Properties of Context-free Languages

Reading: Chapter 7

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 ~~=>~~ ε

3. Eliminate *unit productions* — A ~~=>~~ 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

- <u>Given:</u> G=(V,T,P,S)
- <u>Basis:</u>
  - S is obviously reachable (from itself)
- <u>Induction:</u>
  - 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 ➔ *ε*

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 ➔ ε*

- 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

- <u>Basis:</u>
  - If A→ ε is a production in G, then A is nullable
    (note: A can still have other productions)
- <u>Induction:</u>
  - 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➔$X_1X_2\ldots X_k$, where k≥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➔ε, then remove it

Example: Eliminating ε-productions

- Let L be the language represented by the following CFG G:
  1. S → AB
  2. A → aAA | ε
  3. B → bBB | ε

Simplified grammar

<u>Goal:</u> 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
- <u>Note:</u> $L(G) = L(G_1) \cup \{\varepsilon\}$

<u>G_1:</u>
- 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

What's the point of removing unit transitions ?

Will save #substitutions

E.g.,

before		after
A=>B | …	→	A=>xxx | yyy | zzz | …
B=>C | …		B=> xxx | yyy | zzz | …
C=>D | …		C=> xxx | yyy | zzz | …
D=>xxx | yyy | zzz		D=>xxx | yyy | zzz

A → 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 $A \rightarrow \alpha$, for every $B \rightarrow \alpha$ 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}

<u>Definition:</u>

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:

$B_2 \leftrightarrow C_2$ and so on…

$B_1 \leftrightarrow C_1$

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

Example

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

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 => 0
X_1 => 1

S => AS | BY_1 | **ε**

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

Other Normal Forms

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

$$A \Longrightarrow a\ \alpha$$

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"

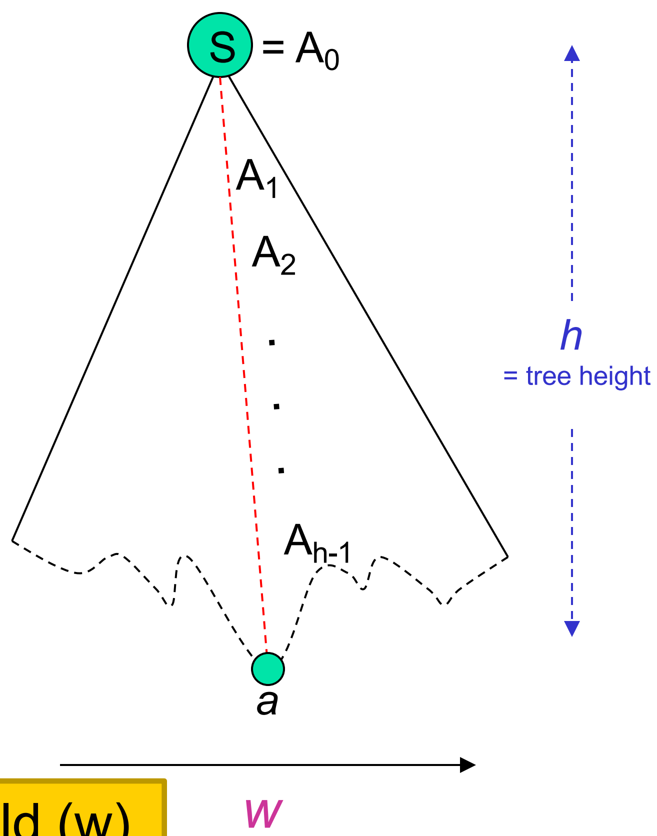
<u>Given:</u>

- 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

<u>Implies:</u>

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

Parse tree for w



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

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

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

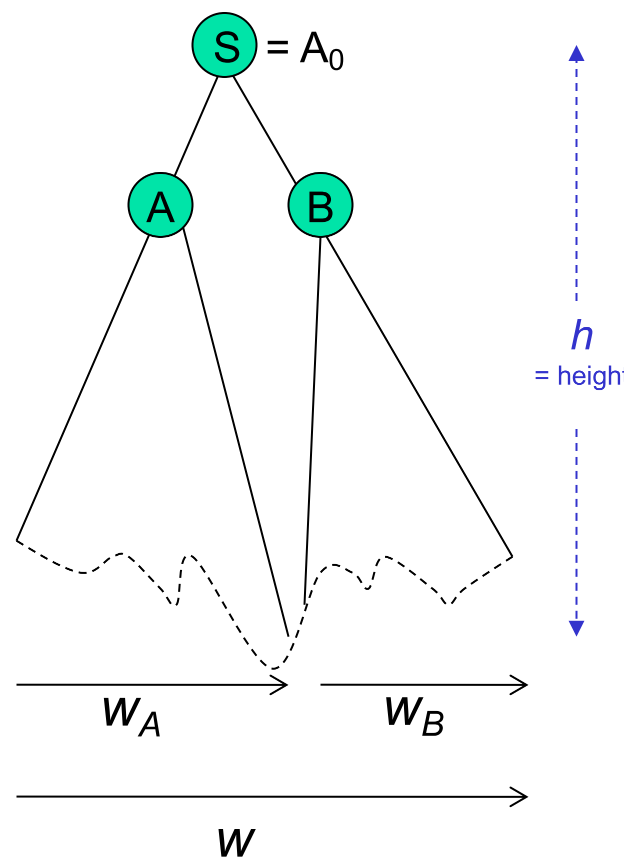
- ➔ $|w| \le 2^{k-2}$

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 $|w_A| \le 2^{k-2}$ and $|w_B| \le 2^{k-2}$
- ➔ $|w| \le 2^{k-1}$

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 $|w| \geq 2^m$, then**
  - **Its parse tree's height is *at least* m+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 $k \geq 0$: $uv^kwx^ky \in L$

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

Proof: Pumping Lemma for CFL

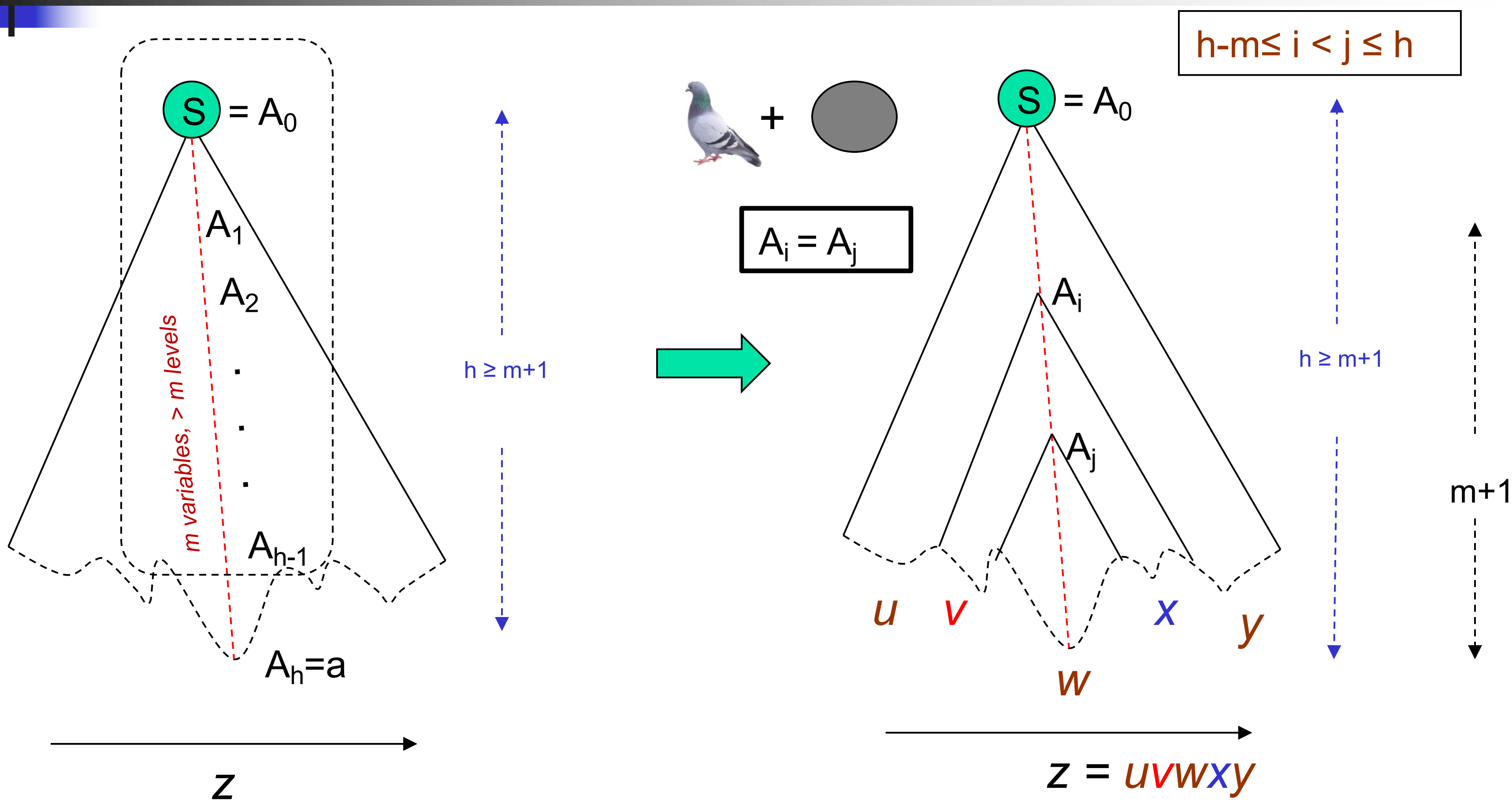
- If L=Φ or contains only ε, then the lemma is trivially satisfied (as it cannot be violated)

- 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

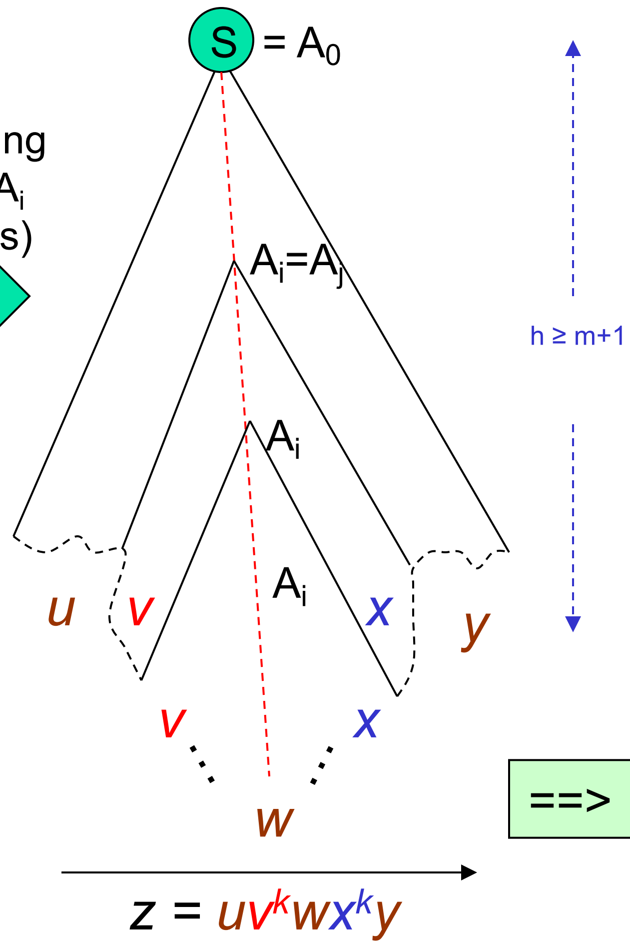
Meaning:
Repetition in the last m+1 variables



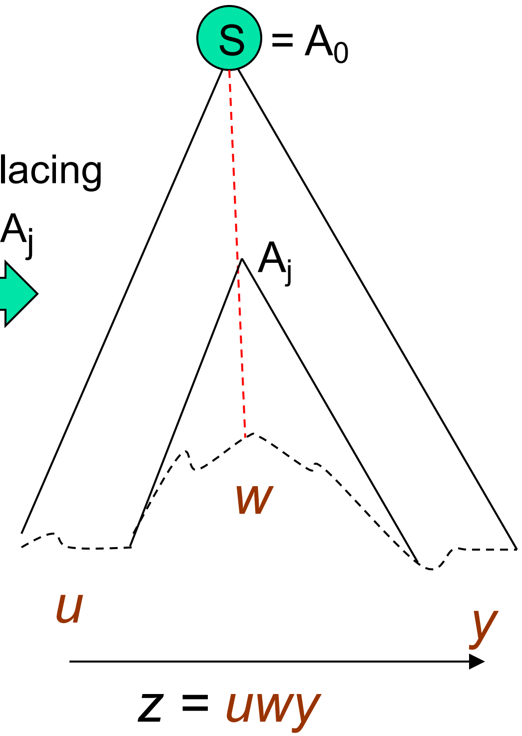
- Therefore, $vx \neq \varepsilon$

Extending the parse tree…

Replacing A_j with A_i (k times)



Or, replacing A_i with A_j



==> For all k≥0: $uv^k wx^k y \in L$

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 $\notin L$

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 $a \in \Sigma$, 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., $x_i \in s(a_i)$

<u>Example:</u>

- Let $\Sigma=\{0,1\}$
- 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 $a^1 b^1 aa$, $a^1 b^1 bb$, $a^2 b^2 aa$, $a^2 b^2 bb$, … 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)?

L		s(L)
w_1		$s(w_1)$
w_2	s(L) →	$s(w_2)$
w_3		$s(w_3)$
w_4		$s(w_4)$
…		…

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

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:

$S \Rightarrow 0S0 \mid 1S1 \mid \varepsilon$

CFG for s(0):

$S_0 \Rightarrow aS_0b \mid ab$

CFG for s(1):

$S_1 \Rightarrow xx \mid yy$

Therefore, CFG for s(L):

$S \Rightarrow S_0SS_0 \mid S_1 S S_1 \mid \varepsilon$
$S_0 \Rightarrow aS_0b \mid ab$
$S_1 \Rightarrow xx \mid yy$

CFLs are closed under ***union***

Let L_1 and L_2 be CFLs

<u>To show:</u> L_2 U L_2 is also a CFL

<u>Let us show by using the result of *Substitution*</u>

- 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

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

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

- <u>Existential proof:</u>
  - $L_1 = \{0^n 1^n 2^i \mid n \geq 1, i \geq 1\}$
  - $L_2 = \{0^i 1^n 2^n \mid n \geq 1, i \geq 1\}$
- Both L_1 and L_2 are CFLs
- But $L_1 \cap L_2$ *cannot* be a CFL
- 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 Σ^*?

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