Context-Free Languages, Parse Trees & Ambiguity

N Geetha

AM & CS

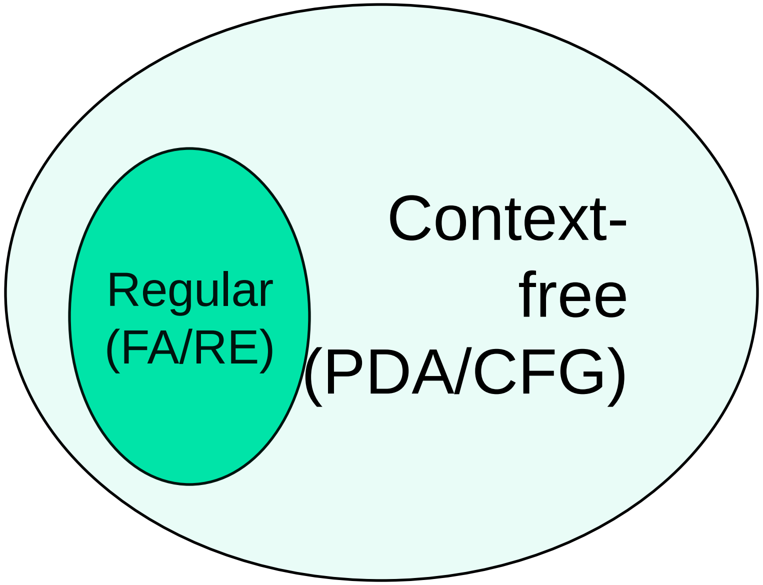
PSG Tech

Context Free Languages

- A *context-free grammar* is a notation for describing languages.
- It is more powerful than finite automata or RE's, but still cannot define all possible languages.
- Useful for nested structures, e.g., parentheses in programming languages.

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.
  - <u>Proof:</u>
    - Let $w=0^N10^N$ (assuming N to be the p/l constant)
    - By Pumping lemma, w can be rewritten as xyz, such that xy^kz is also L (for any $k\geq 0$)
    - But $|xy|\leq N$ and $y\neq\varepsilon$
    - ==> $y=0^+$
    - ==> xy^kz *will NOT* be in L for k=0
    - ==> Contradiction

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. S --> ε
2. S --> 0
3. S --> 1
4. S --> 0S0
5. S --> 1S1

Productions

Terminal

Variable or non-terminal

Same as:
S --> 0S0 | 1S1 | 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

<u>G:</u>
S --> 0S0 | 1S1 | 0 | 1 | ε

- <u>Example:</u> w=01110
- G can generate w as follows:

1. S --> 0S0
2. => 01S10
3. => 01110

<u>Generating a string from a grammar:</u>

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=(N,T,S,P), where:
  - N: set of variables or non-terminals
  - T: set of terminals (= alphabet U {ε})
  - P: set of *productions,* each of which is of the form
    $N \text{ --> } \alpha_1 \mid \alpha_2 \mid \ldots$
    - Where each α_i is an arbitrary string of variables and terminals
  - S : start variable

CFG for the language of binary palindromes:
G=({S},{0,1},S, P)
P: S --> 0 S 0 | 1 S 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 "derive" the string "(((()))()())" using this grammar?

Example #3

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

- CFG?

G:
S --> 0S1 | A
A --> 0A | ε

How would you derive 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?

More examples

- $L_1 = \{0^n 1^n \mid n \geq 1\}$
- $L_2=\{0^i 1^j 2^k \mid i=j \text{ or } j=k, \text{ where } i,j,k \geq 0\}$
- $L_3=\{0^i 1^j 2^k \mid i=j \text{ or } i=k, \text{ where } i,j,k \geq 1\}$
- L_4 = { w / w is a binary palindrome}
- L_5 = { w / w is a pal and w ϵ {a,b} }

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.)

Syntactic Expressions in Programming Languages

*result = a*b + score + 10 * distance + c*

terminals   variables   Operators are also terminals

Regular languages have only terminals

- Reg expression = [a-z][a-z0-1]*
- If we allow only letters a & b, and 0 & 1 for constants (for simplification)
  - Regular expression = (a+b)(a+b+0+1)*

Simple Expressions…

- We can write a CFG for accepting simple expressions
- G = (N,T,S,P)
  - N = {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

Sentential Forms

- Any string of variables and/or terminals derived from the start symbol is called a *sentential form*.
- Formally, α is a sentential form iff $S \Longrightarrow^{*} \alpha$.

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.
  - $L(G) = \{ w \text{ in } T^* \mid S \Longrightarrow^*_G w \}$
- But not all languages are CFL's.
- Intuitively: CFL's can count two things, not three.

BNF Notation

- Grammars for programming languages are often written in BNF (*Backus-Naur Form*).
- Variables are words in <…>; Example: <statement>.
- Terminals are often multicharacter strings indicated by boldface or underline; Example: **while** or WHILE.

BNF Notation – (2)

- Symbol ::= is often used for ->.
- Symbol | is used for “or.”
  - A shorthand for a list of productions with the same left side.
- Example: S -> 0S1 | 01 is shorthand for S -> 0S1 and S -> 01.

BNF Notation

- Symbol … is used for “one or more.”
- Example: <digit> ::= 0|1|2|3|4|5|6|7|8|9

<unsigned integer> ::= <digit>…

  - Note: that’s not exactly the * of RE’s.
- Translation: Replace α… with a new variable A and productions A -> Aα | α.

Example:

- Grammar for unsigned integers can be replaced by:

  U -> UD | D

  D -> 0|1|2|3|4|5|6|7|8|9

BNF Notation: Optional Elements

- Surround one or more symbols by […] to make them optional.
- Example: <statement> ::= **if** <condition> **then** <statement> [; **else** <statement>]
- Translation: replace [α] by a new variable A with productions A -> α | ϵ.

Example: Optional Elements

- Grammar for if-then-else can be replaced by:

S -> iCtSA

A -> ;eS | ϵ

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)

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?

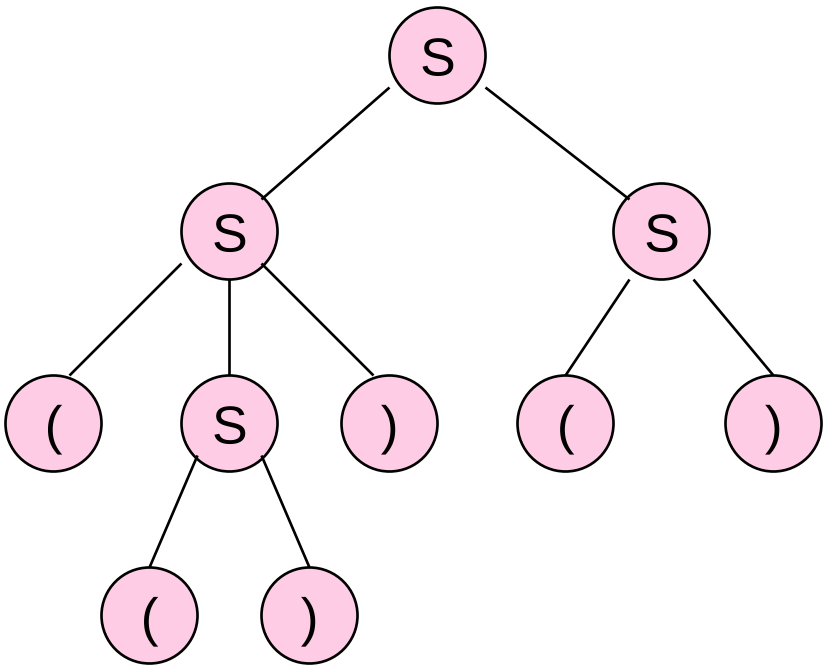
Yes – depending on the grammar

Parse Trees

- *Parse trees* are trees labeled by symbols of a particular CFG.
- Leaves: labeled by a terminal or ε.
- Interior nodes: labeled by a variable.
  - Children are labeled by the right side of a production for the parent.
- Root: must be labeled by the start symbol.

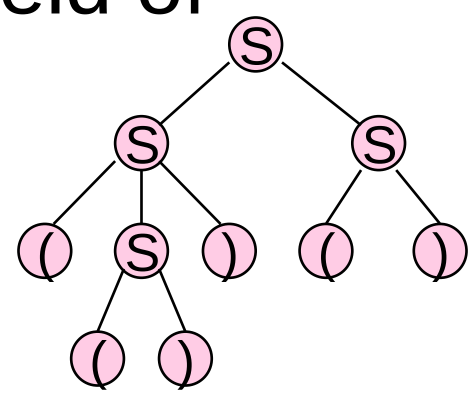
Example: Parse Tree

S -> SS | (S) | ()

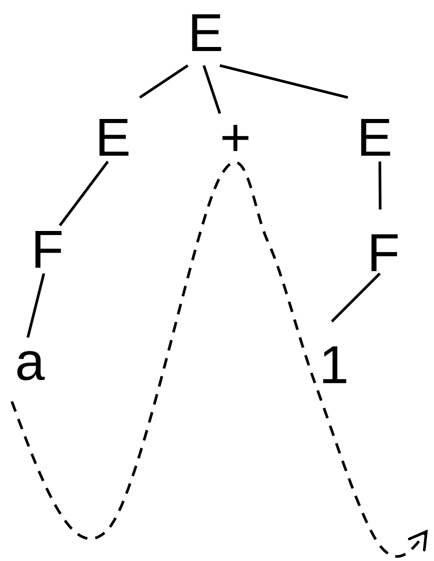


Yield of a Parse Tree

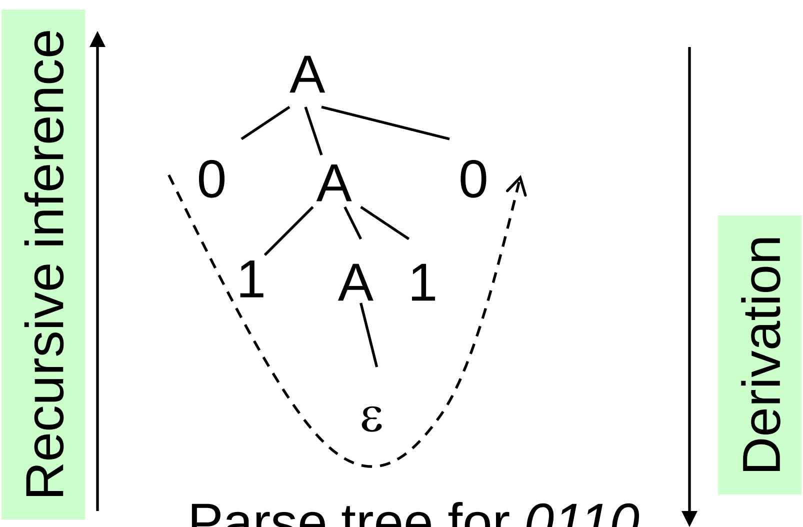
- The concatenation of the labels of the leaves in left-to-right order
  - That is, in the order of a preorder traversal.

  is called the *yield* of the parse tree.
- Example: yield of  is (())()

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

Ambiguity in CFGs

- A word is *ambiguous* if there exists more than one left most derivation.
- 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

Input string: 00111

Can be derived in two ways

LM derivation #1:

S => AS

=> 0A1S

=>0A11S

=> 00111S

=> 00111

LM derivation #2:

S => AS

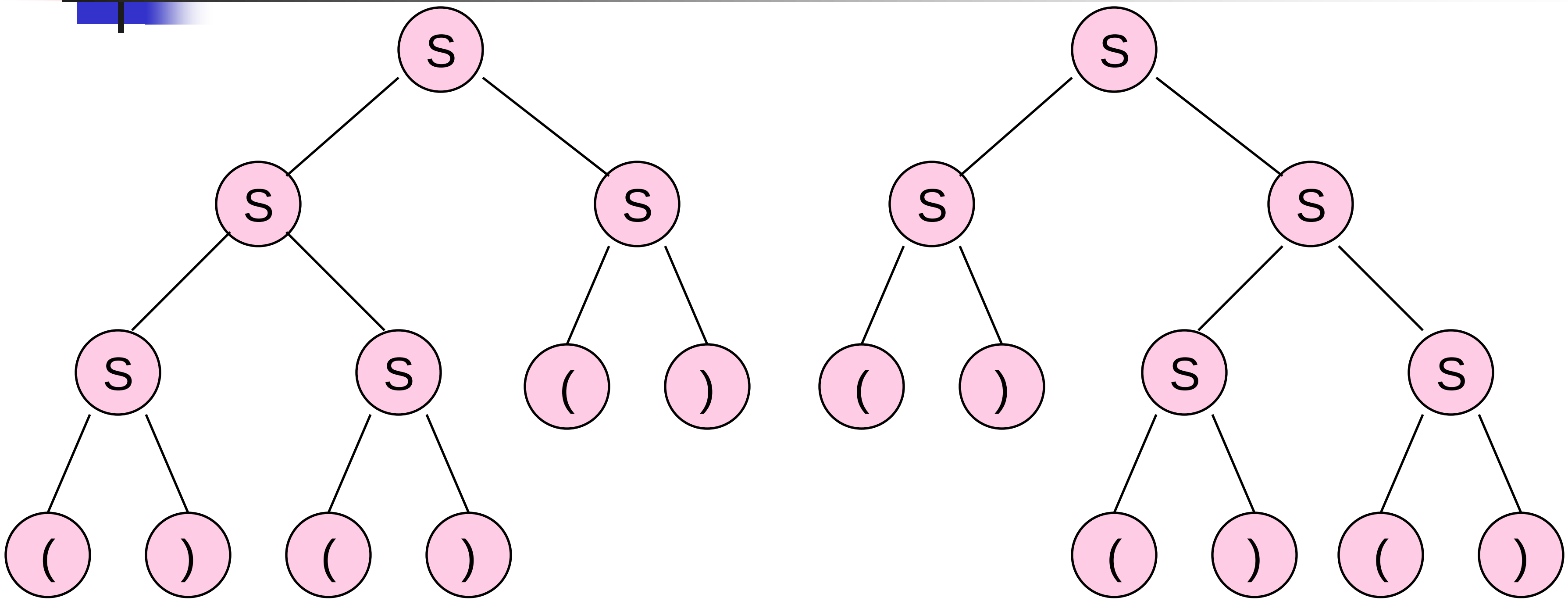
=> A1S

=> 0A11S

=> 00111S

=> 00111

Example 2 : S -> SS | (S) | ()



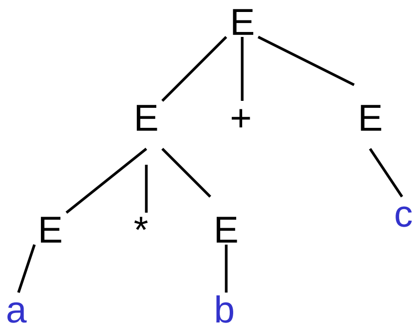
Why does ambiguity matter?

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

Values are different !!!

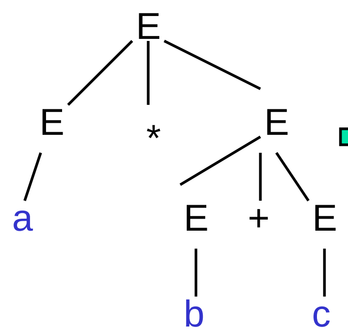
*string = a * b + c*

- 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

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

Another Example

$$S \to aB \mid bA$$
$$A \to a \mid aS \mid bAA$$
$$B \to b \mid bS \mid aBB$$

- Is ab, baba, abbbaa in L?
- How about a, bba?

- What is the language of this CFG?
- Is the CFG ambiguous?

One Possible Ambiguous Grammar

S -> AB | CD

A -> 0A1 | 01 — A generates equal 0’s and 1’s

B -> 2B | 2 — B generates any number of 2’s

C -> 0C | 0 — C generates any number of 0’s

D -> 1D2 | 12 — D generates equal 1’s and 2’s

And there are two derivations of every string with equal numbers of 0’s, 1’s, and 2’s. E.g.:

S => AB => 01B =>012

S => CD => 0D => 012

Summary

- Context-free grammars
- Context-free languages
- Sentential form
- Left-most & right-most derivations
- Parse trees
- Ambiguous grammars
- Removing ambiguity
- CFL/CFG applications
  - parsers, markup languages