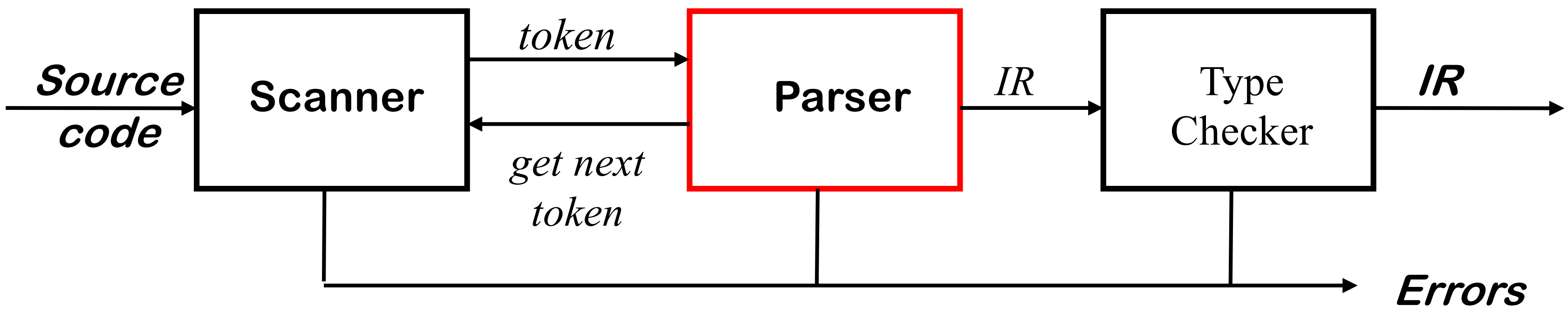
Computer Science 160
Translation of Programming Languages

Instructor: Christopher Kruegel

Syntactic Analysis (Parsing)

The Front End: Parser



Parser

- Input: A sequence of tokens representing the source program
- Output: A parse tree (in practice, an abstract syntax tree)
- While generating the parse tree, parser checks the stream of tokens for grammatical correctness
  - Checks the context-free syntax
- Parser builds an IR representation of the code
  - Generates an abstract syntax tree
- Guides checking at deeper levels than syntax

Specifying Syntax with a Grammar

- Need a mathematical model of syntax — a grammar *G*
  - Context-free grammars
- Need an algorithm for testing membership in *L(G)*
  - Parsing algorithms
- Parsing is the process of discovering a *derivation* for some sentence from the rules of the grammar
  - Equivalently, it is the process of discovering a parse tree
- Natural language analogy
  - Lexical rules correspond to rules that define the valid words
  - Grammar rules correspond to rules that define valid sentences

Specifying Syntax with a Grammar

Context-free syntax is specified with a context-free grammar

Formally, a grammar is a four-tuple, $G = (S,N,T,P)$

- T is a set of *terminal symbols*
  - These correspond to tokens returned by the scanner
  - For the parser tokens are indivisible units of syntax
- N is a set of *non-terminal symbols*
  - These are syntactic variables that can be substituted during a derivation
  - Variables that denote sets of substrings occurring in the language
- S is the *start symbol* : $S \in N$
  - All the strings in $L(G)$ are derived from the start symbol
- P is a set of *productions* or *rewrite rules* : $P : N \rightarrow (N \cup T)^*$

An Example Grammar

1	*Start*	$\rightarrow$	*Expr*
2	*Expr*	$\rightarrow$	*Expr Op Expr*
3		|	num
4		|	id
5	*Op*	$\rightarrow$	+
6		|	-
7		|	*
8		|	/

Start symbol: *S* = *Start*
Non-terminal symbols: *N* = { *Start, Expr, Op* }
Terminal symbols: *T* = { num, id, +, -, *, / }
Productions: *P* = { 1, 2, 3, 4, 5, 6, 7, 8 } (shown above)

Context Free Grammar

- Programming languages have a set of rules that describe the syntactic structure of well-formed programs
- A context free grammar is precise and understandable, yet powerful enough to express these rules
- It is so effective because it embraces the recursive nature of most programming languages
  - Example sentence: if(x){ if(y){ if(z) { } } }
  - Example grammar: I → **if**(*id*) { I }
  - This requires a variable number of states and is thus beyond the ability of regular expressions

Vocabulary

- *Sentence* of G: String of terminals in L(G)
- *Sentential Form* of G: String of non-terminals and terminals from which a sentence of G can be derived.
- *Derivation*: A sequence of rewrites according to productions
- *Production*: A rule which takes a non-terminal and maps it to a string of non-Terminals and terminals
- The process or discovering a derivation is called ***parsing***

Derivations

- At each step, we make two choices
  1. Choose a non-terminal to replace
  2. Choose a production to apply
- Different choices lead to different derivations

Two types of derivation are of interest

- Leftmost derivation — replace leftmost non-terminal at each step
- Rightmost derivation — replace rightmost non-terminal at each step

These are the two *systematic* derivations (the first choice is fixed)

Two Derivations for *x - 2 * y*

Rule	Sentential Form
—	*S*
1	*Expr*
2	*Expr Op Expr*
4	<id,*x*> *Op Expr*
6	<id,*x*> - *Expr*
2	<id,*x*> - *Expr Op Expr*
3	<id,*x*> - <num,*2*> *Op Expr*
7	<id,*x*> - <num,*2*> * *Expr*
4	<id,*x*> - <num,*2*> * <id,*y*>

Leftmost derivation

Rule	Sentential Form
—	*S*
1	*Expr*
2	*Expr Op Expr*
4	*Expr Op* <id,*y*>
7	*Expr* * <id,*y*>
2	*Expr Op Expr* * <id,*y*>
3	*Expr Op* <num,*2*> * <id,*y*>
6	*Expr* - <num,*2*> * <id,*y*>
4	<id,*x*> - <num,*2*> * <id,*y*>

Rightmost derivation

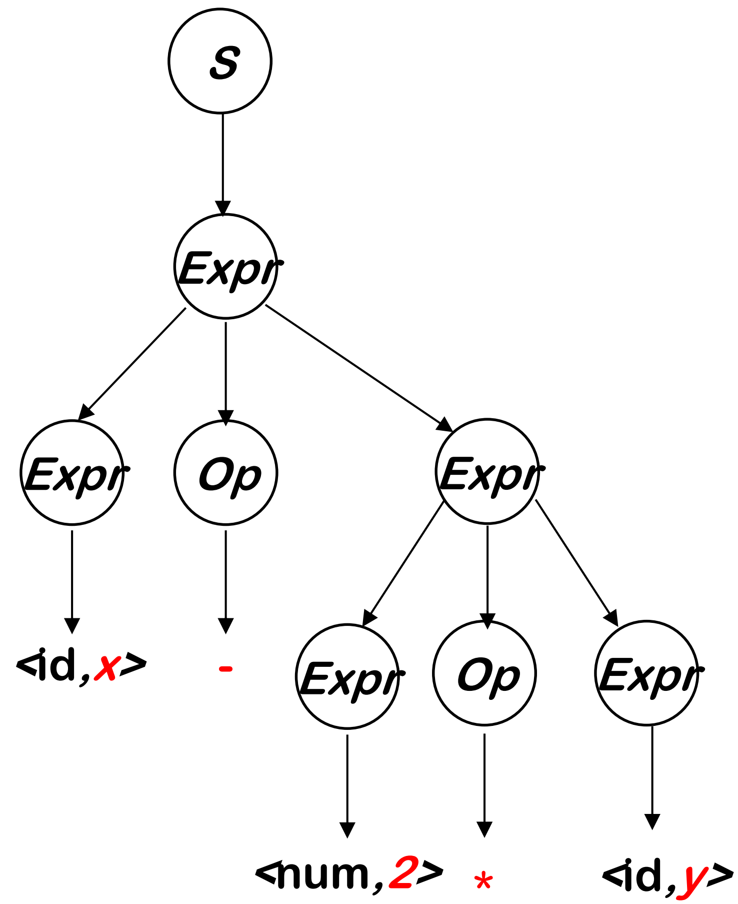
In both cases, $S \Rightarrow^*$ id - num * id

- Note that these two derivations produce different parse trees
- The parse trees imply different evaluation orders!

Derivations and Parse Trees

Leftmost derivation

Rule	Sentential Form
—	*S*
1	*Expr*
2	*Expr Op Expr*
4	<id,*x*> *Op Expr*
6	<id,*x*> - *Expr*
2	<id,*x*> - *Expr Op Expr*
3	<id,*x*> - <num,*2*> *Op Expr*
7	<id,*x*> - <num,*2*> * *Expr*
4	<id,*x*> - <num,*2*> * <id,*y*>

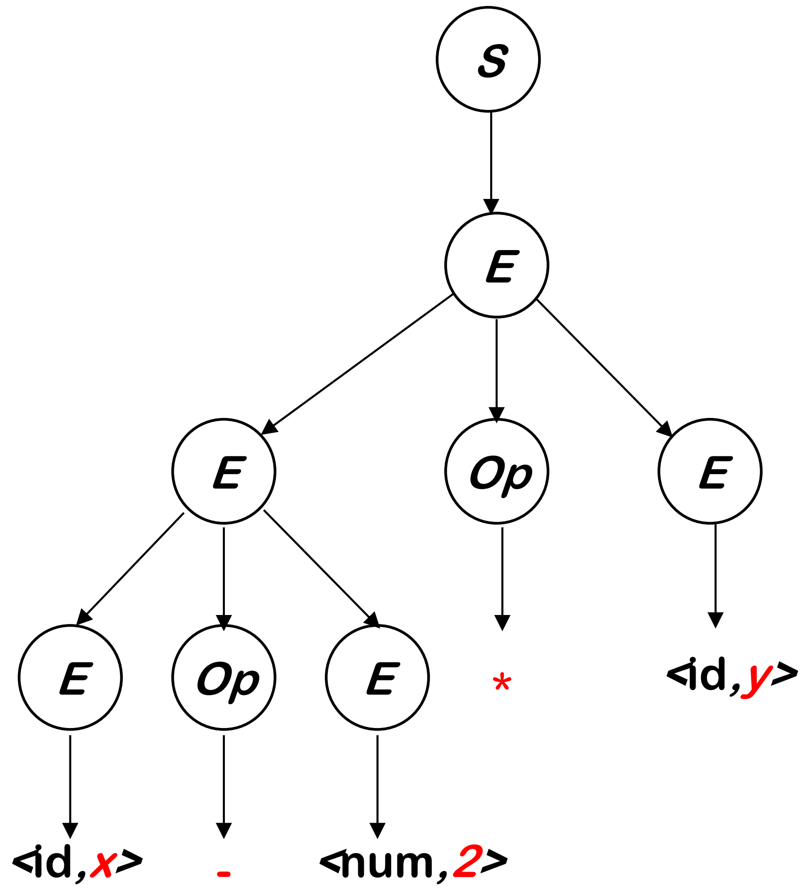


This evaluates as x - (2 * y)

Derivations and Parse Trees

Rightmost derivation

Rule	Sentential Form
—	*S*
1	*Expr*
2	*Expr Op Expr*
4	*Expr Op* <id,*y*>
7	*Expr* * <id,*y*>
2	*Expr Op Expr* * <id,*y*>
3	*Expr Op* <num,*2*> * <id,*y*>
6	*Expr* - <num,*2*> * <id,*y*>
4	<id,*x*> - <num,*2*> * <id,*y*>

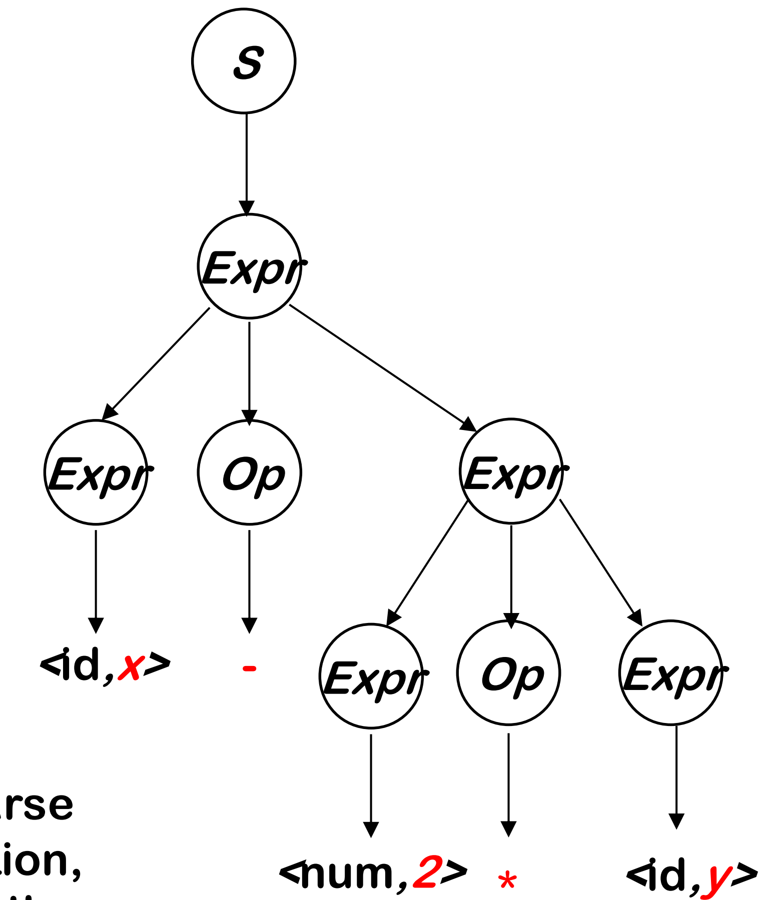


This evaluates as (x - 2) * y

Another Rightmost Derivation

Another rightmost derivation

Rule	Sentential Form
—	*S*
1	*Expr*
2	*Expr Op Expr*
2	*Expr Op Expr Op Expr*
4	*Expr Op Expr Op* <id,*y*>
7	*Expr Op Expr* * <id,*y*>
3	*Expr Op* <num,*2*> * <id,*y*>
6	*Expr* - <num,*2*> * *Expr*
4	<id,*x*> - <num,*2*> * <id,*y*>



This evaluates as x - (2 * y)

This parse tree is different than the parse tree for the previous rightmost derivation, but it is the same as the parse tree for the previous leftmost derivation

Ambiguity

- One grammar can produce *two different parse trees* for the same sentence.
  - From a theoretical standpoint, it is fine. The sentence can be derived from the grammar and everyone is happy
  - The problem is that the way the program is interpreted stems from the parse tree

- We need to ensure that for each sentence in G, there is only one parse tree for that sentence
  - If there is *more than one parse tree* for a given sentence, our grammar is **ambiguous**
  - *To show a grammar G is ambiguous, find a sentence in G with two parse trees*

Ambiguous Grammars

- If a grammar has more than one leftmost derivation for some sentence, then the grammar is ambiguous
- If a grammar has more than one rightmost derivation for some sentence, then the grammar is ambiguous
- If a grammar produces more than one parse tree for some sentence than it is ambiguous

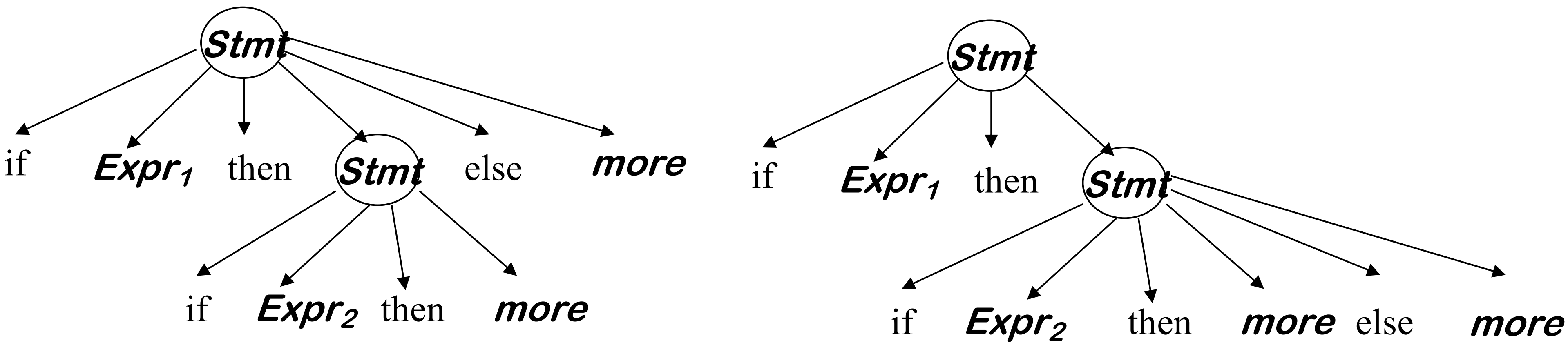
Classic example — the dangling-else problem

1	*Stmt*	$\rightarrow$	if *Expr* then *Stmt*
2		|	if *Expr* then *Stmt* else *Stmt*
		|	*more*

Ambiguity

This sentential form has two parse trees

if *Expr*$_1$ then if *Expr*$_2$ then *more* else *more*



production 2, then production 1

production 1, then production 2

Ambiguity

Removing the ambiguity

- Must rewrite the grammar to avoid generating the problem
- Match each <u>else</u> to innermost unmatched <u>if</u> *(common sense rule)*

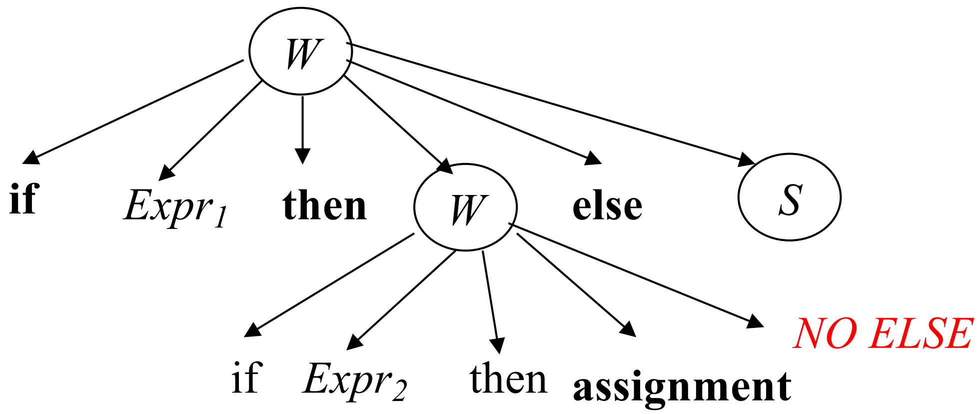
- New rules enforce that only a matched statement can come before an else

Stmt	→	**If** *Expr* **then** *Stmt*
	|	**If** *Expr* **then** *WithElse* **else** *Stmt*
	|	**Assignment**
Withelse	→	**If** *Expr* **then** *WithElse* **else** *WithElse*
	|	**Assignment**

With this grammar, the example has only one parse tree

Ambiguity

Try the dangling-else derivations:



Can't make a parse tree where the "else" associates with the first "if"

Parse Trees and Precedence

Two parse trees for our *expressions grammar* point out a problem:

It has no notion of precedence (implied order of evaluation between different operators)

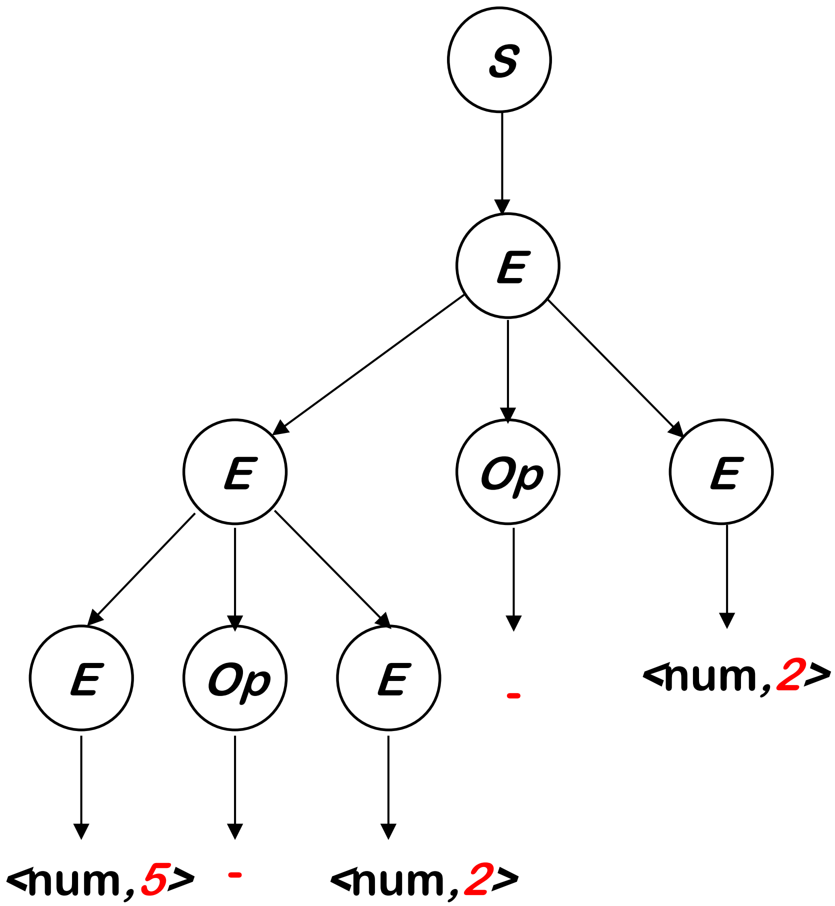
To add precedence

- Create a non-terminal for each *level of precedence*
- Isolate the corresponding part of the grammar
- Force parser to recognize high precedence sub-expressions first

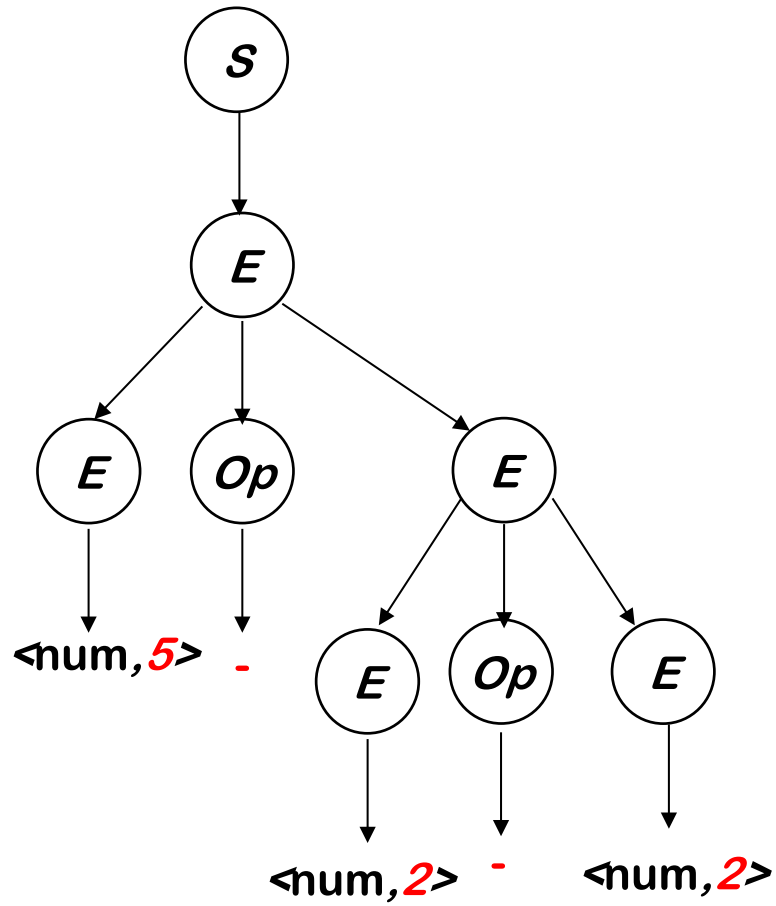
For algebraic expressions

- Multiplication and division, first
- Subtraction and addition, next

Parse Trees and Associativity



Result is 1



Result is 5

Precedence and Associativity

Adding the standard algebraic precedence and using left recursion produces:

1	*S*	→	*Expr*
2	*Expr*	→	*Expr* + *Term*
3		|	*Expr* - *Term*
4		|	*Term*
5	*Term*	→	*Term* * *Factor*
6		|	*Term* / *Factor*
7		|	*Factor*
8	*Factor*	→	num
9		|	id

This grammar is slightly larger

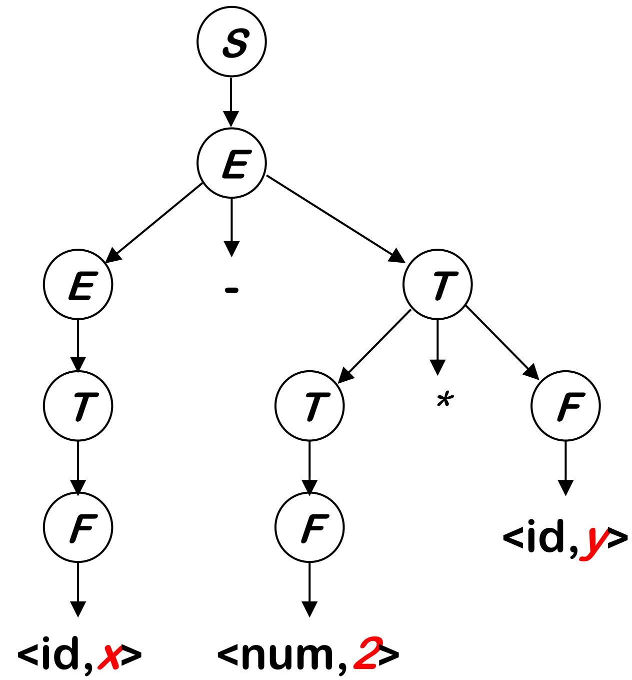
- **Takes more rewriting to reach some of the terminal symbols**
- **Encodes expected precedence**
- **Enforces left-associativity**
- **Produces same parse tree under leftmost & rightmost derivations**

Let's see how it parses our example

Precedence

Rule	Sentential Form
	S
1	*Expr*
3	*Epr* - *Term*
7	*Term* - *Term*
8	*Factor* - *Term*
3	<id,*x*> - *Term*
7	<id,*x*> - *Term* * *Factor*
8	<id,*x*> - *Factor* * *Factor*
4	<id,*x*> - <num,*2*> * *Factor*
7	<id,*x*> - <num,*2*> * <id,*y*>

The leftmost derivation



Its parse tree

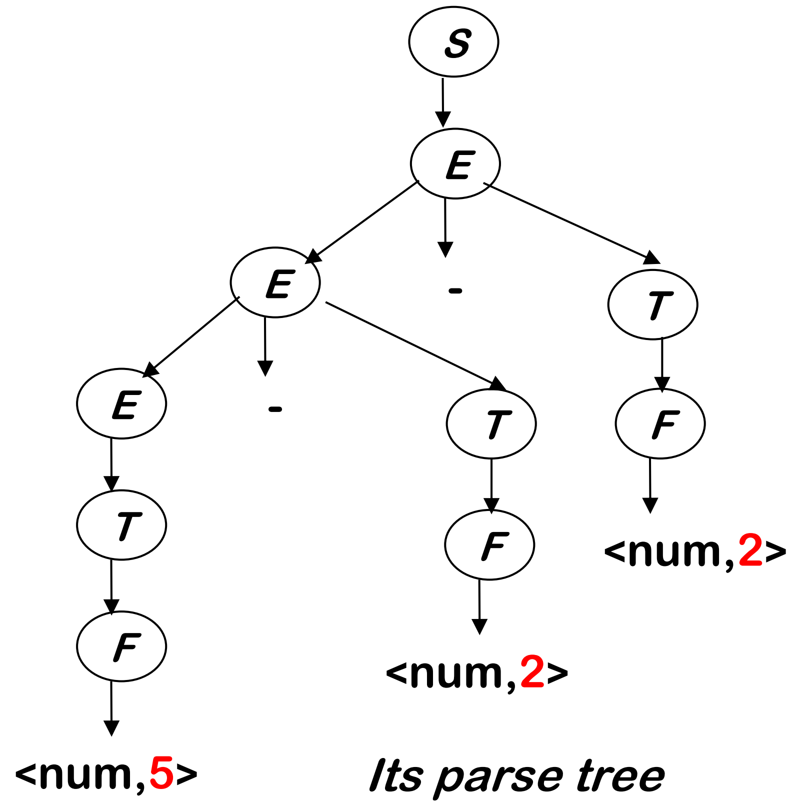
This produces x - (2 * y) , along with an appropriate parse tree.

Both the leftmost and rightmost derivations give the same parse tree and the same evaluation order, because the grammar directly encodes the desired precedence.

Associativity

Rule	Sentential Form
	S
1	*Expr*
3	*Epr* - *Term*
7	*Expr* - *Factor*
8	*Expr* - <num,2>
3	*Expr* - *Term* - <num,2>
7	*Expr* - *Factor* - <num,2>
8	*Expr* - <num,2> - <num,2>
4	*Term* - <num,2> - <num,2>
7	*Factor* - <num,2> - <num,2>
8	<num,5> - <num,2> - <num,2>

The rightmost derivation



Its parse tree

This produces (5 - 2) - 2 , along with an appropriate parse tree.

Both the leftmost and rightmost derivations give the same parse tree and the same evaluation order

Parsing Techniques

Top-down parsers *(LL(1), recursive descent parsers)*

- Start at the root of the parse tree from the start symbol and grow toward leaves (similar to a derivation)
- Pick a production and try to match the input
- Bad "pick" $\Rightarrow$ may need to backtrack
- Some grammars are backtrack-free *(predictive parsing)*

Bottom-up parsers *(LR(1), shift-reduce parsers)*

- Start at the leaves and grow toward root
- We can think of the process as reducing the input string to the start symbol
- At each reduction step, a particular substring matching the right-side of a production is replaced by the symbol on the left-side of the production
- Bottom-up parsers handle a large class of grammars