# CMSC 330: Organization of Programming Languages

Context Free Grammars

Interpreters

- Tokenizer (Scanner / lexer):
  - converts program source into tokens (keywords, variable names, operators, numbers, etc.) using regular expressions
  - "2*7+3" ⟹ [Num 2; Mul; Num 7; Plus; Num 3]
- Parser
  - recognize strings defined as context free grammars
  - converts tokens into an Abstract Syntax Tree (AST)

- Evaluator
  - Runs the program
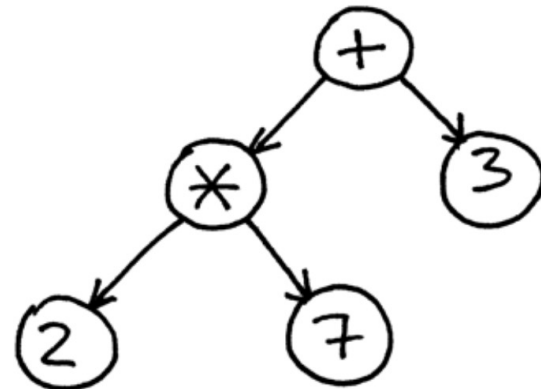
Context-Free Grammar (CFG)

- A way of describing sets of strings (= languages)
  - The notation L(G) denotes the language of strings defined by grammar G

- Example grammar G is S → 0S | 1S | ε

  which says that string s' ϵ L(G) iff
  - s' = ε, or ∃s ϵ L(G) such that s' = 0s, or s' = 1s

- Grammar is same as regular expression (0|1)*

CFGs Are Expressive

▸ CFGs subsume REs, DFAs, NFAs

▸ And CFGs can define languages regexps cannot
- $S \rightarrow (S) \mid \varepsilon$ // represents balanced pairs of ( )'s

▸ As a result, CFGs often used as the basis of parsers for programming languages

Parsing with CFGs

- CFGs formally define languages, but they do not define an *algorithm* for accepting strings

- Several styles of algorithm; each works only for less expressive forms of CFG

  - LL(k) parsing   ⟵   We will discuss this next lecture
  - LR(k) parsing
  - LALR(k) parsing
  - SLR(k) parsing

- Tools exist for building parsers from grammars

  - JavaCC, Yacc, etc.

Formal Definition: Context-Free Grammar

▸ A CFG G is a 4-tuple (Σ, N, P, S)

- Σ – alphabet (finite set of symbols, or terminals)
  - ➢ Often written in lowercase

- N – a finite, nonempty set of nonterminal symbols
  - ➢ Often written in UPPERCASE
  - ➢ It must be that N ∩ Σ = ∅

- P – a set of productions of the form *N → (Σ|N)**
  - ➢ Informally: the nonterminal can be replaced by the string of zero or more terminals / nonterminals to the right of the →
  - ➢ Can think of productions as rewriting rules (more later)

- S ϵ N – the start symbol

Notational Shortcuts

S → aAb
A → aA | b | ε

- A production is of the form
  - left-hand side (LHS) → right hand side (RHS)
- If not specified
  - Assume LHS of first production is the start symbol
- Productions with the same LHS
  - Are usually combined with |
- If a production has an empty RHS
  - It means the RHS is ε

Aside: Backus-Naur Form
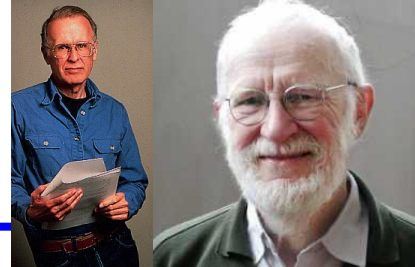
- Context-free grammar production rules are also called Backus-Naur Form or BNF
  - Designed by John Backus and Peter Naur
    - Chair and Secretary of the Algol committee in the early 1960s. Used this notation to describe Algol in 1962

- A production   A → B c D
  is written in BNF as   \<A\> ::= \<B\> c \<D\>

Java Syntax Specification

Programs

`<compilation unit>` ::= `<package declaration>?` `<import declarations>?` `<type declarations>?`

`<type declaration>` ::= `<class declaration>` | `<interface declaration>` | ;

`<class declaration>` ::= `<class modifiers>?` class `<identifier>` `<super>?` `<interfaces>?`
`<class body>`

`<class modifier>` ::= public | abstract | final
`<super>` ::= extends `<class type>`
`<interfaces>` ::= implements `<interface type list>`

`<if then else statement>`::= if (`<expression>`) `<statement no short if>` else `<statement>`

`<while statement>` ::= while (`<expression>`) `<statement>`

`<for statement>` ::= for (`<for init>?` ; `<expression>?` ; `<for update>?`) `<statement>`

Generating Strings

- Think of a grammar as generating strings by rewriting
  - Beginning with the start symbol, repeatedly rewrite a nonterminal per a production in the grammar (replace LHS with RHS)

- Example grammar G

  S → 0S | 1S | ε

- Generate string 011 from G as follows:

  S ⇒ 0S   // using S → 0S
    ⇒ 01S   // using S → 1S
    ⇒ 011S   // using S → 1S
    ⇒ 011   // using S → ε

Accepting Strings (Informally)

- Checking if s ∈ L(G) is called acceptance
  - Algorithm: Find a rewriting from G's start symbol that yields s
    - 011 ∈ L(G) according to the previous rewriting

- Terminology
  - Such a sequence of rewrites is a derivation or parse
  - Discovering the derivation is called parsing

Derivations

- Notation

  - $\Rightarrow$ indicates a derivation of one step

  - $\Rightarrow^+$ indicates a derivation of one or more steps

  - $\Rightarrow^*$ indicates a derivation of zero or more steps

- Example

  - $S \rightarrow 0S \mid 1S \mid \varepsilon$

- For the string 010

  - $S \Rightarrow 0S \Rightarrow 01S \Rightarrow 010S \Rightarrow 010$

  - $S \Rightarrow^+ 010$

  - $010 \Rightarrow^* 010$

Language Generated by Grammar

▸ L(G) the language defined by G is

$$L(G) = \{ \, s \, \epsilon \, \Sigma^* \mid S \Rightarrow^+ s \, \}$$

- • S is the start symbol of the grammar
- • Σ is the alphabet for that grammar

▸ In other words

- • All strings over Σ that can be derived from the start symbol via one or more productions

Quiz #1

Consider the grammar

$S \rightarrow bS \mid T$

$T \rightarrow aT \mid U$

$U \rightarrow cU \mid \varepsilon$

Which of the following strings is generated by this grammar?

A. aba

B. bab

C. accc

D. ca

Quiz #1

Consider the grammar

$S \rightarrow bS \mid T$

$T \rightarrow aT \mid U$

$U \rightarrow cU \mid \varepsilon$

Which of the following strings is generated by this grammar?

A. aba

B. bab

C. accc

D. ca

Quiz #2

- Consider the grammar

    S → bS | T

    T → aT | U

    U → cU | ε

- Which of the following is a derivation of the string aac?

    A. S ⇒ T ⇒ aT ⇒ aTaT ⇒ aaT ⇒ aacU ⇒ aac

    B. S ⇒ T ⇒ aT ⇒ aaT ⇒ aaU ⇒ aacU ⇒ aac

    C. S ⇒ T ⇒ U ⇒ aU ⇒ aaU ⇒ aacU ⇒ aac

    D. S ⇒ aT ⇒ aaT ⇒ aaU ⇒ aacU ⇒ aac

Quiz #2

- Consider the grammar

  S → bS | T

  T → aT | U

  U → cU | ε

- Which of the following is a derivation of the string aac?

  A. S ⇒ T ⇒ aT ⇒ aTaT ⇒ aaT ⇒ aacU ⇒ aac

  B. S ⇒ T ⇒ aT ⇒ aaT ⇒ aaU ⇒ aacU ⇒ aac

  C. S ⇒ T ⇒ U ⇒ aU ⇒ aaU ⇒ aacU ⇒ aac

  D. S ⇒ aT ⇒ aaT ⇒ aaU ⇒ aacU ⇒ aac

Quiz #3

Consider the grammar

$$S \rightarrow aS \mid T$$
$$T \rightarrow bT \mid U$$
$$U \rightarrow cU \mid \varepsilon$$

Which of the following regular expressions accepts the same language as this grammar?

A. (a|b|c)*

B. a*b*c*

C. (a|ab|abc)*

D. abc*

Quiz #3

Consider the grammar

$S \rightarrow$ aS | T

$T \rightarrow$ bT | U

$U \rightarrow$ cU | ε

Which of the following regular expressions accepts the same language as this grammar?

A. (a|b|c)*

B. a*b*c*

C. (a|ab|abc)*

D. abc*

Designing Grammars

1. Use recursive productions to generate an arbitrary number of symbols

   A → xA | ε   // Zero or more x's

   A → yA | y   // One or more y's

2. Use separate non-terminals to generate disjoint parts of a language, and then combine in a production

   a*b*   // a's followed by b's

   S → AB

   A → aA | ε   // Zero or more a's

   B → bB | ε   // Zero or more b's

Designing Grammars

3. To generate languages with matching, balanced, or related numbers of symbols, write productions which generate strings from the middle

$\{a^n b^n \mid n \geq 0\}$ // N a's followed by N b's

$S \rightarrow aSb \mid \varepsilon$

Example derivation: $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$

$\{a^n b^{2n} \mid n \geq 0\}$ // N a's followed by 2N b's

$S \rightarrow aSbb \mid \varepsilon$

Example derivation: $S \Rightarrow aSbb \Rightarrow aaSbbbb \Rightarrow aabbbb$

Designing Grammars

4. For a language that is 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

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

$S \rightarrow T \mid V$

$T \rightarrow aTb \mid U$

$U \rightarrow Ub \mid b$

$V \rightarrow aVc \mid W$

$W \rightarrow Wc \mid c$

Practice

▸ **Try to make a grammar which accepts**

- 0*|1*   • 0^n1^n where $n \geq 0$

$$S \to A \,|\, B$$
$$A \to 0A \,|\, \varepsilon$$
$$B \to 1B \,|\, \varepsilon$$

$$S \to 0S1 \,|\, \varepsilon$$

▸ **Give some example strings from this language**

- $S \to 0 \,|\, 1S$
  - ➤ 0, 10, 110, 1110, 11110, …
- What language is it, as a regexp?
  - ➤ 1*0

Quiz #4

Which of the following grammars describes the same language as 0^n1^m where $m \leq n$?

    A. $S \to 0S1 \mid \varepsilon$

    B. $S \to 0S1 \mid S1 \mid \varepsilon$

    C. $S \to 0S1 \mid 0S \mid \varepsilon$

    D. $S \to SS \mid 0 \mid 1 \mid \varepsilon$

Quiz #4

Which of the following grammars describes the same language as $0^n 1^m$ where $m \leq n$?

A. $S \rightarrow 0S1 \mid \varepsilon$ *same number of 0 and 1*

B. $S \rightarrow 0S1 \mid S1 \mid \varepsilon$ *more 1's*

C. **$S \rightarrow 0S1 \mid 0S \mid \varepsilon$**   *more 0's*

D. $S \rightarrow SS \mid 0 \mid 1 \mid \varepsilon$ *no control of the number*

Parse Trees

▶ Parse tree shows how a string is produced by a grammar

S → S*S | S+S | x | y | z
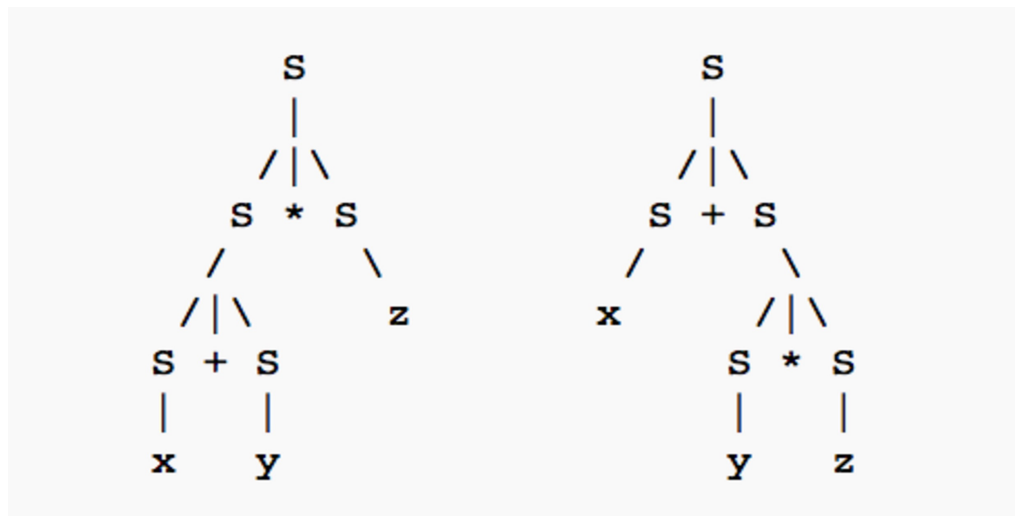
Parse Tree Example

S

S

$S \rightarrow aS \mid T$

$T \rightarrow bT \mid U$

$U \rightarrow cU \mid \varepsilon$

Parse Tree Example

$$S \Rightarrow aS$$

$S \rightarrow aS \mid T$
$T \rightarrow bT \mid U$
$U \rightarrow cU \mid \varepsilon$
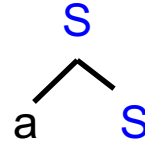
Parse Tree Example

$$S \Rightarrow aS \Rightarrow aT$$

$S \rightarrow aS \mid T$

$T \rightarrow bT \mid U$

$U \rightarrow cU \mid \varepsilon$

```
        S
       / \
      a   S
          |
          T
```
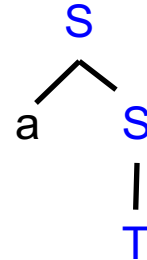
Parse Tree Example

$$S \Rightarrow aS \Rightarrow aT \Rightarrow aU$$

$S \rightarrow aS \mid T$
$T \rightarrow bT \mid U$
$U \rightarrow cU \mid \varepsilon$

```
        S
       / \
      a   S
          |
          T
          |
          U
```
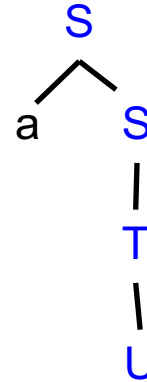
Parse Tree Example

$$S \Rightarrow aS \Rightarrow aT \Rightarrow aU \Rightarrow acU$$

S → aS | T
T → bT | U
U → cU | ε
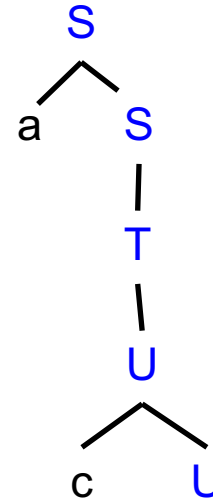
Parse Tree Example

$$S \Rightarrow aS \Rightarrow aT \Rightarrow aU \Rightarrow acU \Rightarrow ac$$

$S \rightarrow aS \mid T$

$T \rightarrow bT \mid U$

$U \rightarrow cU \mid \varepsilon$
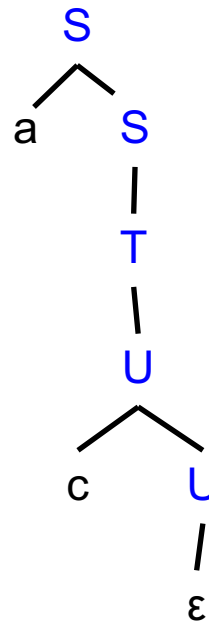
Arithmetic Expressions

- E → a | b | c | E+E | E-E | E*E | (E)

- This describes (or generates) a set of strings
  - {a, b, c, a+b, a+a, a*c, a-(b*a), c*(b + a), …}

- Example strings not in the language
  - d, c(a), a+, b**c, etc.

CFGs and ASTs

- An abstract syntax tree is a data structure that represents a parsed input, e.g., a program expression
  - An AST can be expressed with an OCaml datatype that is very close to the CFG that describes the language syntax

CFG for arithmetic expressions:

- E → a | b | c | d
      | E+E
      | E-E
      | E*E
      | (E)

AST:

```
type expr = A | B | C | D
  | Plus of expr * expr
  | Minus of expr * expr
  | Mult of expr * expr
```

Eventual Goal: Parse a CFG to get an AST

CFG (string):

- E → a | b | c | d
  - | E+E
  - | E-E
  - | E*E
  - | (E)

AST definition (OCaml):

```
type expr = A | B | C | D
  | Plus of expr * expr
  | Minus of expr * expr
  | Mult of expr * expr
```

a-c   parses to   `Minus (A, C)`

a-(b*a)   parses to   `Minus (A, Mult (B,A))`

c*(b+d)   parses to   `Mult (C, Plus (B,D))`

Parse Trees for Expressions

- A parse tree shows the structure of an expression as it corresponds to a grammar

  E → a | b | c | d | E+E | E-E | E*E | (E)

         a   a*c   c*(b+d)

Parse Trees for Expressions

▶ A parse tree shows the structure of an expression as it corresponds to a grammar
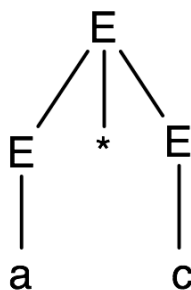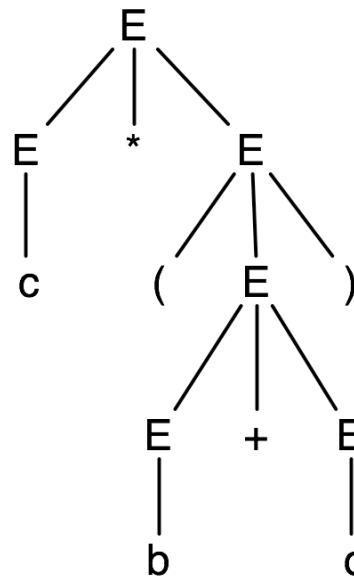
$E \rightarrow a \mid b \mid c \mid d \mid E+E \mid E-E \mid E*E \mid (E)$

Abstract Syntax Trees

▸ A parse tree and an AST are not the same thing

- The latter is a data structure produced by parsing

a*c
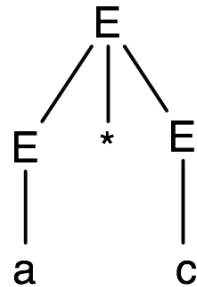
c*(b+d)
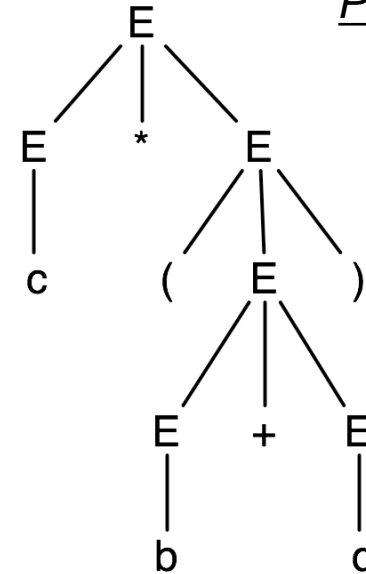
Parse trees

ASTs
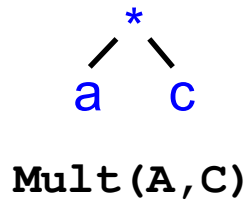
Mult(A,C)
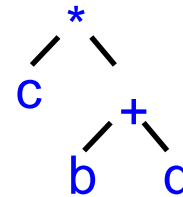
Mult(C,Plus(B,D))

Practice

E → a | b | c | d | E+E | E-E | E*E | (E)

Make a parse tree for…

- a*b
- a+(b-c)
- d*(d+b)-a
- (a+b)*(c-d)
- a+(b-c)*d

Leftmost and Rightmost Derivation

▶ Leftmost derivation
- Leftmost nonterminal is replaced in each step

▶ Rightmost derivation
- Rightmost nonterminal is replaced in each step

▶ Grammar

$S \rightarrow AB$

$A \rightarrow a$

$B \rightarrow b$

Leftmost derivation for "ab"
$$S \Rightarrow AB \Rightarrow aB \Rightarrow ab$$

Rightmost derivation for "ab"
$$S \Rightarrow AB \Rightarrow Ab \Rightarrow ab$$

Parse Tree For Derivations

▶ **Parse tree may be same for both leftmost & rightmost derivations**

- Example Grammar: S → a | SbS   String: aba

  Leftmost Derivation

     S ⇒ SbS ⇒ abS ⇒ aba

  Rightmost Derivation

     S ⇒ SbS ⇒ Sba ⇒ aba

  ```
        S
       /|\
      S b S
      |   |
      a   a
  ```
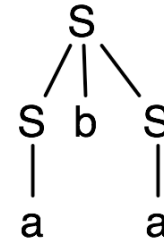
- Parse trees don't show order productions are applied

• **Every parse tree has a unique leftmost and a unique rightmost derivation**

Parse Tree For Derivations (cont.)

▶ Not every string has a unique parse tree

- Example Grammar: S → a | SbS   String: ababa

  Leftmost Derivation

  S ⇒ SbS ⇒ abS ⇒ abSbS ⇒ ababS ⇒ ababa

  Another Leftmost Derivation

  S ⇒ SbS ⇒ SbSbS ⇒ abSbS ⇒ ababS ⇒ ababa
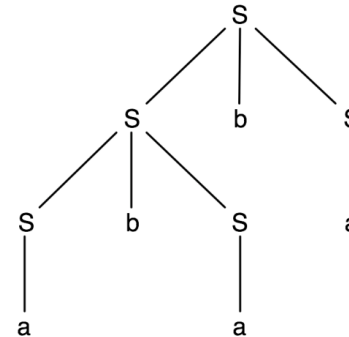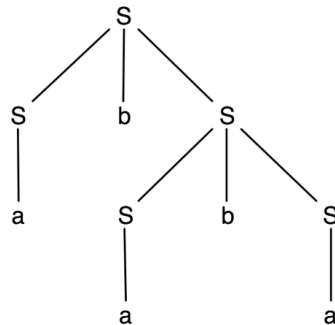
Ambiguity

- A grammar is ambiguous if a string may have multiple leftmost derivations
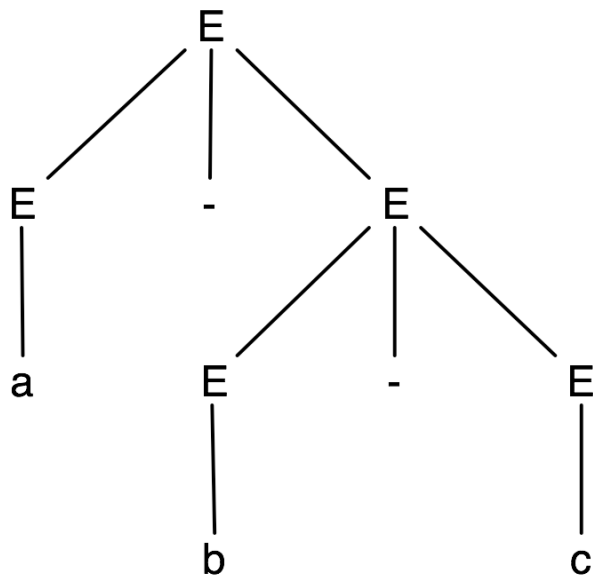
I saw a girl with a telescope.

CFGs for Programming Languages

▶ Recall that our goal is to describe programming languages with CFGs

▶ We had the following example which describes limited arithmetic expressions

   $E \rightarrow a \mid b \mid c \mid E+E \mid E-E \mid E*E \mid (E)$

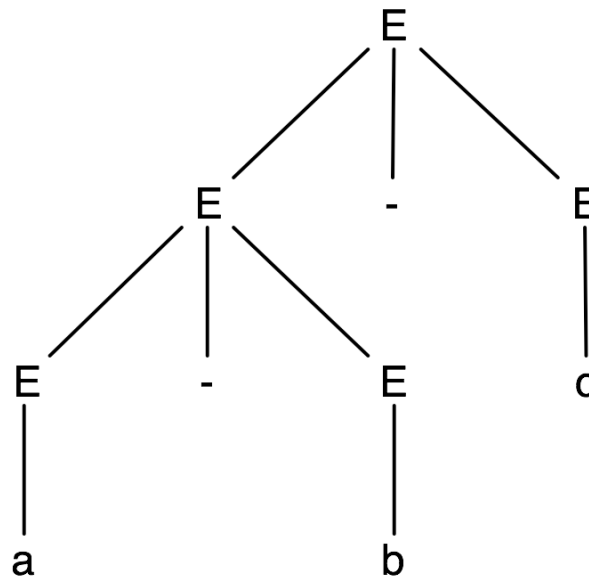▶ What's wrong with using this grammar?
   • It's ambiguous!

Example: a-b-c

$E \Rightarrow E\text{-}E \Rightarrow a\text{-}E \Rightarrow a\text{-}E\text{-}E \Rightarrow$
$a\text{-}b\text{-}E \Rightarrow a\text{-}b\text{-}c$

$E \Rightarrow E\text{-}E \Rightarrow E\text{-}E\text{-}E \Rightarrow$
$a\text{-}E\text{-}E \Rightarrow a\text{-}b\text{-}E \Rightarrow a\text{-}b\text{-}c$

Corresponds to a-(b-c)

Corresponds to (a-b)-c

Example: a-b*c

E ⇒ E-E ⇒ a-E ⇒ a-E*E ⇒
a-b*E ⇒ a-b*c

E ⇒ E-E ⇒ E-E*E ⇒
a-E*E ⇒ a-b*E ⇒ a-b*c
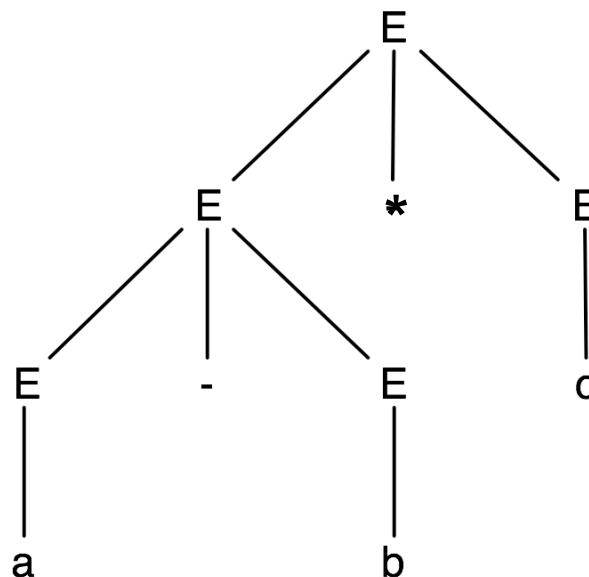
Corresponds to a-(b*c)

Corresponds to (a-b)*c

Another Example: If-Then-Else

Aka the dangling else problem

<stmt> → <assignment> | <if-stmt> | ...
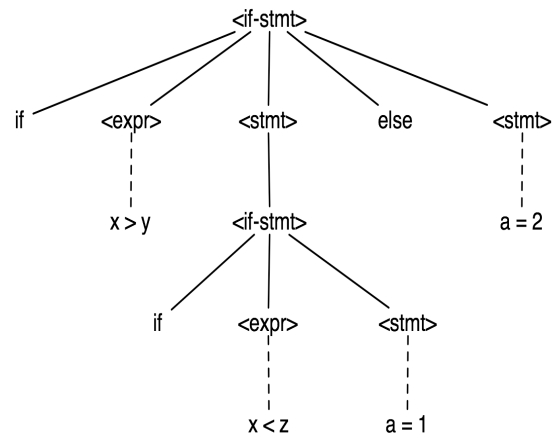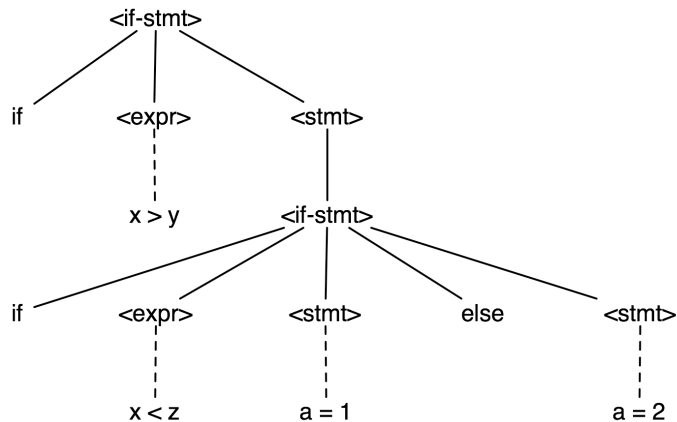
<if-stmt> → if (<expr>) <stmt> |

              if (<expr>) <stmt> else <stmt>

    (Recall < >'s are used to denote nonterminals)

- Consider the following program fragment

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

    (Note: Ignore newlines)

Two Parse Trees

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

Quiz #5

Which of the following grammars is ambiguous?

     A.   S → 0SS1 | 0S1 | ε

     B.   S → A1S1A | ε

          A → 0

     C.   S → (S, S, S) | 1

     D.   None of the above.

Quiz #5

Which of the following grammars is ambiguous?

A. S → 0SS1 | 0S1 | ε

B. S → A1S1A | ε

    A → 0

C. S → (S, S, S) | 1

D. None of the above.

S →0SS1 →0S1 →01
S →0S1 →01

Dealing With Ambiguous Grammars

- ▶ **Ambiguity is bad**
  - Syntax is correct, but semantics differ depending on choice
    - ➢ Different associativity   (a-b)-c vs. a-(b-c)
    - ➢ Different precedence   (a-b)*c vs. a-(b*c)
    - ➢ Different control flow   if (if else) vs. if (if) else

- ▶ **Two approaches**
  - Rewrite grammar
    - ➢ **Grammars are not unique** – can have multiple grammars for the same language. But result in different parses.
  - Use special parsing rules
    - ➢ Depending on parsing tool

Fixing the Expression Grammar

▸ Require right operand to not be bare expression

  E → E+T | E-T | E*T | T

  T → a | b | c | (E)

▸ Corresponds to left associativity

▸ Now only one parse tree for a-b-c
  • Find derivation
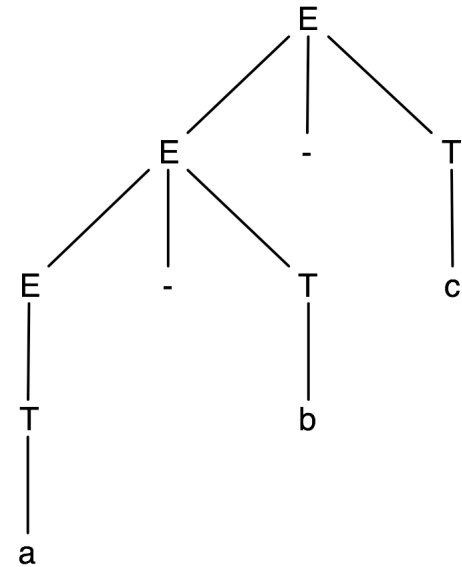
What if we want Right Associativity?

► Left-recursive productions
  - Used for left-associative operators
  - Example

    E → E+T | E-T | E*T | T

    T → a | b | c | (E)

► Right-recursive productions
  - Used for right-associative operators
  - Example

    E → T+E | T-E | T*E | T

    T → a | b | c | (E)

A Different Problem

▶ How about the string a+b*c ?

   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

▶ Solution – Introduce new nonterminals
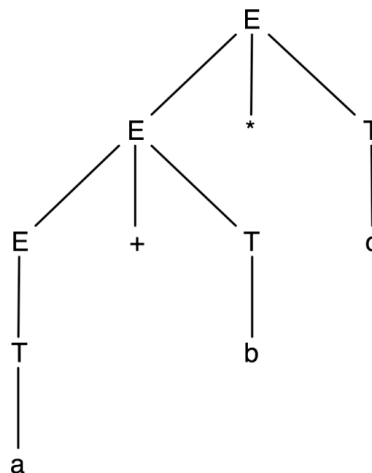
Final Expression Grammar

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

Derivation of a+b*c:

E → E+T→T+T→P+T→a+T→a+T*P→a+P*P→a+b*P→a+b*c

Fixing the Expression Grammar

- Controlling precedence of operators
  - Introduce new nonterminals
  - Precedence increases closer to operands

- Controlling associativity of operators
  - Introduce new nonterminals
  - Assign associativity based on production form
    - E → E+T (left associative) vs. E → T+E (right associative)
      - But parsing method might limit form of rules

Conclusion

▶ Context Free Grammars (CFGs) can describe programming language syntax

- They are a kind of formal language that is more powerful than regular expressions

▶ CFGs can also be used as the basis for programming language parsers (details later)

- But the grammar should not be ambiguous
  ➢ May need to change more natural grammar to make it so
- Parsing often aims to produce abstract syntax trees
  ➢ Data structure that records the key elements of program