Context-Free Languages & Grammars
(CFLs & CFGs)

The structure and the content of the lecture is based on http://www.eecs.wsu.edu/~ananth/CptS317/Lectures/index.htm

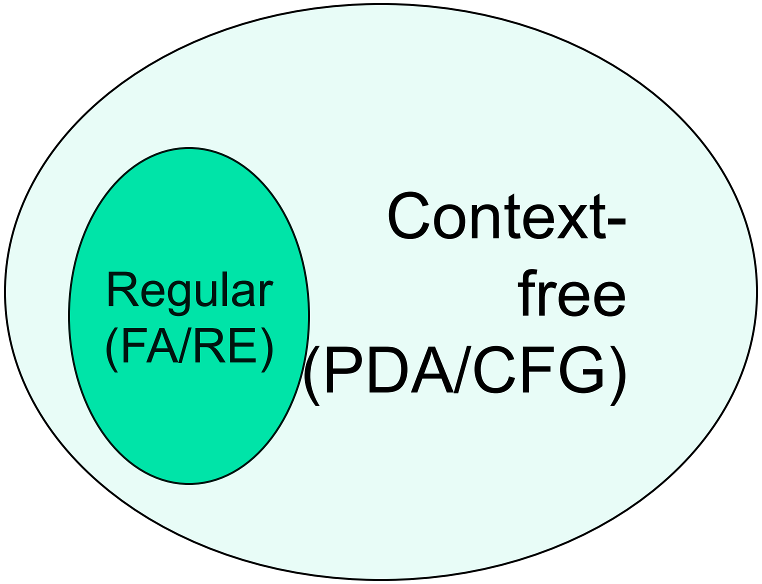
Not all languages are regular

- 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.
  - <u>Proof:</u>
    - Let $w=0^{N-1}010^N$ (assuming N to be the p/l constant)
    - By Pumping lemma, w can be rewritten as xyz ($x=0^{N-1}$, $y=0$, $z=10^N$), such that xy^kz is also L (for any $k\geq 0$)
    - But $|xy|\leq N$ and $y\neq\varepsilon$
    - ==> 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. 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 non-terminals
  - T: set of terminals (= alphabet U $\{\varepsilon\}$)
  - P: set of *productions,* each of which is of the form
    $V \implies \alpha_1 \mid \alpha_2 \mid \ldots$
    - Where each α_i is an arbitrary string of non-terminals and terminals
  - S: start variable

CFG for the language of binary palindromes:
$G=(\{A\},\{0,1\},P,A)$
P: $A \to 0\,A\,0 \mid 1\,A\,1 \mid 0 \mid 1 \mid \varepsilon$

More examples

- Parenthesis matching in code (see HW1)
- 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

Why is this problem important?
Example!

- Language of balanced paranthesis

  e.g., ()(((())))((()))….

- CFG?

G:
S => (S) | SS | ε

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

Example #4

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

if *C* **then** *S1* **else** *S2*

(Or)

if *C* **then** *S*

CFG?

G:
S -> if C then S else S | if C then S
S (start symbol),
S, C – nonterminals
if, then, else – terminals (+ other terminals depending on what you allow in C and S

Example #5

A serious programing language grammar:

`http://marvin.cs.uidaho.edu/Teaching/CS445/c-Grammar.pdf`

More examples (@seminar)

- $L_1 = \{0^n \mid n \geq 0\}$
- $L_2 = \{0^n \mid n \geq 1\}$
- $L_3 = \{0^i 1^j 2^k \mid i=j \text{ or } j=k, \text{ where } i,j,k \geq 0\}$
- $L_4 = \{0^i 1^j 2^k \mid i=j \text{ or } i=k, \text{ where } i,j,k \geq 1\}$

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)

More applications

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

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)

- Transitivity:

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

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

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

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)

How to prove that your CFGs are correct?

(using induction)

CFG & CFL

<u>G_{pal}:</u>
A => 0A0 | 1A1 | 0 | 1 | ε

- <u>Theorem:</u> A string w in (0|1)* is in $L(G_{pal})$, if and only if, w is a palindrome.
- <u>Proof:</u>
  - Use induction
    - (IF) on string length
    - (ONLY IF) on length of derivation

Connection between CFLs and RLs

What kind of grammars result for regular languages?

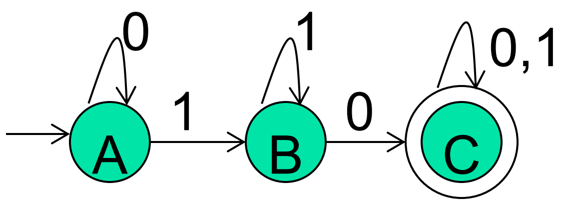
CFLs & Regular Languages

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

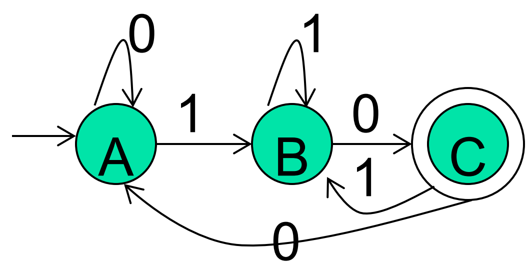
Where:
• A & B are variables,
• w is a **string** of terminals

- 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?

GFG: generative and analytical aspect

- Generative aspect: Use G to derive strings w in L(G).
- Analytical aspect: Given a CFG G and strings w, how do you decide if w in L(G) and, if so, how do you determine the derivation tree or the sequence of production rules that produce w? This is called the problem of parsing.

Parse trees

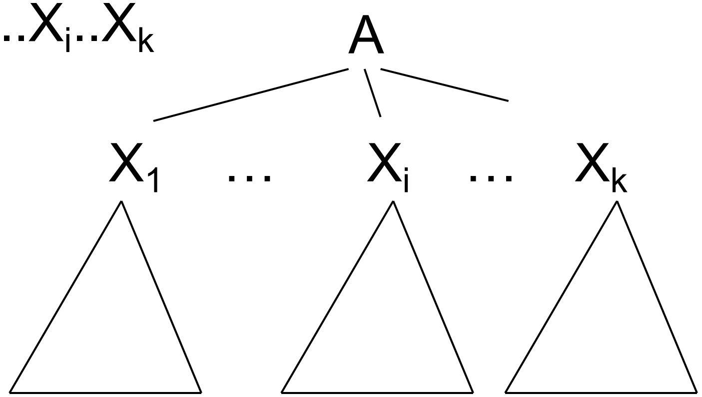
Parse Trees

- 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, $A ==> X_1 X_2 \ldots X_k$, then any internal node labeled A has k children which are labeled from $X_1, X_2, \ldots X_k$ from left to right

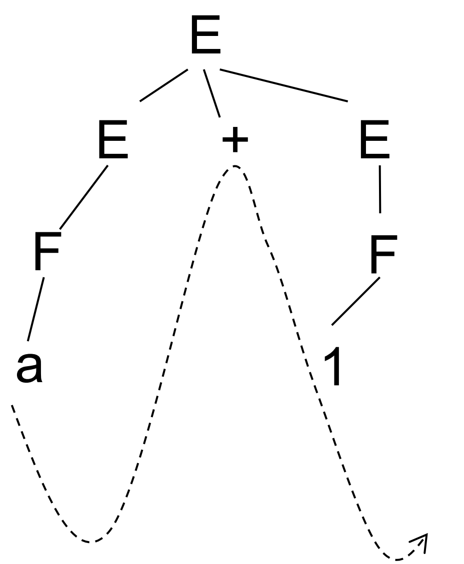
---

Parse tree for production and all other subsequent productions:

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

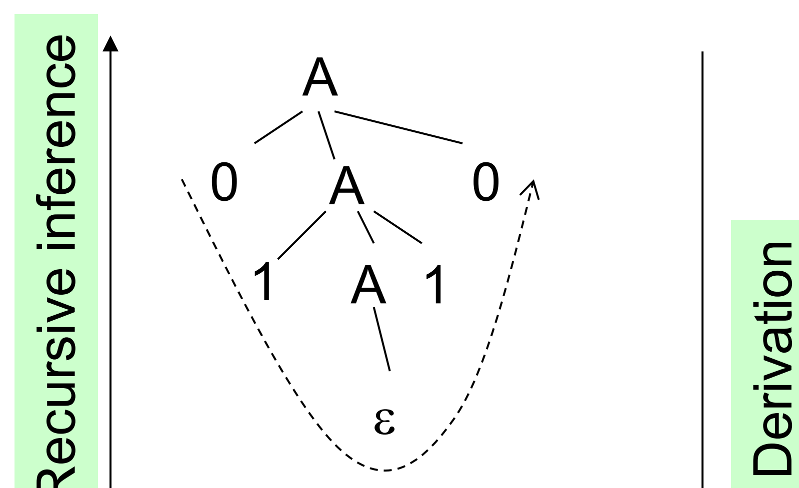


Examples



Parse tree for *a + 1*

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



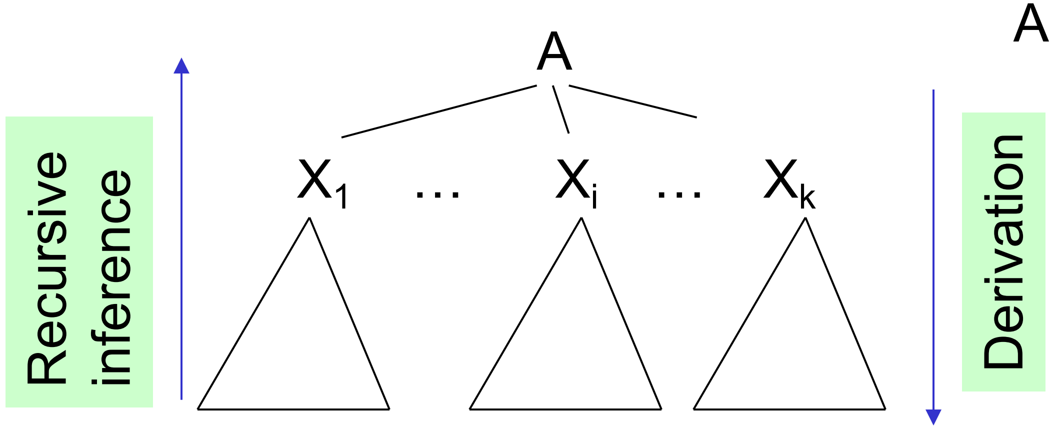
Parse tree for *0110*

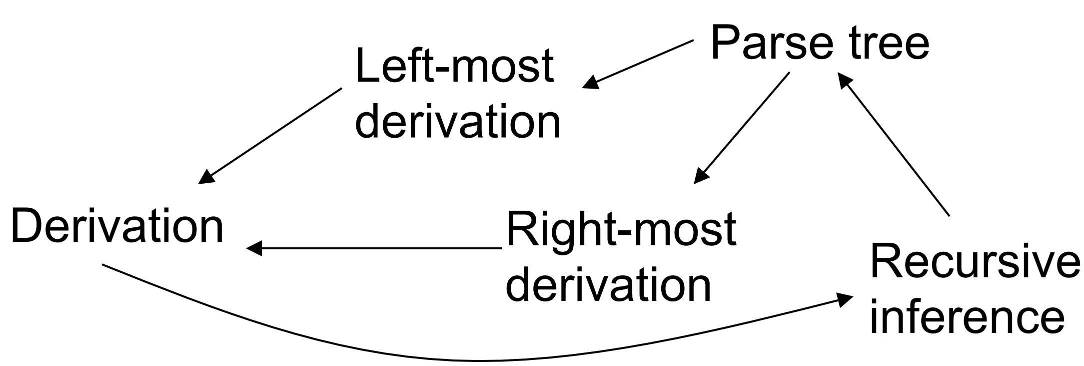
G:
A => 0A0 | 1A1 | 0 | 1 | ε

Parse Trees, Derivations, and Recursive Inferences

Production:

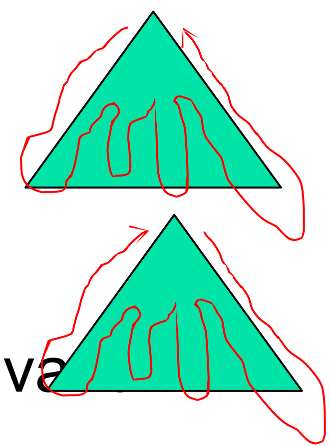
$A ==> X_1..X_i..X_k$





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 deriva
- Derivation ==> Recursive inference
  - Reverse the order of productions
- Recursive inference ==> Parse trees
  - bottom-up traversal of parse tree



More in the compiling techniques lecture!

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

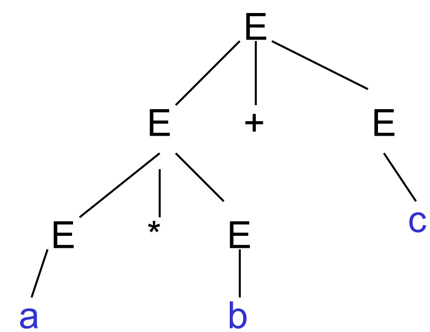
Why does ambiguity matter?

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

Values are different !!!

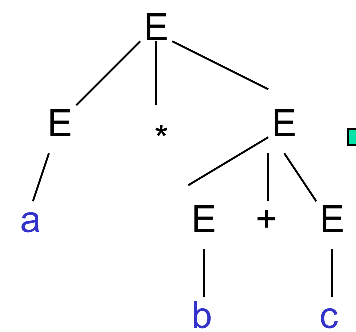
*string = a * b + c*

- 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
- Formal proof: **complex**
- Intuition (bonus points): 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