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 $\{\varepsilon\}$)
  - 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?

<u>G:</u>

S => (S) | SS | ε

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

Example #3

- A grammar for L = $\{0^m1^n \mid m \geq n\}$

- CFG?

<u>G:</u>
S => 0S1 | A
A => 0A | ε

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

More examples

- $L_1 = \{0^n \mid n \geq 0\}$
- $L_2 = \{0^n \mid n \geq 1\}$
- $L_3=\{0^i1^j2^k \mid i=j \text{ or } j=k, \text{ where } i,j,k \geq 0\}$
- $L_4=\{0^i1^j2^k \mid i=j \text{ or } i=k, \text{ where } i,j,k \geq 1\}$

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)

Structure of a production

head	derivation	body

A =======> $\alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_k$

The above is same as:

1. $A ==> \alpha_1$
2. $A ==> \alpha_2$
3. $A ==> \alpha_3$

…

K. $A ==> \alpha_k$

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

G:
A => 0A0 | 1A1 | 0 | 1 | ε

<u>Example:</u>

- w = 01110
- Is w a palindrome?

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.
  - $L(G) = \{ w \text{ in } T^* \mid S \Rightarrow^*_G w \}$

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
- ==> 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)

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

Parse trees

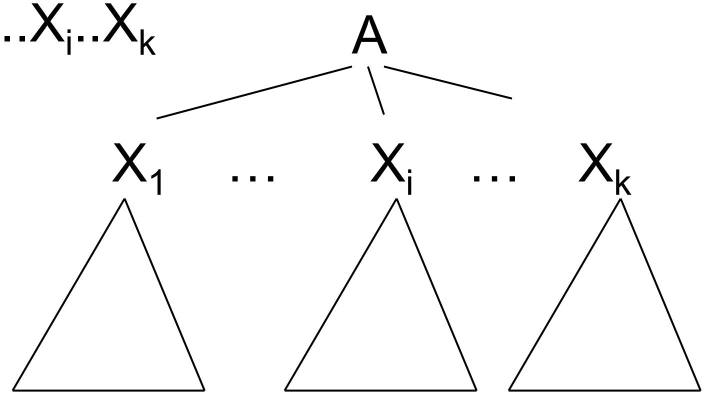
Parse Trees

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

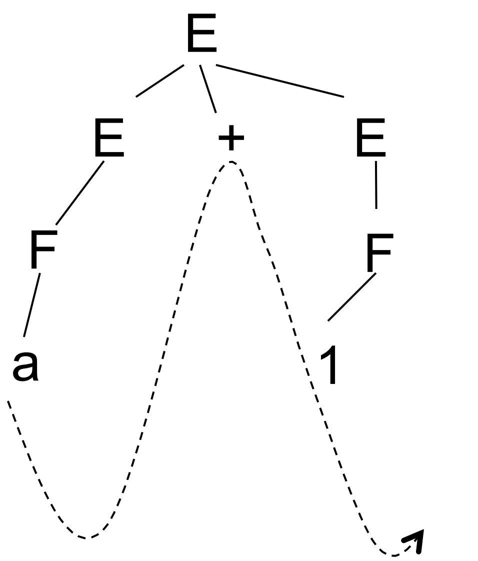
---

<u>Parse tree for production and all other subsequent productions:</u>

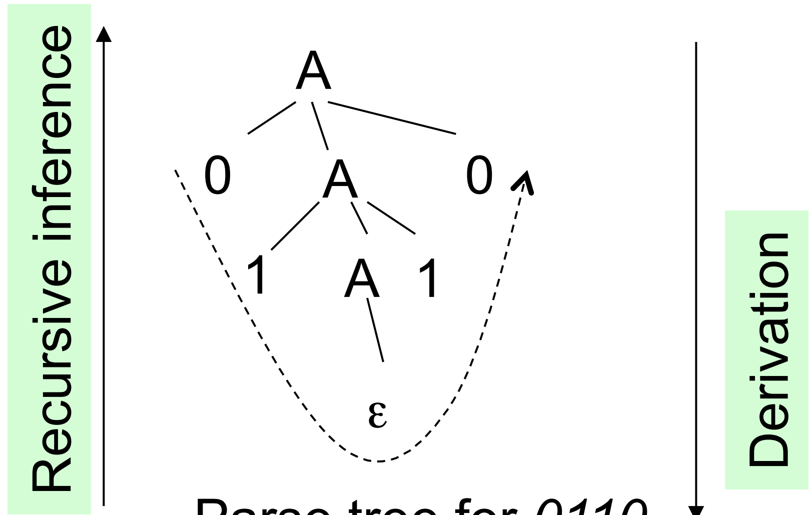
$A ==> X_1..X_i..X_k$



Examples



Parse tree for *a + 1*



Parse tree for *0110*

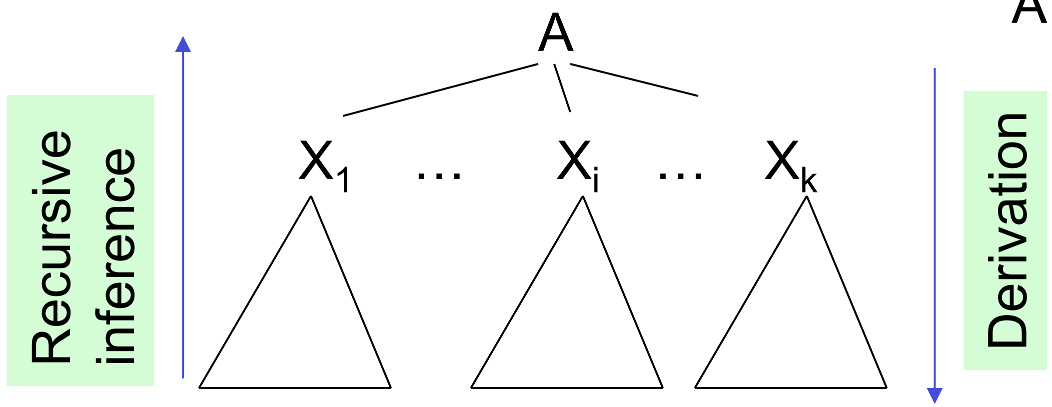
G:
E => E+E | E*E | (E) | F
F => aF | bF | 0F | 1F | 0 | 1 | a | b

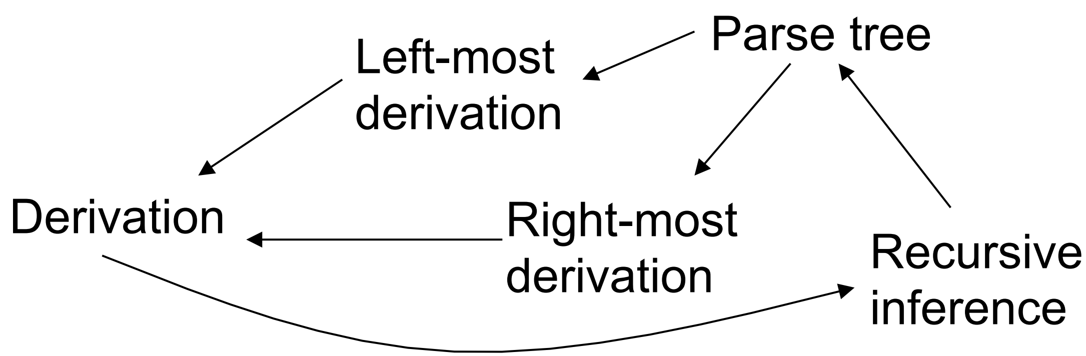
G:
A => 0A0 | 1A1 | 0 | 1 | ε

Parse Trees, Derivations, and Recursive Inferences

Production:

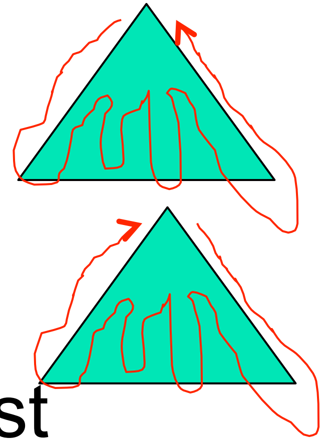
$A \Longrightarrow X_1..X_i..X_k$





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?

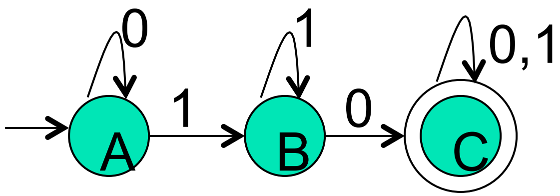
CFLs & Regular Languages

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

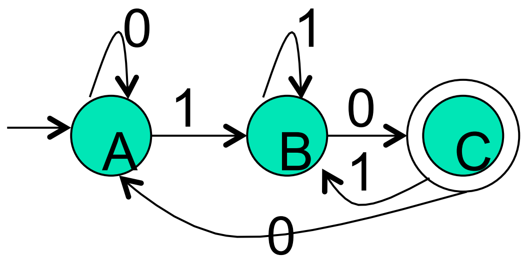
Where:
- A & B are variables,
- w is a string of terminals

- 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

Some Examples



Right linear CFG?



Right linear CFG?

- A => 01B | C
  B => 11B | 0C | 1A
  C => 1A | 0 | 1

Finite Automaton?

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

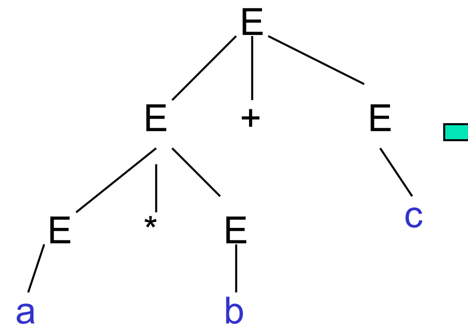
Why does ambiguity matter?

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

Values are different !!!

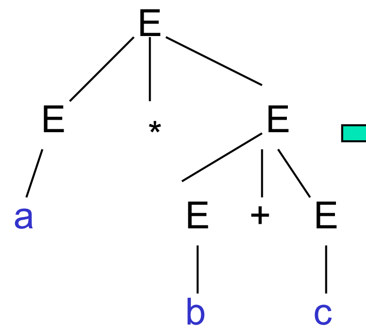
*string = a * b + c*

- <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: $a^n b^n c^n d^n$

Summary

- Context-free grammars
- Context-free languages
- Productions, derivations, recursive inference, parse trees
- Left-most & right-most derivations
- Ambiguous grammars
- Removing ambiguity
- CFL/CFG applications
  - parsers, markup languages