Parse Trees & Ambiguity

Parse Trees

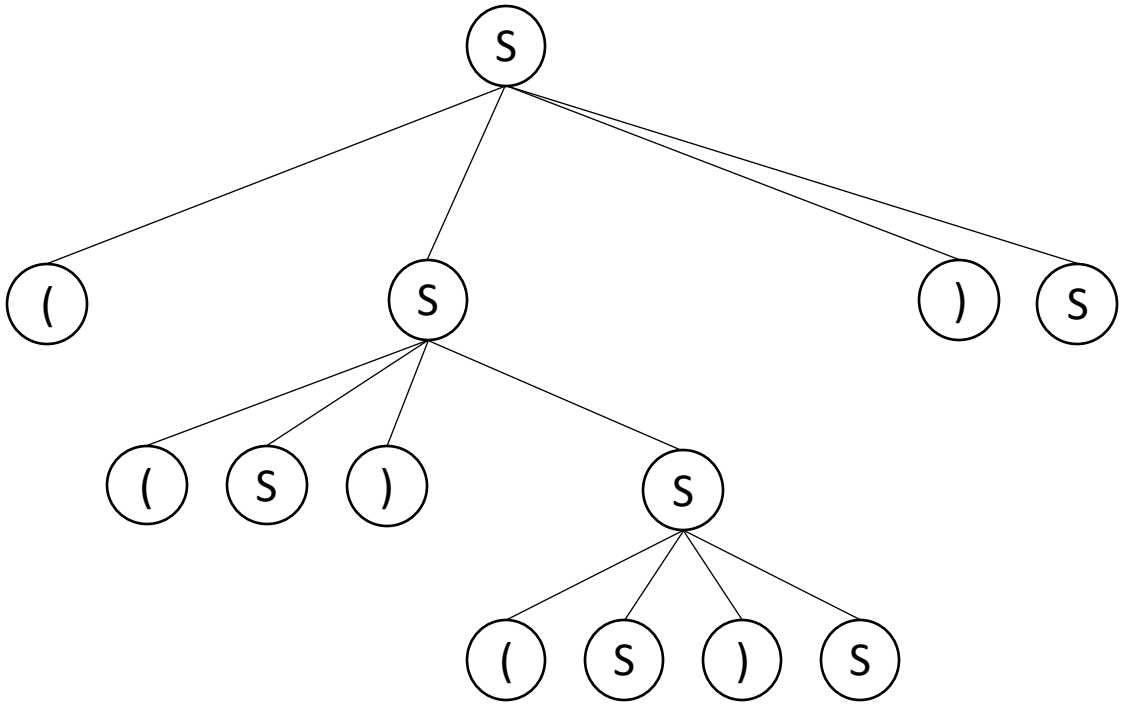
- Each derivation uniquely defines a **parse tree** for a string, which represents the structure of the string in terms of the grammar.

  $S \Rightarrow (S)S \Rightarrow ((S)S)S \Rightarrow (()S)S \Rightarrow (()(S)S)S \Rightarrow (()()S)S \Rightarrow (()())S \Rightarrow (()())$

Parse Trees

- Each derivation uniquely defines a **parse tree** for a string, which represents the structure of the string in terms of the grammar.

S
⇒ (S)S
⇒ ((S)S)S
⇒ (()S)S
⇒ (()(S)S)S
⇒ (()()S)S
⇒ (()())S
⇒ (()())



Parse Trees

- Each derivation uniquely defines a **parse tree** for a string, which represents the structure of the string in terms of the grammar.

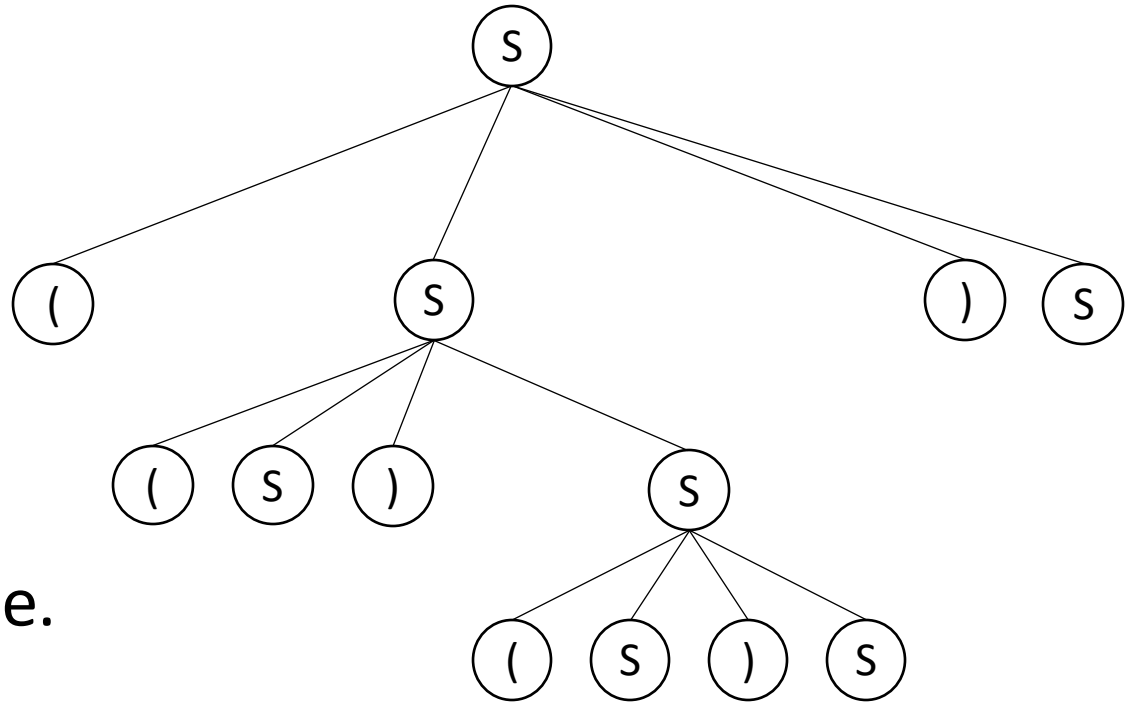
$$S \Rightarrow (S)S \Rightarrow (S) \Rightarrow ((S)S) \Rightarrow ((S)(S)S) \Rightarrow ((S)(S)) \Rightarrow (()(S)) \Rightarrow (()())$$

Parse Trees

- Each derivation uniquely defines a **parse tree** for a string, which represents the structure of the string in terms of the grammar.

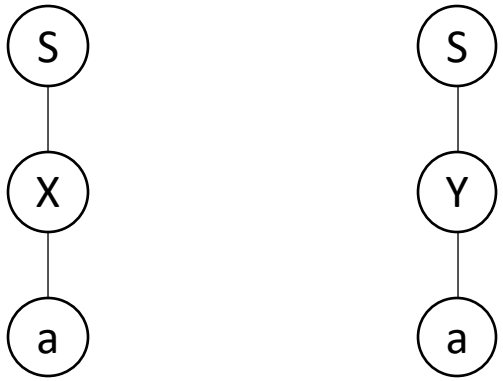
S
⇒ (S)S
⇒ (S)
⇒ ((S)S)
⇒ ((S)(S)S)
⇒ ((S)(S))
⇒ (()(S))
⇒ (()())

Same tree.



Ambiguous Grammars

- Can the same string have multiple parse trees?
- For some grammars, yes.
- Consider S → X | Y, X → a, Y → a. Two parse trees for "a".
- If there are two distinct parse trees for the same string, the grammar is **ambiguous.**



Ambiguity and Derivations

- We saw that even if there are multiple derivations for a string, they might correspond to the same tree.
- Thus a string having multiple derivations does *not* imply the grammar is ambiguous.
- However, if a string has multiple derivations, and those derivations expand nonterminal symbols in a *consistent order*, this does imply the grammar is ambiguous.
- A **leftmost derivation** is a derivation that always expands the *leftmost* nonterminal when there is a choice. Similarly, a **rightmost derivation** always expands the rightmost nonterminal.

Ambiguity and Derivations

- Consider this grammar: S → (S) | SS | ε
- There are two **leftmost derivations** of the string ()()(). Both start with:

$$S \Rightarrow SS$$

- Then we have two choices for how to expand the **leftmost** S:

$$S \Rightarrow \mathbf{S}S \Rightarrow \mathbf{(S)}S \Rightarrow ()S \Rightarrow ()SS \Rightarrow ()(S)S \Rightarrow ()()S \Rightarrow ()()(S) \Rightarrow ()()()$$
$$S \Rightarrow \mathbf{S}S \Rightarrow \mathbf{SS}S \Rightarrow (S)SS \Rightarrow ()SS \Rightarrow ()(S)S \Rightarrow ()()S \Rightarrow ()()(S) \Rightarrow ()()()$$

- Actually, there are infinitely many leftmost derivations of this string. Can you think of a third one?

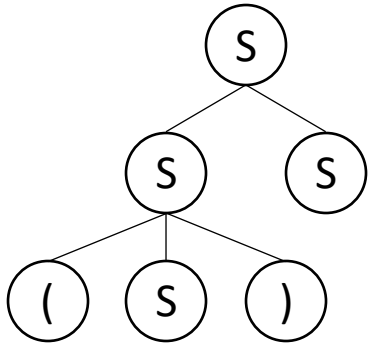
Ambiguity and Derivations

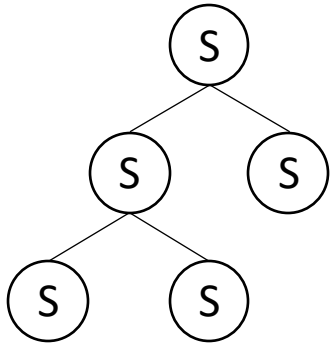
- Consider this grammar: S → (S) | SS | ε
- There are two **leftmost derivations** of the string ()()().

  S ⇒ **S**S ⇒ **(S)**S ⇒ ()S ⇒ ()SS ⇒ ()(S)S ⇒ ()()S ⇒ ()()(S) ⇒ ()()()

  S ⇒ **S**S ⇒ **SS**S ⇒(S)SS ⇒ ()SS ⇒ ()(S)S ⇒ ()()S ⇒ ()()(S) ⇒ ()()()

- Notice that these correspond to different parse trees, since we used two different rules to expand the same symbol at the same point.





Ambiguity in Arithmetic Expressions

- The following example shows that ambiguity can cause real problems in the context of parsing programming languages.
- Here's an ambiguous grammar for arithmetic expressions with addition, multiplication and 3:

$$E \rightarrow E + E \mid E * E \mid 3$$

- The expression 3 + 3 * 3 has two distinct parse trees, corresponding to these two derivations:

$E \Rightarrow E + E \Rightarrow 3 + E \Rightarrow 3 + E * E \Rightarrow 3 + 3 * E \Rightarrow 3 + 3 * 3$

$E \Rightarrow E * E \Rightarrow E + E * E \Rightarrow 3 + E * E \Rightarrow 3 + 3 * E \Rightarrow 3 + 3 * 3$

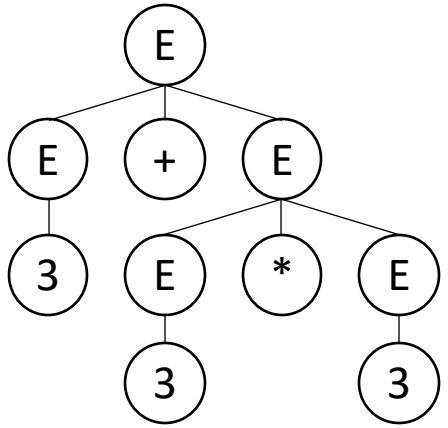
Ambiguity in Arithmetic Expressions

- Ambiguous grammar for arithmetic expressions with addition, multiplication and 3:

$$E \rightarrow E + E \mid E * E \mid 3$$

- Parse tree 1:

$$E \Rightarrow E + E \Rightarrow 3 + E \Rightarrow 3 + E * E \Rightarrow 3 + 3 * E \Rightarrow 3 + 3 * 3$$

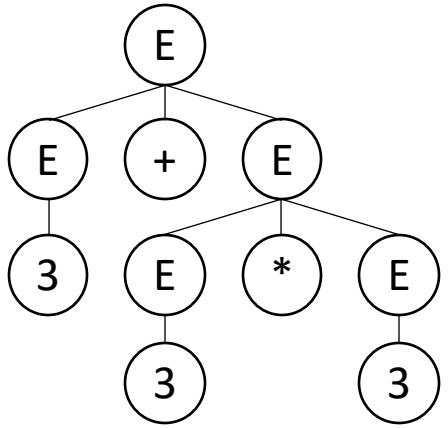


Ambiguity in Arithmetic Expressions

- Ambiguous grammar for arithmetic expressions with addition, multiplication and 3:

$$E \rightarrow E + E \mid E * E \mid 3$$

- If we used this tree to evaluate the expression, we would get: 3 + 3 * 3 = 3 + 9 = 12

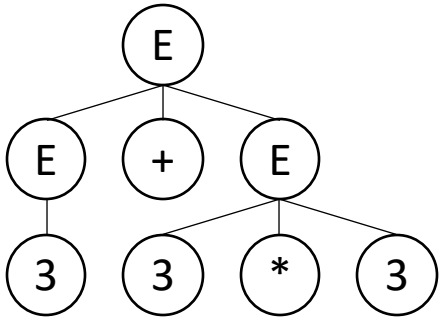


Ambiguity in Arithmetic Expressions

- Ambiguous grammar for arithmetic expressions with addition, multiplication and 3:

$$E \rightarrow E + E \mid E * E \mid 3$$

- If we used this tree to evaluate the expression, we would get: 3 + 3 * 3 = 3 + 9 = 12

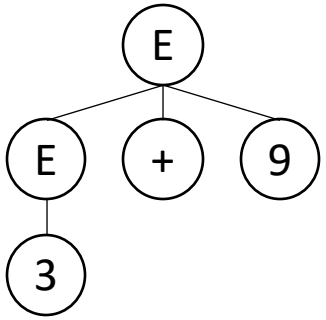


Ambiguity in Arithmetic Expressions

- Ambiguous grammar for arithmetic expressions with addition, multiplication and 3:

$$E \rightarrow E + E \mid E * E \mid 3$$

- If we used this tree to evaluate the expression, we would get: 3 + 3 * 3 = 3 + 9 = 12

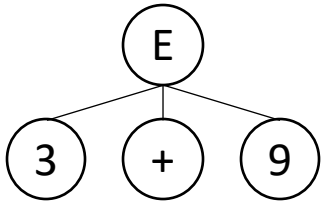


Ambiguity in Arithmetic Expressions

- Ambiguous grammar for arithmetic expressions with addition, multiplication and 3:

$$E \rightarrow E + E \mid E * E \mid 3$$

- If we used this tree to evaluate the expression, we would get:
  3 + 3 * 3 = 3 + 9 = 12

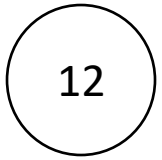


Ambiguity in Arithmetic Expressions

- Ambiguous grammar for arithmetic expressions with addition, multiplication and 3:

$$E \rightarrow E + E \mid E * E \mid 3$$

- If we used this tree to evaluate the expression, we would get:
  3 + 3 * 3 = 3 + 9 = 12



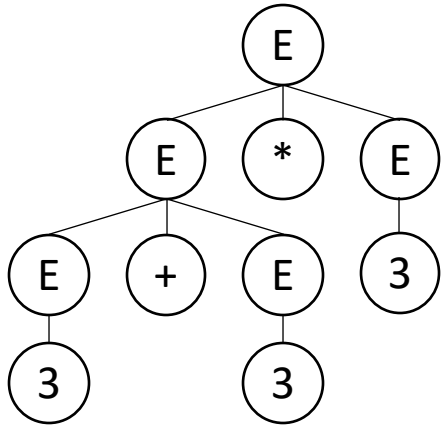
Ambiguity in Arithmetic Expressions

- Ambiguous grammar for arithmetic expressions with addition, multiplication and 3:

$$E \rightarrow E + E \mid E * E \mid 3$$

- Parse tree 2:

$$E \Rightarrow E * E \Rightarrow E + E * E \Rightarrow 3 + E * E \Rightarrow 3 + 3 * E \Rightarrow 3 + 3 * 3$$

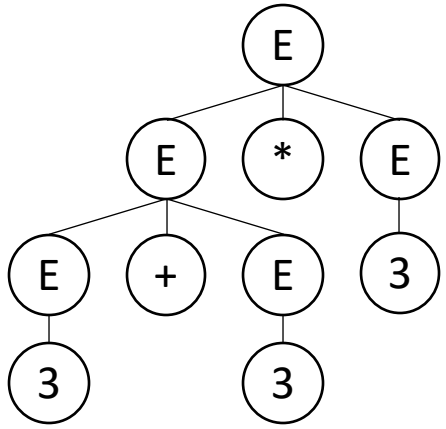


Ambiguity in Arithmetic Expressions

- Ambiguous grammar for arithmetic expressions with addition, multiplication and 3:

$$E \rightarrow E + E \mid E * E \mid 3$$

- If we use this tree to evaluate the expression…

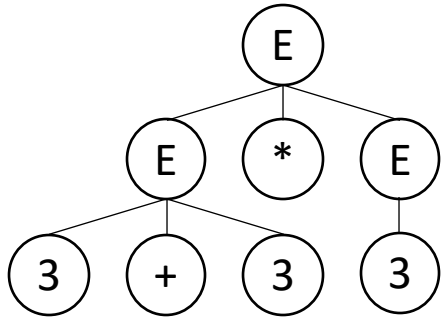


Ambiguity in Arithmetic Expressions

- Ambiguous grammar for arithmetic expressions with addition, multiplication and 3:

$$E \rightarrow E + E \mid E * E \mid 3$$

- If we use this tree to evaluate the expression…

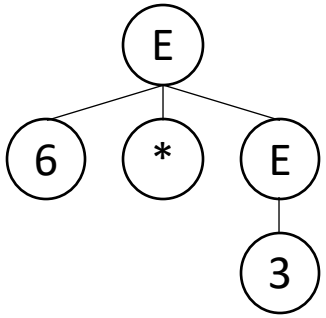


Ambiguity in Arithmetic Expressions

- Ambiguous grammar for arithmetic expressions with addition, multiplication and 3:

$$E \rightarrow E + E \mid E * E \mid 3$$

- If we use this tree to evaluate the expression…

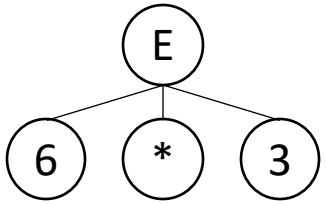


Ambiguity in Arithmetic Expressions

- Ambiguous grammar for arithmetic expressions with addition, multiplication and 3:

$$E \rightarrow E + E \mid E * E \mid 3$$

- If we use this tree to evaluate the expression…

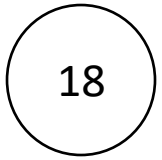


Ambiguity in Arithmetic Expressions

- Ambiguous grammar for arithmetic expressions with addition, multiplication and 3:

$$E \rightarrow E + E \mid E * E \mid 3$$

- If we use this tree to evaluate the expression…
  We get 3 + 3 * 3 = 18 (Wrong!!)

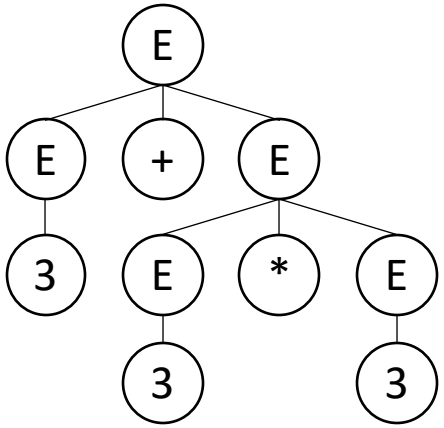


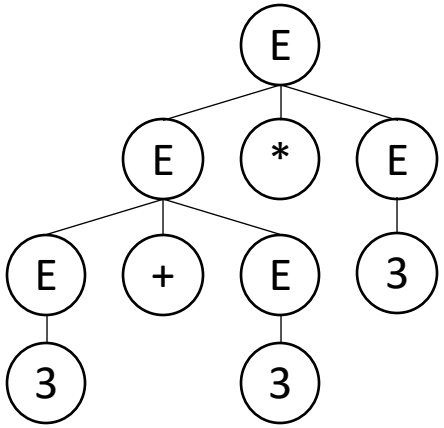
Ambiguity in Arithmetic Expressions

- Ambiguous grammar for arithmetic expressions with addition, multiplication and 3:

$$E \rightarrow E + E \mid E * E \mid 3$$

- There are two parse trees, but only one corresponds to the correct *order of operations.*



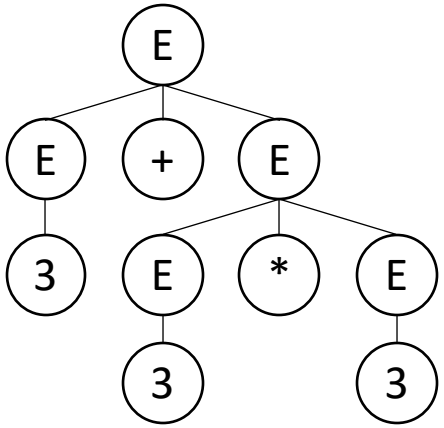


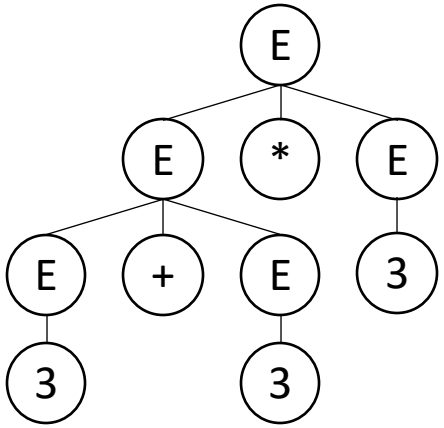
Ambiguity in Arithmetic Expressions

- Ambiguous grammar for arithmetic expressions with addition, multiplication and 3:

$$E \rightarrow E + E \mid E * E \mid 3$$

- It's not enough to just find any parse tree. We need to ensure we *unambiguously* find the correct one.



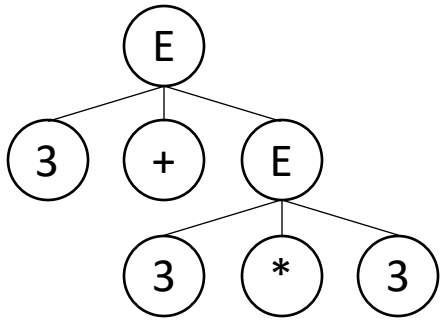


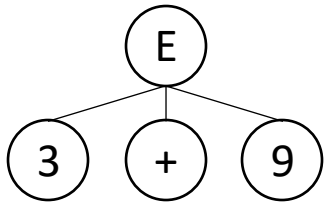
Resolving Ambiguity

- There is no general way to do this.
- In fact, there are context-free languages that are *inherently ambiguous*, i.e., all grammars for the language are ambiguous.
- However, there are tricks that work with practical grammars, e.g., grammars for programming languages.
- We will discuss specifically how to design a grammar for arithmetic expressions that enforces a specific order of operations.
- There are three aspects we will discuss: **precedence**, **associativity**, and **grouping**.

Operator Precedence

- **Precedence** refers to the order of priority for evaluating operations, e.g., multiplication comes before addition.
- When evaluating an expression using a tree, the operations that are *lowest* in the tree get evaluated first.





- We create *levels* in our grammar to enforce precedence.

Operator Precedence

- When evaluating an expression using a tree, the operations that are *lowest* in the tree get evaluated first.
- Make a list of operators in your language and group them into levels, where the *lowest numbered level* is the *highest precedence.*
- If the relative order can vary (e.g. multiplication and division), group the operators on the same level.
- Create one nonterminal per level, and allow the level to decrease. The lowest level can become a number or variable.
- Example: Instead of $E \rightarrow E + E \mid E * E \mid 3$,

$$L_2 \rightarrow L_2 + L_1 \mid L_1 \qquad L_1 \rightarrow L_1 * L_0 \mid L_0 \qquad L_0 \rightarrow 3$$

Operator Associativity

- **Associativity** determines how operators at the same precedence level are evaluated.
- An operator is called **associative** if the order of evaluation doesn't matter, e.g. 1 + 2 + 3 = (1 + 2) + 3 = 1 + (2 + 3).
- Addition and multiplication are associative, but subtraction and division are not:

$$(1 - 2) - 3 = -4 \text{ but } 1 - (2 - 3) = 2$$

- Subtraction and division are **left associative**, meaning we normally just evaluate them from left to right.
- Exponentiation is **right associative:** 2^{3^2} is 2^9 rather than 8^2.

Operator Associativity

- Operators at the same precedence level must either all be left associative or all right associative.
  - "Associative" operations like addition and multiplication count as both, since it doesn't matter whether we view the operation as left or right associative.
- Why? Suppose two operations with different associativity had the same precedence, e.g., subtraction and exponentiation.

$$2 - 3^2$$

  - Because subtraction is left associative, this should be $(2 - 3)^2 = 1$.
  - Because exponentiation is right associative, this should be $2 - 3^2 = -7$.
  - There are two valid answers and precedence doesn't let us choose one.

Operator Associativity

- Because levels of precedence must share a common associativity direction, associativity is enforced separately on each level.
- To enforce left associativity, the productions should be **left recursive:**

$$L_n \rightarrow L_n - L_{n-1} \mid L_n / L_{n-1} \mid L_{n-1}$$

- To enforce right associativity, they should be **right recursive:**

$$L_n \rightarrow L_{n-1} ** L_n \mid L_{n-1} = L_n \mid L_{n-1}$$

- The C/C++ assignment operator, which returns the assigned value, is another example of a right associative operator.
  - a = b = c = 4 evaluates as a = b = (c = 4) ⇒ a = (b = 4) ⇒ a = 4 ⇒ 4.

Operator Grouping

- It is important to be able to override precedence and associativity rules by **grouping** expressions with parentheses.
- 5 – (4 – 3 * (2 + 1)) forces the subtraction to be done right associatively, and the addition to be done before the multiplication.
- This is simple to handle. At the level of *terms* (numbers, variables, etc.) add the following production:

$$L_0 \rightarrow (L_{max})$$

- L_{max} means the highest-numbered level (lowest precedence).
- Parenthesized expressions appear lower in the tree than everything unparenthesized, and thus get evaluated first.

Example: Unambiguous Expression Grammar

$L_2 \rightarrow L_2 + L_1 \mid L_2 - L_1 \mid L_1$

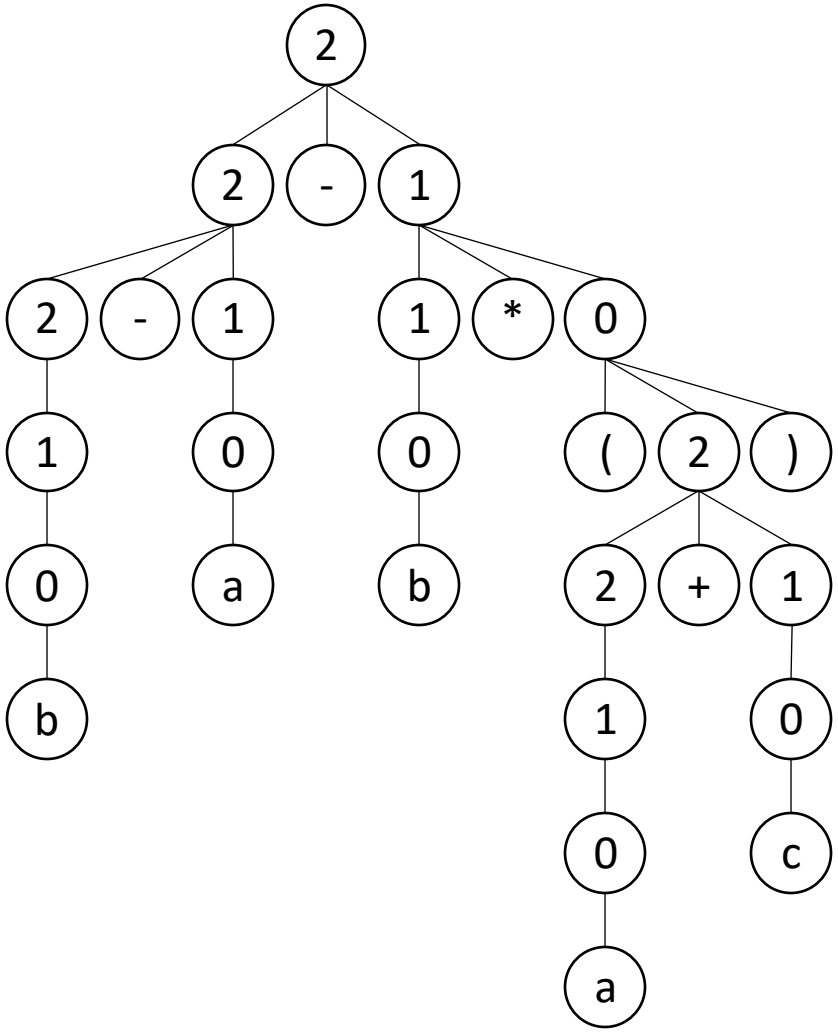
$L_1 \rightarrow L_1 * L_0 \mid L_1 / L_0 \mid L_0$

$L_0 \rightarrow a \mid b \mid c \mid (L_2)$

b – a – b * (a + c)

- Left associative operations
- * and / before + and –

(b – a) – (b * (a + c))



Example: Unambiguous Expression Grammar

$L_2 \rightarrow L_1 * L_2 \mid L_1 / L_2 \mid L_1$

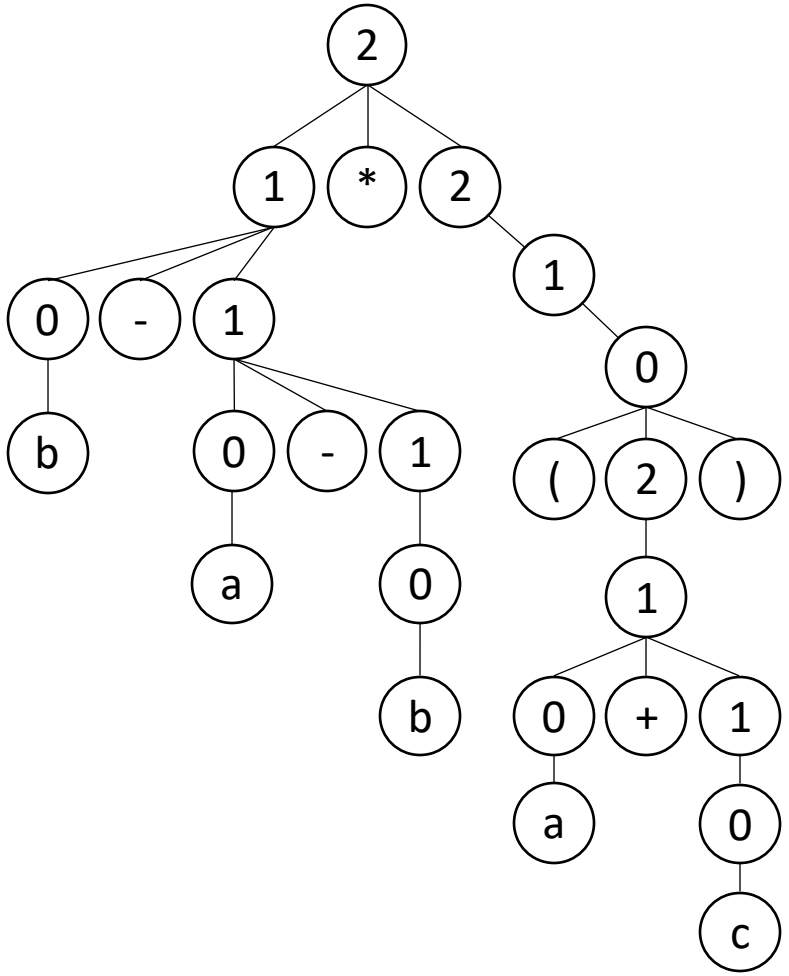
$L_1 \rightarrow L_0 + L_1 \mid L_0 - L_1 \mid L_0$

$L_0 \rightarrow a \mid b \mid c \mid (L_2)$

b – a – b * (a + c)

- Right associative operations
- + and – before * and /

(b – (a – b)) * (a + c)



"Practical" Example: WLP4 Grammar

```
expr → term
expr → expr PLUS term
expr → expr MINUS term

term → factor
term → term STAR factor
term → term SLASH factor
term → term PCT factor

factor → ID
factor → NUM
factor → NULL
factor → LPAREN expr RPAREN
factor → AMP lvalue
factor → STAR factor
factor → NEW INT LBRACK expr RBRACK
factor → ID LPAREN RPAREN
factor → ID LPAREN arglist RPAREN
```

Top-Down Parsing

Parsing Algorithms

- A **parsing algorithm** finds a derivation for a string with respect to a CFG, if one exists, and otherwise reports a parsing error.
- Since a derivation uniquely determines a parse tree, such an algorithm can also produce a parse tree by building a tree as each derivation step is found.
- Aside from the parser detecting syntax errors, obtaining a tree representation of a program is very useful for further analysis.
- Let's take a look at the outline of a simple parsing algorithm.

A Parsing Algorithm: Informal Pseudocode

- We will make the following assumptions:
  - We are working with an unambiguous grammar.
  - We want to produce a leftmost derivation.
  - We have a magic function called **Predict** which takes the string derived so far and magically predicts the next rule to use. This function returns "null" if the parse is impossible to complete no matter what rule we use.
  - The input string is "s" and the start symbol is "S".

```
derivedString = S
while derivedString != s
  if Predict(derivedString) returns a rule
    store the rule in the list of derivation steps
    apply the rule to the leftmost nonterminal in derivedString
  else ERROR
output the sequence of derivation steps
```

Top-Down Parsing

- This might seem like basically the only way to do parsing (with all the difficulty hidden inside the "Predict" function) but we actually made a significant decision: we are parsing from the **top down**.
- That is, we are starting from the start symbol and **expanding** it to obtain the input string, producing a derivation in the usual order.
- **Bottom-up parsing** starts from the input string and **reduces** it to the start symbol, producing a "reverse-order" derivation.
- First we will learn about **LL parsing**, a simple top-down parsing algorithm, and its limitations. We ultimately decide not to use a top-down approach.
- Then we will learn about **LR parsing**, a complicated bottom-up parsing algorithm, and the one we end up using in this course.

More Realistic Pseudocode

- The most obvious problem with our previous pseudocode is that we need to explain how to implement the magic Predict function.
- However, this line is also very vague:

```
apply the rule to the leftmost nonterminal in derivedString
```

- If we do this in the naïve way (string replacement) then every time we apply a rule, we need to do a search-and-replace.
- This is linear time in (length of derivedString) + (length of rule RHS).
- We can reduce this to linear time in just (length of rule RHS) with smart use of data structures.

Parsing with a Stack

- Instead of storing the "derivedString" (current derivation step) as a string, we will store it as a sequence of symbols on a stack.
- Furthermore, whenever there are terminal symbols on top of the stack, we will pop them and "match" them against the input string.
- After matching as many terminal symbols as possible, the top of the stack will be a nonterminal – the one we want to expand.
- We can then expand it by popping the nonterminal and pushing the right hand side of the rule we expand by (in reverse order).
- We maintain the following invariant: The current derivation step is (matched symbols) + (stack symbols).

Example: Top-Down Parsing with Stack

- S → AeB, A → ab | cd, B → f | gh, input string "abegh"

Unread Input	Matched Symbols	(Top) Stack (Bottom)	Action
abegh		S	

Example: Top-Down Parsing with Stack

- S → AeB, A → ab | cd, B → f | gh, input string "abegh"

Unread Input	Matched Symbols	(Top) Stack (Bottom)	Action
abegh		S	Apply S → AeB
abegh		AeB	

Example: Top-Down Parsing with Stack

- S → AeB, A → ab | cd, B → f | gh, input string "abegh"

Unread Input	Matched Symbols	(Top) Stack (Bottom)	Action
abegh		S	Apply S → AeB
abegh		AeB	Apply A → ab
abegh		abeB	

Example: Top-Down Parsing with Stack

- S → AeB, A → ab | cd, B → f | gh, input string "abegh"

Unread Input	Matched Symbols	(Top) Stack (Bottom)	Action
abegh		S	Apply S → AeB
abegh		AeB	Apply A → ab
abegh		abeB	Match a
begh	a	beB	

Example: Top-Down Parsing with Stack

- S → AeB, A → ab | cd, B → f | gh, input string "abegh"

Unread Input	Matched Symbols	(Top) Stack (Bottom)	Action
abegh		S	Apply S → AeB
abegh		AeB	Apply A → ab
abegh		abeB	Match a
begh	a	beB	Match b
egh	ab	eB	

Example: Top-Down Parsing with Stack

- S → AeB, A → ab | cd, B → f | gh, input string "abegh"

Unread Input	Matched Symbols	(Top) Stack (Bottom)	Action
abegh		S	Apply S → AeB
abegh		AeB	Apply A → ab
abegh		abeB	Match a
begh	a	beB	Match b
egh	ab	eB	Match e
gh	abe	B	

Example: Top-Down Parsing with Stack

- S → AeB, A → ab | cd, B → f | gh, input string "abegh"

Unread Input	Matched Symbols	(Top) Stack (Bottom)	Action
abegh		S	Apply S → AeB
abegh		AeB	Apply A → ab
abegh		abeB	Match a
begh	a	beB	Match b
egh	ab	eB	Match e
gh	abe	B	Apply B → gh
gh	abe	gh	

Example: Top-Down Parsing with Stack

- S → AeB, A → ab | cd, B → f | gh, input string "abegh"

Unread Input	Matched Symbols	(Top) Stack (Bottom)	Action
abegh		S	Apply S → AeB
abegh		AeB	Apply A → ab
abegh		abeB	Match a
begh	a	beB	Match b
egh	ab	eB	Match e
gh	abe	B	Apply B → gh
gh	abe	gh	Match g
h	abeg	h	

Example: Top-Down Parsing with Stack

- S → AeB, A → ab | cd, B → f | gh, input string "abegh"

Unread Input	Matched Symbols	(Top) Stack (Bottom)	Action
abegh		S	Apply S → AeB
abegh		AeB	Apply A → ab
abegh		abeB	Match a
begh	a	beB	Match b
egh	ab	eB	Match e
gh	abe	B	Apply B → gh
gh	abe	gh	Match g
h	abeg	h	Match h
	abegh		Accept