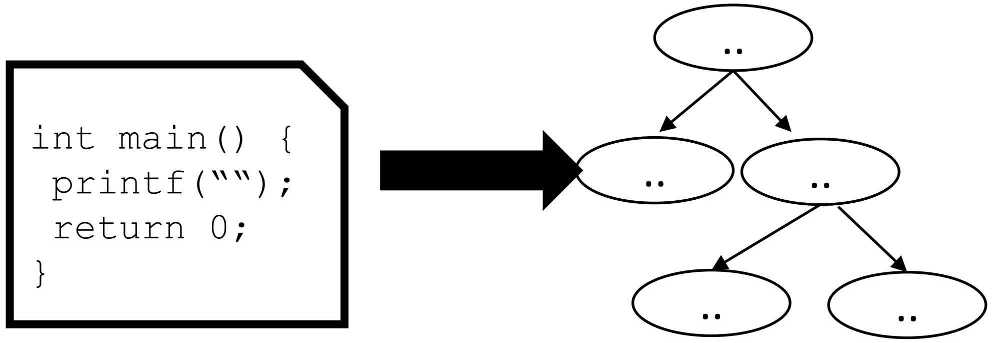
CSE110A: Compilers

Topics:

- *Syntactic Analysis continued*
  - *Derivations*
  - *Parse trees*
  - *Precedence and associativity*



```
int main() {
 printf("");
 return 0;
}
```

Ambiguous grammars

- What happens when different derivations have different parse trees?

```
1: Statement ::= ″if″ Expr “then” Statement “else” Statement
2:           |   ″if″ Expr “then” Statement
3:           |    Assignment
4:           |    ....
```

can we derive this string?

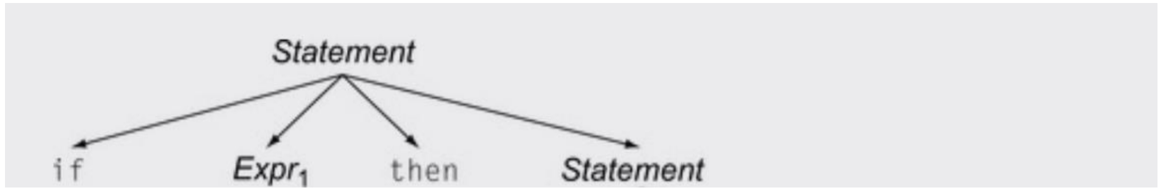
if $Expr_1$ **then** **if** $Expr_2$ **then** $Assignment_1$ **else** $Assignment_2$

Next few figures taken from the EAC book (Chapter 3.1)

Ambiguous grammars

```
1: Statement ::= "if" Expr "then" Statement "else" Statement
2:            |   "if" Expr "then" Statement
3:            |    Assignment
4:            |    ....
```

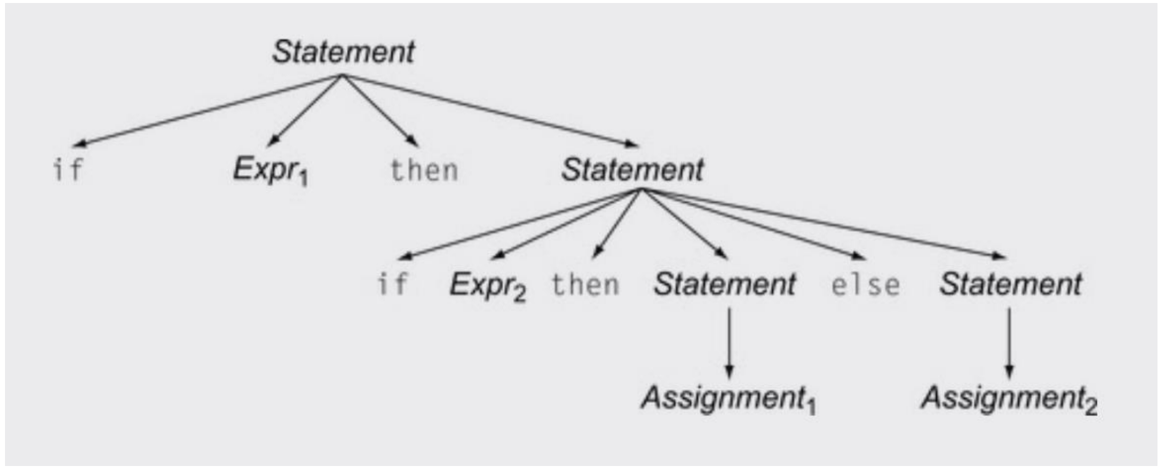
if $Expr_1$ **then** **if** $Expr_2$ **then** $Assignment_1$ **else** $Assignment_2$



Ambiguous grammars

```
1: Statement ::= ”if” Expr “then” Statement “else” Statement
2:             |   ”if” Expr “then” Statement
3:             |    Assignment
4:             |    ....
```

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

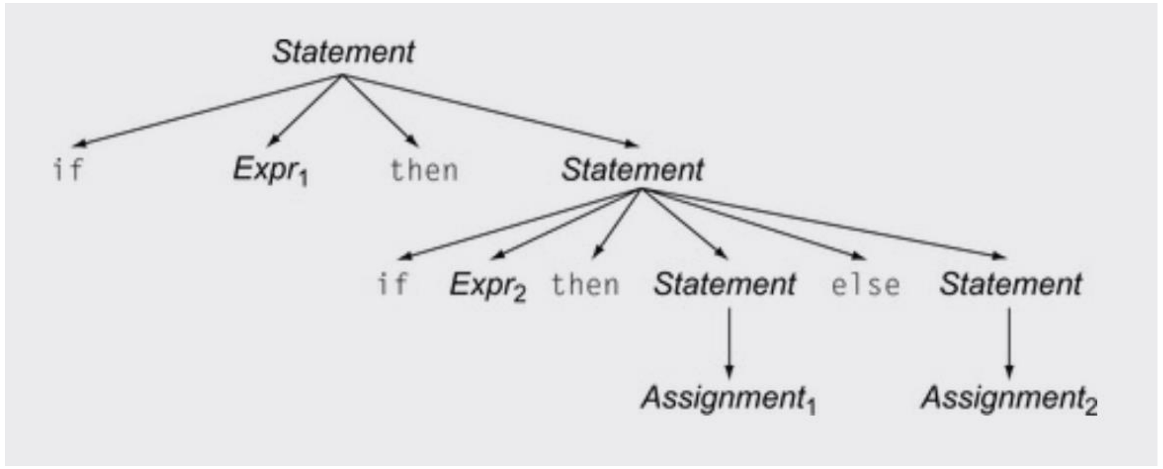


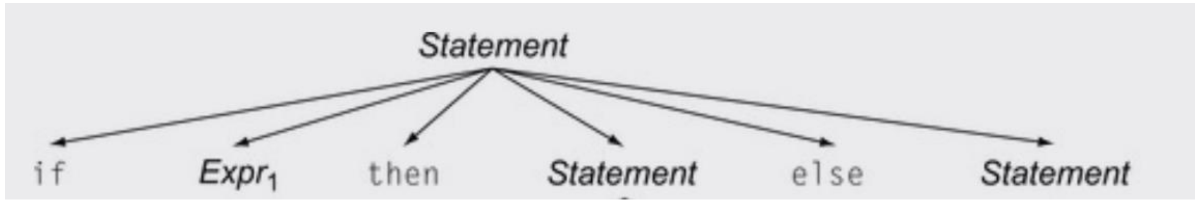
Valid derivation

Ambiguous grammars

```
1: Statement ::= "if" Expr "then" Statement "else" Statement
2:           |    "if" Expr "then" Statement
3:           |     Assignment
4:           |     ....
```

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



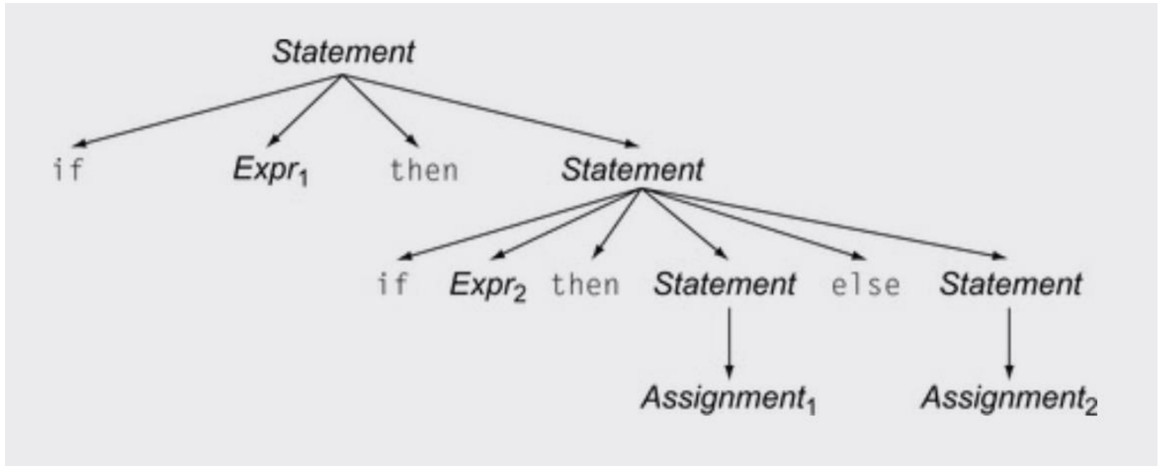


Valid derivation

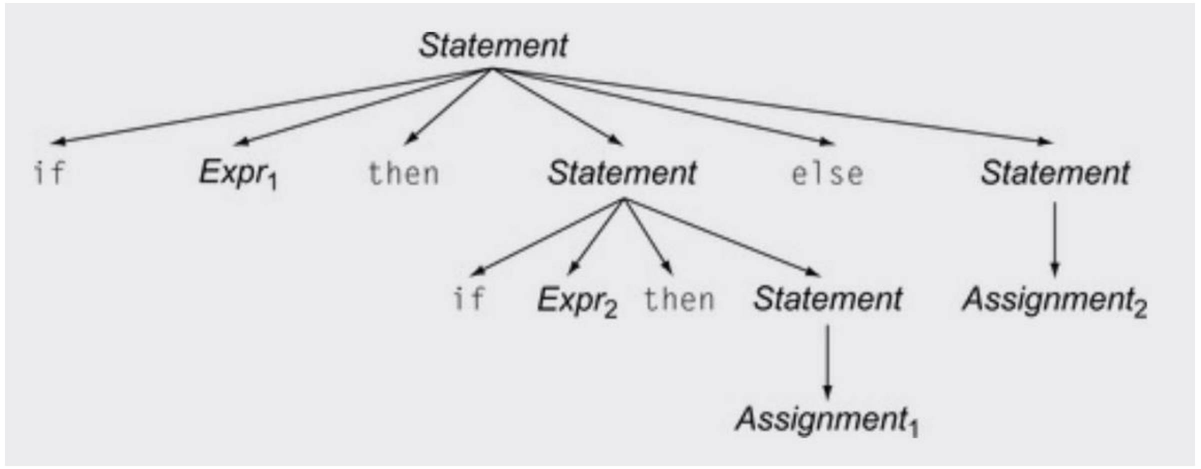
Ambiguous grammars

```
1: Statement ::= "if" Expr "then" Statement "else" Statement
2:             |   "if" Expr "then" Statement
3:             |    Assignment
4:             |    ....
```

$\boxed{\textbf{if } Expr_1 \textbf{ then } \textbf{if } Expr_2 \textbf{ then } Assignment_1 \textbf{ else } Assignment_2}$



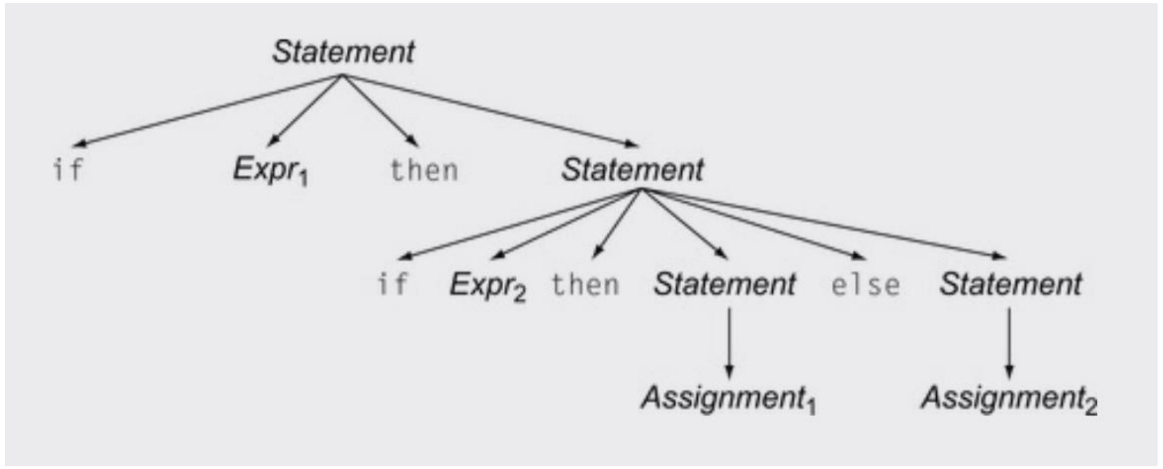
Valid derivation



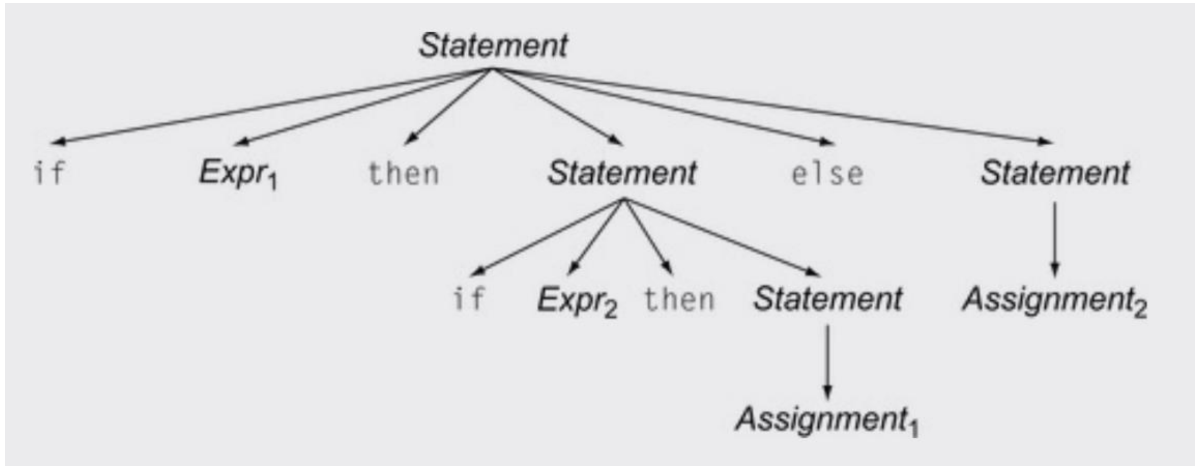
And another valid derivation

Ambiguous grammars

Is this an issue? Don't we only care if a grammar can derive a string?



Valid derivation



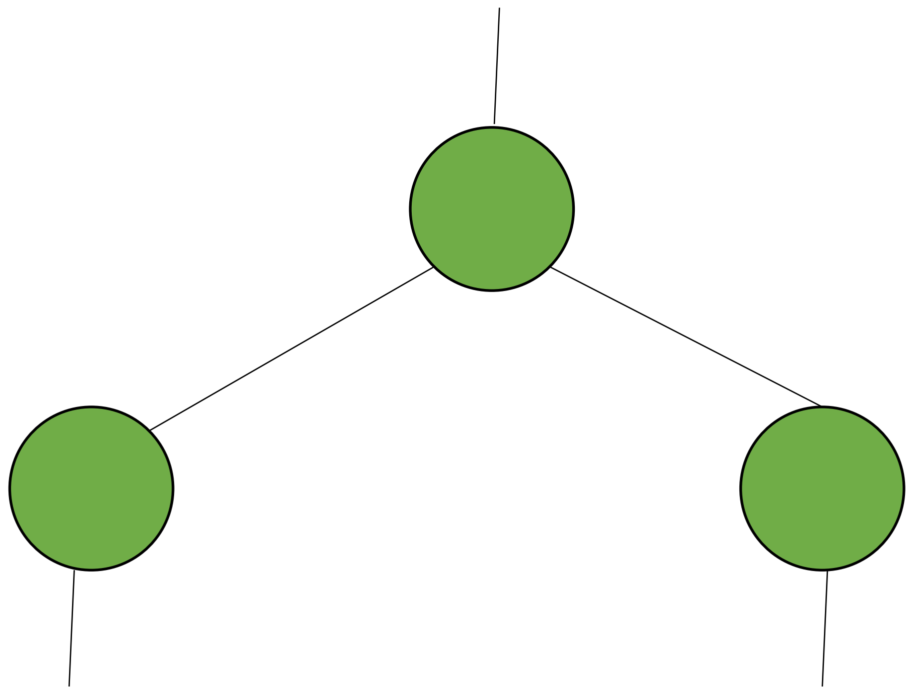
And another valid derivation

Meaning into structure

- We want to start encoding meaning into the parse structure. We will want as much structure as possible as we continue through the compiler

- The structure is that we want evaluation of program to correspond to a post order traversal of the parse tree (also called the natural traversal)

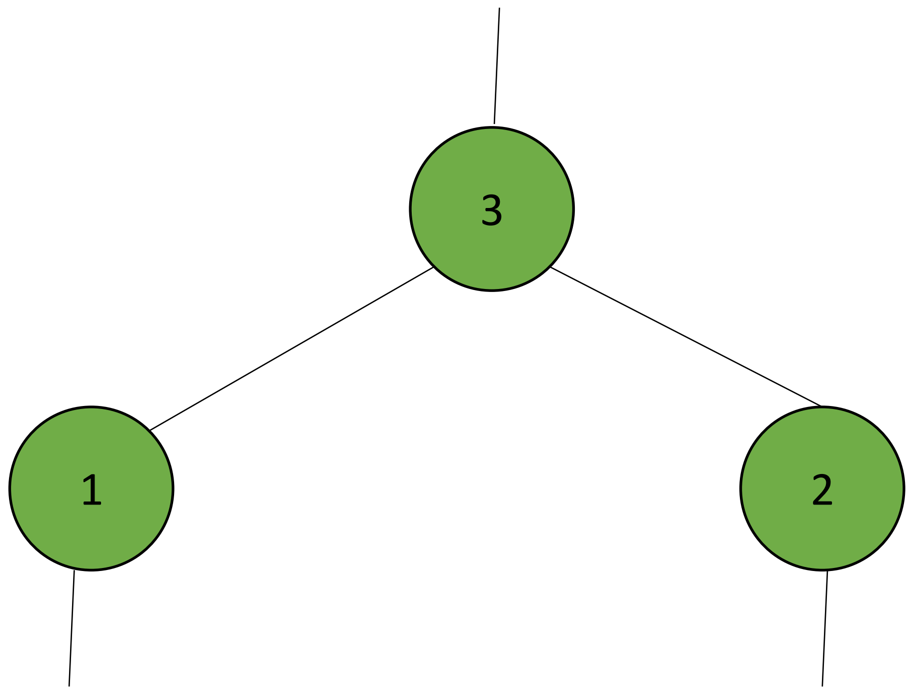
Post order traversal



visiting for for different types
of traversals:

pre order?
in order?
post order?

Post order traversal

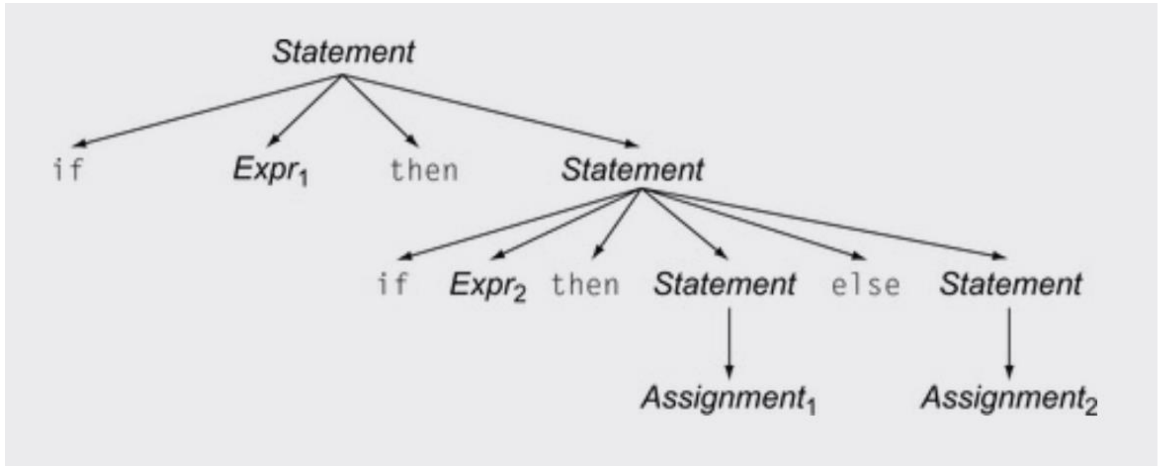


visiting for for different types of traversals:

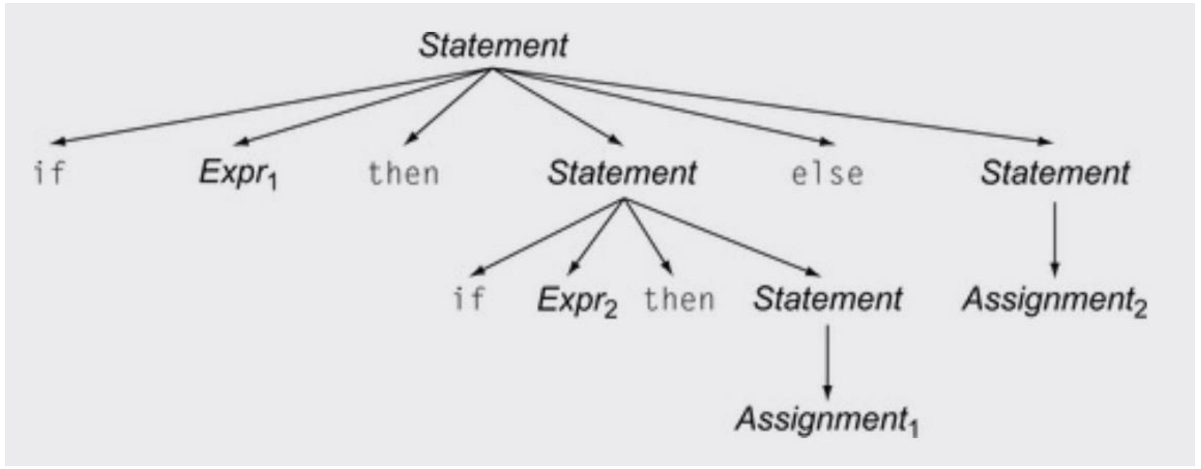
post order

Ambiguous grammars

Encoding meaning into structure can result in very different programs



Valid derivation



Also a valid derivation

Programming language structure

```
int x = 1; //true
int y = 0; //false
int check0 = 0;


if (x)
if (y)
pass();
else
check0 = 1;
```

pop quiz: what is the value of check0 at the end?

Programming language structure

```
x = 1
y = 0
check0 = 0


if (x):
if (y):
pass
else:
check0 = 1


print(check0)
```

How does Python handle this?

Programming language structure

```
x = 1
y = 0
check0 = 0


if (x):
if (y):
pass
else:
check0 = 1


print(check0)
```

How does Python handle this?

```
x = 1
y = 0
check0 = 0


if (x):
    if (y):
        pass
    else:
        check0 = 1


print(check0)
```

Invalid syntax, you need to indent, which makes it clear

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
```

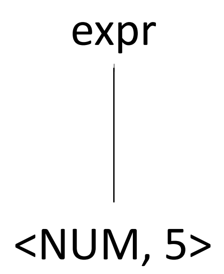
```
expr ::= NUM
     | expr PLUS expr
     | expr TIMES expr
     | LPAREN expr RPAREN
```

expr

Parse trees examples

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

```
input: 5
```



Parse trees examples

```
input: 5*6
```

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

Parse trees examples

```
input: 5*6
```

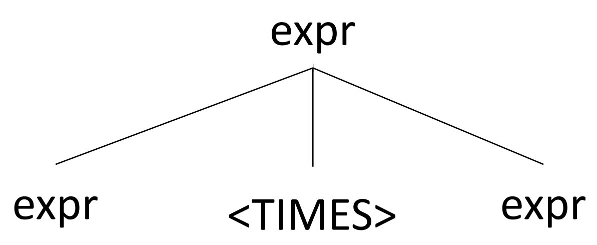
```
expr ::= NUM
     | expr PLUS expr
     | expr TIMES expr
     | LPAREN expr RPAREN
```

expr

Parse trees examples

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

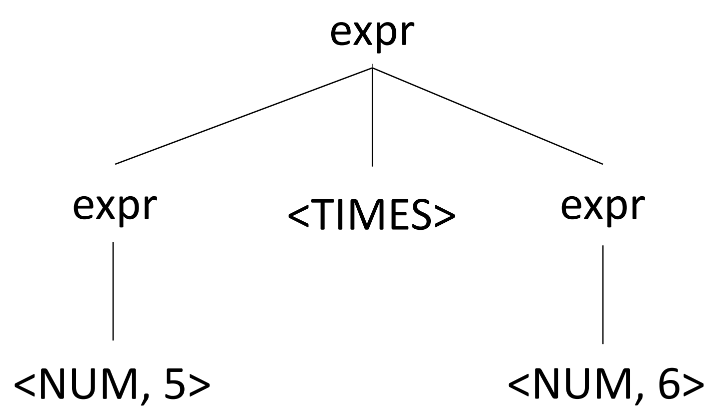
```
input: 5*6
```



Parse trees examples

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

```
input: 5*6
```



Parse trees examples

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

```
input: 5**6
```

What happens in an error?

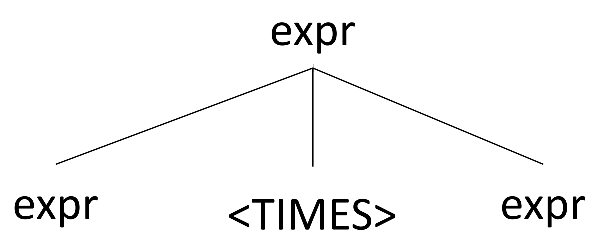
expr

Parse trees examples

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

```
input: 5**6
```

What happens in an error?

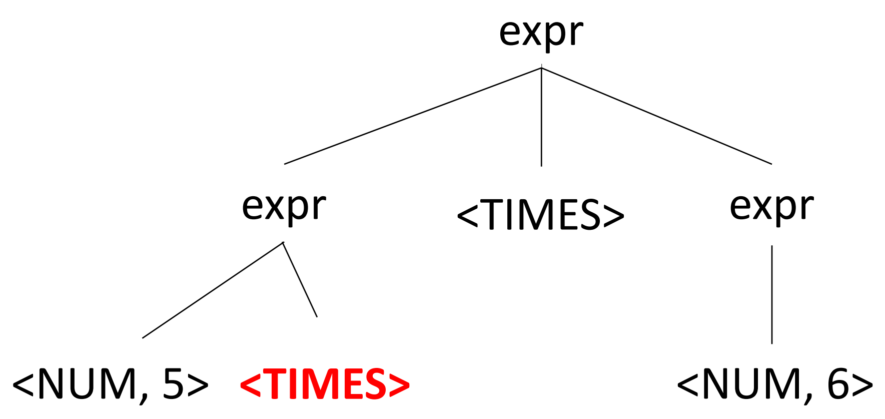


Parse trees examples

```
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
```

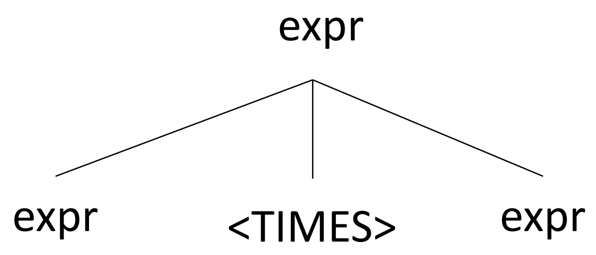
```
expr ::= NUM
      | expr PLUS expr
      | expr TIMES expr
      | LPAREN expr RPAREN
```

expr

Parse trees examples

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

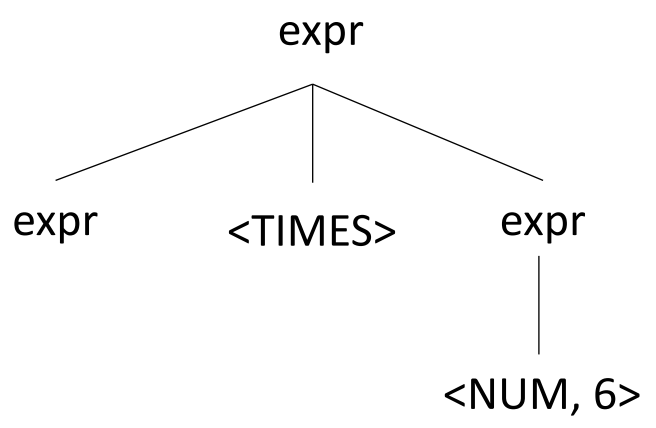
```
input: (1+5)*6
```



Parse trees examples

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

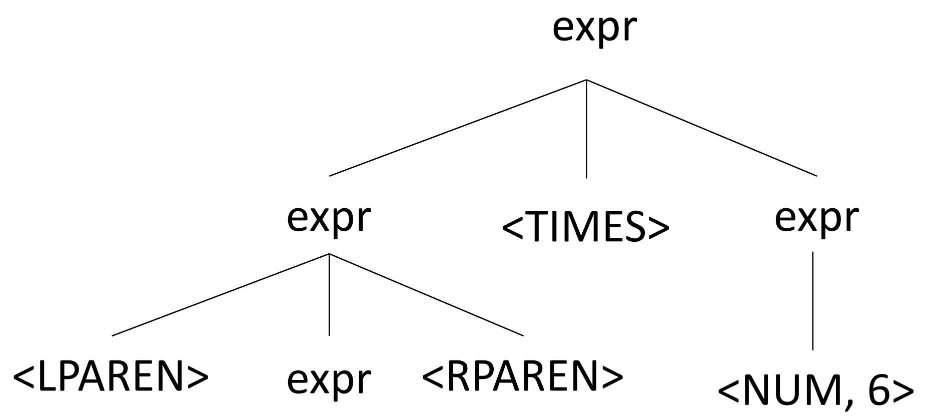
```
input: (1+5)*6
```



Parse trees examples

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

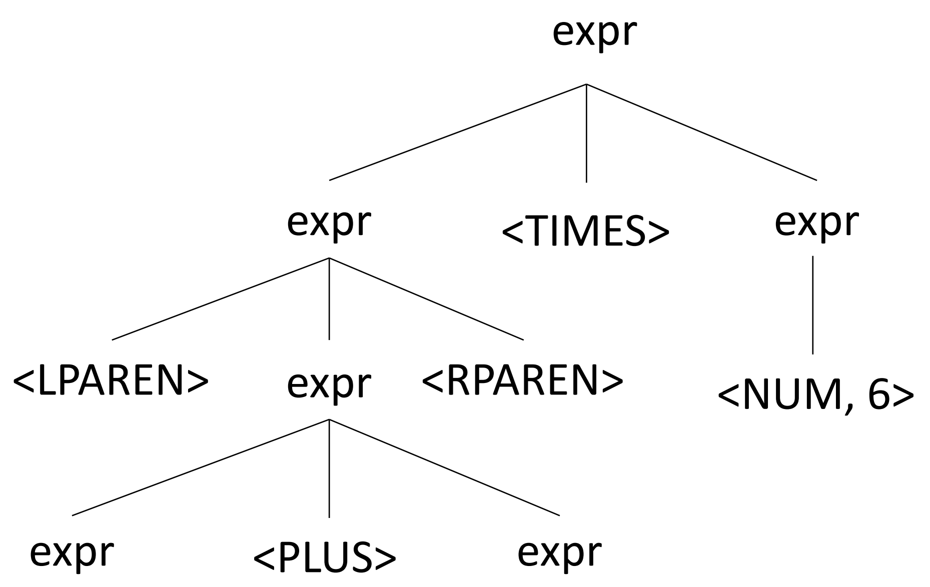
`input: (1+5)*6`



Parse trees examples

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

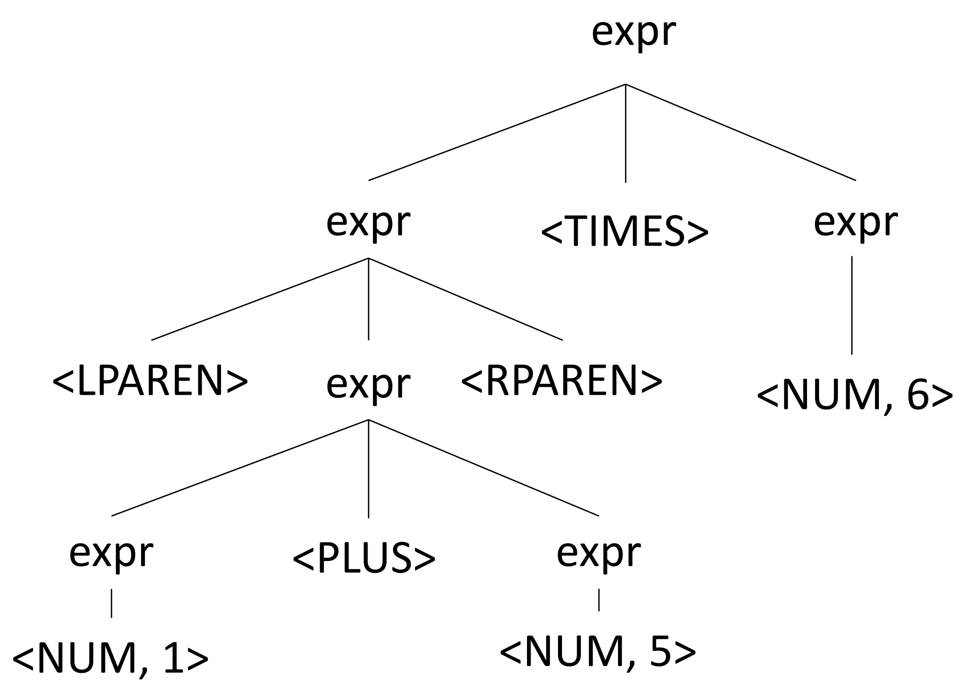
```
input: (1+5)*6
```



Parse trees examples

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

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

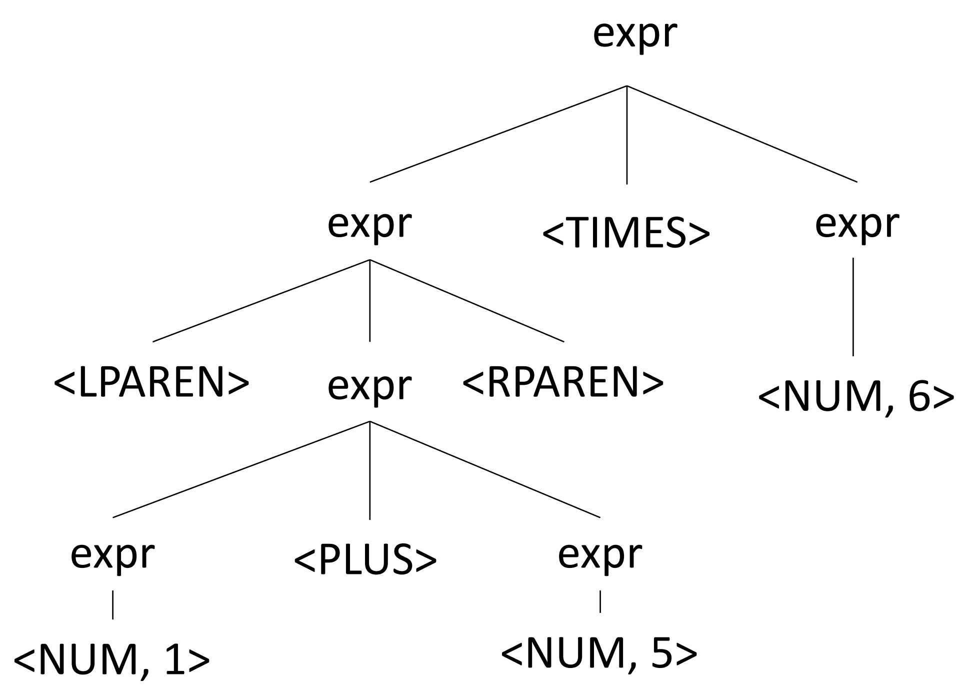


Parse trees examples

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

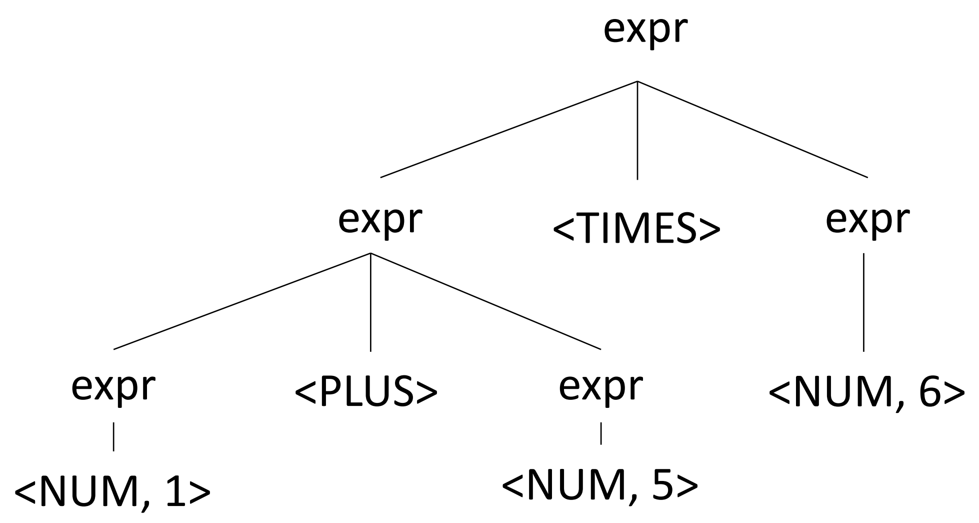
- How about : `1 + 5 * 6`

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

Parse trees

- How about: `1 + 5 * 6`

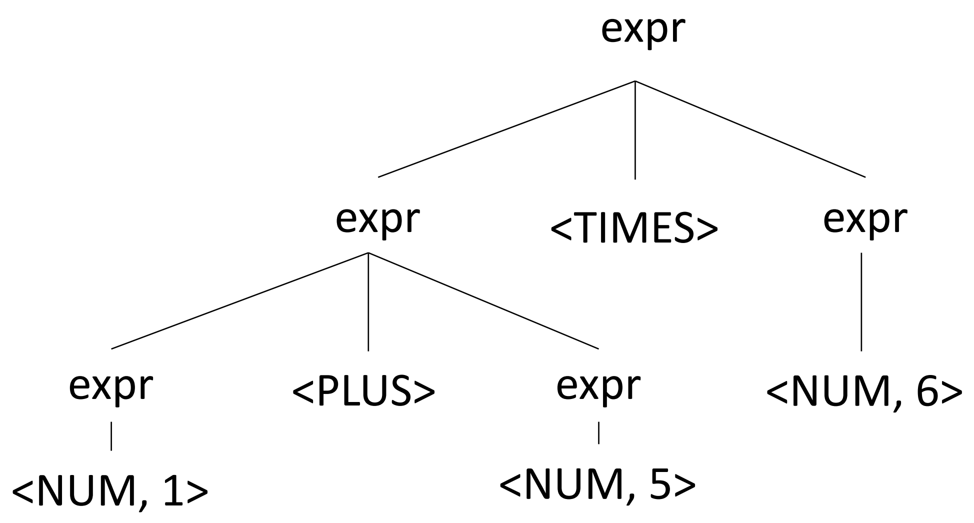
```
expr ::= NUM
     | expr PLUS expr
     | expr TIMES expr
     | LPAREN expr RPAREN
```



Ambiguous Grammars

- `input: 1 + 5 * 6`

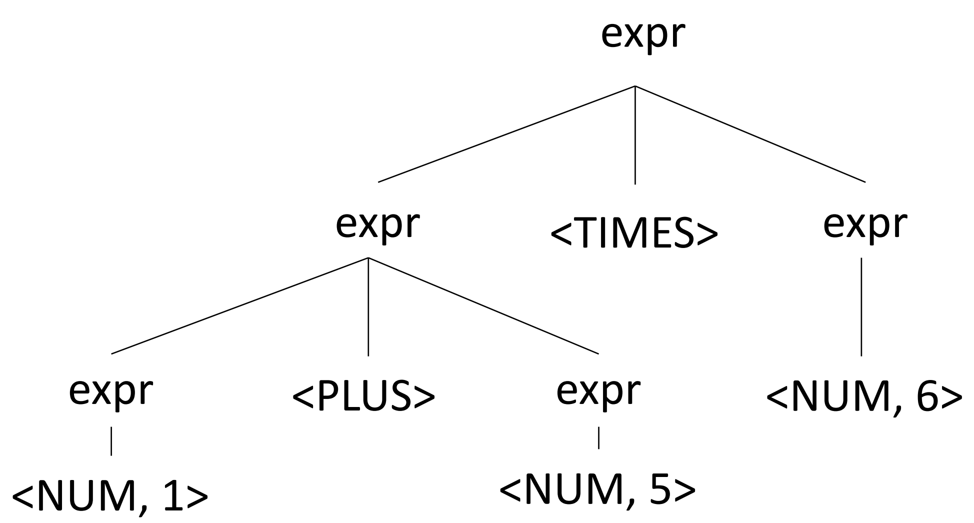
```
expr ::= NUM
       | expr PLUS expr
       | expr TIMES expr
       | LPAREN expr RPAREN
```

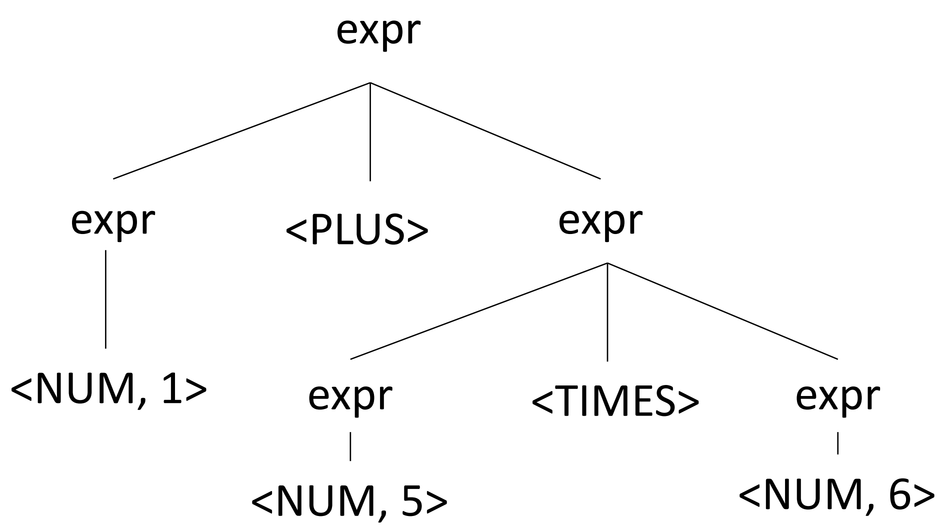


Ambiguous Grammars

- `input: 1 + 5 * 6`

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





Avoiding Ambiguity

- How to avoid ambiguity related to precedence?
- Define precedence into the grammar:
  - Ambiguity comes from conflicts. Explicitly define how to deal with conflicts by indicating that:

      * has higher precedence than +

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

Avoiding 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 Ambiguity

Second way: 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

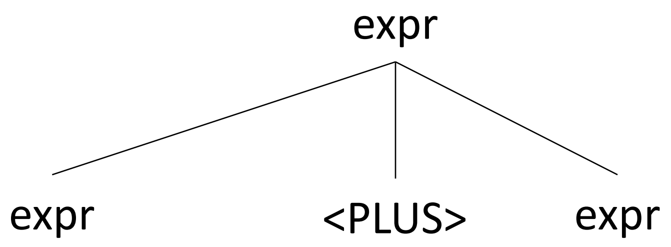
`input: 1+5*6`

Operator	Name	Productions

+	`expr`	`: expr PLUS expr`
		`| term`

*	`term`	`: term TIMES term`
		`| factor`

()	`factor`	`: LPAREN expr RPAREN`
		`| NUM`



Now lets create a parse tree

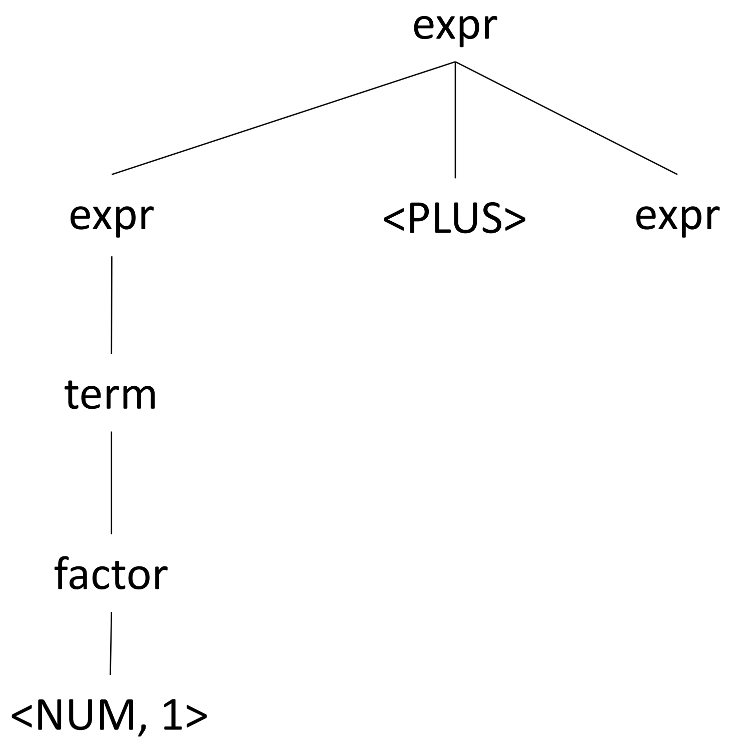
`input: 1+5*6`

Operator	Name	Productions

+	`expr`	`: expr PLUS expr`
		`| term`

*	`term`	`: term TIMES term`
		`| factor`

()	`factor`	`: LPAREN expr RPAREN`
		`| NUM`



Now lets create a parse tree

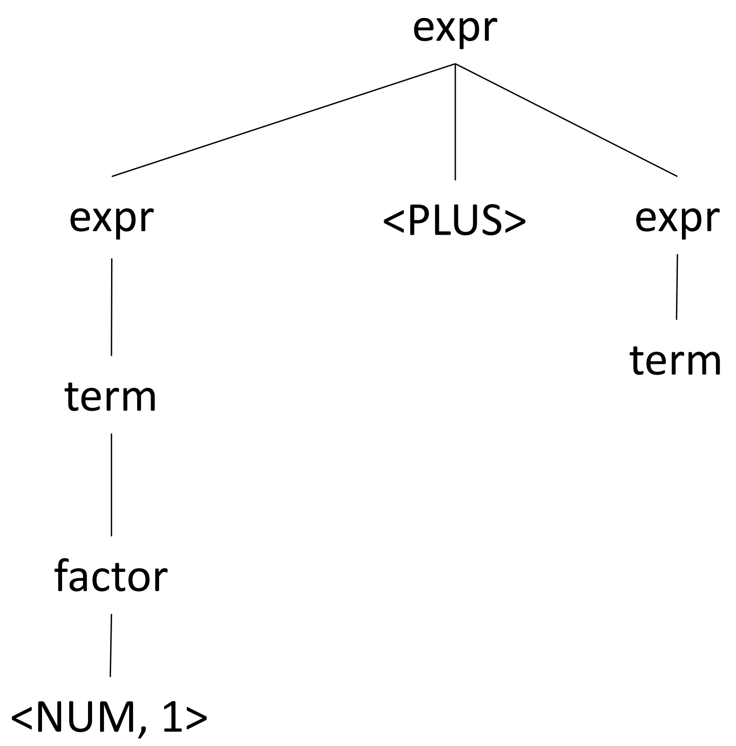
`input: 1+5*6`

Operator	Name	Productions

+	`expr`	`: expr PLUS expr`
		`| term`

*	`term`	`: term TIMES term`
		`| factor`

()	`factor`	`: LPAREN expr RPAREN`
		`| NUM`



Now lets create a parse tree

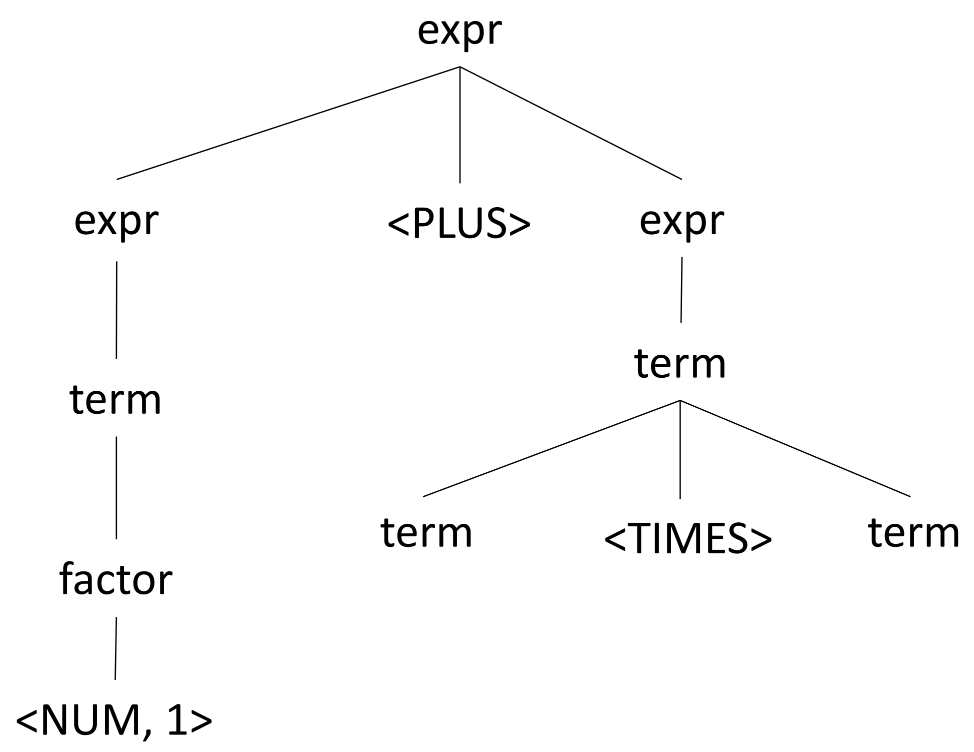
`input: 1+5*6`

Operator	Name	Productions

+	`expr`	`: expr PLUS expr`
		`| term`

*	`term`	`: term TIMES term`
		`| factor`

()	`factor`	`: LPAREN expr RPAREN`
		`| NUM`



Now lets create a parse tree

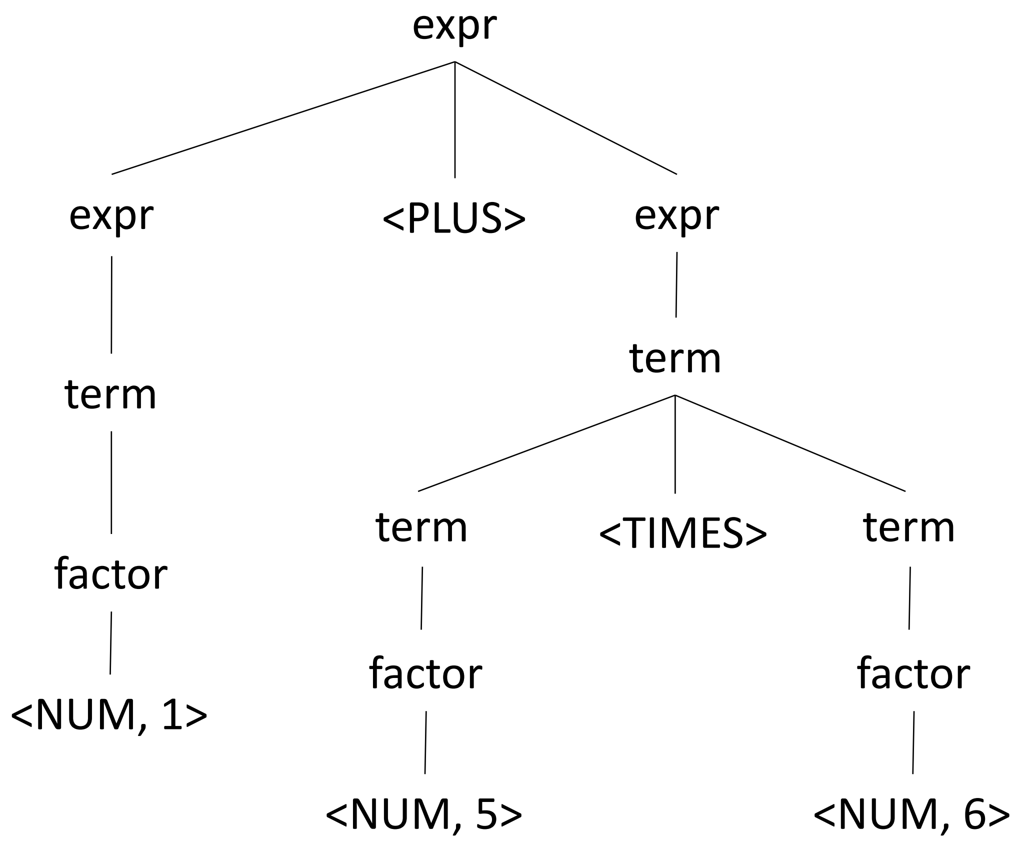
`input: 1+5*6`

Operator	Name	Productions

+	`expr`	`: expr PLUS expr`
		 `| term`

*	`term`	`: term TIMES term`
		 `| factor`

()	`factor`	`: LPAREN expr RPAREN`
		 `| NUM`



Parsing Regular Expressions

(and considering precedence)

Let’s try it for an RE language, {| . * ()}

- Assume . is a concatenation operator
- Terminals are in upper-case

Operator	**Name (LHS)**	**Productions (RHS)**

|	`choice`	`choice PIPE choice`
		`concat`

.	`concat`	`concat DOT concat`
		`star`

*	`star`	`star STAR`
		`unit`

()	`unit`	`LPAR choice RPAR`
		`CHAR`

Parsing Regular Expressions

(and considering precedence)

Let’s try it for an RE language, {| . * ()}

- Assume . is a concatenation operator
- Terminals are in upper-case

Operator	Name	Productions
|	`choice`	`: choice PIPE choice` `| concat`
.	`concat`	`: concat DOT concat` `| starred`
*	`starred`	`: starred STAR` `| unit`
()	`unit`	`: LPAREN choice RPAREN` `| CHAR`

Parsing Regular Expressions

(and considering precedence)

Let's try it for an RE language, {| . * ()}

- Assume . is a concatenation operator
- Terminals are in upper-case

```
input: a.b | c*
```

Operator	Name	Productions

|	`choice`	`: choice PIPE choice`
		`| concat`

.	`concat`	`: concat DOT concat`
		`| starred`

*	`starred`	`: starred STAR`
		`| unit`

()	`unit`	`: LPAREN choice RPAREN`
		`| CHAR`

Parsing Regular Expressions

(and considering precedence)

Let’s try it for an RE language, {| . * ()}

- Assume . is a concatenation operator
- Terminals are in upper-case

Operator	**Name**	**Productions**

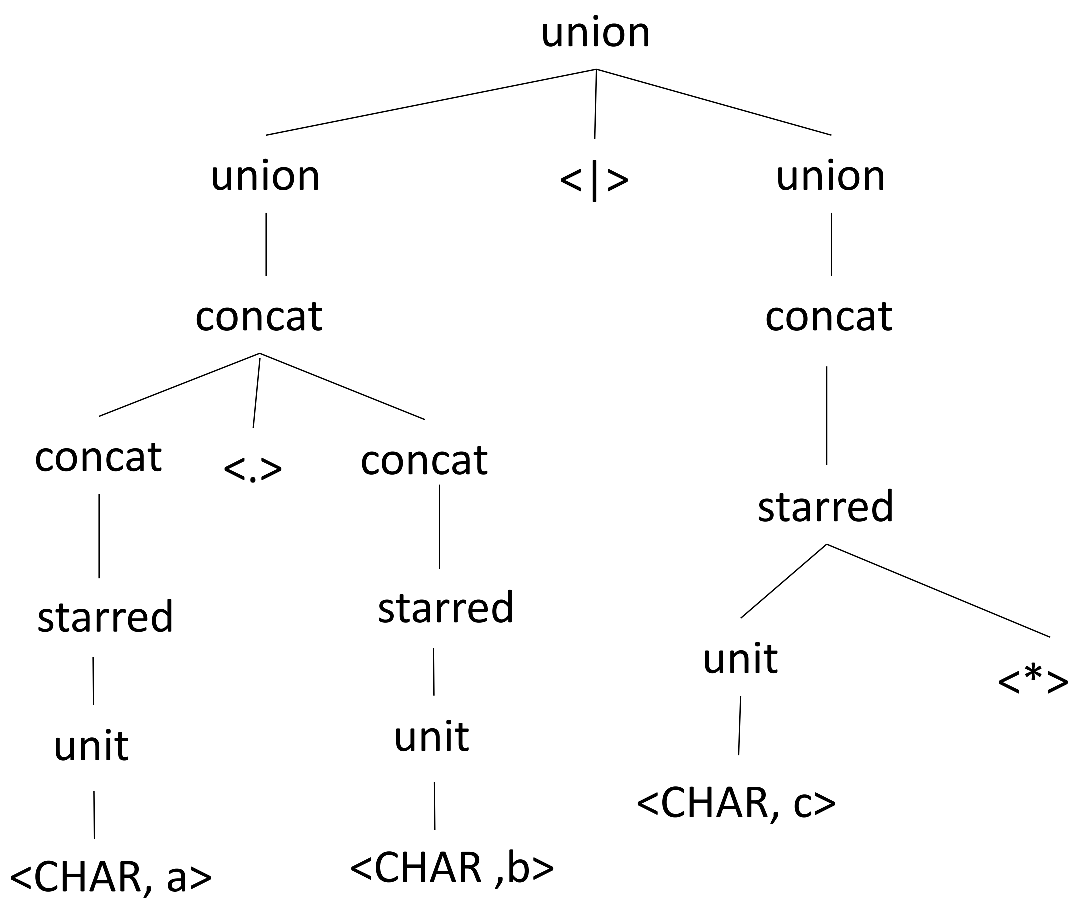
|	`choice`	`: choice PIPE choice`
		`| concat`

.	`concat`	`: concat DOT concat`
		`| starred`

*	`starred`	`: starred STAR`
		`| unit`

()	`unit`	`: LPAREN choice RPAREN`
		`| CHAR`

`input: a.b | c*`



How many levels of precedence does C have?

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

Have we removed all ambiguity?

Let’s make some more parse trees

`input: 2+3+4`

Operator	Name	Productions

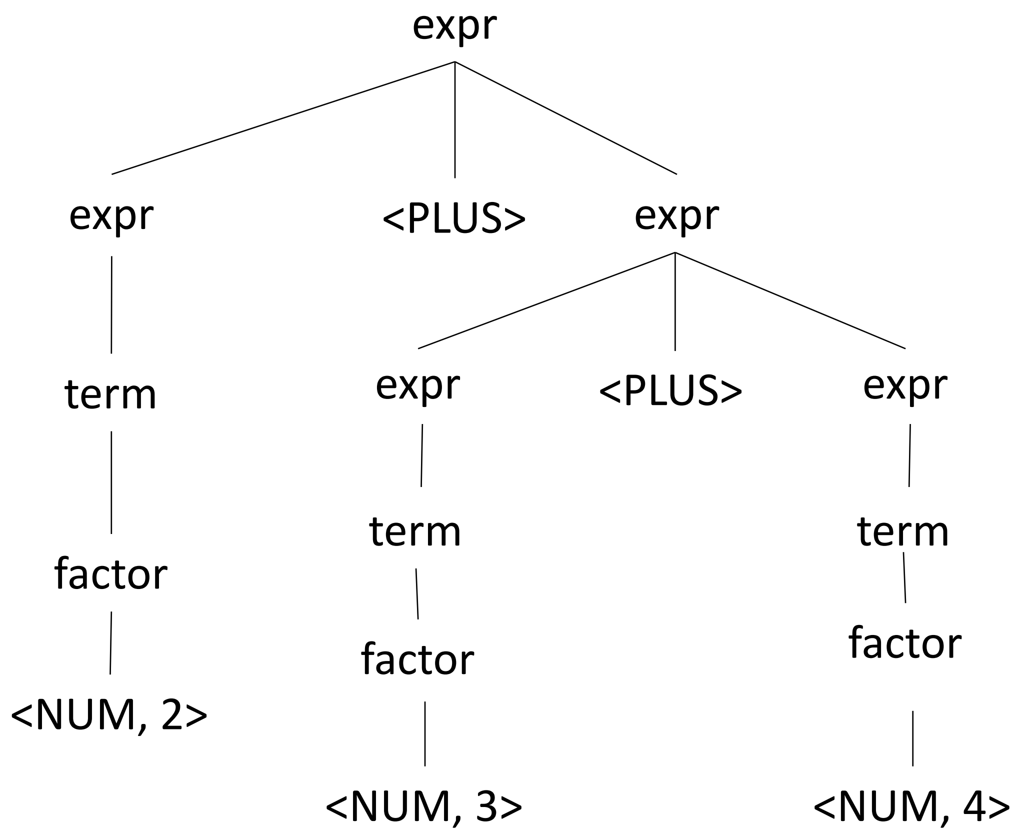
+	`expr`	`: expr PLUS expr`
		`| term`

*	`term`	`: term TIMES term`
		`| factor`

()	`factor`	`: LP expr RP`
		`| NUM`

Let’s make some more parse trees

`input: 2+3+4`



Operator	**Name**	**Productions**

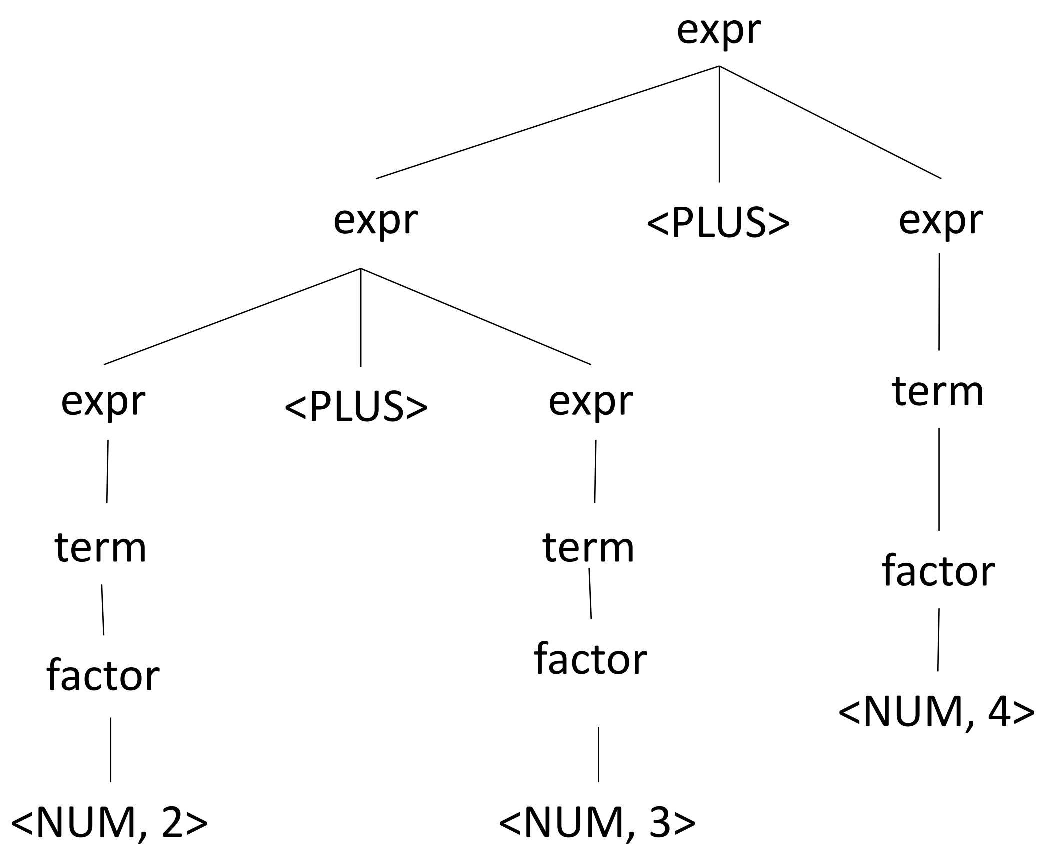
+	`expr`	`: expr PLUS expr`
		`| term`

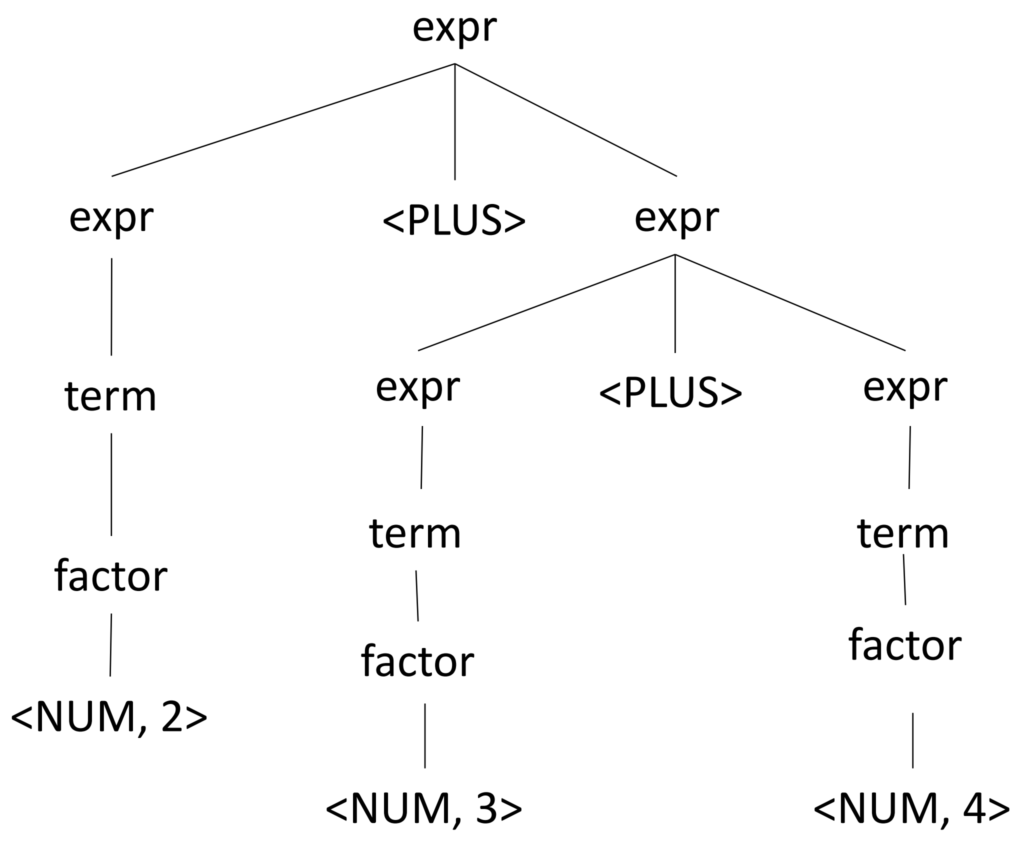
*	`term`	`: term TIMES term`
		`| factor`

()	`factor`	`: LP expr RP`
		`| NUM`

This is ambiguous, is it an issue?

`input: 2+3+4`



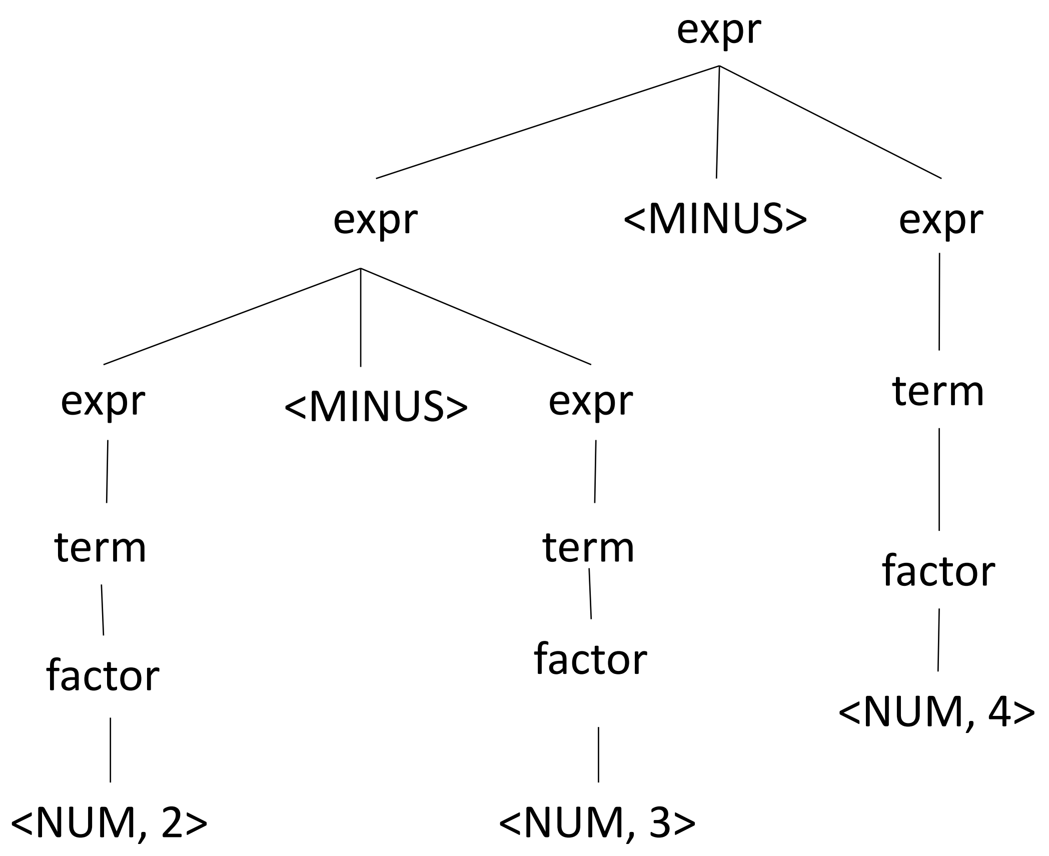


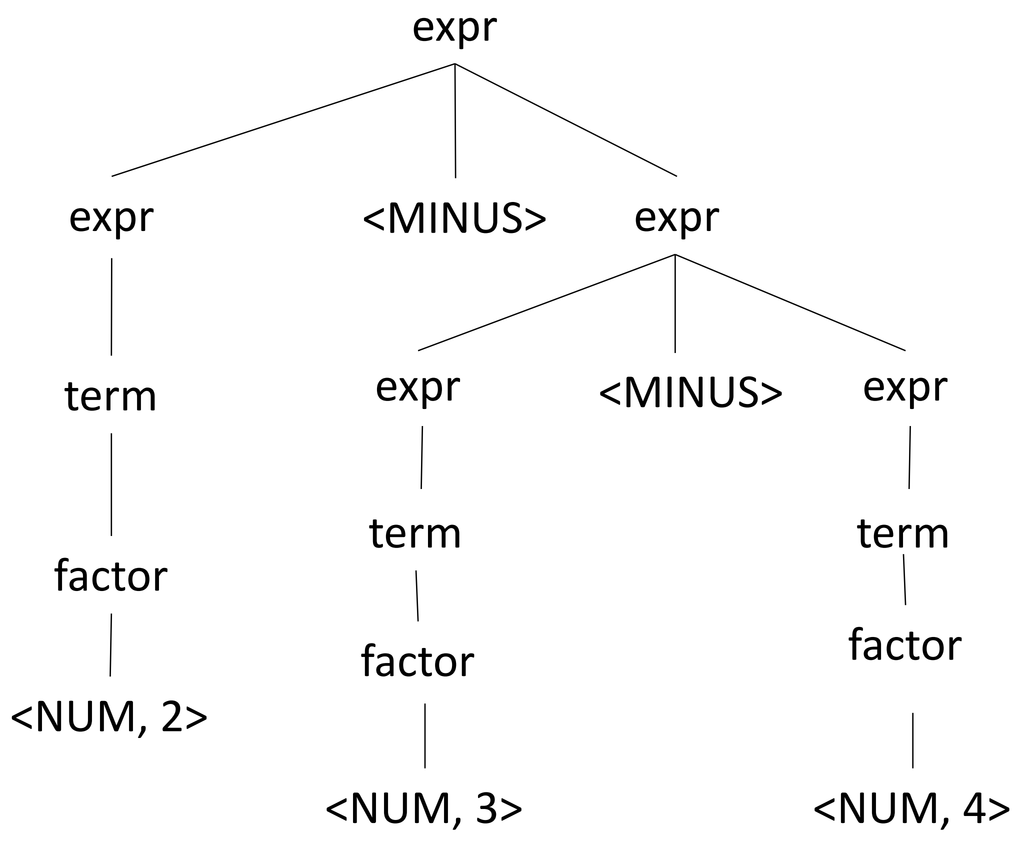
What about for a different operator?

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

What about for a different operator?

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





Which one is right?

Associativity

Describes the order in which apply the same operator

Sometimes it doesn’t matter:

- When?

Associativity

Describes the order in which apply the same operator

Sometimes it doesn't matter:

- Integer arithmetic
- Integer multiplication

These operators are said to be associative

Good test:

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

What about floating-point arithmetic?

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)
  - 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) allow associativity specification through keywords:
  - “+”: left, “^”: right

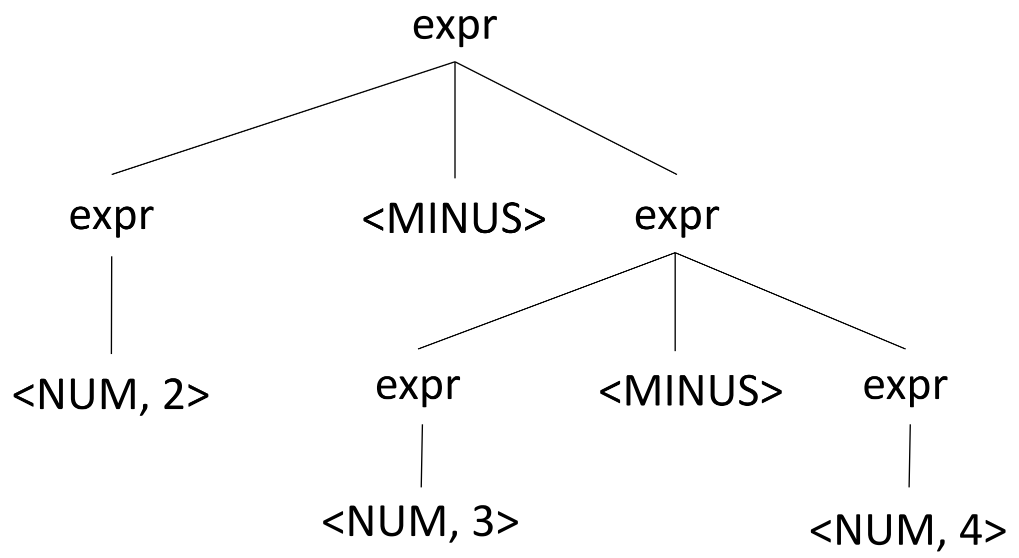
- Like precedence, we can also encode it into the production rules

Ambiguous associativity for a single operator

Operator	Name	Productions

-	`expr`	`: expr MINUS expr`
		`| NUM`

`input: 2-3-4`

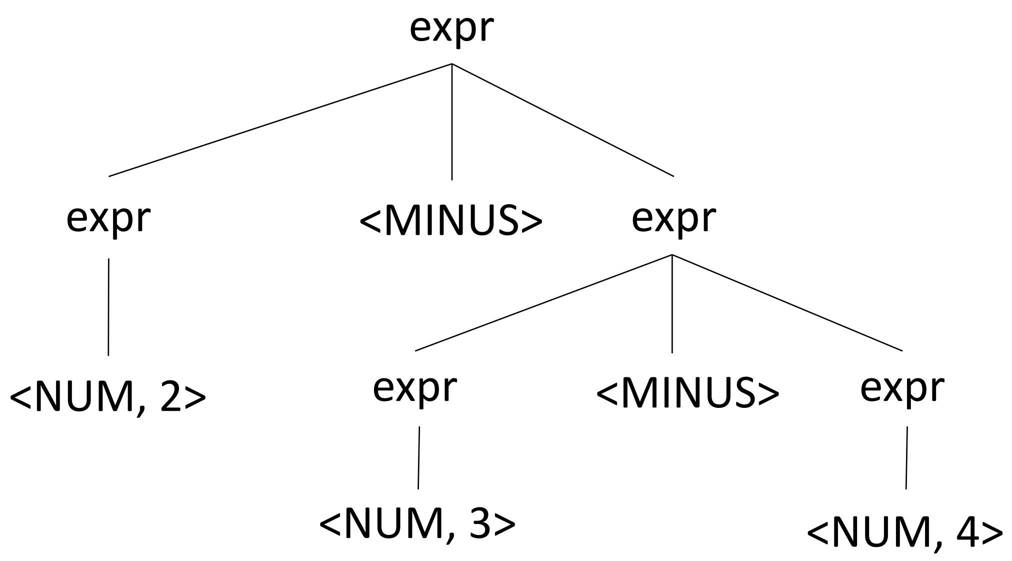


Associativity for a single operator

Operator	Name	Productions

-	`expr`	`: expr MINUS NUM`
		`| NUM`

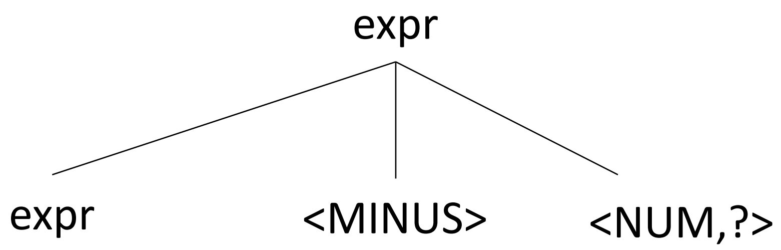
`input: 2-3-4`



No longer allowed

Associativity for a single operator

`input: 2-3-4`



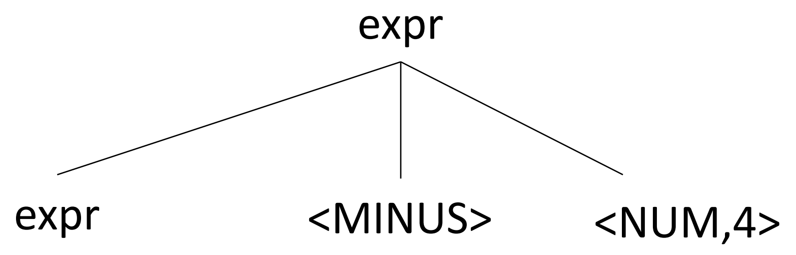
Operator	Name	Productions

-	`expr`	`: expr MINUS NUM`
		`| NUM`

Lets start over

Associativity for a single operator

`input: 2-3-4`



Operator	Name	Productions

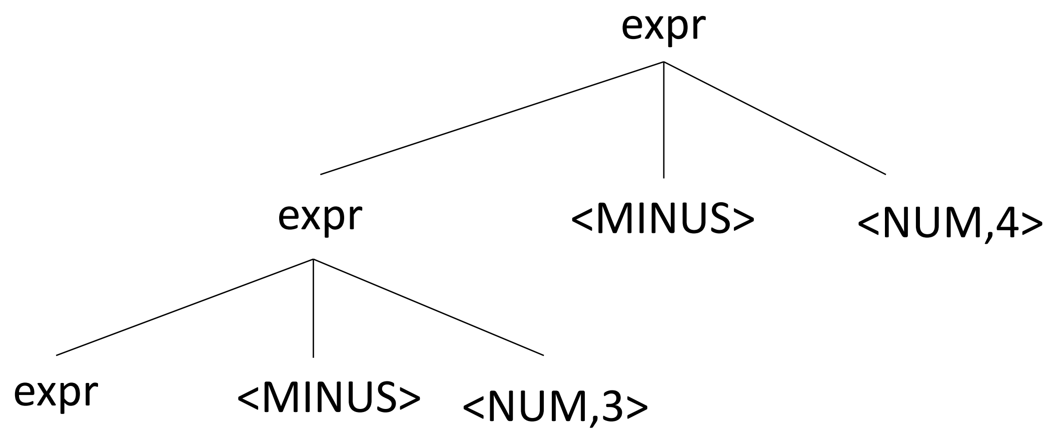
-	`expr`	`: expr MINUS NUM`
		`| NUM`

Associativity for a single operator

Operator	Name	Productions

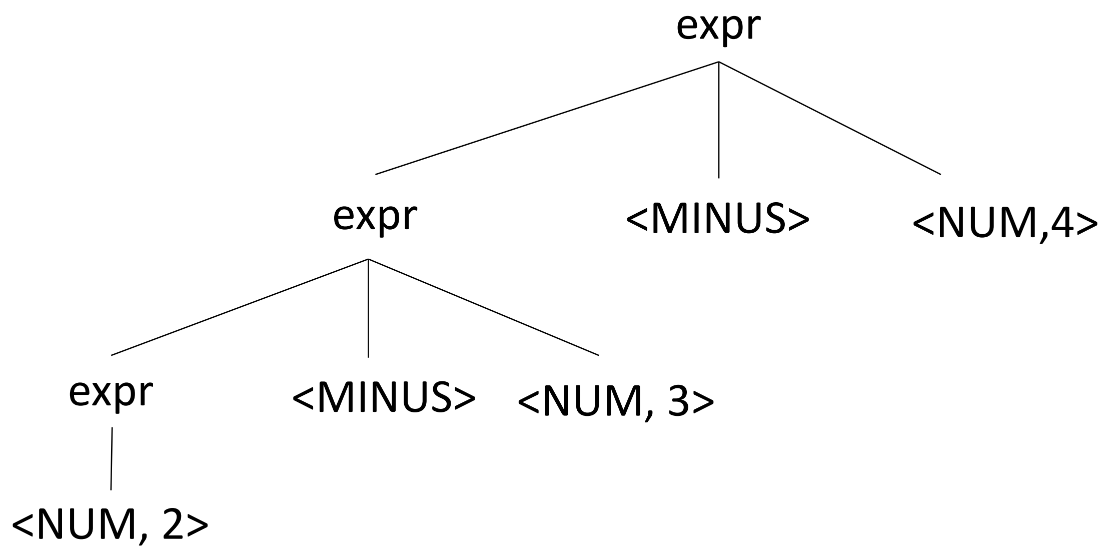
-	`expr`	`: expr MINUS NUM`
		`| NUM`

`input: 2-3-4`



Associativity for a single operator

`input: 2-3-4`



Operator	Name	Productions

-	`expr`	`: expr MINUS NUM`
		`| NUM`

Left associative:
No longer ambiguous

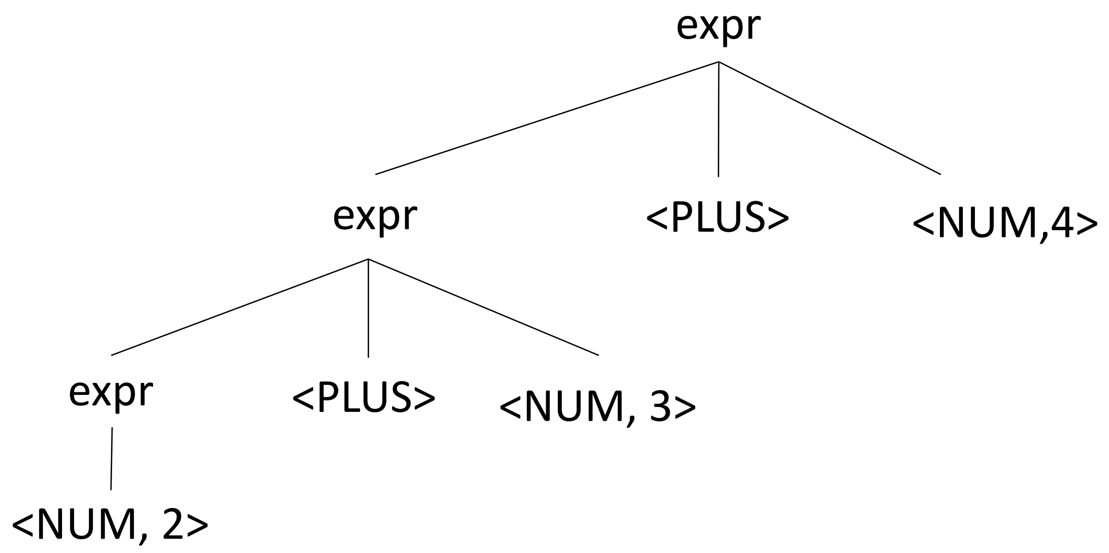
Should you have associativity when its not required?

Benefits?
Drawbacks?

Operator	Name	Productions

+	`expr`	`: expr PLUS NUM`
		`| NUM`

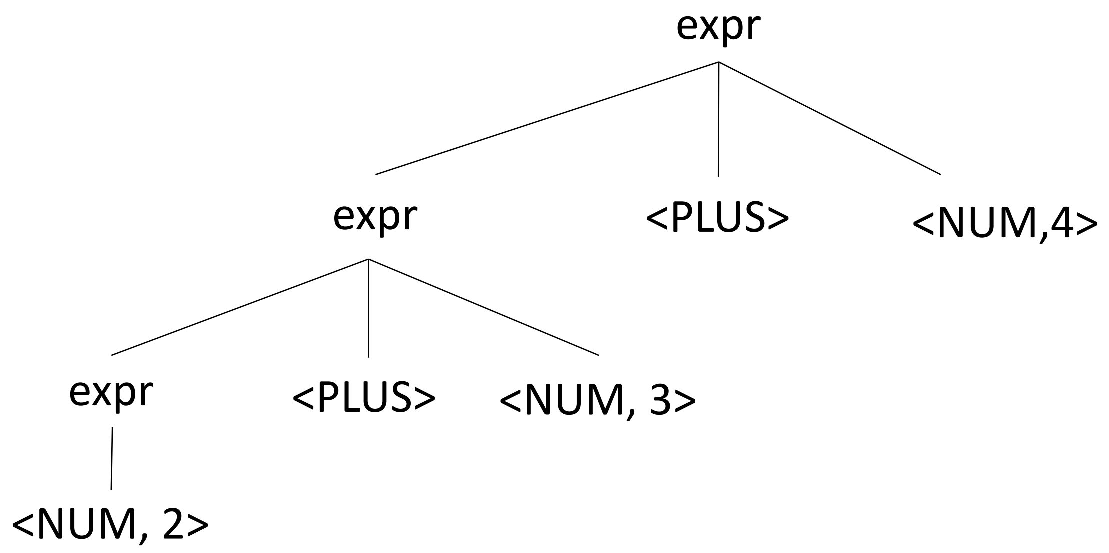
`input: 2+3+4`



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 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 associativities does C have?

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

Next time: algorithms for syntactic analysis

- Top down parsing
  - oracle parsing
  - removing left recursion
  - constructing lookahead sets