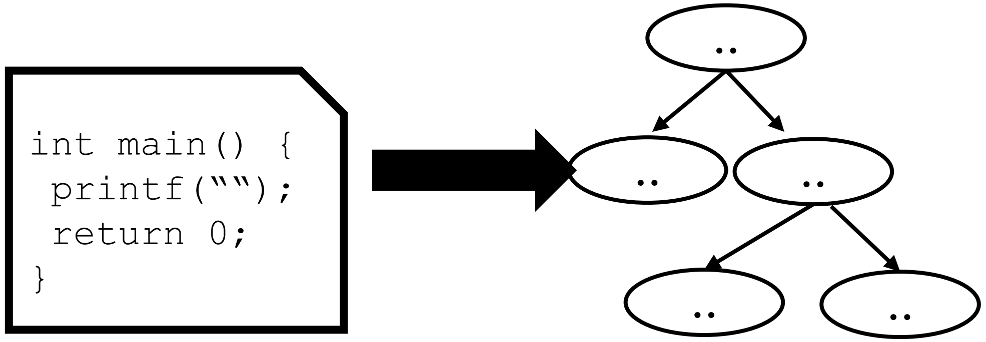
CSE110A: Compilers



Topics: **Final Review of Grammar**

- *Ambiguous Grammars and Precedence*
  Ambiguous Grammars and Associativity
- *Top-Down / Bottom-Up Parsers*

Ambiguous expressions

- First lets define tokens:
  - `NUM = ”[0-9]+”`
  - `PLUS = ‘\+’`
  - `TIMES = ’\*’`
  - `LP = ‘\(’`
  - `RP = \)’`

Lets define a simple expression language

```
Expr := NUM
      | Expr PLUS Expr
      | Expr TIMES Expr
      | LP Expr RP
```

Parse trees examples

input: 5

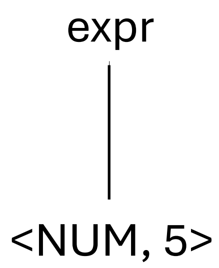
```
expr ::= NUM
      | expr PLUS expr
      | expr TIMES expr
      | LPAREN expr RPAREN
```

expr

Parse trees examples

input: 5

```
expr ::= NUM
     | expr PLUS expr
     | expr TIMES expr
     | LPAREN expr RPAREN
```



Parse trees examples

```
input: 5*6
```

```
expr ::= NUM
     | expr PLUS expr
     | expr TIMES expr
     | LPAREN expr RPAREN
```

Parse trees examples

input: 5*6

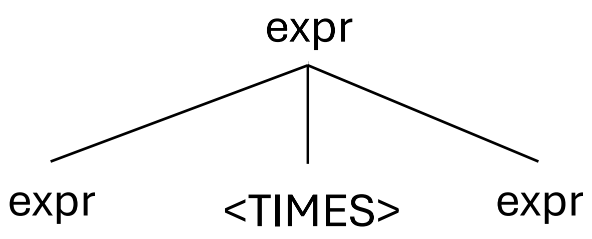
```
expr ::= NUM
     | expr PLUS expr
     | expr TIMES expr
     | LPAREN expr RPAREN
```

expr

Parse trees examples

```
expr ::= NUM
       | expr PLUS expr
       | expr TIMES expr
       | LPAREN expr RPAREN
```

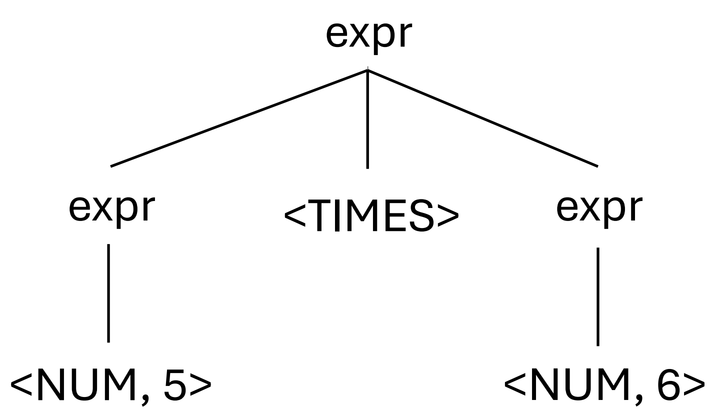
`input: 5*6`



Parse trees examples

```
expr ::= NUM
      | expr PLUS expr
      | expr TIMES expr
      | LPAREN expr RPAREN
```

```
input: 5*6
```



Parse trees examples

input: 5**6

What happens in an error?

```
expr ::= NUM
      | expr PLUS expr
      | expr TIMES expr
      | LPAREN expr RPAREN
```

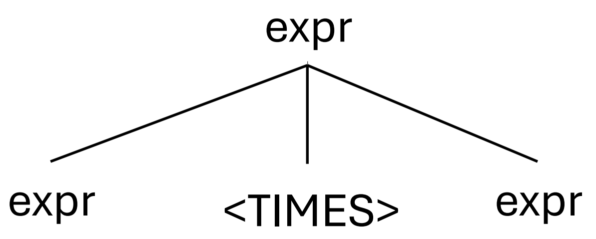
expr

Parse trees examples

```
expr ::= NUM
     | expr PLUS expr
     | expr TIMES expr
     | LPAREN expr RPAREN
```

`input: 5**6`

What happens in an error?

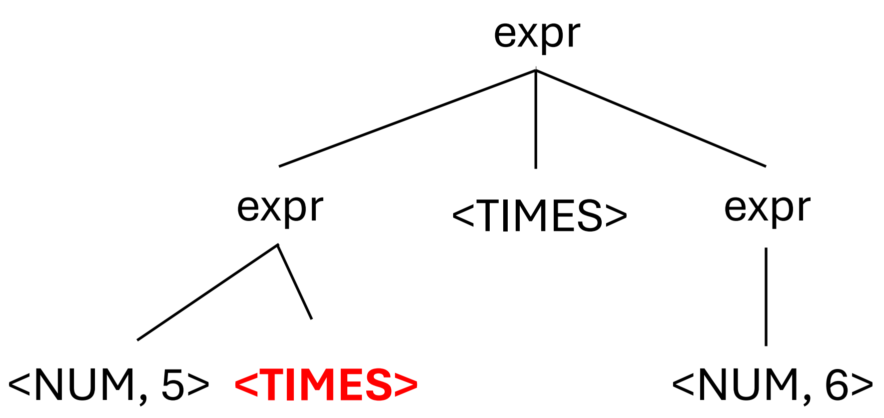


Parse trees examples

```
expr ::= NUM
      | expr PLUS expr
      | expr TIMES expr
      | LPAREN expr RPAREN
```

`input: 5**6`

What happens in an error?



Not possible!

Parse trees examples

```
input: (1+5)*6
```

```
expr ::= NUM
      | expr PLUS expr
      | expr TIMES expr
      | LPAREN expr RPAREN
```

Parse trees examples

```
input: (1+5)*6
```

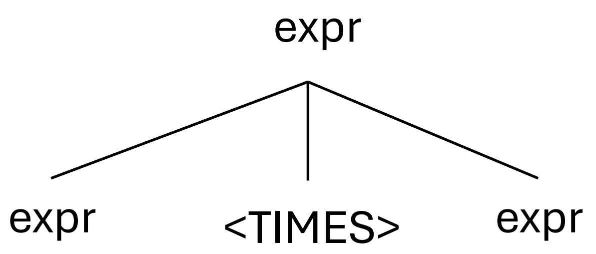
```
expr ::= NUM
     | expr PLUS expr
     | expr TIMES expr
     | LPAREN expr RPAREN
```

expr

Parse trees examples

```
expr ::= NUM
      | expr PLUS expr
      | expr TIMES expr
      | LPAREN expr RPAREN
```

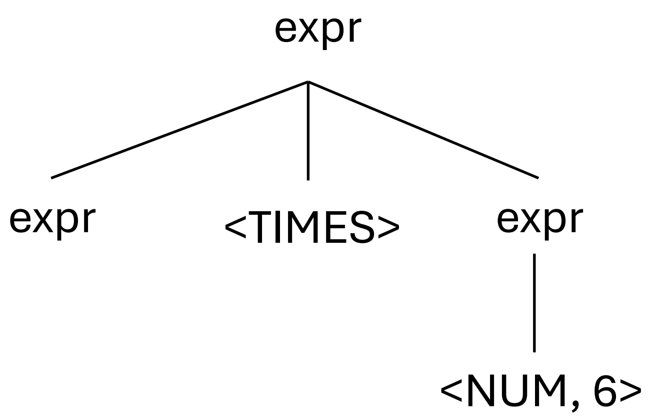
`input: (1+5)*6`



Parse trees examples

```
expr ::= NUM
      | expr PLUS expr
      | expr TIMES expr
      | LPAREN expr RPAREN
```

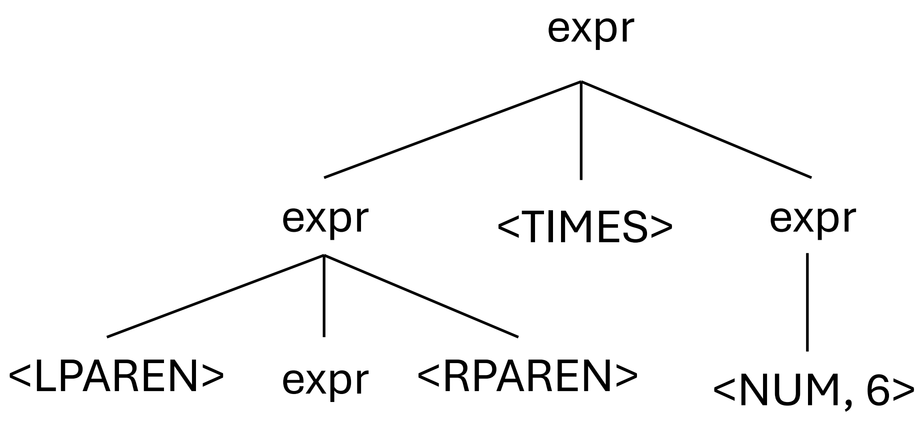
`input: (1+5)*6`



Parse trees examples

```
input: (1+5)*6
```

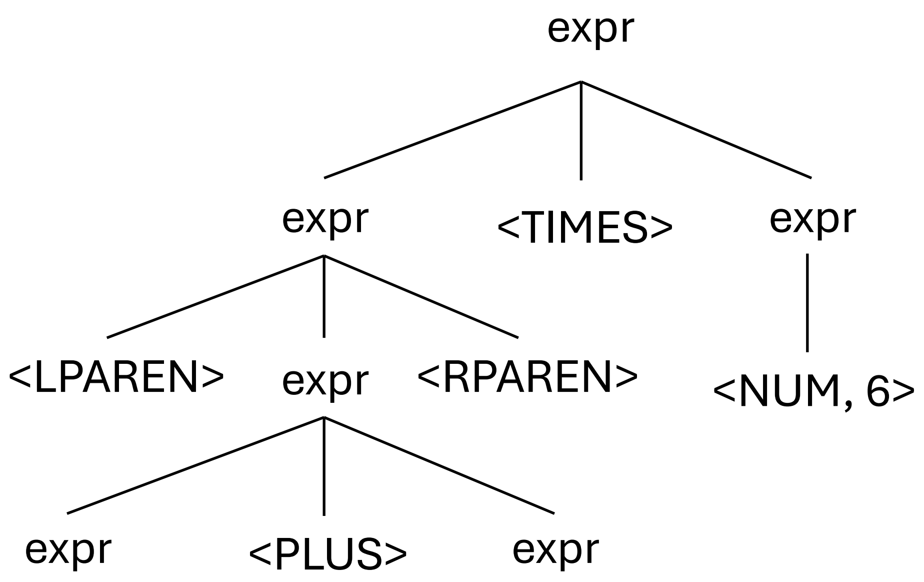
```
expr ::= NUM
      | expr PLUS expr
      | expr TIMES expr
      | LPAREN expr RPAREN
```



Parse trees examples

```
input: (1+5)*6
```

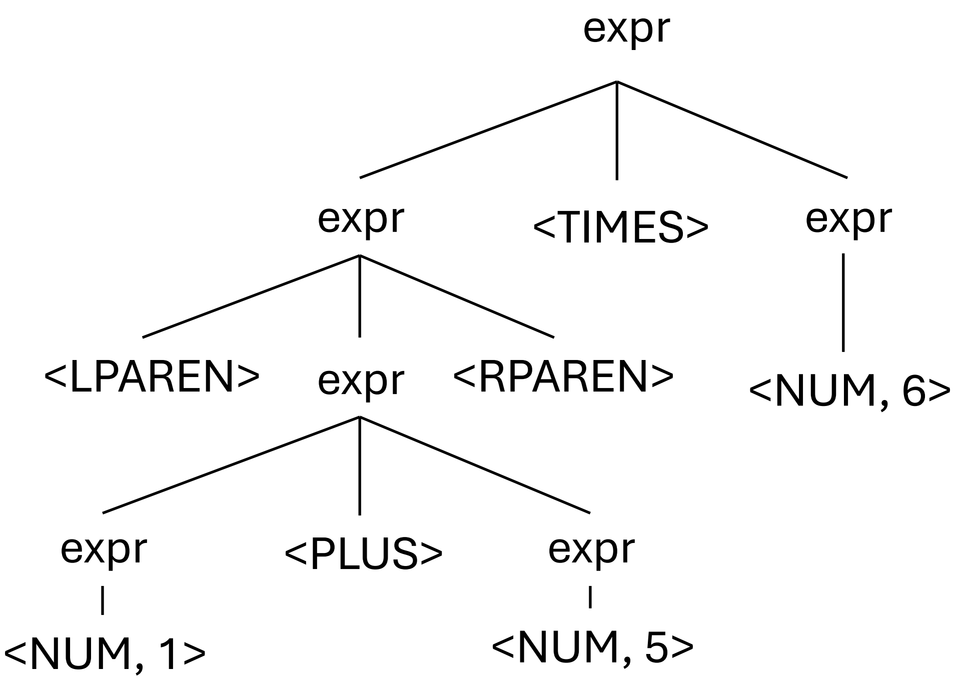
```
expr ::= NUM
      | expr PLUS expr
      | expr TIMES expr
      | LPAREN expr RPAREN
```



Parse trees examples

```
expr ::= NUM
     | expr PLUS expr
     | expr TIMES expr
     | LPAREN expr RPAREN
```

`input: (1+5)*6`

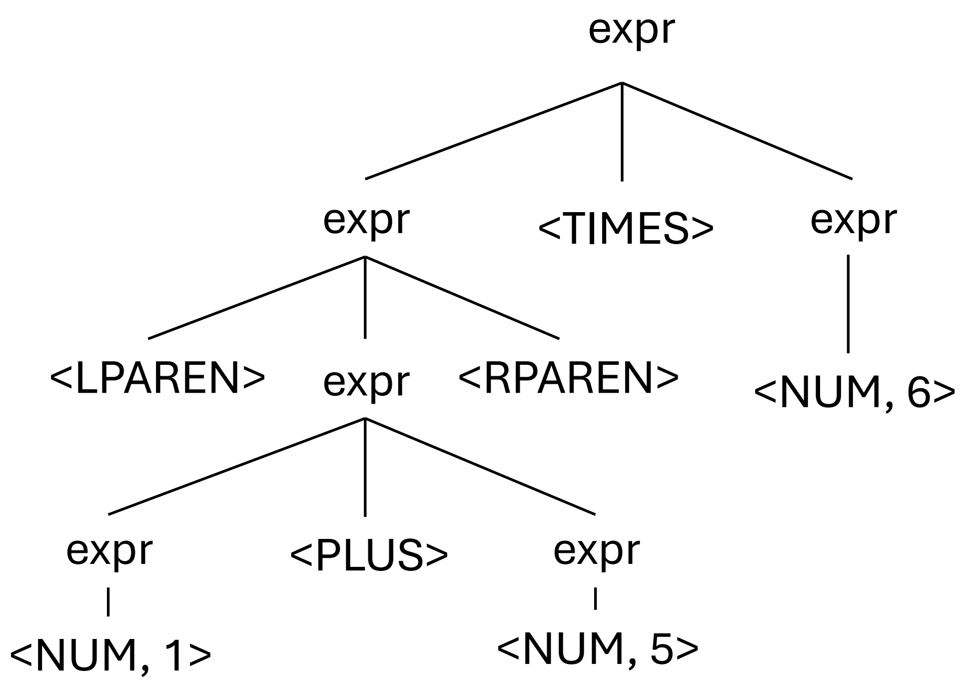


Parse trees examples

Does this parse tree capture the structure we want?

`input: (1+5)*6`

```
expr ::= NUM
     | expr PLUS expr
     | expr TIMES expr
     | LPAREN expr RPAREN
```



Parse trees

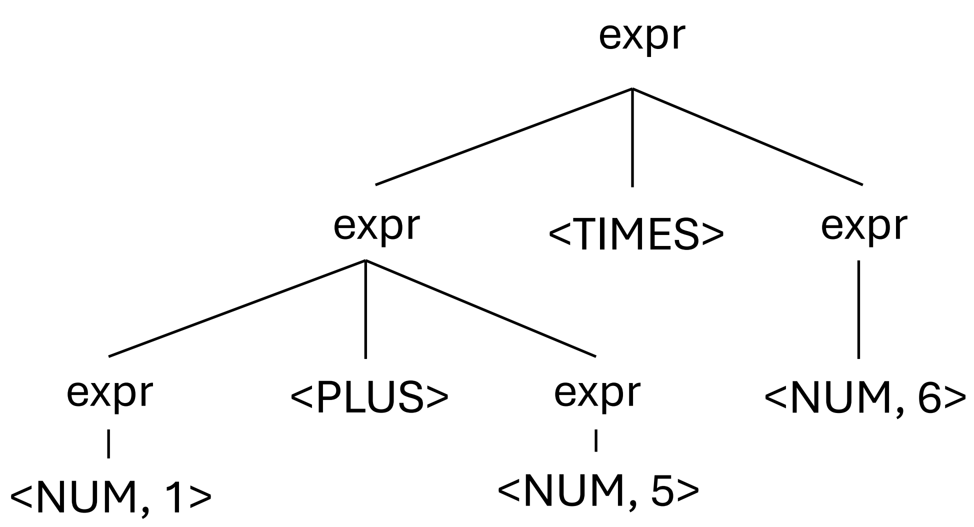
- How about : `1 + 5 * 6`

```
expr ::= NUM
     | expr PLUS expr
     | expr TIMES expr
     | LPAREN expr RPAREN
```

Parse trees

- How about : `1 + 5 * 6`

```
expr ::= NUM
     | expr PLUS expr
     | expr TIMES expr
     | LPAREN expr RPAREN
```



AMBIGUOUS GRAMMARS AND PRECEDENCE IN EXPRESSIONS

Ambiguous Precedence of Two Operators

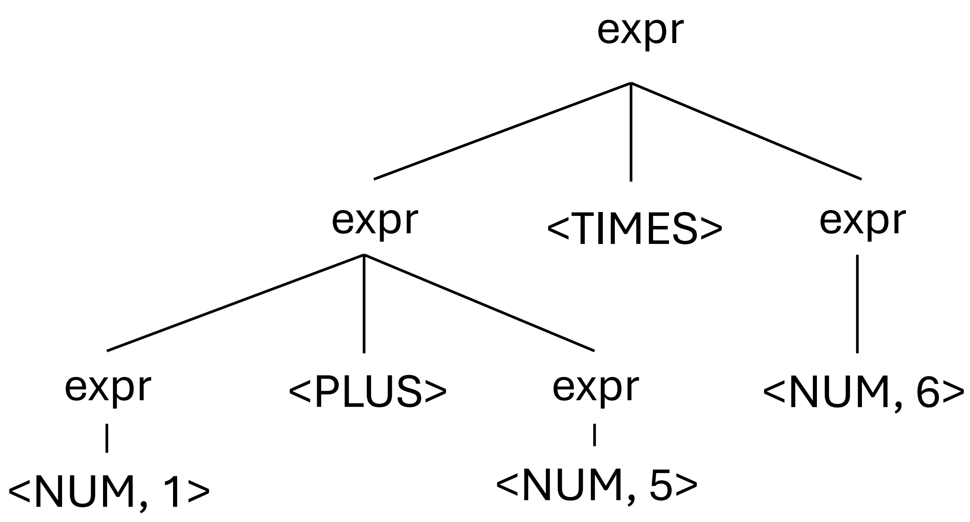
```
expr ::= NUM
      | expr PLUS expr
      | expr TIMES expr
      | LPAREN expr RPAREN
```

Operator	Name	Productions
+, *	expr	: expr PLUS expr | expr TIMES expr | LPAREN expr RPAREN

Ambiguous Grammars

- `input: 1 + 5 * 6`

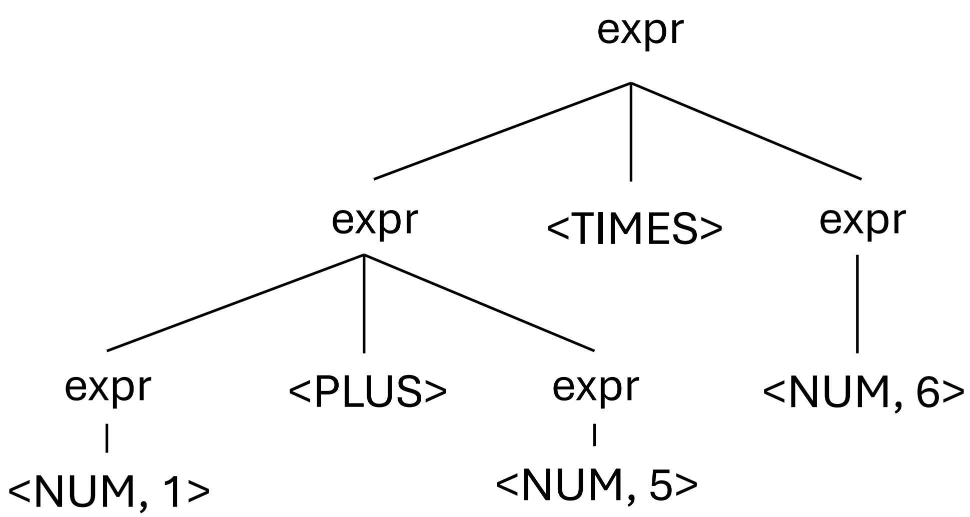
```
expr ::= NUM
      | expr PLUS expr
      | expr TIMES expr
      | LPAREN expr RPAREN
```

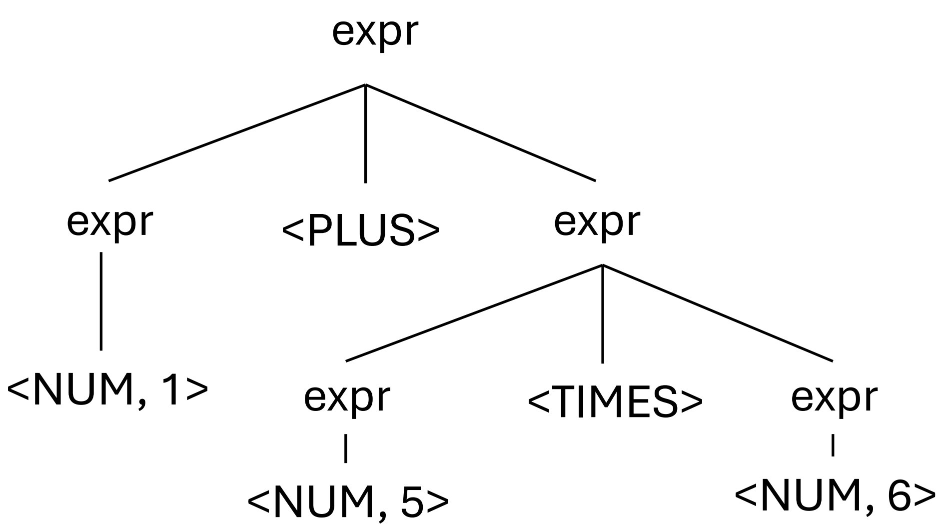


Ambiguous Grammars

- `input: 1 + 5 * 6`

```
expr ::= NUM
       | expr PLUS expr
       | expr TIMES expr
       | LPAREN expr RPAREN
```





Avoiding Ambiguity

- How to avoid ambiguity related to precedence?

- **One Way:** Define precedence into the grammar:
  - Ambiguity comes from conflicts. Explicitly define how to deal with conflicts by explicitly indicating that:

    * has higher precedence than +

- Some parser generators support this,
  - e.g. YACC(C), Bison (C), Antlr (Java), PLY(Python)

Avoiding Precedence Ambiguity

- How to avoid ambiguity related to precedence?

- **Second way**: add new production rules
  - **One non-terminal for each level of precedence**
  - lowest precedence at the top
  - highest precedence at the bottom

- Lets try with expressions and the following:

  `+ * ()`

Avoiding Precedence Ambiguity

For the second way: use new production rules

- **One non-terminal for each level of precedence**
- lowest precedence at the top
- **highest precedence at the bottom**

Operator	Name	Productions
+	`expr`	`: expr PLUS expr` `| term`
*	`term`	`: term TIMES term` `| factor`
()	`factor`	`: LPAREN expr RPAREN` `| NUM`

Precedence increases going down

Now lets create a parse tree

`input: 1+5*6`

Operator	Name	Productions
+	`expr`	`: expr PLUS expr` `| term`
*	`term`	`: term TIMES term` `| factor`
()	`factor`	`: LPAREN expr RPAREN` `| NUM`

Now lets create a parse tree

`input: 1+5*6`

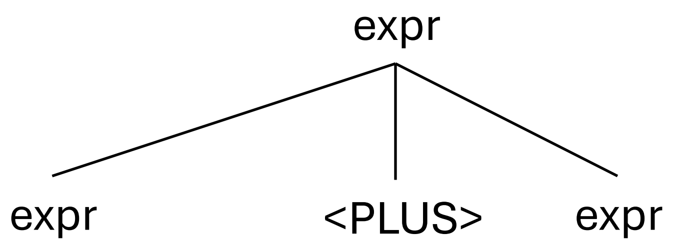
expr

Operator	Name	Productions
+	`expr`	`: expr PLUS expr` `| term`
*	`term`	`: term TIMES term` `| factor`
()	`factor`	`: LPAREN expr RPAREN` `| NUM`

Now lets create a parse tree

`input: 1+5*6`

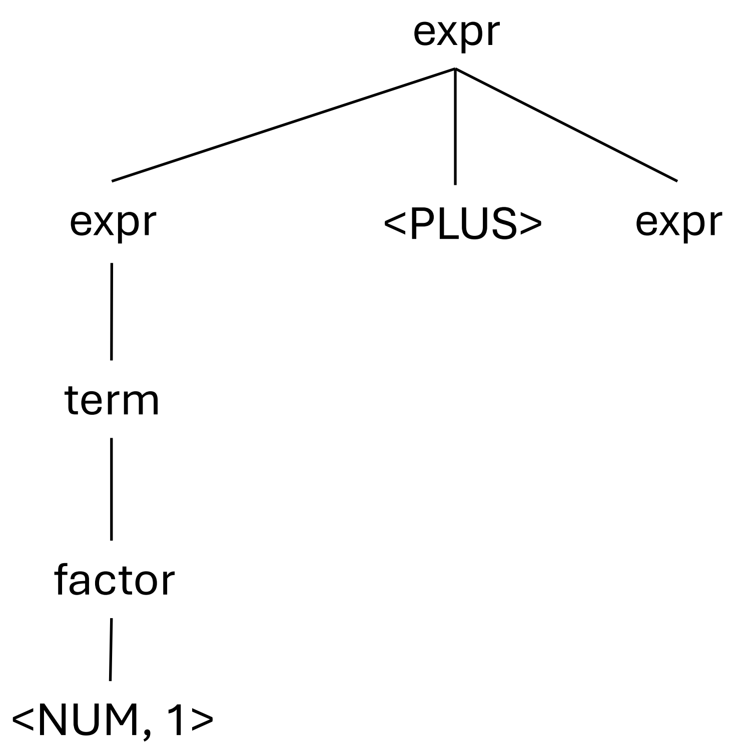
Operator	Name	Productions
+	`expr`	`: expr PLUS expr` `| term`
*	`term`	`: term TIMES term` `| factor`
()	`factor`	`: LPAREN expr RPAREN` `| NUM`



Now lets create a parse tree

`input: 1+5*6`

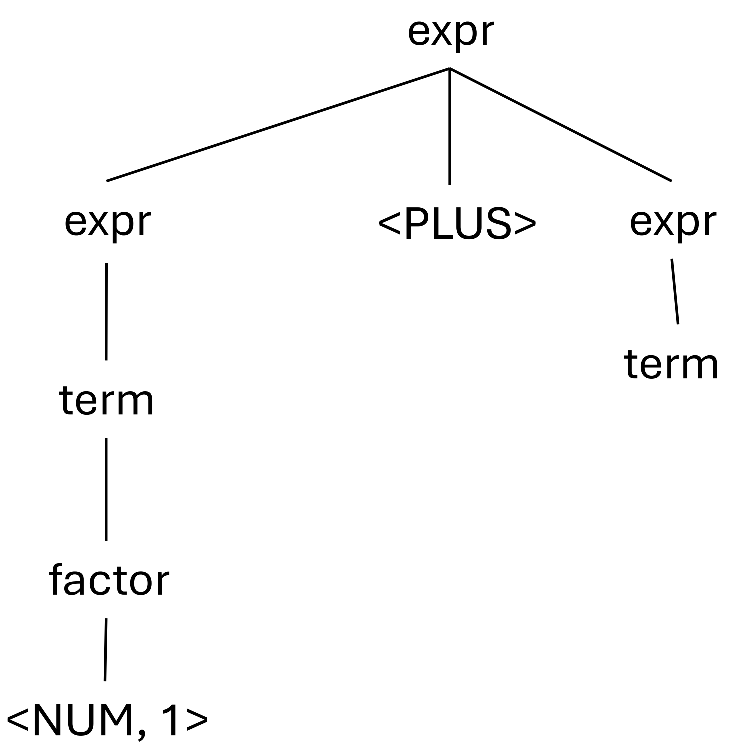
Operator	Name	Productions
+	`expr`	`: expr PLUS expr` `| term`
*	`term`	`: term TIMES term` `| factor`
()	`factor`	`: LPAREN expr RPAREN` `| NUM`



Now lets create a parse tree

`input: 1+5*6`

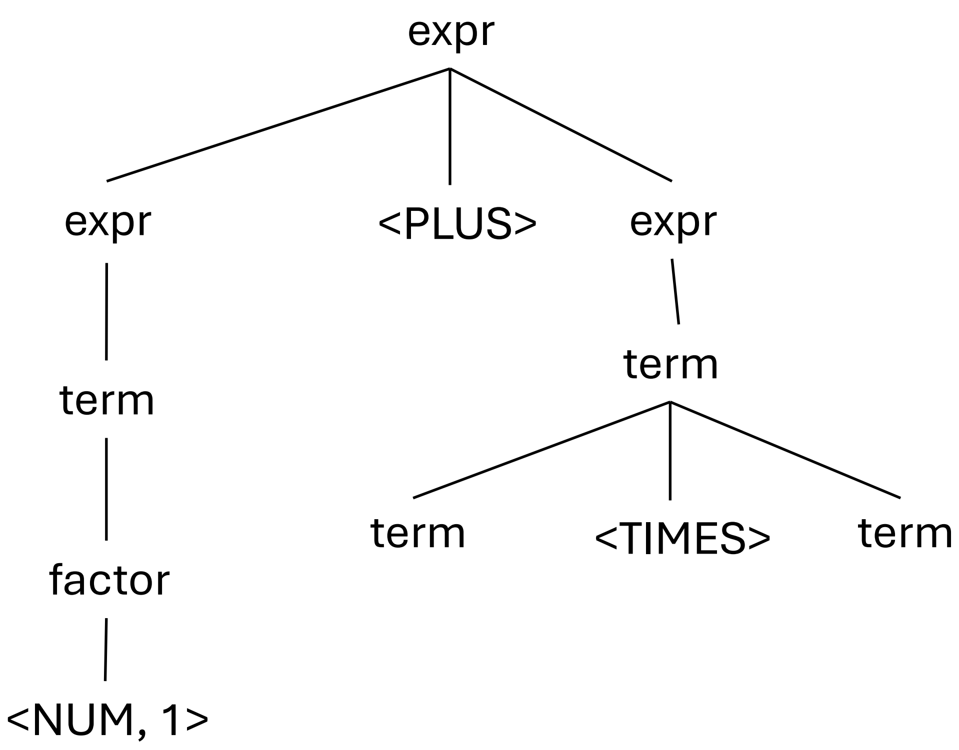
Operator	Name	Productions
+	`expr`	`: expr PLUS expr` `| term`
*	`term`	`: term TIMES term` `| factor`
()	`factor`	`: LPAREN expr RPAREN` `| NUM`



Now lets create a parse tree

`input: 1+5*6`

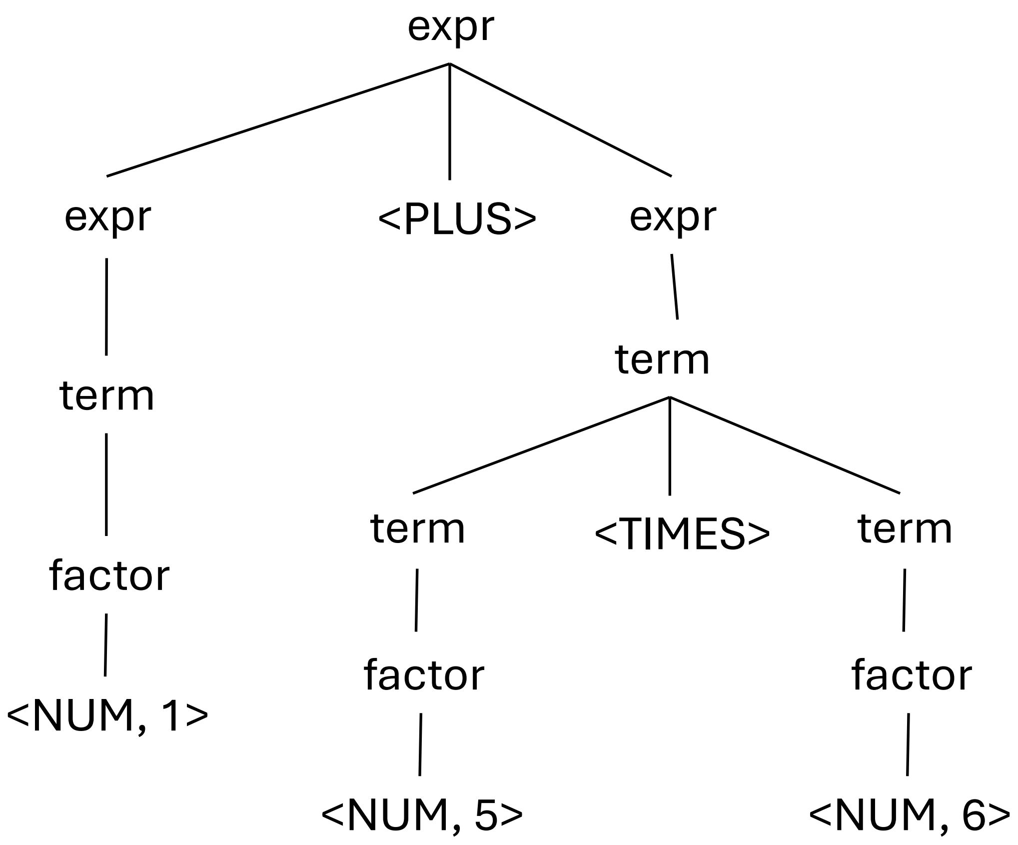
Operator	Name	Productions
+	`expr`	`: expr PLUS expr` `| term`
*	`term`	`: term TIMES term` `| factor`
()	`factor`	`: LPAREN expr RPAREN` `| NUM`



Now lets create a parse tree

`input: 1+5*6`

Operator	Name	Productions
+	`expr`	`: expr PLUS expr` `| term`
*	`term`	`: term TIMES term` `| factor`
()	`factor`	`: LPAREN expr RPAREN` `| NUM`



Fixing Grammar for Associativity

Let’s make some more parse trees

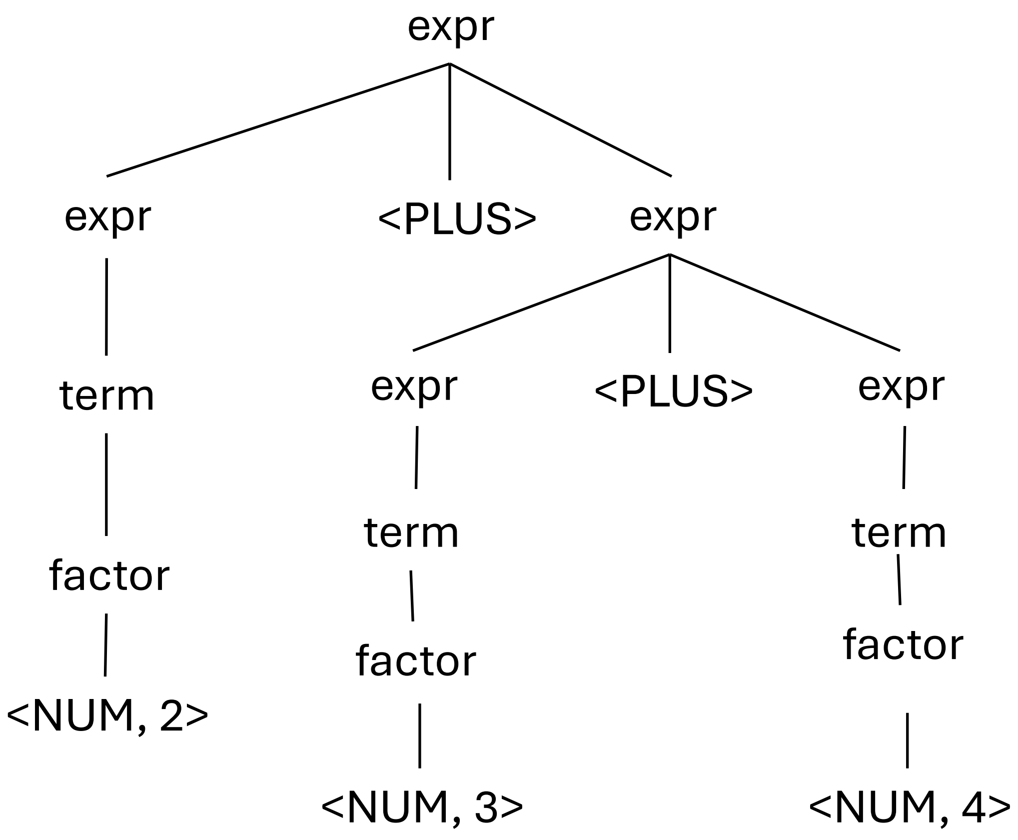
input: 2+3+4

Operator	Name	Productions
+	expr	: expr PLUS expr | term
*	term	: term TIMES term | factor
()	factor	: LP expr RP | NUM

Let’s make some more parse trees

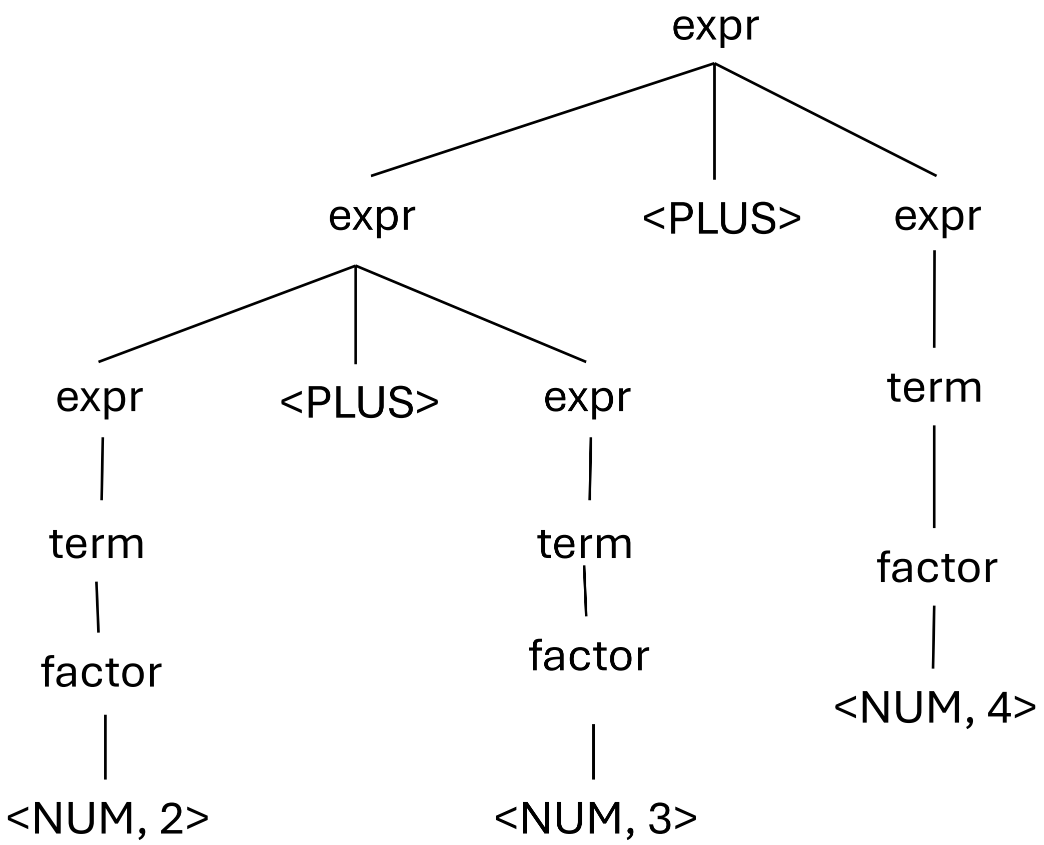
Operator	Name	Productions
+	`expr`	`: expr PLUS expr` `| term`
*	`term`	`: term TIMES term` `| factor`
()	`factor`	`: LP expr RP` `| NUM`

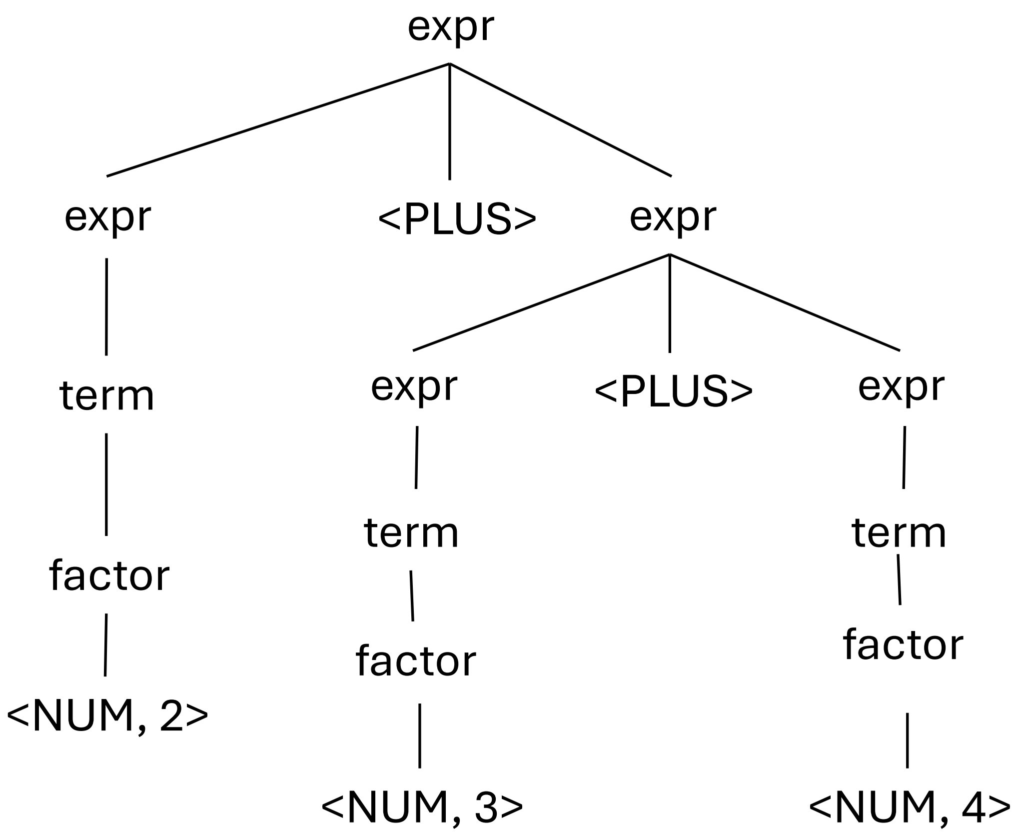
`input: 2+3+4`



This is ambiguous, is it an issue?

`input: 2+3+4`



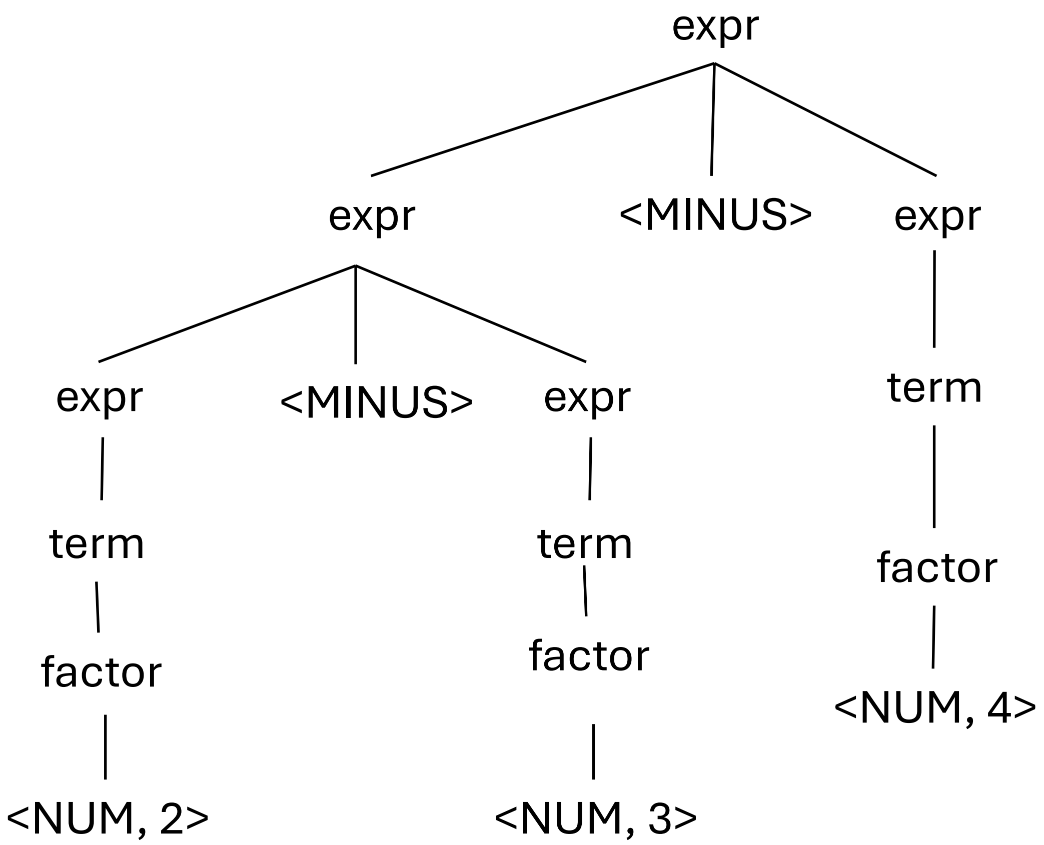


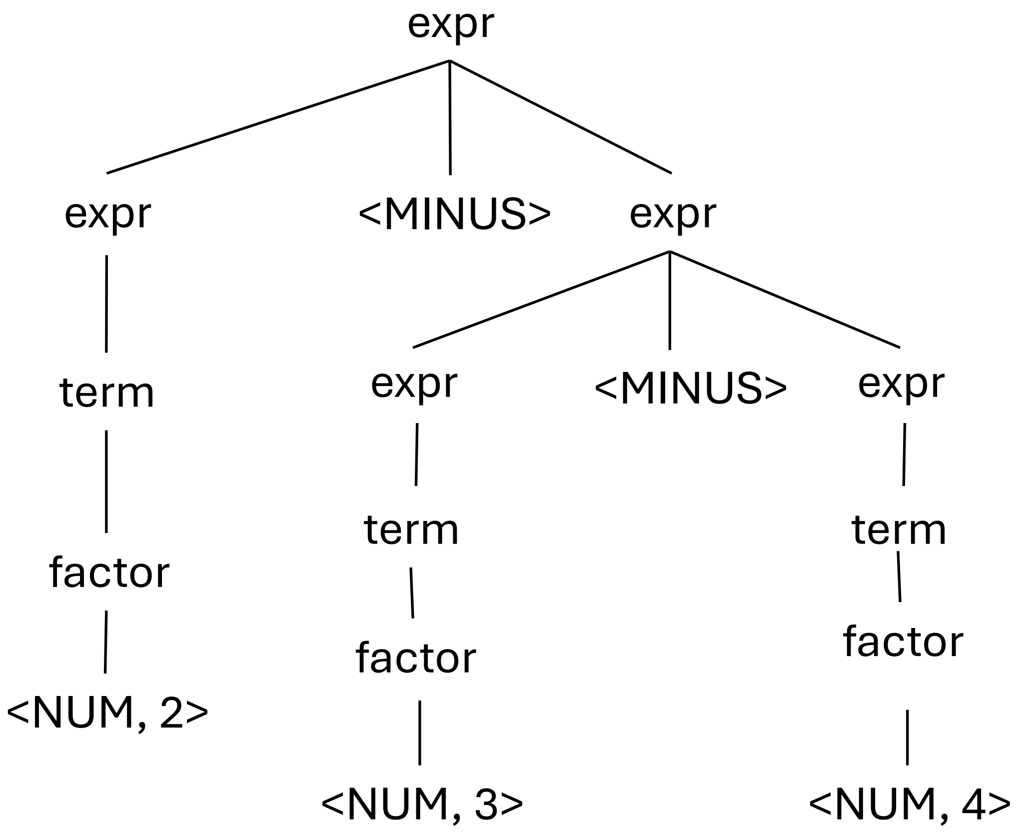
What about for a different operator?

```
input: 2-3-4
```

What about for a different operator?

`input: 2-3-4`



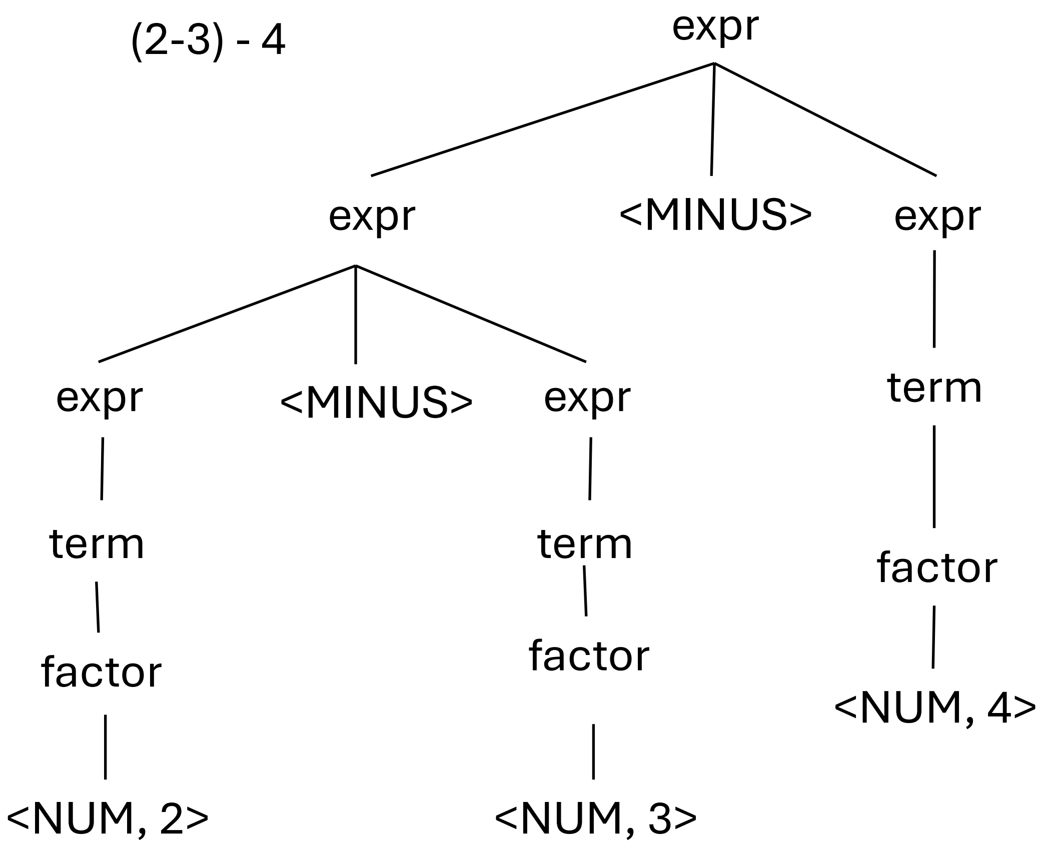


Which one is right?

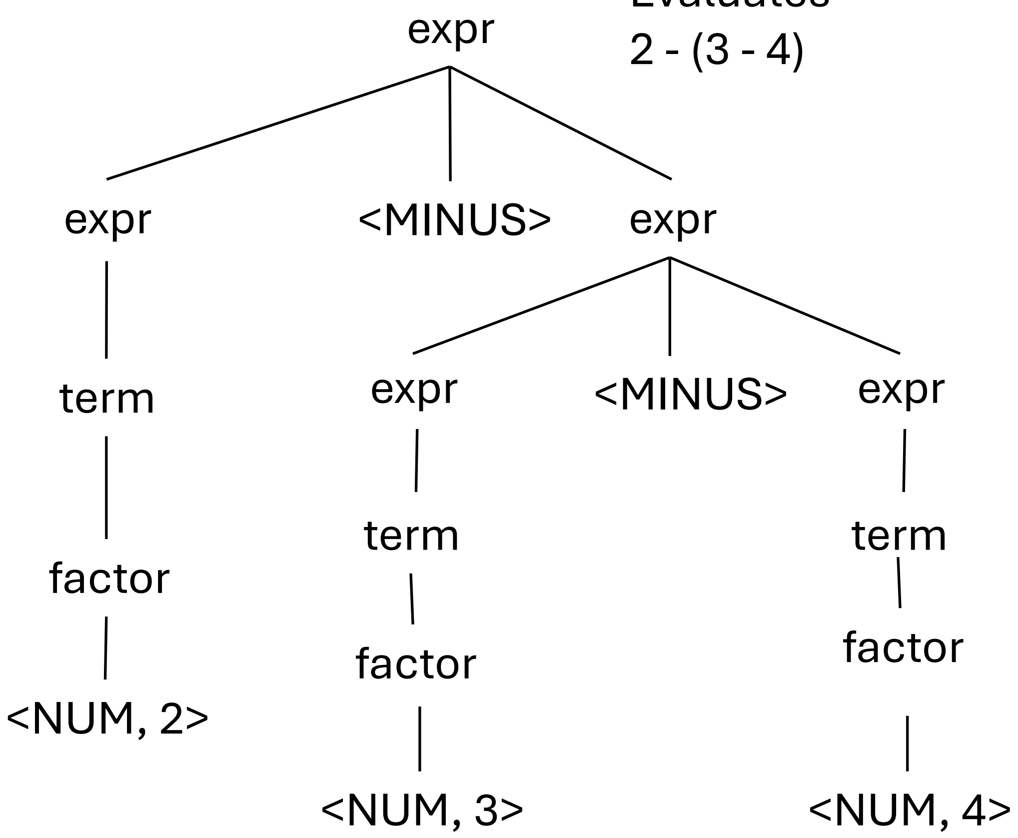
What about for a different operator?

`input: 2-3-4`

Evaluates
(2-3) - 4



Evaluates
2 - (3 - 4)



Which one is right?

Associativity

If an operator is not considered associative then we define

- left to right (left-associative)
  - `2-3-4` is evaluated as ((2-3) - 4)
  - What other operators are left-associative
- right-to-left (right-associative)
  - Any operators you can think of?

Associativity

If an operator is not associative then we define

- left to right (left-associative)
  - `2-3-4` is evaluated as ((2-3) - 4)
  - What other operators are left-associative
- right-to-left (right-associative)
  - Assignment, power operator

How to encode associativity?

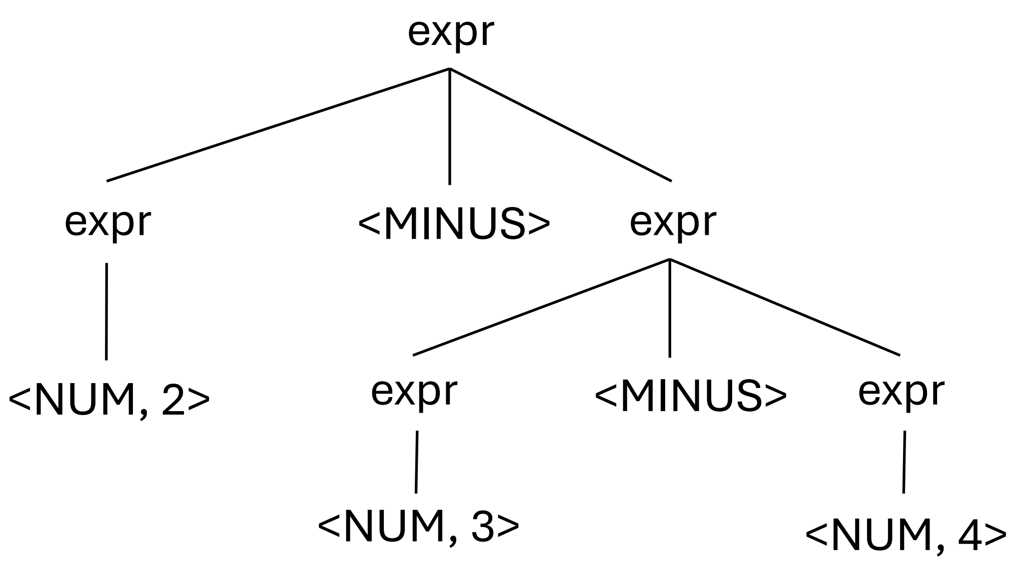
- Like precedence, some tools (e.g. YACC/Bison) allow associativity specification through keywords:
  - "+": left, "^": right

- Also like precedence, **we can also encode it into the production rules**

Ambiguous Associativity for a single operator

Operator	Name	Productions
-	`expr`	`: expr MINUS expr` `| NUM`

`input: 2-3-4`

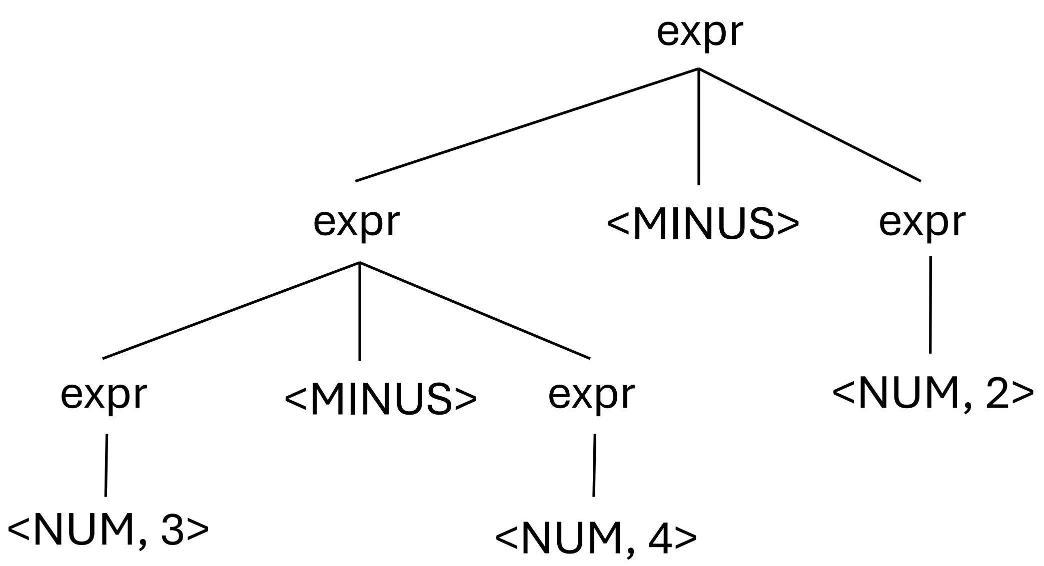


We want to disallow this parse tree

Ambiguous Associativity for a single operator

Operator	Name	Productions
-	`expr`	`: expr MINUS expr` `| NUM`

`input: 2-3-4`



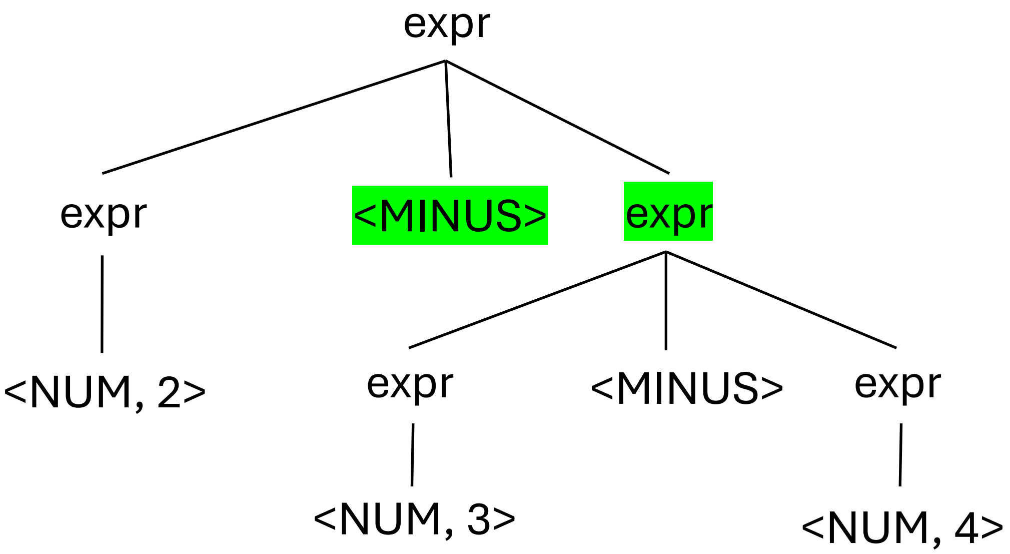
We want this one

Left associativity for a single operator

<table>
<tr><th>Operator</th><th>Name</th><th>Productions</th></tr>
<tr><td>-</td><td><code>expr</code></td><td><code>: expr MINUS NUM</code>
<code>| NUM</code></td></tr>
</table>

Left recursion leads to left-associative (not ideal for top-down parsing) but works for bottom-up parsing.

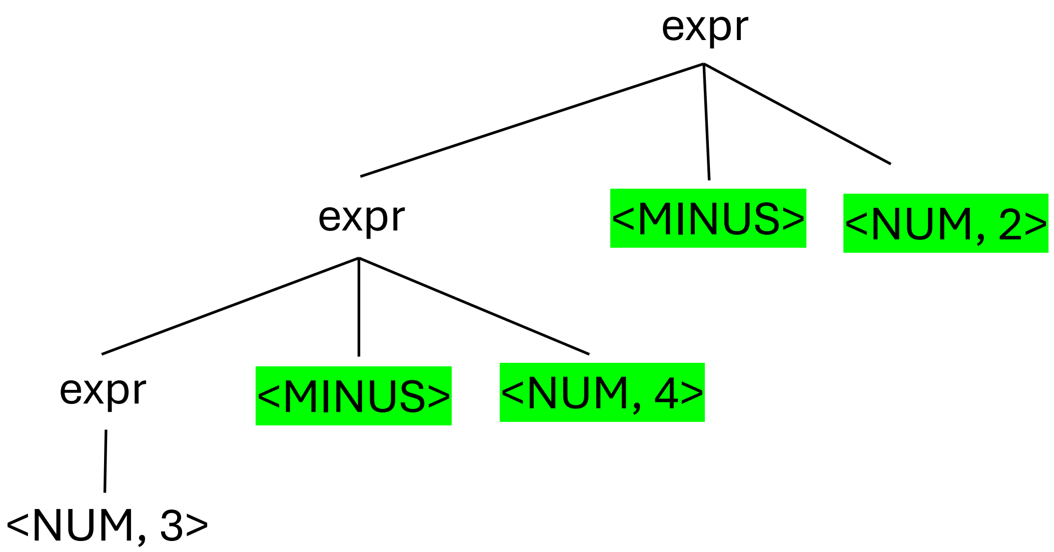
`input: 2-3-4`



No longer allowed

Left associativity for a single operator

`input: 2-3-4`



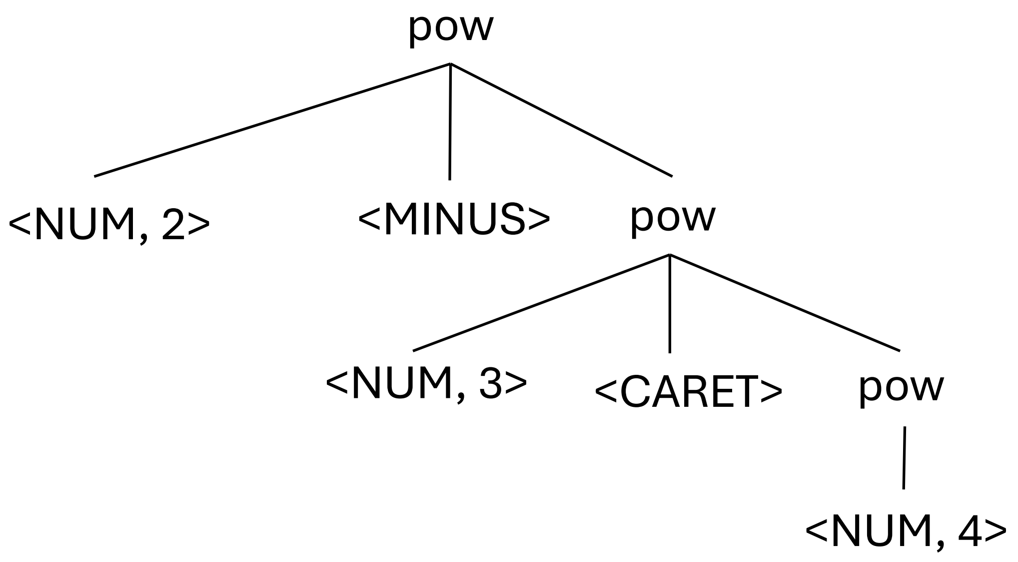
Operator	Name	Productions
-	`expr`	`: expr MINUS NUM` `| NUM`

Left recursion leads to left-associative (not ideal for top-down parsing) but works for bottom-up parsing.

Valid operations with single operator occur from left to right or "left-associative"

Right associativity for a single operator

`input: 2^3^4`



<table>
<tr><th>Operator</th><th>Name</th><th>Productions</th></tr>
<tr><td>-</td><td><code>pow</code></td><td><code>: NUM CARET pow</code>
<code>| NUM</code></td></tr>
</table>

Right recursion leads to right-associative

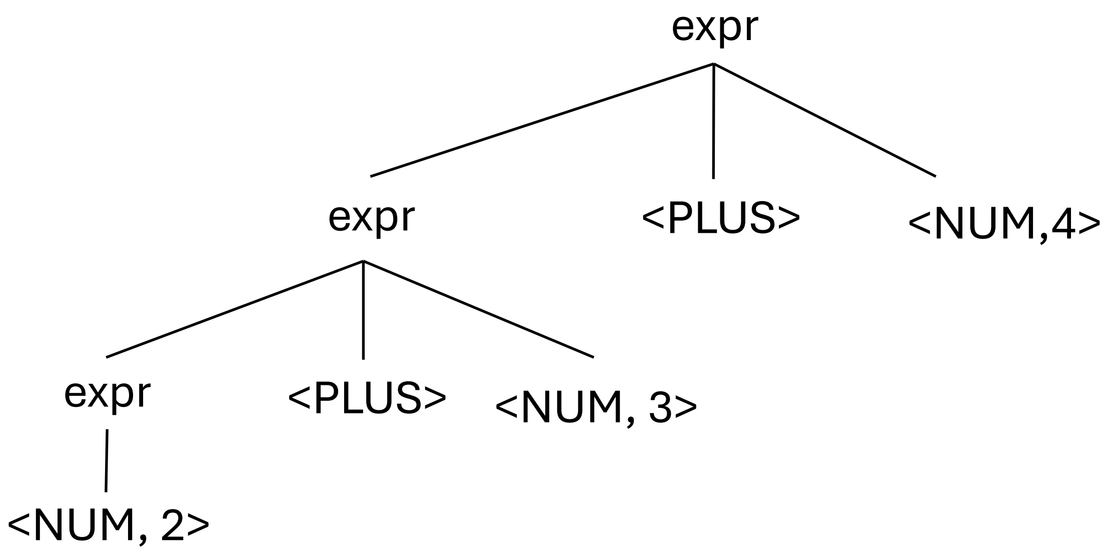
Raising to a power operations should be right-associative, the operation on right performs first.

Should you have associativity when its not required?

Benefits?
Drawbacks?

Operator	Name	Productions
+	`expr`	`: expr PLUS expr` `| NUM`

`input: 2+3+4`

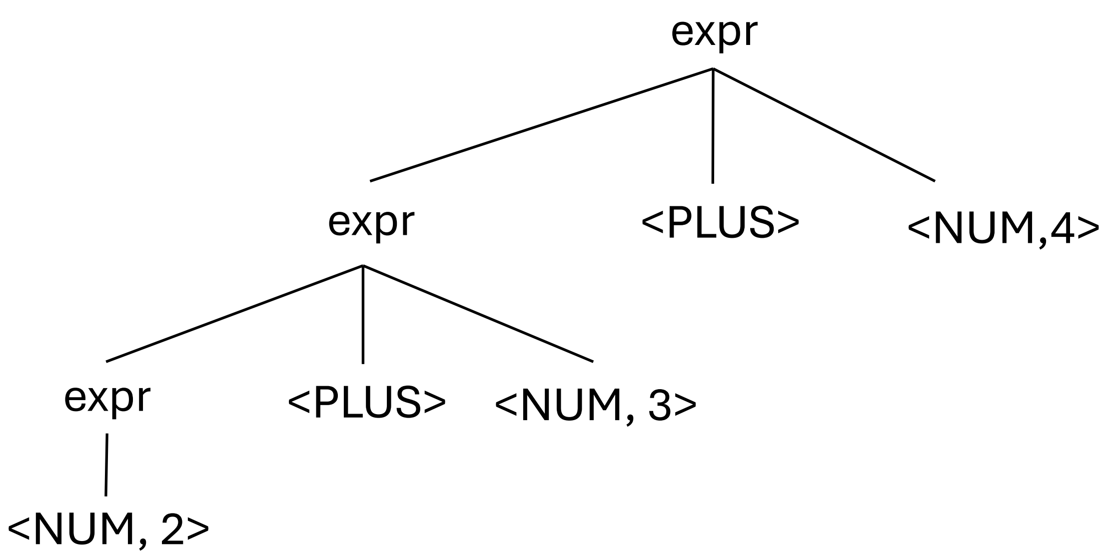


Should you have associativity when its not required?

Benefits?
Drawbacks?

Operator	Name	Productions
+	`expr`	`: expr PLUS NUM` `| NUM`

`input: 2+3+4`



Good design principle to avoid ambiguous grammars, even when strictly not required too.

Helps with debugging, etc. etc.

Many tools will warn if it detects ambiguity

Let’s make a richer expression grammar

Let’s do operators `[+,*,-,/,^]` *and* `()`

Operator	Name	Productions

```
Tokens:
    NUM    = “[0-9]+”
    PLUS   = ‘\+’
    TIMES  = ’\*’
    LP     = ‘\(’
    RP     = \)’
    MINUS  = ‘-’
    DIV    = ‘/’
    CARROT =’ \^’
```

Let’s make a richer expression grammar

Let’s do operators `[+,*,-,/,^]` *and* `()`

Operator	**Name**	**Productions**

+,-	`expr`	`: expr PLUS term`
		`| expr MINUS term`
		`| term`

*,/	`term`	`: term TIMES pow`
		`| term DIV pow`
		`| pow`

^	`pow`	`: factor CARROT pow` `| factor`
()	`factor`	`: LPAR expr RPAR` `| NUM`

```
Tokens:
    NUM    = “[0-9]+”
    PLUS   = ‘\+’
    TIMES  = ’\*’
    LP     = ‘\(’
    RP     = \)’
    MINUS  = ‘-’
    DIV    = ‘/’
    CARROT =’ \^’
```

What associativity do operators in C have?

- https://en.cppreference.com/w/c/language/operator_precedence

Algorithms for Parsing

One goal:

- Given a string *s* and a CFG *G*, determine if *G* can derive *s*

- We will do that by implicitly attempting to derive a parse tree for *S*

- Two different approaches, each with different trade-offs:
  - Top down
  - Bottom up

Top-down parsing

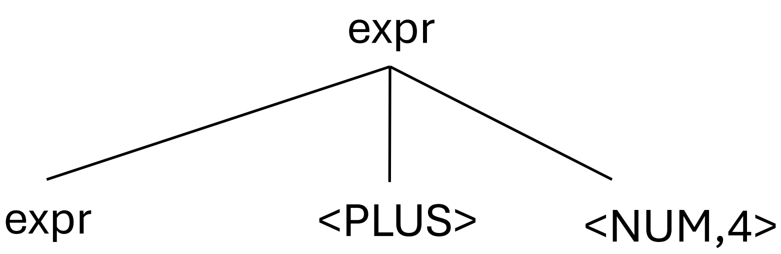
`input: 2+3+4`

expr

Operator	Name	Productions
+	`expr`	`: expr PLUS NUM` `| NUM`

Top-down parsing

`input: 2+3+4`

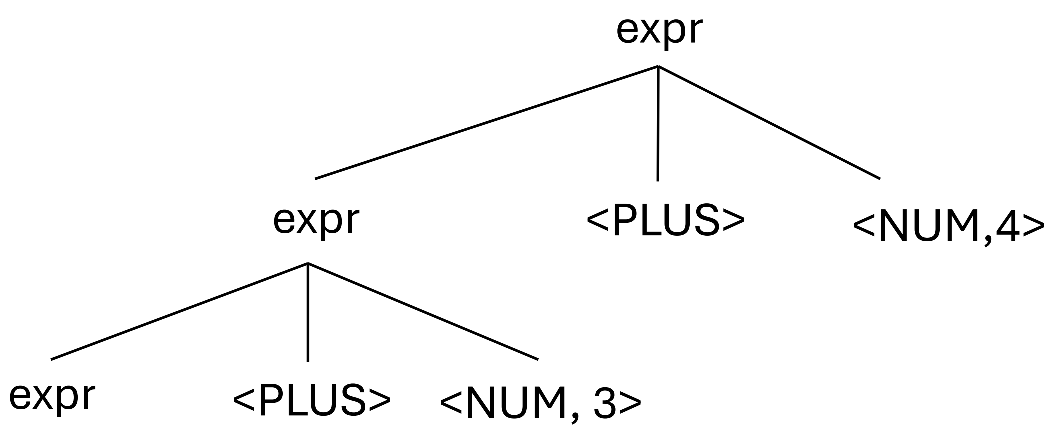


Operator	Name	Productions
+	`expr`	`: expr PLUS NUM` `| NUM`

Top-down parsing

`input: 2+3+4`

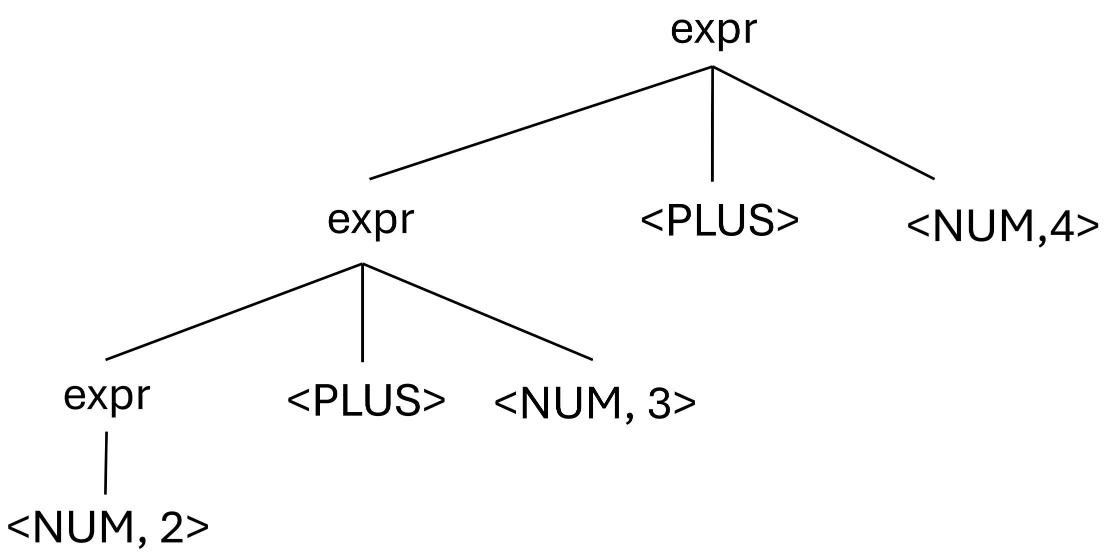
<table>
<tr><th>Operator</th><th>Name</th><th>Productions</th></tr>
<tr><td>+</td><td><code>expr</code></td><td><code>: expr PLUS NUM</code>
<code>| NUM</code></td></tr>
</table>



Top-down parsing

Operator	Name	Productions
+	`expr`	`: expr PLUS NUM` `| NUM`

`input: 2+3+4`



Top-down parsing

Pros:

- Algorithm is simpler
- Faster than bottom-up
- Easier recovery

Cons:

- Not efficient on arbitrary grammars
- Many grammars need to be re-written

Bottom-up parsing

`input: 2+3+4`

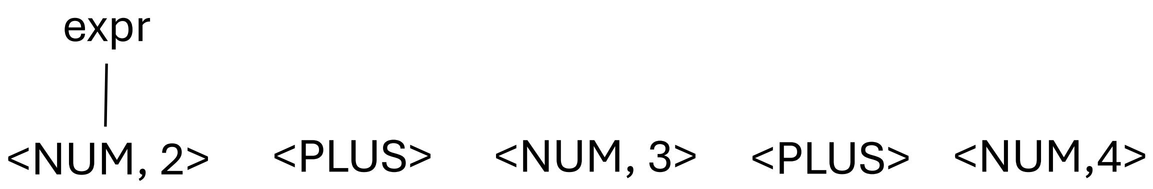
Operator	Name	Productions
+	`expr`	`: expr PLUS NUM` `| NUM`

<NUM, 2> <PLUS> <NUM, 3> <PLUS> <NUM,4>

Bottom-up parsing

`input: 2+3+4`

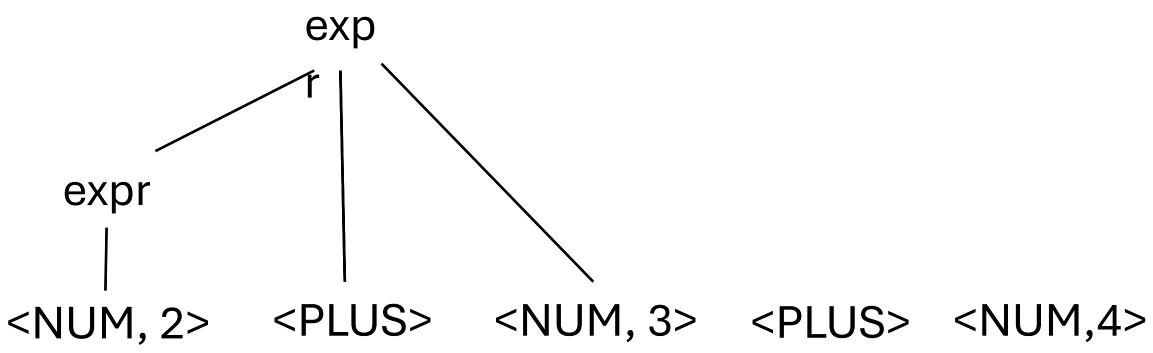
Operator	Name	Productions
+	`expr`	`: expr PLUS NUM` `| NUM`



Bottom-up parsing

`input: 2+3+4`

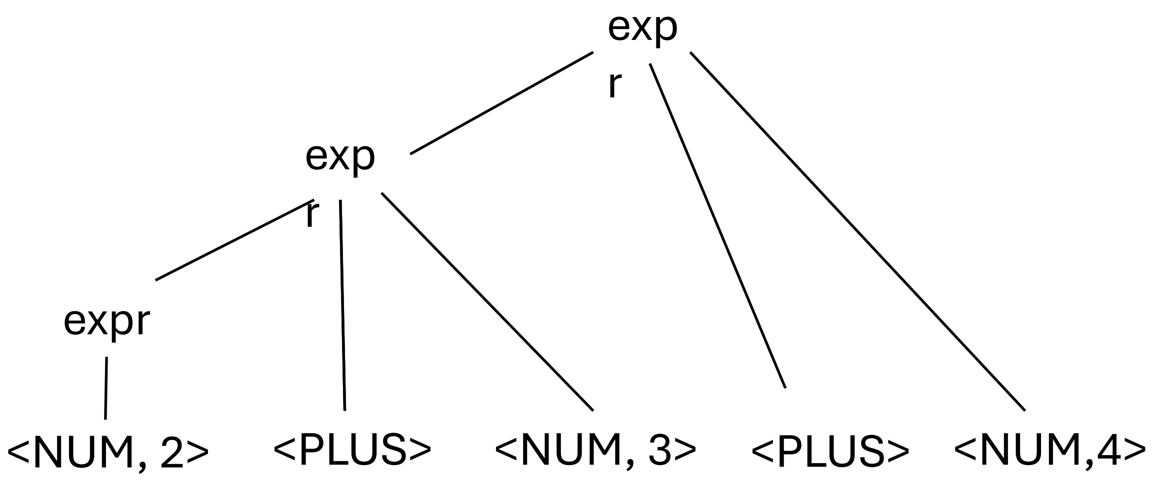
Operator	Name	Productions
+	`expr`	`: expr PLUS NUM` `| NUM`



Bottom-up parsing

`input: 2+3+4`

Operator	Name	Productions
+	`expr`	`: expr PLUS NUM` `| NUM`



Bottom up

Pros:

- can handle grammars expressed more naturally
- can encode precedence and associativity even if grammar is ambiguous

Cons:

- algorithm is complicated
- in many cases slower than top down

Top Down

Eithe LL(1) Table Driven Top-Down
Recursive-Descent (Manually Coded)

Do left-factoring on the grammar to avoid infinite recursion, then apply First Set, Follow Set, First+ Set to create predictive grammar without backtracking.

```
root = start symbol;
focus = root;
push(None);
to_match = s.token();

while (true):
  if (focus is a nonterminal)
    pick next rule (A ::= B1,B2,B3...BN);
    push(BN... B3, B2);
    focus = B1

  else if (focus == to_match)
    to_match = s.token()
    focus = pop()

  else if (to_match == None and focus == None)
    Accept
```

What can go wrong

```
1: Expr   ::= Expr Op Unit
2:          |   Unit
3: Unit   ::= '(' Expr ')'
4:          |    ID
5: Op     ::= '+'
6:          |   '*'
```

Can we derive the string `(a+b)*c`

Variable	Value
focus	
to_match	
s.istring	
stack	

Expanded Rule	Sentential Form
start	Expr
2	Expr Op Unit
2	Expr Op Unit Op Unit
2	Expr Op Unit Op Unit Op Unit
2	Expr Op Unit ....

Infinite recursion!

Top down parsing does not handle left recursion

```
1: Expr   ::= Expr Op Unit
2:          |   Unit
3: Unit   ::= '(' Expr ')'
4:          |    ID
5: Op     ::= '+'
6:          |   '*'
```

direct left recursion

```
1: Expr_base ::= Unit
2:             |   Expr_op
3: Expr_op   ::= Expr_base Op Unit
4: Unit      ::= '(' Expr_base ')'
5:              |    ID
6: Op        ::= '+'
7:             |   '*'
```

indirect left recursion

Top down parsing cannot handle either of these

Top down parsing does not handle left recursion

- In general, any CFG can be re-written without left recursion
  - However, the transformation may affect associativity
  - or increase the number of rules
  - but it is always possible