Unit 4
Context-Free Languages Grammars (CFLs & CFGs)

By Prashant Gautam

Context-Free Languages

$\{a^n b^n : n \geq 0\}$ $\{ww^R\}$

Regular Languages

$a^* b^*$ $(a+b)^*$

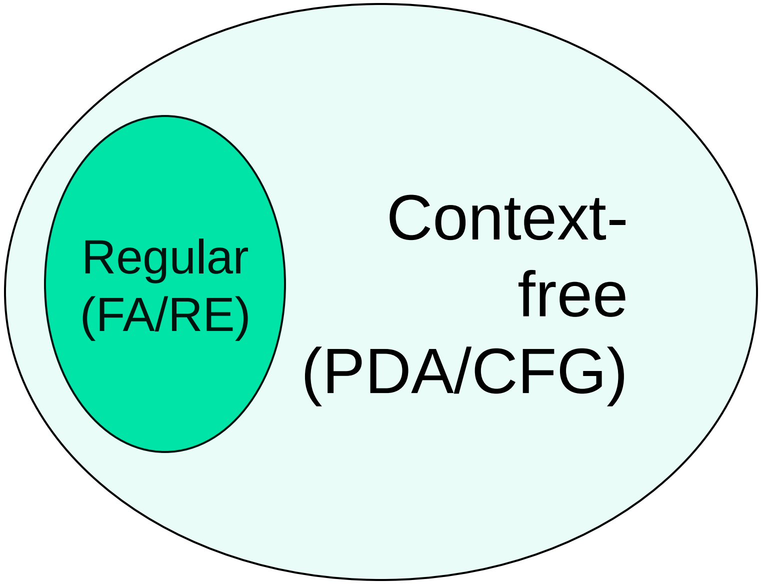
Not all languages are regular

- So what happens to the languages which are not regular?

- Can we still come up with a language recognizer?
  - i.e., something that will accept (or reject) strings that belong (or do not belong) to the language?

Context-Free Languages

- A language class larger than the class of regular languages
- Supports natural, recursive notation called "context-free grammar"
- Applications:
  - Parse trees, compilers
  - XML



An Example

- A palindrome is a word that reads identical from both ends
  - E.g., madam, redivider, malayalam, 010010010
- Let L = { w | w is a binary palindrome}
- Is L regular?
  - No.

But the language of palindromes…

is a CFL, because it supports recursive substitution (in the form of a CFG)

- This is because we can construct a "*grammar*" like this:

1. A ==> ε
2. A ==> 0
3. A ==> 1
4. A ==> 0A0
5. A ==> 1A1

Productions

Terminal

Variable or non-terminal

Same as:
A => 0A0 | 1A1 | 0 | 1 | ε

How does this grammar work?

How does the CFG for palindromes work?

An input string belongs to the language (i.e., accepted) iff it can be generated by the CFG

G:
A → 0A0 | 1A1 | 0 | 1 | ε

- Example: w=01110
- G can generate w as follows:

1. A => 0A0
2. => 01A10
3. => 01110

Generating a string from a grammar:

1. Pick and choose a sequence of productions that would allow us to generate the string.
2. At every step, substitute one variable with one of its productions.

Context-Free Grammar: Definition

- A context-free grammar G=(V,T,P,S), where:
  - V: set of variables or non-terminals
  - T: set of terminals (= alphabet U {ε})
  - P: set of *productions,* each of which is of the form
    V ==> α_1 | α_2 | …
    - Where each α_i is an arbitrary string of variables and terminals
  - S ==> start variable

CFG for the language of binary palindromes:
G=({A},{0,1},P,A)
P: A ==> 0 A 0 | 1 A 1 | 0 | 1 | ε

More examples

- Parenthesis matching in code
- Syntax checking
- In scenarios where there is a general need for:
  - Matching a symbol with another symbol, or
  - Matching a count of one symbol with that of another symbol, or
  - Recursively substituting one symbol with a string of other symbols

Example #2

- Language of balanced paranthesis

  e.g., ()((((())))((())))….
- CFG?

G:
S => (S) | SS | ε

How would you “interpret” the string “(((()))()())” using this grammar?

Example #3

- A grammar for $L = \{0^m 1^n \mid m \geq n\}$

- CFG?

G:
$S \Rightarrow 0S1 \mid A$
$A \Rightarrow 0A \mid \varepsilon$

How would you interpret the string "00000111" using this grammar?

Example #4

A program containing **if-then(-else)** statements

if *Condition* **then** *Statement* **else** *Statement*

(Or)

if *Condition* **then** *Statement*

CFG?

$$stmt \rightarrow \textbf{if} \ expr \ \textbf{then} \ stmt$$

$$stmt \rightarrow \textbf{if} \ expr \ \textbf{then} \ stmt \ \textbf{else} \ stmt$$

$$stmt \rightarrow other\text{-}stmt$$

Applications of CFLs & CFGs

- Compilers use parsers for syntactic checking
- Parsers can be expressed as CFGs
  1. Balancing paranthesis:
     - B ==> BB | (B) | *Statement*
     - *Statement* ==> …
  2. If-then-else:
     - S ==> SS | *if Condition then Statement else Statement* | *if Condition then Statement* | *Statement*
     - *Condition* ==> …
     - *Statement* ==> …
  3. C paranthesis matching { … }
  4. Pascal *begin-end* matching
  5. YACC (Yet Another Compiler-Compiler)

More applications

- Markup languages
  - Nested Tag Matching
    - HTML
      - <html> …<p> … … </p> … </html>
    - XML
      - <PC> … <MODEL> … </MODEL> .. <RAM> … </RAM> … </PC>

Tag-Markup Languages

Roll ==> <ROLL> Class Students </ROLL>

Class ==> <CLASS> Text </CLASS>

Text ==> Char Text | Char

Char ==> a | b | … | z | A | B | .. | Z

Students ==> Student Students | ε

Student ==> <STUD> Text </STUD>

Here, the left hand side of each production denotes one non-terminals (e.g., “Roll”, “Class”, etc.)
Those symbols on the right hand side for which no productions (i.e., substitutions) are defined are terminals (e.g., ‘a’, ‘b’, ‘|’, ‘<‘, ‘>’, “ROLL”, etc.)

Structure of a production

head　　derivation　　body

$A \Longrightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_k$

The above is same as:

1. $A \Longrightarrow \alpha_1$
2. $A \Longrightarrow \alpha_2$
3. $A \Longrightarrow \alpha_3$

…

K. $A \Longrightarrow \alpha_k$

CFG conventions

- Terminal symbols ← a, b, c…
- Non-terminal symbols ← A,B,C, …
- Terminal <u>or</u> non-terminal symbols ← X,Y,Z
- Terminal strings ← w, x, y, z
- Arbitrary strings of terminals and non-terminals ← α, β, γ, ..

String membership

How to say if a string belong to the language defined by a CFG?

1. Derivation
   - Head to body
2. Recursive inference
   - Body to head

Both are equivalent forms

<u>Example:</u>

- w = 01110
- Is w a palindrome?

<u>G:</u>

A => 0A0 | 1A1 | 0 | 1 | ε

A => 0A0
=> 01A10
=> 01110

Simple Expressions…

- We can write a CFG for accepting simple expressions
- G = (V,T,P,S)
  - V = {E,F}
  - T = {0,1,a,b,+,*,(,)}
  - S = {E}
  - P:
    - E ==> E+E | E*E | (E) | F
    - F ==> aF | bF | 0F | 1F | a | b | 0 | 1

Generalization of derivation

- Derivation is *head* ==> *body*

- A==>X (A derives X in a single step)
- A ==>*$_G$ X (A derives X in a multiple steps)

- <u>Transitivity:</u>

  IFA ==>*$_G$B, and B ==>*$_G$C, THEN A ==>*$_G$ C

Context-Free Language

- The language of a CFG, $G=(V,T,P,S)$, denoted by $L(G)$, is the set of terminal strings that have a derivation from the start variable S.
  - $L(G) = \{ w \text{ in } T^* \mid S \Longrightarrow^*_G w \}$

Derivations

- Two basic requirements for a grammar are :

1. To generate a valid string.
2. To recognize a valid string.

Derivation is a process that generates a valid string with the help of grammar by replacing the non-terminals on the left with the string on the right side of the production.

Types of derivations

- The two types of derivation are:

1. Left most derivation
2. Right most derivation.

- In leftmost derivations, the leftmost non-terminal in each sentinel is always chosen first for replacement.
- In rightmost derivations, the rightmost non-terminal in each sentinel is always chosen first for replacement.

Sentinels:

Given a grammar G with start symbol S, if $S \rightarrow \alpha$, where α may contain non-terminals or

terminals, then α is called the sentinel form of G.

Left-most & Right-most Derivation Styles

G:
E => E+E | E*E | (E) | F
F => aF | bF | 0F | 1F | ε

Derive the string a*(ab+10) from G:

$E \overset{*}{=}\!\!=\!>_G a*(ab+10)$

Left-most derivation:

Always substitute leftmost variable

- E
- ==> E * E
- ==> F * E
- ==> aF * E
- ==> a * E
- ==> a * (E)
- ==> a * (E + E)
- ==> a * (F + E)
- ==> a * (aF + E)
- ==> a * (abF + E)
- ==> a * (ab + E)
- ==> a * (ab + F)
- ==> a * (ab + 1F)
- ==> a * (ab + 10F)
- ==> a * (ab + 10)

Right-most derivation:

Always substitute rightmost variable

- E
- ==> E * E
- ==> E * (E)
- ==> E * (E + E)
- ==> E * (E + F)
- ==> E * (E + 1F)
- ==> E * (E + 10F)
- ==> E * (E + 10)
- ==> E * (F + 10)
- ==> E * (aF + 10)
- ==> E * (abF + 0)
- ==> E * (ab + 10)
- ==> F * (ab + 10)
- ==> aF * (ab + 10)
- ==> a * (ab + 10)

Example:

- Given grammar G : E → E+E | E*E | (E) | - E | id
- Sentence to be derived : – (id+id)

LEFTMOST DERIVATION	RIGHTMOST DERIVATION
E → - E	E → - E
E → - (E)	E → - (E)
E → - (E+E)	E → - (E+E)
E → - (id+E)	E → - (E+id)
E → - (id+id)	E → - (id+id)

String that appear in leftmost derivation are called left sentinel forms.
String that appear in rightmost derivation are called right sentinel forms

Leftmost vs. Rightmost derivations

Q1) For every leftmost derivation, there is a rightmost derivation, and vice versa. True or False?

True - will use parse trees to prove this

Q2) Does every word generated by a CFG have a leftmost and a rightmost derivation?

Yes – easy to prove (reverse direction)

Q3) Could there be words which have more than one leftmost (or rightmost) derivation?

Yes – depending on the grammar

Ambiguity

- A grammar that produces more than one parse for some sentence is said to be **ambiguous grammar**.

Given grammar

G : E → E+E | E*E | (E) | - E | id

The sentence **id+id*id** has the following two distinct leftmost derivations:

$E \rightarrow E + E$

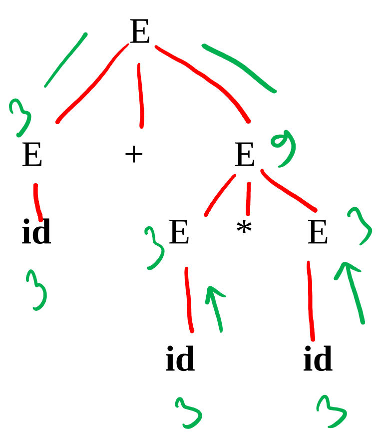
$E \rightarrow id + E$

$E \rightarrow id + E * E$

$E \rightarrow id + id * E$

$E \rightarrow id + id * id$

The two corresponding parse trees are :



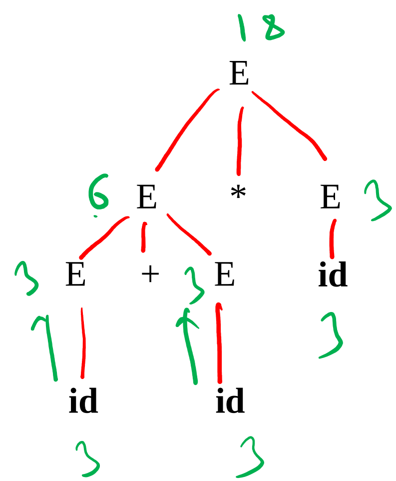
$E \rightarrow E * E$

$E \rightarrow E + E * E$

$E \rightarrow id + E * E$

$E \rightarrow id + id * E$

$E \rightarrow id + id * id$



Parse trees

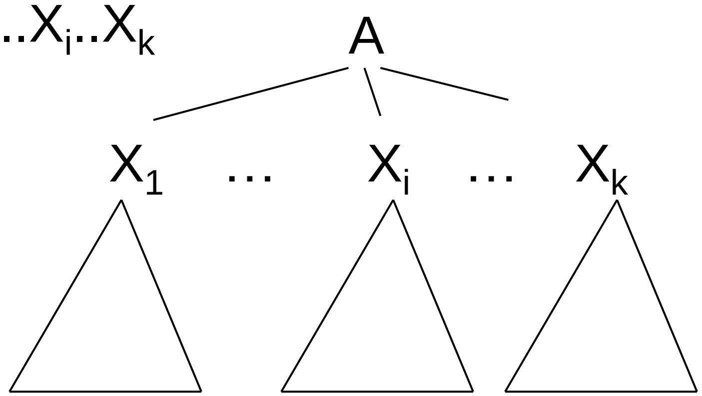
Parse Trees

- Each CFG can be represented using a *parse tree:*
  - Each internal node is labeled by a variable in V
  - Each leaf is terminal symbol
  - For a production, $A ==> X_1X_2 \ldots X_k$, then any internal node labeled A has k children which are labeled from $X_1, X_2, \ldots X_k$ from left to right

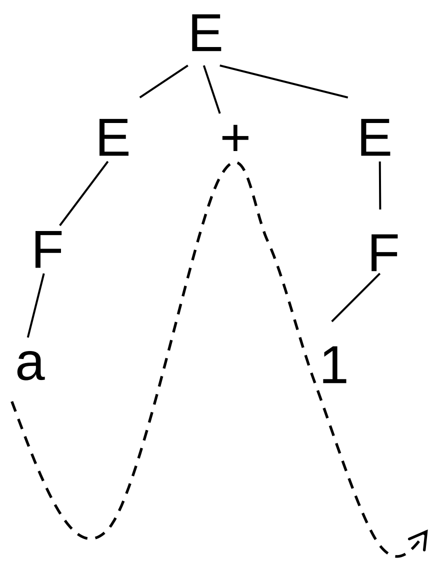
---

Parse tree for production and all other subsequent productions:

$A ==> X_1..X_i..X_k$

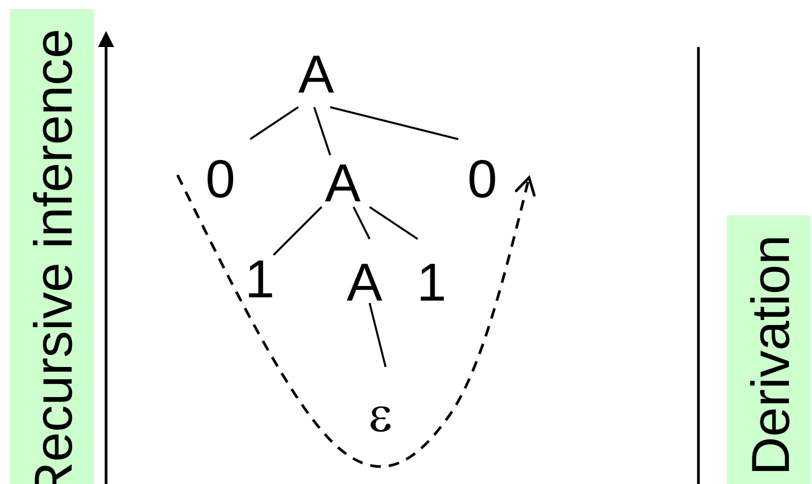


Examples



Parse tree for *a + 1*

G:
E => E+E | E*E | (E) | F
F => aF | bF | 0F | 1F | 0 | 1 | a | b



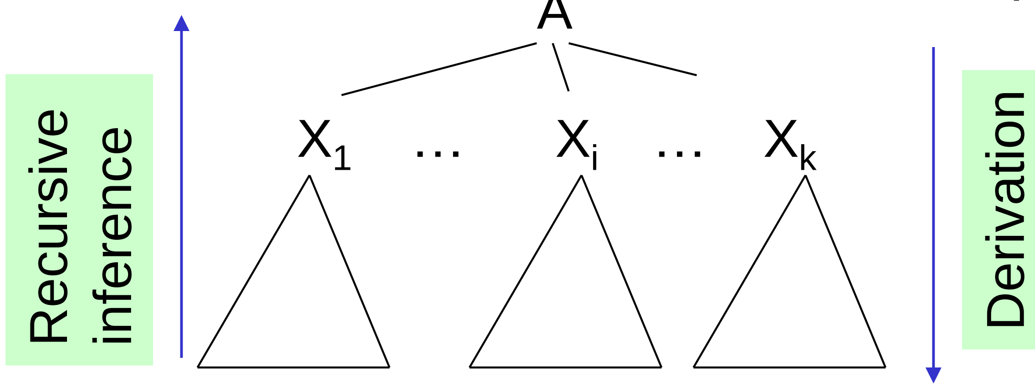
Parse tree for *0110*

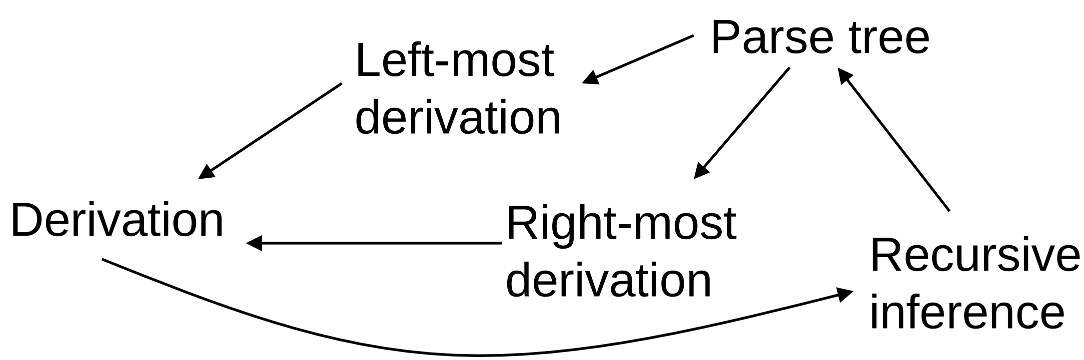
G:
A => 0A0 | 1A1 | 0 | 1 | ε

Parse Trees, Derivations, and Recursive Inferences

Production:

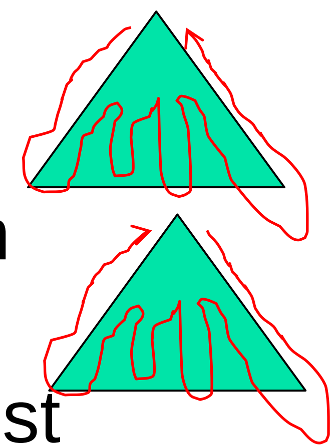
$A \Rightarrow X_1..X_i..X_k$





Interchangeability of different CFG representations

- Parse tree ==> left-most derivation
  - DFS left to right
- Parse tree ==> right-most derivation
  - DFS right to left
- ==> left-most derivation == right-most derivation
- Derivation ==> Recursive inference
  - Reverse the order of productions
- Recursive inference ==> Parse trees
  - bottom-up traversal of parse tree



Connection between CFLs and RLs

What kind of grammars result for regular languages?

CFLs & Regular Languages

- A CFG is said to be *right-linear* if all the productions are one of the following two forms: *A ==> wB (or) A ==> w*

Where:
• A & B are variables,
• w is a string of terminals

- Theorem 1: Every right-linear CFG generates a regular language
- Theorem 2: Every regular language has a right-linear grammar
- Theorem 3: Left-linear CFGs also represent RLs

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

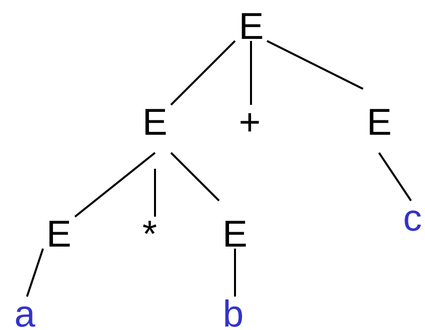
Why does ambiguity matter?

Values are different !!!

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

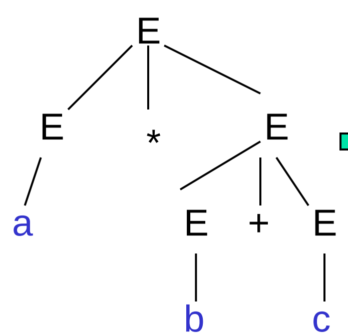
*string = a * b + c*

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



(a*b)+c

- <u>LM derivation #2</u>
  - 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

Ambiguous version:

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

How will this avoid ambiguity?

Inherently Ambiguous CFLs

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

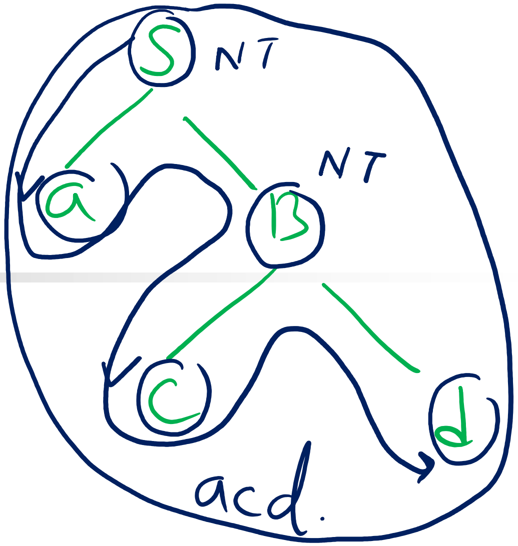
- A CFL is said to be *inherently ambiguous* if every CFG that describes it is ambiguous

Example:

- $L = \{ a^n b^n c^m d^m \mid n,m \geq 1\} \cup \{a^n b^m c^m d^n \mid n,m \geq 1\}$
- L is inherently ambiguous
- Why?

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

Parse Tree

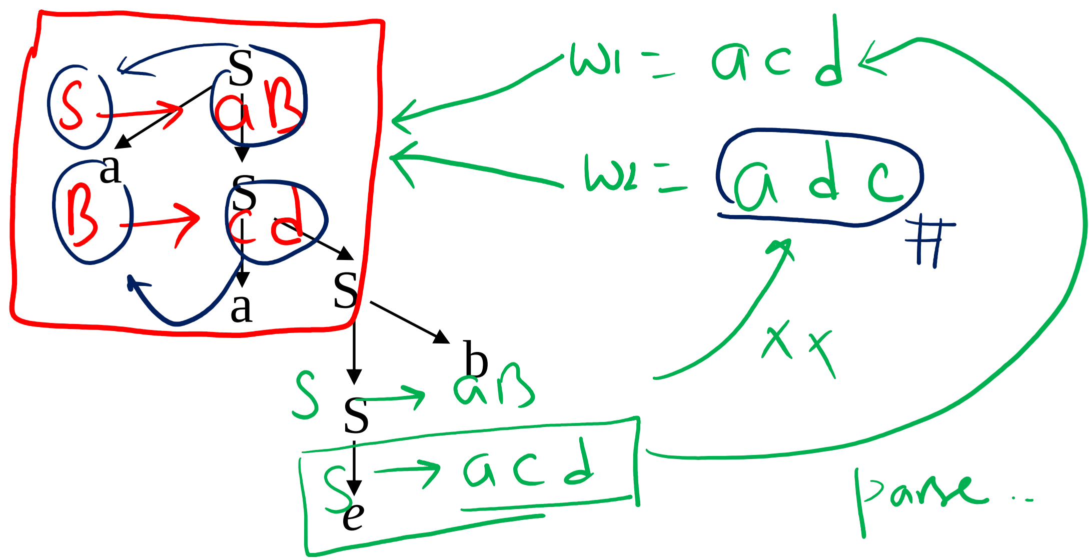


A parse tree of a derivation is a tree in which:

- Each internal node is labeled with a nonterminal

- If a rule $A \rightarrow A_1A_2\ldots A_n$ occurs in the derivation then A is a parent node of nodes labeled $A_1, A_2, \ldots, A_n$

Parsing: ? e.g.



Parse Trees

S → A | A B
A → ε | **a** | A **b** | A A
B → **b** | **b c** | B **c** | **b** B

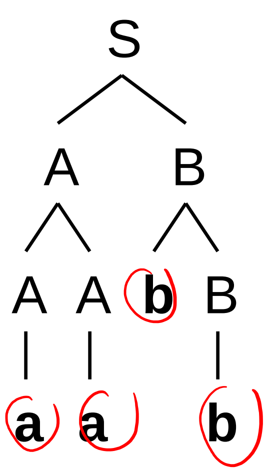
Sample derivations:

S ⇒ AB ⇒ AAB ⇒ **a**AB ⇒ **aa**B ⇒ **aab**B ⇒ **aabb**

S ⇒ AB ⇒ A**b**B ⇒ A**bb** ⇒ AA**bb** ⇒ A**abb** ⇒ **aabb**

These two derivations use same productions, but in different orders.

- This ordering difference is often uninteresting.
- *Derivation trees* give way to abstract away ordering differences.



Root label = start node.

Each interior label = variable.

Each parent/child relation = derivation step.

Each leaf label = terminal or ε.

a

All leaf labels together = derived string = *yield*.

aabb

Leftmost, Rightmost Derivations

Definition. A **left-most derivation** of a sentential form is one in which rules transforming the left-most nonterminal are always applied

Definition. A **right-most derivation** of a sentential form is one in which rules transforming the right-most nonterminal are always applied

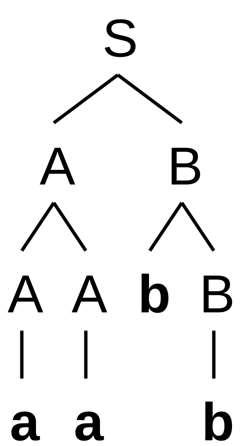
Leftmost & Rightmost Derivations

$S \rightarrow A \mid A\,B$
$A \rightarrow \varepsilon \mid \mathbf{a} \mid A\,\mathbf{b} \mid A\,A$
$B \rightarrow \mathbf{b} \mid \mathbf{b}\,\mathbf{c} \mid B\,\mathbf{c} \mid \mathbf{b}\,B$

Sample derivations:

$S \Rightarrow AB \Rightarrow AAB \Rightarrow \mathbf{a}AB \Rightarrow \mathbf{aa}B \Rightarrow \mathbf{aab}B \Rightarrow \mathbf{aabb}$

$S \Rightarrow AB \Rightarrow A\mathbf{b}B \Rightarrow A\mathbf{bb} \Rightarrow AA\mathbf{bb} \Rightarrow A\mathbf{abb} \Rightarrow \mathbf{aabb}$



These two derivations are special.

1st derivation is *leftmost*.
Always picks leftmost variable.

2nd derivation is *rightmost*.
Always picks rightmost variable.

Left / Rightmost Derivations

- In proofs…
  - Restrict attention to left- or rightmost derivations.
- In parsing algorithms…
  - Restrict attention to left- or rightmost derivations.
  - E.g., recursive descent uses leftmost; **`yacc`** uses rightmost.

Derivation Trees

Ambiguity

$S \rightarrow A \mid A\,B$

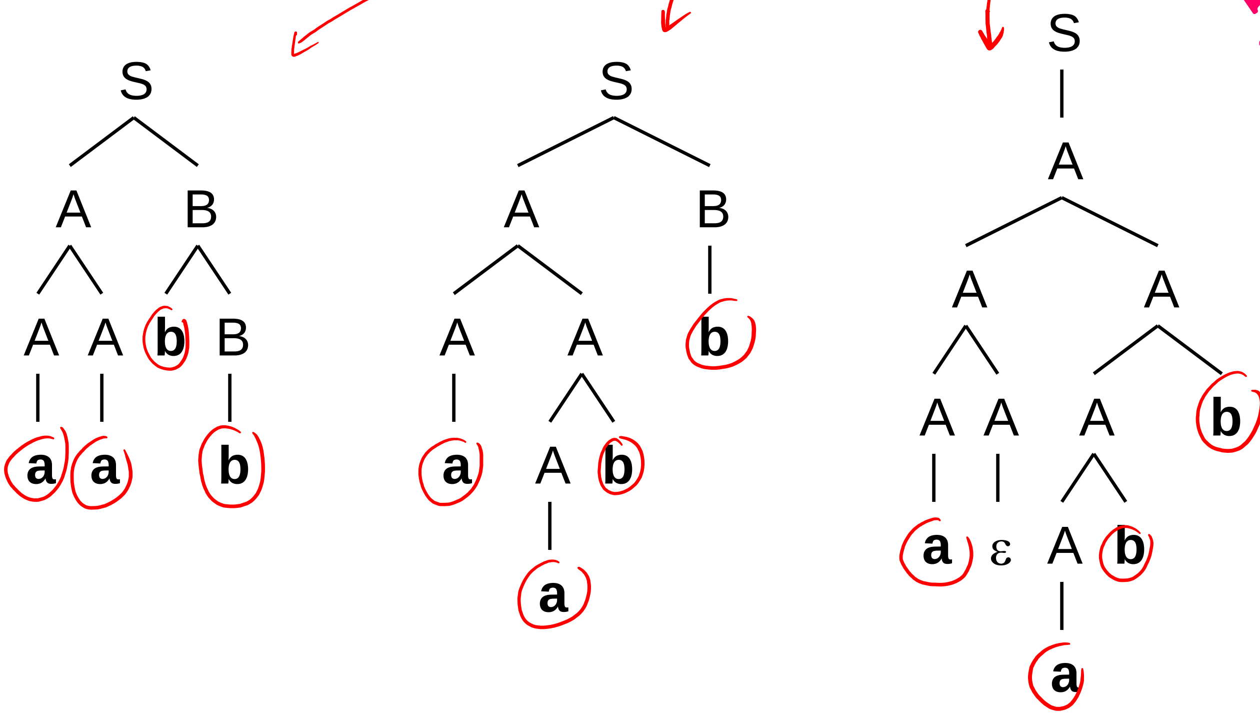
$A \rightarrow \varepsilon \mid \mathbf{a} \mid A\,\mathbf{b} \mid A\,A$

$B \rightarrow \mathbf{b} \mid \mathbf{b}\,\mathbf{c} \mid B\,\mathbf{c} \mid \mathbf{b}\,B$

w = **aabb**

Other derivation trees for this string?

? ?



Infinitely many others possible.

Ambiguous Grammar

Definition. A grammar G is ambiguous if there is a word w $\in$ L(G) having are least two different parse trees

$S \rightarrow A$
$S \rightarrow B$
$S \rightarrow AB$
$A \rightarrow aA$
$B \rightarrow bB$
$A \rightarrow e$
$B \rightarrow e$

ambiguous.

we can multiple PT.

Notice that a has at least two left-most derivations

Ambiguity

CFG *ambiguous* $\Leftrightarrow$ any of following equivalent statements:

- $\exists$ string w with multiple derivation trees.
- $\exists$ string w with multiple leftmost derivations.
- $\exists$ string w with multiple rightmost derivations.

Defining ambiguity of <u>grammar</u>, not language.

Ambiguity & Disambiguation

Given an ambiguous grammar, would like an equivalent unambiguous grammar.

- Allows you to know more about structure of a given derivation.
- Simplifies inductive proofs on derivations.
- Can lead to more efficient parsing algorithms.
- In programming languages, want to impose a canonical structure on derivations. E.g., for **1+2×3**.

12/18

Strategy: Force an ordering on all derivations.

Disambiguation: Example 1

$$\begin{aligned} Exp &\rightarrow \mathbf{n} \\ &\mid \; Exp \;\mathbf{+}\; Exp \\ &\mid \; Exp \times Exp \end{aligned}$$

?

$E \rightarrow E + E \mid E \times E \mid id$

What is an equivalent unambiguous grammar?

$$\begin{aligned} Exp &\rightarrow Term \\ &\mid \; Term \;\mathbf{+}\; Exp \\ Term &\rightarrow \mathbf{n} \\ &\mid \; \mathbf{n} \times Term \end{aligned}$$

$E \rightarrow E + T \mid T$

$T \rightarrow T \times id \mid id$

Uses

- operator precedence
- left-associativity

Disambiguation

What is a general algorithm?

None exists!

There are CFLs that are *inherently ambiguous*

Every CFG for this language is ambiguous.

E.g., $\{\mathbf{a}^n\mathbf{b}^n\mathbf{c}^m\mathbf{d}^m \mid n \geq 1, m \geq 1\} \cup \{\mathbf{a}^n\mathbf{b}^m\mathbf{c}^m\mathbf{d}^n \mid n \geq 1, m \geq 1\}$.

So, can't necessarily eliminate ambiguity!

CFG Simplification

~~gmy~~

(How)

Can't always eliminate ambiguity.

But, CFG simplification & restriction still useful theoretically & pragmatically.

why?

$\boxed{\text{ad}}$

- Simpler grammars are easier to understand.
- Simpler grammars can lead to faster parsing.
- Restricted forms useful for some parsing algorithms.
- Restricted forms can give you more knowledge about derivations.

$A \to Aa$ $\quad Aa \to AAa \to AAAaa \to AAAAaaa$

CFG Simplification (How?)

1. Eliminate ambiguity. (if possible)
2. **Eliminate "useless" variables.** ✓
3. **Eliminate ε-productions: A→ε.** ✓
4. **Eliminate unit productions: A→B.** ✓
5. Eliminate redundant productions.
6. Trade left- & right-recursion.

gnp.

Eliminate "useless" variables.

w = aabb

aaaaA X X

- T-->aaB | ~~abA~~ | aaT
- ~~A--> aA~~
- B-->ab | b
- ~~C--> ad~~

A → aA → aaA → aaaA → aaaaA

because it doesn't occur in rhs. (body of production)

T → aaB | aaT

B → ab | b

T → aaT

→ aaaaB

aaaaab

Eliminate "useless" variables.

$T \to aaB \mid \cancel{abA} \mid aaT$

$\cancel{A \to aA}$

$B \to ab \mid b$

$\cancel{C \to ad}$

$\Rightarrow$

removing useless symbols.

$T \to aaB \mid aaT$

$B \to ab \mid b$

Eliminate e-productions: A→ e.

- S--> XYX
- X--> 0X | e
- Y --> 1Y | e

Eliminate e-productions: A→ e.

$S \to XYX$

$X \to 0X \mid \epsilon$

$Y \to 1Y \mid \epsilon$

Eliminate all ϵ-production

$S \to XX \mid XY \mid YX \mid X \mid Y$

$X \to 0X \mid 0$

$Y \to 1Y \mid 1$

CFG with e-prdn

simplified CFG

4.

How?

what

Eliminate unit productions: A→B

- S-->0A | 1B | C
- A--> 0S | 00
- B--> 1 | A
- C --> 01

① S → C

② B → A

These are the two unit productions in the given G.

Eliminate unit productions: A→B

- S-->0A | 1B | C
- A--> 0S | 00
- B--> 1 | A
- C --> 01

$S \to 0A$

$S \to 1B$

$\boxed{S \cancel{\to} C}$ $S \to 01$

$A \to 0S$

$A \to 00$

$B \to 1$

$\boxed{B \cancel{\to} A}$ $B \to 0S | 00$

$C \to 01$

unit prod

$S \to 0A | 1B | 01$
$A \to 0S | 00$
$B \to 1 | 0S | 00$
$C \to 01$

Simplified CFG

Examples

simply

$$S \to AB \mid \lambda$$
$$A \to aA \mid a$$
$$C \to cC \mid c$$

$$S \to \lambda$$
$$A \to aA \mid a$$

$$S \to \lambda$$

B does not derive terminal string; *C* unreachable.

A unreachable.

$$S \to BD$$
$$B \to BD \mid B$$
$$D \to DB \mid D$$

Empty set of productions

"Non-termination"

CFG Simplification: Example

How can the following be simplified?

?

?

$S \to A\ B$

$S \to A\ C\ D$

$A \to A\ \mathbf{a}$

$A \to \mathbf{a}$

$A \to \mathbf{a}\ A$

$A \to \mathbf{a}$

$C \to \varepsilon$

$D \to \mathbf{d}\ D$

$D \to E$

$E \to \mathbf{e}\ A\ \mathbf{e}$

$F \to \mathbf{f}\ \mathbf{f}$

1) Delete: B useless because nothing derivable from B.

2) Delete either $A \to A\mathbf{a}$ or $A \to \mathbf{a}A$.

3) Delete one of the idential productions.

4) Delete & also replace $S \to ACD$ with $S \to AD$.

5) Replace with $D \to \mathbf{e}A\mathbf{e}$.

6) Delete: E useless after change #5.

7) Delete: F useless because not derivable from S.

Trading Left- & Right-Recursion

Left recursion: $A \rightarrow A\ \alpha$

Right recursion: $A \rightarrow \alpha\ A$

Most algorithms have trouble with one,

In recursive descent, avoid left recursion.

Normal Forms

By Prashant Gautam

what ?

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

$A \to BC$ $A \to a$ CNF

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. $\boxed{A \rightarrow BC}$ 1) *where A,B,C are variables, or*

ii. $\boxed{A \rightarrow a}$ 2) *where a is a terminal*

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

elimination

CNF checklist

1) A → BC
2) A → a

Is this grammar in CNF?

G₁:

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

E → E+T

××

<u>Checklist:</u>

- 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

Practice

$S \rightarrow AB$

$S \rightarrow C$

$A \rightarrow a$

$B \rightarrow b$

How to convert a G into CNF?

- Assumption: G has no ε-productions, unit productions or useless symbols
- If there is, just do simplify.

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 $A \rightarrow B_1\underbrace{B_2\underbrace{B_3\ldots B_k}_{C_2}}_{C_1}$ by:

B_1 C_1 B_2 C_2 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:

$S \Rightarrow AS \mid BABC$

$A \Rightarrow A1 \mid 0A1 \mid 01$

$B \Rightarrow 0B \mid 0$

$C \Rightarrow 1C \mid 1$

G in CNF:

$X_0 \Rightarrow 0$

$X_1 \Rightarrow 1$

$S \Rightarrow AS \mid BY_1$

$Y_1 \Rightarrow AY_2$

$Y_2 \Rightarrow BC$

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

$Y_3 \Rightarrow AX_1$

$B \Rightarrow X_0B \mid 0$

$C \Rightarrow X_1C \mid 1$

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

Example #2 (HomeWork)

G:

1. $E \rightarrow E+T \mid T*F \mid (E) \mid Ia \mid Ib \mid I0 \mid I1$
2. $T \rightarrow T*F \mid (E) \mid Ia \mid Ib \mid I0 \mid I1$
3. $F \rightarrow (E) \mid Ia \mid Ib \mid I0 \mid I1$
4. $I \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1$

Step (1)

1. $E \rightarrow EX_{+}T \mid TX_{*}F \mid X_{(}EX_{)} \mid IX_{a} \mid IX_{b} \mid IX_{0} \mid IX_{1}$
2. $T \rightarrow TX_{*}F \mid X_{(}EX_{)} \mid IX_{a} \mid IX_{b} \mid IX_{0} \mid IX_{1}$
3. $F \rightarrow X_{(}EX_{)} \mid IX_{a} \mid IX_{b} \mid IX_{0} \mid IX_{1}$
4. $I \rightarrow X_{a} \mid X_{b} \mid IX_{a} \mid IX_{b} \mid IX_{0} \mid IX_{1}$
5. $X_{+} \rightarrow +$
6. $X_{*} \rightarrow *$
7. $X_{+} \rightarrow +$
8. $X_{(} \rightarrow ($
9. …….

Step (2)

1. $E \rightarrow EC_{1} \mid TC_{2} \mid X_{(}C_{3} \mid IX_{a} \mid IX_{b} \mid IX_{0} \mid IX_{1}$
2. $C_{1} \rightarrow X_{+}T$
3. $C_{2} \rightarrow X_{*}F$
4. $C_{3} \rightarrow EX_{)}$
5. $T \rightarrow$ ………
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 => 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==>a α, where a is a terminal, i.e. a ε T* and α is a string of zero or more variables. i.e. α ε V*

So we can rewrite as:

A→ aV* with a ε T*

Or,

A→ aV+

A→ a with a ε T

Eliminating Left Recursion

- A grammar is said to be *left recursive* if it has a non-terminal *A* such that there is a derivation $A \rightarrow A\alpha$ for some string α.
- Top-down parsing methods cannot handle left-recursive grammars. Hence, left recursion can be eliminated as follows:
- If there is a production $A \rightarrow A\alpha \mid \beta$ it can be replaced with a sequence

  $\mathbf{A \rightarrow \beta A'}$

  $\mathbf{A' \rightarrow \alpha A' \mid \varepsilon}$

without changing the set of strings derivable from A.

Immediate Left - Recursion

$A \to A\alpha \mid \beta$

$\Downarrow$ Eliminate immediate left recursion

$A \to \beta A'$
$A' \to \alpha A' \mid \epsilon$

In general,

$A \to A\,\alpha_1 \mid \ldots \mid A\,\alpha_m \mid \beta_1 \mid \ldots \mid \beta_n$ where $\beta_1 \ldots \beta_n$ do not start with A

$\Downarrow$ eliminate immediate left recursion

$A \to \beta_1\, A' \mid \ldots \mid \beta_n\, A'$
$A' \to \alpha_1\, A' \mid \ldots \mid \alpha_m\, A' \mid \varepsilon$ an equivalent grammar

Example :

- Consider the following grammar for arithmetic expressions:

  E → E+T | T

  T → T*F | F

  F → (E) | id

- First eliminate the left recursion for E as

  E → TE'

  E' → +TE' | ε

- Then eliminate for T as

  T → FT' T'→ *FT' | ε

Thus the obtained grammar after eliminating left recursion is

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow (E) \mid id$

Non-Immediate Left-Recursion

- By just eliminating the immediate left-recursion, we may not get a grammar which is not left recursive.

$S \rightarrow Aa \mid b$

$A \rightarrow Sc \mid d$ This grammar is not immediately left-recursive, but it is still left-recursive.

$\underline{S} \Rightarrow Aa \Rightarrow \underline{S}ca$ or

$\underline{A} \Rightarrow Sc \Rightarrow \underline{A}ac$ causes to a left-recursion

So, we have to eliminate all left-recursions from our grammar

CFG to GNF Steps

- **Step 1: Convert the grammar into CNF.**

If the given grammar is not in CNF, convert it into CNF. You can refer the following topic to convert the CFG into CNF: Chomsky normal form

- **Step 2: If the grammar exists left recursion, eliminate it.**

If the context free grammar contains left recursion, eliminate it. You can refer the following topic to eliminate left recursion: Left Recursion

- **Step 3: In the grammar, convert the given production rule into GNF form.**

If any production rule in the grammar is not in GNF form, convert it.

Example

1. $S \rightarrow XB \mid AA$
2. $A \rightarrow a \mid SA$
3. $B \rightarrow b$
4. $X \rightarrow a$

As the given grammar G is already in CNF and there is no left recursion, so we can skip step 1 and step 2 and directly go to step 3.

- The production rule $A \rightarrow SA$ is not in GNF, so we substitute $S \rightarrow XB \mid AA$ in the production rule $A \rightarrow SA$ as:

1. $S \rightarrow XB \mid AA$
2. $A \rightarrow a \mid XBA \mid AAA$
3. $B \rightarrow b$
4. $X \rightarrow a$

- The production rule $S \rightarrow XB$ and $B \rightarrow XBA$ is not in GNF, so we substitute $X \rightarrow a$ in the production rule $S \rightarrow XB$ and $B \rightarrow XBA$ as:

$S \rightarrow aB \mid AA$

$A \rightarrow a \mid aBA \mid AAA$

$B \rightarrow b$

$X \rightarrow a$

- Now we will remove left recursion

($A \rightarrow AAA$), we get:

1. $S \rightarrow aB \mid AA$
2. $A \rightarrow aC \mid aBAC$
3. $C \rightarrow AAC \mid \varepsilon$
4. $B \rightarrow b$
5. $X \rightarrow a$

- Now we will remove null production $C \rightarrow \varepsilon$, we get:

1. $S \rightarrow aB \mid AA$
2. $A \rightarrow aC \mid aBAC \mid a \mid aBA$
3. $C \rightarrow AAC \mid AA$
4. $B \rightarrow b$
5. $X \rightarrow a$

- The production rule $S \rightarrow AA$ is not in GNF, so we substitute $A \rightarrow aC \mid aBAC \mid a \mid aBA$ in production rule $S \rightarrow AA$ as:

1. $S \rightarrow aB \mid aCA \mid aBACA \mid aA \mid aBAA$
2. $A \rightarrow aC \mid aBAC \mid a \mid aBA$
3. $C \rightarrow AAC$
4. $C \rightarrow aCA \mid aBACA \mid aA \mid aBAA$
5. $B \rightarrow b$
6. $X \rightarrow a$

- The production rule $C \rightarrow AAC$ is not in GNF, so we substitute $A \rightarrow aC \mid aBAC \mid a \mid aBA$ in production rule $C \rightarrow AAC$ as:

1. $S \rightarrow aB \mid aCA \mid aBACA \mid aA \mid aBAA$
2. $A \rightarrow aC \mid aBAC \mid a \mid aBA$
3. $C \rightarrow aCAC \mid aBACAC \mid aAC \mid aBAAC$
4. $C \rightarrow aCA \mid aBACA \mid aA \mid aBAA$
5. $B \rightarrow b$
6. $X \rightarrow a$

Assignment (CFG to GNF)

- $S \rightarrow AB$

  $A \rightarrow BS|b$

  $B \rightarrow SA|a$

Bakus Naur Form

- Notation Technique used for CFG
- Used for Specifying language Syntax

- <Symbol>:= exp1 | exp2 | exp3 …

BNF: Example

```
(2.0 * PI) / n

<expression>  ::=  <expression> + <term>
                |  <expression> - <term>
                |  <term>
<term>        ::=  <term> * <factor>
                |  <term> / <factor>
                |  <factor>
<factor>      ::=  number
                |  name
                |  ( <expression> )
```

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*