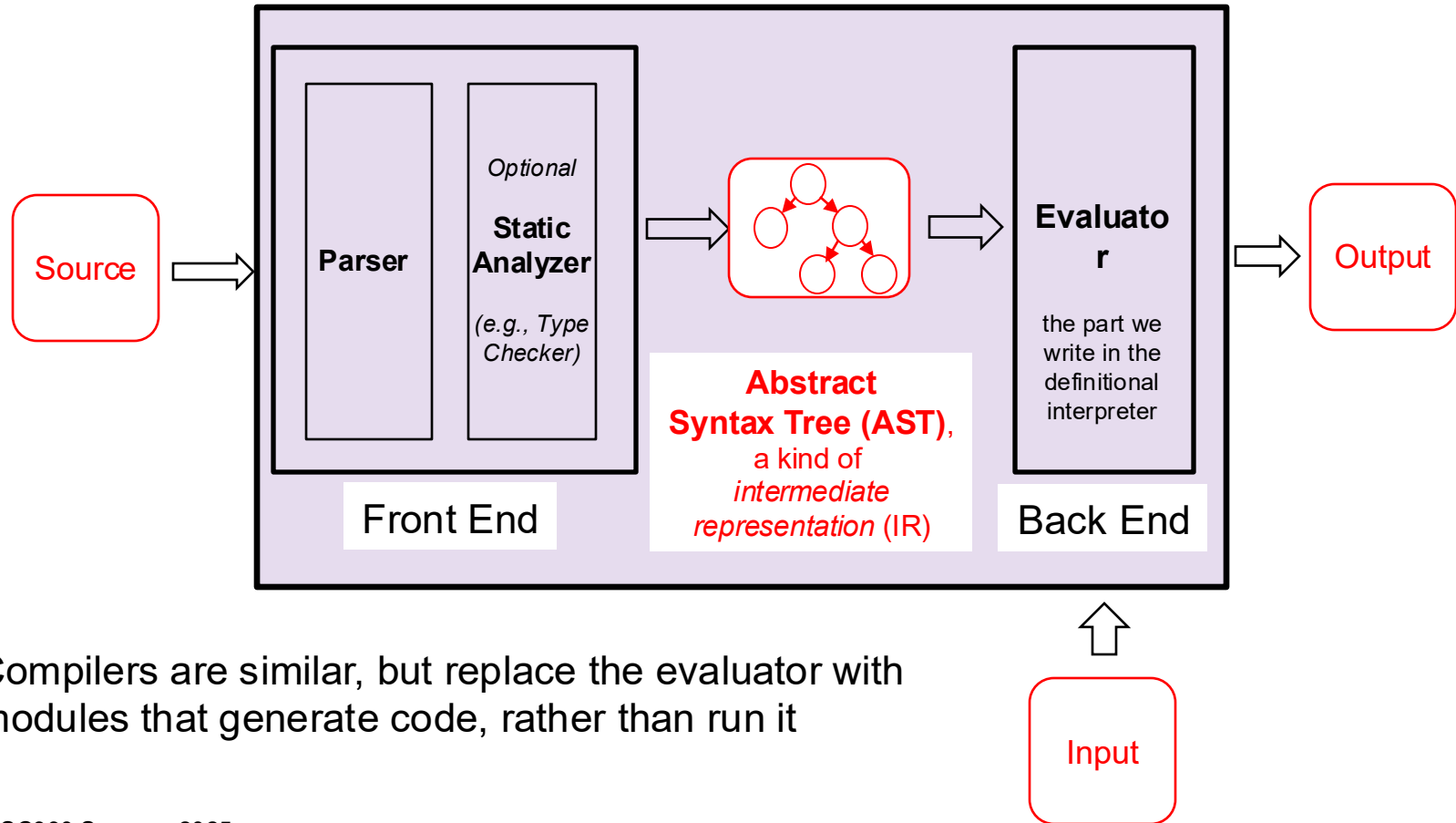


# CMSC 330: Organization of Programming Languages

---

## Context Free Grammars

# Interpreters



Compilers are similar, but replace the evaluator with modules that generate code, rather than run it

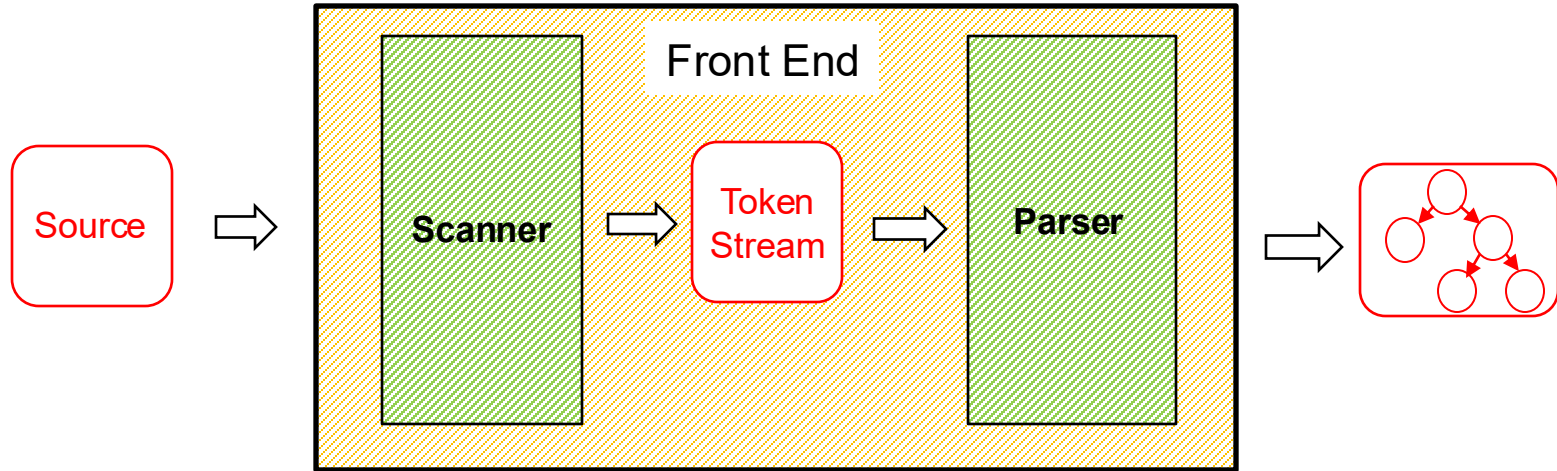
# Implementing the Front End

---

- ▶ Goal: Convert program text into an **Abstract Syntax Tree**
- ▶ ASTs are easier to work with
  - Analyze, optimize, execute the program
- ▶ Do this using regular expressions?
  - **Won't work!**
  - Regular expressions cannot reliably parse paired braces `{ ... }`, parentheses `(( ... ))`, etc.
- ▶ Instead: Regexp 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). Parsers recognize strings defined as **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
  - $S \rightarrow ( S ) \mid \epsilon$       // 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  $(\Sigma, N, P, S)$ 
  - $\Sigma$  – 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  $N \cap \Sigma = \emptyset$
  - $P$  – 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



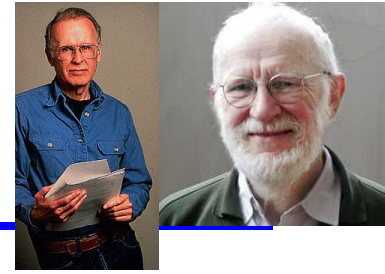
# Notational Shortcuts

$S \rightarrow aBc$	$S \rightarrow aBc$	// S is start symbol
$A \rightarrow aA$		
b		// $A \rightarrow b$
		// $A \rightarrow \varepsilon$

- ▶ A production is of the form
  - left-hand side (LHS)  $\rightarrow$  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  $\varepsilon$

# 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 is written in BNF as

$$A \rightarrow B \ c \ D$$
$$\langle A \rangle ::= \langle B \rangle \ c \ \langle D \rangle$$

# 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**

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

- ▶ 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** from  $G$ 's start symbol that yields  $s$ 
    - $011 \in L(G)$  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
  - $\Sigma$  is the alphabet for that grammar
- ▶ In other words
    - All strings over  $\Sigma$  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. ccc
- C. bab
- 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. ccc

C. bab

D. ca



## Quiz #2

---

- ▶ Consider the grammar

$$S \rightarrow bS \mid T$$

$$T \rightarrow aT \mid U$$

$$U \rightarrow cU \mid \varepsilon$$

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

A.  $S \Rightarrow T \Rightarrow aT \Rightarrow aTaT \Rightarrow aaT \Rightarrow aacU \Rightarrow aac$

B.  $S \Rightarrow T \Rightarrow U \Rightarrow aU \Rightarrow aaU \Rightarrow aacU \Rightarrow aac$

C.  $S \Rightarrow aT \Rightarrow aaT \Rightarrow aaU \Rightarrow aacU \Rightarrow aac$

D.  $S \Rightarrow T \Rightarrow aT \Rightarrow aaT \Rightarrow aaU \Rightarrow aacU \Rightarrow aac$

## Quiz #2

---

- ▶ Consider the grammar

$$S \rightarrow bS \mid T$$

$$T \rightarrow aT \mid U$$

$$U \rightarrow cU \mid \varepsilon$$

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

A.  $S \Rightarrow T \Rightarrow aT \Rightarrow aTaT \Rightarrow aaT \Rightarrow aacU \Rightarrow aac$

B.  $S \Rightarrow T \Rightarrow U \Rightarrow aU \Rightarrow aaU \Rightarrow aacU \Rightarrow aac$

C.  $S \Rightarrow aT \Rightarrow aaT \Rightarrow aaU \Rightarrow aacU \Rightarrow aac$

D.  $S \Rightarrow T \Rightarrow aT \Rightarrow aaT \Rightarrow aaU \Rightarrow aacU \Rightarrow aac$

## Quiz #3

---

Consider the grammar

$$S \rightarrow bS \mid T$$

$$T \rightarrow aT \mid U$$

$$U \rightarrow cU \mid \varepsilon$$

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

A.  $(a|b|c)^*$

B.  $b^*a^*c^*$

C.  $(b|ba|bac)^*$

D.  $bac^*$

## Quiz #3

---

Consider the grammar

$$S \rightarrow bS \mid T$$

$$T \rightarrow aT \mid U$$

$$U \rightarrow cU \mid \varepsilon$$

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

A.  $(a|b|c)^*$

B.  $b^*a^*c^*$

C.  $(b|ba|bac)^*$

D.  $bac^*$

# Designing Grammars

---

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

$A \rightarrow xA \mid \epsilon$  // 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

$S \rightarrow AB$

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

$B \rightarrow bB \mid \epsilon$  // 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

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

$$S \rightarrow T \mid V$$

$$T \rightarrow aTb \mid U$$

$$U \rightarrow Ub \mid b$$

$$V \rightarrow aVc \mid W$$

$$W \rightarrow Wc \mid c$$

# Practice

---

- ▶ Try to make a grammar which accepts

- $0^*|1^*$
- $0^n1^n$  where  $n \geq 0$

$$S \rightarrow A \mid B$$

$$A \rightarrow 0A \mid \varepsilon$$

$$B \rightarrow 1B \mid \varepsilon$$

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

- ▶ Give some example strings from this language

- $S \rightarrow 0 \mid 1S$

- 0, 10, 110, 1110, 11110, ...

- What language is it, as a regexp?

- $1^*0$



## Quiz #4

---

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

- A.  $S \rightarrow 0S1 \mid \epsilon$
- B.  $S \rightarrow 0S1 \mid S1 \mid \epsilon$
- C.  $S \rightarrow 0S1 \mid 0S \mid \epsilon$
- D.  $S \rightarrow SS \mid 0 \mid 1 \mid \epsilon$

## 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
  - *Will be useful for spotting ambiguity; discussed later*
  - Root node of parse tree 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  $\epsilon$
- ▶ 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

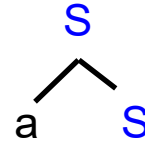
---

$$S \Rightarrow aS$$

$$S \rightarrow aS \mid T$$

$$T \rightarrow bT \mid U$$

$$U \rightarrow cU \mid \varepsilon$$



# Parse Tree Example

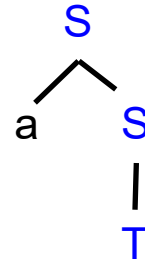
---

$$S \Rightarrow aS \Rightarrow aT$$

$$S \rightarrow aS \mid T$$

$$T \rightarrow bT \mid U$$

$$U \rightarrow cU \mid \varepsilon$$



# Parse Tree Example

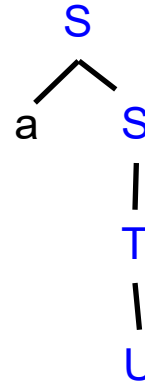
---

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

$S \rightarrow aS \mid T$

$T \rightarrow bT \mid U$

$U \rightarrow cU \mid \varepsilon$



# Parse Tree Example

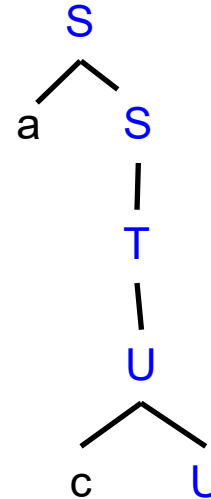
---

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

$S \rightarrow aS \mid T$

$T \rightarrow bT \mid U$

$U \rightarrow cU \mid \varepsilon$





# Parse Tree Example

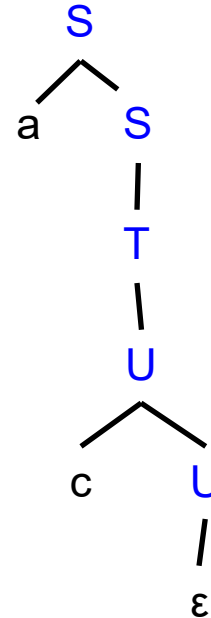
---

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

$S \rightarrow aS \mid T$

$T \rightarrow bT \mid U$

$U \rightarrow cU \mid \varepsilon$



# 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:

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

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):

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

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

a-(b\*a)   parses to

c\*(b+d)   parses to

Minus (A, C)

Minus (A, Mult (B,A))

Mult (C, Plus (B,D))

# Parse Trees for Expressions

---

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

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

a

a\*c

c\*(b+d)

# Parse Trees for Expressions

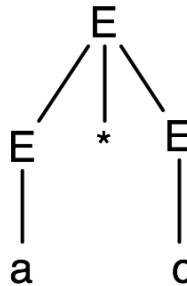
- ▶ A **parse tree** shows the structure of an expression as it corresponds to a grammar

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

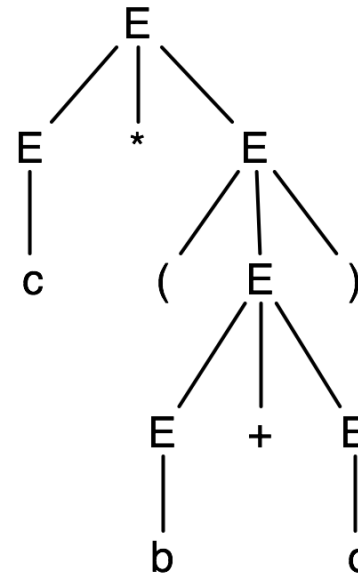
a



a\*c



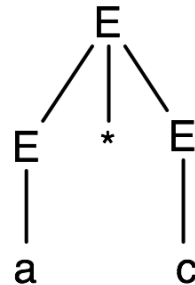
c\*(b+d)



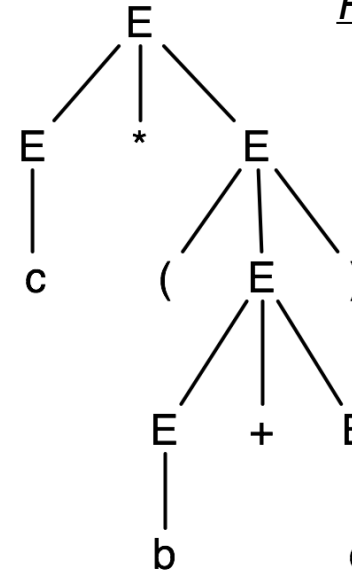
# Abstract Syntax Trees

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

$a * c$

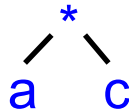


$c * (b + d)$

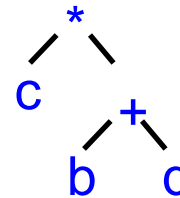


*Parse trees*

ASTs



`Mult(A, C)`



`Mult(C, Plus(B, D))`

# Practice

---

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

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

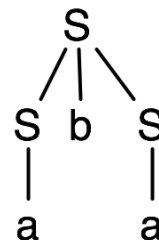
- Example Grammar:  $S \rightarrow a \mid SbS$      String:  $aba$

Leftmost Derivation

$S \Rightarrow SbS \Rightarrow abS \Rightarrow aba$

Rightmost Derivation

$S \Rightarrow SbS \Rightarrow Sba \Rightarrow aba$



- 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

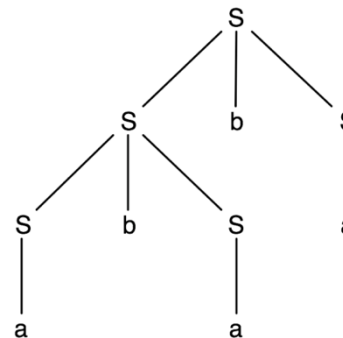
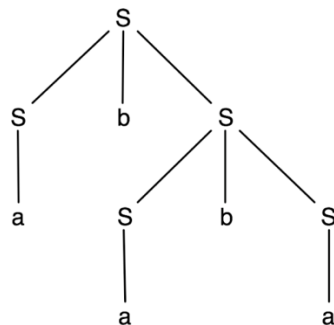
- Example Grammar:  $S \rightarrow a \mid SbS$     String: **ababa**

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

I saw a girl with a telescope.



# Ambiguity

---

- ▶ A grammar is **ambiguous** if a string may have multiple **leftmost** derivations

- Equivalent to multiple parse trees
- Can be hard to determine

1.  $S \rightarrow aS \mid T$

$$T \rightarrow bT \mid U$$

$$U \rightarrow cU \mid \varepsilon$$

**No**

2.  $S \rightarrow T \mid T$

$$T \rightarrow Tx \mid Tx \mid x \mid x$$

**Yes**

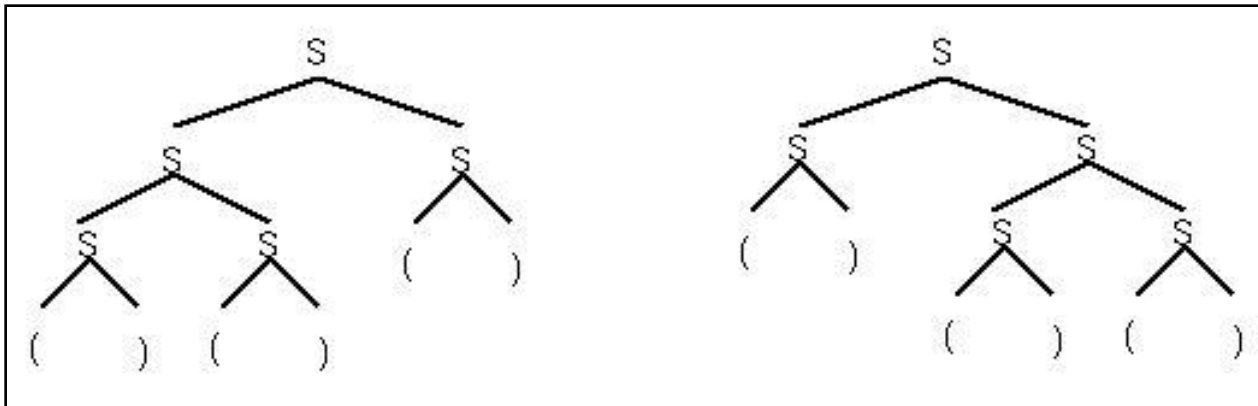
3.  $S \rightarrow SS \mid () \mid (S)$

**?**

# Ambiguity (cont.)

## ► Example

- Grammar:  $S \rightarrow SS \mid () \mid (S)$       String:  $()()()$
- 2 distinct (leftmost) derivations (and parse trees)
  - $S \Rightarrow \underline{SS} \Rightarrow \underline{SSS} \Rightarrow ()\underline{SS} \Rightarrow ()()\underline{S} \Rightarrow ()()()$
  - $S \Rightarrow \underline{SS} \Rightarrow ()\underline{S} \Rightarrow ()\underline{SS} \Rightarrow ()()\underline{S} \Rightarrow ()()()$



# CFGs for Programming Languages

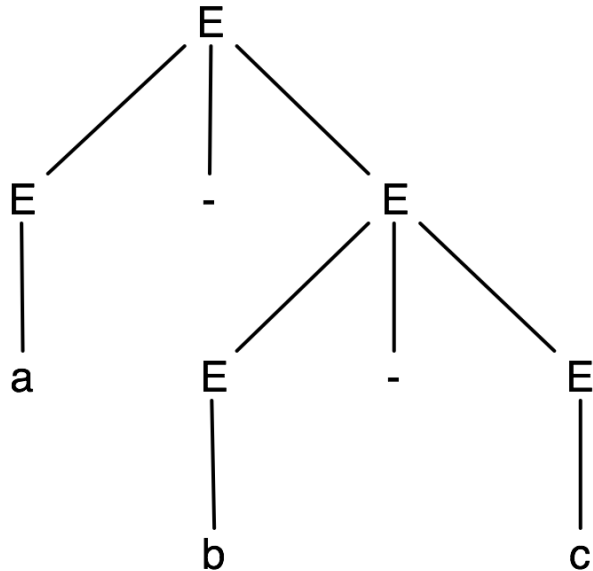
---

- ▶ 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

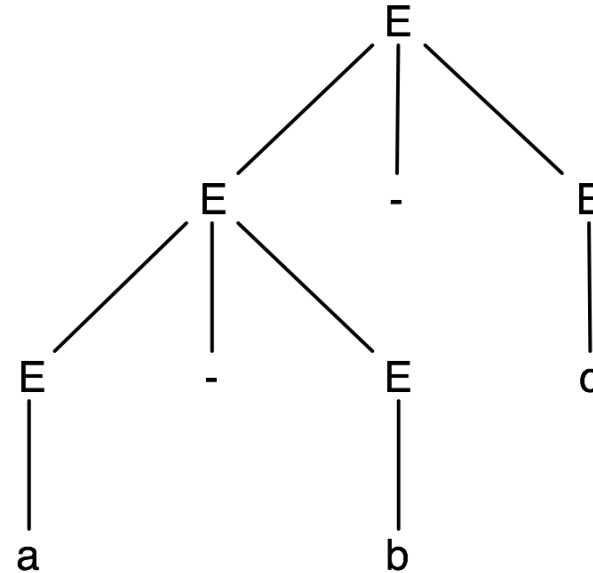
---

$E \Rightarrow E-E \Rightarrow a-E \Rightarrow a-E-E \Rightarrow$   
 $a-b-E \Rightarrow a-b-c$



Corresponds to  $a-(b-c)$

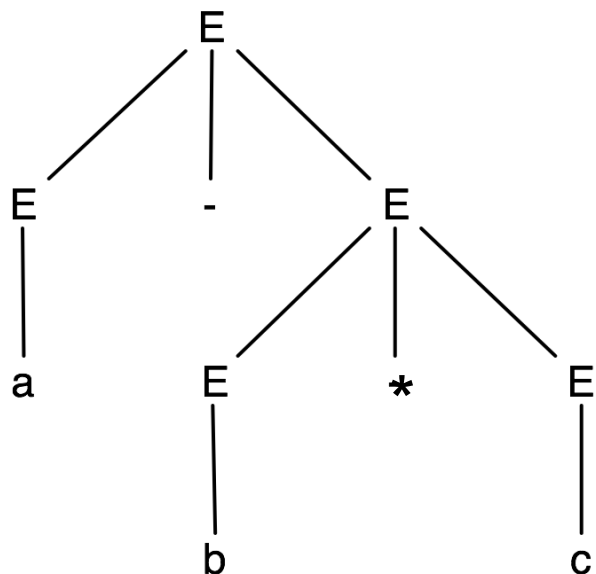
$E \Rightarrow E-E \Rightarrow E-E-E \Rightarrow$   
 $a-E-E \Rightarrow a-b-E \Rightarrow a-b-c$



Corresponds to  $(a-b)-c$

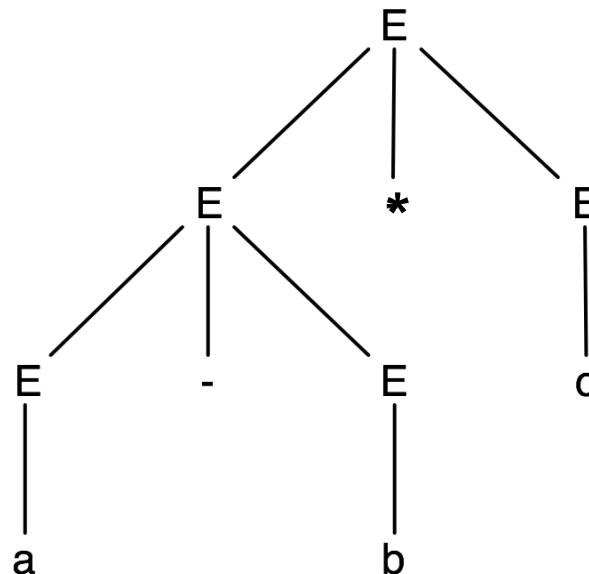
# Example: $a-b^*c$

$E \Rightarrow E-E \Rightarrow a-E \Rightarrow a-E^*E \Rightarrow$   
 $a-b^*E \Rightarrow a-b^*c$



Corresponds to  $a-(b^*c)$

$E \Rightarrow E-E \Rightarrow E-E^*E \Rightarrow$   
 $a-E^*E \Rightarrow a-b^*E \Rightarrow a-b^*c$



Corresponds to  $(a-b)^*c$



# Another Example: If-Then-Else

---

Aka the dangling else problem

$\langle \text{stmt} \rangle \rightarrow \langle \text{assignment} \rangle \mid \langle \text{if-stmt} \rangle \mid \dots$

$\langle \text{if-stmt} \rangle \rightarrow \text{if } (\langle \text{expr} \rangle) \langle \text{stmt} \rangle \mid$

$\text{if } (\langle \text{expr} \rangle) \langle \text{stmt} \rangle \text{ else } \langle \text{stmt} \rangle$

(Recall  $\langle \rangle$ 's are used to denote nonterminals)

- Consider the following program fragment

if ( $x > y$ )

if ( $x < z$ )

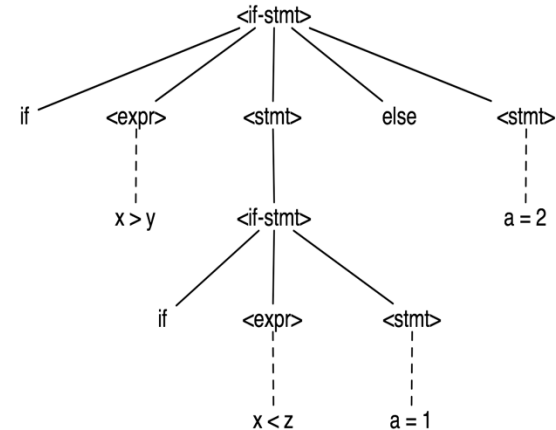
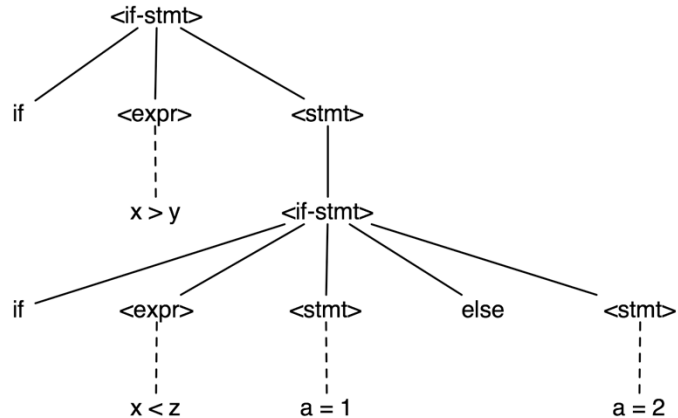
a = 1;

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.  $S \rightarrow 0SS1 \mid 0S1 \mid \varepsilon$

B.  $S \rightarrow A1S1A \mid \varepsilon$

$$A \rightarrow 0$$

C.  $S \rightarrow (S, S, S) \mid 1$

D. None of the above.

## Quiz #5

---

Which of the following grammars is ambiguous?

A.  $S \rightarrow 0SS1 \mid 0S1 \mid \epsilon$

B.  $S \rightarrow A1S1A \mid \epsilon$

$$A \rightarrow 0$$

C.  $S \rightarrow (S, S, S) \mid 1$

D. None of the above.

$$\begin{aligned} S &\rightarrow 0SS1 \rightarrow 0S1 \rightarrow 01 \\ S &\rightarrow 0S1 \rightarrow 01 \end{aligned}$$

# 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  $\text{if (if else)}$  vs.  $\text{if (if) else}$

- ▶ 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

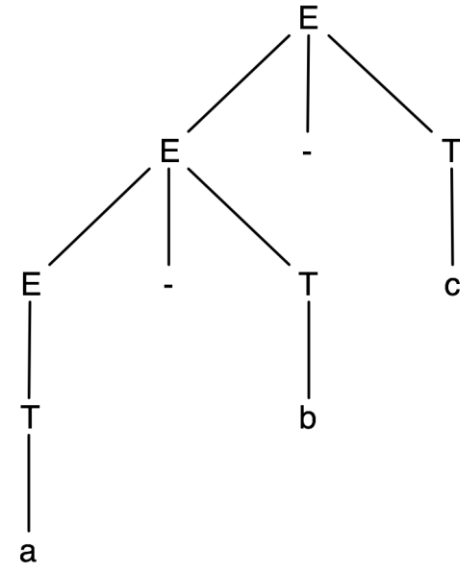
---

- Require right operand to not be bare expression

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

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

- 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

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

# A Different Problem

---

- ▶ How about the string  $a+b*c$  ?

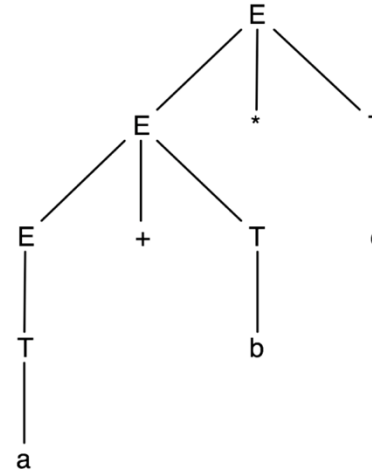
$E \rightarrow E+T \mid E-T \mid E*T \mid T$

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

- ▶ 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)

Derivation of  $a+b*c$ :

$$E \rightarrow E+T \rightarrow T+T \rightarrow P+T \rightarrow a+T \rightarrow a+T*P \rightarrow a+P*P \rightarrow a+b*P \rightarrow a+b*c$$

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