# CMSC 330: Organization of Programming Languages

Context-Free Grammars, con't.

## Tips for Designing Grammars (cont'd)

4. For a language that's the union of other languages, use separate nonterminals for each part of the union and then combine

$$\{ a^n(b^m|c^m) \mid m > n \geq 0\}$$

Can be rewritten as

$$\{ a^nb^m \mid m > n \geq 0\} \cup \{ a^nc^m \mid m > n \geq 0\}$$



## Tips for Designing Grammars (cont'd)

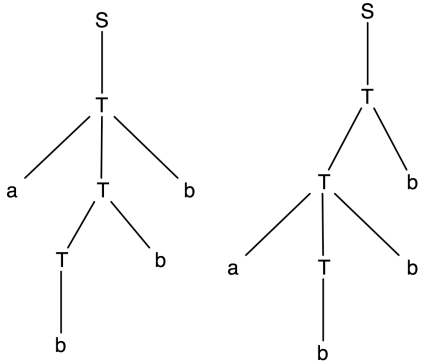
$\{ a^nb^m \mid m > n \geq 0\} \cup \{ a^nc^m \mid m > n \geq 0\}$

$S \rightarrow T \mid U$

$T \rightarrow aTb \mid Tb \mid b$ T generates the first set of strings

$U \rightarrow aUc \mid Uc \mid c$ U generates the second set of strings

- What about the string abbb?
  - Ambiguous!






## Tips for Designing Grammars (cont'd)

$\{ a^nb^m \mid m > n \geq 0\} \cup \{ a^nc^m \mid m > n \geq 0\}$

Will this fix the ambiguity?

$S \rightarrow T \mid U$

$T \rightarrow aTb \mid bT \mid b$

$U \rightarrow aUc \mid cU \mid c$

- It's not ambiguous, but it can generate invalid strings such as babb

# Tips for Designing Grammars (cont'd)

$\{ a^n b^m \mid m > n \geq 0 \} \cup \{ a^n c^m \mid m > n \geq 0 \}$

Unambiguous version

$S \rightarrow T \mid V$
$T \rightarrow aTb \mid U$
$U \rightarrow Ub \mid b$
$V \rightarrow aVc \mid W$
$W \rightarrow Wc \mid c$



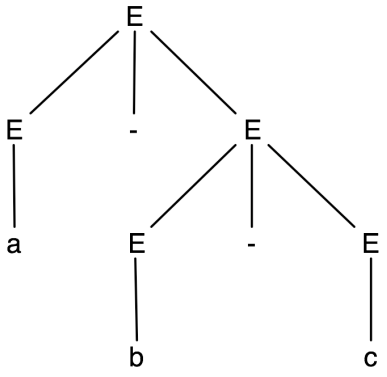
# CFGs for Languages

- Recall that our goal is to describe programming languages with CFGs
- We had the following example which describes limited arithmetic expressions
  $E \rightarrow a \mid b \mid c \mid E+E \mid E-E \mid E*E \mid (E)$
- What's wrong with using this grammar?
  - It's ambiguous!



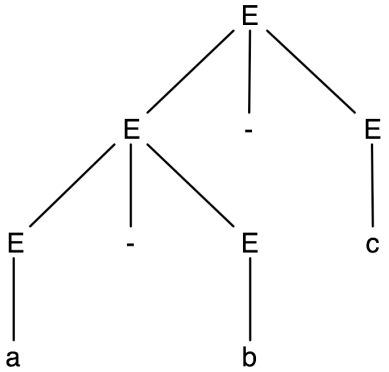
# Example: a-b-c (E → a | b | c | E+E | E-E | E*E | (E))

E ⇒ E-E ⇒ a-E ⇒ a-E-E ⇒
a-b-E ⇒ a-b-c




Corresponds to a-(b-c)

E ⇒ E-E ⇒ E-E-E ⇒
a-E-E ⇒ a-b-E ⇒ a-b-c




Corresponds to (a-b)-c



# The Issue: Associativity

- Ambiguity is bad here because if a compiler needs to generate code for this expression, it doesn't know what the programmer intended
- So what do we mean when we write a-b-c?
  - In mathematics, this has only one meaning- it's (a-b)-c, since subtraction is *left-associative*
  - a-(b-c) would be the meaning if subtraction was *right-associative*
- Two approaches to handle ambiguity:
  - Rewrite the grammar
  - Use special parsing rules, depending on the parsing method being used (covered in CMSC 430)

# Another Example: If-Then-Else

<stmt> ::= <assignment> | <if-stmt> | ...

<if-stmt> ::= if (<expr>) <stmt> |
if (<expr>) <stmt> else <stmt>

– (Here <>'s are used to denote nonterminals and ::= for productions)

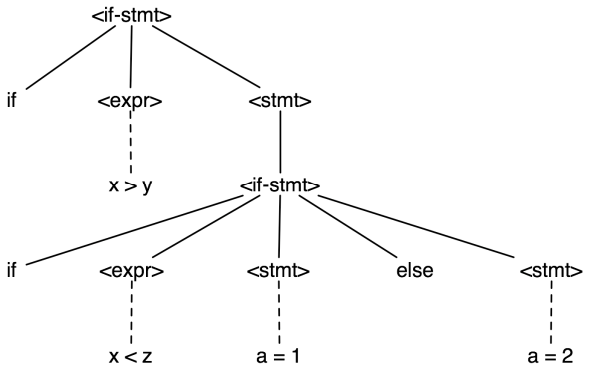
- Consider the following program fragment:

```
if (x > y)
  if (x < z)
    a = 1;
  else a = 2;
```

– Note: ignore spaces and newlines



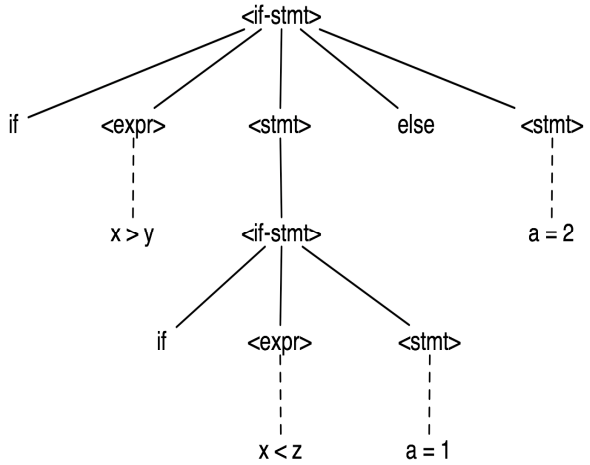
# Parse Tree #1




- else belongs to inner if



# Parse Tree #2

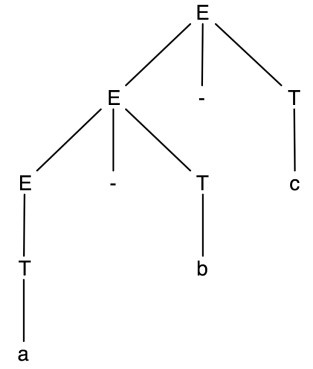



- else belongs to outer if



# Fixing the Expression Grammar

- Idea: Require that the right operand of all of the operators is not a bare expression
  – E → E+T | E-T | E*T | T
  – T → a | b | c | (E)




- Now there's only one parse tree for a-b-c

– Exercise: Give a derivation in this grammar for the string a-(b-c)

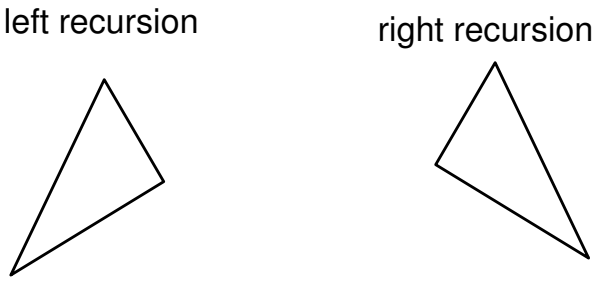
# What if We Wanted Right-Associativity?

- Left-recursive productions are used for left-associative operators
- Right-recursive productions are used for right-associative operators
- Left:
  - E → E+T | E-T | E*T | T
  - T → a | b | c | (E)
- Right:
  - E → T+E | T-E | T*E | T
  - T → a | b | c | (E)



# Parse Tree Shape

- The kind of recursion/associativity determines the shape of the parse tree




  - Exercise: draw a parse tree for a-b-c in the prior grammar in which subtraction is right-associative



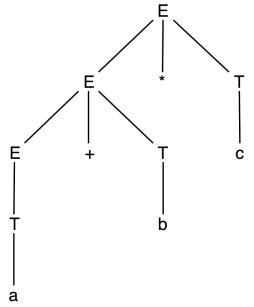
# A Different Problem

- How about the string a+b*c?

  The grammar was

  E → E+T | E-T | E*T | T

  T → a | b | c | (E)




- Doesn't have correct precedence for *
  - When a nonterminal has productions for several operators, they effectively have the same precedence
- How can we fix this?



# Final Expression Grammar

| | |
|---|---|
| E → E+T \| E-T \| T | lowest precedence operators |
| T → T*P \| P | higher precedence |
| P → a \| b \| c \| (E) | highest precedence (parentheses) |

- Exercises:
  - Construct tree and left and and right derivations for

    a+b*c  a*(b+c)  a*b+c  a-b-c
  - See what happens if you change the last set of productions to P → a | b | c | E | (E)
  - See what happens if you change the first set of productions to E → E +T | E-T | T | P

# Regular expressions and CFGs

| | Description | Machine |
|---|---|---|
| regular languages | regular expressions | DFAs, NFAs |
| context-free languages | context-free grammars | pushdown automata (PDAs) |

- Programming languages are neither regular nor context-free
  - Usually almost context-free, with some hacks



# Context-free Grammars in Practice

- Regular expressions are used to turn raw text into a string of *tokens*
  - E.g., “if”, “then”, “identifier”, etc.
  - Whitespace and comments are simply skipped
  - These tokens are the input for the next phase of compilation
  - Standard tools used are lex and flex
- CFGs are used to turn tokens into parse trees
  - This process is called *parsing*
  - Standard tools used are yacc and bison
- Those trees are then analyzed by the compiler, which eventually produces object code