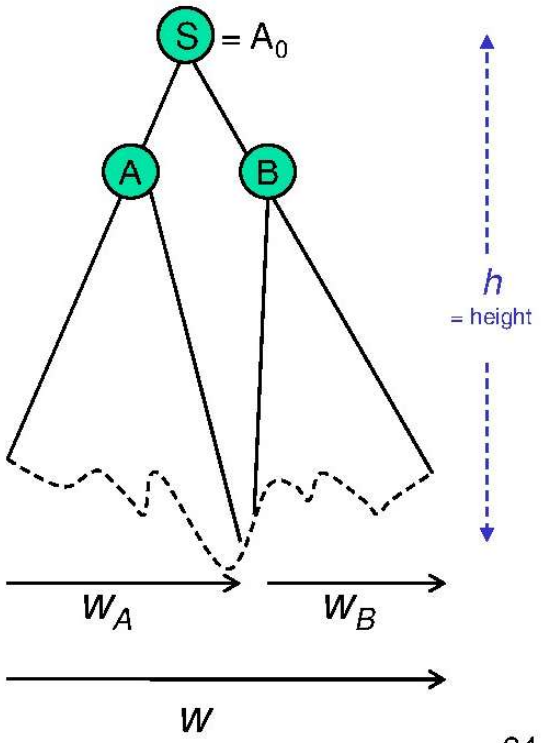
- $|w| \leq 2^{k-2}$

Ind. Step: h = k

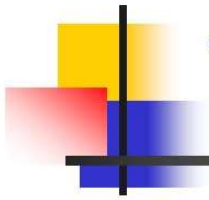
S will have exactly two children: S→AB

- Heights of A & B subtrees are at most h-1
- $w = w_A w_B$, where $|w_A| \leq 2^{k-2}$ and $|w_B| \leq 2^{k-2}$
- $|w| \leq 2^{k-1}$

Parse tree for w



Implication of the Parse Tree Theorem (assuming CNF)

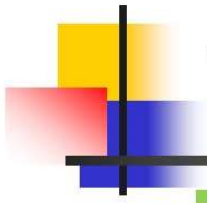


Fact:

- If the height of a parse tree is h, then
  - ==> $|w| \leq 2^{h-1}$

Implication:

- **If $|w| \geq 2^h$, then**
  - **Its parse tree's height is *at least* h+1**



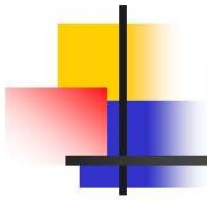
The Pumping Lemma for CFLs

Let L be a CFL.

Then there exists a constant N, s.t.,

- if $z \in L$ s.t. $|z| \geq N$, then we can write $z = uvwxy$, such that:
  1. $|vwx| \leq N$
  2. $vx \neq \varepsilon$
  3. For all $k \geq 0$: $uv^k wx^k y \in L$

Note: we are pumping in two places (v & x)



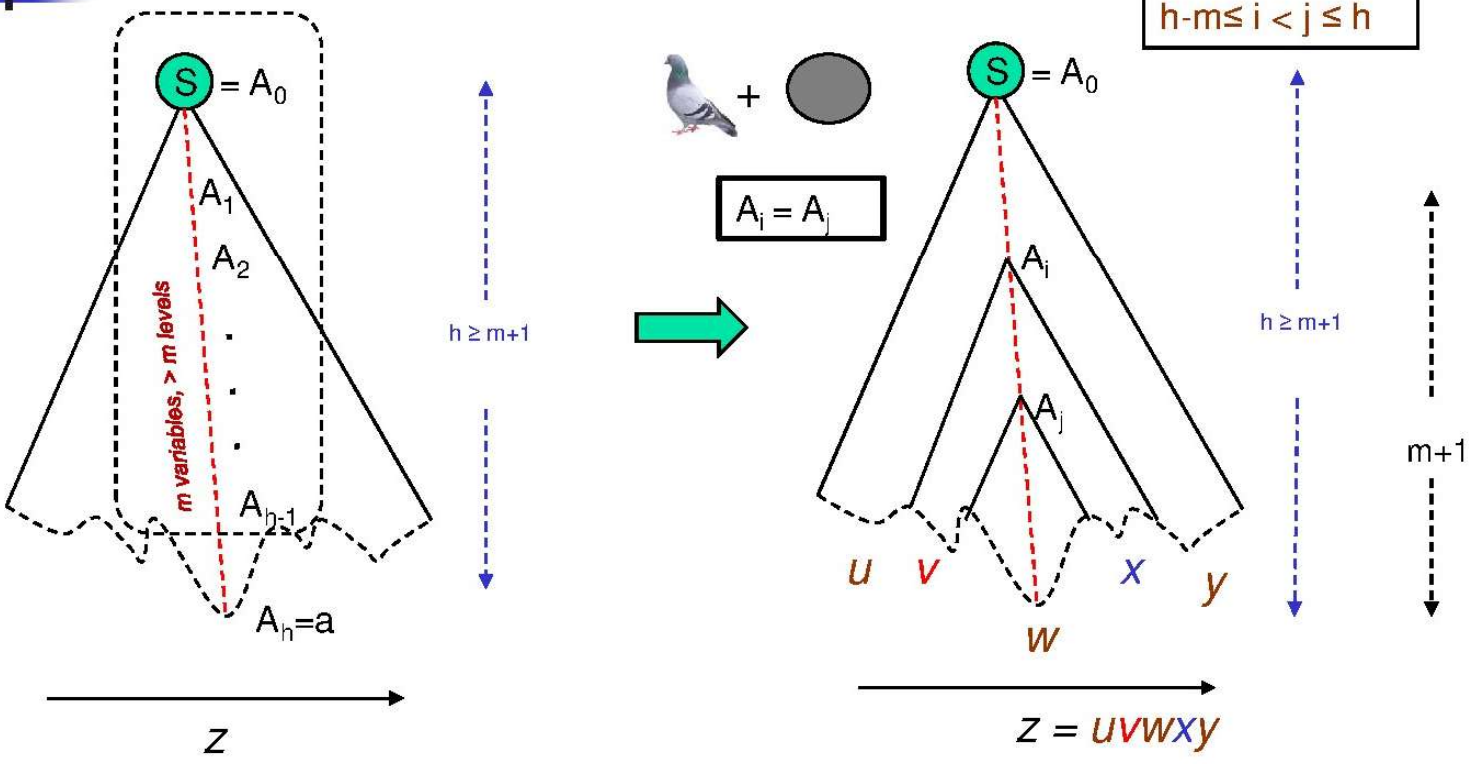
Proof: Pumping Lemma for CFL

- If L=Φ or contains only ε, then the lemma is trivially satisfied (as it cannot be violated)

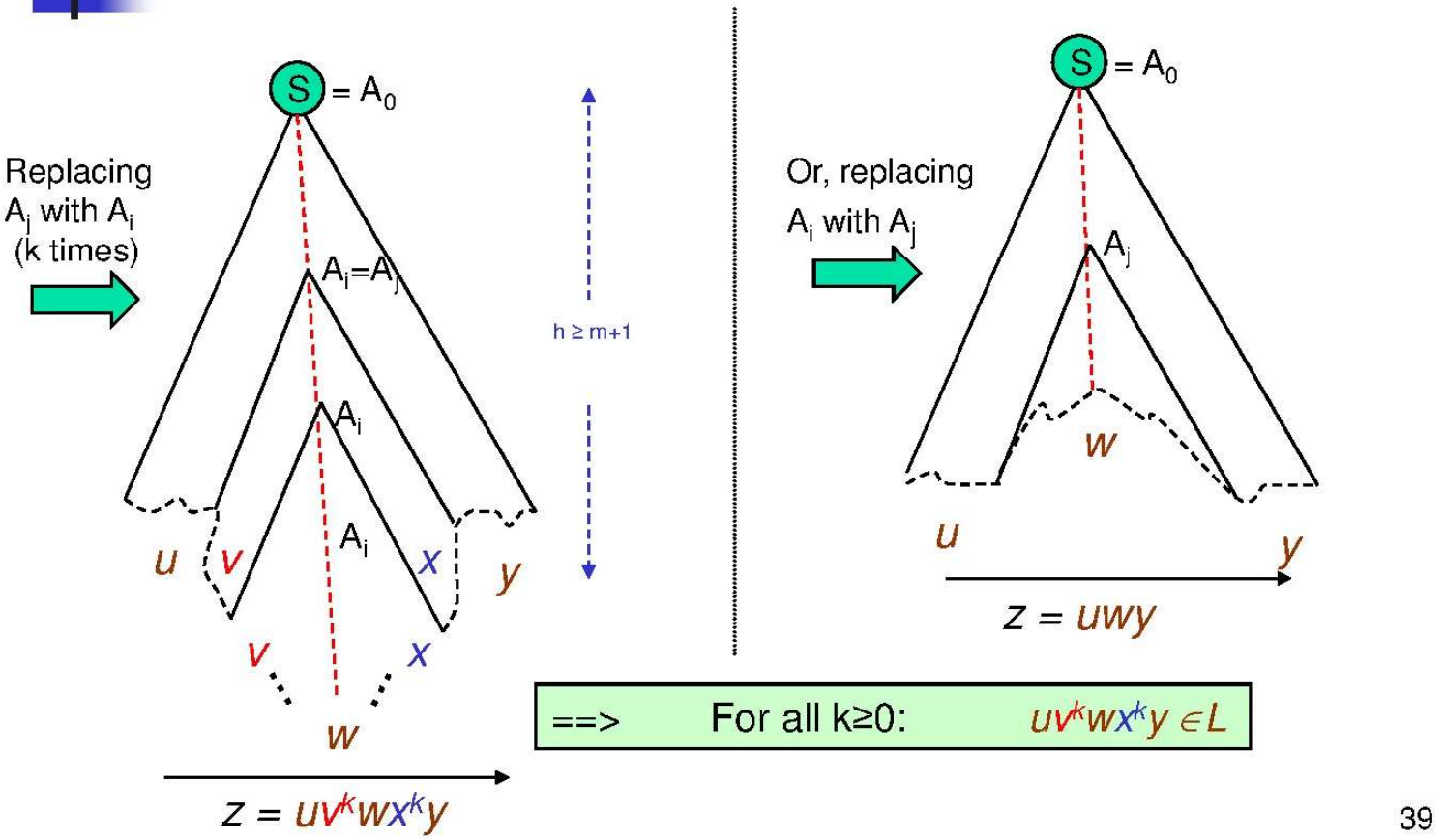
- For any other L which is a CFL:
  - Let G be a CNF grammar for L
  - Let m = number of variables in G
  - Choose $N=2^m$.
  - Pick any $z \in L$ s.t. $|z| \geq N$

    ➔ the parse tree for z should have a height ≥ m+1
    (by the parse tree theorem)

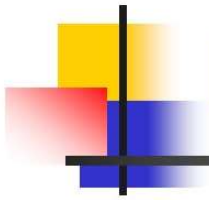
Parse tree for z



Extending the parse tree…



==> For all k≥0: $uv^kwx^ky \in L$



Proof contd..

- Also, since A_i's subtree no taller than m+1

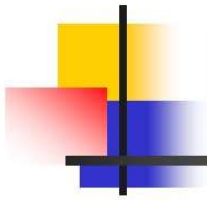
  ==> the string generated under A_i's subtree, which is vwx, cannot be longer than 2^m (=N)

  But, 2^m =N

  ==> |vwx| ≤ N

  This completes the proof for the pumping lemma.

Application of Pumping Lemma for CFLs

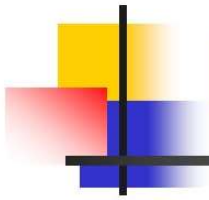


Example 1: $L = \{a^m b^m c^m \mid m>0 \}$

Claim: L is not a CFL

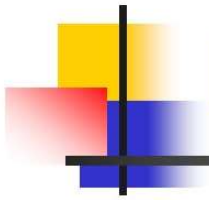
Proof:

- Let N <== P/L constant
- Pick $z = a^N b^N c^N$
- Apply pumping lemma to z and show that there exists at least one other string constructed from z (obtained by pumping up or down) that is $\notin$ L



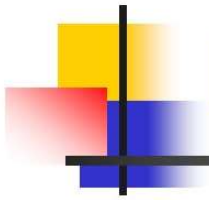
Proof contd…

- z = uvwxy
- As $z = a^N b^N c^N$ *and* $|vwx| \leq N$ *and* $vx \neq \varepsilon$
  - ==> v, x cannot contain all three symbols (a,b,c)
  - ==> we can pump up or pump down to build another string which is $\notin$ L



Example #2 for P/L application

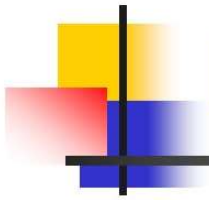
- L = { ww | w is in {0,1}*}
- Show that L is not a CFL
  - Try string z = 0^N0^N
    - what happens?
  - Try string z = $0^N1^N0^N1^N$
    - what happens?



Example 3

- L = { 0^{k^2} | k is any integer)

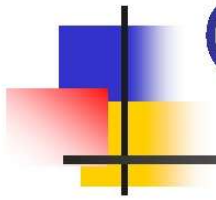
- Prove L is not a CFL using Pumping Lemma



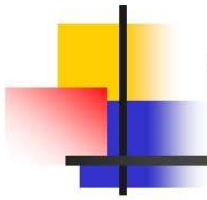
Example 4

- $L = \{a^i b^j c^k \mid i<j<k \}$

- Prove that L is not a CFL



CFL Closure Properties



Closure Property Results

- CFLs are closed under:
  - Union
  - Concatenation
  - Kleene closure operator
  - Substitution
  - Homomorphism, inverse homomorphism
  - reversal
- CFLs are *not* closed under:
  - Intersection
  - Difference
  - Complementation

Note: Reg languages are closed under these operators

Strategy for Closure Property Proofs

- First prove "closure under **substitution**"
- Using the above result, prove other closure properties
- CFLs are closed under:
  - Union
  - Concatenation
  - Kleene closure operator
  - Substitution
  - Homomorphism, inverse homomorphism
  - Reversal

Prove this first

Note: s(L) can use a different alphabet

The ***Substitution*** operation

For each $a \in \sum$, then let s(a) be a language

If $w=a_1a_2\ldots a_n \in L$, then:

- $s(w) = \{ x_1x_2 \ldots \} \in s(L)$, s.t., $x_i \in s(a_i)$

Example:

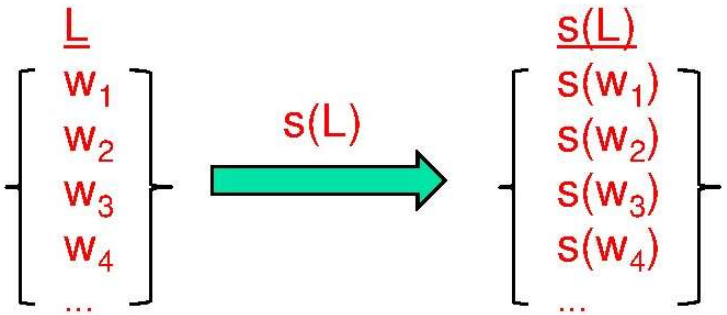
- Let $\sum=\{0,1\}$
- Let: $s(0) = \{a^nb^n \mid n \geq 1\}$, $s(1) = \{aa,bb\}$
- If w=01, s(w)=s(0).s(1)
  - E.g., s(w) contains a^1b^1aa, a^1b^1bb, a^2b^2aa, a^2b^2bb, … and so on.

CFLs are closed under Substitution

IF L is a CFL and a substititution defined on L, s(L), is s.t., s(a) is a CFL for every symbol a, THEN:

- s(L) is also a CFL

What is s(L)?



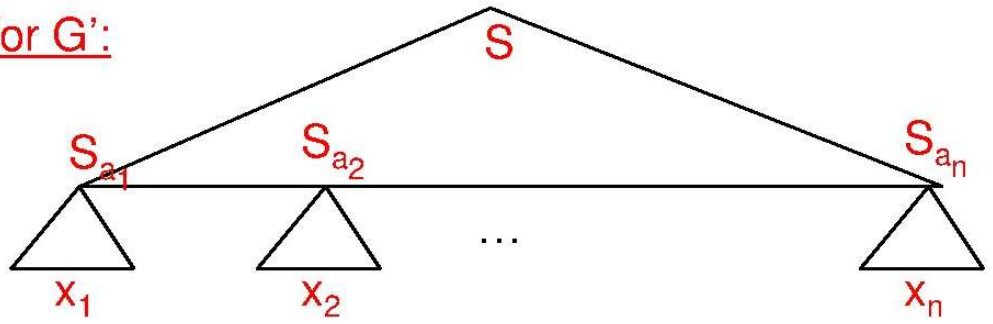
Note: each s(w) is itself a set of strings

CFLs are closed under *Substitution*

- G=(V,T,P,S) : CFG for L
- Because every s(a) is a CFL, there is a CFG for each s(a)
  - Let $G_a = (V_a,T_a,P_a,S_a)$
- Construct G'=(V',T',P',S) for s(L)

- P' consists of:
  - The productions of P, but with every occurrence of terminal "a" in their bodies replaced by S_a.
  - All productions in any P_a, for any a $\in \sum$

Parse tree for G':



Substitution of a CFL: example

- Let L = language of binary palindromes s.t., substitutions for 0 and 1 are defined as follows:
  - $s(0) = \{a^n b^n \mid n \geq 1\}$, $s(1) = \{xx, yy\}$
- Prove that s(L) is also a CFL.

CFG for L:

S=> 0S0|1S1|ε

CFG for s(0):

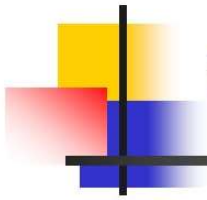
S_0=> aS_0b | ab

CFG for s(1):

S_1=> xx | yy

Therefore, CFG for s(L):

S=> S_0SS_0 | S_1 **S** S_1 |ε
S_0=> aS_0b | ab
S_1=> xx | yy



CFLs are closed under ***union***

Let L_1 and L_2 be CFLs

To show: L_2 U L_2 is also a CFL

Let us show by using the result of *Substitution*

- Make a new language:
  - L_{new} = {a,b} s.t., s(a) = L_1 and s(b) = L_2

  ==> s(L_{new}) == same as == L_1 U L_2

- A more direct, alternative proof
  - Let S_1 and S_2 be the starting variables of the grammars for L_1 and L_2
    - Then, **S_{new} => S_1 | S_2**

CFLs are closed under *concatenation*

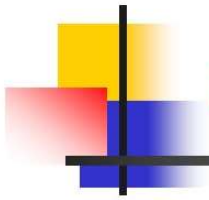
- Let L_1 and L_2 be CFLs

Let us show by using the result of *Substitution*

- Make L_{new}= {ab} s.t.,
  $s(a) = L_1$ and $s(b) = L_2$

  ==> $L_1 L_2 = s(L_{new})$

- A proof without using substitution?

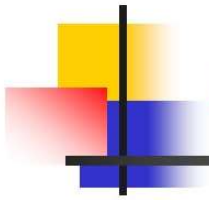


CFLs are closed under *Kleene Closure*

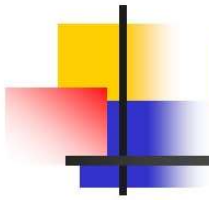
- Let L be a CFL
- Let $L_{new} = \{a\}^*$ and $s(a) = L_1$
  - Then, $L^* = s(L_{new})$

We won't use substitution to prove this result

CFLs are closed under *Reversal*



- Let L be a CFL, with grammar G=(V,T,P,S)
- For L^R, construct $G^R=(V,T,P^R,S)$ s.t.,
  - If A==> α is in P, then:
    - A==> α^R is in P^R
    - (that is, reverse every production)



CFLs are *not* closed under Intersection

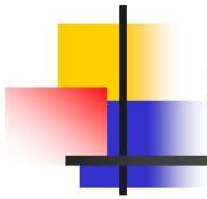
- Existential proof:
  - $L_1 = \{0^n1^n2^i \mid n \geq 1, i \geq 1\}$
  - $L_2 = \{0^i1^n2^n \mid n \geq 1, i \geq 1\}$
- Both L_1 and L_1 are CFLs
  - Grammars?
- But $L_1 \cap L_2$ *cannot* be a CFL
  - Why?
- We have an example, where intersection is not closed.
- Therefore, CFLs are not closed under intersection

CFLs are not closed under complementation

- Follows from the fact that CFLs are not closed under intersection

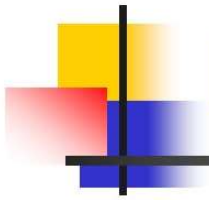
- $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$

Logic: if CFLs were to be closed under complementation
➔ the whole right hand side becomes a CFL (because CFL is closed for union)
➔ the left hand side (intersection) is also a CFL
➔ but we just showed CFLs are NOT closed under intersection!
➔ CFLs *cannot* be closed under complementation.



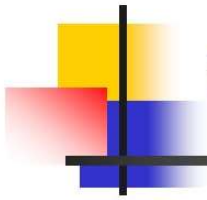
CFLs are not closed under difference

- Follows from the fact that CFLs are not closed under complementation
- Because, if CFLs are closed under difference, then:
  - $\overline{L} = \Sigma^* - L$
  - So $\overline{L}$ has to be a CFL too
  - Contradiction



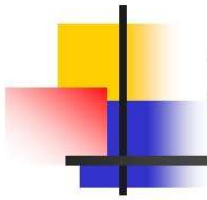
Decision Properties

- Emptiness test
  - Generating test
  - Reachability test
- Membership test
  - PDA acceptance



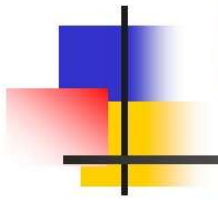
“Undecidable” problems for CFL

- Is a given CFG G ambiguous?
- Is a given CFL inherently ambiguous?
- Is the intersection of two CFLs empty?
- Are two CFLs the same?
- Is a given L(G) equal to $\sum^*$?



Summary

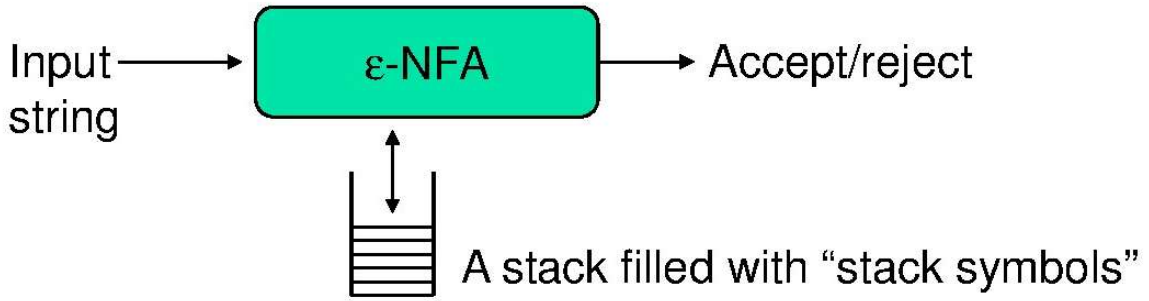
- Normal Forms
  - Chomsky Normal Form
  - Griebach Normal Form
  - Useful in proroving P/L
- Pumping Lemma for CFLs
  - Main difference: $z=uv^iwx^iy$
- Closure properties
  - Closed under: union, concatentation, reversal, Kleen closure, homomorphism, substitution
  - Not closed under: intersection, complementation, difference



Pushdown Automata (PDA)

PDA - the automata for CFLs

- What is?
  - FA to Reg Lang, PDA is to CFL
- PDA == [ε -NFA + "a stack"]
- Why a stack?



Pushdown Automata - Definition

- A PDA P := (Q,$\sum$,Γ, δ,q_0,Z_0,F):
  - Q: states of the ε-NFA
  - $\sum$: input alphabet
  - Γ : stack symbols
  - δ: transition function
  - q_0: start state
  - Z_0: Initial stack top symbol
  - F: Final/accepting states

old state *Stack top* *input symb.* *new state(s)* *new Stack top(s)*

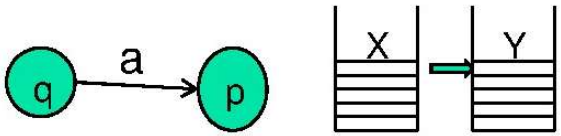
$$\delta : Q \times \Gamma \times \Sigma \Rightarrow Q \times \Gamma$$

δ : The Transition Function

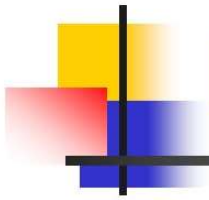
δ(q,a,X) = {(p,Y), …}

1. state transition from q to p
2. a is the next input symbol
3. X is the current stack *top* symbol
4. Y is the replacement for X; it is in Γ^* (a string of stack symbols)
   - i. Set Y = ε for: Pop(X)
   - ii. If Y=X: stack top is unchanged
   - iii. If $Y=Z_1Z_2\ldots Z_k$: X is popped and is replaced by Y in reverse order (i.e., Z_1 will be the new stack top)

Non-determinism



	Y = ?	Action
i)	Y=ε	Pop(X)
ii)	Y=X	Pop(X) Push(X)
iii)	$Y=Z_1Z_2..Z_k$	Pop(X) Push(Z_k) Push(Z_{k-1}) … Push(Z_2) Push(Z_1)



Example

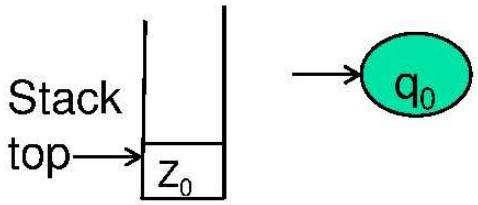
Let $L_{wwr} = \{ww^R \mid w \text{ is in } (0+1)^*\}$

- CFG for L_{wwr} : $S ==> 0S0 \mid 1S1 \mid \varepsilon$
- PDA for L_{wwr} :
- $P := (Q, \sum, \Gamma, \delta, q_0, Z_0, F)$

$= (\{q_0, q_1, q_2\}, \{0,1\}, \{0,1,Z_0\}, \delta, q_0, Z_0, \{q_2\})$

PDA for L_{wwr}

Initial state of the PDA:



1. $\delta(q_0,0, Z_0)=\{(q_0,0Z_0)\}$
2. $\delta(q_0,1, Z_0)=\{(q_0,1Z_0)\}$

} First symbol push on stack

3. $\delta(q_0,0, 0)=\{(q_0,00)\}$
4. $\delta(q_0,0, 1)=\{(q_0,01)\}$
5. $\delta(q_0,1, 0)=\{(q_0,10)\}$
6. $\delta(q_0,1, 1)=\{(q_0,11)\}$

} Grow the stack by pushing new symbols on top of old (w-part)

7. $\delta(q_0, \varepsilon, 0)=\{(q_1, 0)\}$
8. $\delta(q_0, \varepsilon, 1)=\{(q_1, 1)\}$
9. $\delta(q_0, \varepsilon, Z_0)=\{(q_1, Z_0)\}$

} Switch to popping mode (boundary between w and w^R)

10. $\delta(q_1,0, 0)=\{(q_1, \varepsilon)\}$
11. $\delta(q_1,1, 1)=\{(q_1, \varepsilon)\}$

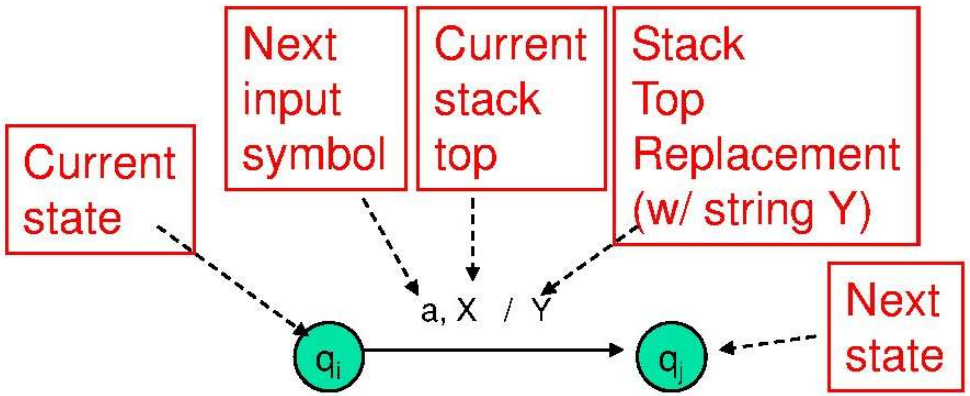
} Shrink the stack by popping matching symbols (w^R-part)

12. $\delta(q_1, \varepsilon, Z_0)=\{(q_2, Z_0)\}$

} Enter acceptance state

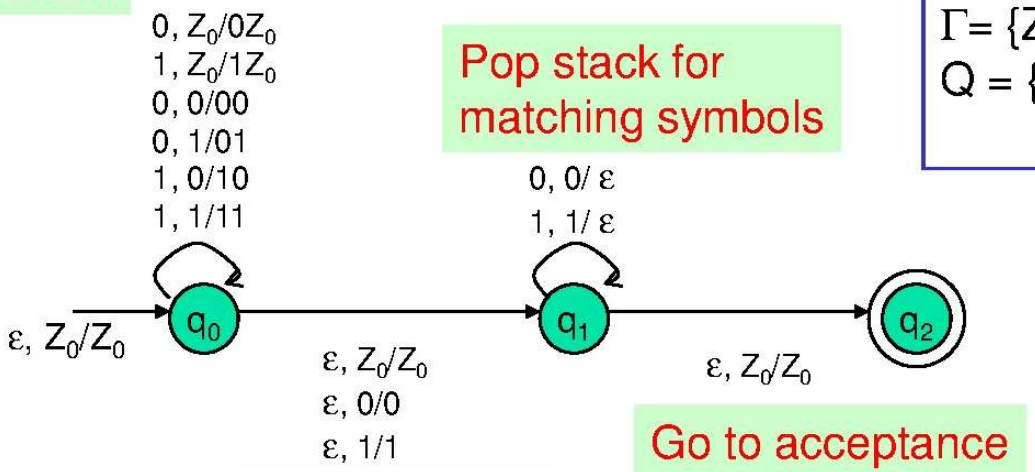
PDA as a state diagram

$\delta(q_i, a, X)=\{(q_j, Y)\}$



PDA for L_{wwr}: Transition Diagram

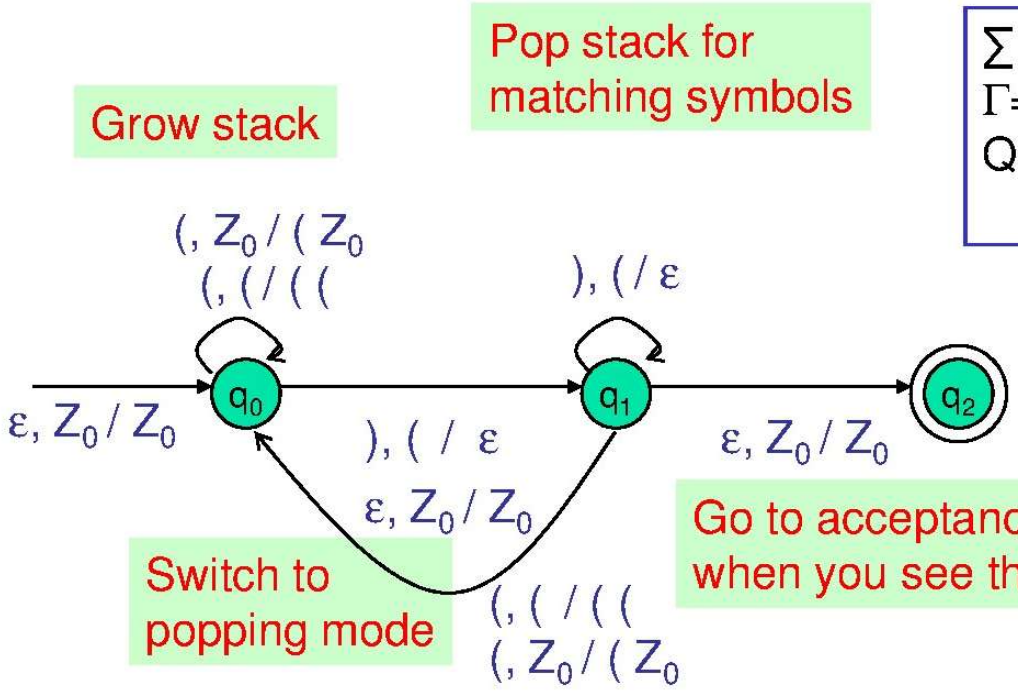
Grow stack



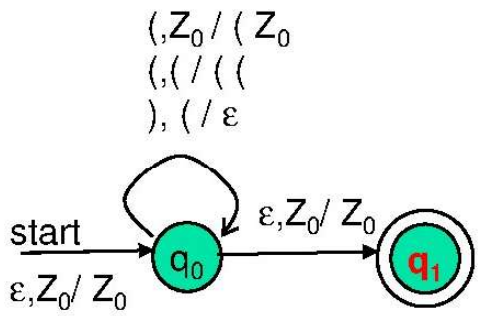
$\Sigma = \{0, 1\}$
$\Gamma = \{Z_0, 0, 1\}$
$Q = \{q_0, q_1, q_2\}$

This would be a non-deterministic PDA

Example 2: language of balanced paranthesis

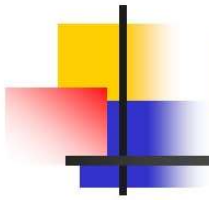


Example 2: language of balanced paranthesis (another design)



$\Sigma = \{ (,) \}$
$\Gamma = \{Z_0, (\}$
$Q = \{q_0, q_1\}$

PDA's Instantaneous Description (ID)



A PDA has a configuration at any given instance:

(q,w,y)

- q - current state
- w - remainder of the input (i.e., unconsumed part)
- y - current stack contents as a string from top to bottom of stack

---

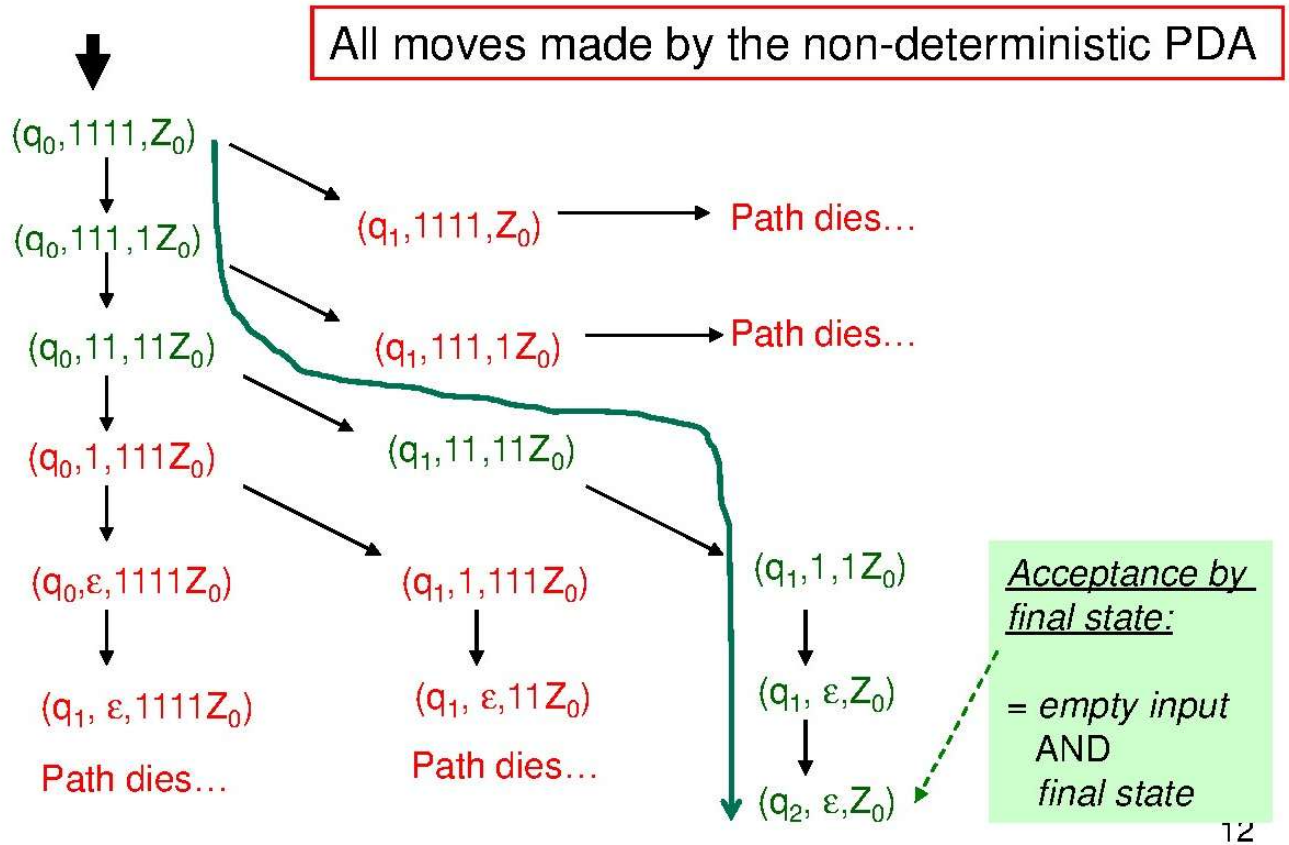
If $\delta(q,a, X)=\{(p, A)\}$ is a transition, then the following are also true:

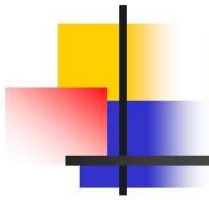
- $(q, a, X) \vdash (p,\varepsilon,A)$
- $(q, aw, XB) \vdash (p,w,AB)$

|--- sign is called a "turnstile notation" and represents one move

|---* sign represents a sequence of moves

How does the PDA for L_{wwr} work on input "1111"?





Principles about IDs

- Theorem 1: If for a PDA, (q, x, A) |---* (p, y, B), then for any string $w \in \Sigma^*$ and $\gamma \in \Gamma^*$, it is also true that:
  - (q, x w, A γ) |---* (p, y w, B γ)

- Theorem 2: If for a PDA, (q, x w, A) |---* (p, y w, B), then it is also true that:
  - (q, x, A) |---* (p, y, B)

There are two types of PDAs that one can design:
those that accept by final state or by empty stack

Acceptance by…

- *PDAs that accept by **final state**:*
  - For a PDA P, the language accepted by P, denoted by L(P) by *final state*, is:
    - $\{w \mid (q_0,w,Z_0) \vdash^* (q,\varepsilon, A)\}$, s.t., $q \in F$

Checklist:
- input exhausted?
- in a final state?

- *PDAs that accept by **empty stack**:*
  - For a PDA P, the language accepted by P, denoted by N(P) by *empty stack*, is:
    - $\{w \mid (q_0,w,Z_0) \vdash^* (q, \varepsilon, \varepsilon)\}$, for any $q \in Q$.

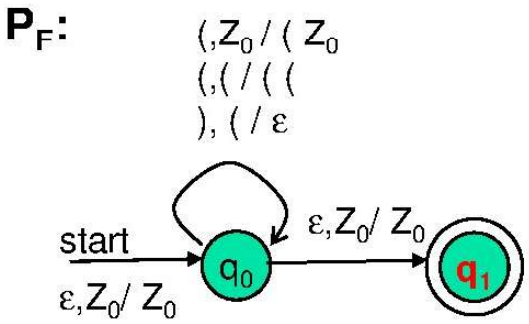
Q) Does a PDA that accepts by empty stack need any final state specified in the design?

Checklist:
- input exhausted?
- is the stack empty?

Example: L of balanced parenthesis

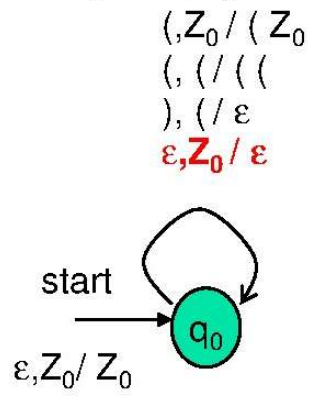
PDA that accepts by final state

P_F:



An equivalent PDA that accepts by empty stack

P_N:



How will these two PDAs work on the input: ((( )) ( )) ( )

PDA for L_{wwr}: Proof of correctness

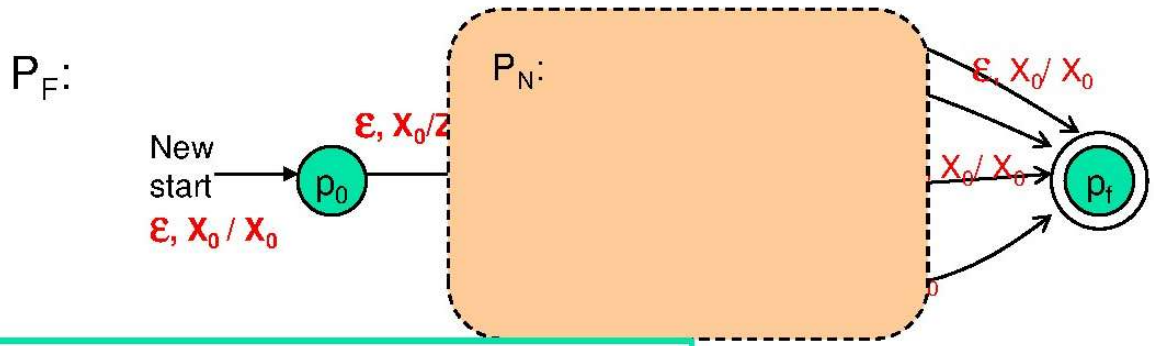
- Theorem: The PDA for L_{wwr} accepts a string x by final state if and only if x is of the form ww^R.
- Proof:
  - *(if-part)* If the string is of the form ww^R then there exists a sequence of IDs that leads to a final state: (q_0,ww^R,Z_0) |---* (q_0,w^R,wZ_0) |---* (q_1,w^R,wZ_0) |---* (q_1, ε,Z_0) |---* (q_2, ε,Z_0)
  - *(only-if part)*
    - Proof by induction on |x|

PDAs accepting by final state and empty stack are equivalent

- P_F <= PDA accepting by final state
  - $P_F = (Q_F, \Sigma, \Gamma, \delta_F, q_0, Z_0, F)$
- P_N <= PDA accepting by empty stack
  - $P_N = (Q_N, \Sigma, \Gamma, \delta_N, q_0, Z_0)$
- Theorem:
  - *(P_N ==> P_F)* For every P_N, there exists a P_F s.t. $L(P_F)=L(P_N)$
  - *(P_F ==> P_N)* For every P_F, there exists a P_N s.t. $L(P_F)=L(P_N)$

P_N==> P_F construction

- Whenever P_N's stack becomes empty, make P_F go to a final state without consuming any addition symbol
- To detect empty stack in P_N: P_F pushes a new stack symbol X_0 (not in Γ of P_N) initially before simultating P_N



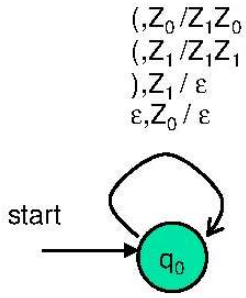
$P_F = (Q_N \cup \{p_0, p_f\}, \Sigma, \Gamma \cup \{X_0\}, \delta_F, p_0, X_0, \{p_f\})$

Example: Matching parenthesis "(" ")"

P_N: $(\{q_0\}, \{(,)\}, \{Z_0,Z_1\}, \delta_N, q_0, Z_0)$

δ_N:

$\delta_N(q_0,(,Z_0) = \{ (q_0,Z_1Z_0) \}$
$\delta_N(q_0,(,Z_1) = \{ (q_0, Z_1Z_1) \}$
$\delta_N(q_0,),Z_1) = \{ (q_0, \varepsilon) \}$
$\delta_N(q_0, \varepsilon,Z_0) = \{ (q_0, \varepsilon) \}$

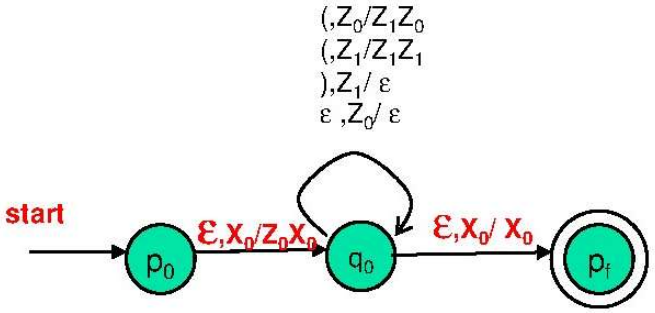


Accept by empty stack

P_f: $(\{p_0,q_0,p_f\}, \{(,)\}, \{X_0,Z_0,Z_1\}, \delta_f, p_0, X_0, p_f)$

δ_f:

$\delta_f(p_0, \varepsilon,X_0) = \{ (q_0,Z_0) \}$
$\delta_f(q_0,(,Z_0) = \{ (q_0,Z_1 Z_0) \}$
$\delta_f(q_0,(,Z_1) = \{ (q_0, Z_1Z_1) \}$
$\delta_f(q_0,),Z_1) = \{ (q_0, \varepsilon) \}$
$\delta_f(q_0, \varepsilon,Z_0) = \{ (q_0, \varepsilon) \}$
$\delta_f(p_0, \varepsilon,X_0) = \{ (p_f, X_0) \}$



Accept by final state

How to convert an final state PDA into an empty stack PDA?

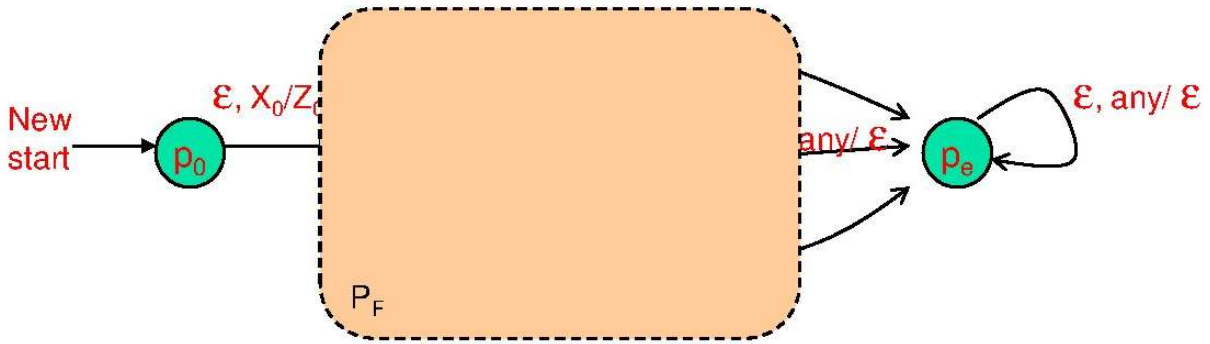
P_F==> P_N construction

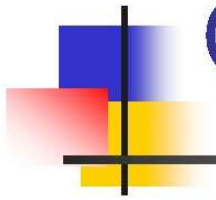
- Main idea:
  - Whenever P_F reaches a final state, just make an ε-transition into a new end state, clear out the stack and accept
  - Danger: What if P_F design is such that it clears the stack midway *without* entering a final state?

    ➔ to address this, add a new start symbol X_0 (not in Γ of P_F)

$P_N = (Q \cup \{p_0, p_e\}, \Sigma, \Gamma \cup \{X_0\}, \delta_N, p_0, X_0)$

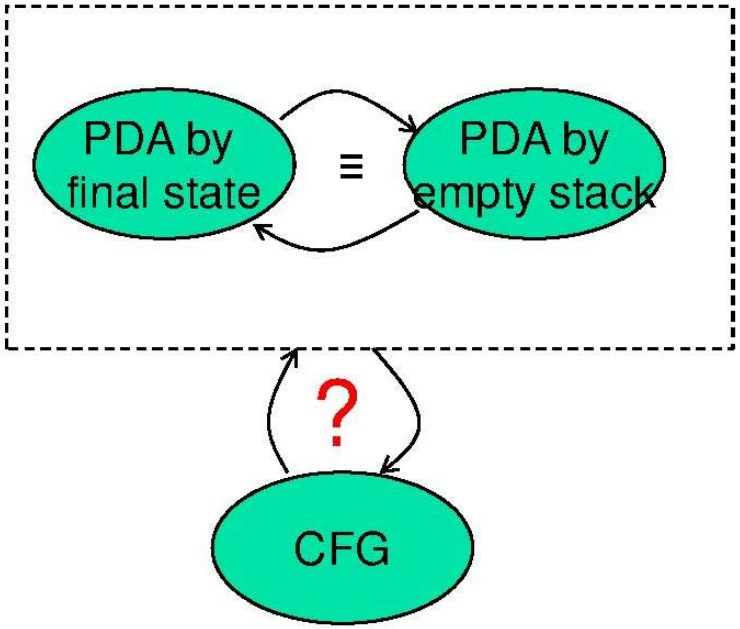
P_N:





Equivalence of PDAs and CFGs

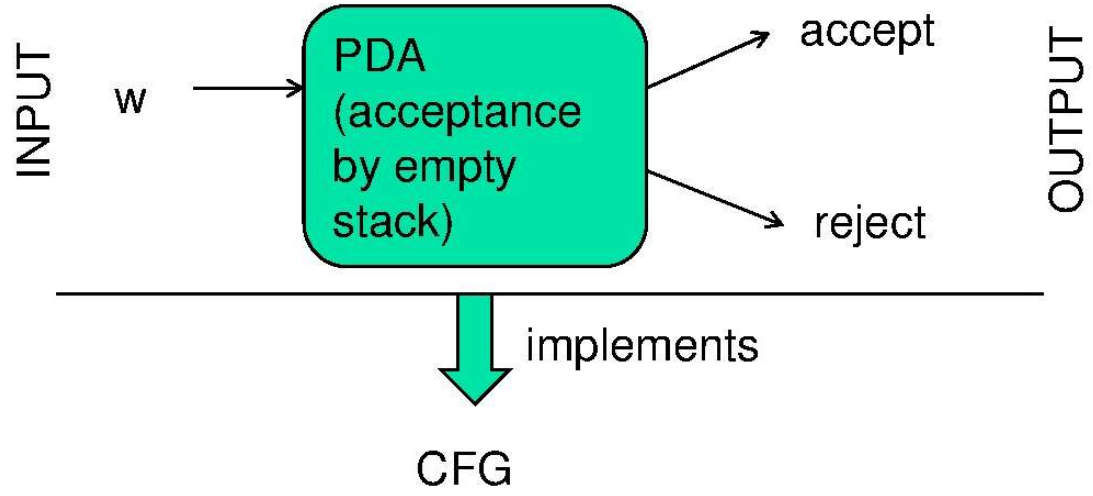
CFGs == PDAs ==> CFLs



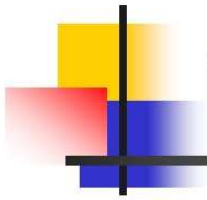
This is same as: “implementing a CFG using a PDA”

Converting CFG to PDA

Main idea: The PDA simulates the leftmost derivation on a given w, and upon consuming it fully it either arrives at acceptance (by empty stack) or non-acceptance.



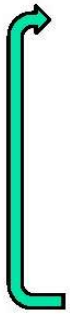
This is same as: “implementing a CFG using a PDA”



Converting a CFG into a PDA

Main idea: The PDA simulates the leftmost derivation on a given w, and upon consuming it fully it either arrives at acceptance (by empty stack) or non-acceptance.

Steps:



1. Push the right hand side of the production onto the stack, with leftmost symbol at the stack top
2. If stack top is the leftmost variable, then replace it by all its productions (each possible substitution will represent a *distinct* path taken by the non-deterministic PDA)
3. If stack top has a terminal symbol, and if it matches with the next symbol in the input string, then pop it

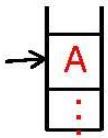
State is inconsequential (only one state is needed)

Formal construction of PDA from CFG

Note: Initial stack symbol (S) same as the start variable in the grammar

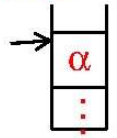
- Given: G= (V,T,P,S)
- Output: $P_N = (\{q\}, T, V \cup T, \delta, q, S)$
- δ:

Before:

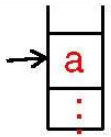


- For all $A \in V$, add the following transition(s) in the PDA:
  - $\delta(q, \varepsilon, A) = \{ (q, \alpha) \mid \text{“}A ==> \alpha\text{”} \in P\}$

After:

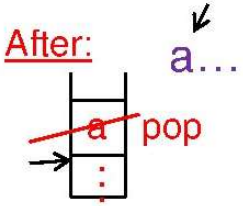


Before:



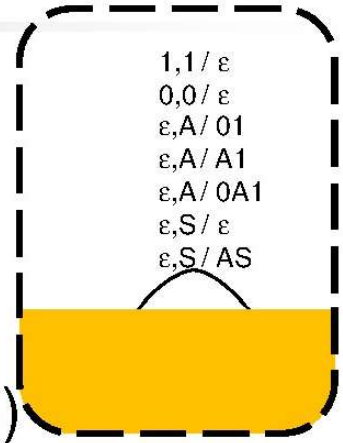
- For all *a* $\in$ T, add the following transition(s) in the PDA:
  - $\delta(q,a,a) = \{ (q, \varepsilon) \}$

After:



Example: CFG to PDA

- G = ({S,A}, {0,1}, P, S)
- P:
  - S ==> AS | ε
  - A ==> 0A1 | A1 | 01
- PDA = ({q}, {0,1}, {0,1,A,S}, δ, q, S)
- δ:
  - δ(q, ε , S) = { (q, AS), (q, ε)}
  - δ(q, ε , A) = { (q,0A1), (q,A1), (q,01) }
  - δ(q, 0, 0) = { (q, ε) }
  - δ(q, 1, 1) = { (q, ε) }



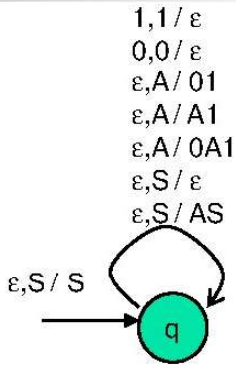
How will this new PDA work?

Lets simulate string 0011

Simulating string 0011 on the new PDA …

PDA (δ):

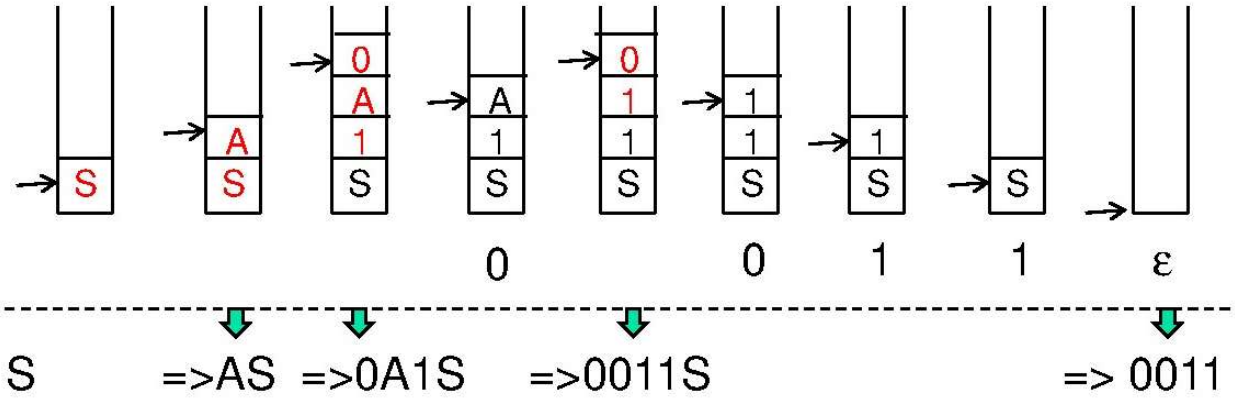
δ(q, ε , S) = { (q, AS), (q, ε)}
δ(q, ε , A) = { (q,0A1), (q,A1), (q,01) }
δ(q, 0, 0) = { (q, ε) }
δ(q, 1, 1) = { (q, ε) }



Leftmost deriv.:

S => AS
=> 0A1S
=> 0011S
=> 0011

Stack moves (shows only the successful path):



Accept by empty stack

Proof of correctness for CFG ==> PDA construction

- Claim: A string is accepted by G iff it is accepted (by empty stack) by the PDA
- Proof:
  - (*only-if part*)
    - Prove by induction on the number of derivation steps
  - (*if part*)
    - If $(q, wx, S) \vdash^{*} (q,x,B)$ then $S \Rightarrow^{*}_{lm} wB$

Converting a PDA into a CFG

- Main idea: Reverse engineer the productions from transitions

If $\delta(q,a,Z) \Rightarrow (p, Y_1Y_2Y_3\ldots Y_k)$:

1. State is changed from q to p;
2. Terminal *a* is consumed;
3. Stack top symbol Z is popped and replaced with a sequence of k variables.

- Action: Create a grammar variable called "[qZp]" which includes the following production:
  - $[qZp] \Rightarrow a[pY_1q_1]\,[q_1Y_2q_2]\,[q_2Y_3q_3]\ldots\,[q_{k-1}Y_kq_k]$

- Proof discussion (in the book)

Example: Bracket matching

- To avoid confusion, we will use *b*="(" and *e*=")"

P_N: ($\{q_0\}$, $\{b,e\}$, $\{Z_0,Z_1\}$, δ, q_0, Z_0)

1. $\delta(q_0,b,Z_0) = \{ (q_0,Z_1Z_0) \}$
2. $\delta(q_0,b,Z_1) = \{ (q_0,Z_1Z_1) \}$
3. $\delta(q_0,e,Z_1) = \{ (q_0, \varepsilon) \}$
4. $\delta(q_0, \varepsilon, Z_0) = \{ (q_0, \varepsilon) \}$

0. $S \Rightarrow [q_0Z_0q_0]$
1. $[q_0Z_0q_0] \Rightarrow b\ [q_0Z_1q_0]\ [q_0Z_0q_0]$
2. $[q_0Z_1q_0] \Rightarrow b\ [q_0Z_1q_0]\ [q_0Z_1q_0]$
3. $[q_0Z_1q_0] \Rightarrow e$
4. $[q_0Z_0q_0] \Rightarrow \varepsilon$

Let $A=[q_0Z_0q_0]$
Let $B=[q_0Z_1q_0]$

0. $S \Rightarrow A$
1. $A \Rightarrow b\ B\ A$
2. $B \Rightarrow b\ B\ B$
3. $B \Rightarrow e$
4. $A \Rightarrow \varepsilon$

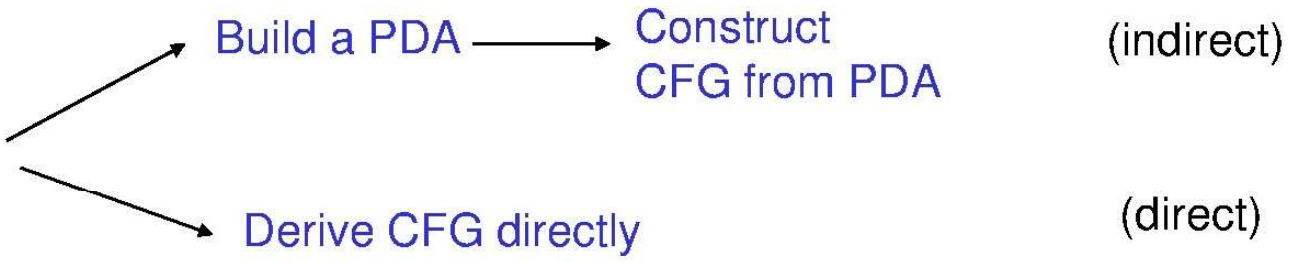
Simplifying,

0. $S \Rightarrow b\ B\ S \mid \varepsilon$
1. $B \Rightarrow b\ B\ B \mid e$

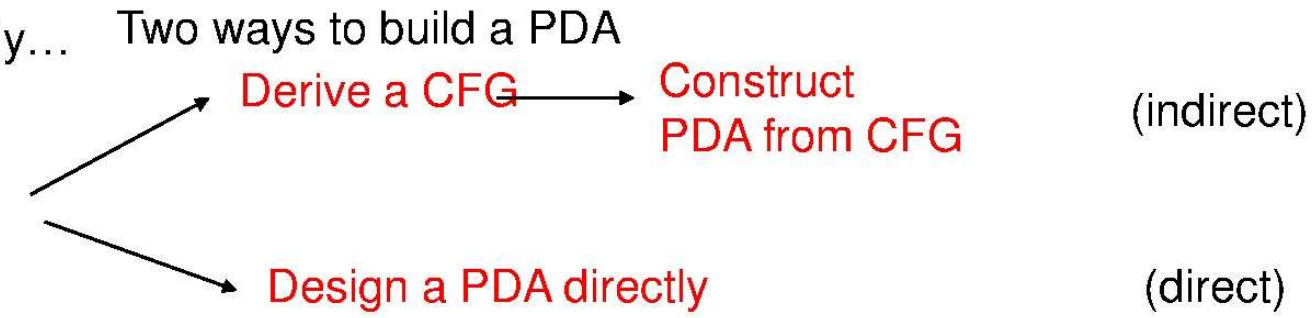
If you were to directly write a CFG:

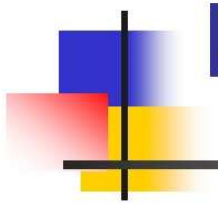
$S \Rightarrow b\ S\ e\ S \mid \varepsilon$

Two ways to build a CFG



Similarly…

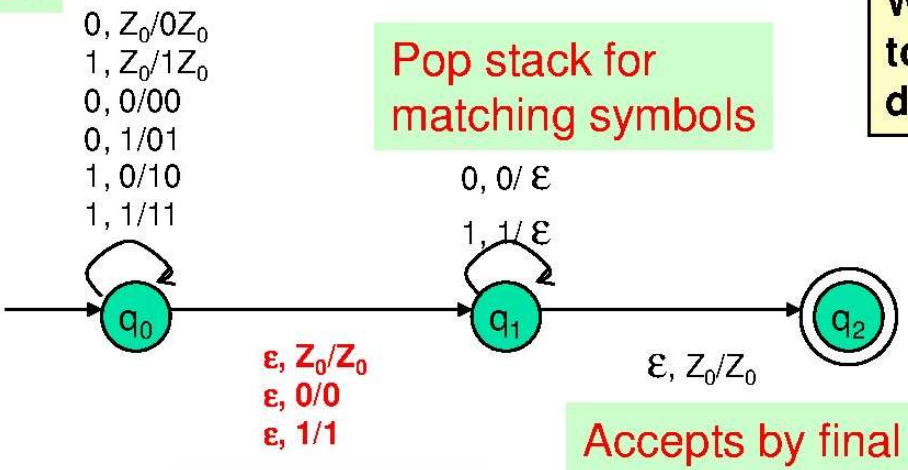




Deterministic PDAs

This PDA for L_{wwr} is non-deterministic

Grow stack



Pop stack for matching symbols

Why does it have to be non-deterministic?

Switch to popping mode

Accepts by final state

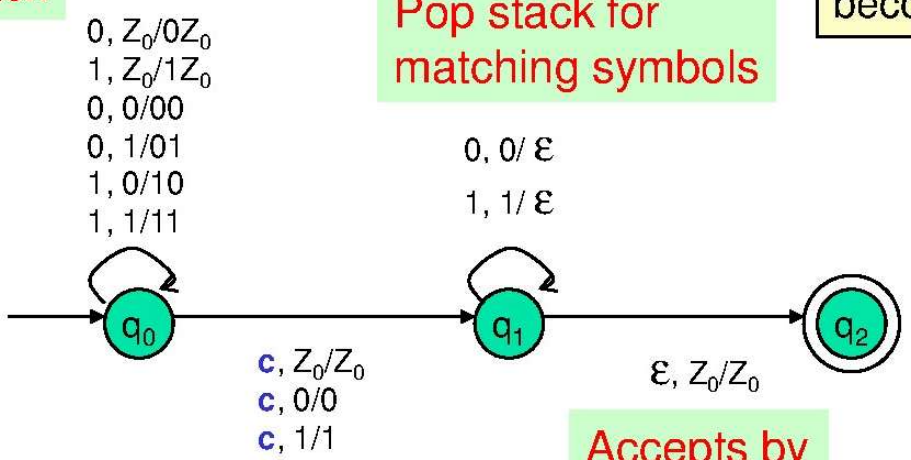
To remove guessing, impose the user to insert c in the middle

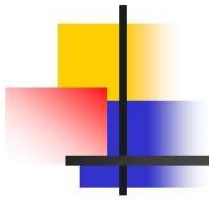
Example shows that: Nondeterministic PDAs ≠ D-PDAs

D-PDA for L_{wcwr} = {wcw^R | c is some special symbol not in w}

Note:
• all transitions have become deterministic

Grow stack



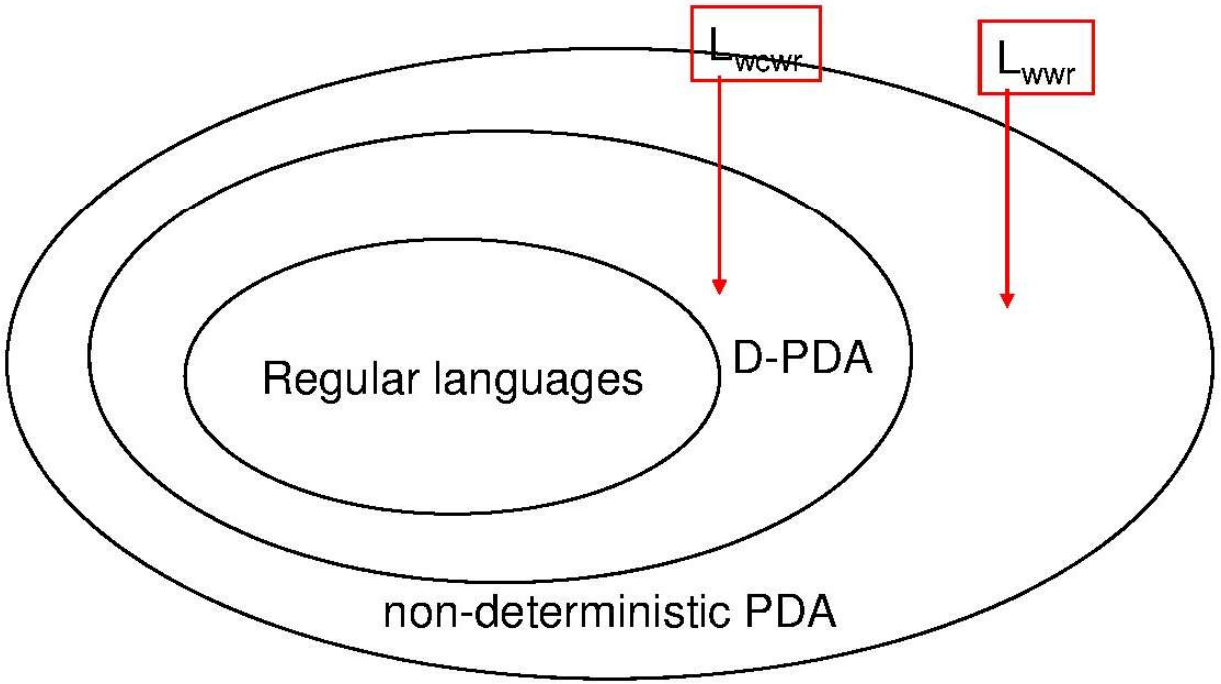


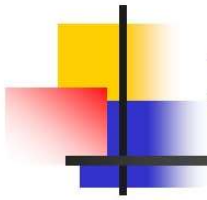
Deterministic PDA: Definition

- A PDA is *deterministic* if and only if:
  1. $\delta(q,a,X)$ has *at most one* member for any $a \in \Sigma \cup \{\varepsilon\}$

➔ If $\delta(q,a,X)$ is non-empty for some $a \in \Sigma$, then $\delta(q, \varepsilon, X)$ must be empty.

PDA vs DPDA vs Regular languages





Summary

- PDAs for CFLs and CFGs
  - Non-deterministic
  - Deterministic
- PDA acceptance types
  1. By final state
  2. By empty stack
- PDA
  - IDs, Transition diagram
- Equivalence of CFG and PDA
  - CFG => PDA construction
  - PDA => CFG construction