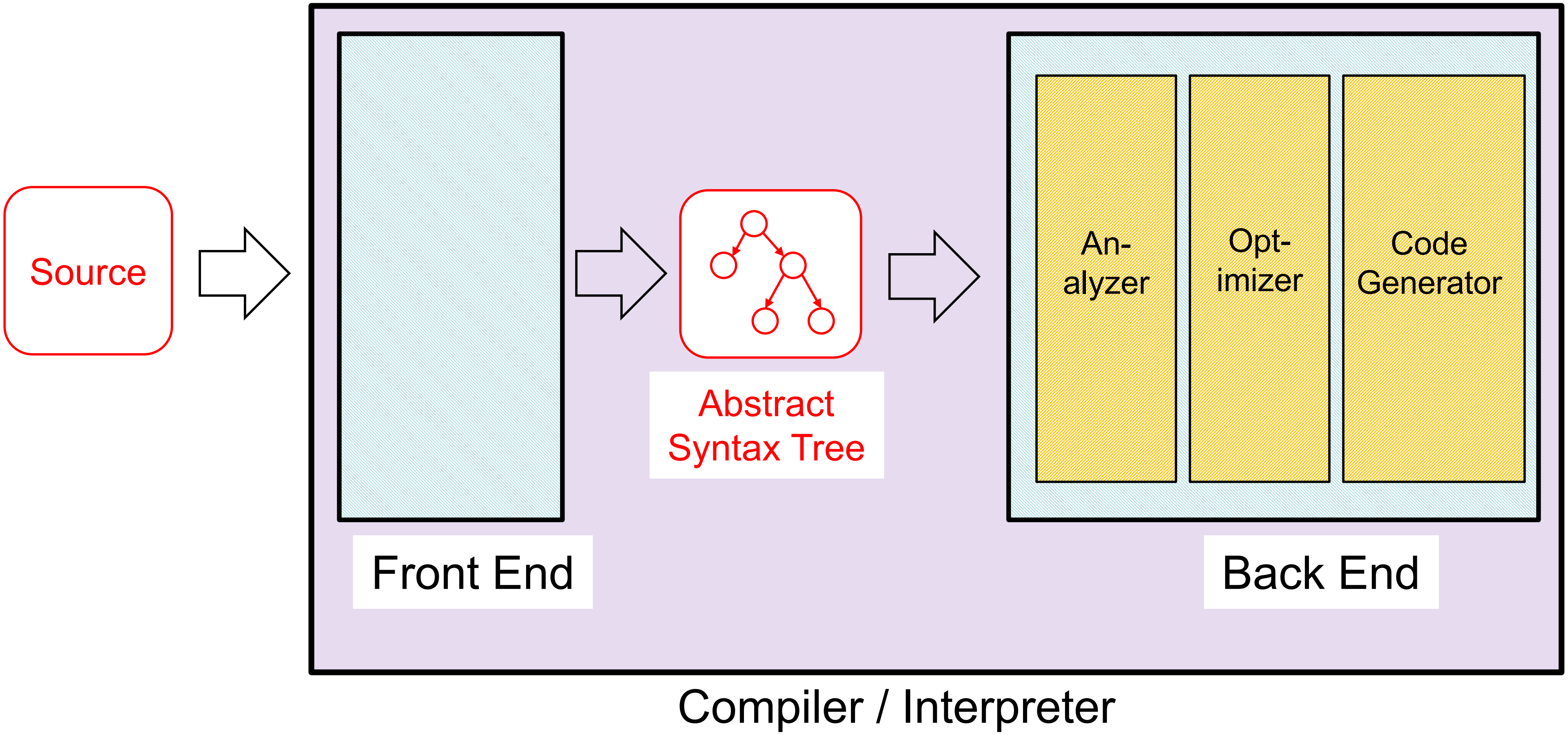
CMSC 330: Organization of Programming Languages

Context Free Grammars

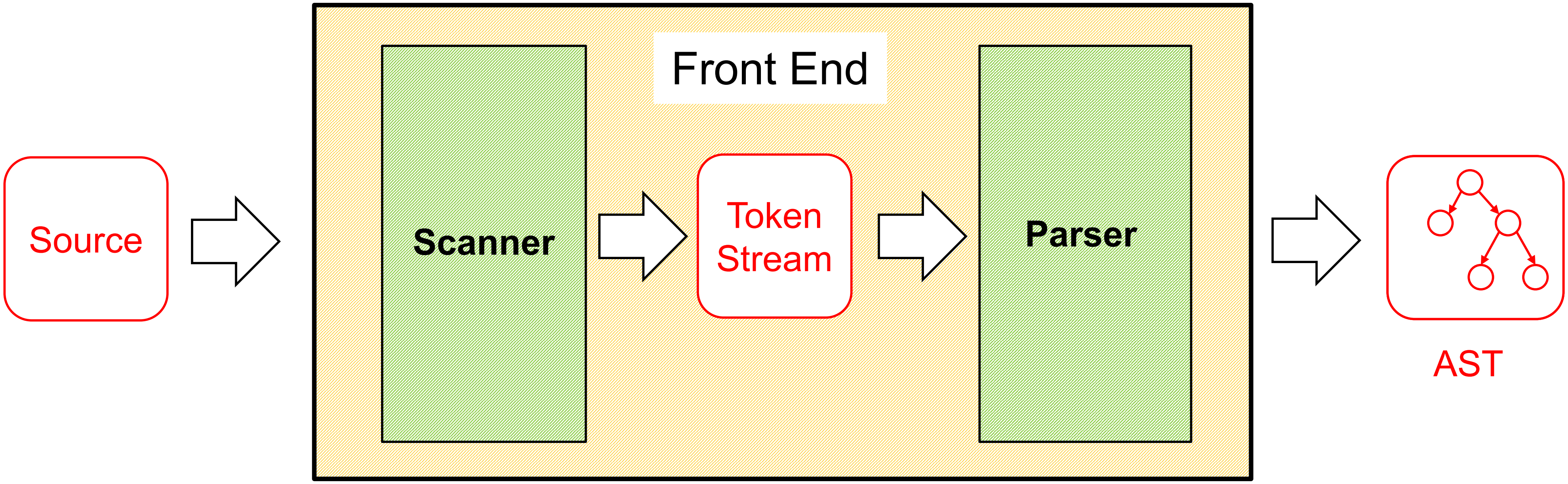
Architecture of Compilers, Interpreters



Implementing the Front End

- Goal: Convert program text into an AST
  - Abstract Syntax Tree
- ASTs are easier to work with
  - Analyze, optimize, execute the program
- Idea: Do this using regular expressions?
  - Won't work!
  - Regular expressions cannot reliably parse paired braces {{ … }}, parentheses (((…))), etc.
- Instead: Regexps for tokens (**scanning**), and Context Free Grammars for **parsing** tokens

Front End – Scanner and Parser



- Scanner / lexer converts program source into tokens (keywords, variable names, operators, numbers, etc.) using regular expressions
- Parser converts tokens into an AST (abstract syntax tree) using context free grammars

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

  which says that string $s' \in L(G)$ iff

  - $s' = \varepsilon$, or $\exists s \in L(G)$ such that $s' = 0s$, or $s' = 1s$
- Grammar is same as regular expression $(0|1)^*$
  - Generates / accepts the same set of strings

CFGs Are Expressive

- CFGs subsume REs, DFAs, NFAs
  - There is a CFG that generates any regular language
  - But: REs are often better notation for those languages
- 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 $N \cap \Sigma = \varnothing$
  - P – a set of productions of the form $N \rightarrow (\Sigma|N)^*$
    - Informally: the nonterminal can be replaced by the string of zero or more terminals / nonterminals to the right of the $\rightarrow$
    - Can think of productions as rewriting rules (more later)
  - $S \in N$ – the start symbol

Notational Shortcuts

$$
\begin{array}{lll}
S \rightarrow aBc & & \text{// S is start symbol} \\
A \rightarrow aA & & \\
\quad \mid \ b & & \text{// } A \rightarrow b \\
\quad \mid & & \text{// } A \rightarrow \varepsilon
\end{array}
$$

- 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

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>
  - Non-terminals written with angle brackets and uses ::= instead of →
  - Often see hybrids that use ::= instead of → but drop the angle brackets on non-terminals

Generating Strings

- We can think of a grammar as generating strings by rewriting
- Example grammar G

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

- Generate string 011 from G as follows:

  $S \Rightarrow 0S$ // using $S \rightarrow 0S$

  $\Rightarrow 01S$ // using $S \rightarrow 1S$

  $\Rightarrow 011S$ // using $S \rightarrow 1S$

  $\Rightarrow 011$ // using $S \rightarrow \varepsilon$

Accepting Strings (Informally)

- Checking if $s \in L(G)$ is called acceptance
  - Algorithm: Find a rewriting starting from G's start symbol that yields s
  - A rewriting is some sequence of productions (rewrites) applied starting at the start symbol
    - $011 \in L(G)$ 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

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

  - 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

Practice

- Given the grammar

  $S \rightarrow aS \mid T$

  $T \rightarrow bT \mid U$

  $U \rightarrow cU \mid \varepsilon$

  - Provide derivations for the following strings
    - b $\quad S \Rightarrow T \Rightarrow bT \Rightarrow bU \Rightarrow b$
    - ac $\quad S \Rightarrow aS \Rightarrow aT \Rightarrow aU \Rightarrow acU \Rightarrow ac$
    - bbc $\quad S \Rightarrow T \Rightarrow bT \Rightarrow bbT \Rightarrow bbU \Rightarrow bbcU \Rightarrow bbc$
  - Does the grammar generate the following?
    - $S \Rightarrow^+ ccc$ Yes $\quad S \Rightarrow^+ bS$ No
    - $S \Rightarrow^+ bab$ No $\quad S \Rightarrow^+ Ta$ No

Practice

- Given the grammar

  $S \rightarrow aS \mid T$

  $T \rightarrow bT \mid U$

  $U \rightarrow cU \mid \varepsilon$

  - Name language accepted by grammar
    - a*b*c*
  - Give a different grammar accepting language

  $S \rightarrow ABC$
  $A \rightarrow aA \mid \varepsilon$ // a*
  $B \rightarrow bB \mid \varepsilon$ // b*
  $C \rightarrow cC \mid \varepsilon$ // c*

Designing Grammars

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

   $A \rightarrow xA \mid \varepsilon$ // Zero or more x's

   $A \rightarrow yA \mid y$ // 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

   $S \rightarrow AB$

   $A \rightarrow aA \mid \varepsilon$ // Zero or more a's

   $B \rightarrow bB \mid \varepsilon$ // 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 $n \geq 0$ – 0^n1^m where $m \leq n$

$S \rightarrow A \mid B$
$A \rightarrow 0A \mid \varepsilon$
$B \rightarrow 1B \mid \varepsilon$

$S \rightarrow 0S1 \mid \varepsilon$

$S \rightarrow 0S1 \mid 0S \mid \varepsilon$

- Give some example strings from this language
  - $S \rightarrow 0 \mid 1S$
    - 0, 10, 110, 1110, 11110, …
  - What language is it, as a regexp?
    - 1*0

CFGs for Language Syntax

- When discussing operational semantics, we used BNF-style grammars to define ASTs

$$e ::= x \mid n \mid e + e \mid \texttt{let } x = e \texttt{ in } e$$

  - This grammar defined an AST for expressions synonymous with an OCaml datatype
- We can *also* use this grammar to define a language parser
  - However, while it is fine for defining ASTs, this grammar, if used directly for parsing, is ambiguous

Arithmetic Expressions

- $E \rightarrow a \mid b \mid c \mid E+E \mid E-E \mid E*E \mid (E)$
  - An expression E is either a letter a, b, or c
  - Or an E followed by + followed by an E
  - etc…
- 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.

Formal Description of Example

- Formally, the grammar we just showed is
  - Σ = { +, -, *, (,), a, b, c } // terminals
  - N = { E } // nonterminals
  - P = { E $\rightarrow$ a, E $\rightarrow$ b, E $\rightarrow$ c, // productions
    E $\rightarrow$ E-E, E $\rightarrow$ E+E,
    E $\rightarrow$ E*E,
    E $\rightarrow$ (E)
    }
  - S = E // start symbol

(Non-)Uniqueness of Grammars

- Different grammars generate the same set of strings (language)

- The following grammar generates the same set of strings as the previous grammar

  $E \rightarrow E+T \mid E-T \mid T$

  $T \rightarrow T*P \mid P$

  $P \rightarrow (E) \mid a \mid b \mid c$

Parse Trees

- Parse tree shows how a string is produced by a grammar
  - Root node is the start symbol
  - Every internal node is a nonterminal
  - Children of an internal node
    - Are symbols on RHS of production applied to nonterminal
  - Every leaf node is a terminal or ε

- Reading the leaves left to right
  - Shows the string corresponding to the tree

Parse Tree Example

S

S

$S \rightarrow aS \mid T$

$T \rightarrow bT \mid U$

$U \rightarrow cU \mid \varepsilon$

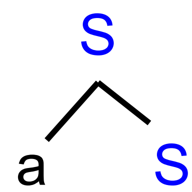
Parse Tree Example

$$S \Rightarrow aS$$

$S \rightarrow aS \mid T$

$T \rightarrow bT \mid U$

$U \rightarrow cU \mid \varepsilon$



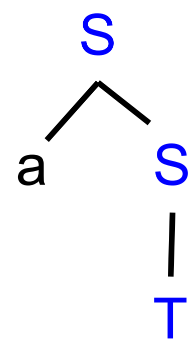
Parse Tree Example

$$S \Rightarrow aS \Rightarrow aT$$

$S \rightarrow aS \mid T$

$T \rightarrow bT \mid U$

$U \rightarrow cU \mid \varepsilon$



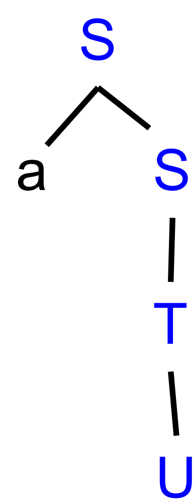
Parse Tree Example

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

$S \rightarrow aS \mid T$

$T \rightarrow bT \mid U$

$U \rightarrow cU \mid \varepsilon$



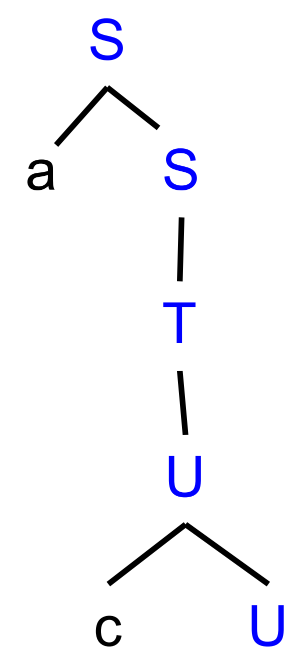
Parse Tree Example

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

$S \rightarrow aS \mid T$

$T \rightarrow bT \mid U$

$U \rightarrow cU \mid \varepsilon$



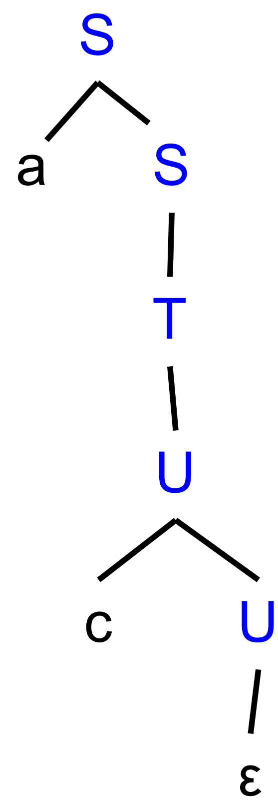
Parse Tree Example

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

$S \rightarrow aS \mid T$

$T \rightarrow bT \mid U$

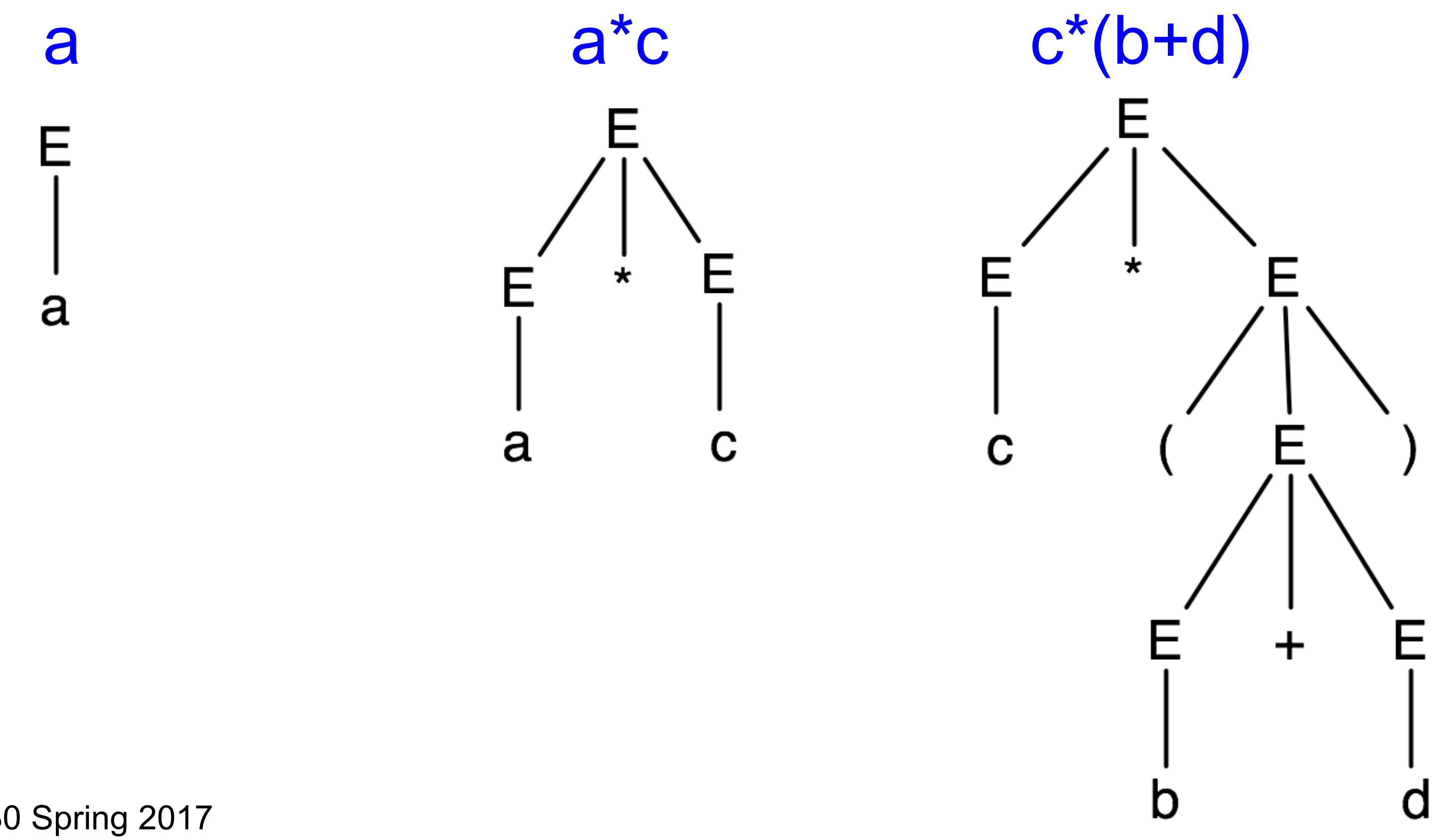
$U \rightarrow cU \mid \varepsilon$



Parse Trees for Expressions

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

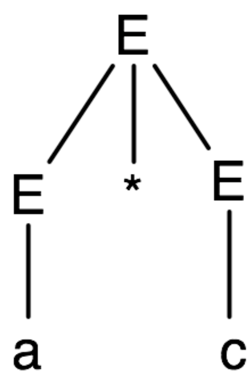
$E \to a \mid b \mid c \mid d \mid E+E \mid E-E \mid E*E \mid (E)$



Abstract Syntax Trees

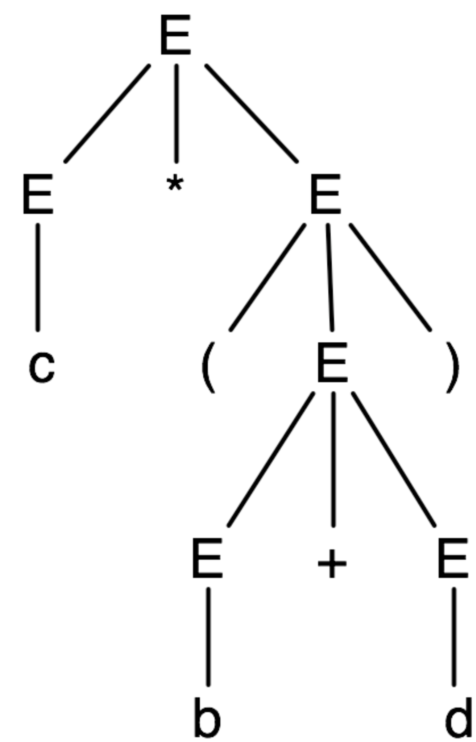
- A parse tree and an AST are not the same thing
  - The latter is a data structure produced by parsing

a*c

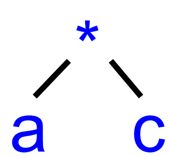


c*(b+d)

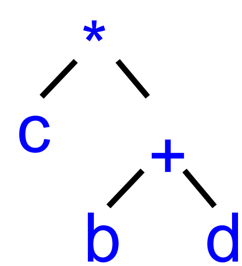
Parse trees



ASTs



`Mult(Var("a"),Var("c"))`



`Mult(Var("c"),Plus(Var("b"),Var("d")))`

Practice

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

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
- Example
  - 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

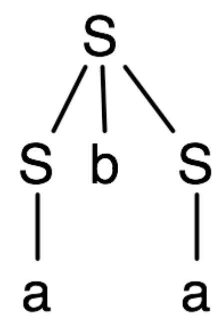
- Parse tree may be same for both leftmost & rightmost derivations
  - Example Grammar: S → a | SbS   String: aba

    Leftmost Derivation

    S ⇒ SbS ⇒ abS ⇒ aba

    Rightmost Derivation

    S ⇒ SbS ⇒ Sba ⇒ aba



  - 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.)

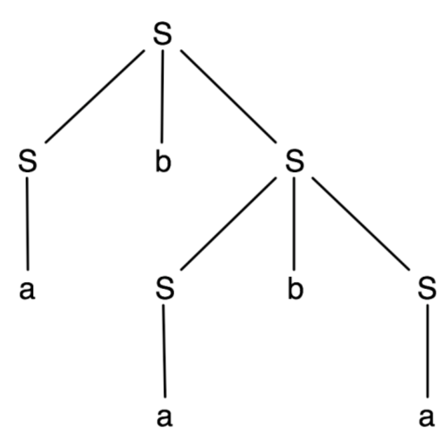
- Not every string has a unique parse tree
  - Example Grammar: $S \rightarrow a \mid SbS$ String: ababa

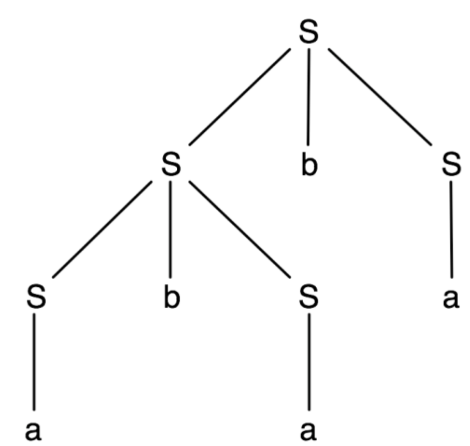
    Leftmost Derivation

    $S \Rightarrow SbS \Rightarrow abS \Rightarrow abSbS \Rightarrow ababS \Rightarrow ababa$

    Another Leftmost Derivation

    $S \Rightarrow SbS \Rightarrow SbSbS \Rightarrow abSbS \Rightarrow ababS \Rightarrow ababa$



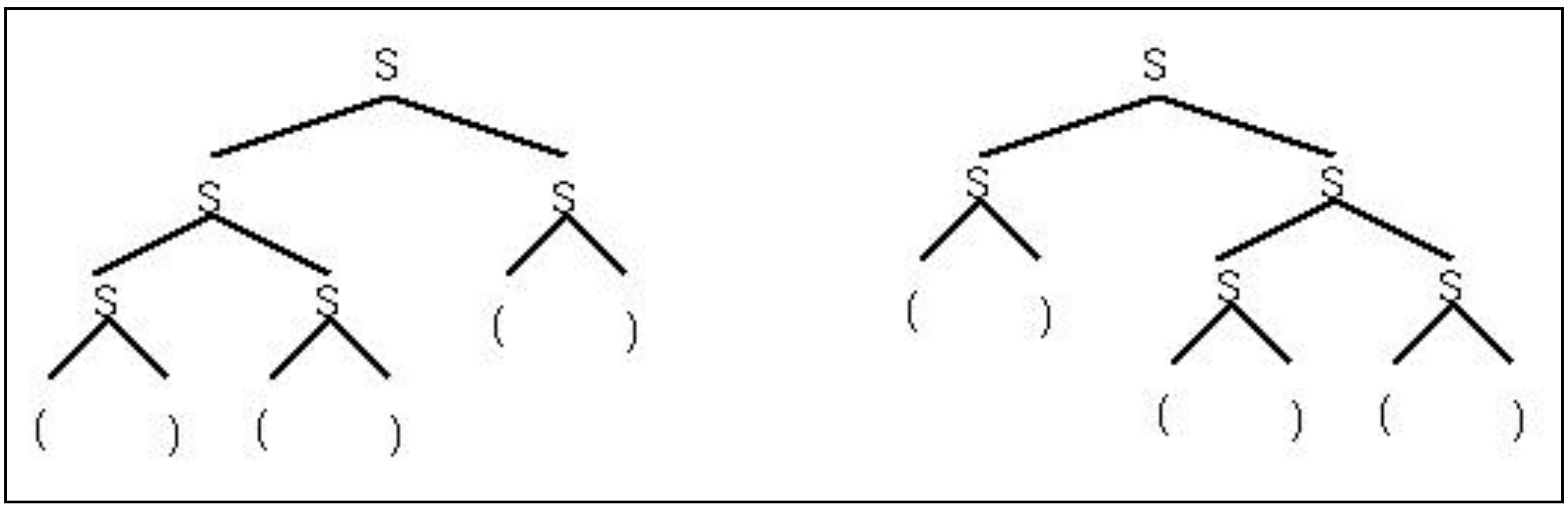


Ambiguity

- A grammar is ambiguous if a string may have multiple leftmost derivations
  - Equivalent to multiple parse trees
  - Can be hard to determine
    1. $S \rightarrow aS \mid T$
       $T \rightarrow bT \mid U$ **No**
       $U \rightarrow cU \mid \varepsilon$
    2. $S \rightarrow T \mid T$
       $T \rightarrow Tx \mid Tx \mid x \mid x$ **Yes**
    3. $S \rightarrow SS \mid () \mid (S)$ **?**

Ambiguity (cont.)

- Example
  - Grammar: S → SS | () | (S) String: ()()()
  - 2 distinct (leftmost) derivations (and parse trees)
    - $S \Rightarrow \underline{S}S \Rightarrow \underline{S}SS \Rightarrow ()\underline{S}S \Rightarrow ()()\underline{S} \Rightarrow ()()()$
    - $S \Rightarrow \underline{S}S \Rightarrow ()\underline{S} \Rightarrow ()\underline{S}S \Rightarrow ()()\underline{S} \Rightarrow ()()()$



CFGs for Programming Languages

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

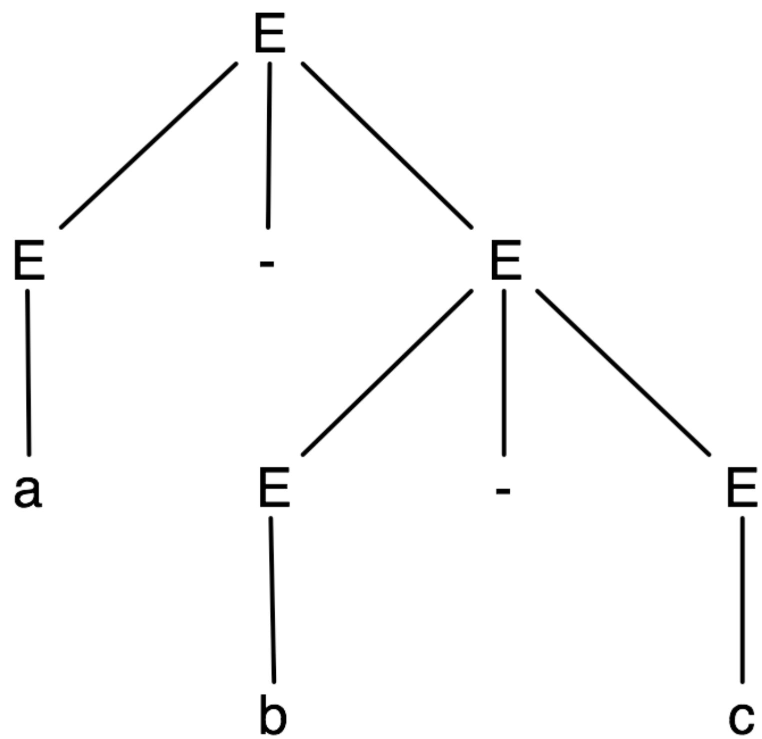
- We had the following example which describes limited arithmetic expressions

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

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

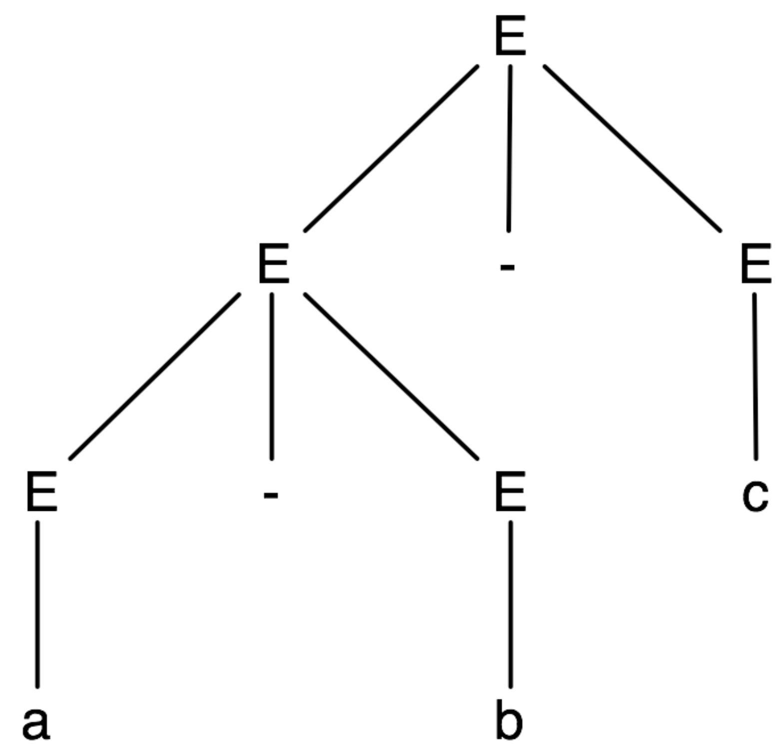
Example: a-b-c

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



Corresponds to a-(b-c)

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

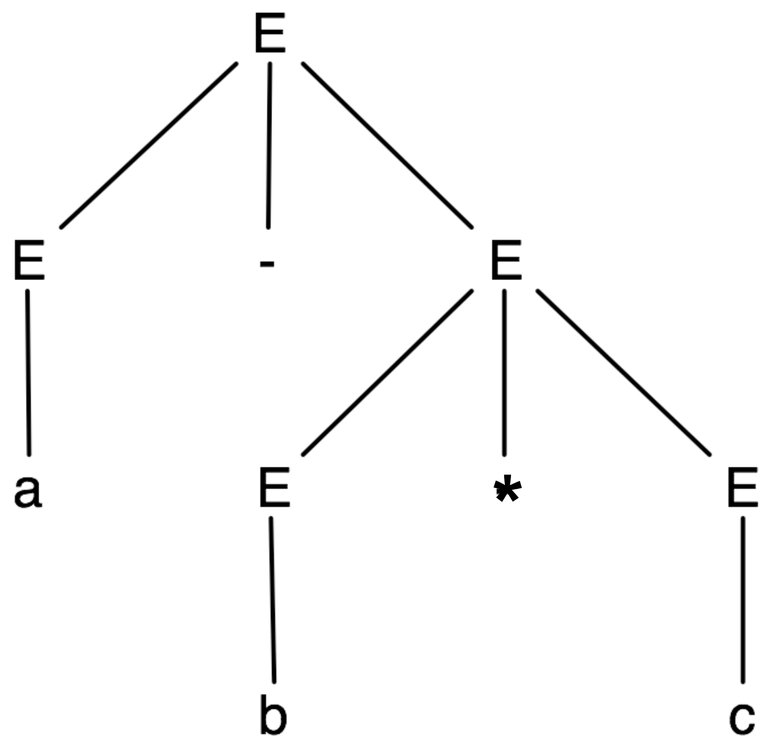


Corresponds to (a-b)-c

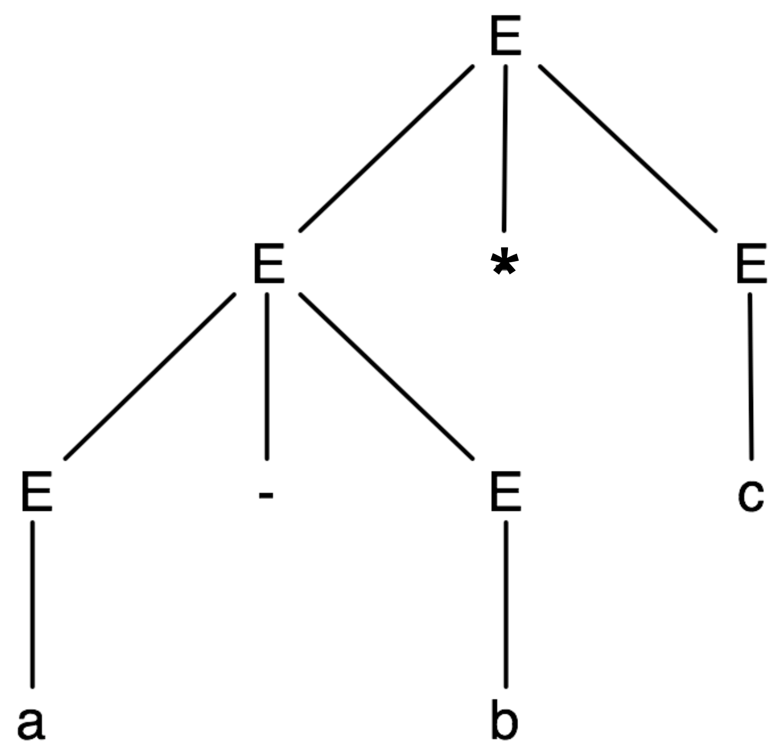
Example: a-b*c

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

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



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>

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

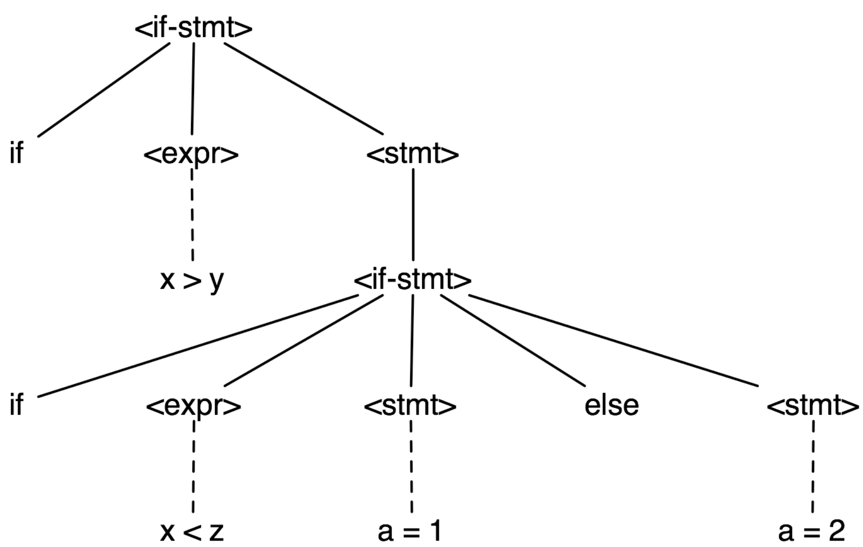
- Consider the following program fragment

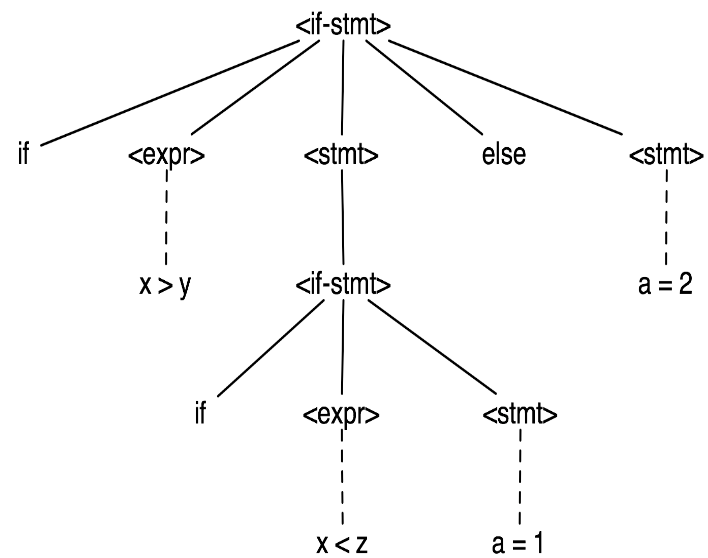
```
if (x > y)
  if (x < z)
    a = 1;
  else a = 2;
```

(Note: Ignore newlines)

Two Parse Trees

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





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
  - Use special parsing rules
    - Depending on parsing method (learn in CMSC 430)

Fixing the Expression Grammar

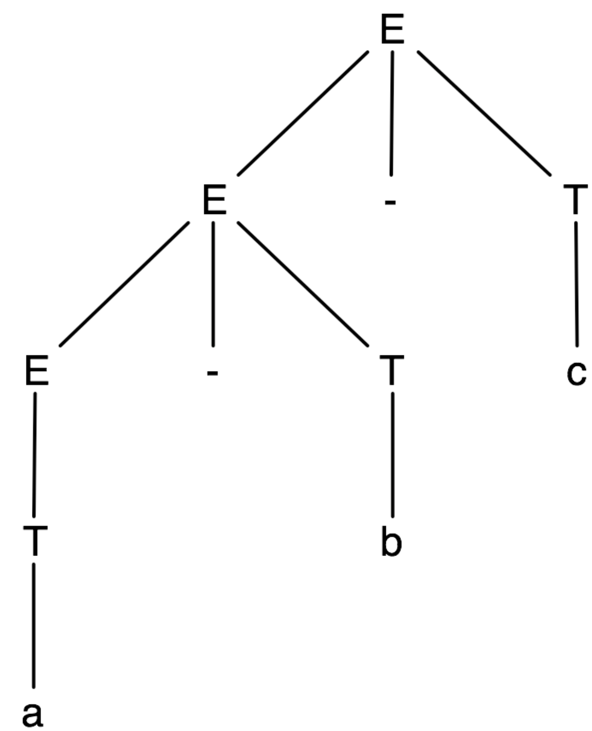
- Require right operand to not be bare expression

  $E \rightarrow E+T \mid E\text{-}T \mid E*T \mid T$

  $T \rightarrow a \mid b \mid c \mid (E)$

- 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

    $E \rightarrow E+T \mid E-T \mid E*T \mid T$

    $T \rightarrow a \mid b \mid c \mid (E)$
- Right-recursive productions
  - Used for right-associative operators
  - Example

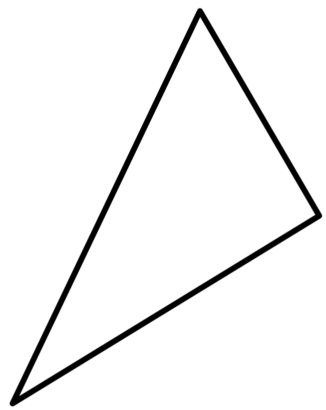
    $E \rightarrow T+E \mid T-E \mid T*E \mid T$

    $T \rightarrow a \mid b \mid c \mid (E)$

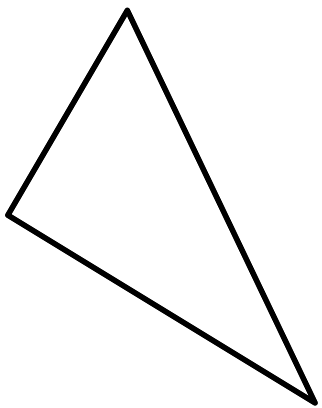
Parse Tree Shape

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

left recursion



right recursion

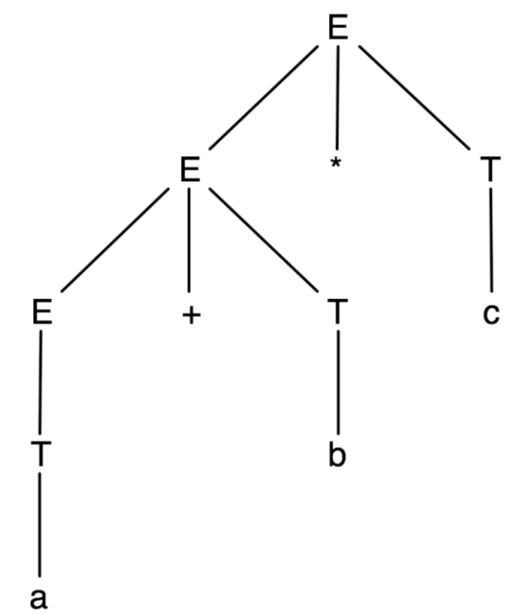


A Different Problem

- How about the string a+b*c ?

  $E \rightarrow E+T \mid E-T \mid E*T \mid T$

  $T \rightarrow a \mid b \mid c \mid (E)$



- 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

$E \rightarrow E+T \mid E-T \mid T$	lowest precedence operators
$T \rightarrow T*P \mid P$	higher precedence
$P \rightarrow a \mid b \mid c \mid (E)$	highest precedence (parentheses)

- 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
    - $E \rightarrow E+T$ (left associative) vs. $E \rightarrow T+E$ (right associative)

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