

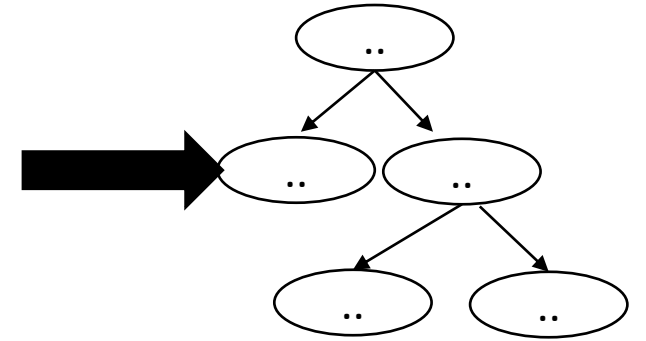
# CSE110A: Compilers

April 19, 2023

## Topics:

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

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



# Announcements

- HW 1 is out
  - Due next Monday at Midnight
- Plenty of help available
  - I have office hours tomorrow
  - TAs and tutors have office hours on Friday
  - ***No guaranteed help over the weekend or off business hours***
- Thanks to those who are asking/answering questions on Piazza
  - And especially for finding bugs in the HW!
  - No need to worry about line numbers or submit branches

# Quiz

*With the discussions, we didn't get through all the material to answer all the quiz questions, but we will today*

# Quiz

A production rule consists of:

---

☐ Terminals

---

☐ Regular Expressions

---

☐ Non-terminals

---

☐ function calls

# Context-free grammar

We will use *Backus–Naur form* (BNF) form

- Production rules contain a sequence of either non-terminals or terminals
- In our class, terminals will either be string constants or tokens
- Traditionally tokens will be all caps.

Examples:

```
add_expr ::= NUM '+' NUM
```

```
mult_expr ::= NUM '*' NUM
```

```
joint_expr ::= add_expr '*' add_expr
```

```
simple_expr ::= NUM '+' NUM  
            | NUM '*' NUM
```

# Quiz

There are certain patterns that regular expressions can express that context-free grammars cannot express. But it is not an issue because those patterns do not show up in practice

---

☐ True

---

☐ False

# Any RE can be expressed in BNF

- We just need to show fundamental operators
  - concat, choice, star

# Any RE can be expressed in BNF

- We just need to show fundamental operators
  - concat, choice, star

`add_expr ::= NUM '+' NUM`



# Any RE can be expressed in BNF

- We just need to show fundamental operators
  - concat, choice, star

```
simple_expr ::= NUM '+' NUM  
           | NUM '*' NUM
```

# Any RE can be expressed in BNF

- We just need to show fundamental operators
  - concat, choice, star

*How to express “a\*” in BNF?*

`a_star ::= ?`

# Any RE can be expressed in BNF

- We just need to show fundamental operators
  - concat, choice, **star**

*How to express "a\*" in BNF?*

```
a_star ::= ""  
         |  "a"  
         |  "a" a_star
```

# Quiz

a left derivation will always produce the same parse tree as a right derivation

☐ True

☐ False

*We didn't get this far in the lecture*

# Quiz

Different programming languages make structure more or less explicit, e.g. using ()s and {}s.

Write a few sentences on any programming language experience you have w.r.t. structure and how you use it. For example do you use {}s when you write if statements, even if they contain a single statement? Why or Why not? Do you think Python's use of whitespace is a good construct for structure? Have you ever used [S expressions](#) in a Lisp language?

# Programming language structure

```
if (x) {  
    my_var++;  
}
```

vs.

```
if (x)  
    my_var++;
```

*Should conditionals require braces?*

5 + 6 \* 3

vs.

5 + (6 \* 3)

*should expressions require parenthesis?*

(+ 5 (\* 6 3))

vs.

(+ 5 (\* 6 3))

*Do expressions (lisp) require explicit structure*

***What are pros and cons of each?***

# Programming language structure

```
int x = 1;  
int y = 0;  
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

# Review

- How to derive strings from a context free grammar

# Deriving strings

*Give each production rule a numeric id*

```
1: SheepNoise ::= 'baa' SheepNoise
2:              | 'baa'
```

RULE	Sentential Form
start	SheepNoise

RULE	Sentential Form
start	SheepNoise

# Deriving strings

*Give each production rule a numeric id*

```
1: SheepNoise ::= 'baa' SheepNoise
2:              | 'baa'
```

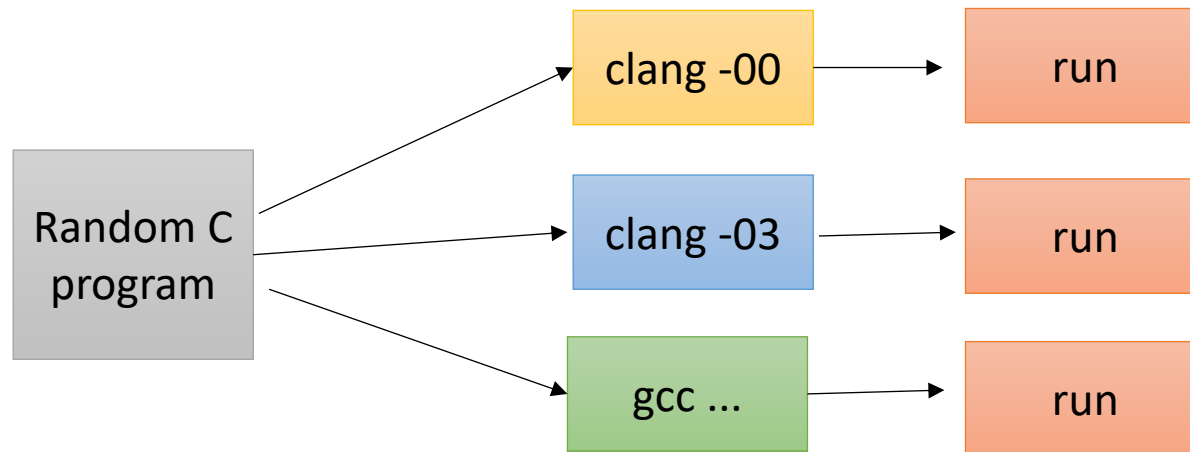
RULE	Sentential Form
start	SheepNoise
2	baa

RULE	Sentential Form
start	SheepNoise
1	baa SheepNoise
2	baa baa

Case study in string derivation: CSmith

# Case study in string derivation: CSmith

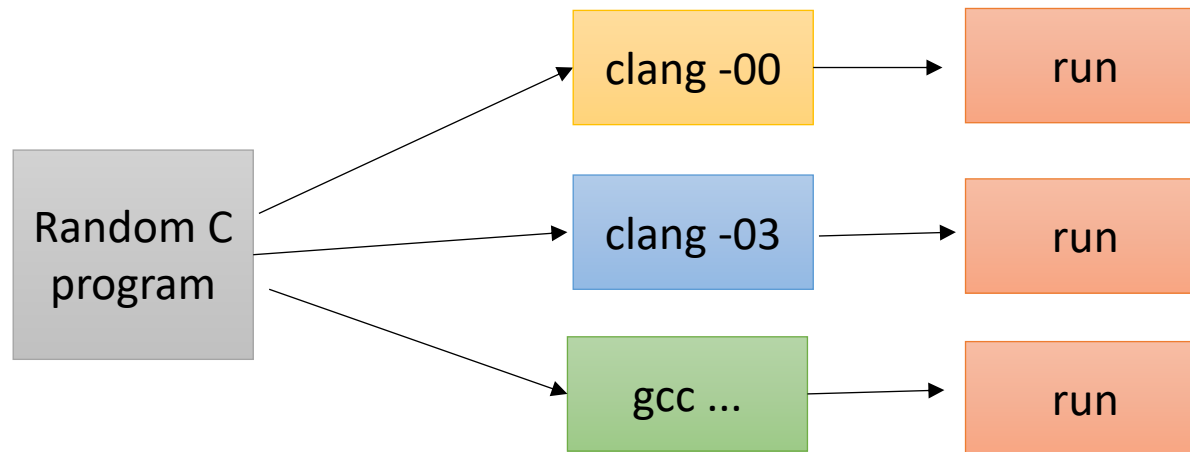
- Using a CFG, you can derive random strings in a language
- C-Smith
  - Generates random C programs
  - Used to test compiler correctness



*Check outcome. Is it the same?  
if not, then there is a bug in one  
of the compilers*

# Case study

- 400+ compiler bugs found
- Demo



*Check outcome. Is it the same?  
if not, then there is a bug in one  
of the compilers*

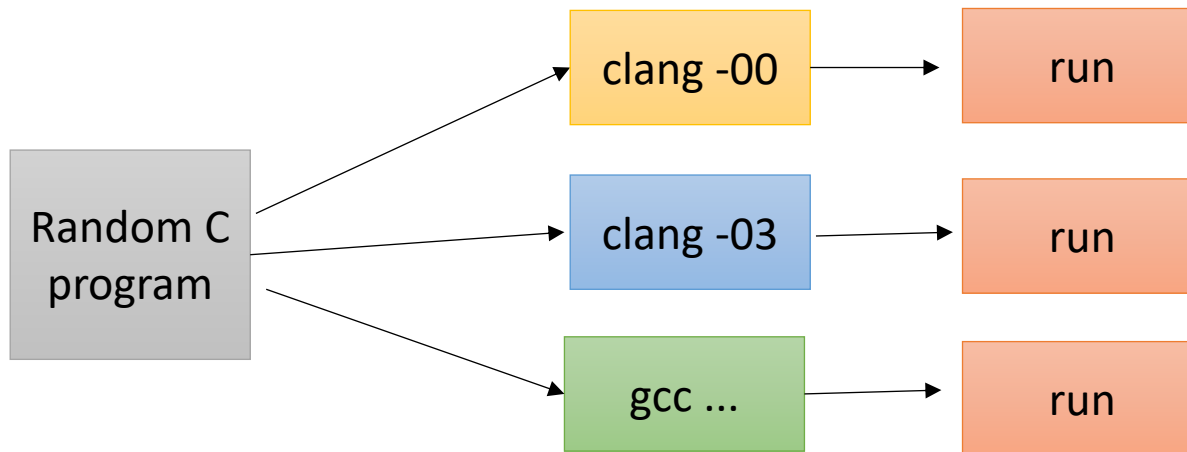
# Case study

- Big challenge: Undefined behavior
- Even though the program is syntactically valid, the behavior may be undefined

```
int main() {  
    int x;  
    printf("%d\n", x);  
    return 0;  
}
```

Uninitialized variables can return anything!

Use advanced compiler analysis to catch these issues



*Check outcome. Is it the same?  
if not, then there is a bug in one  
of the compilers*



# Moving on to new material

- We are going to start with the derivation we did at the end of class last time

# A more complicated derivation

```

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

```

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

[illegible]

# A more complicated derivation

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

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

RULE	Sentential Form
start	Expr
2	Expr Op ID
5	Expr * ID
1	(Expr) * ID
2	(Expr Op ID) * ID
4	(Expr + ID) * ID
3	(ID + ID) * ID

# A more complicated derivation

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

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

We can visualize this as a tree:

RULE	Sentential Form
start	Expr
2	Expr Op ID
5	Expr * ID
1	(Expr) * ID
2	(Expr Op ID) * ID
4	(Expr + ID) * ID
3	(ID + ID) * ID

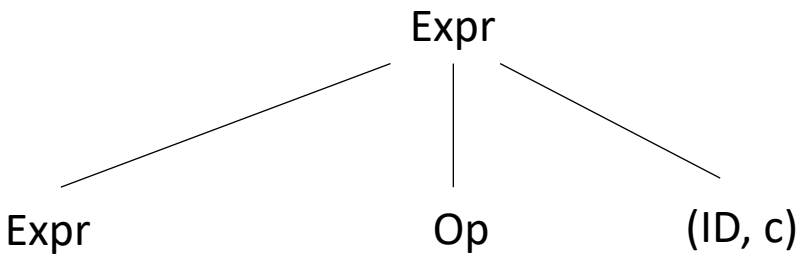
Expr

# A more complicated derivation

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

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

We can visualize this as a tree:



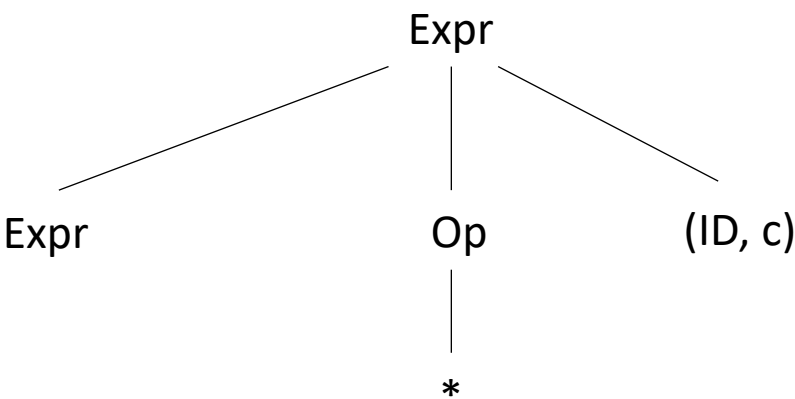
RULE	Sentential Form
start	Expr
2	Expr Op ID
5	Expr * ID
1	(Expr) * ID
2	(Expr Op ID) * ID
4	(Expr + ID) * ID
3	(ID + ID) * ID

# A more complicated derivation

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

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

We can visualize this as a tree:



RULE	Sentential Form
start	Expr
2	Expr Op ID
5	Expr * ID
1	(Expr) * ID
2	(Expr Op ID) * ID
4	(Expr + ID) * ID
3	(ID + ID) * ID

# A more complicated derivation

```

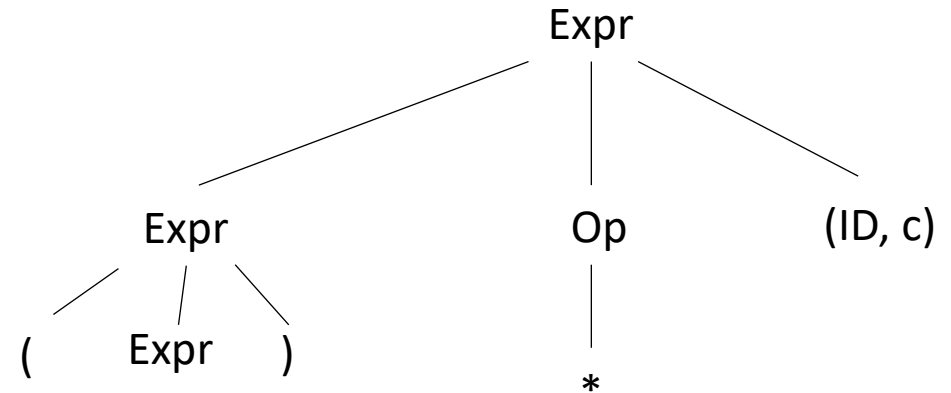
1: Expr ::= '(' Expr ')'
2:       | Expr Op ID
3:       | ID
4: Op ::= '+'
5:       | '*'

```

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

We can visualize this as a tree:

RULE	Sentential Form
start	Expr
2	Expr Op ID
5	Expr * ID
1	(Expr) * ID
2	(Expr Op ID) * ID
4	(Expr + ID) * ID
3	(ID + ID) * ID



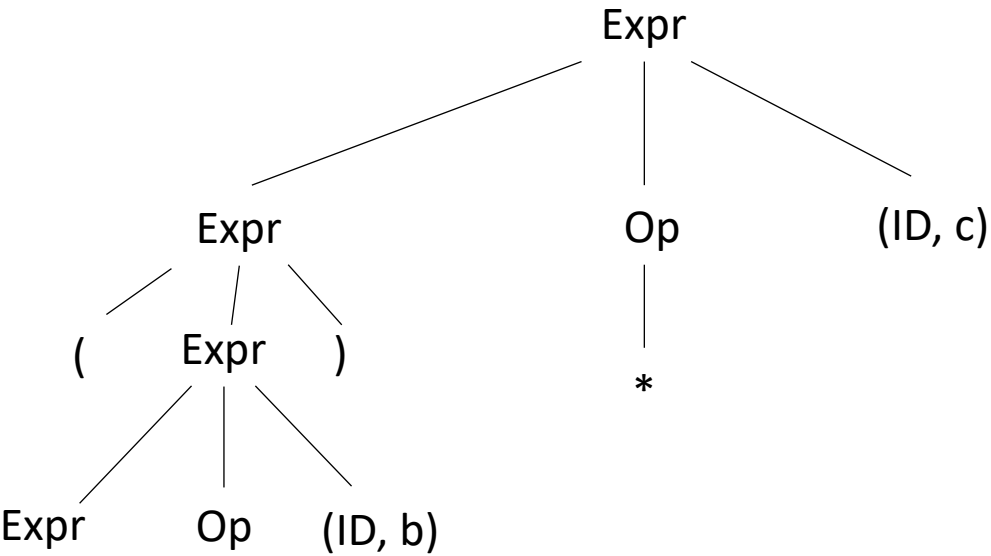
# A more complicated derivation

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

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

We can visualize this as a tree:

RULE	Sentential Form
start	Expr
2	Expr Op ID
5	Expr * ID
1	(Expr) * ID
2	(Expr Op ID) * ID
4	(Expr + ID) * ID
3	(ID + ID) * ID





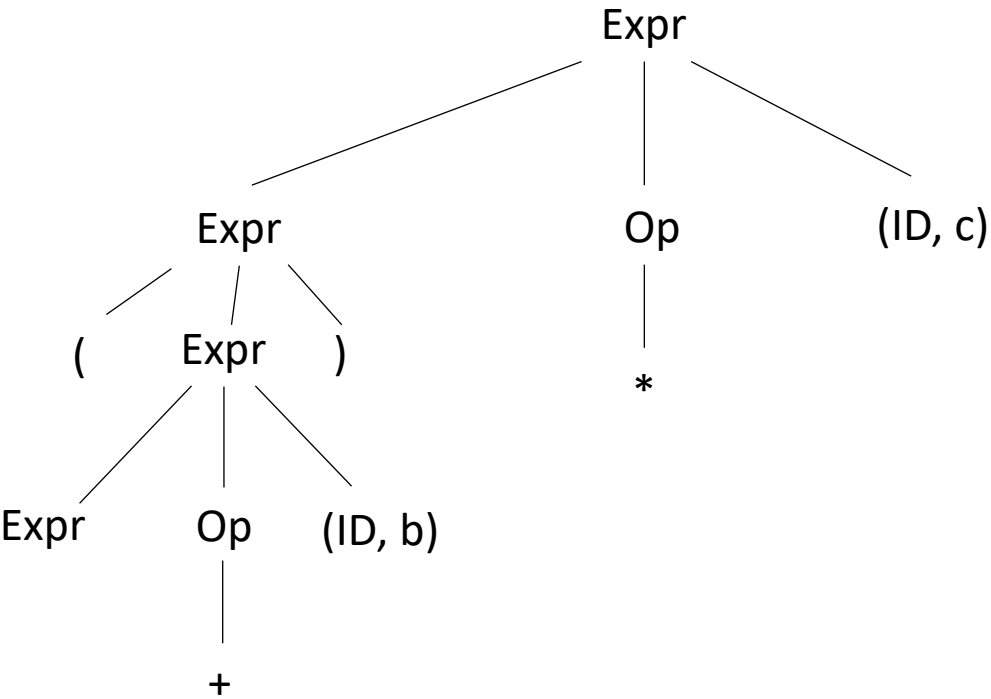
# A more complicated derivation

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

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

We can visualize this as a tree:

RULE	Sentential Form
start	Expr
2	Expr Op ID
5	Expr * ID
1	(Expr) * ID
2	(Expr Op ID) * ID
4	(Expr + ID) * ID
3	(ID + ID) * ID



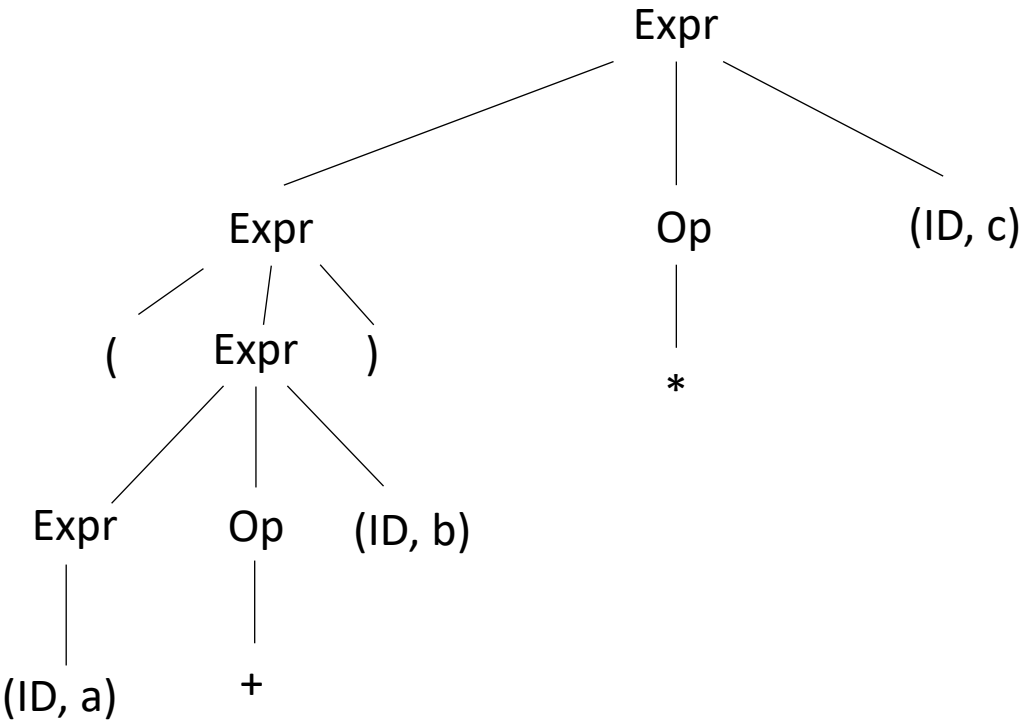
# A more complicated derivation

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

*Are there other ways to derive (a+b)\*c?*

We can visualize this as a tree:

RULE	Sentential Form
start	Expr
2	Expr Op ID
5	Expr * ID
1	(Expr) * ID
2	(Expr Op ID) * ID
4	(Expr + ID) * ID
3	(ID + ID) * ID



# A more complicated derivation

```

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

```

*Are there other ways to derive  $(a+b) * c$ ?*

RULE	Sentential Form
start	Expr
2	Expr Op ID
5	Expr * ID
1	(Expr) * ID
2	(Expr Op ID) * ID
4	(Expr + ID) * ID
3	(ID + ID) * ID

[illegible]

# A more complicated derivation

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

*Are there other ways to derive (a+b)\*c?*

RULE	Sentential Form
start	Expr
2	Expr Op ID
5	Expr * ID
1	(Expr) * ID
2	(Expr Op ID) * ID
4	(Expr + ID) * ID
3	(ID + ID) * ID

*right derivation*

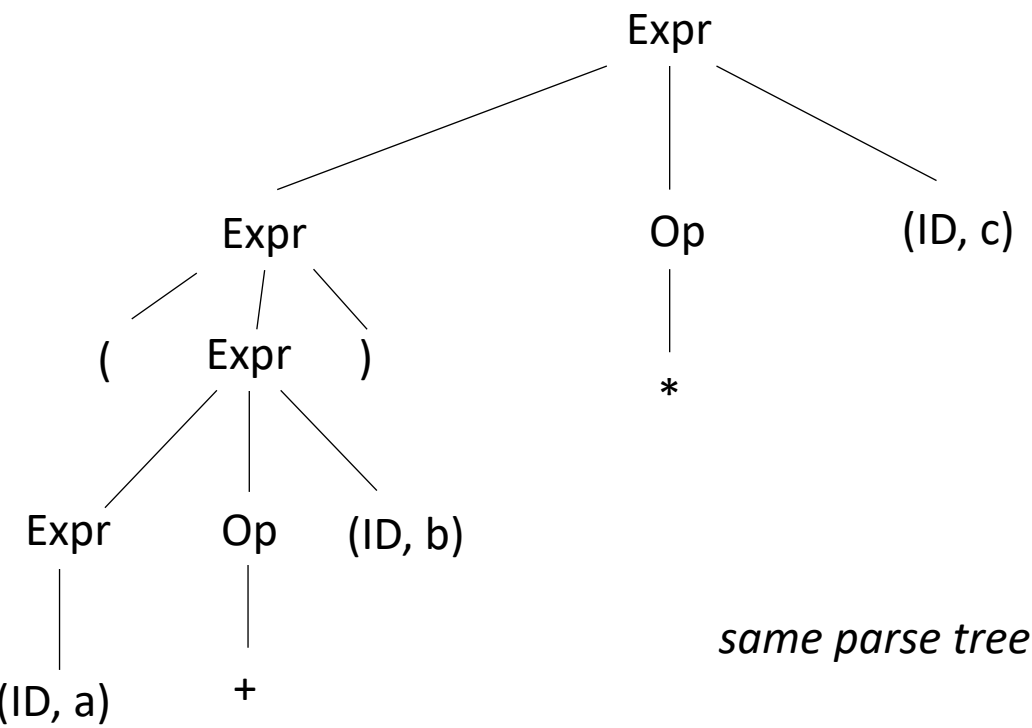
RULE	Sentential Form
start	Expr
2	Expr Op ID
1	(Expr) Op ID
2	(Expr Op ID) Op ID
3	(ID Op ID) Op ID
4	(ID + ID) Op ID
5	(ID + ID) + ID

*left derivation*

# A more complicated derivation

- 1: Expr ::= '(' Expr ')'
- 2:       | Expr Op ID
- 3:       | ID
- 4: Op ::= '+'
- 5:       | '\*'

Are there other ways to derive ( a+b ) \*c?



same parse tree

RULE	Sentential Form
start	Expr
2	Expr Op ID
1	(Expr) Op ID
2	(Expr Op ID) Op ID
3	(ID Op ID) Op ID
4	(ID + ID) Op ID
5	(ID + ID) + ID

left derivation

# 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:           | ....
```

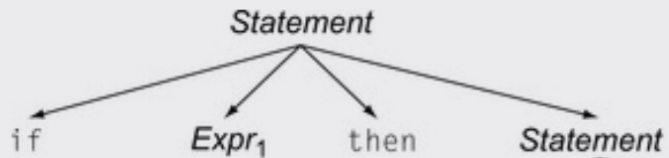
can we derive this string?

`if Expr1 then if Expr2 then Assignment1 else Assignment2`

# Ambiguous grammars

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

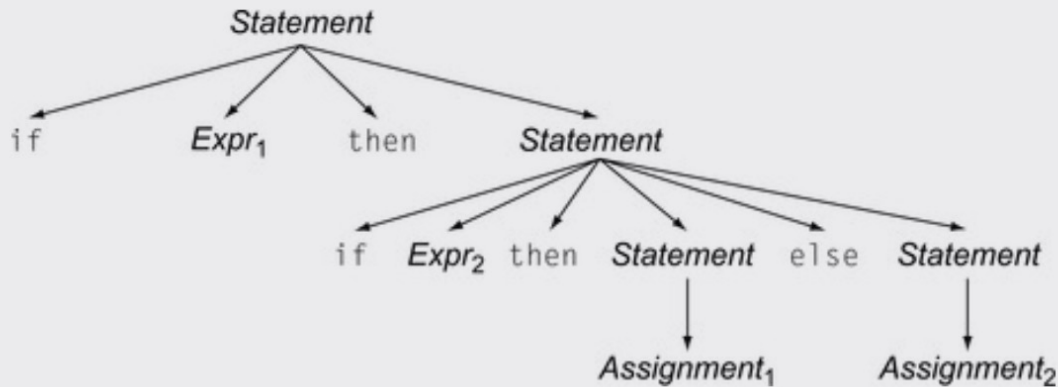
`if Expr1 then if Expr2 then Assignment1 else Assignment2`



# Ambiguous grammars

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

`if Expr1 then if Expr2 then Assignment1 else Assignment2`



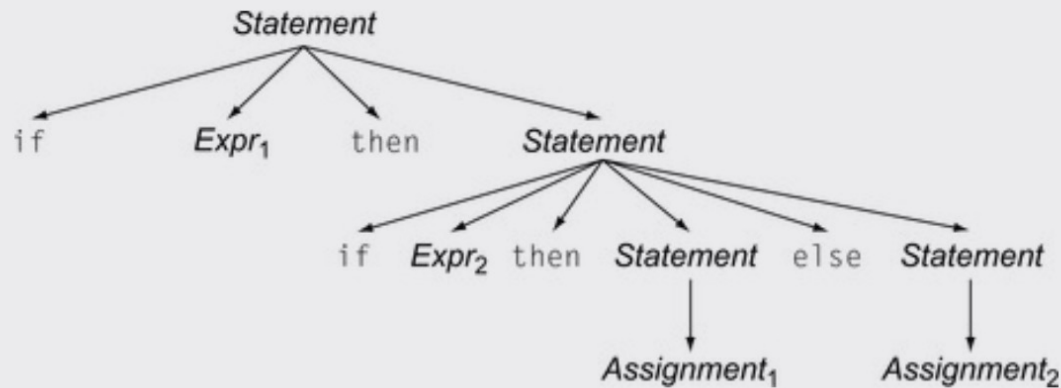
*Valid derivation*



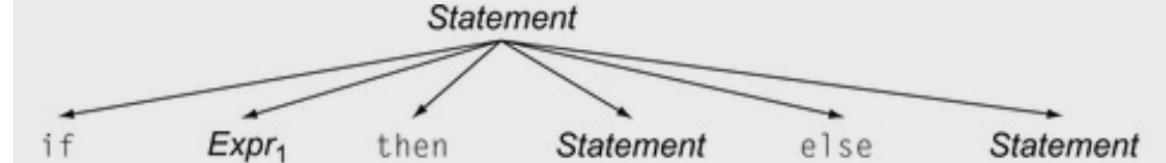
# Ambiguous grammars

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

*if* Expr<sub>1</sub> *then* *if* Expr<sub>2</sub> *then* Assignment<sub>1</sub> *else* Assignment<sub>2</sub>



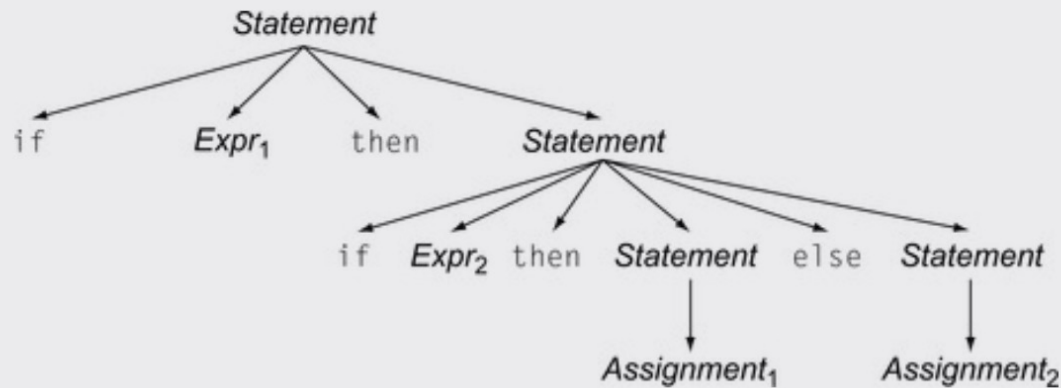
*Valid derivation*



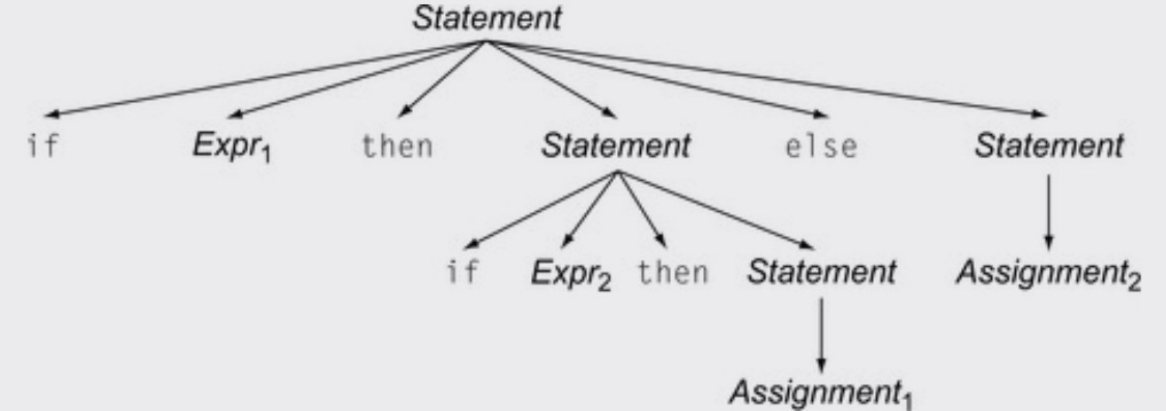
# Ambiguous grammars

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

*if* Expr<sub>1</sub> *then* *if* Expr<sub>2</sub> *then* Assignment<sub>1</sub> *else* Assignment<sub>2</sub>



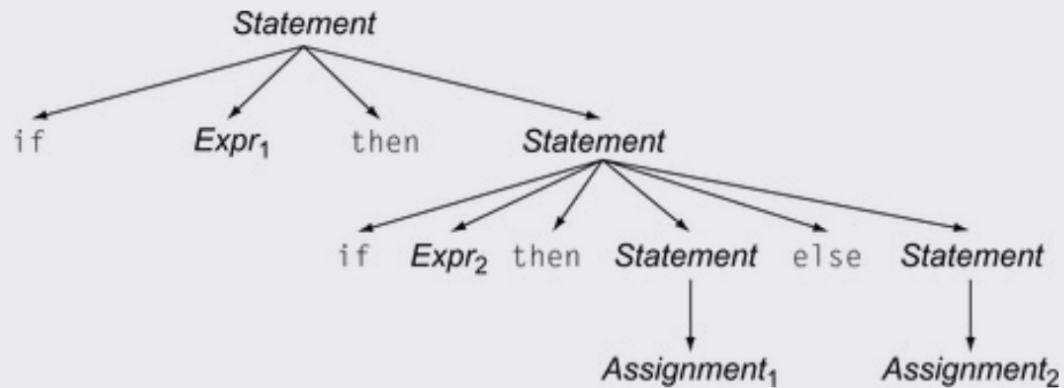
*Valid derivation*



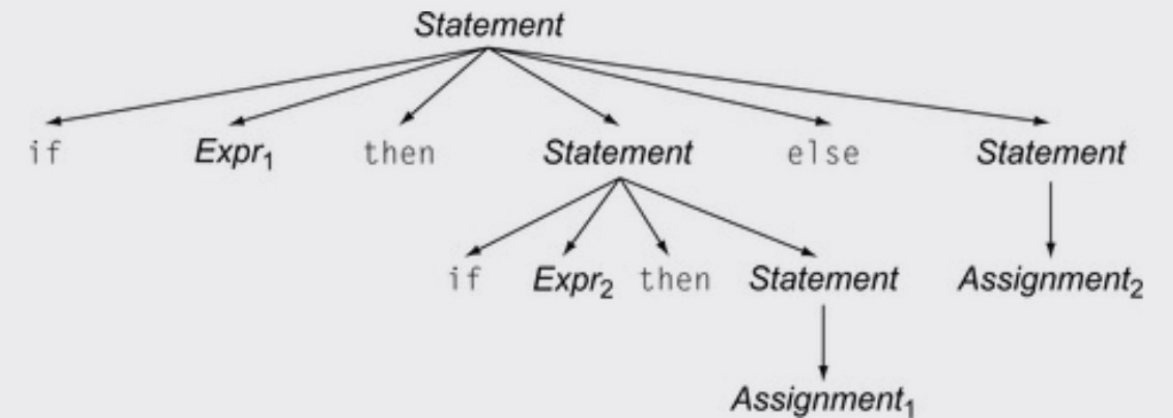
*Also a valid derivation*

# Ambiguous grammars

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



*Valid derivation*

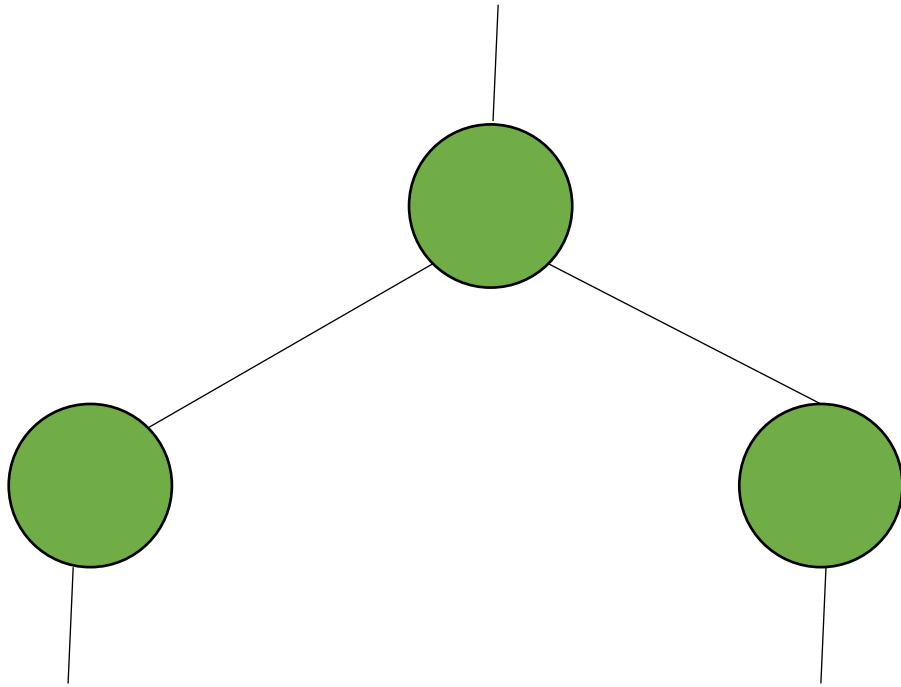


*Also a valid derivation*

# 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



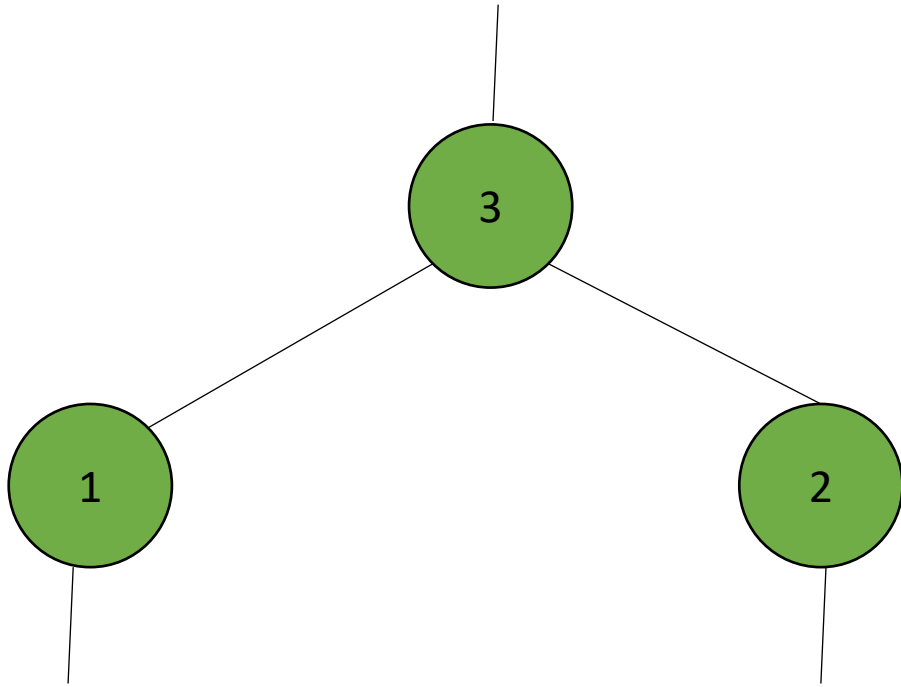
visiting for for different types  
of traversals:

pre order?

in order?

post order?

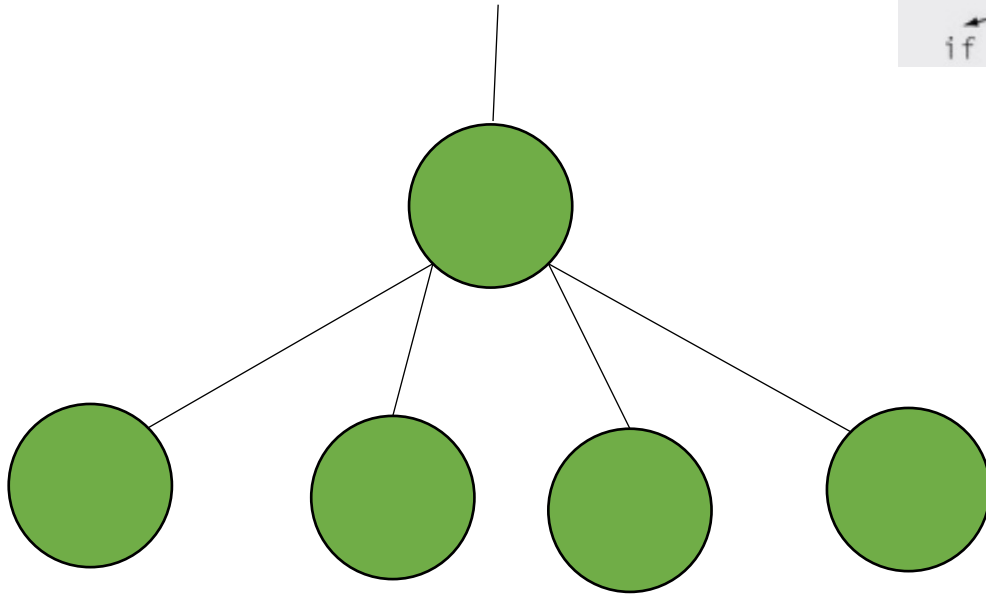
# Post order traversal



visiting for for different types  
of traversals:

post order

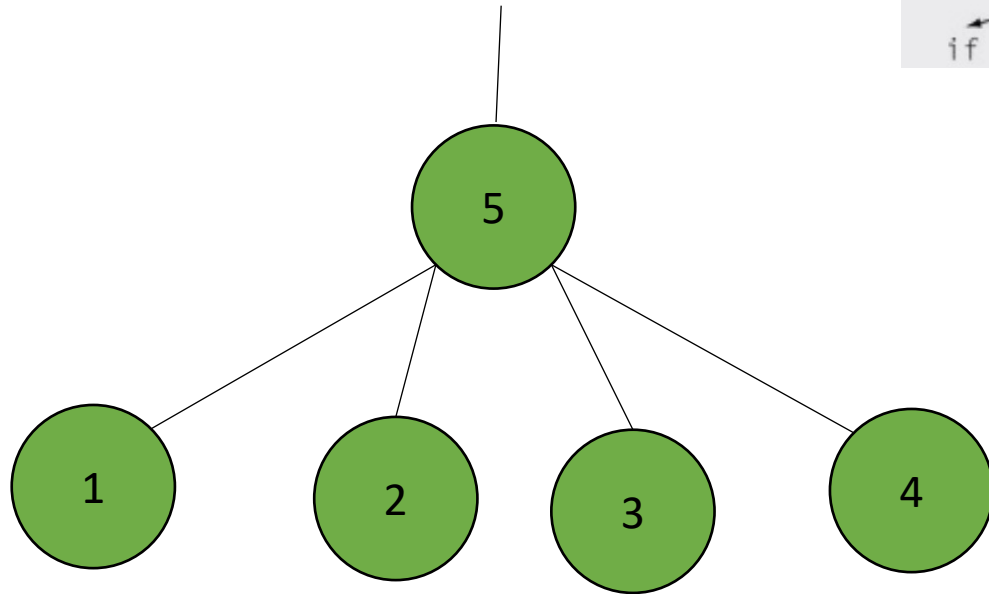
# Post order traversal



What if you have more children?

post order

# Post order traversal



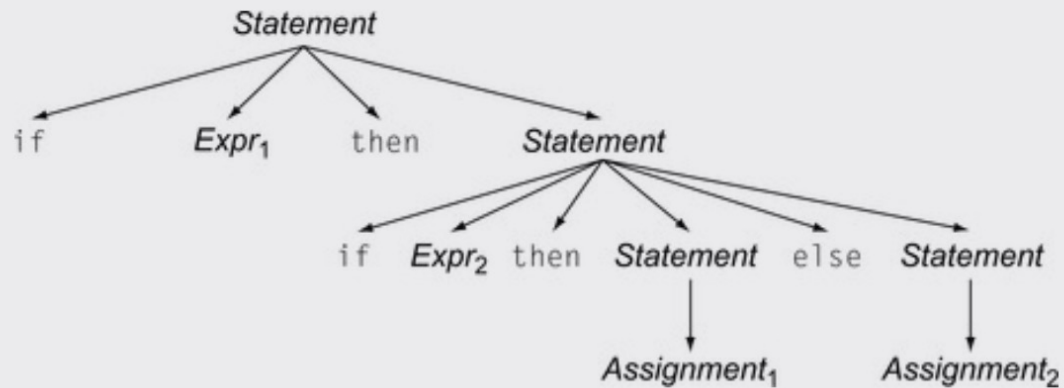
What if you have more children?

post order

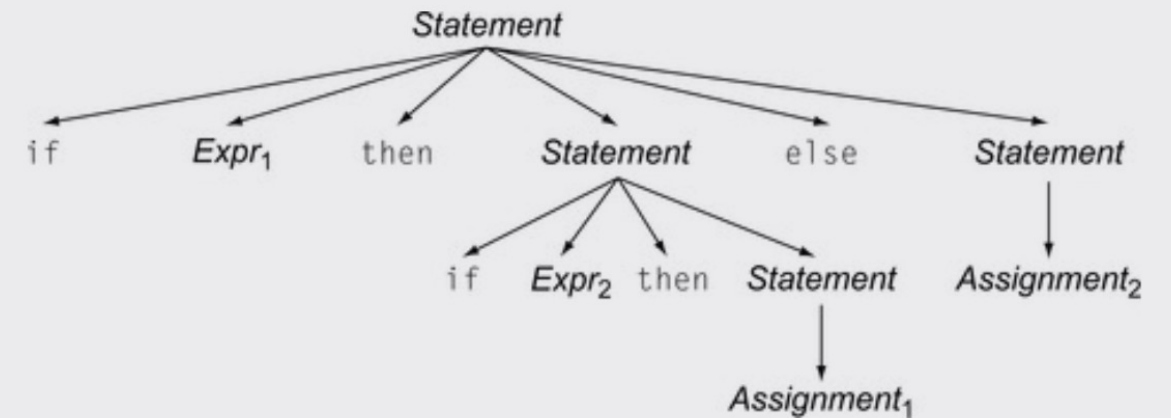


# Ambiguous grammars

When we encode meaning into structure, these are very different programs



*Valid derivation*



*Also a valid derivation*

# 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

```
      expr
      |
      <NUM, 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`

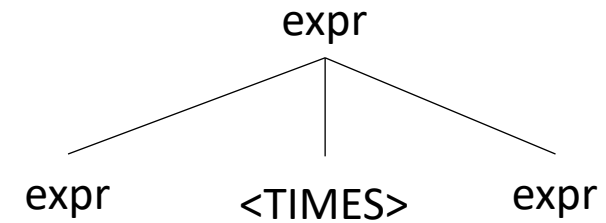
`| LPAREN expr RPAREN`

expr

# Parse trees examples

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

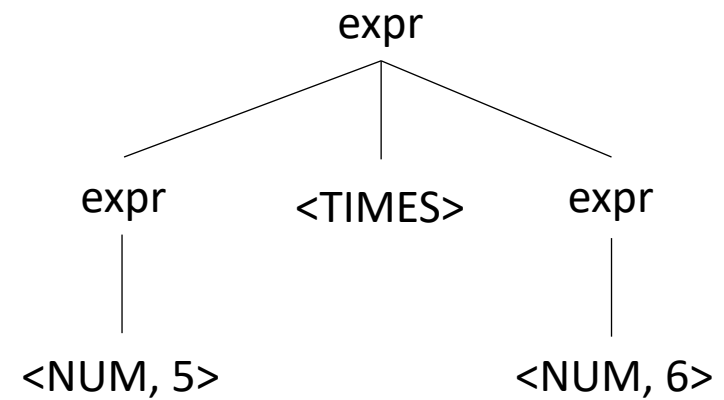
input: 5\*6



# 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

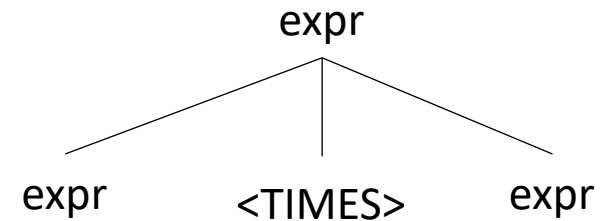
expr

What happens  
in an error?

# Parse trees examples

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

input: 5\*\*6



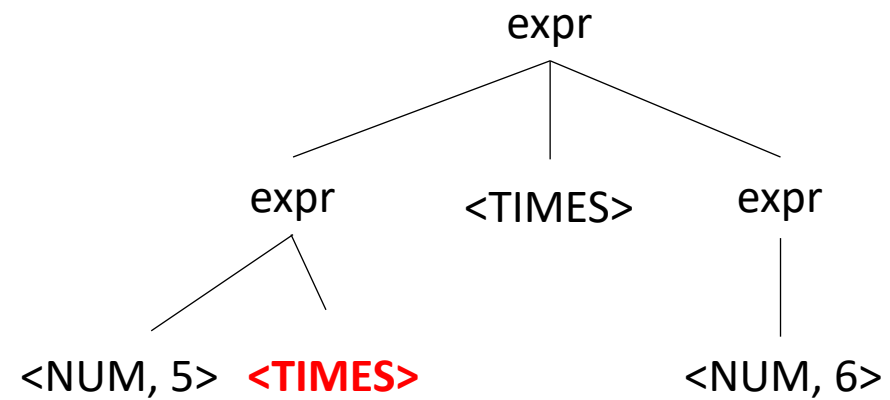
What happens  
in an error?

# 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

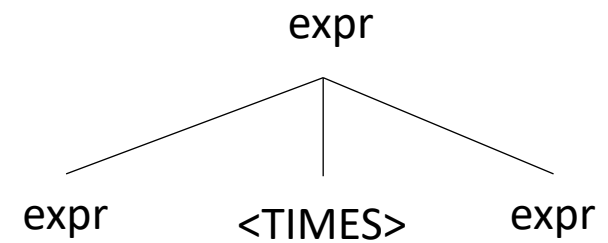
```
expr ::= NUM
      | expr PLUS expr
      | expr TIMES expr
      | LPAREN expr RPAREN
```

expr

# Parse trees examples

input: (1+5)\*6

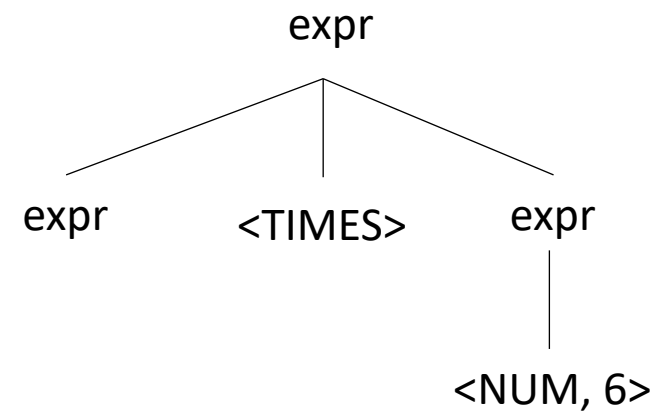
```
expr ::= NUM
      | expr PLUS expr
      | expr TIMES expr
      | LPAREN expr RPAREN
```



# Parse trees examples

input: (1+5)\*6

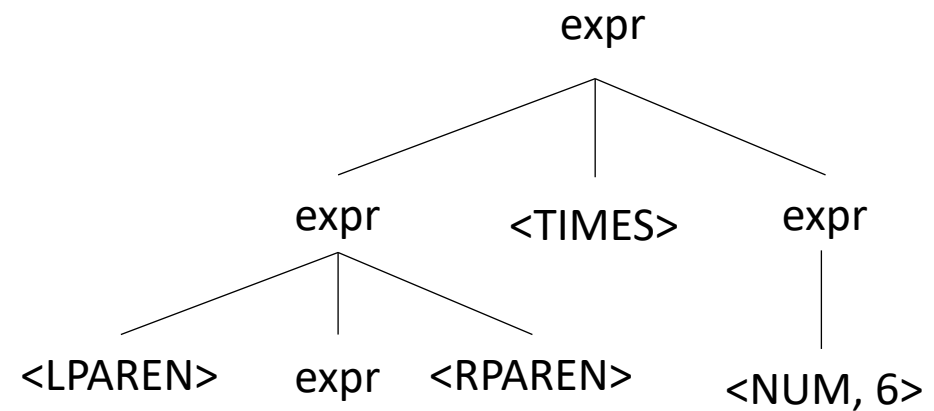
```
expr ::= NUM
      | expr PLUS expr
      | expr TIMES expr
      | LPAREN expr RPAREN
```



# Parse trees examples

input: (1+5)\*6

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

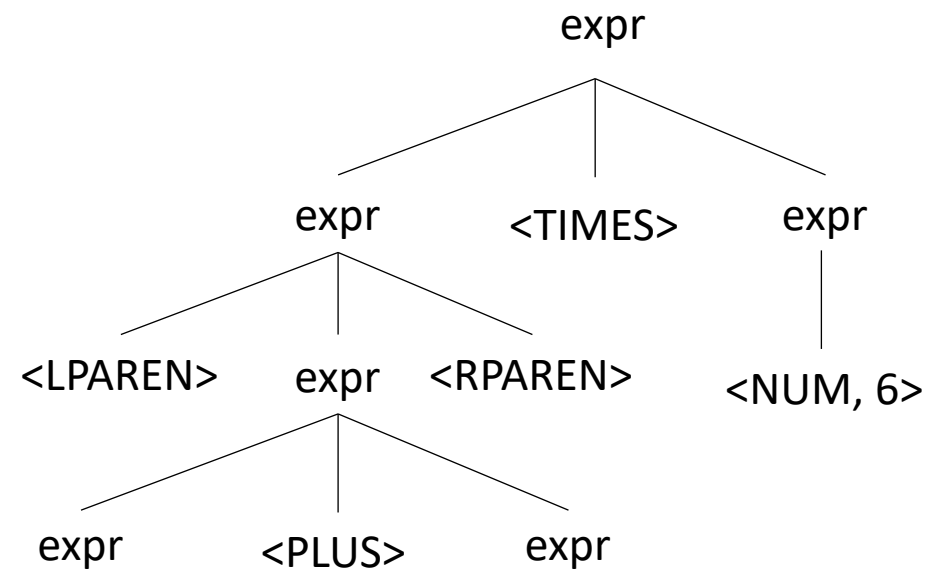




# Parse trees examples

input: (1+5)\*6

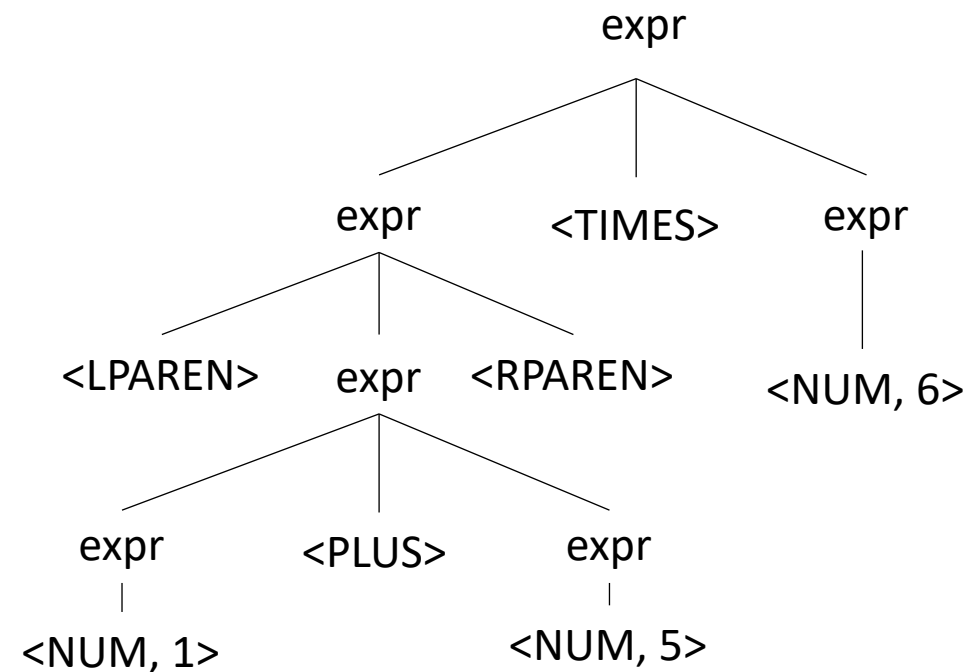
```
expr ::= NUM
      | expr PLUS expr
      | expr TIMES expr
      | LPAREN expr RPAREN
```



# Parse trees examples

input: (1+5)\*6

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

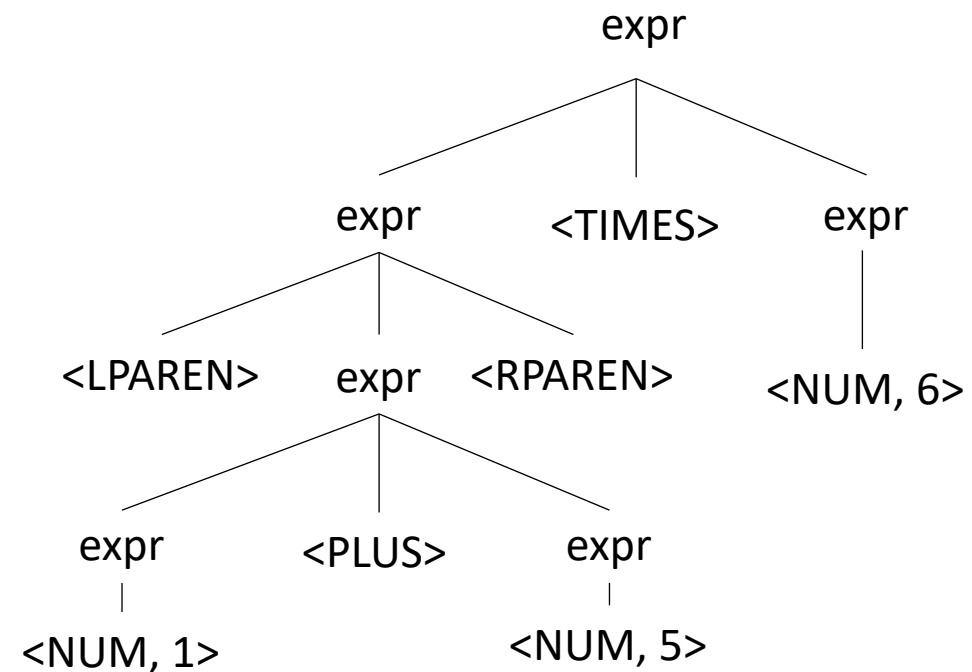


# Parse trees examples

*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

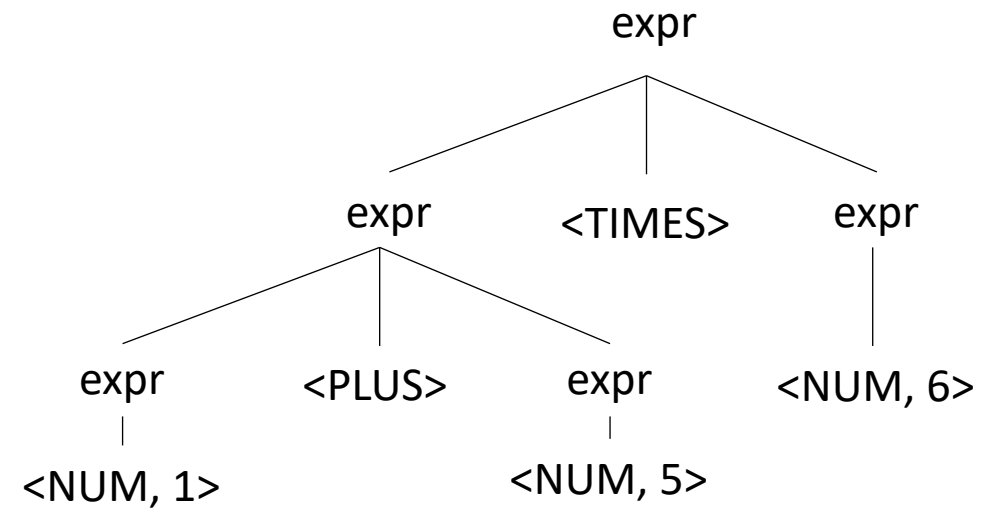
- How about: 1 + 5 \* 6

```
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

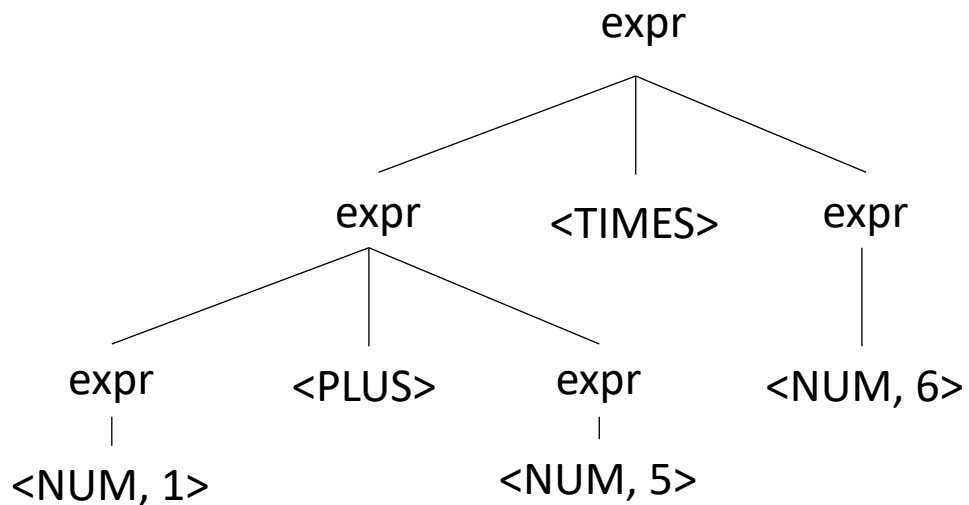
- input: 1 + 5 \* 6

expr ::= NUM

| expr PLUS expr

| expr TIMES expr

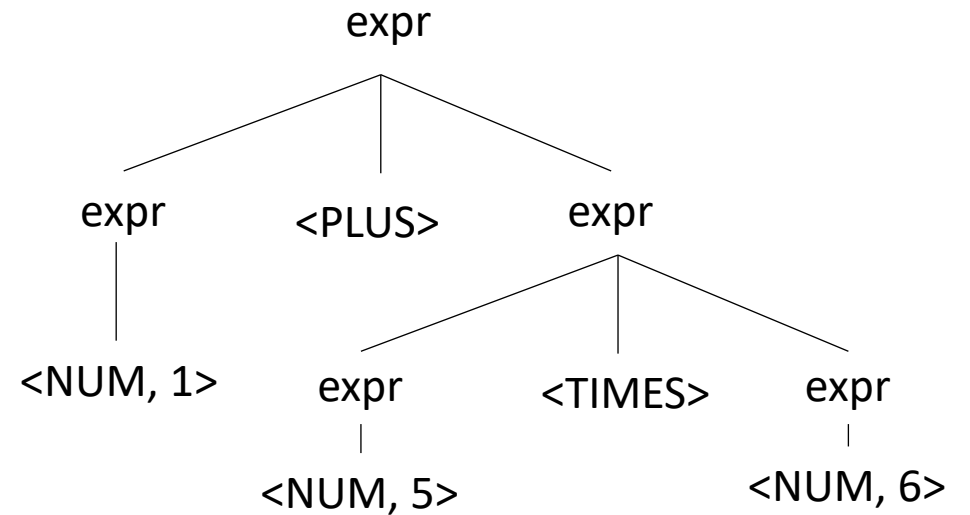
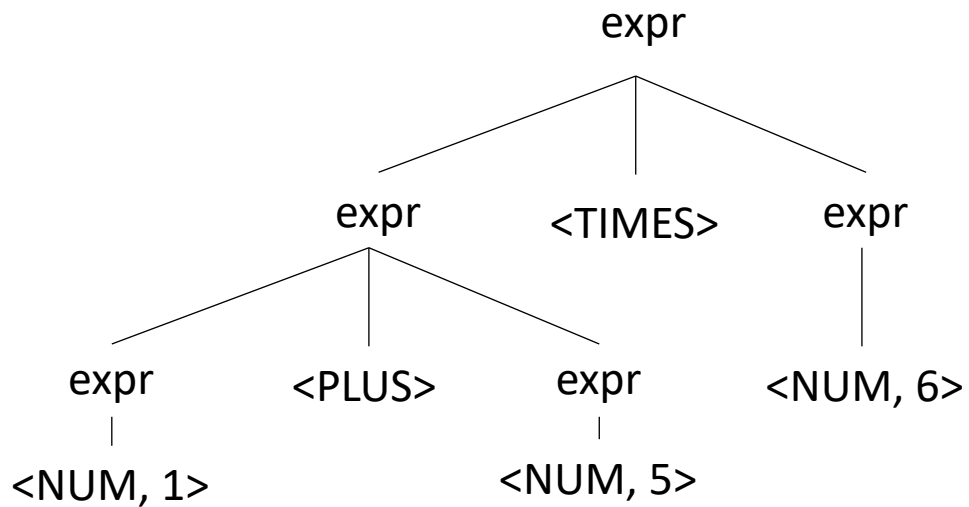
| LPAREN expr RPAREN



# 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: ambiguity comes from conflicts. Explicitly define how to deal with conflicts, e.g. write\* has higher precedence than +
- Some parser generators support this, e.g. Yacc



# Avoiding Ambiguity

- How to avoid ambiguity related to precedence?
- **Second way:** 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

- How to avoid ambiguity related to precedence?
- **Second way:** 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:
  - + \* ()

Precedence  
increases going down

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

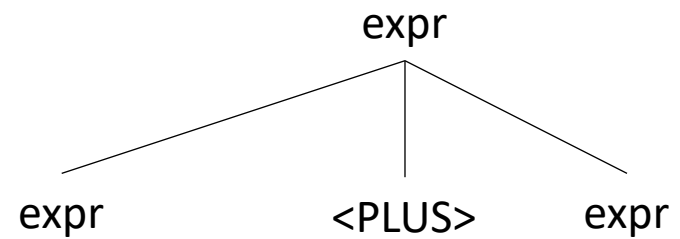
expr

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

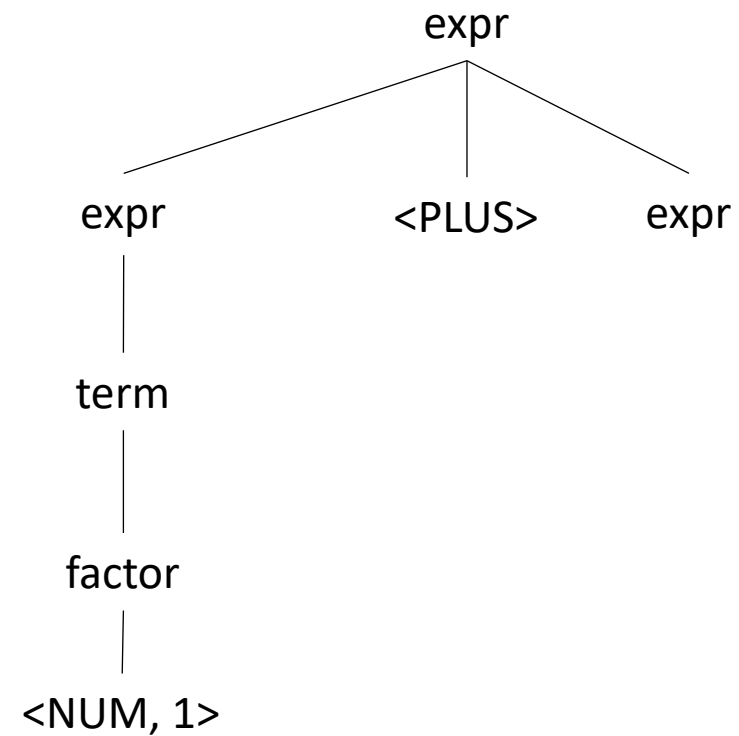
Operator	Name	Productions
+	expr	: expr PLUS expr   term
*	term	: term TIMES term   factor
()	factor	: LPAREN expr RPAREN   NUM



# Now lets create a parse tree

input: 1+5\*6

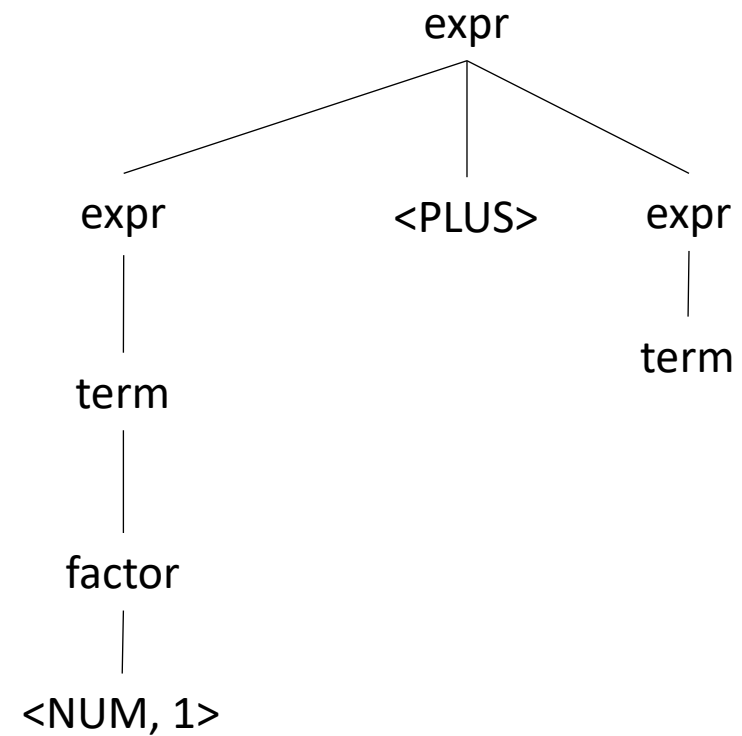
Operator	Name	Productions
+	expr	: expr PLUS expr   term
*	term	: term TIMES term   factor
()	factor	: LPAREN expr RPAREN   NUM



# Now lets create a parse tree

input: 1+5\*6

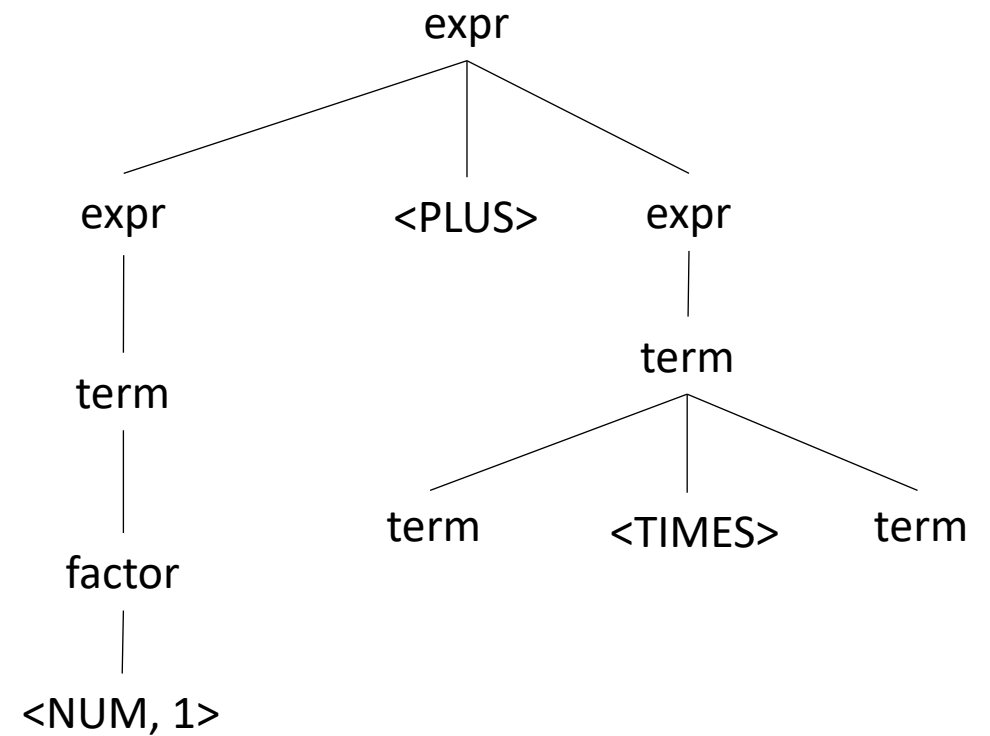
Operator	Name	Productions
+	expr	: expr PLUS expr   term
*	term	: term TIMES term   factor
()	factor	: LPAREN expr RPAREN   NUM



# 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

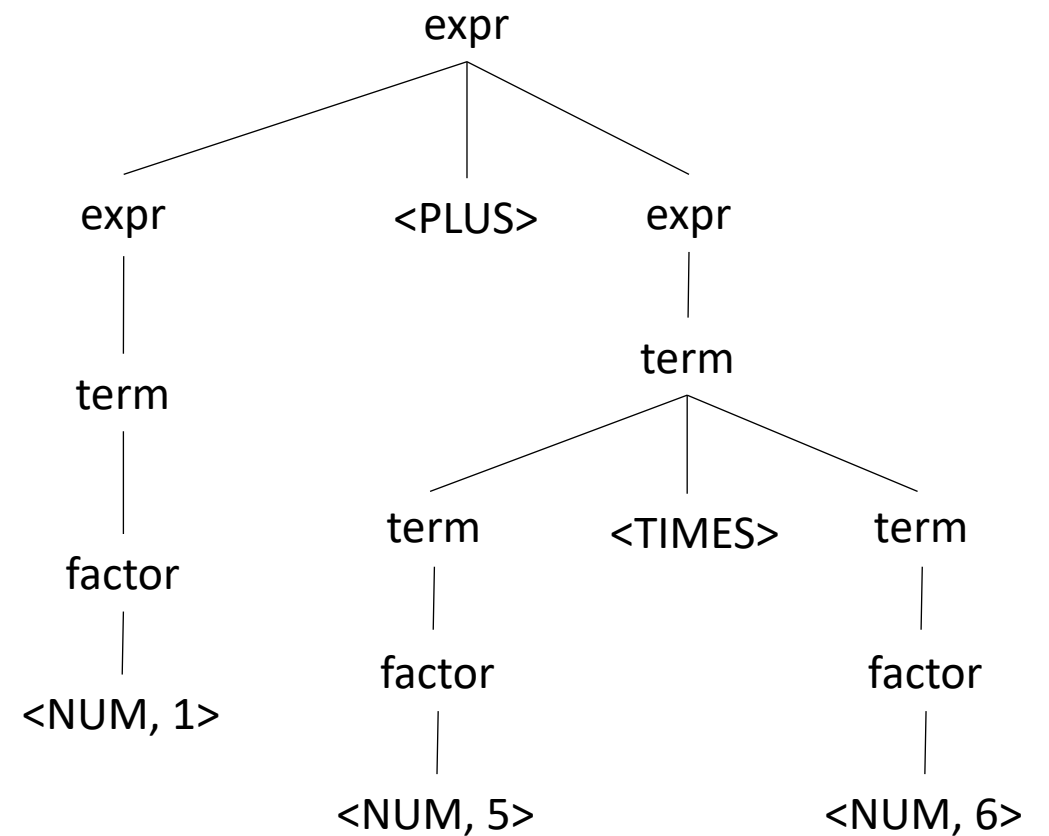




# Now lets create a parse tree

input: 1+5\*6

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



# Parsing REs

Let's try it for regular expressions,  $\{ | . * () \}$

- *Assume . is concat*

Operator	Name	Productions

# Parsing REs

Let's try it for regular expressions,  $\{ | \cdot * () \}$

- *Assume  $\cdot$  is concat*

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

# Parsing REs

Let's try it for regular expressions,  $\{| \cdot * ()\}$

- Assume  $\cdot$  is concat

input:  $a.b \mid c^*$

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

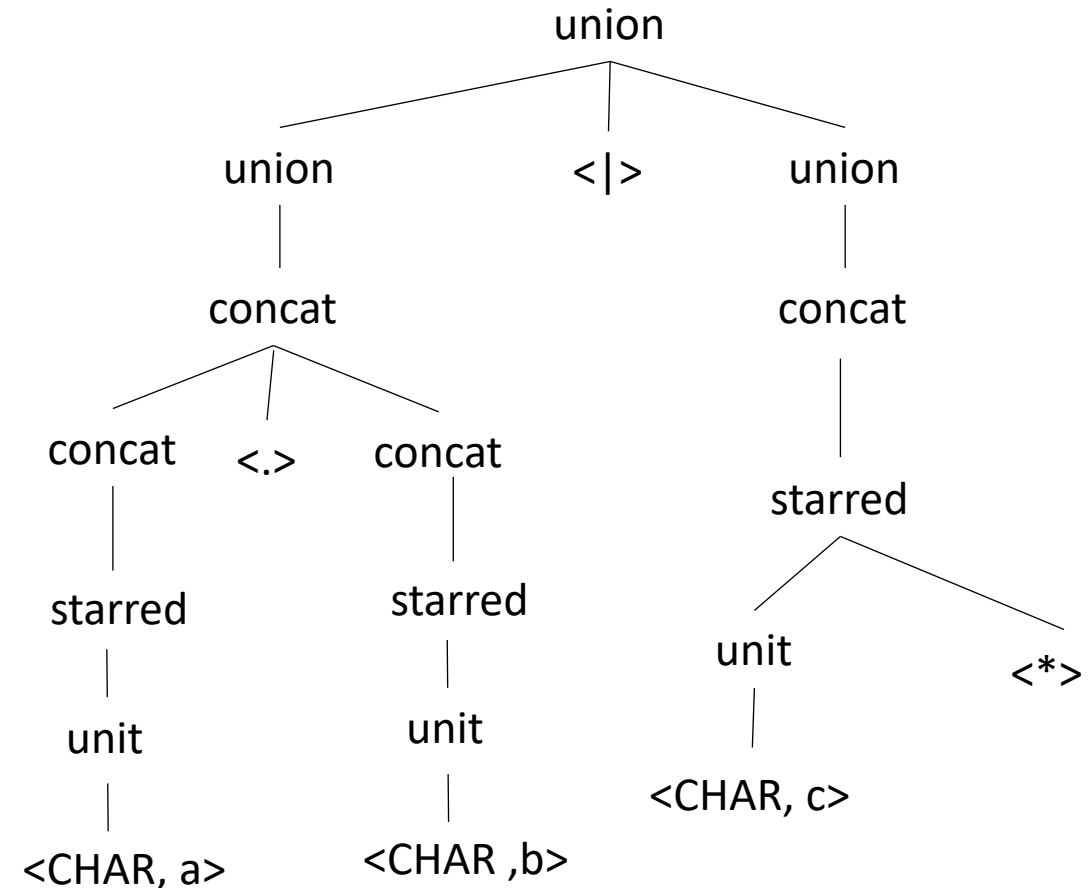
# Parsing REs

Let's try it for regular expressions,  $\{| \cdot * ()\}$

- Assume  $\cdot$  is concat

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](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

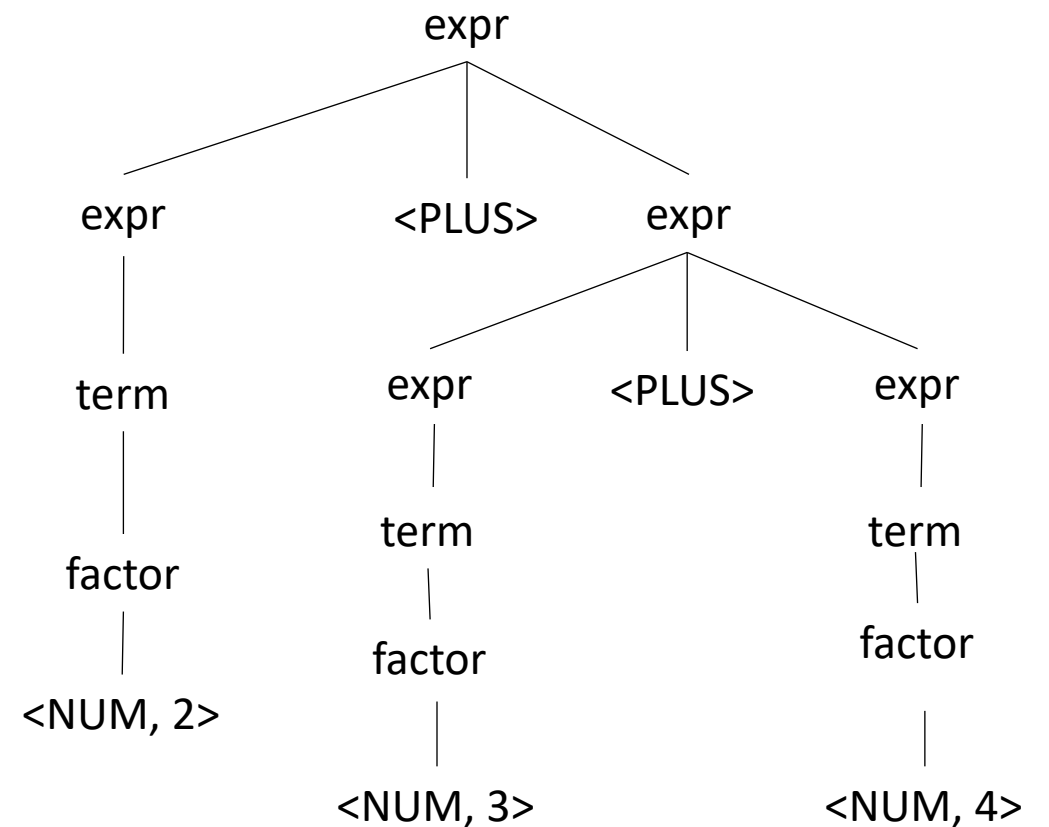
Operator	Name	Productions
+	expr	: expr PLUS expr   term
*	term	: term TIMES term   factor
()	factor	: LP expr RP   NUM



# Let's make some more parse trees

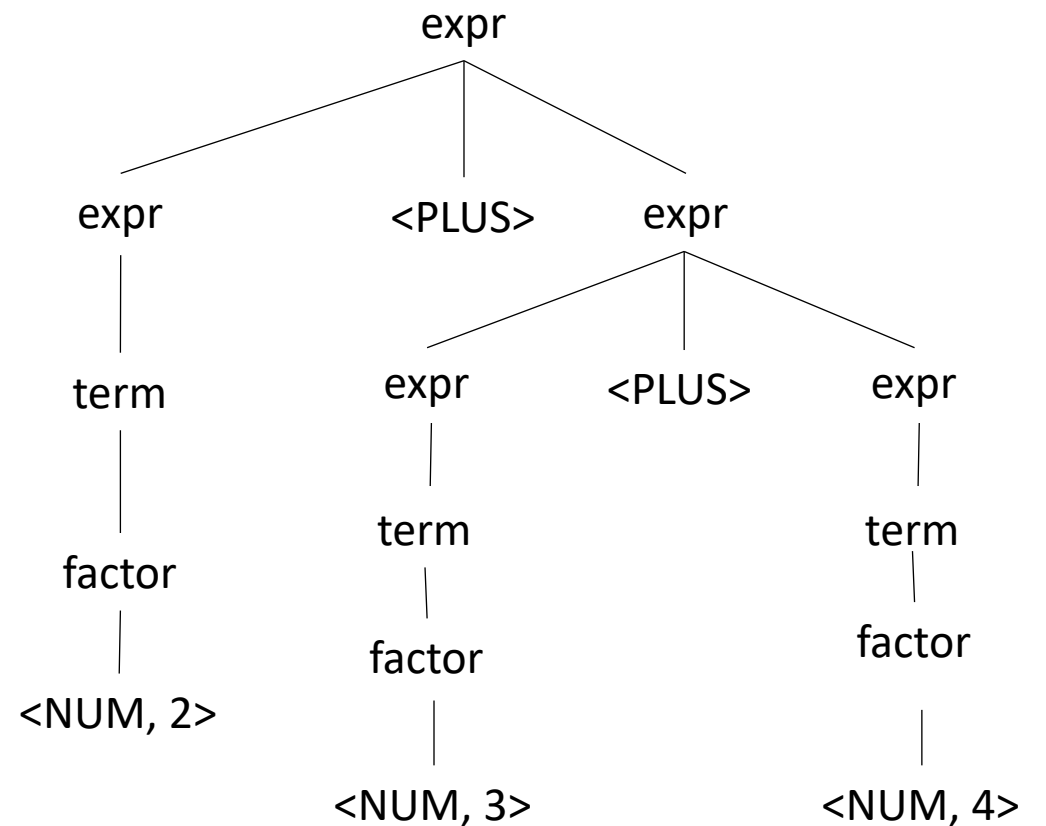
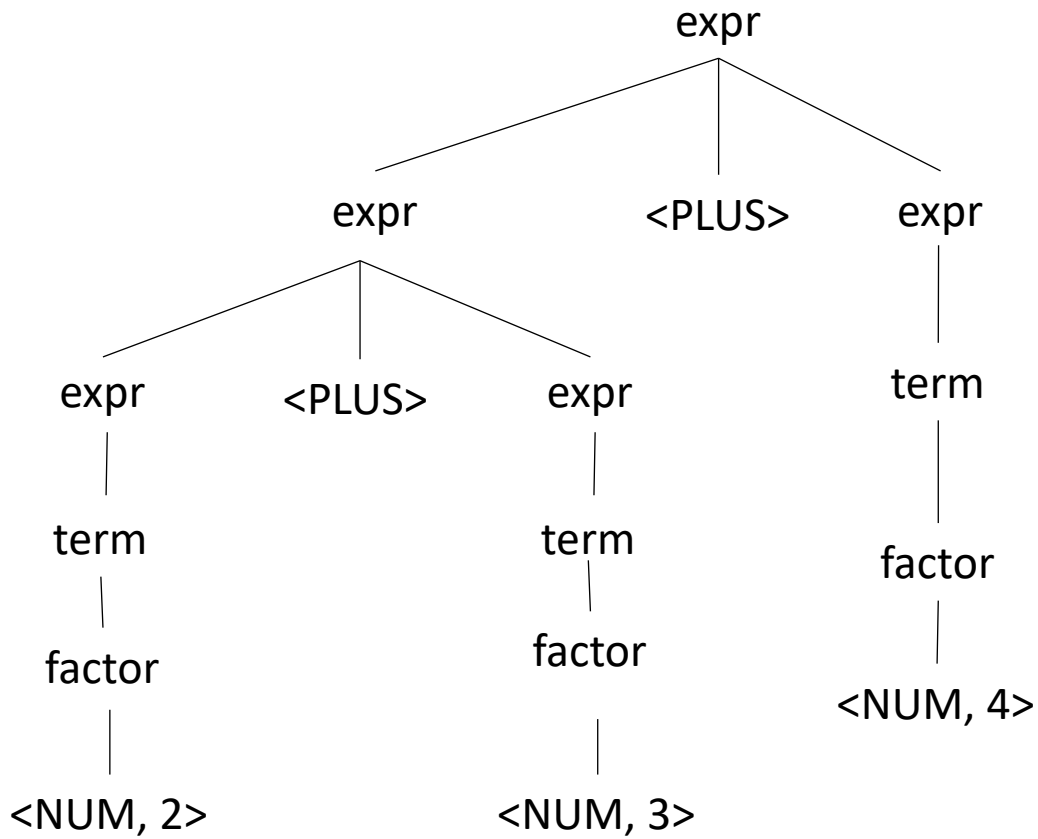
input: 2+3+4

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

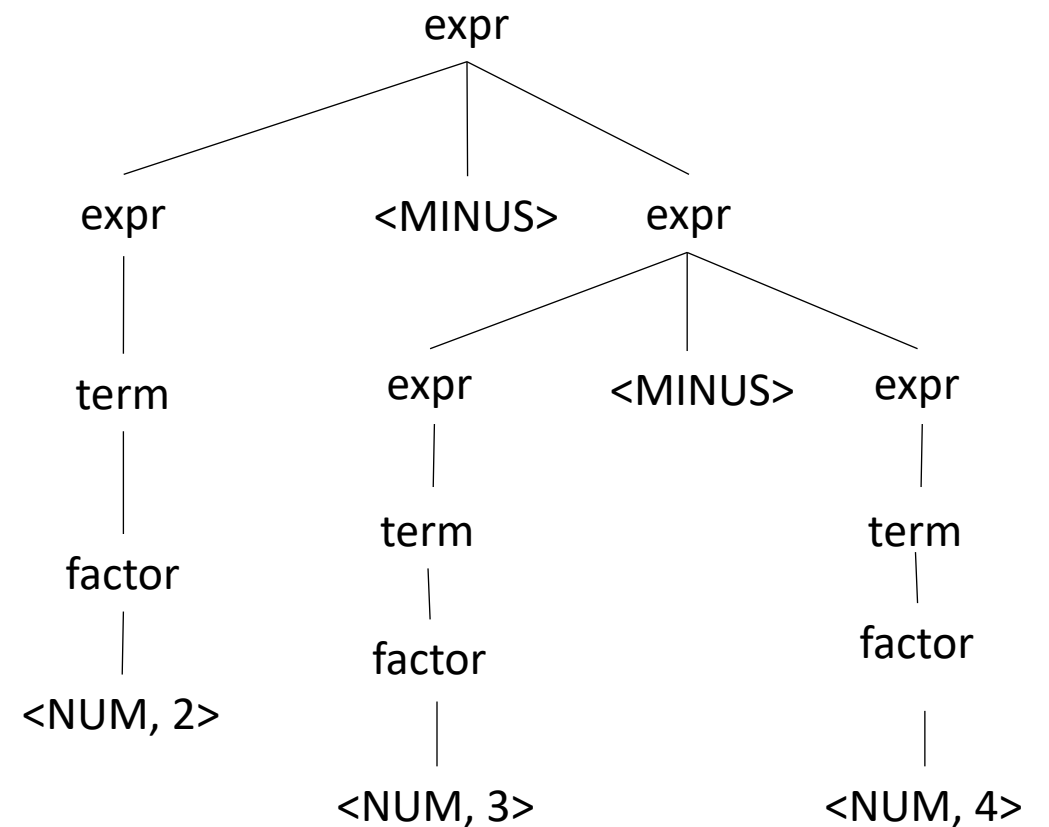
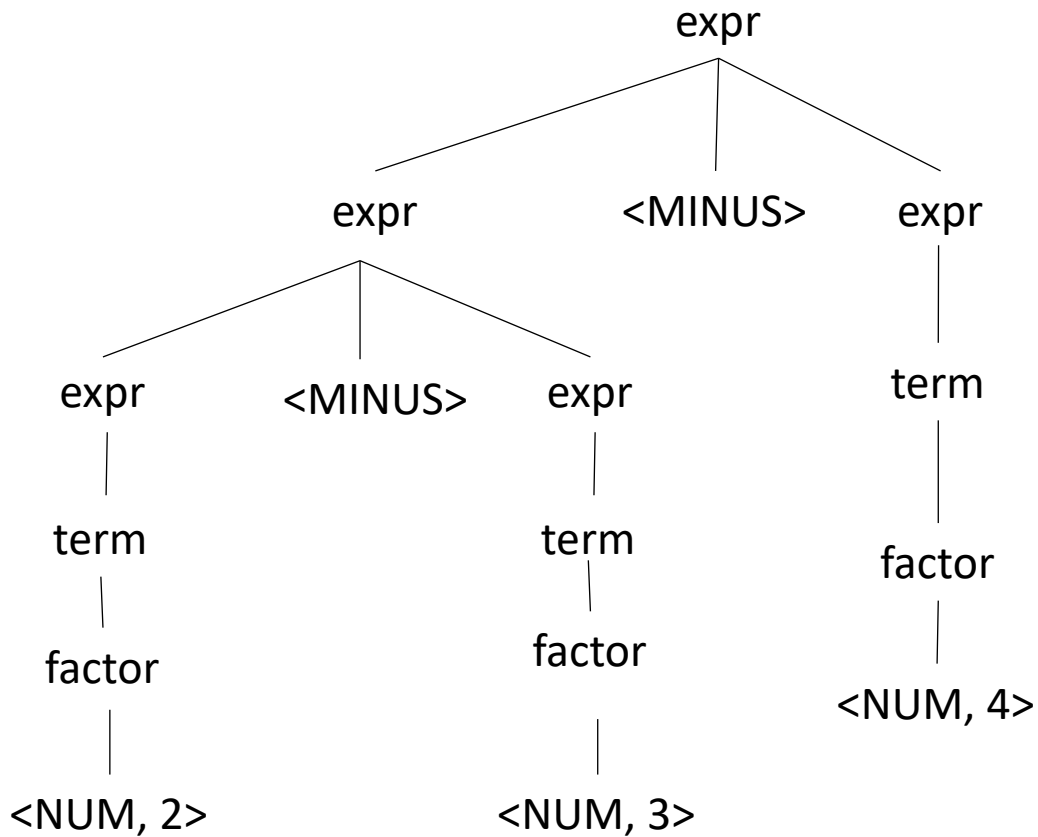


What about for a different operator?

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:

- $((a \text{ OP } b) \text{ OP } c) == (a \text{ OP } (b \text{ OP } c))$

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



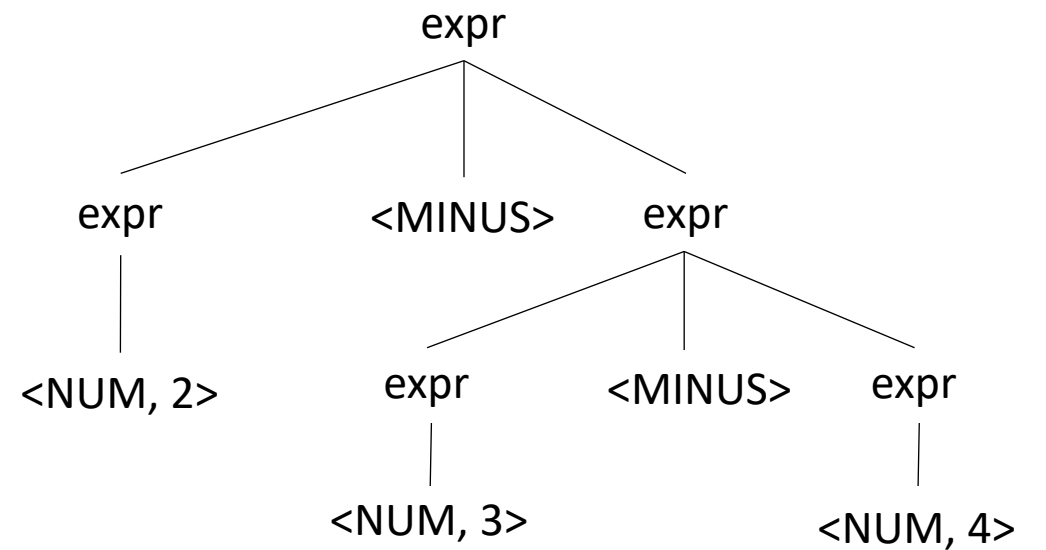
# How to encode associativity?

- 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

# Associativity for a single operator

input: 2-3-4

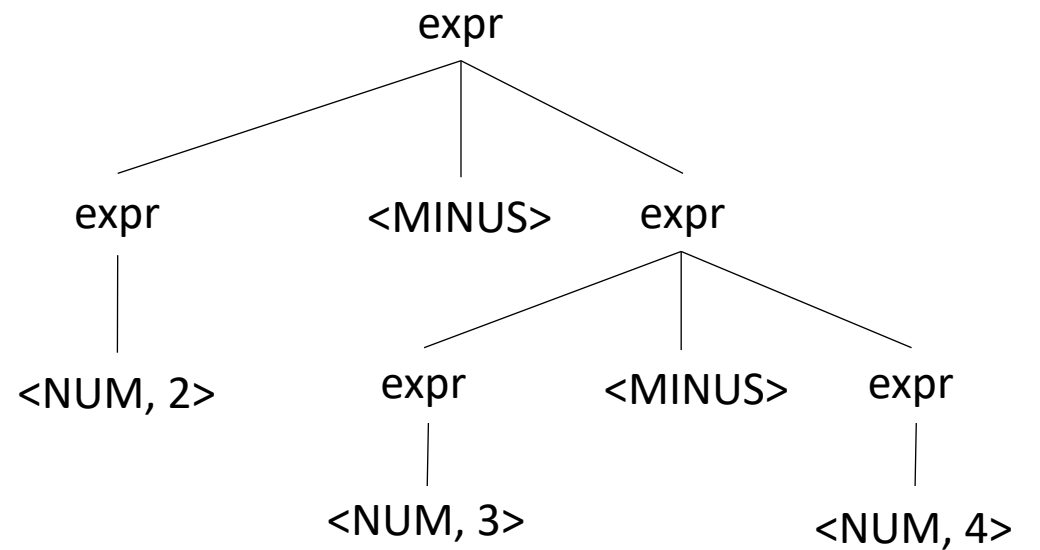
Operator	Name	Productions
-	expr	: expr MINUS expr   NUM



# Associativity for a single operator

input: 2-3-4

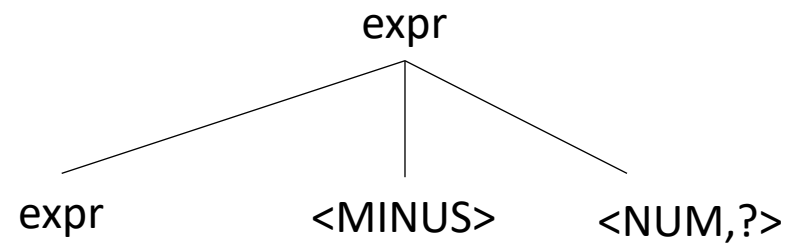
Operator	Name	Productions
-	expr	: expr MINUS NUM   NUM



*No longer allowed*

# Associativity for a single operator

input: 2-3-4

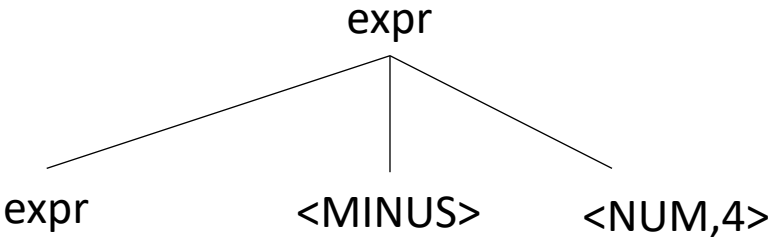


Operator	Name	Productions
-	expr	: expr MINUS NUM   NUM

Lets start over

# Associativity for a single operator

input: 2-3-4

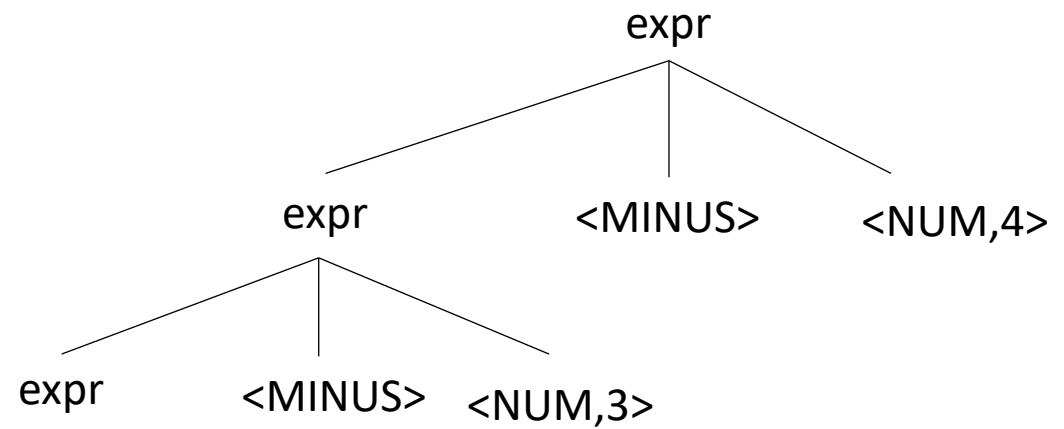


Operator	Name	Productions
-	expr	: expr MINUS NUM   NUM

# Associativity for a single operator

input: 2-3-4

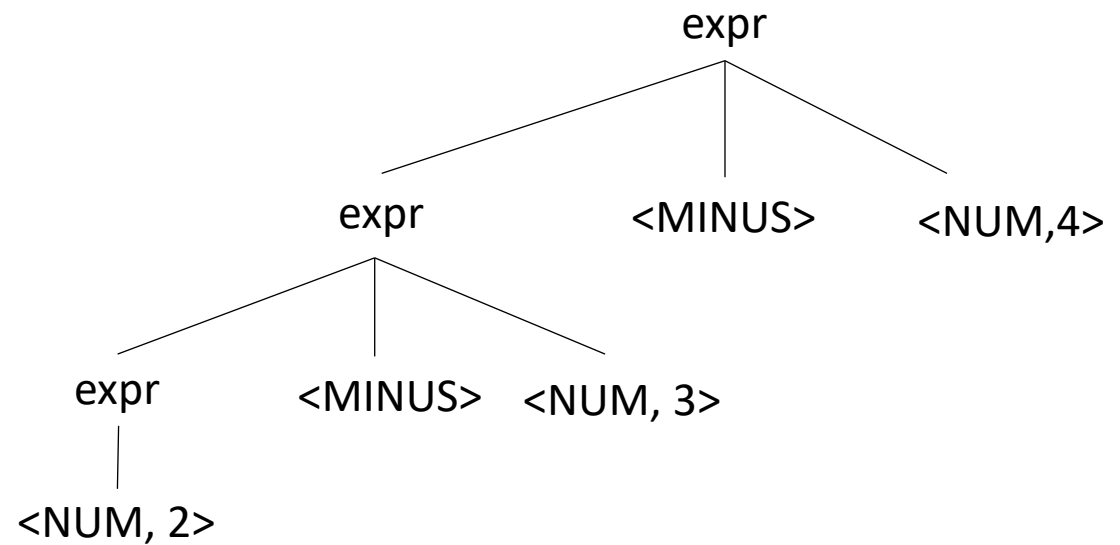
Operator	Name	Productions
-	expr	: expr MINUS NUM   NUM



# Associativity for a single operator

input: 2-3-4

Operator	Name	Productions
-	expr	: expr MINUS NUM   NUM



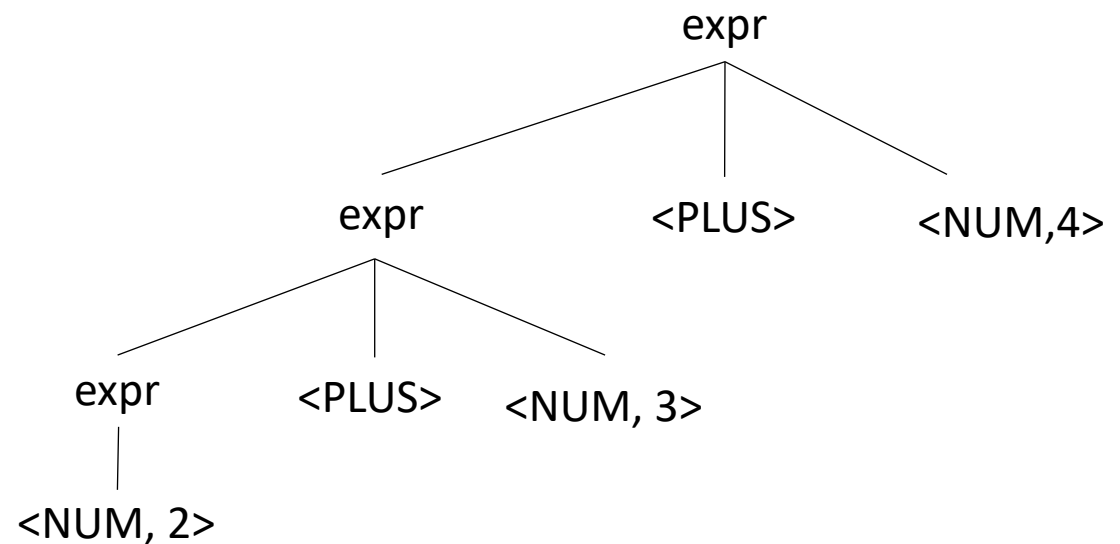
# Should you have associativity when its not required?

Benefits?

Drawbacks?

Operator	Name	Productions
+	expr	: expr PLUS NUM   NUM

input: 2+3+4



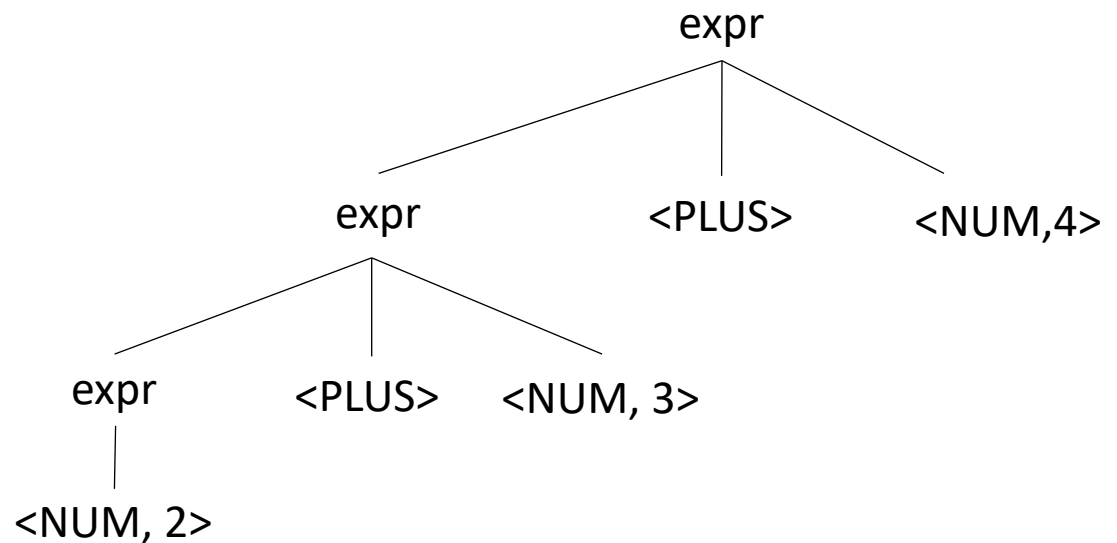


# Should you have associativity when its not required?

Benefits?  
Drawbacks?

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	$: \text{expr PLUS term}$ $  \text{expr MINUS term}$ $  \text{term}$
$*, /$	term	$: \text{term TIMES pow}$ $  \text{term DIV pow}$ $  \text{pow}$
$^$	pow	$: \text{factor CARROT pow}$ $  \text{factor}$
$()$	factor	$: \text{LPAR expr RPAR}$ $  \text{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](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