

Syntax Analysis – Part I

EECS 483 – Lecture 4

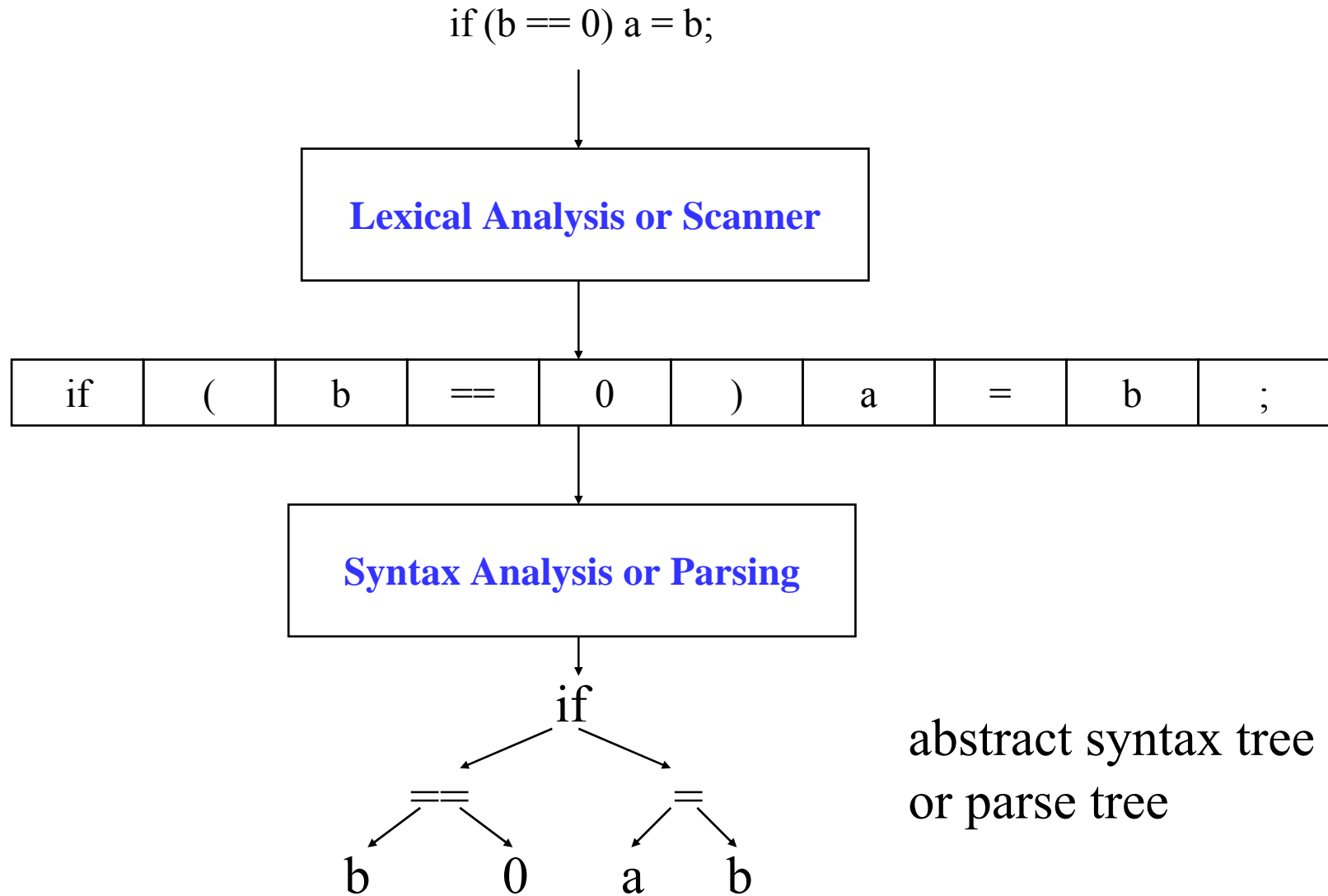
University of Michigan

Monday, September 17, 2006

Announcements/Reading

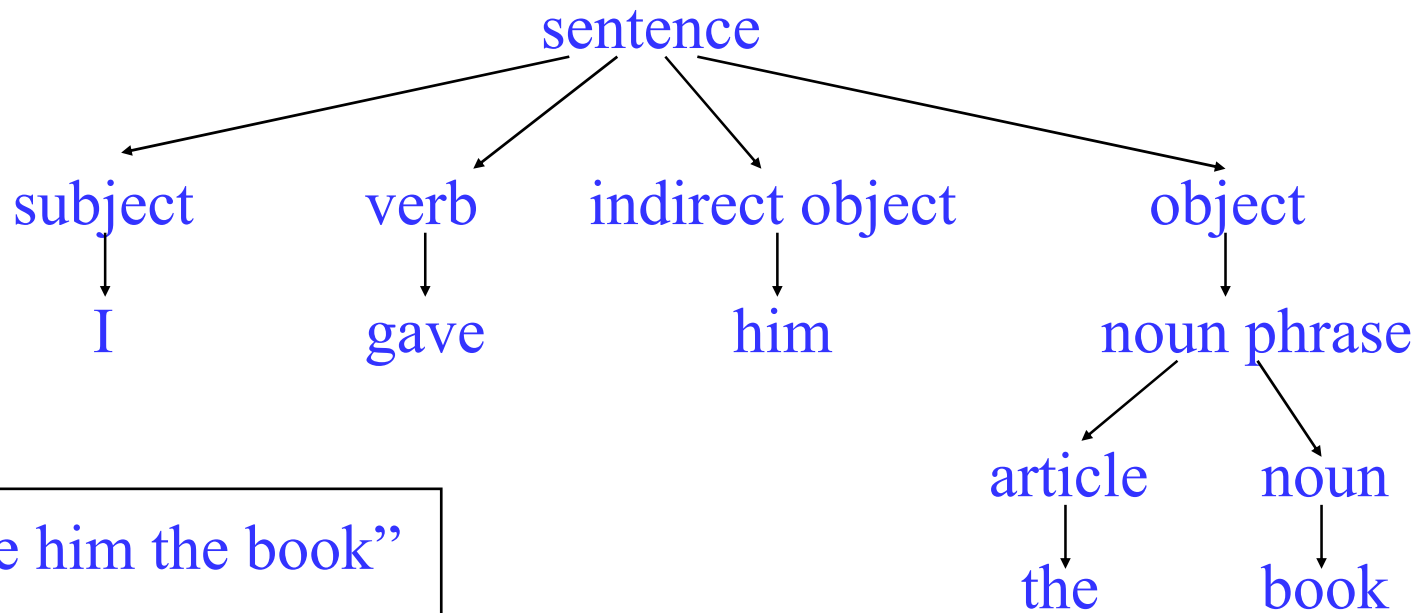
- ❖ Reading - Ch 4
 - » 4.1 – 4.3 for today
- ❖ Simon does not have office hours on Fri
 - » Just Tue/Thu, 1:30-3:30
 - » My mistake on the website

Where is Syntax Analysis Performed?



Parsing Analogy

- Syntax analysis for natural languages
 - Recognize whether a sentence is grammatically correct
 - Identify the function of each word



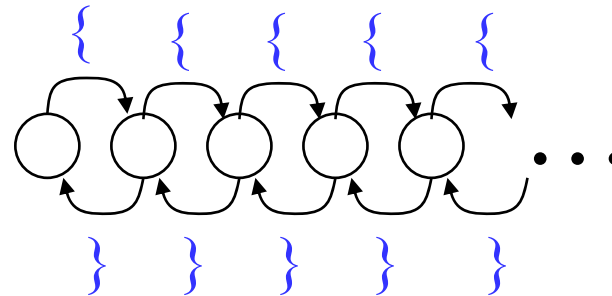
Syntax Analysis Overview

- ❖ Goal – Determine if the input token stream satisfies the syntax of the program
- ❖ What do we need to do this?
 - » An expressive way to describe the syntax
 - » A mechanism that determines if the input token stream satisfies the syntax description
- ❖ For lexical analysis
 - » Regular expressions describe tokens
 - » Finite automata = mechanisms to generate tokens from input stream

Just Use Regular Expressions?

- ❖ REs can expressively describe tokens
 - » Easy to implement via DFAs
- ❖ So just use them to describe the syntax of a programming language??
 - » **NO!** – They don't have enough power to express any non-trivial syntax
 - » Example – Nested constructs (blocks, expressions, statements) – Detect balanced braces:

{ } { } { } { }



- **We need unbounded counting!**
- **FSA's cannot count except in a strictly modulo fashion**

Context-Free Grammars

❖ Consist of 4 components:

- » Terminal symbols = token or ϵ
- » Non-terminal symbols = syntactic variables
- » Start symbol S = special non-terminal
- » Productions of the form LHS \rightarrow RHS
 - LHS = single non-terminal
 - RHS = string of terminals and non-terminals
 - Specify how non-terminals may be expanded

$S \rightarrow a S a$
$S \rightarrow T$
$T \rightarrow b T b$
$T \rightarrow \epsilon$

- ❖ Language generated by a grammar is the set of strings of terminals derived from the start symbol by repeatedly applying the productions
- » $L(G)$ = language generated by grammar G

CFG - Example

❖ Grammar for balanced-parentheses language

» $S \rightarrow (S) S$ ← ? Why is the final S required?

» $S \rightarrow \varepsilon$

- 1 non-terminal: S
- 2 terminals: “(”, “)”
- Start symbol: S
- 2 productions

❖ If grammar accepts a string, there is a derivation of that string using the productions

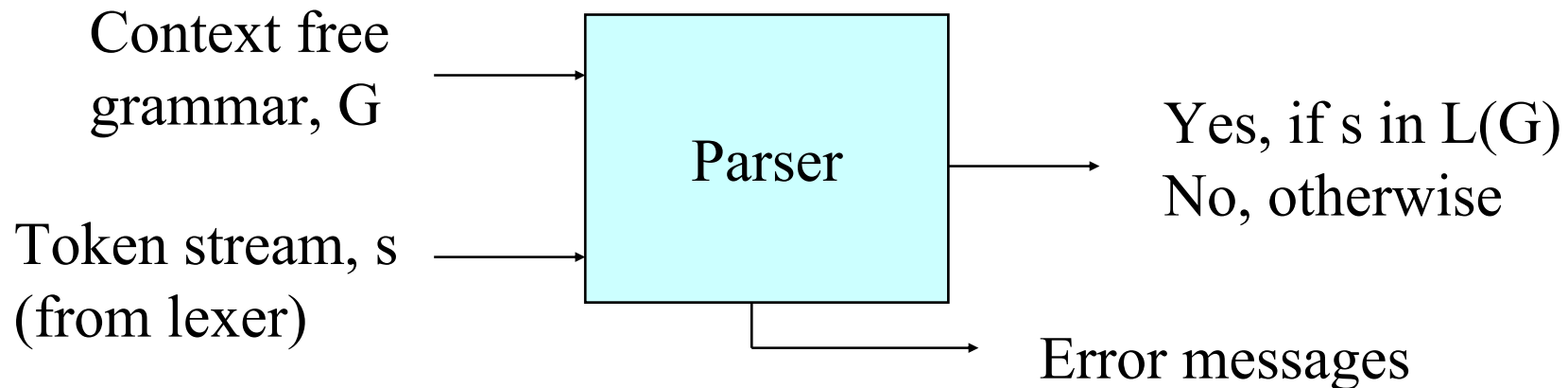
» “(())”

» $S = (S) \varepsilon = ((S) S) \varepsilon = ((\varepsilon) \varepsilon) \varepsilon = (())$

More on CFGs

- ❖ Shorthand notation – vertical bar for multiple productions
 - » $S \rightarrow a S a \mid T$
 - » $T \rightarrow b T b \mid \varepsilon$
- ❖ CFGs powerful enough to expression the syntax in most programming languages
- ❖ Derivation = successive application of productions starting from S
- ❖ Acceptance? = Determine if there is a derivation for an input token stream

A Parser



Syntax analyzers (parsers) = CFG acceptors which also output the corresponding derivation when the token stream is accepted

Various kinds: LL(k), LR(k), SLR, LALR

RE is a Subset of CFG

Can inductively build a grammar for each RE

ϵ	$S \rightarrow \epsilon$
a	$S \rightarrow a$
$R1 R2$	$S \rightarrow S1 S2$
$R1 R2$	$S \rightarrow S1 S2$
$R1^*$	$S \rightarrow S1 S \epsilon$

Where

$G1$ = grammar for $R1$, with start symbol $S1$

$G2$ = grammar for $R2$, with start symbol $S2$

Grammar for Sum Expression

❖ Grammar

- » $S \rightarrow E + S \mid E$
- » $E \rightarrow \text{number} \mid (S)$

❖ Expanded

- » $S \rightarrow E + S$
 - » $S \rightarrow E$
 - » $E \rightarrow \text{number}$
 - » $E \rightarrow (S)$
- 4 productions
2 non-terminals (S,E)
4 terminals: “(”, “)”, “+”, number
start symbol: S

Constructing a Derivation

- ❖ Start from S (the start symbol)
- ❖ Use productions to derive a sequence of tokens
- ❖ For arbitrary strings α , β , γ and for a production: $A \rightarrow \beta$
 - » A single step of the derivation is
 - » $\alpha A \gamma \Rightarrow \alpha \beta \gamma$ (substitute β for A)
- ❖ Example
 - » $S \rightarrow E + S$
 - » $(\underline{S} + E) + E \rightarrow (\underline{E + S} + E) + E$

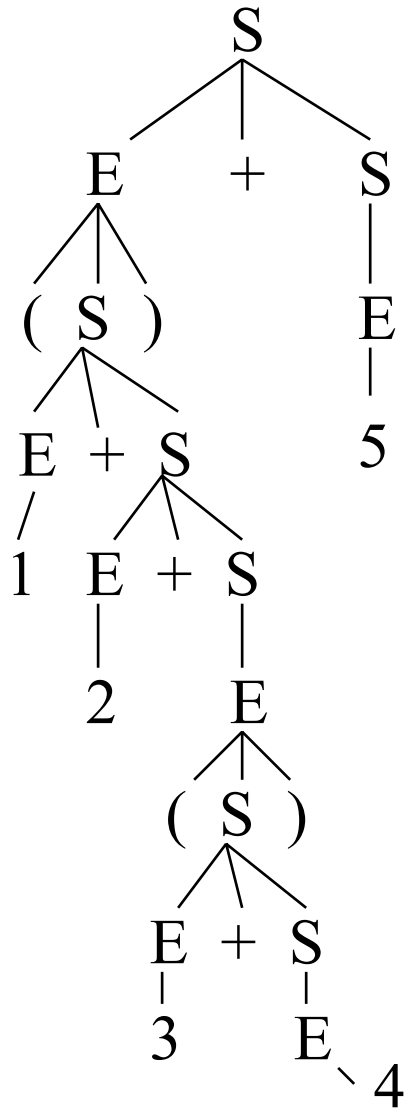
Class Problem

» $S \rightarrow E + S \mid E$

» $E \rightarrow \text{number} \mid (S)$

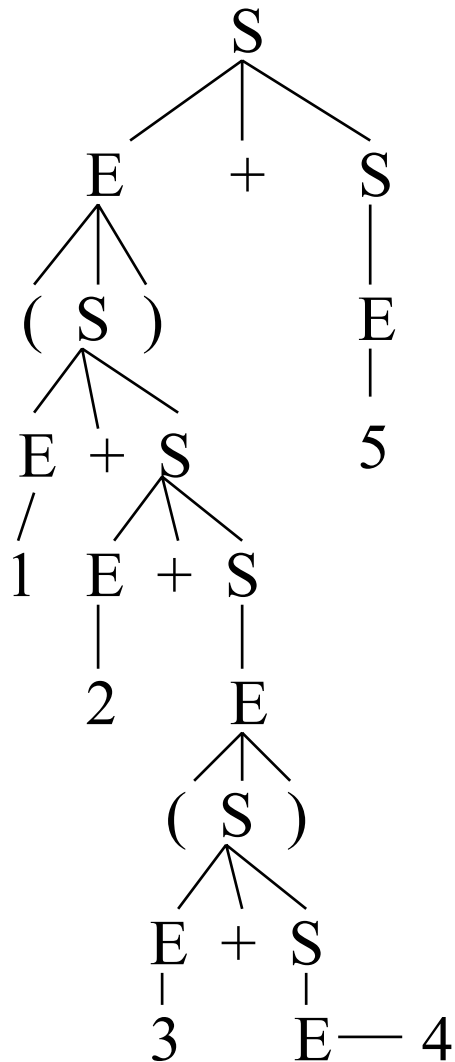
❖ Derive: $(1 + 2 + (3 + 4)) + 5$

Parse Tree

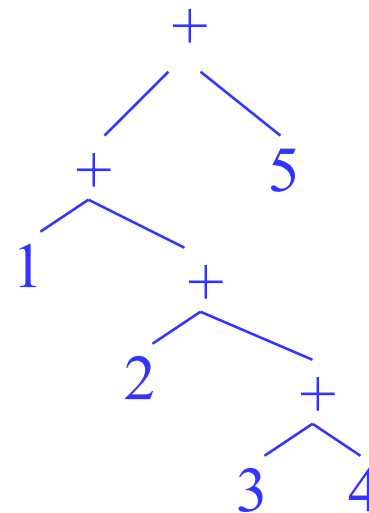
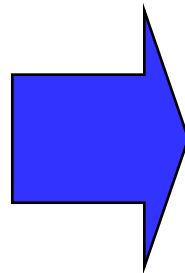


- Parse tree = tree representation of the derivation
- Leaves of the tree are terminals
- Internal nodes are non-terminals
- No information about the order of the derivation steps

Parse Tree vs Abstract Syntax Tree



Parse tree also called “concrete syntax”



AST discards (abstracts) unneeded information – more compact format

Derivation Order

- ❖ Can choose to apply productions in any order, select non-terminal and substitute RHS of production
- ❖ Two standard orders: left and right-most
- ❖ Leftmost derivation
 - » In the string, find the leftmost non-terminal and apply a production to it
 - » $E + S \rightarrow 1 + S$
- ❖ Rightmost derivation
 - » Same, but find rightmost non-terminal
 - » $E + S \rightarrow E + E + S$

Leftmost/Rightmost Derivation Examples

» $S \rightarrow E + S \mid E$

» $E \rightarrow \text{number} \mid (S)$

» Leftmost derive: $(1 + 2 + (3 + 4)) + 5$

$S \rightarrow E + S \rightarrow (S) + S \rightarrow (E + S) + S \rightarrow (1 + S) + S \rightarrow (1 + E + S) + S \rightarrow$
 $(1 + 2 + S) + S \rightarrow (1 + 2 + E) + S \rightarrow (1 + 2 + (S)) + S \rightarrow (1 + 2 + (E + S)) + S \rightarrow$
 $(1 + 2 + (3 + S)) + S \rightarrow (1 + 2 + (3 + E)) + S \rightarrow (1 + 2 + (3 + 4)) + S \rightarrow$
 $(1 + 2 + (3 + 4)) + E \rightarrow (1 + 2 + (3 + 4)) + 5$

» Now, rightmost derive the same input string

$S \rightarrow E + S \rightarrow E + E \rightarrow E + 5 \rightarrow (S) + 5 \rightarrow (E + S) + 5 \rightarrow (E + E + S) + 5 \rightarrow$
 $(E + E + E) + 5 \rightarrow (E + E + (S)) + 5 \rightarrow (E + E + (E + S)) + 5 \rightarrow$
 $(E + E + (E + E)) + 5 \rightarrow (E + E + (E + 4)) + 5 \rightarrow (E + E + (3 + 4)) + 5 \rightarrow$
 $(E + 2 + (3 + 4)) + 5 \rightarrow (1 + 2 + (3 + 4)) + 5$

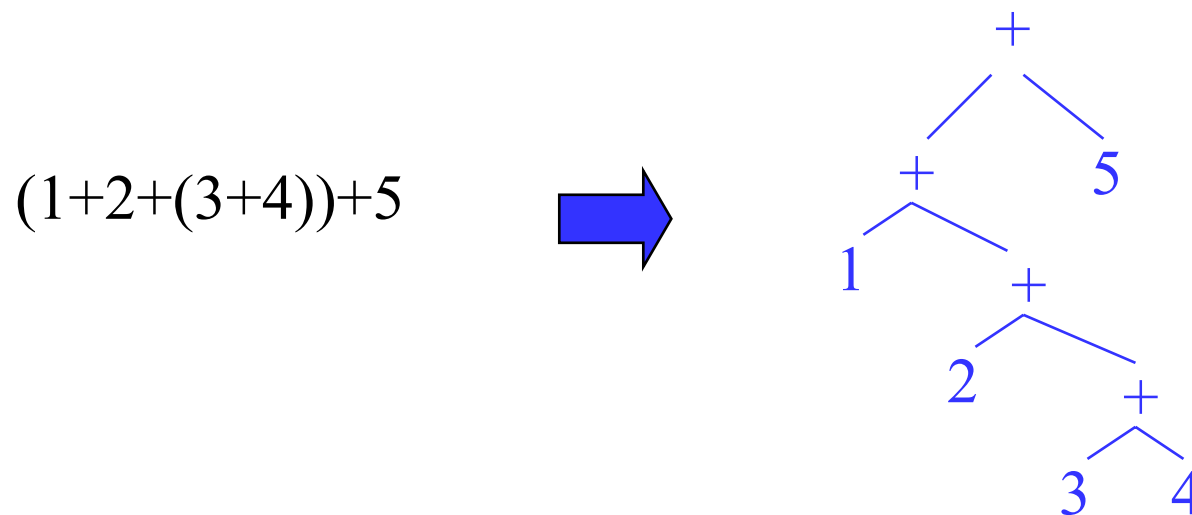
Result: Same parse tree: same productions chosen, but in diff order

Class Problem

- » $S \rightarrow E + S \mid E$
- » $E \rightarrow \text{number} \mid (S) \mid -S$
- ❖ Do the rightmost derivation of : $1 + (2 + -(3 + 4)) + 5$

Ambiguous Grammars

- ❖ In the sum expression grammar, leftmost and rightmost derivations produced identical parse trees
- ❖ $+$ operator associates to the right in parse tree regardless of derivation order



An Ambiguous Grammar

- ❖ + associates to the right because of the right-recursive production: $S \rightarrow E + S$
- ❖ Consider another grammar
 - » $S \rightarrow S + S \mid S * S \mid \text{number}$
- ❖ Ambiguous grammar = different derivations produce different parse trees
 - » More specifically, G is ambiguous if there are 2 distinct leftmost (rightmost) derivations for some sentence

Ambiguous Grammar - Example

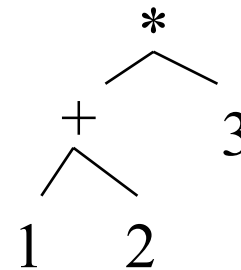
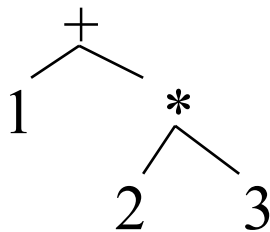
$$S \rightarrow S + S \mid S * S \mid \text{number}$$

Consider the expression: $1 + 2 * 3$

Derivation 1: $S \rightarrow S+S \rightarrow$
 $1+S \rightarrow 1+S*S \rightarrow 1+2*S \rightarrow$
 $1+2*3$

Derivation 2: $S \rightarrow S*S \rightarrow$
 $S+S*S \rightarrow 1+S*S \rightarrow 1+2*S \rightarrow$
 $1+2*3$

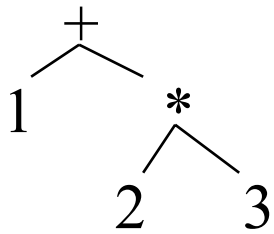
2 leftmost derivations



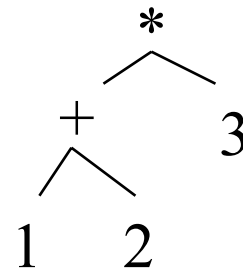
But, obviously not equal!

Impact of Ambiguity

- ❖ Different parse trees correspond to different evaluations!
- ❖ Thus, program meaning is not defined!!



= 7



= 9

Can We Get Rid of Ambiguity?

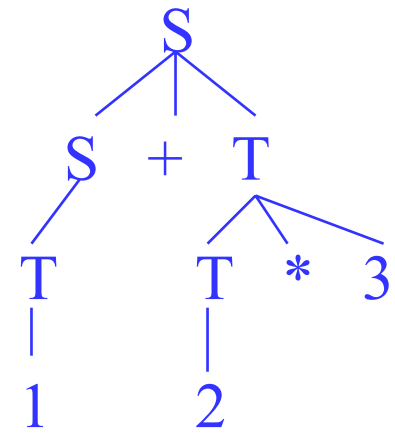
- ❖ Ambiguity is a function of the grammar, not the language!
- ❖ A context-free language L is inherently ambiguous if all grammars for L are ambiguous
- ❖ Every deterministic CFL has an unambiguous grammar
 - » So, no deterministic CFL is inherently ambiguous
 - » No inherently ambiguous programming languages have been invented
- ❖ To construct a useful parser, must devise an unambiguous grammar

Eliminating Ambiguity

- ❖ Often can eliminate ambiguity by adding nonterminals and allowing recursion only on right or left

- » $S \rightarrow S + T \mid T$

- » $T \rightarrow T * \text{num} \mid \text{num}$



- » T non-terminal enforces precedence
- » Left-recursion; left associativity

A Closer Look at Eliminating Ambiguity

- ❖ Precedence enforced by
 - » Introduce distinct non-terminals for each precedence level
 - » Operators for a given precedence level are specified as RHS for the production
 - » Higher precedence operators are accessed by referencing the next-higher precedence non-terminal

Associativity

- ❖ An operator is either left, right or non associative
 - » Left: $a + b + c = (a + b) + c$
 - » Right: $a \wedge b \wedge c = a \wedge (b \wedge c)$
 - » Non: $a < b < c$ is illegal (thus undefined)
- ❖ Position of the recursion relative to the operator dictates the associativity
 - » Left (right) recursion \rightarrow left (right) associativity
 - » Non: Don't be recursive, simply reference next higher precedence non-terminal on both sides of operator

Class Problem

$S \rightarrow S + S \mid S - S \mid S * S \mid S / S \mid (S) \mid -S \mid S \wedge S \mid \text{num}$

Enforce the standard arithmetic precedence rules and remove all ambiguity from the above grammar

Precedence (high to low)

$()$, unary $-$

\wedge

$*$, $/$

$+$, $-$

Associativity

$\wedge =$ right

rest are left