

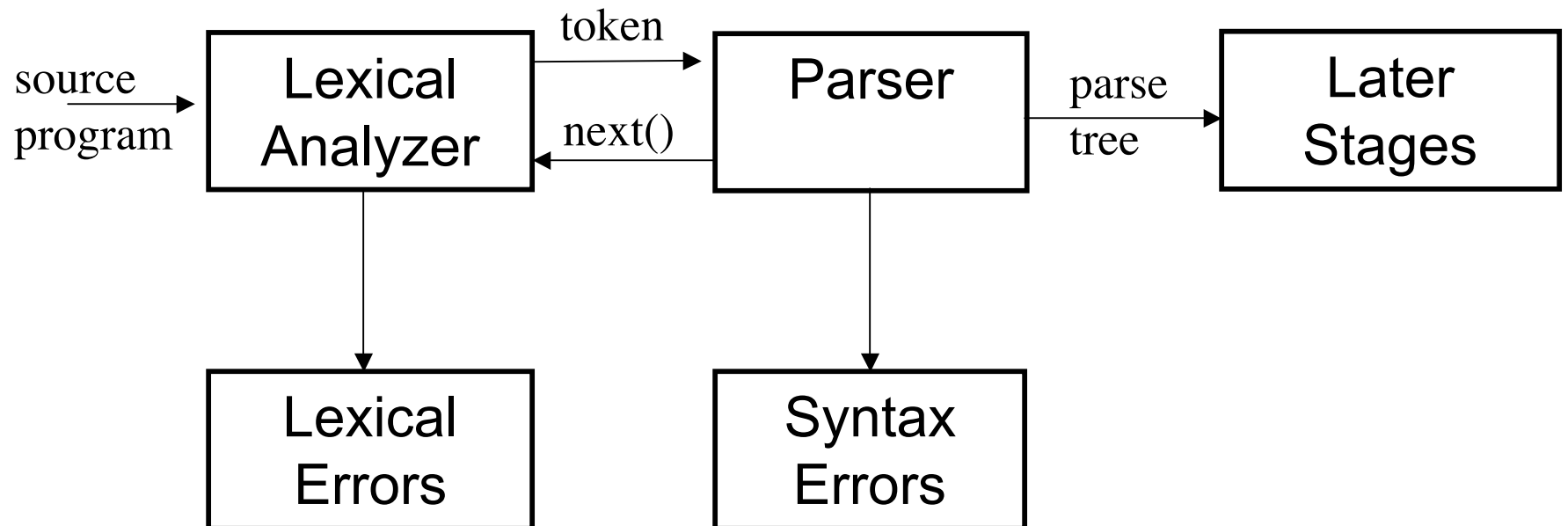
# CMPT 755

## Compilers

Anoop Sarkar

<http://www.cs.sfu.ca/~anoop>

# Parsing



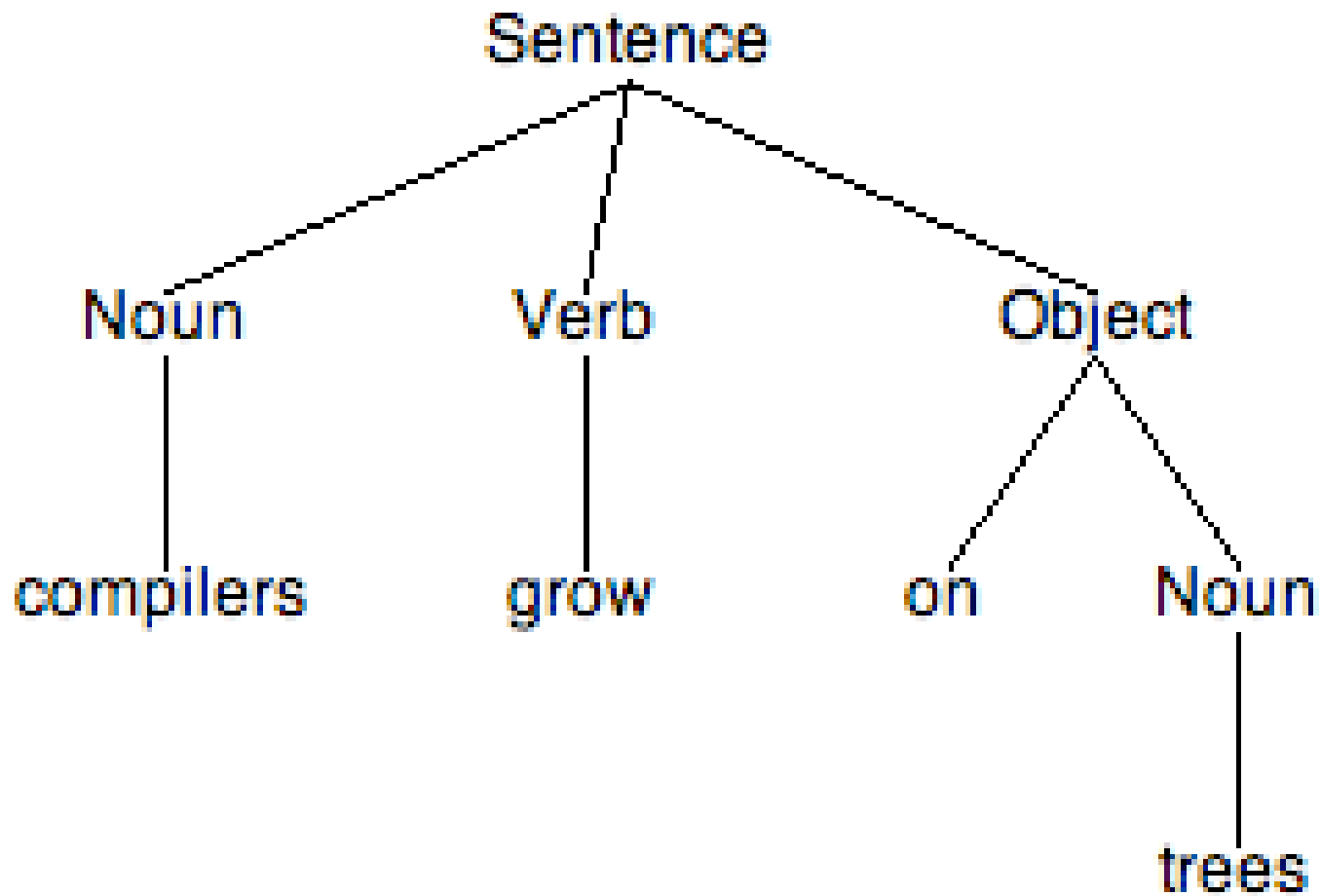
# Context-free Grammars

- Set of rules by which valid sentences can be constructed.
- Example:
  - Sentence  $\rightarrow$  Noun Verb Object
  - Noun  $\rightarrow$  *trees* | *compilers*
  - Verb  $\rightarrow$  *are* | *grow*
  - Object  $\rightarrow$  *on* Noun | Adjective
  - Adjective  $\rightarrow$  *slowly* | *interesting*
- What strings can Sentence *derive*?
- Syntax only – no semantic checking

# Derivations of a CFG

- *compilers grow on trees*
- *compilers grow on* **Noun**
- *compilers grow* **Object**
- *compilers* **Verb Object**
- **Noun Verb Object**
- **Sentence**

# Derivations and parse trees



# Why use grammars for PL?

- Precise, yet easy-to-understand specification of language
- Construct parser automatically
  - Detect potential problems
- Structure and simplify remaining compiler phases
- Allow for evolution

# CFG Notation

- A reference grammar is a concise description of a context-free grammar
- For example, a reference grammar can use regular expressions on the right hand sides of CFG rules
- Can even use ideas like comma-separated lists to simplify the reference language definition

# Writing a CFG for a PL

- First write (or read) a reference grammar of what you want to be valid programs
- For now, we only worry about the structure, so the reference grammar might choose to over-generate in certain cases (e.g. `bool x = 20;` )
- Convert the reference grammar to a CFG
- Certain CFGs might be easier to work with than others (this is the **essence** of the study of CFGs and their parsing algorithms for compilers)



# CFG Notation

- Normal CFG notation

$$E \rightarrow E * E$$

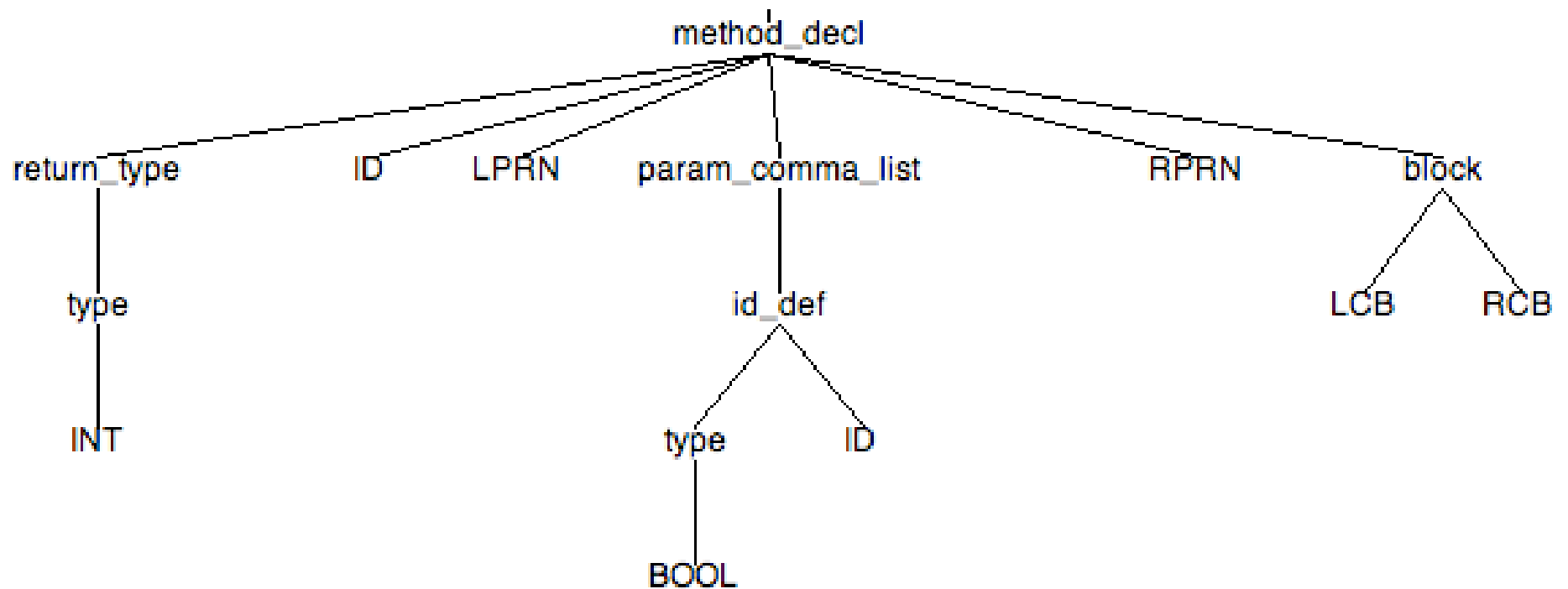
$$E \rightarrow E + E$$

- Backus Naur notation

$$E ::= E * E \mid E + E$$

(an or-list of right hand sides)

# Parse Trees for programs



# Arithmetic Expressions

- $E \rightarrow E + E$
- $E \rightarrow E * E$
- $E \rightarrow ( E )$
- $E \rightarrow - E$
- $E \rightarrow \mathbf{id}$

# Leftmost derivations for **id + id \* id**

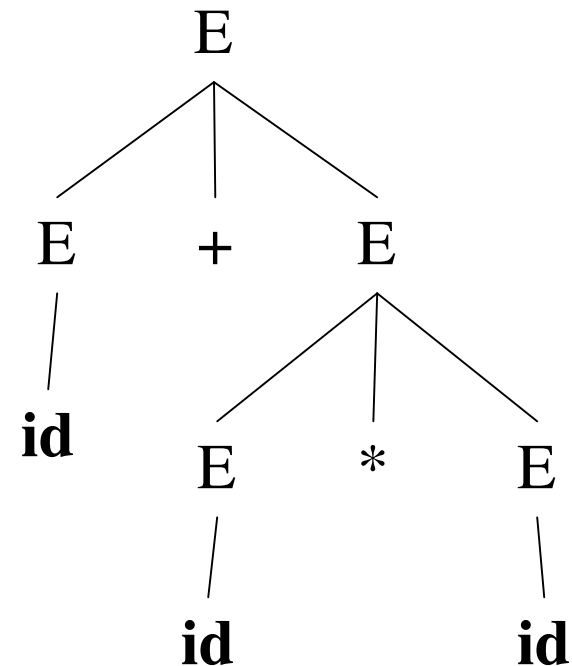
•  $E \Rightarrow E + E$

$\Rightarrow \text{id} + E$

$\Rightarrow \text{id} + E * E$

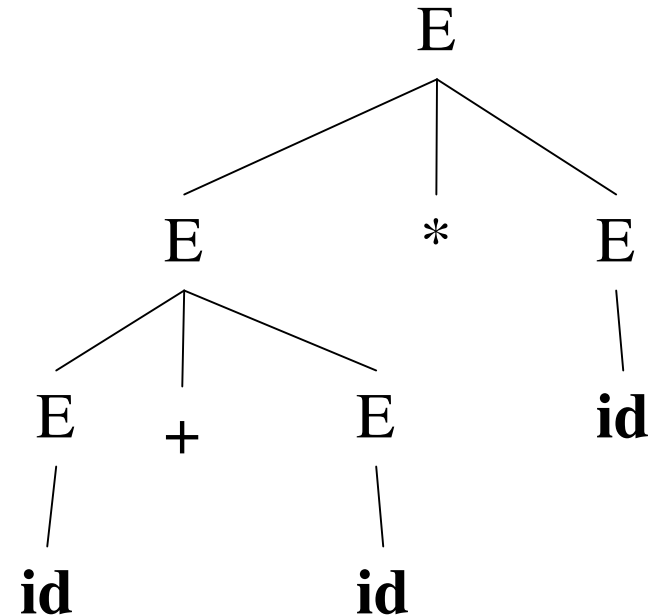
$\Rightarrow \text{id} + \text{id} * E$

$\Rightarrow \text{id} + \text{id} * \text{id}$



# Leftmost derivations for **id + id \* id**

- $E \Rightarrow E * E$   
 $\Rightarrow E + E * E$   
 $\Rightarrow \text{id} + E * E$   
 $\Rightarrow \text{id} + \text{id} * E$   
 $\Rightarrow \text{id} + \text{id} * \text{id}$



# Ambiguity

- Grammar is ambiguous if more than one parse tree is possible for some sentences
- Examples in English:
  - Two sisters reunited after 18 years in checkout counter
- Ambiguity is not acceptable in PL
  - Unfortunately, it's undecidable to check whether a grammar is ambiguous

# Ambiguity

- Alternatives
  - Massage grammar to make it unambiguous
  - Rely on “default” parser behavior
  - Augment parser
- Consider the original ambiguous grammar:
$$\begin{array}{ll} E \rightarrow E + E & E \rightarrow E * E \\ E \rightarrow ( E ) & E \rightarrow - E \\ E \rightarrow \mathbf{id} & \end{array}$$
- How can we change the grammar to get only one tree for the input **id + id \* id**

# Dangling else ambiguity

- Original Grammar (ambiguous)

Stmt  $\rightarrow$  **if** Expr **then** Stmt **else** Stmt

Stmt  $\rightarrow$  **if** Expr **then** Stmt

Stmt  $\rightarrow$  Other

- Unambiguous grammar

Stmt  $\rightarrow$  MatchedStmt

Stmt  $\rightarrow$  UnmatchedStmt

MatchedStmt  $\rightarrow$  **if** Expr **then** MatchedStmt **else** MatchedStmt

MatchedStmt  $\rightarrow$  Other

UnmatchedStmt  $\rightarrow$  **if** Expr **then** Stmt

UnmatchedStmt  $\rightarrow$  **if** Expr **then** MatchedStmt **else** UnmatchedStmt



# Dangling else ambiguity

- Original Grammar (ambiguous)

Stmt  $\rightarrow$  **if** Expr **then** Stmt **else** Stmt

Stmt  $\rightarrow$  **if** Expr **then** Stmt

Stmt  $\rightarrow$  Other

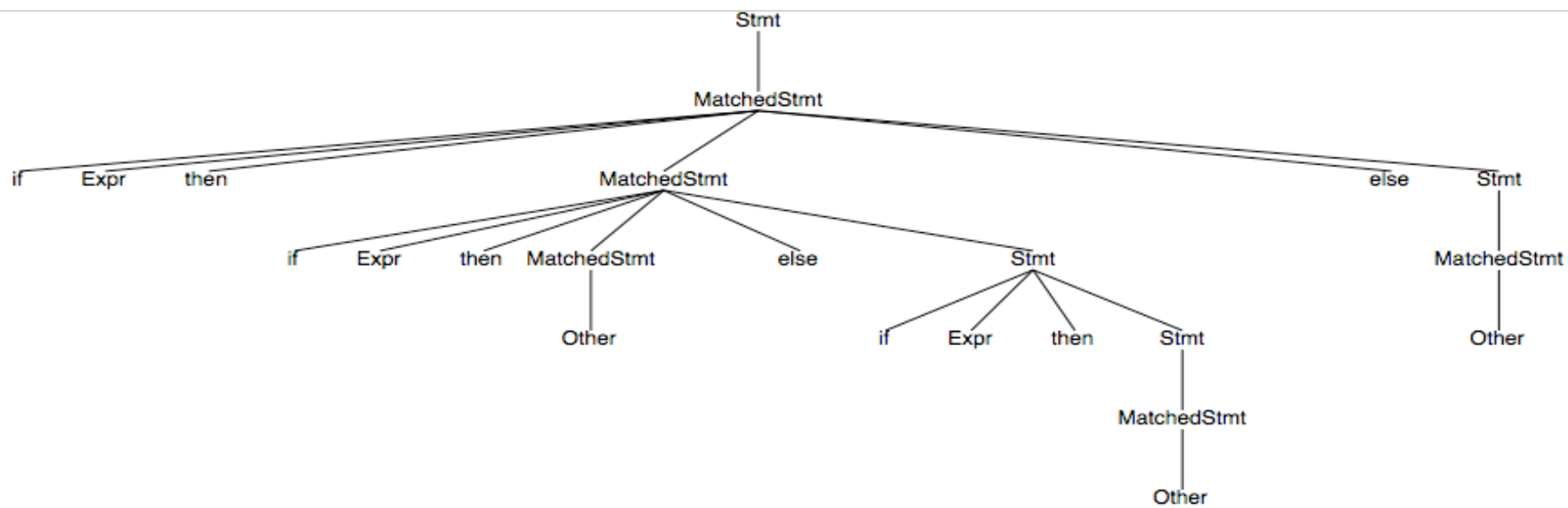
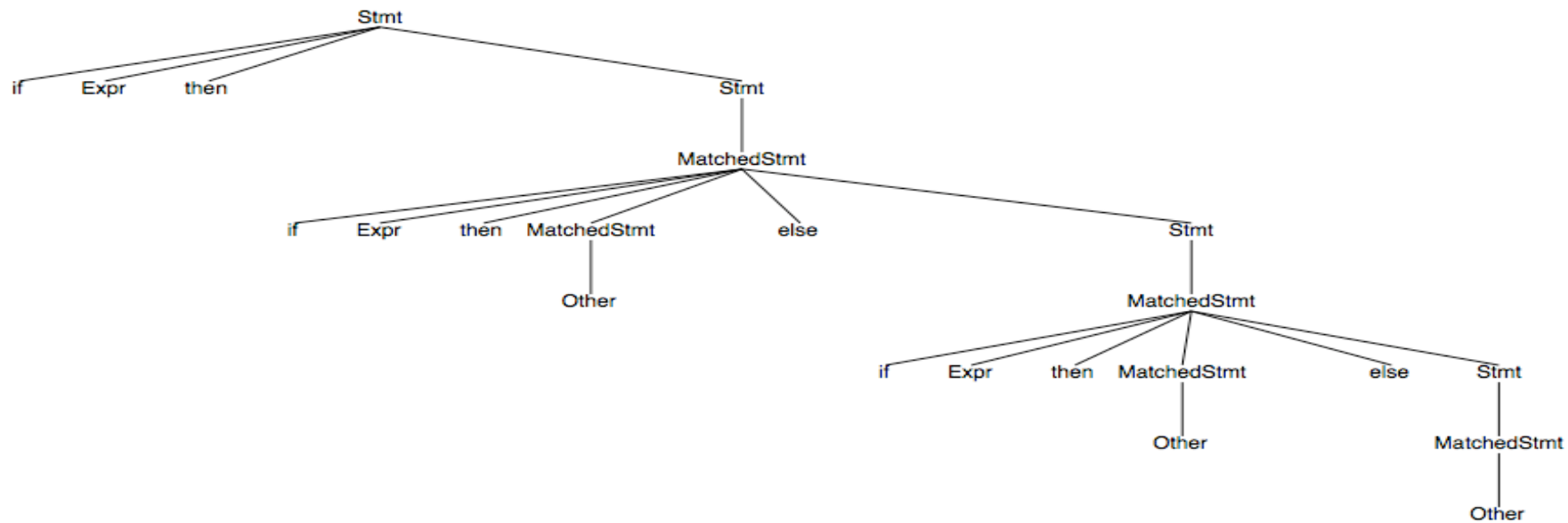
- Modified Grammar (unambiguous?)

Stmt  $\rightarrow$  **if** Expr **then** Stmt

Stmt  $\rightarrow$  MatchedStmt

MatchedStmt  $\rightarrow$  **if** Expr **then** MatchedStmt **else** Stmt

MatchedStmt  $\rightarrow$  Other



# Other Ambiguous Grammars

- Consider the grammar
$$R \rightarrow R \text{ '}' R \mid R R \mid R \text{ '*' } \mid \text{'(' } R \text{ ')'} \mid a \mid b$$
- What does this grammar generate?
- What's the parse tree for  $ab^*a$
- Is this grammar ambiguous?

# Left Factoring

- Original Grammar (ambiguous)

Stmt  $\rightarrow$  **if** Expr **then** Stmt **else** Stmt

Stmt  $\rightarrow$  **if** Expr **then** Stmt

Stmt  $\rightarrow$  Other

- Left-factored Grammar (still ambiguous):

Stmt  $\rightarrow$  **if** Expr **then** Stmt OptElse

Stmt  $\rightarrow$  Other

OptElse  $\rightarrow$  **else** Stmt  $\mid \epsilon$

# Left Factoring

- In general, for rules

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \dots \mid \alpha\beta_n \mid \gamma$$

- Left factoring is achieved by the following grammar transformation:

$$\begin{aligned} A &\rightarrow \alpha A' \mid \gamma \\ A' &\rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n \end{aligned}$$

# Grammar Transformations

- $G$  is converted to  $G'$  s.t.  $L(G') = L(G)$
- Left Factoring
- Removing cycles:  $A \Rightarrow^+ A$
- Removing  $\varepsilon$ -rules of the form  $A \rightarrow \varepsilon$
- Eliminating left recursion
- Conversion to normal forms:
  - Chomsky Normal Form,  $A \rightarrow B C$  and  $A \rightarrow a$
  - Greibach Normal Form,  $A \rightarrow a \beta$

# Eliminating Left Recursion

- Simple case, for left-recursive pair of rules:

$$A \rightarrow A\alpha \mid \beta$$

- Replace with the following rules:

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \mid \epsilon$$

- Elimination of immediate left recursion

# Eliminating Left Recursion

- Example:

$$E \rightarrow E + T, E \rightarrow T$$

- Without left recursion:

$$E \rightarrow T E_1, E_1 \rightarrow + T E_1, E_1 \rightarrow \varepsilon$$

- Simple algorithm doesn't work for 2-step recursion:

$$S \rightarrow A a, S \rightarrow b$$

$$A \rightarrow A c, A \rightarrow S d, A \rightarrow \varepsilon$$



# Eliminating Left Recursion

- Problem CFG:  
 $S \rightarrow A a, S \rightarrow b$   
 $A \rightarrow A c, A \rightarrow S d, A \rightarrow \varepsilon$
- Expand possibly left-recursive rules:  
 $S \rightarrow A a, S \rightarrow b$   
 $A \rightarrow A c, A \rightarrow A a d, A \rightarrow b d, A \rightarrow \varepsilon$
- Eliminate immediate left-recursion  
 $S \rightarrow A a, S \rightarrow b$   
 $A \rightarrow b d A_1, A \rightarrow A_1, A_1 \rightarrow c A_1, A_1 \rightarrow a d A_1, A_1 \rightarrow \varepsilon$

# Eliminating Left Recursion

- We cannot use the algorithm if the non-terminal also derives epsilon. Let's see why:

$$A \rightarrow AAa \mid b \mid \varepsilon$$

- Using the standard lrec removal algorithm:

$$A \rightarrow bA_1 \mid A_1$$

$$A_1 \rightarrow AaA_1 \mid \varepsilon$$

# Eliminating Left Recursion

- First we eliminate the epsilon rule:

$$A \rightarrow AAa \mid b \mid \varepsilon$$

- Since  $A$  is the start symbol, create a new start symbol to generate the empty string:

$$A_1 \rightarrow A \mid \varepsilon \quad A \rightarrow AAa \mid Aa \mid a \mid b$$

- Now we can do the usual lrec algorithm:

$$A_1 \rightarrow A \mid \varepsilon \quad A \rightarrow aA_2 \mid bA_2$$

$$A_2 \rightarrow AaA_2 \mid aA_2 \mid \varepsilon$$

# Non-CF Languages

- The pumping lemma for CFLs [Bar-Hillel] is similar to the pumping lemma for RLs
- For a string  $wuxvy$  in a CFL for  $u, v \neq \varepsilon$  and the string is long enough then  $wu^n xv^n y$  is also in the CFL for  $n \geq 0$
- Not strong enough to work for every non-CF language (cf. Ogden's Lemma)

# Non-CF Languages

$$L_1 = \{w cw \mid w \in (a|b)^*\}$$

$$L_2 = \{a^n b^m c^n d^m \mid n \geq 1, m \geq 1\}$$

$$L_3 = \{a^n b^n c^n \mid n \geq 0\}$$

# CF Languages

$$L_4 = \{wcw^R \mid w \in (a|b)^*\}$$

$$S \rightarrow aSa \mid bSb \mid c$$

$$L_5 = \{a^n b^m c^m d^n \mid n \geq 1, m \geq 1\}$$

$$S \rightarrow aSd \mid aAd$$

$$A \rightarrow bAc \mid bc$$

# Summary

- CFGs can be used describe PL
- Derivations correspond to parse trees
- Parse trees represent structure of programs
- Ambiguous CFGs exist
- Some forms of ambiguity can be fixed by changing the grammar
- Grammars can be simplified by left-factoring
- Left recursion in a CFG can be eliminated