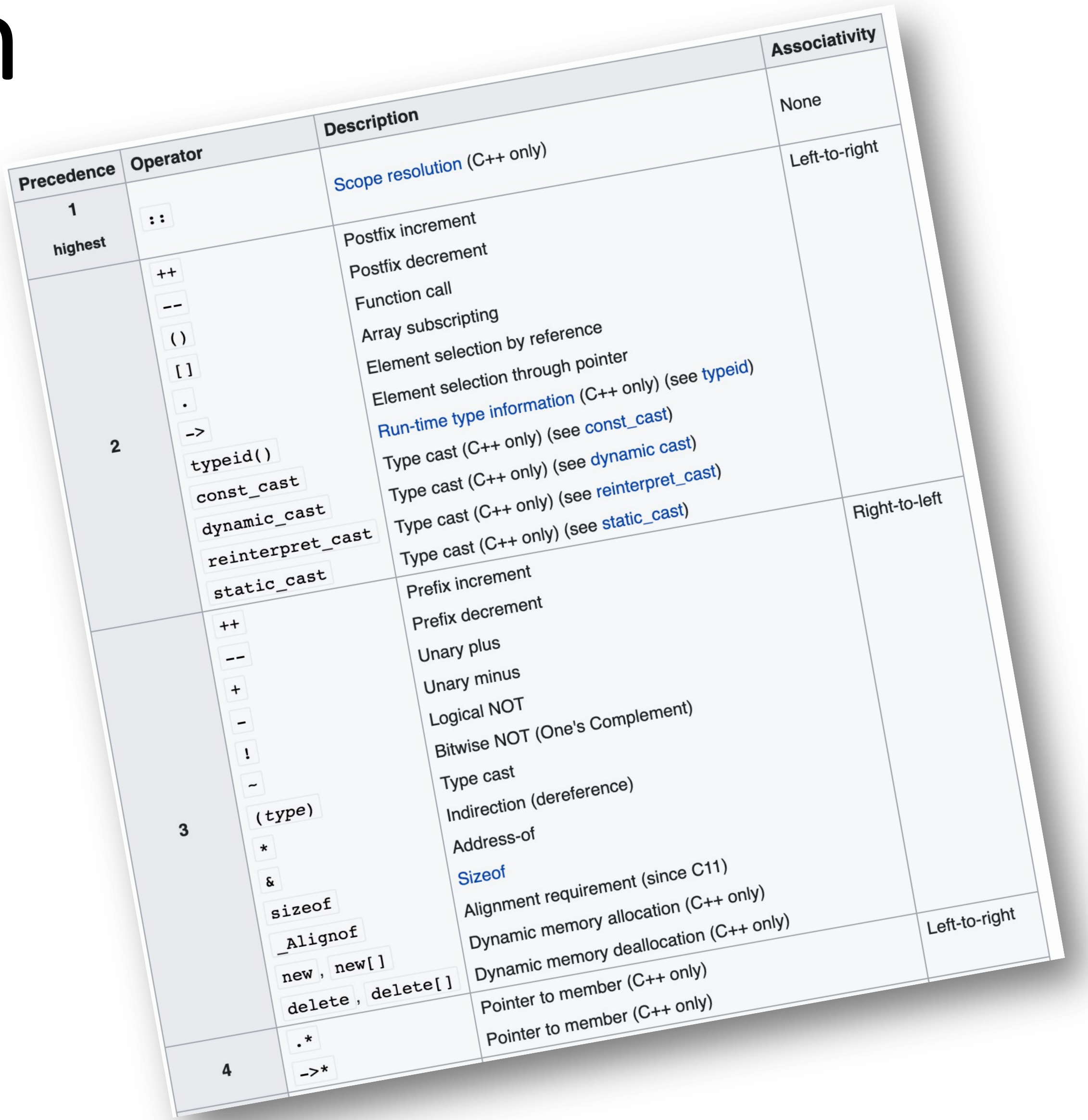
CSE211: Compiler Design

Oct. 1, 2021

- **Topic**: Parsing overview 3 (associativity and production actions)

- **Questions**:
  - *What is associativity?*
  - *What are some operators that are associative and what are some that are not?*



Precedence	Operator	Description	Associativity
1 highest	`::`	Scope resolution (C++ only)	None
2	`++`	Postfix increment	Left-to-right
	`--`	Postfix decrement	
	`()`	Function call	
	`[]`	Array subscripting	
	`.`	Element selection by reference	
	`->`	Element selection through pointer	
	`typeid()`	Run-time type information (C++ only) (see typeid)	
	`const_cast`	Type cast (C++ only) (see const_cast)	
	`dynamic_cast`	Type cast (C++ only) (see dynamic cast)	
	`reinterpret_cast`	Type cast (C++ only) (see reinterpret_cast)	
	`static_cast`	Type cast (C++ only) (see static_cast)	
3	`++`	Prefix increment	Right-to-left
	`--`	Prefix decrement	
	`+`	Unary plus	
	`-`	Unary minus	
	`!`	Logical NOT	
	`~`	Bitwise NOT (One's Complement)	
	`(type)`	Type cast	
	`*`	Indirection (dereference)	
	`&`	Address-of	
	`sizeof`	Sizeof	
	`_Alignof`	Alignment requirement (since C11)	
	`new`, `new[]`	Dynamic memory allocation (C++ only)	
	`delete`, `delete[]`	Dynamic memory deallocation (C++ only)	
4	`.*`	Pointer to member (C++ only)	Left-to-right
	`->*`	Pointer to member (C++ only)	

Announcements

- Homework 1 will be released on Monday
- if you have ideas for projects, we can start discussing!
- Join the slack for discussions
  - Thanks to Farid for some initial discussion points!
- New people:
  - Introductions

Review

- How do we define a context-free grammar?

BNF Production Rules

- Tokens:
    - NUM = [0-9]+
    - PLUS = ‘\+’
    - TIMES = ’*’
    - LP = ‘\(’
    - RP = \)’

```
expression : NUM
           | expression PLUS expression
           | expression TIMES expression
           | LP expression RP
```

Review

- How do we determine if a string matches a context-free grammar?

Parse trees

- A string is accepted by a BNF form if and only if there exists a parse tree.

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

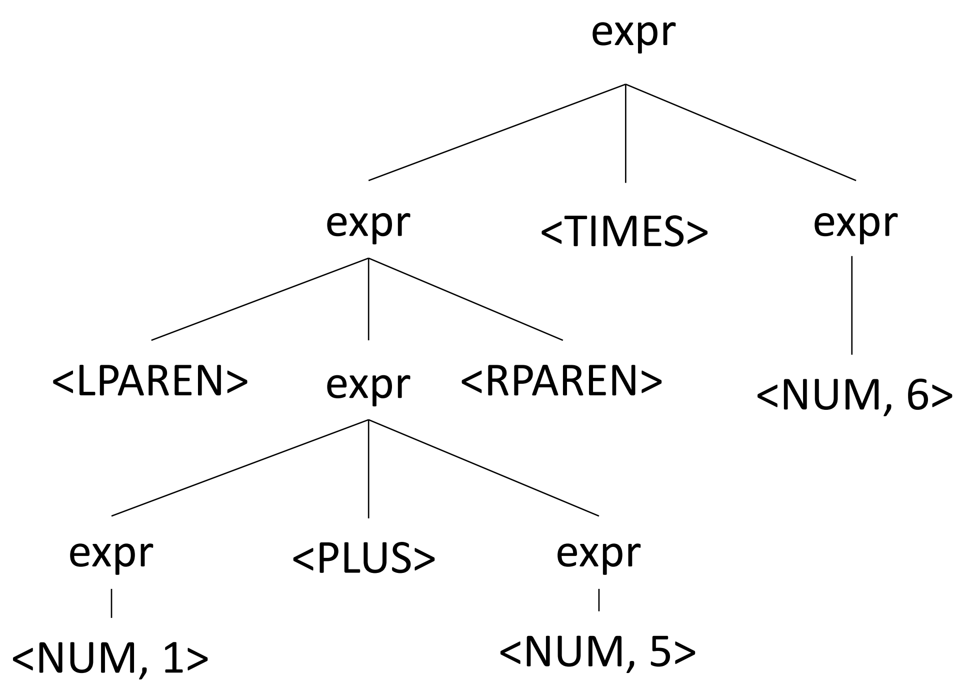
```
expr : NUM
       | expr PLUS expr
       | expr TIMES expr
       | LPAREN expr RPAREN
```

Parse trees

- A string is accepted by a BNF form if and only if there exists a parse tree.

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

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



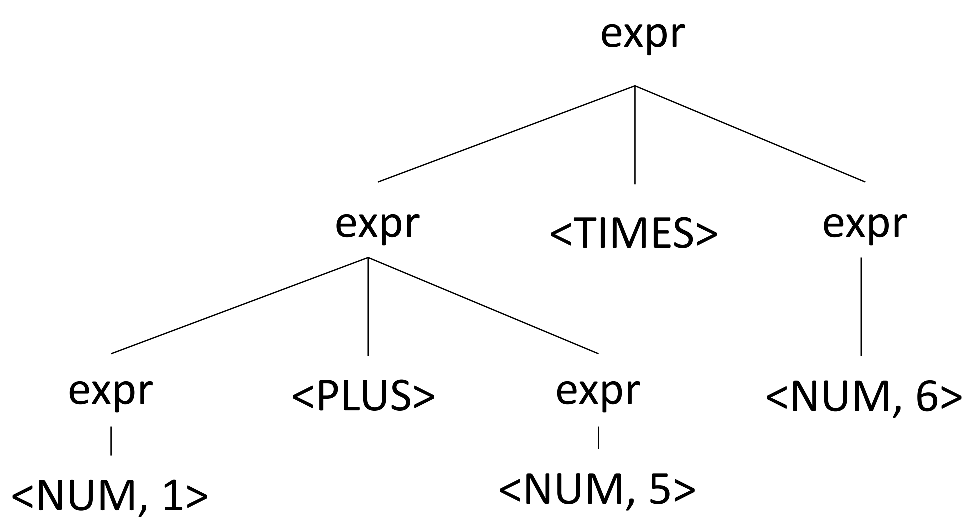
Review

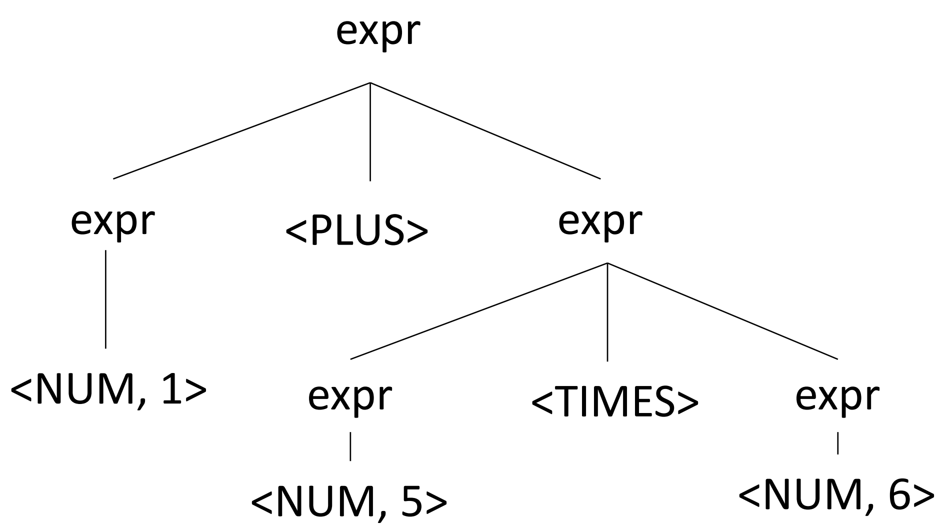
- What do we call it when a CFG can produce 2 different parse trees for the same string? Is this an issue?

Ambiguous grammars

- `input: 1 + 5 * 6`

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





Review

- How do we encode precedence in a CFG?

Now lets create a parse tree

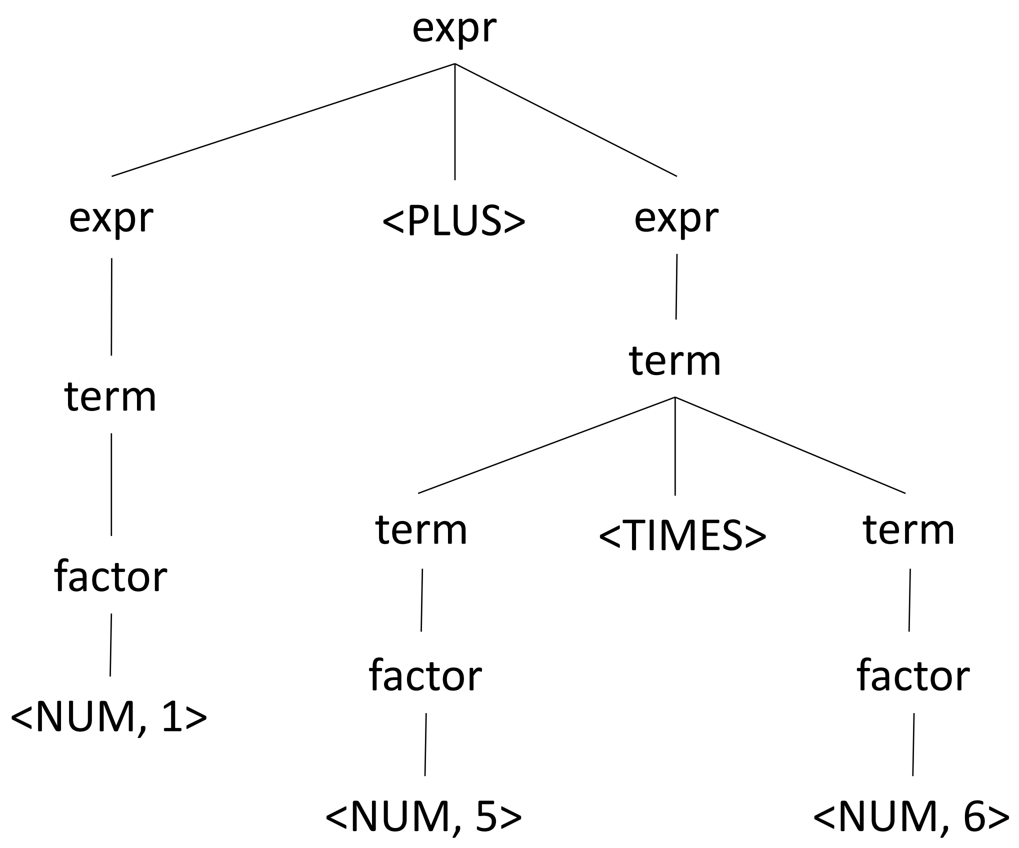
`input: 1+5*6`

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

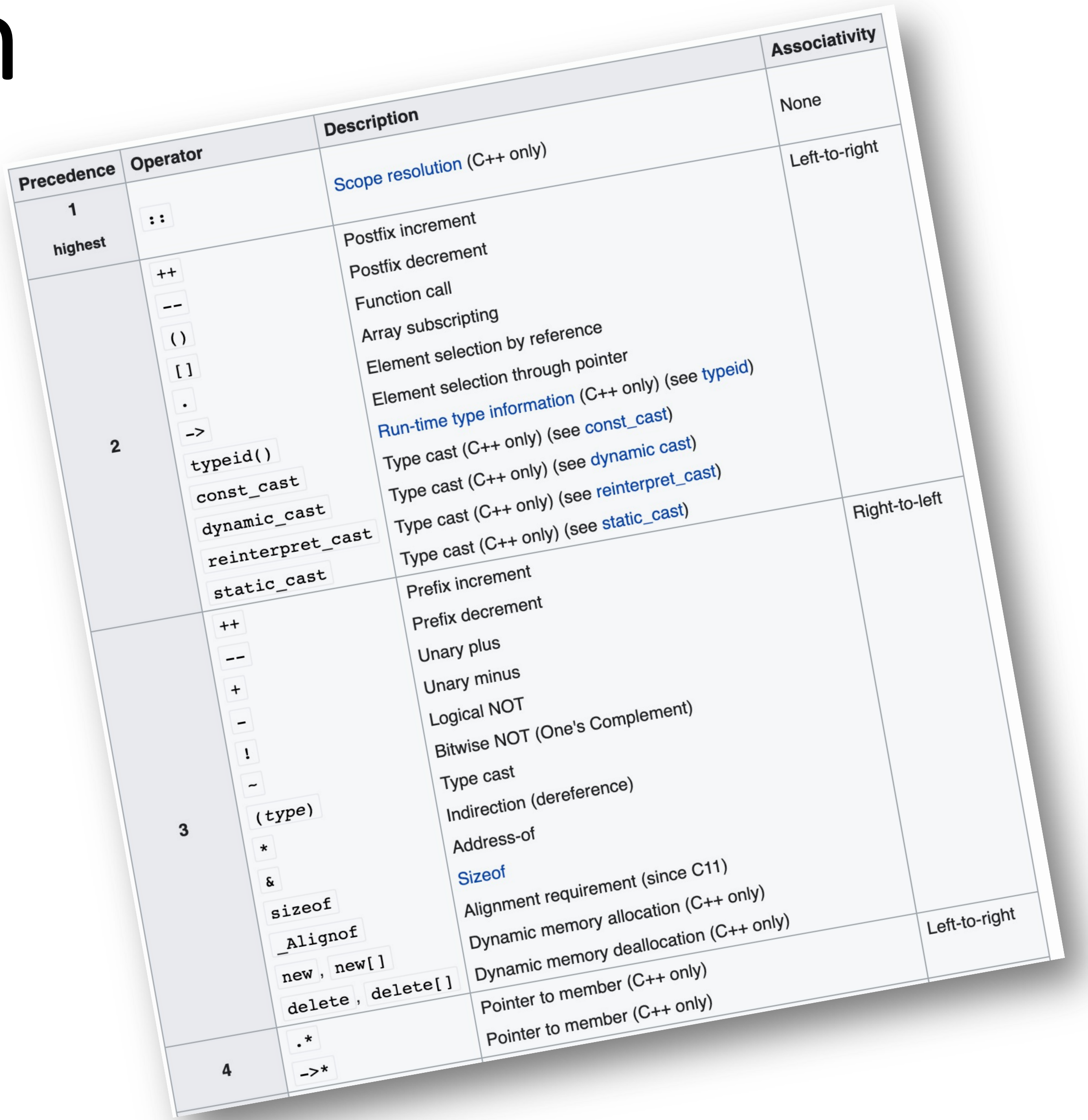


CSE211: Compiler Design

Oct. 1, 2021

- **Topic**: Parsing overview 3 (associativity and production actions)

- **Questions**:
  - *What is associativity?*
  - *What are some operators that are associative and what are some that are not?*



Precedence	Operator	Description	Associativity
1 highest	`::`	Scope resolution (C++ only)	None
2	`++`	Postfix increment	Left-to-right
	`--`	Postfix decrement	
	`()`	Function call	
	`[]`	Array subscripting	
	`.`	Element selection by reference	
	`->`	Element selection through pointer	
	`typeid()`	Run-time type information (C++ only) (see typeid)	
	`const_cast`	Type cast (C++ only) (see const_cast)	
	`dynamic_cast`	Type cast (C++ only) (see dynamic cast)	
	`reinterpret_cast`	Type cast (C++ only) (see reinterpret_cast)	
	`static_cast`	Type cast (C++ only) (see static_cast)	
3	`++`	Prefix increment	Right-to-left
	`--`	Prefix decrement	
	`+`	Unary plus	
	`-`	Unary minus	
	`!`	Logical NOT	
	`~`	Bitwise NOT (One's Complement)	
	`(type)`	Type cast	
	`*`	Indirection (dereference)	
	`&`	Address-of	
	`sizeof`	Sizeof	
	`_Alignof`	Alignment requirement (since C11)	
	`new`, `new[]`	Dynamic memory allocation (C++ only)	
	`delete`, `delete[]`	Dynamic memory deallocation (C++ only)	
4	`.*`	Pointer to member (C++ only)	Left-to-right
	`->*`	Pointer to member (C++ only)	

Let’s make some more parse trees

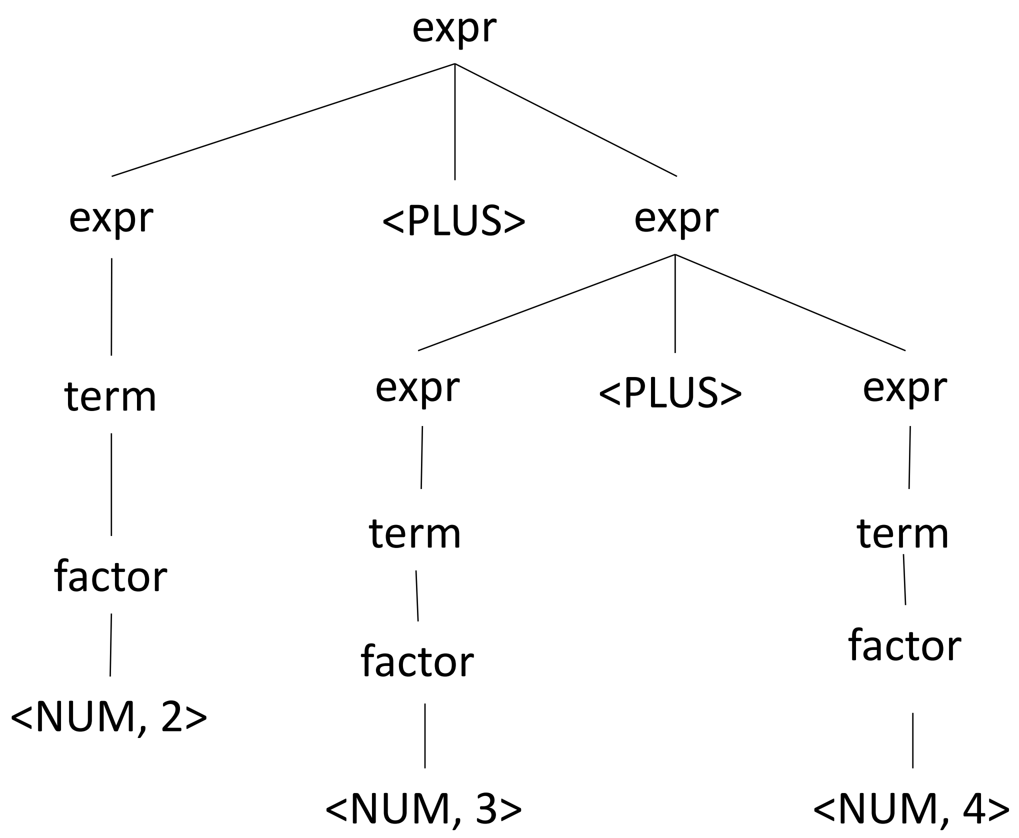
`input: 2+3+4`

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

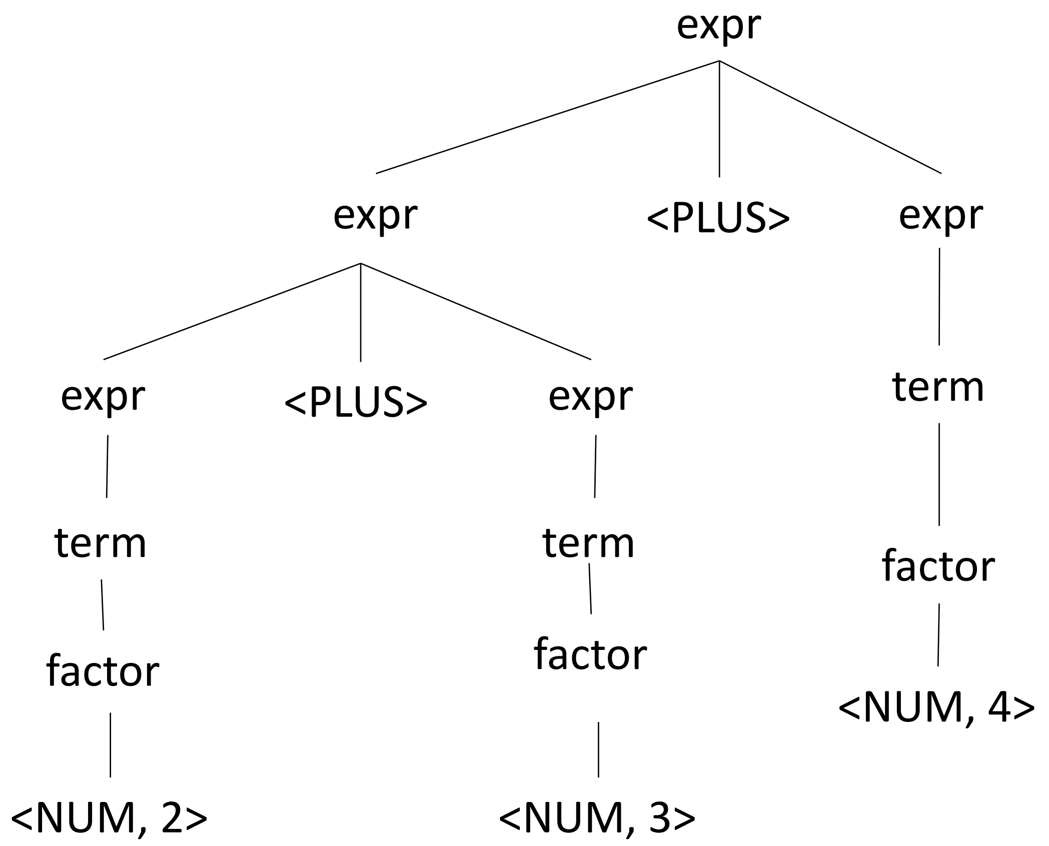
Operator	Name	Productions
+	expr	: expr PLUS expr | term
*	term	: term TIMES term | factor
()	factor	: LP expr RP | NUM

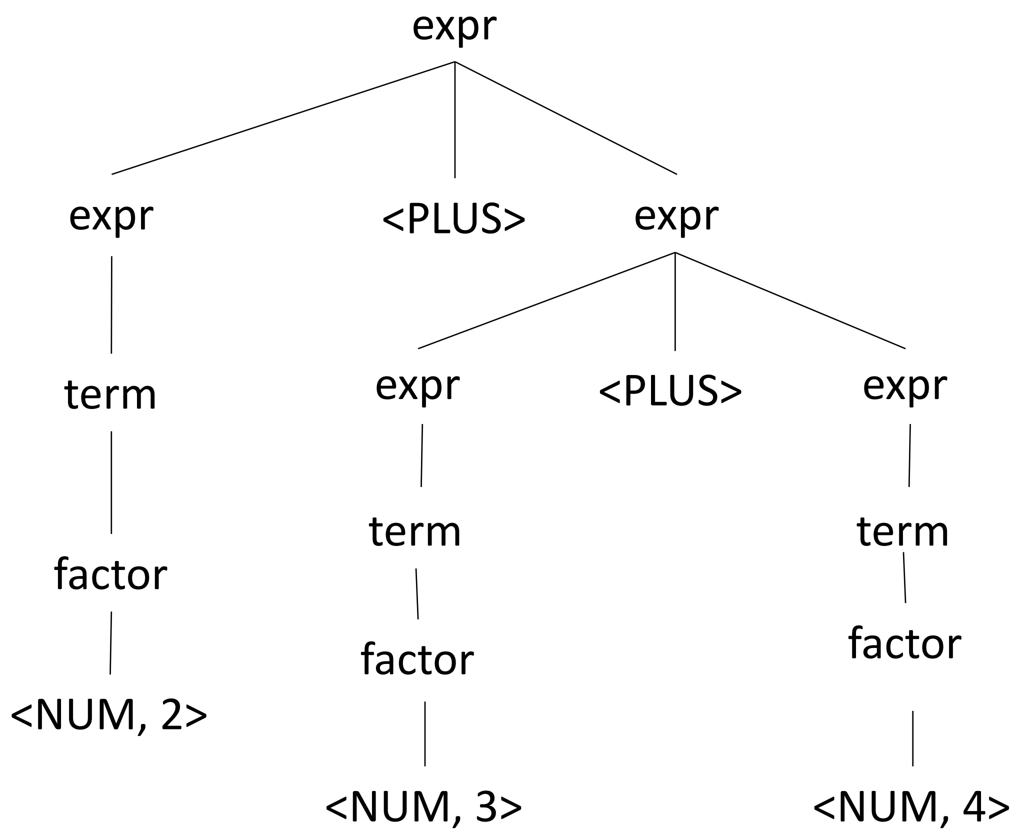
`input: 2+3+4`



This is ambiguous, is it an issue?

`input: 2+3+4`



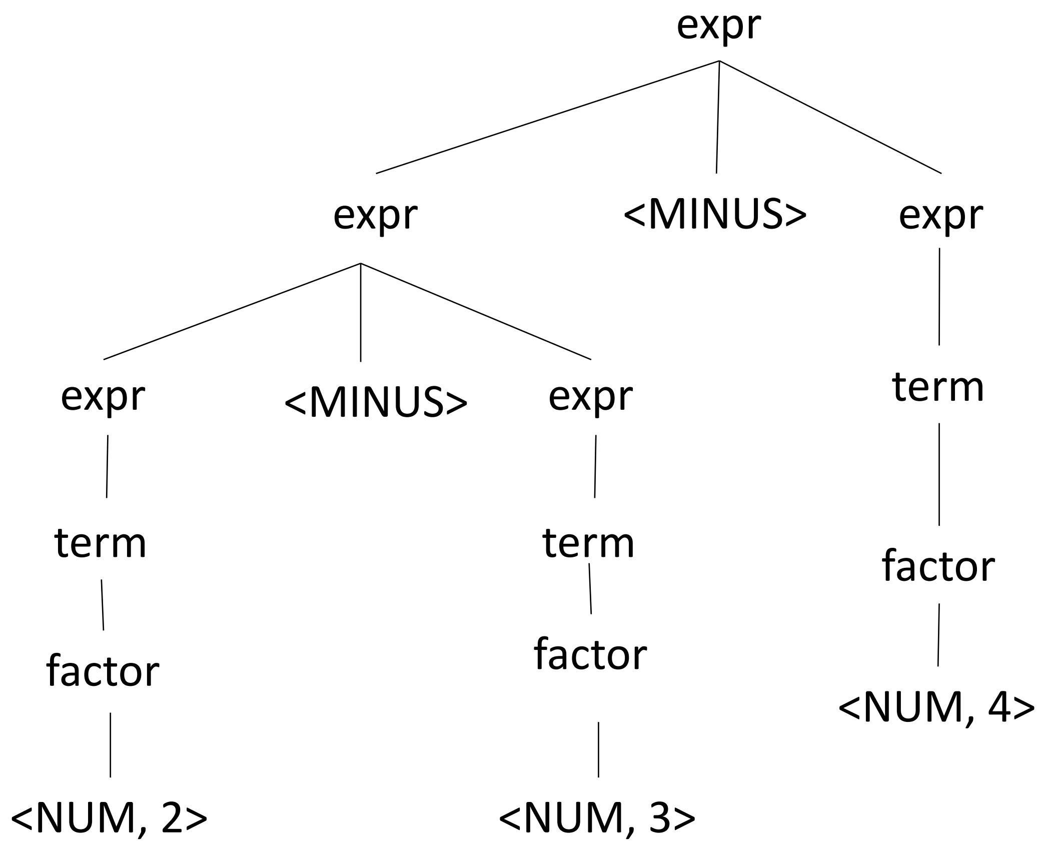


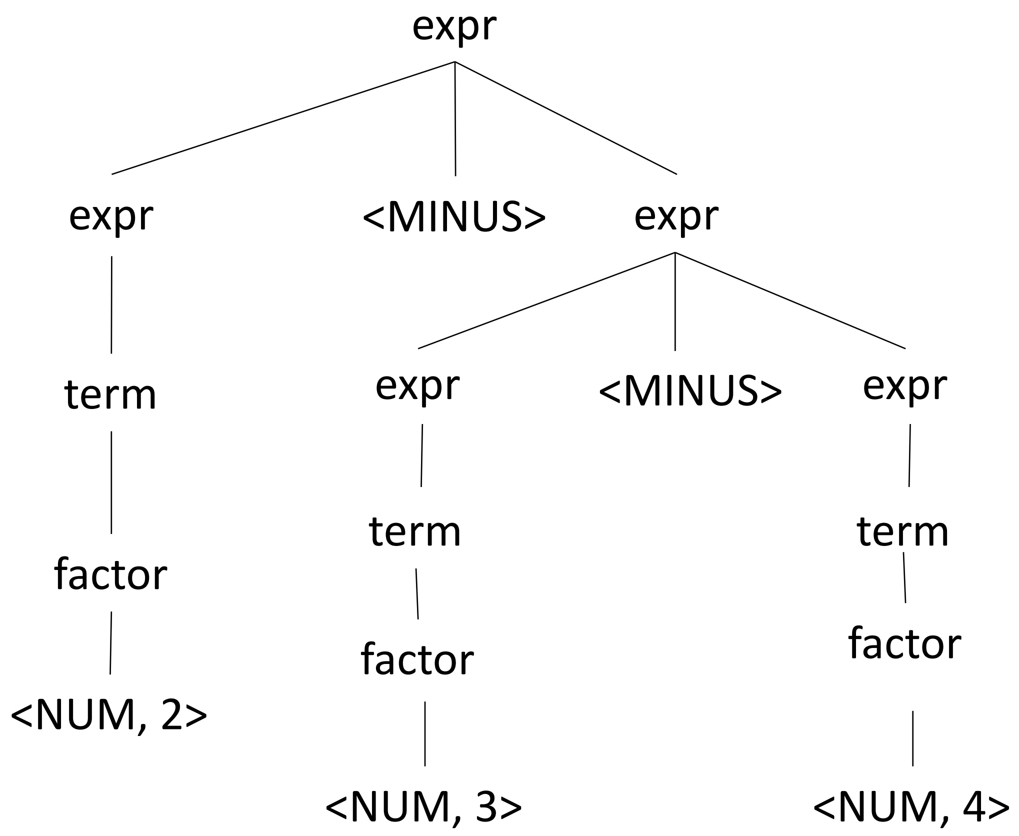
What about for a different operator?

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

What about for a different operator?

`input: 2-3-4`





Which one is right?

Associativity

The order in which we evaluate the same operator

Sometimes it doesn’t matter:

- Integer arithmetic
- Integer multiplication
- What else?

Good test:

- ((a OP b) OP c) == (a OP (b OP c))

What about floating point arithmetic?

Associativity

The order in which we evaluate the same operator

- 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

The order in which we evaluate the same operator

- 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?
  - Assignment, power operator

How to encode associativity?

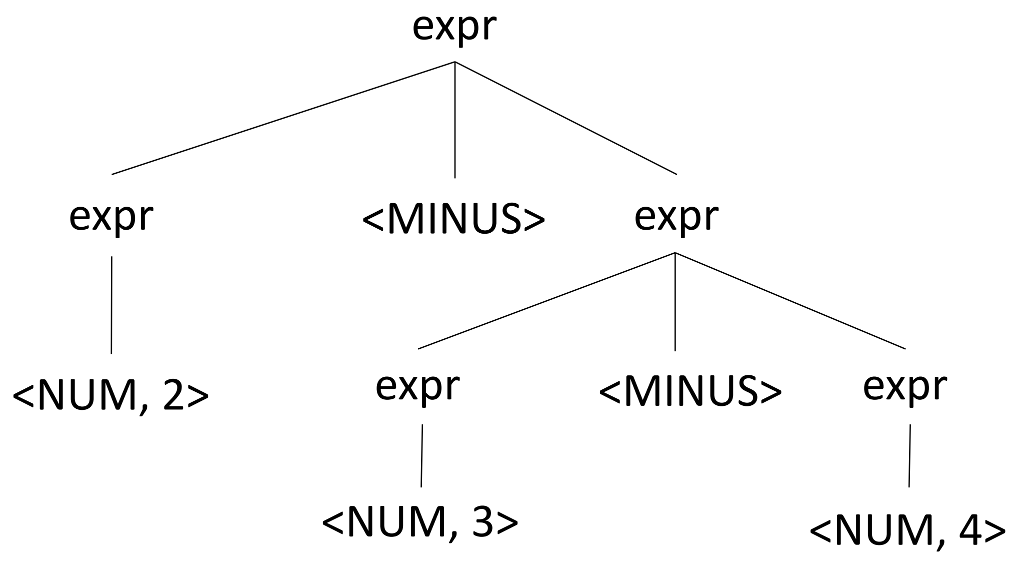
- Like precedence, some tools (e.g. YACC) allow associativity specification through keywords:
  - “+”: left, “^”: right

- Like precedence, we can also encode it into the production rules

Associativity for a single operator

Operator	Name	Productions
-	expr	: expr MINUS expr | NUM

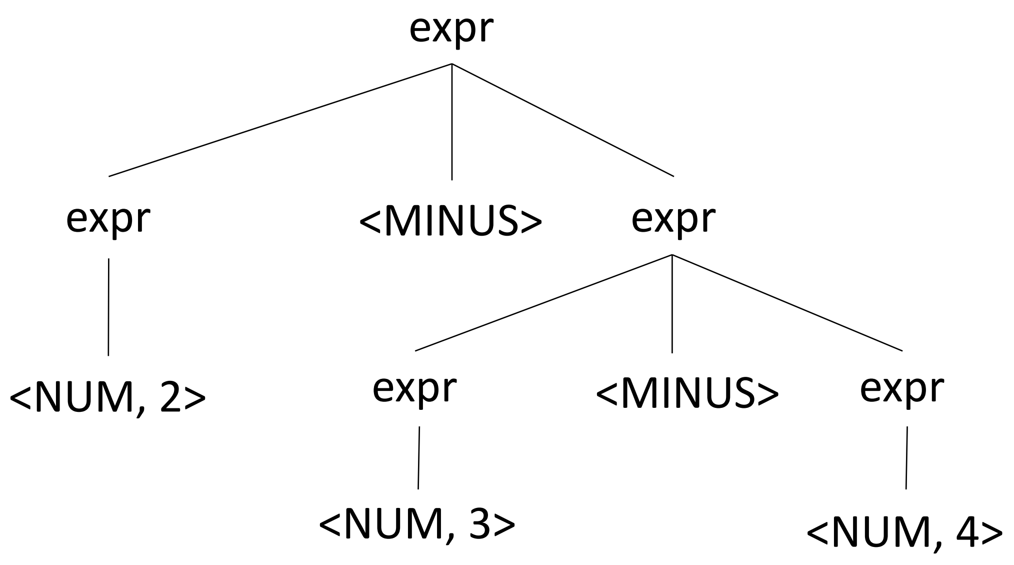
```
input: 2-3-4
```



Associativity for a single operator

Operator	Name	Productions
-	expr	: expr MINUS NUM | NUM

`input: 2-3-4`

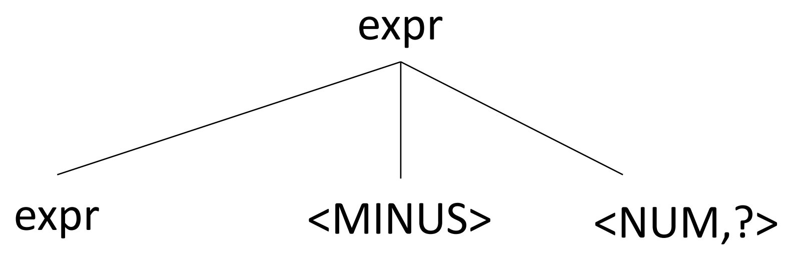


No longer allowed

Associativity for a single operator

Operator	Name	Productions
-	expr	: expr MINUS NUM | NUM

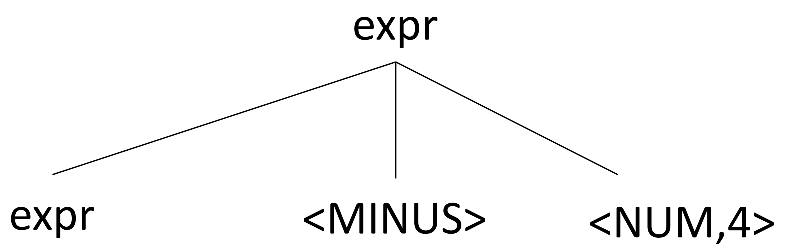
```
input: 2-3-4
```



Lets start over

Associativity for a single operator

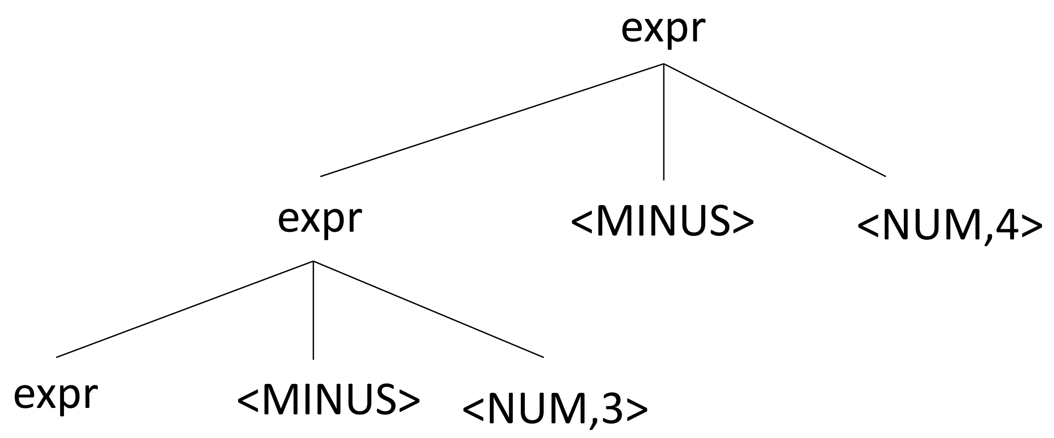
`input: 2-3-4`



Operator	Name	Productions
-	expr	: expr MINUS NUM | NUM

Associativity for a single operator

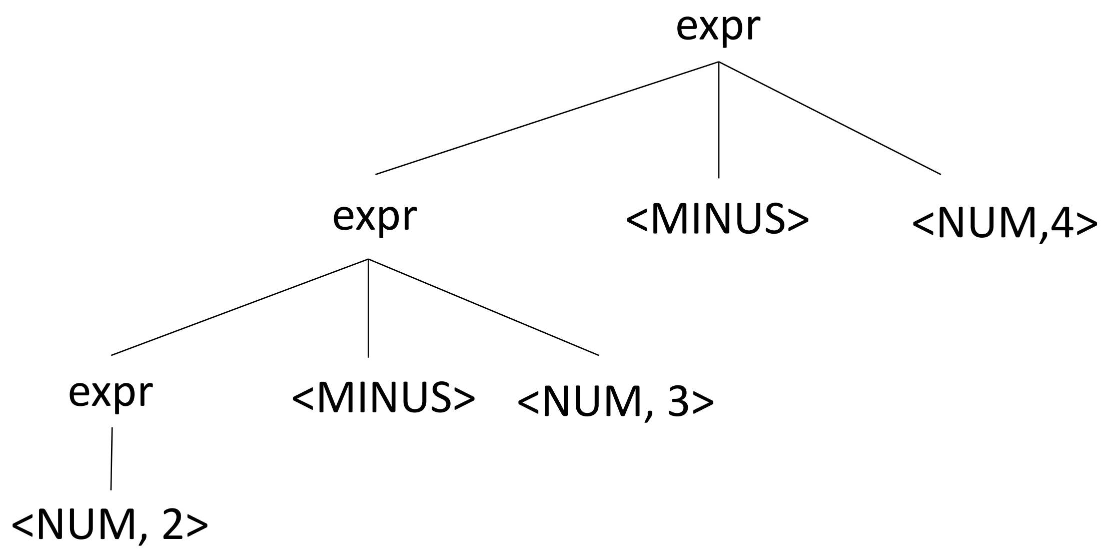
`input: 2-3-4`



Operator	Name	Productions
-	expr	: expr MINUS NUM | NUM

Associativity for a single operator

`input: 2-3-4`



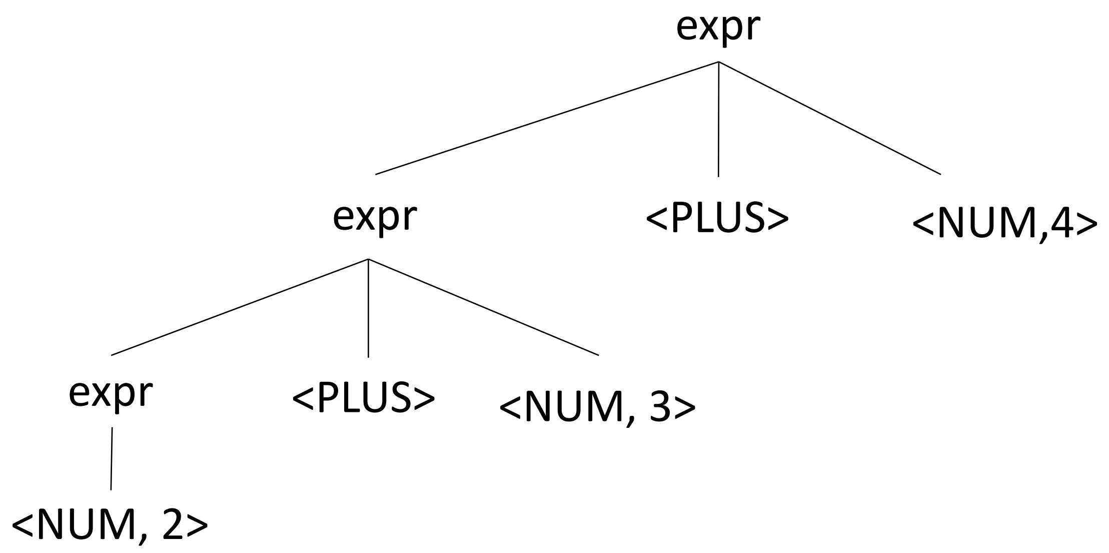
Operator	Name	Productions
-	expr	: expr MINUS NUM | NUM

Should you have associativity when its not required?

Benefits?
Drawbacks?

Operator	Name	Productions
+	expr	: expr PLUS NUM | NUM

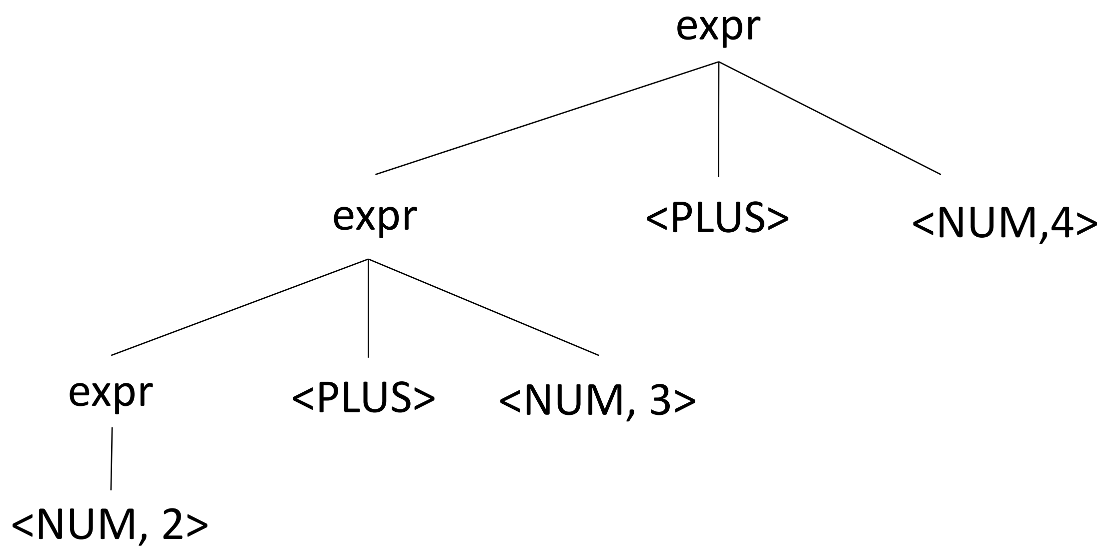
```
input: 2+3+4
```



Should you have associativity when its not required?

Benefits?
Drawbacks?

`input: 2+3+4`



Operator	Name	Productions
+	expr	: expr PLUS NUM | NUM

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 grammar

Let’s add minus, division and power to our grammar

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 = ‘\^’
```

Let’s make a richer grammar

Let’s add minus, division and power to our grammar

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

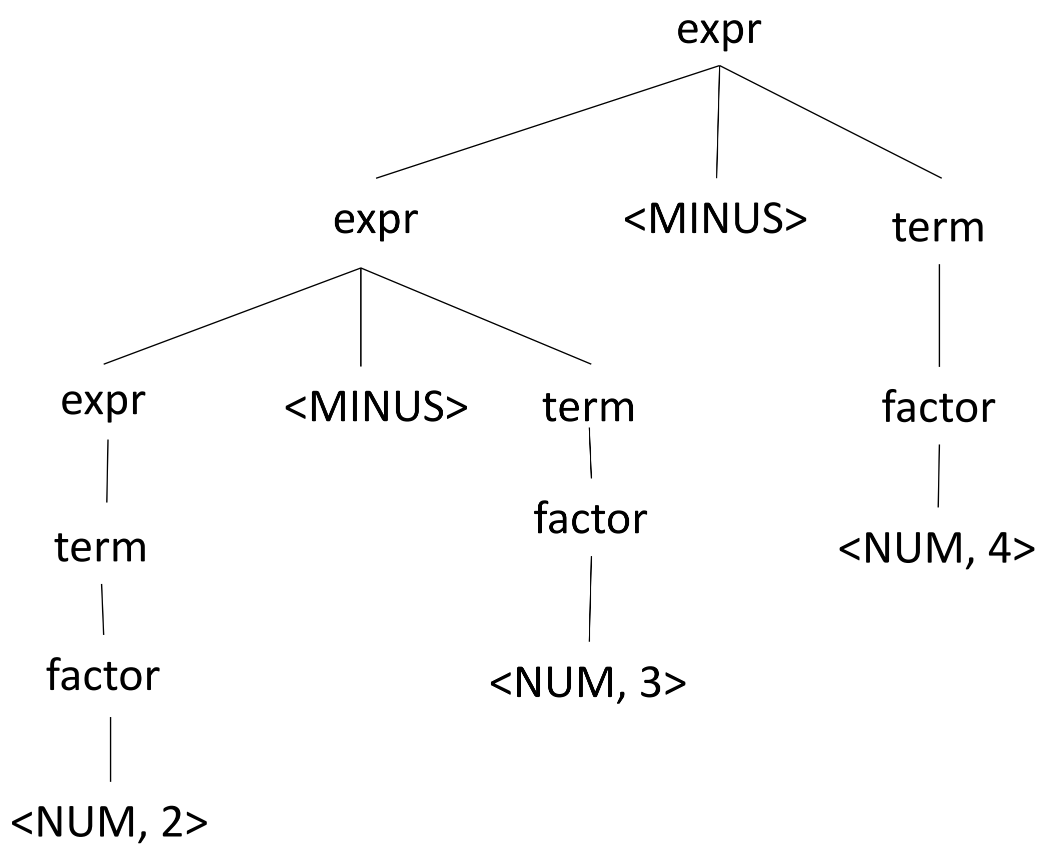
Tokens:

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

Let’s make a richer grammar

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

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



Production rules in a compiler

- Great to check if a string is grammatically correct

- But can the production rules actually help us with compilation??

Production actions

- Each production *option* is associated with a code block
  - It can use values from its children
  - it returns a value to its parent
  - Executed in a post-order traversal (natural order traversal)

Production actions

`input: 1+5*6`

Example: executing a mathematical expression during parsing

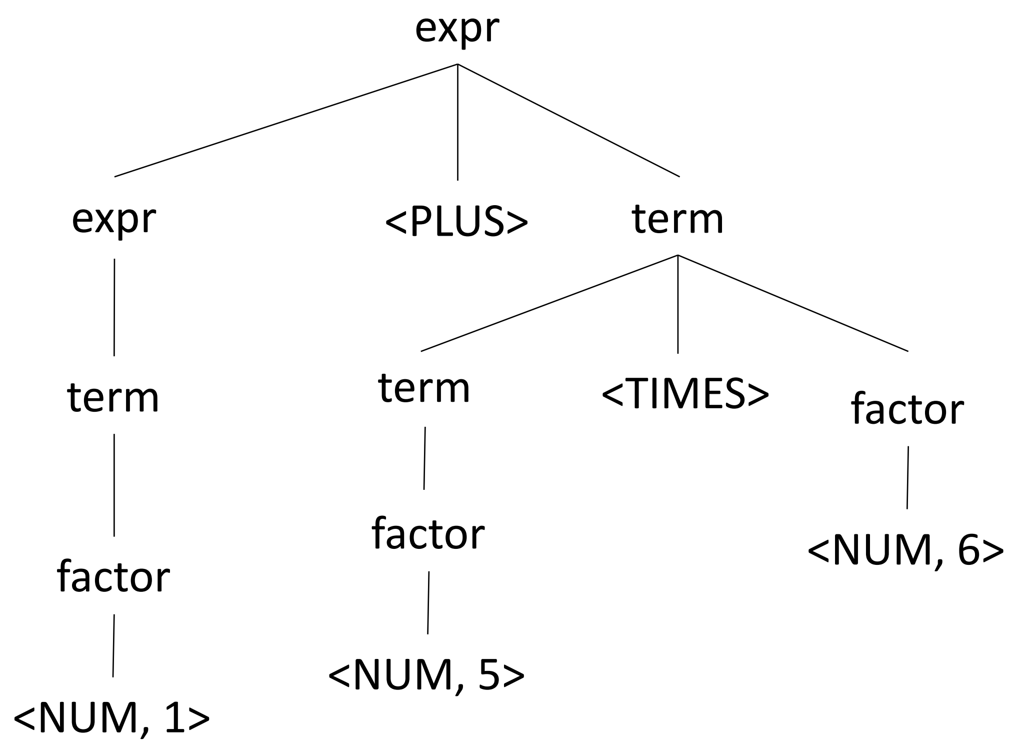
Children values are passed in as an array `C`, indexed from left to right

Operator	Name	Productions	Actions

+,-	expr	: expr PLUS term
		| expr MINUS term
		| term
			`{ret C[0] + C[2]}`
			`{}`
			`{ret C[0]}`

*,/	term	: term TIMES factor
		: term DIV factor
		| factor
			`{ret C[0] * C[2]}`
			`{}`
			`{ret C[0]}`

()	factor	: LPAR expr RPAR
		| NUM
			`{}`
			`{ret int(C[0])}`



Production actions

`input: 1+5*6`

Example: executing a mathematical expression during parsing

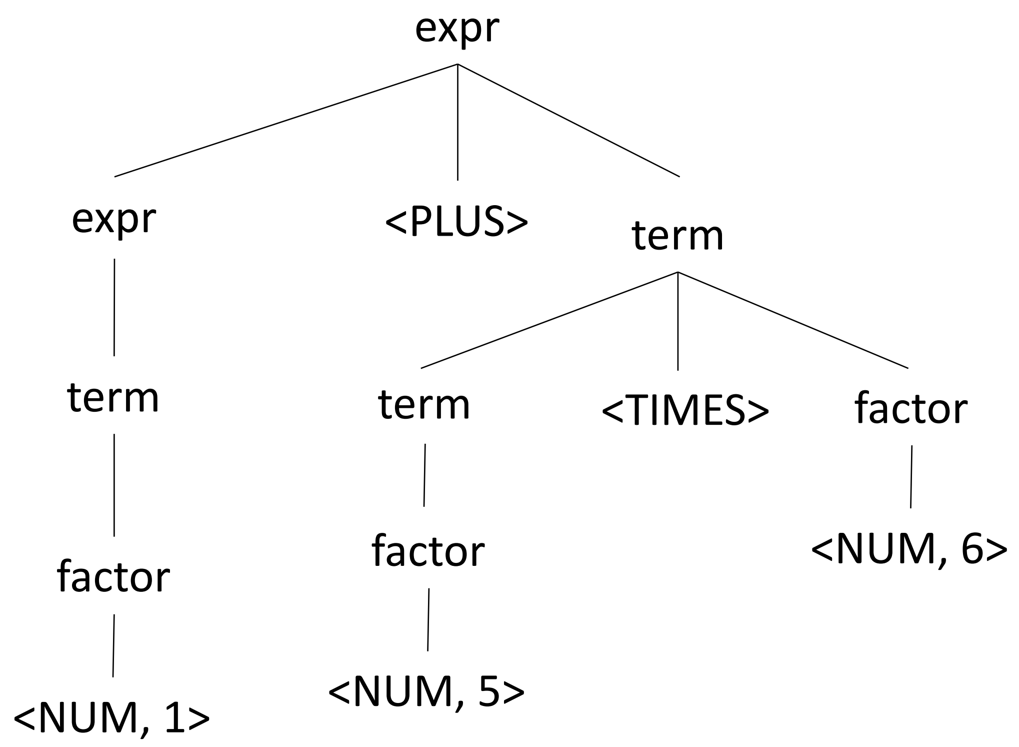
Children values are passed in as an array `C`, indexed from left to right

Operator	Name	Productions	Actions

+,-	expr	: expr PLUS term
		| expr MINUS term
		| term
			`{ret C[0] + C[2]}`
			`{ret C[0] - C[2]}`
			`{ret C[0]}`

*,/	term	: term TIMES factor
		: term DIV factor
		| factor
			`{ret C[0] * C[2]}`
			`{ret C[0] / C[2]}`
			`{ret C[0]}`

()	factor	: LPAR expr RPAR
		| NUM
			`{ret C[1]}`
			`{ret int(C[0])}`



We have just implemented a simple arithmetic interpreter!
Could this be in a compiler?

Next week

- We will look at LEX and YACC
- Homework will be released on Monday
- Enjoy your weekend!