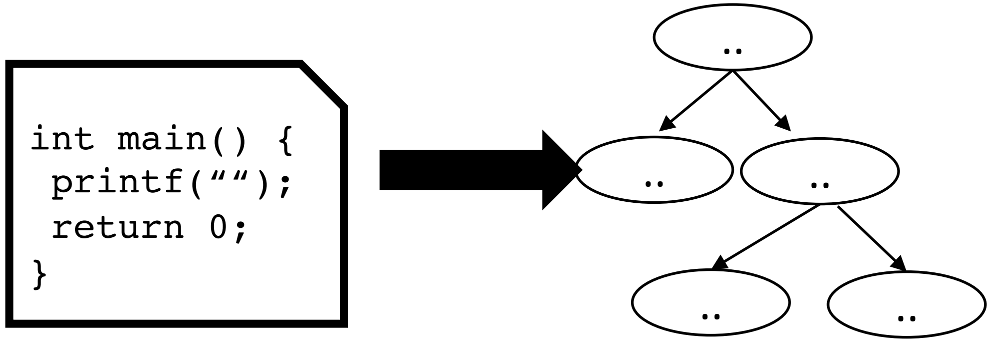
CSE110A: Compilers
April 13, 2022

Topics:

- *Syntactic Analysis continued*
  - *Precedence and associativity part 2*
  - *Top down parsing*
    - *Oracle parser*
    - *Rewriting to avoid left recursion*



Announcements

- HW 1 is due on Monday at midnight

- For help
  - Ask on Piazza: ***No guaranteed help over the weekend or off business hours***
  - Yanwen has office hours on Monday 1-2, but I hope you will not save it that late!

- Test case post on Piazza
  - It is best to share the test cases there so that teaching staff can look over them.

- Plan on HW2 assigned on Monday (due 2 weeks later)

Announcements

- For part 4:
  - *You must use the same tokens that you created in part 2 and used in part 3*
  - *You must build the RE programmatically*
  - *Keep track of the token actions in a separate data structure*

Quiz

Quiz

What is an example of input recognized by the following grammar?

A → A x

A → y

- ○ xxxxxxxxy
- ○ xyyyyyyyy
- ○ yxxxxxxxx
- ○ yyyyyyyyx

Quiz

What is an example of input recognized by the following grammar?

1 $A \rightarrow A\ x$

2 $A \rightarrow y$

How about this one? `xxxxxxxxxy`

RULE	Sentential Form
start	A

Quiz

What is an example of input recognized by the following grammar?

1 A → A x

2 A → y

How about this one? `xxxxxxxxxy`

RULE	Sentential Form
start	A

Applying either rule gives us a sentential form that won’t create the string

Quiz

What is an example of input recognized by the following grammar?

1 $A \rightarrow A\ x$

2 $A \rightarrow y$

How about this one? `xyyyyyyyy`

RULE	Sentential Form
start	A

Quiz

What is an example of input recognized by the following grammar?

1 A → A x

2 A → y

How about this one? `xyyyyyyyy`

RULE	Sentential Form
start	A

Similar reason:
strings that are
longer than 1 character
cannot end in y

Quiz

What is an example of input recognized by the following grammar?

1 $A \rightarrow A\ x$

2 $A \rightarrow y$

How about this one? `yxxxxxxxx`

RULE	Sentential Form
start	A

Quiz

What is an example of input recognized by the following grammar?

1 A → A x

2 A → y

How about this one? yxxxxxxxx

RULE	Sentential Form
start	A
1	A x
1	A x x
2	y xxxxxxxx

9 more rows, then eventually

Quiz

What is an example of input recognized by the following grammar?

1 A → A x
2 A → y

How about this one? `yyyyyyyyx`

RULE	Sentential Form
start	A

Quiz

What is an example of input recognized by the following grammar?

1 A → A x

2 A → y

How about this one? `yyyyyyyyx`

We can only produce 1 y, so we cannot derive this string

RULE	Sentential Form
start	A

Quiz

What is an example of input recognized by the following grammar?

1 $A \rightarrow A\ x$

2 $A \rightarrow y\ A$

What if we changed the rules?? Does this work?

How about this one? `yyyyyyyyx`

RULE	Sentential Form
start	A

Quiz

What is an example of input recognized by the following grammar?

1 A → A x

2 A → y A

What if we changed the rules?? Does this work?

How about this one? yyyyyyyyx

We need a terminating string:
A -> ""

RULE	**Sentential Form**
start	A

Quiz

Which grammar is ambiguous ?

- $E \rightarrow E + E$
  $E \rightarrow x$
- $E \rightarrow E + x$
  $E \rightarrow x$
- $E \rightarrow x + E$
  $E \rightarrow x$
- $E \rightarrow x + x$
  $E \rightarrow x$

Quiz

Which grammar is ambiguous ?

- $E \rightarrow E + E$
  $E \rightarrow x$
- $E \rightarrow E + x$
  $E \rightarrow x$
- $E \rightarrow x + E$
  $E \rightarrow x$
- $E \rightarrow x + x$
  $E \rightarrow x$

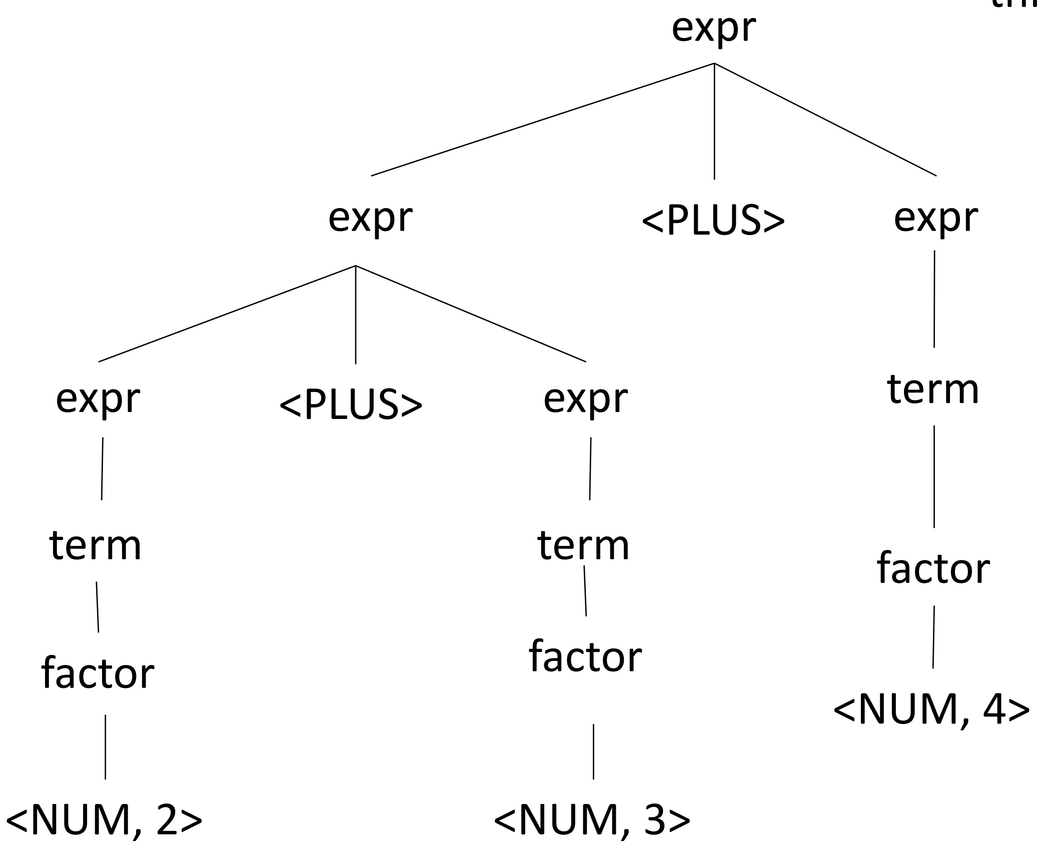
Let’s look at some examples.

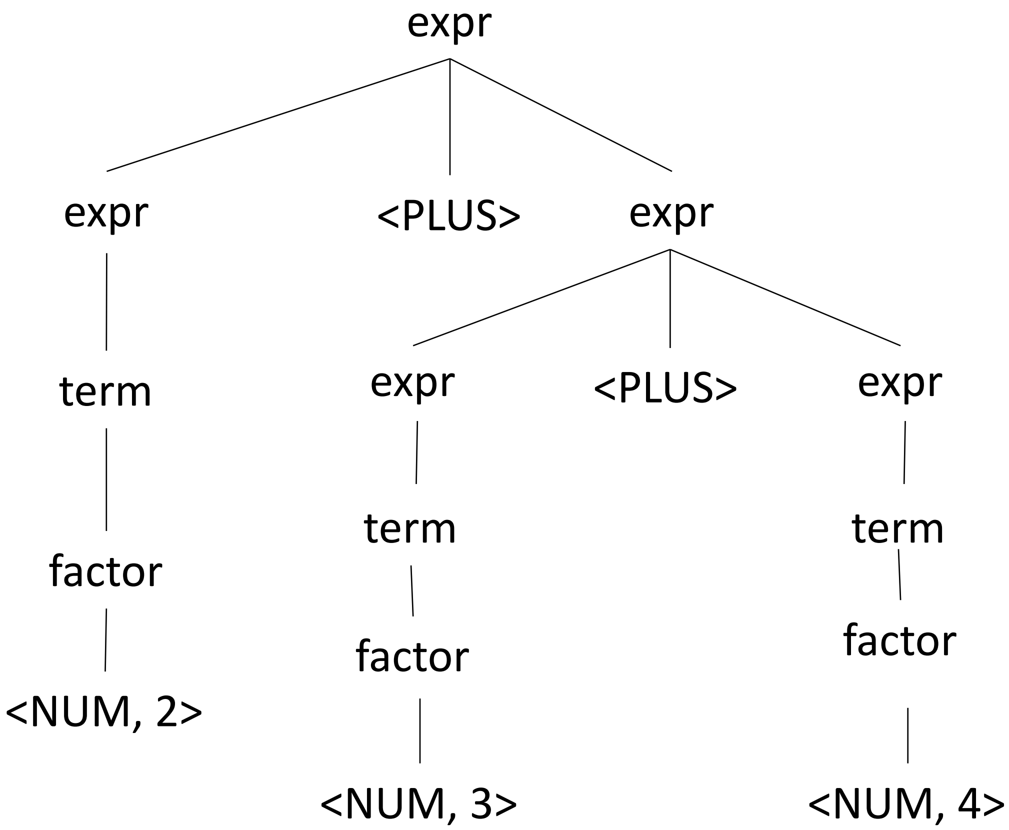
Let’s assume that E is an ”expr” and x is a number

○ E → E + E
E → x

input: 2+3+4

Both parse trees are valid,
this grammar is ambiguous

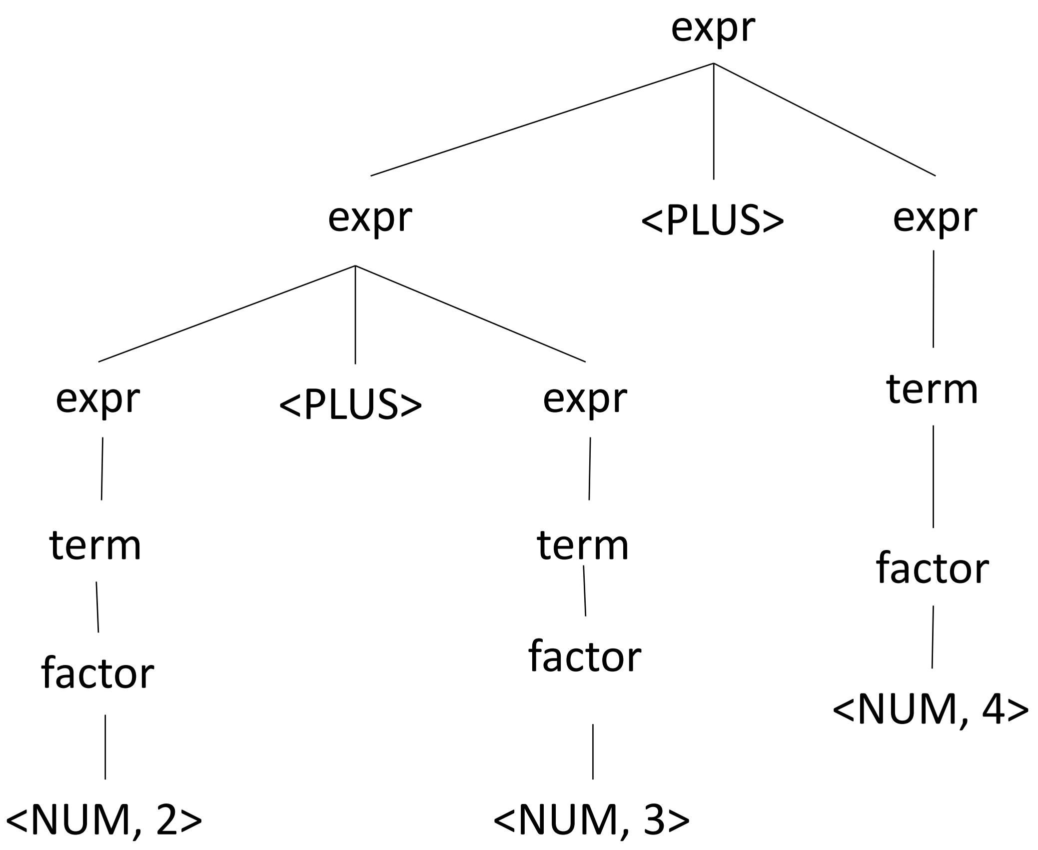


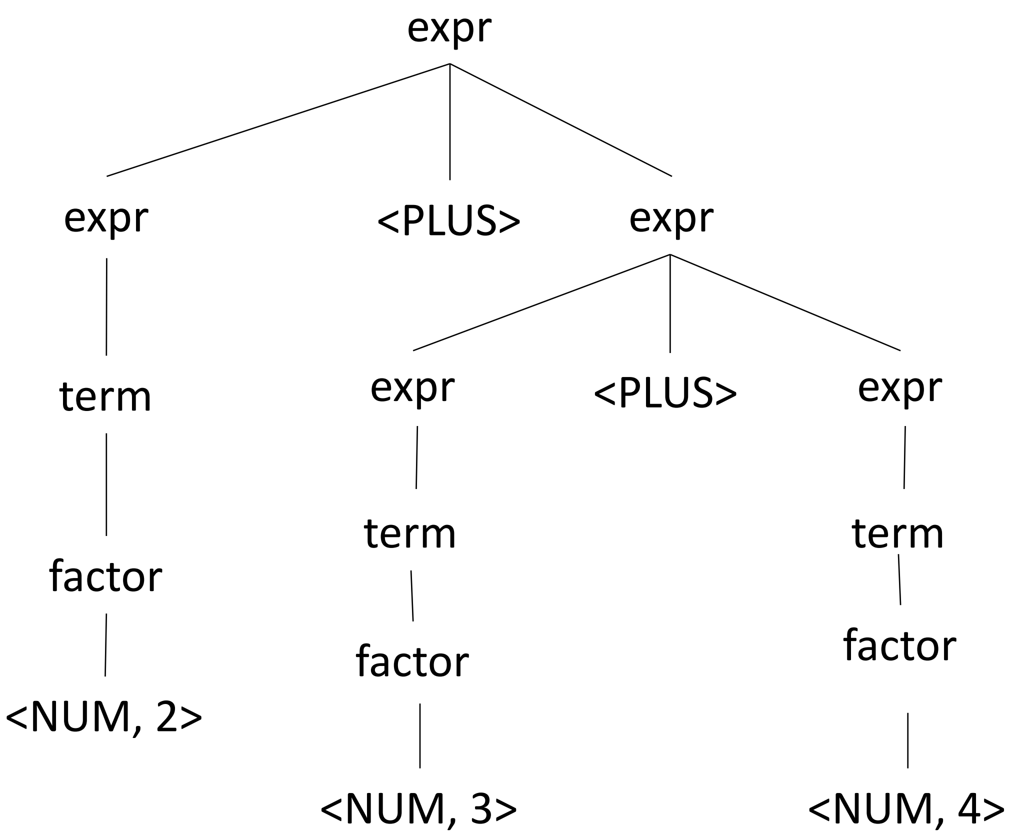


○ E → E + x
E → x

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

What about this one?

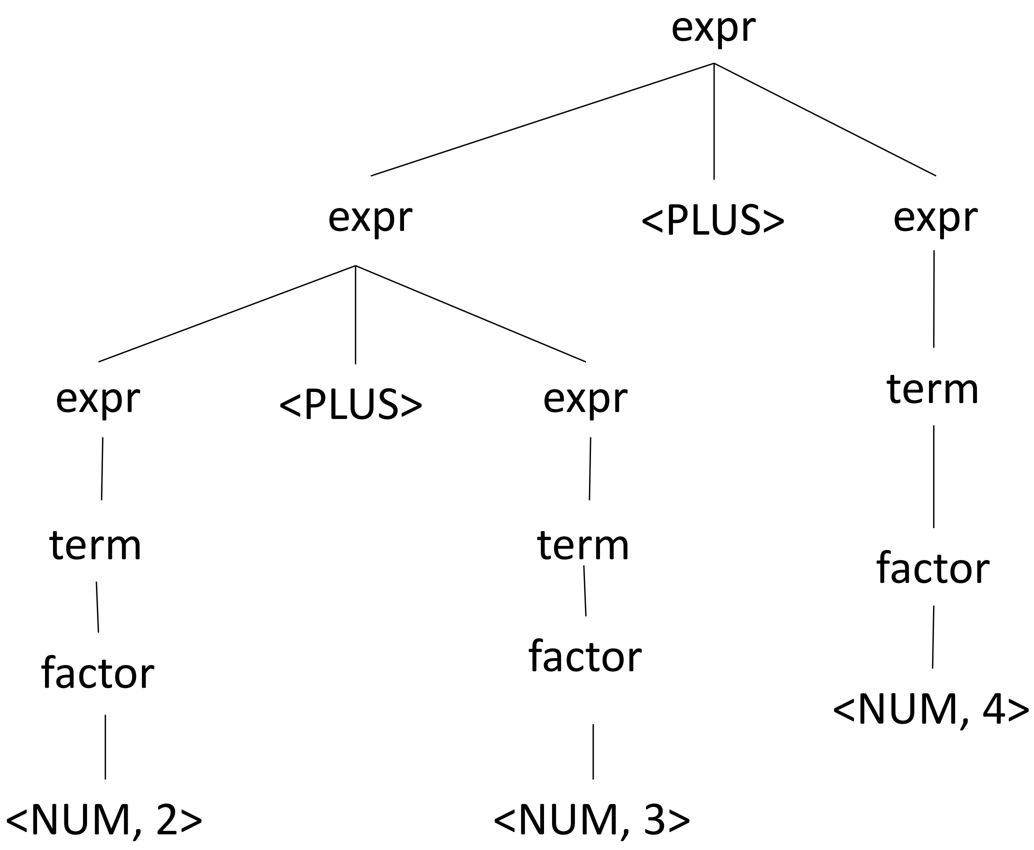




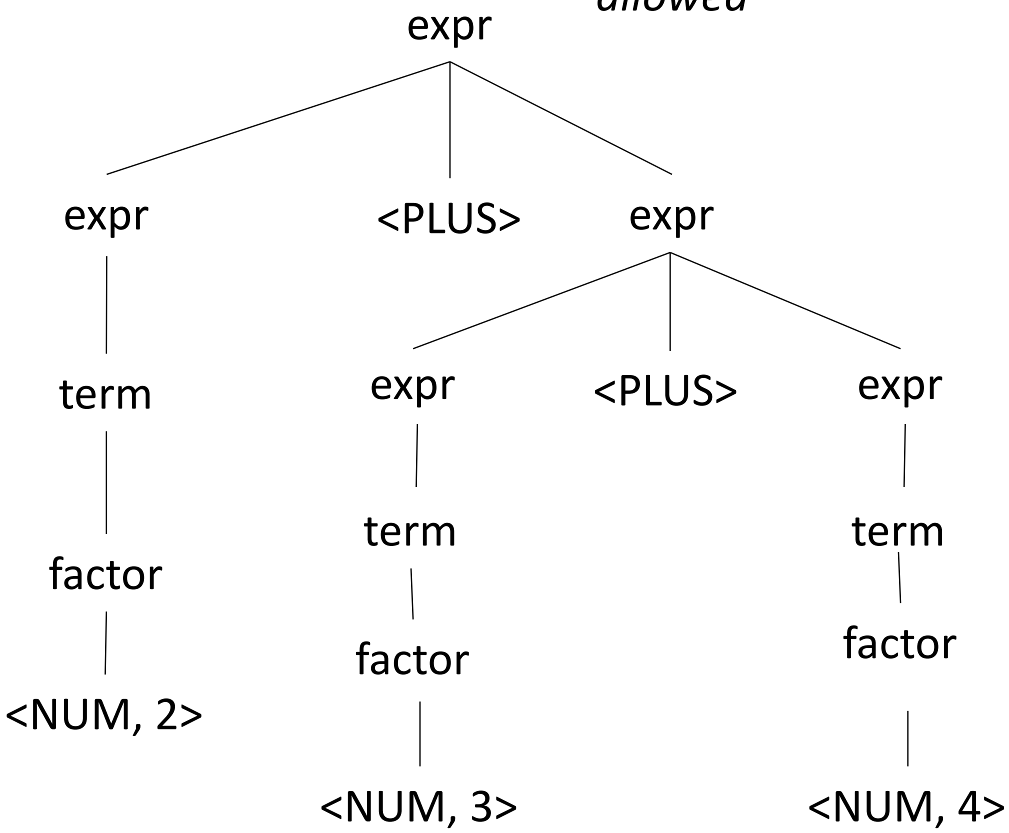
○ E → E + x
E → x

input: 2+3+4

What about this one?



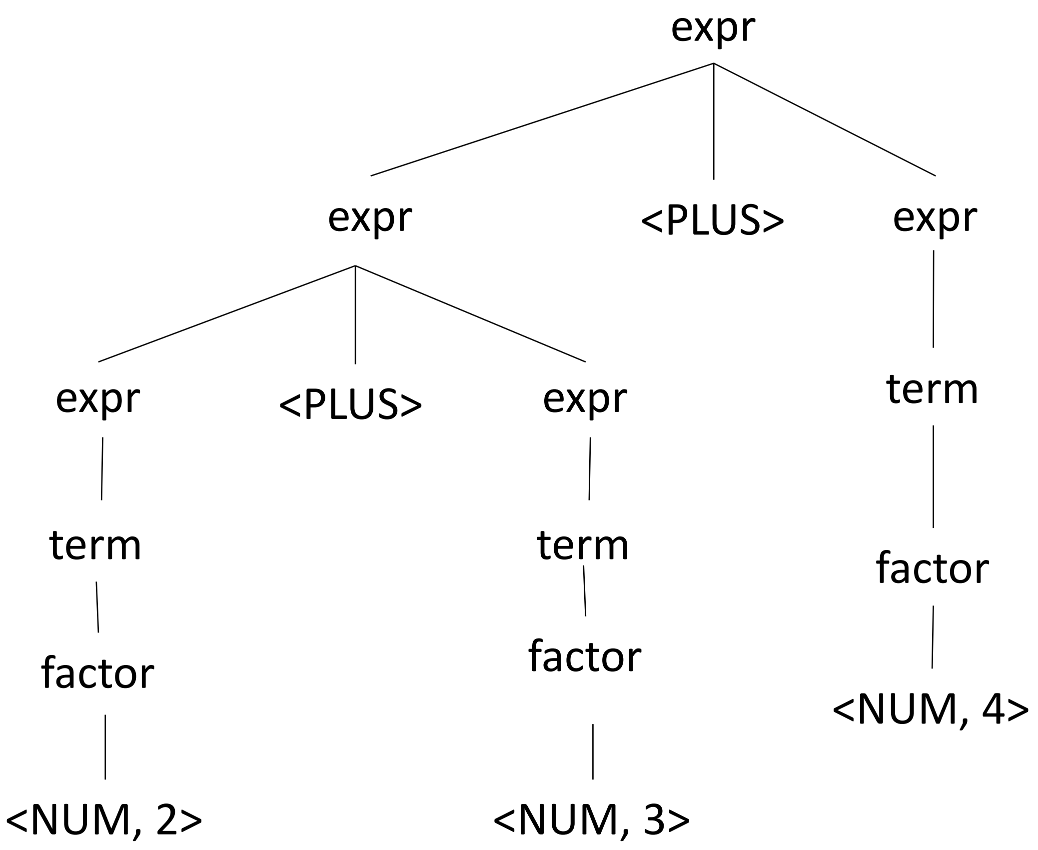
Doesn't allow an expression on the RHS. This parse tree is not allowed

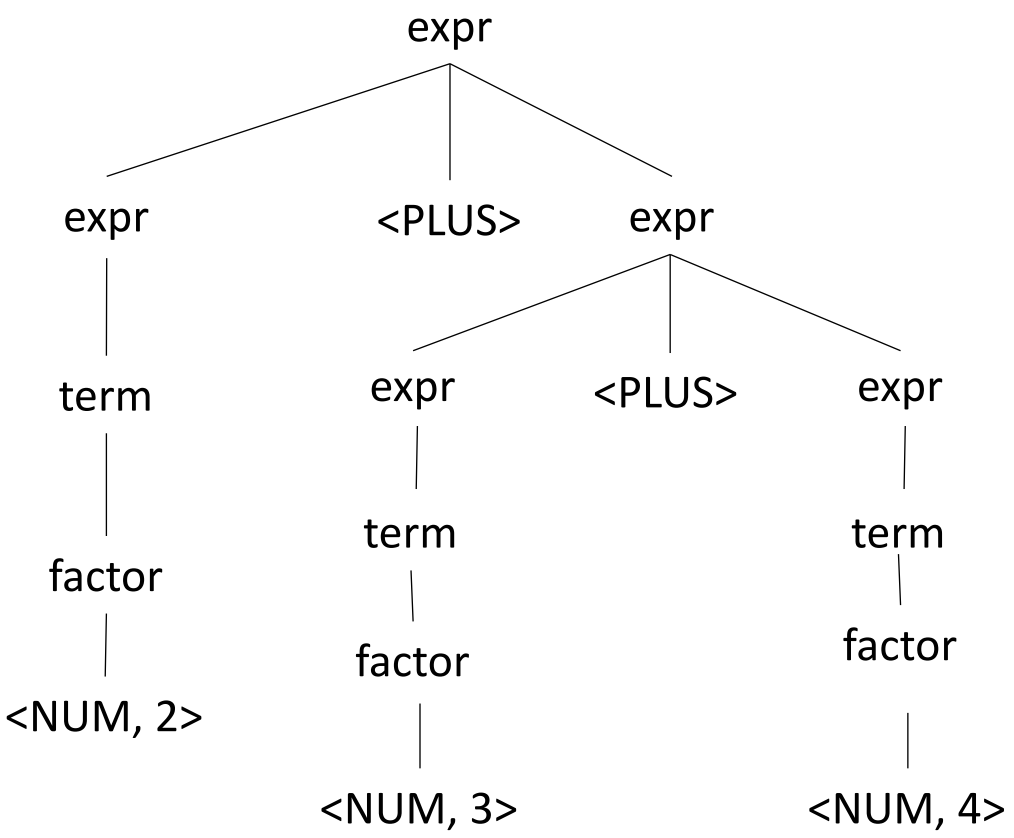


○ E → x + E
E → x

input: 2+3+4

What about this one?



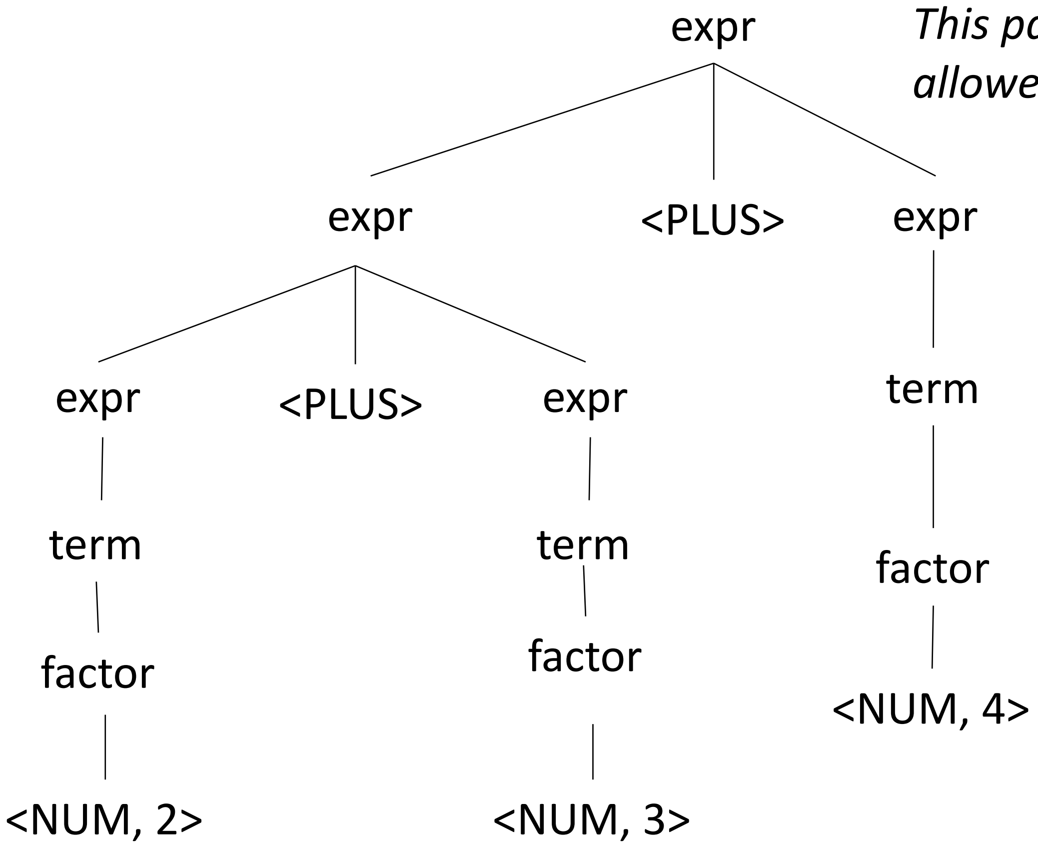


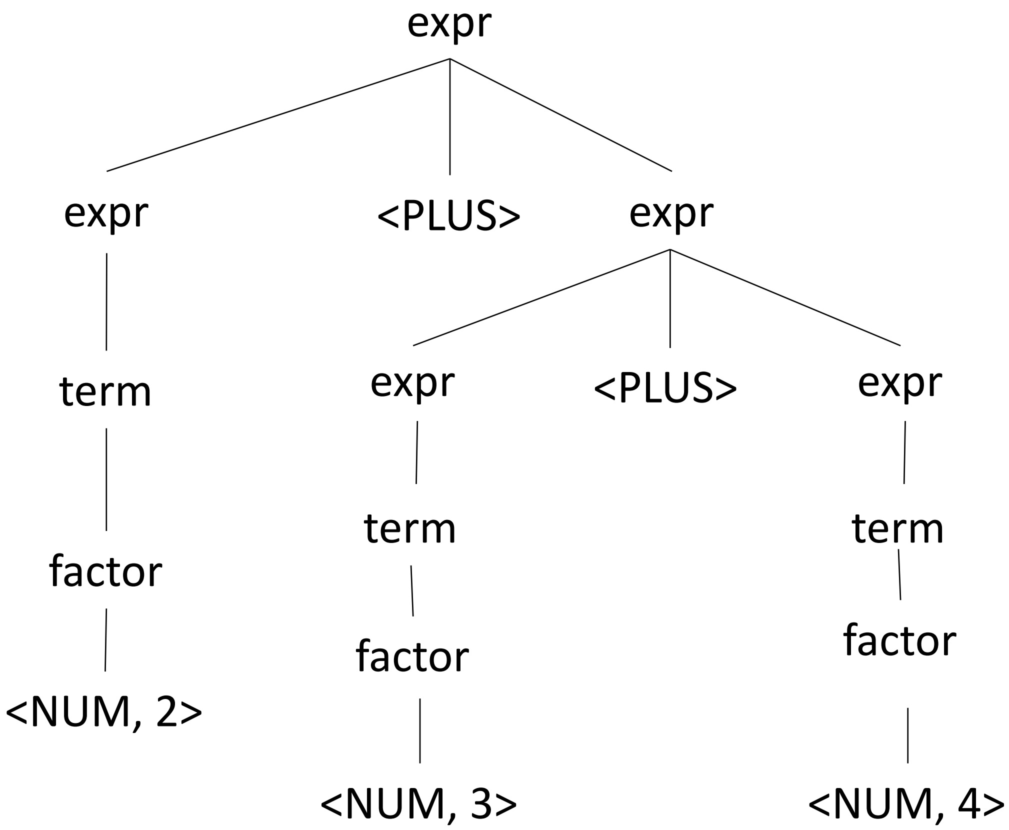
○ E → x + E
E → x

input: 2+3+4

What about this one?

Doesn't allow an expression on the LHS. This parse tree is not allowed

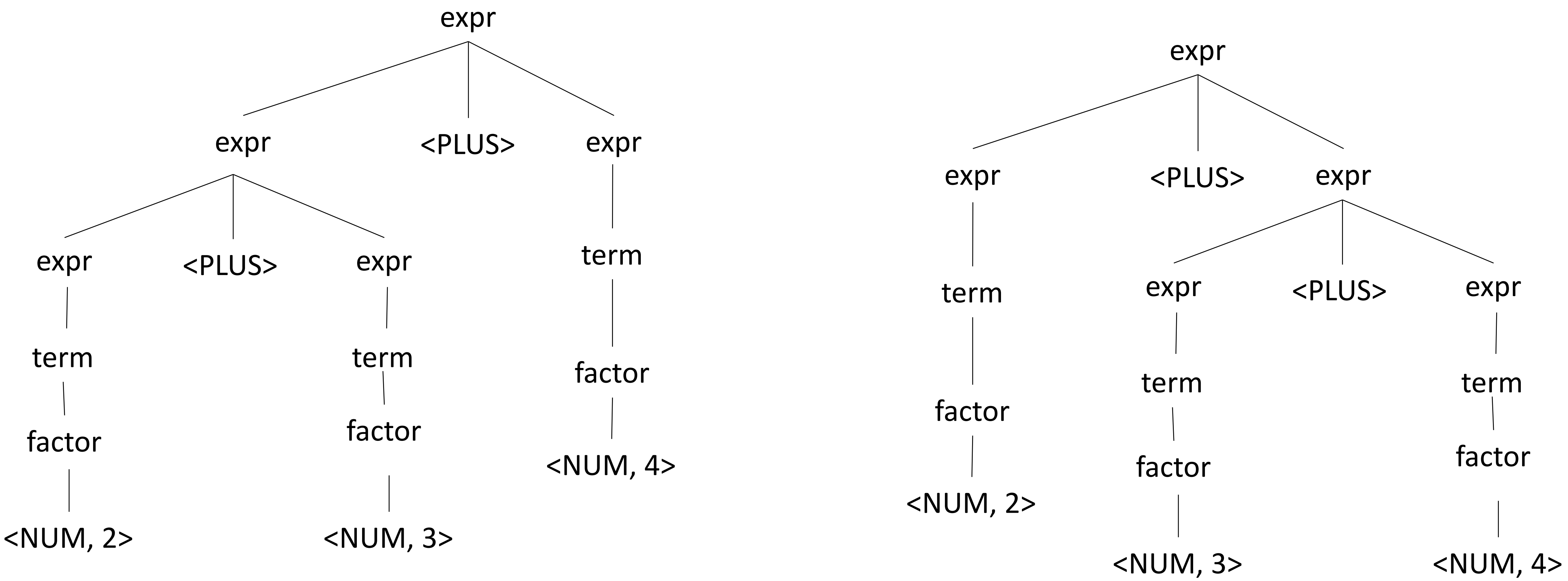




○ $E \rightarrow x + x$
$E \rightarrow x$

`input: 2+3+4`

What about this one?

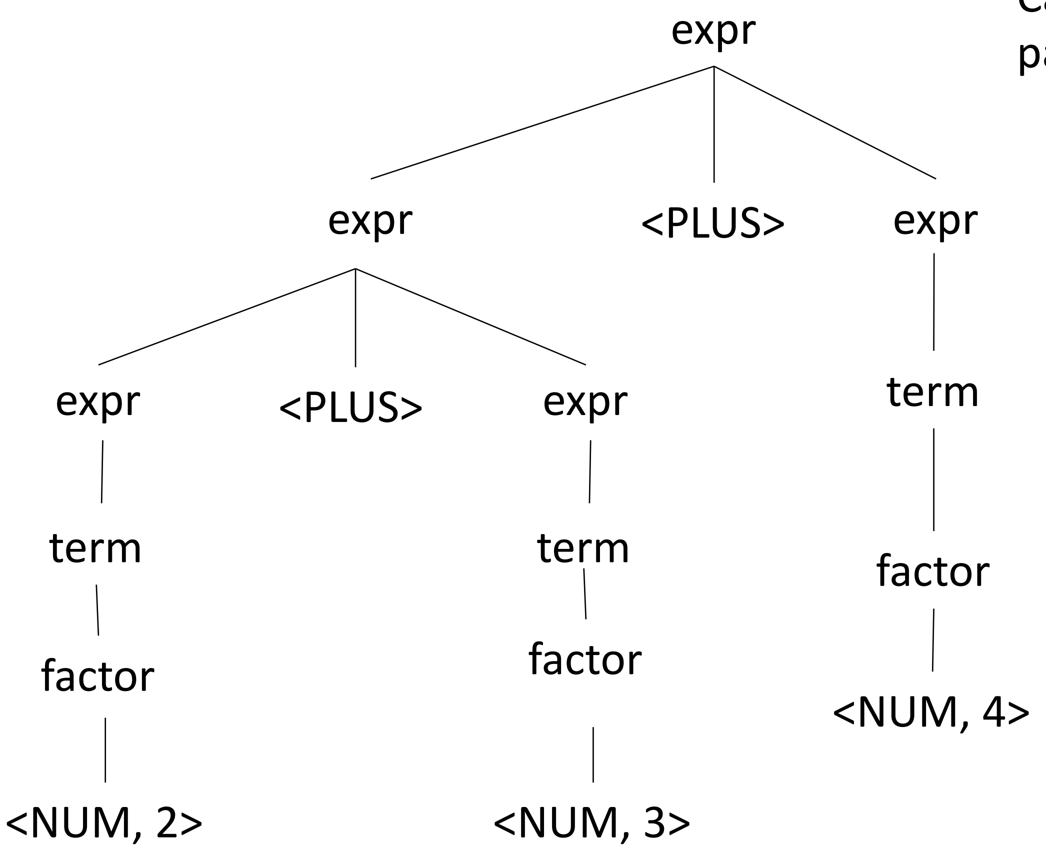


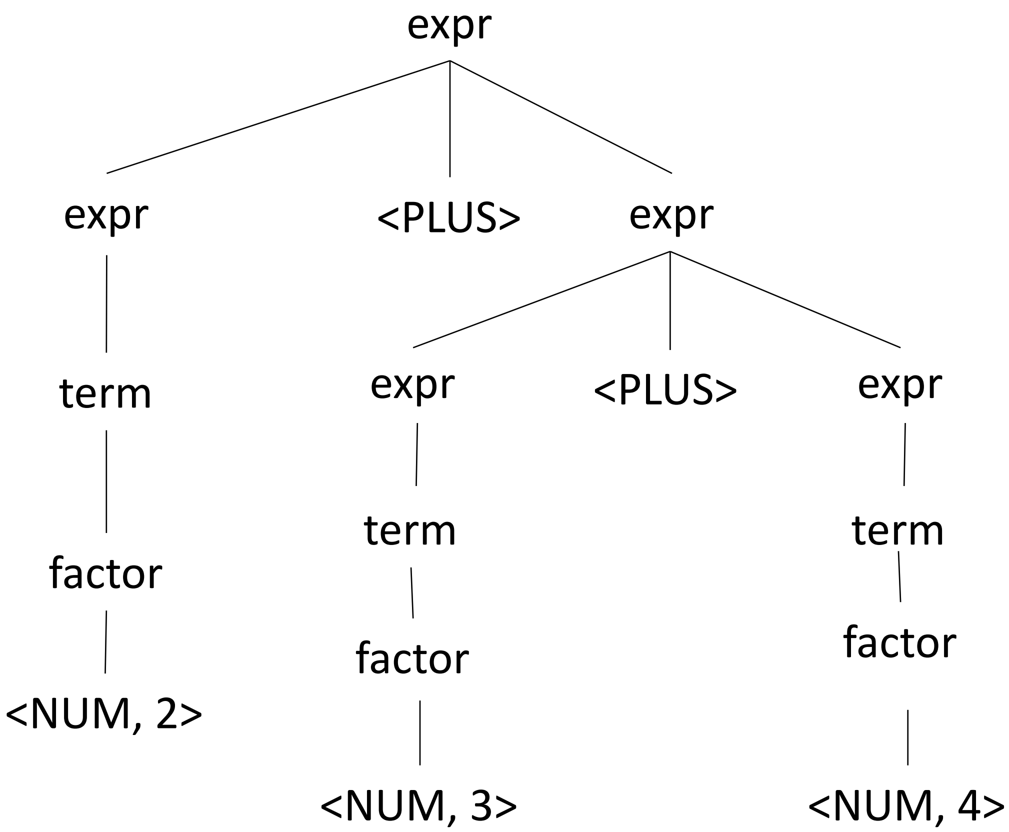
○ $E \rightarrow x + x$
$E \rightarrow x$

input: 2+3+4

What about this one?

Cannot produce either parse tree!



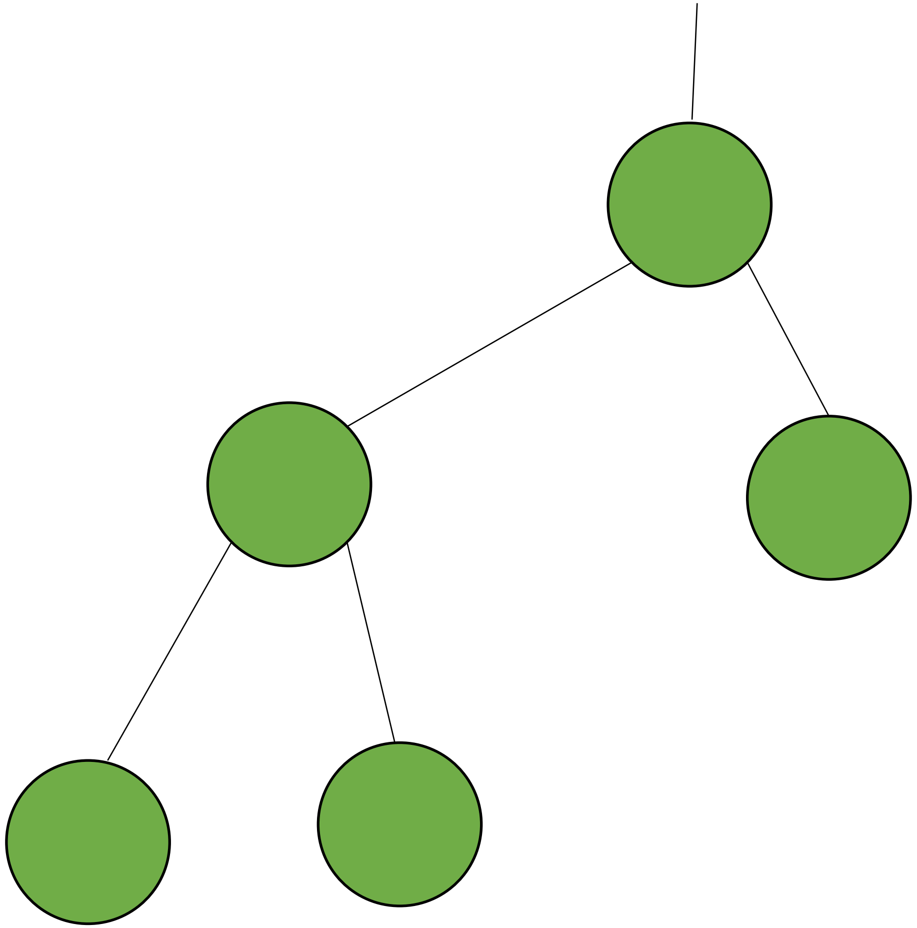


Quiz

operators with higher precedence should appear in production rules that appear higher in the parse tree

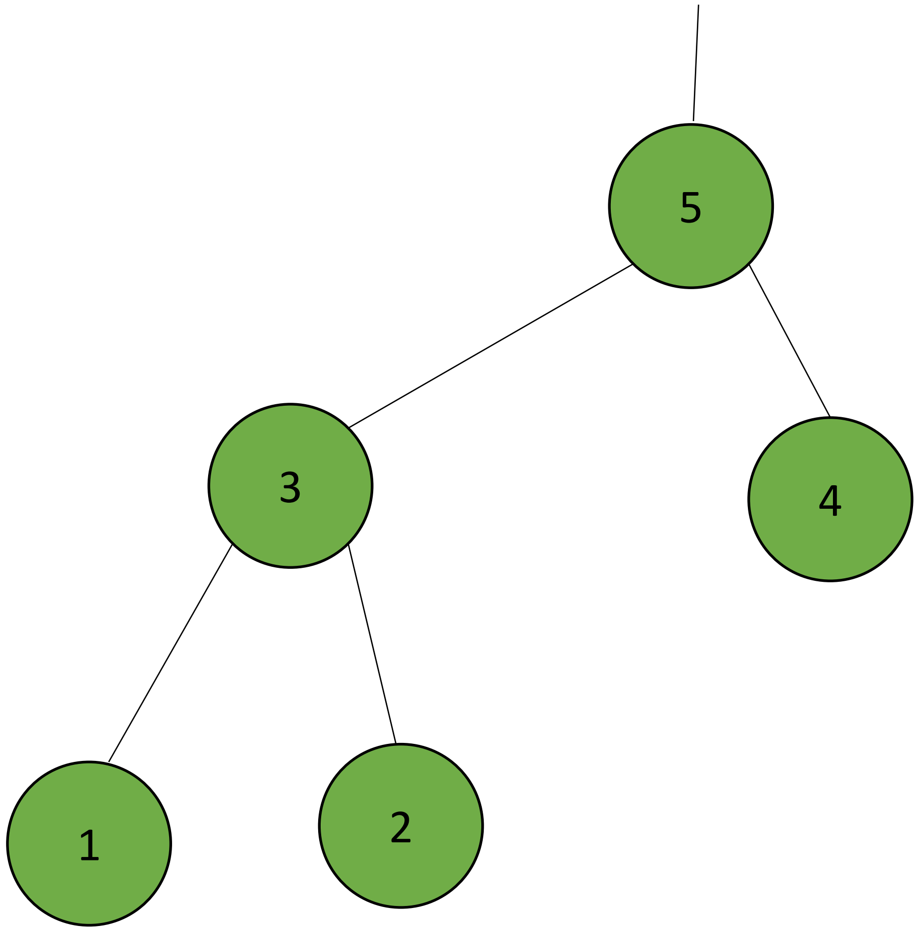
- ○ True
- ○ False

Post order traversal



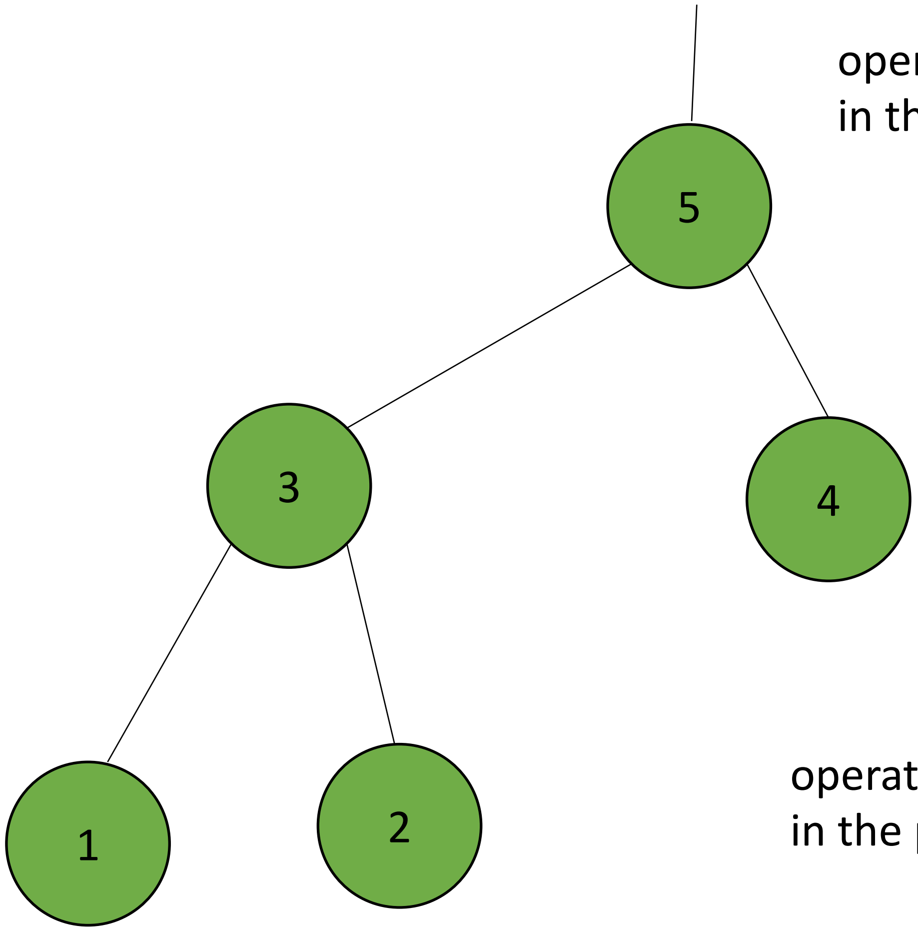
What is the post order traversal of this tree?

Post order traversal



What is the post order traversal of this tree?

Post order traversal



operators that are higher
in the parse tree get evaluated later

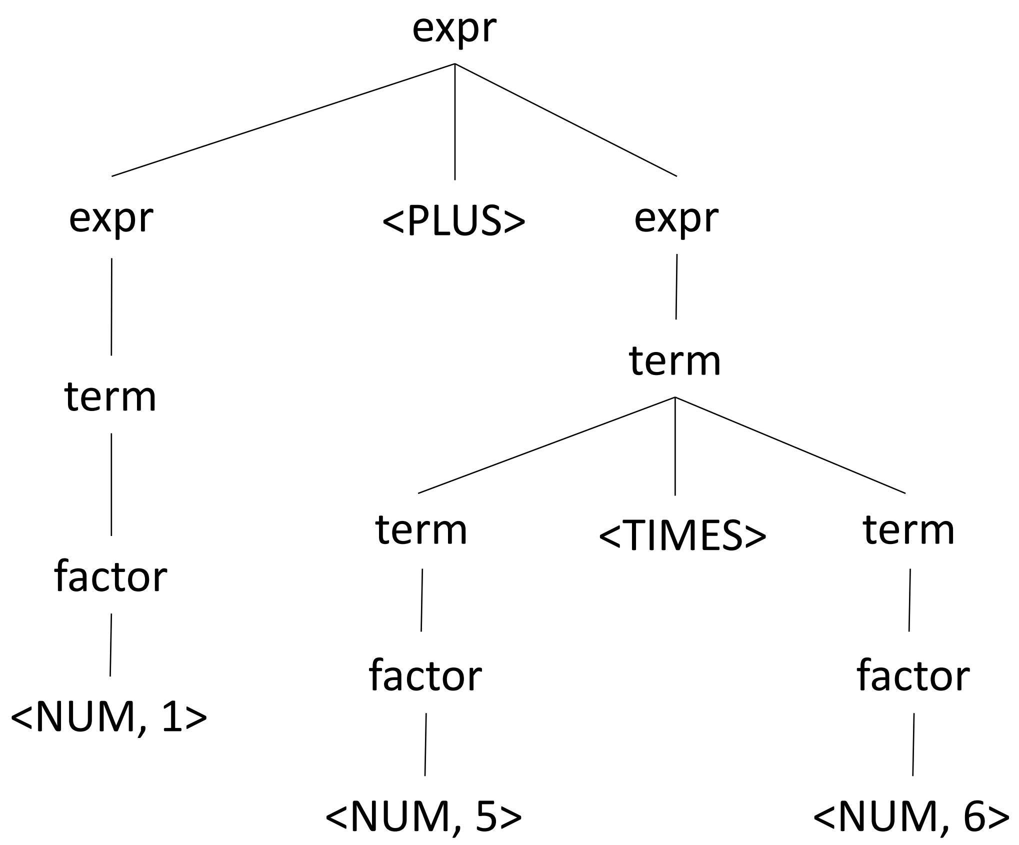
What is the post order traversal
of this tree?

operators that are lower
in the parse tree get evaluated earlier

Evaluating a parse tree

`input: 1+5*6`

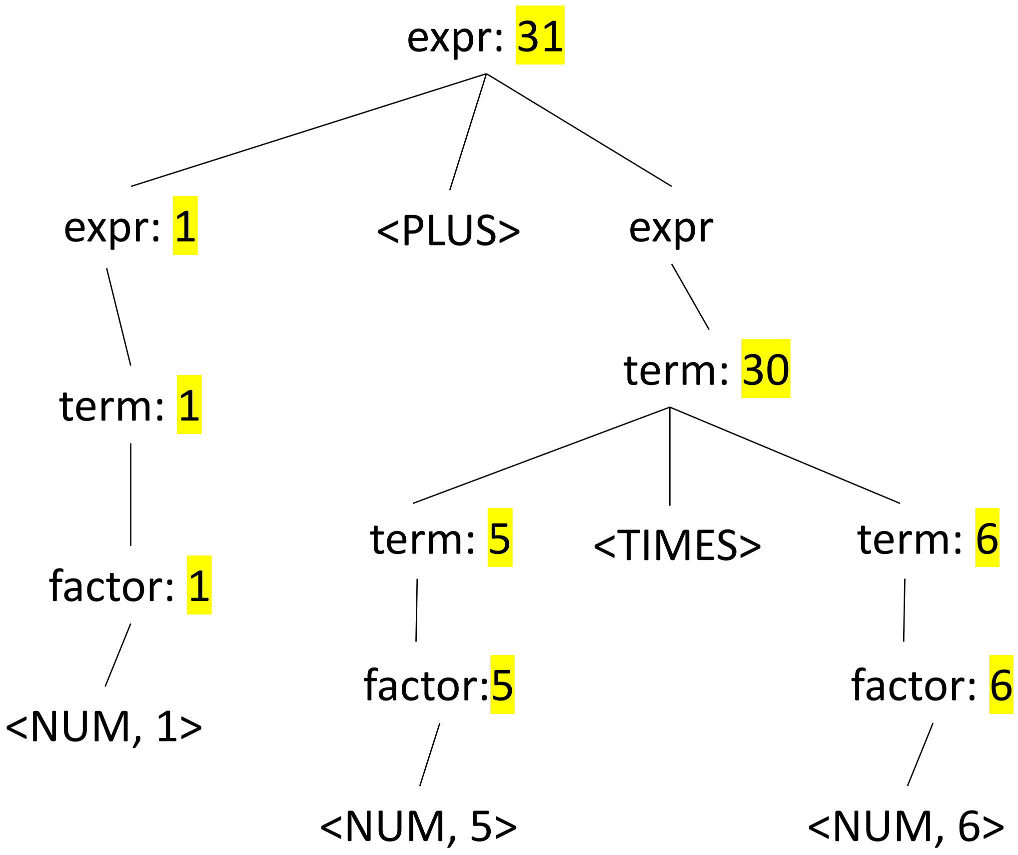
Operator	Name	Productions
+	`expr`	`: expr PLUS expr` `| term`
*	`term`	`: term TIMES term` `| factor`
()	`factor`	`: LPAREN expr RPAREN` `| NUM`



Evaluating a parse tree

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

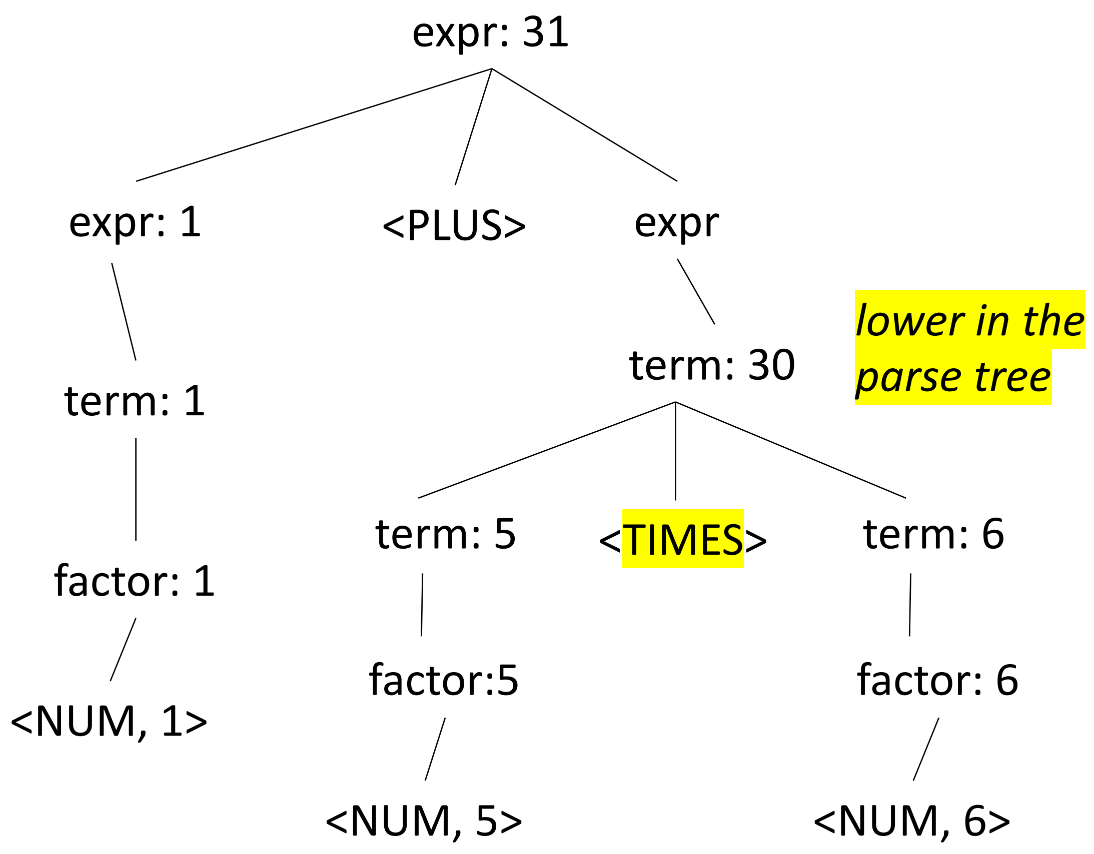
```
input: 1+5*6
```



Evaluating a parse tree

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

`input: 1+5*6`



Avoiding Ambiguity

Precedence increases going down

- new production rules
  - One non-terminal for each level of precedence
  - lowest precedence at the top
  - highest precedence at the bottom
- How would we add power? ^

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

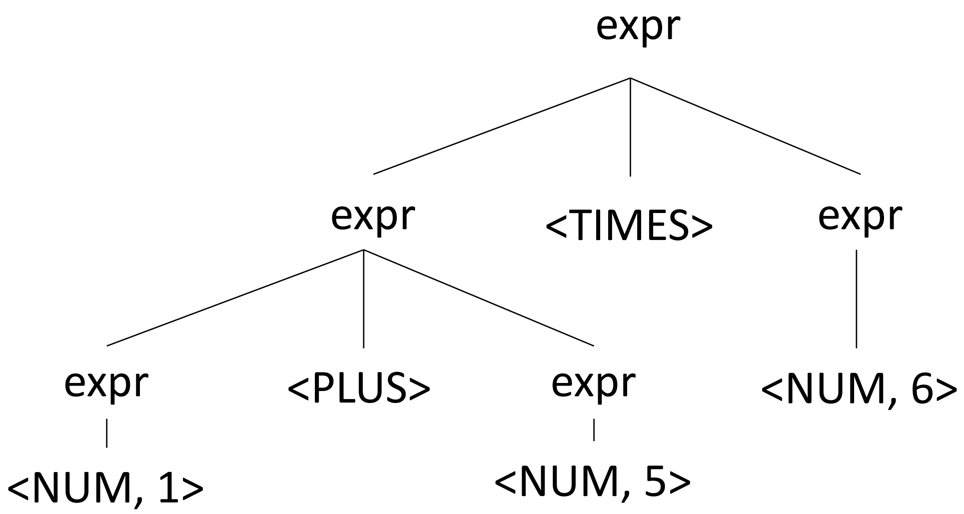
Quiz

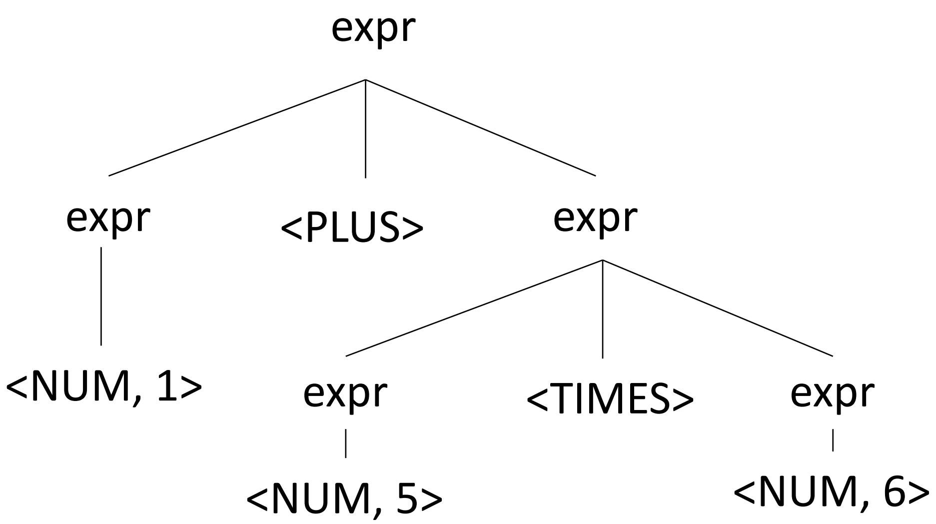
Write a few sentences about why it might be bad to have an ambiguous grammars

Ambiguous grammars

- `input: 1 + 5 * 6`

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



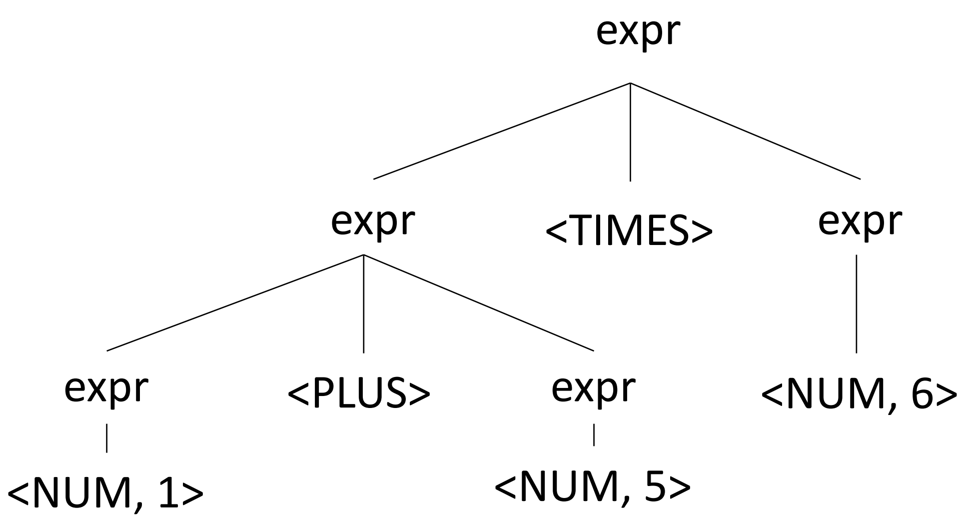


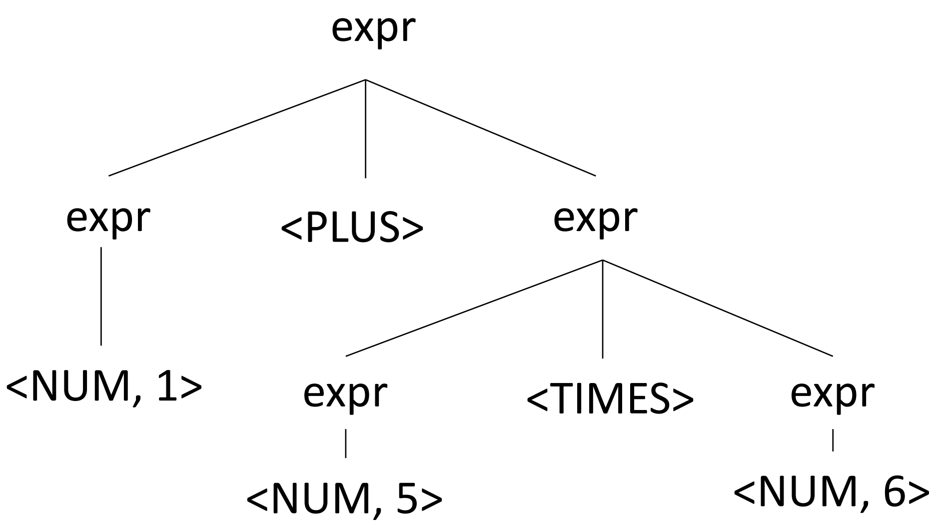
Ambiguous grammars

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

- `input: 1 + 5 * 6`

Evaluations are different!





Review

- I think the quiz did a good job covering last lectures material

New material

- Continue our discussion on associativity

Let’s make some more parse trees

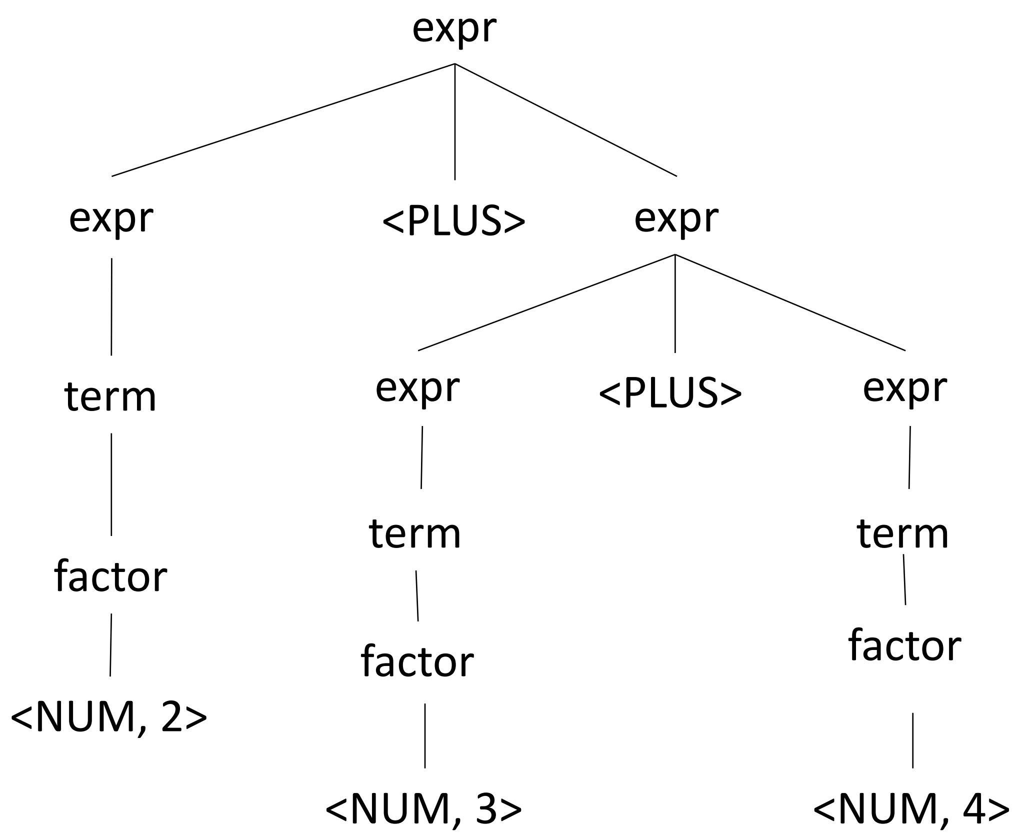
`input: 2+3+4`

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

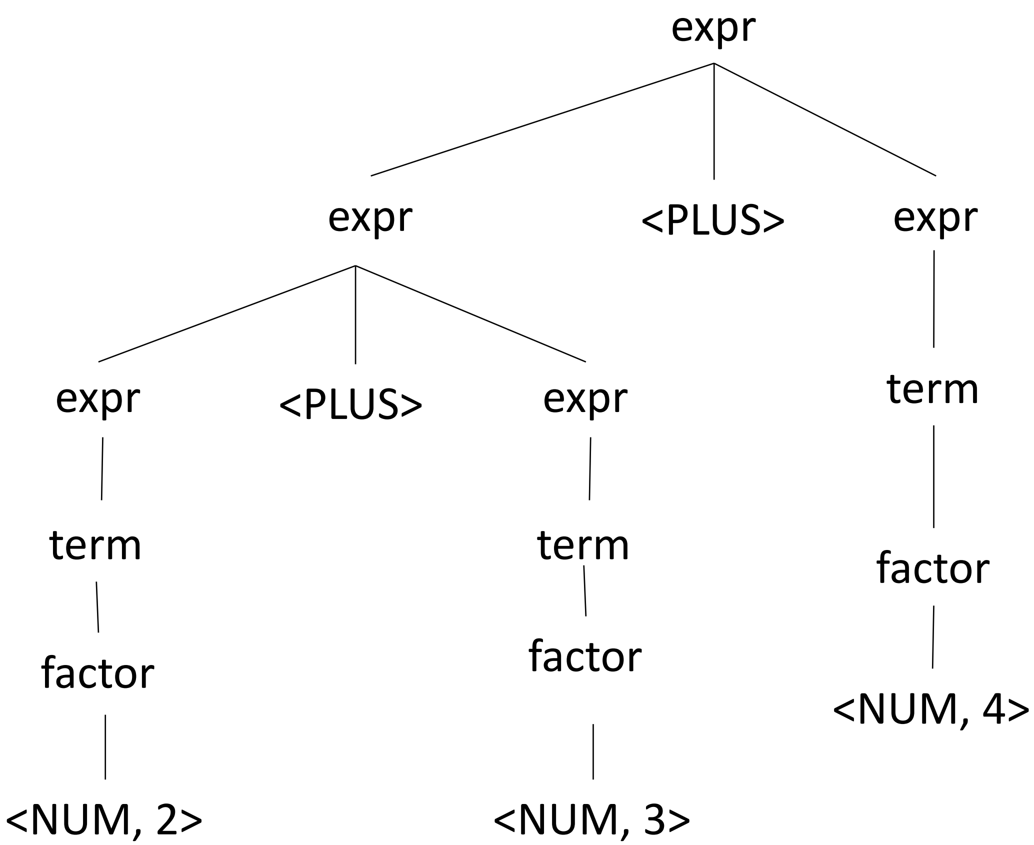
Operator	Name	Productions
+	expr	: expr PLUS expr | term
*	term	: term TIMES term | factor
()	factor	: LP expr RP | NUM

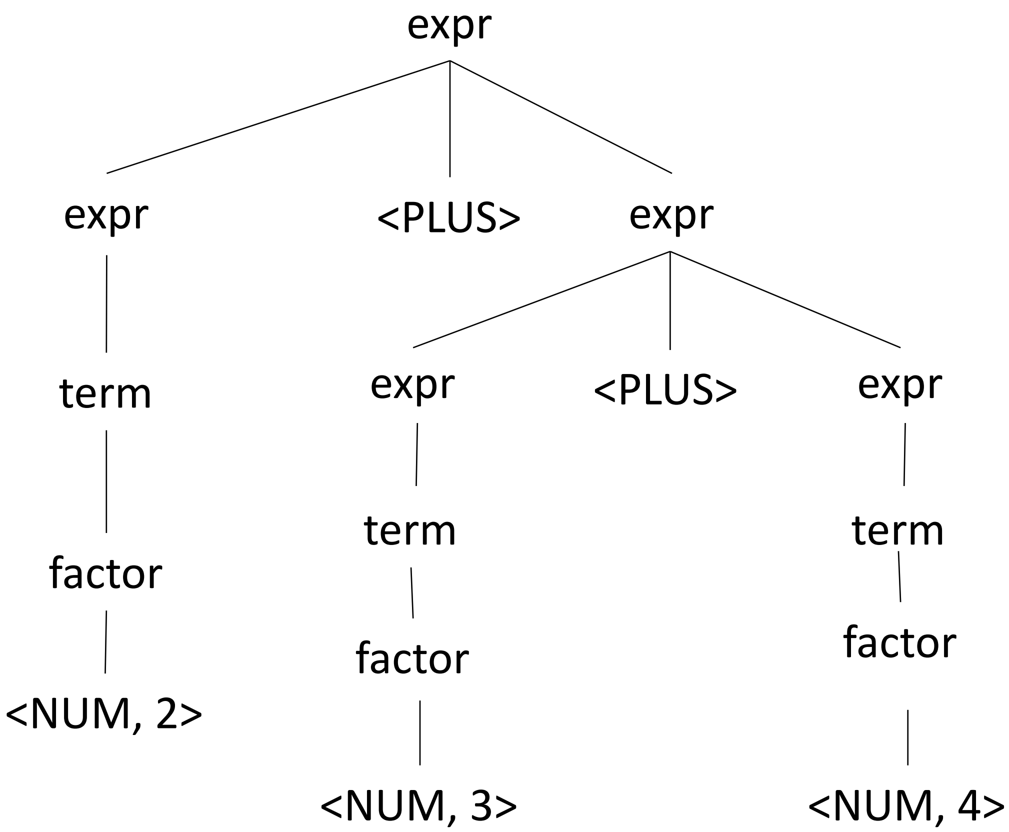
input: 2+3+4



This is ambiguous, is it an issue?

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



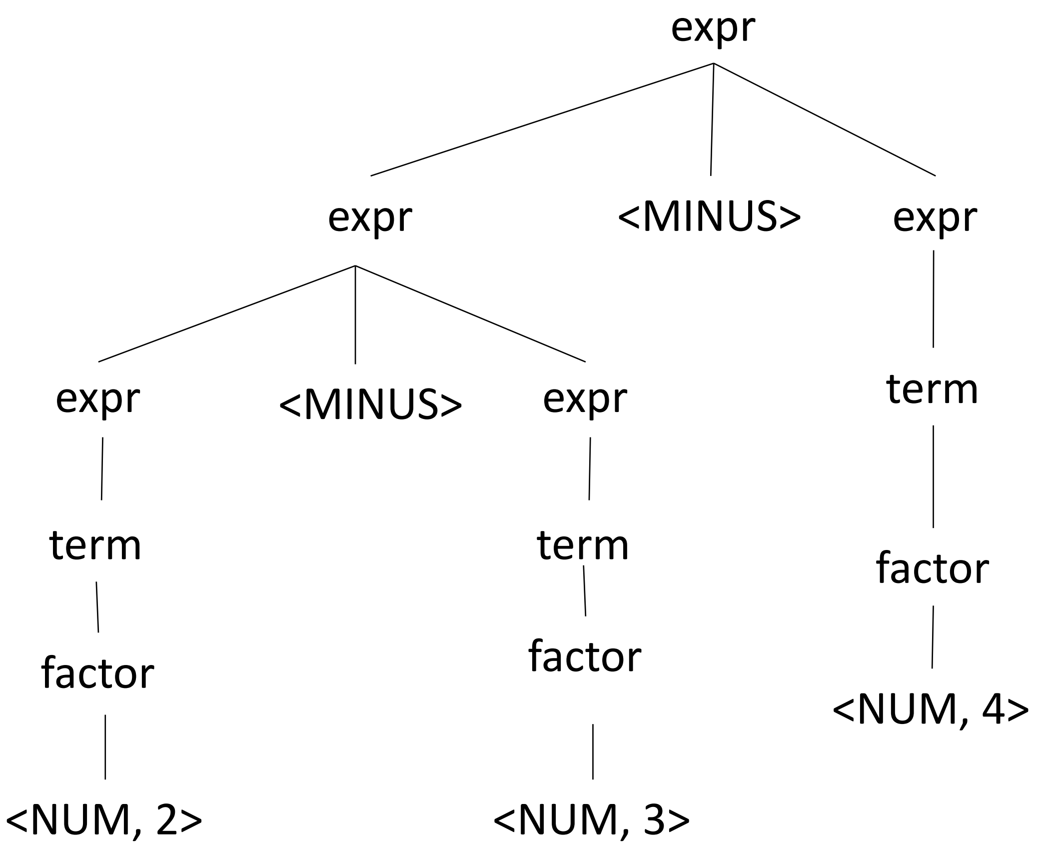


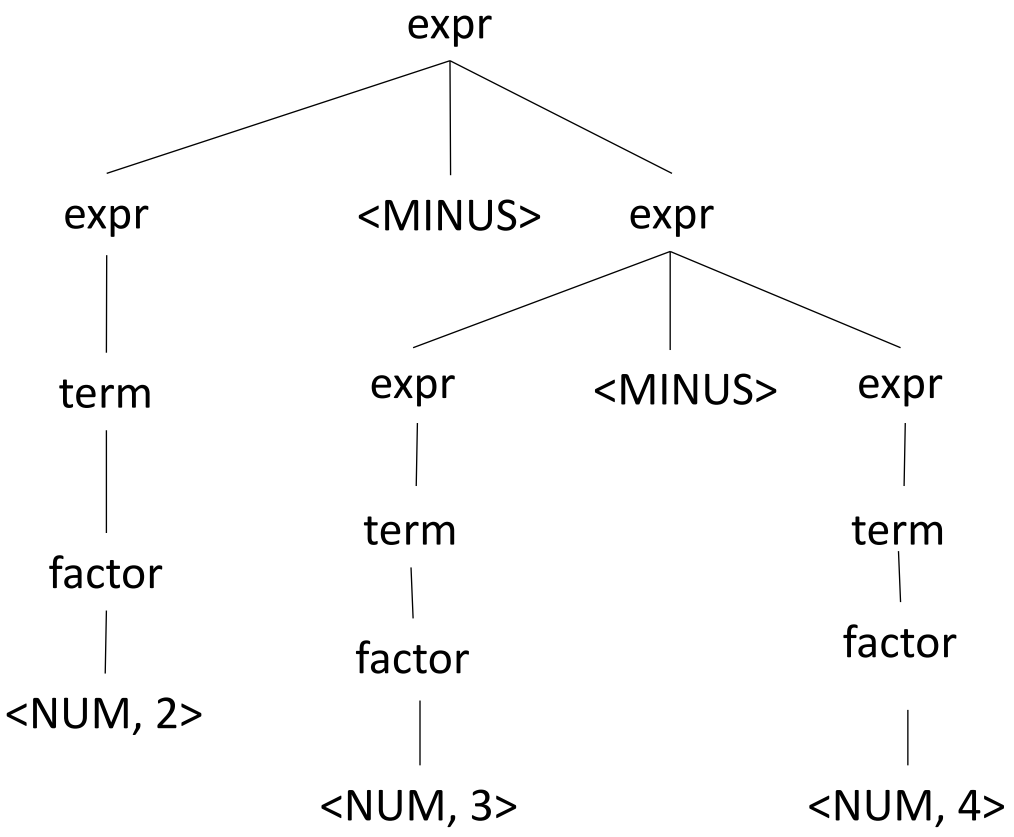
What about for a different operator?

`input: 2-3-4`

What about for a different operator?

`input: 2-3-4`



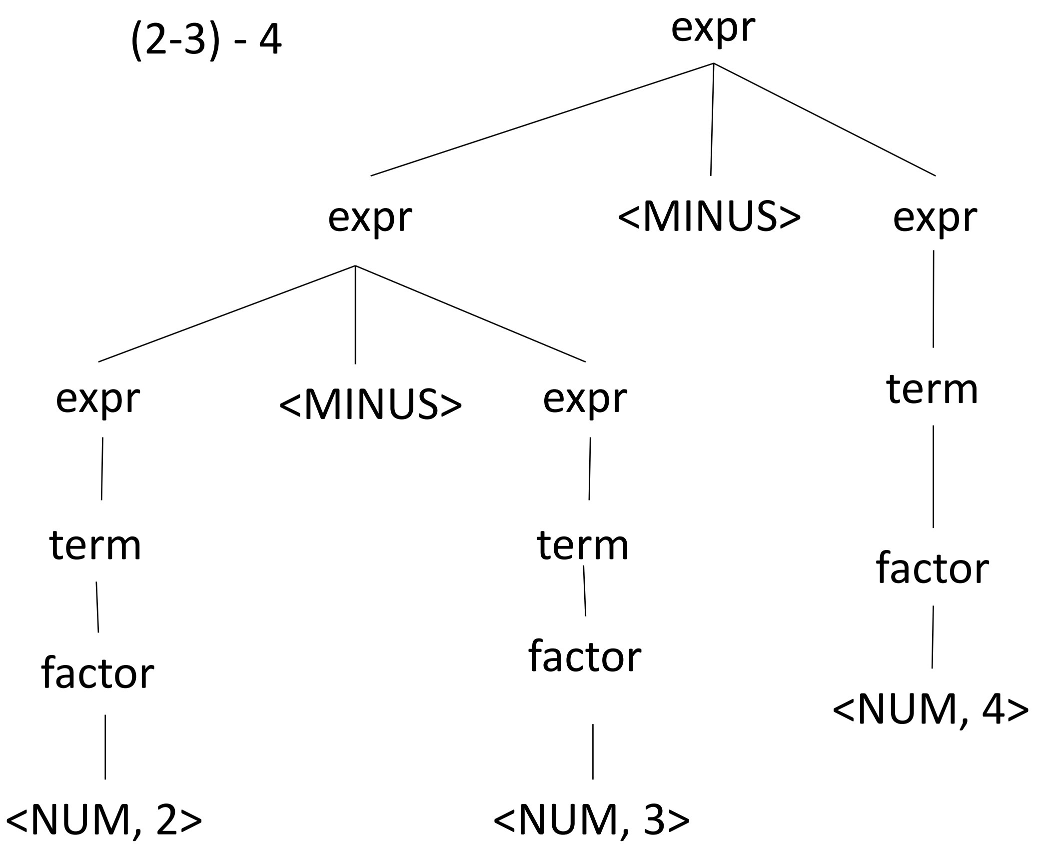


Which one is right?

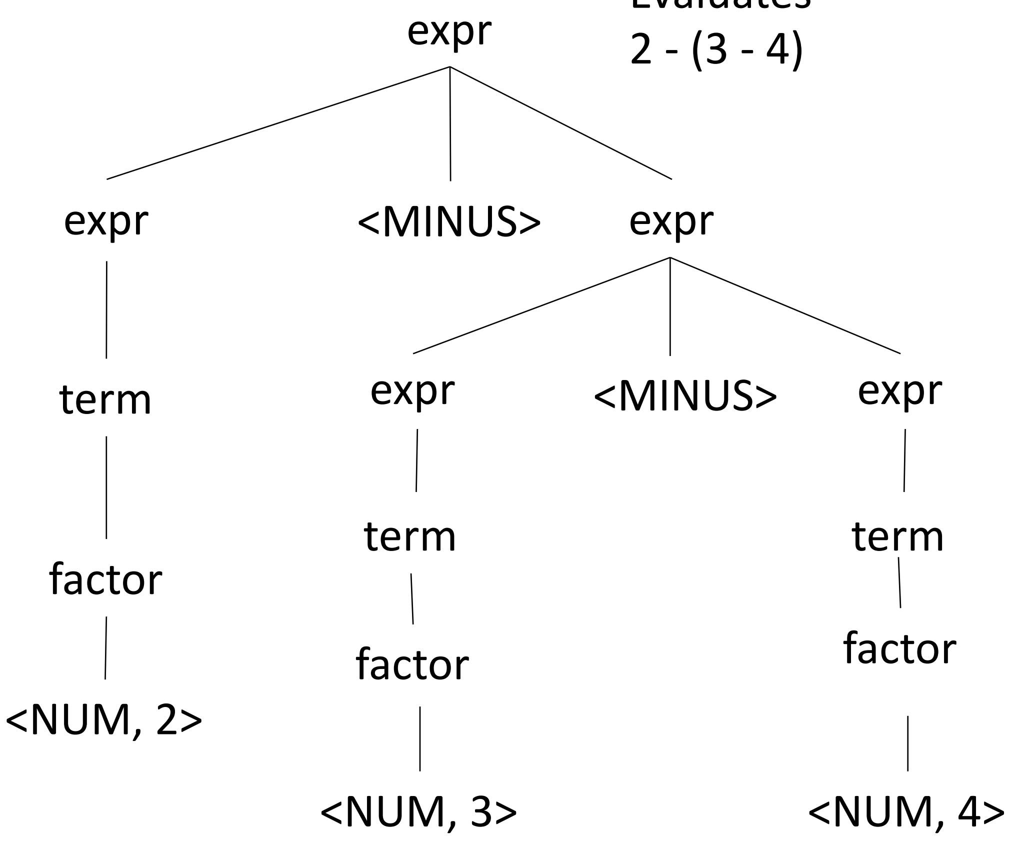
What about for a different operator?

`input: 2-3-4`

Evaluates
(2-3) - 4



Evaluates
2 - (3 - 4)



Which one is right?

Associativity

If an operator is not 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?

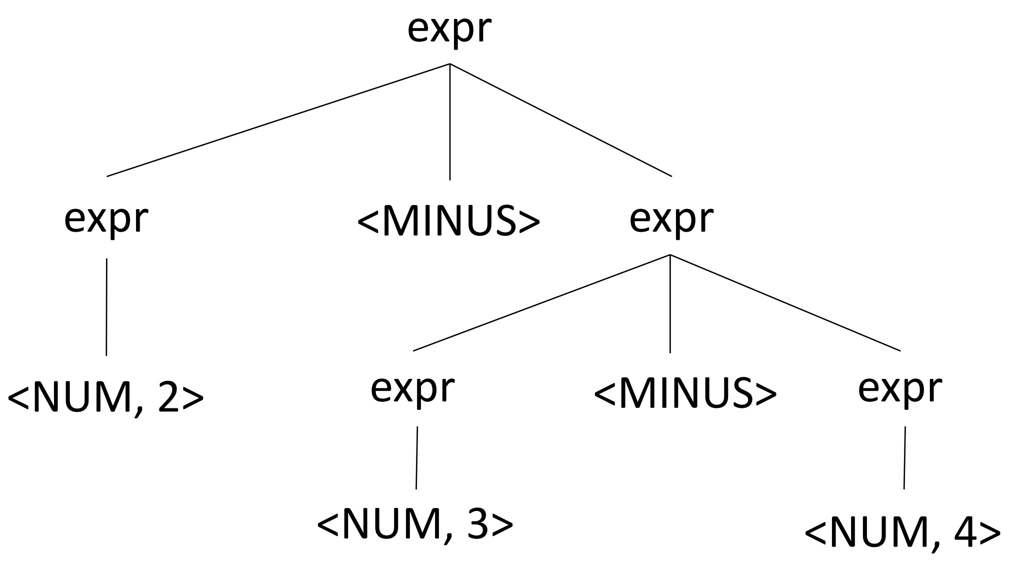
- Like precedence, some tools (e.g. YACC) allow associativity specification through keywords:
  - “+”: left, “^”: right

- Also 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`

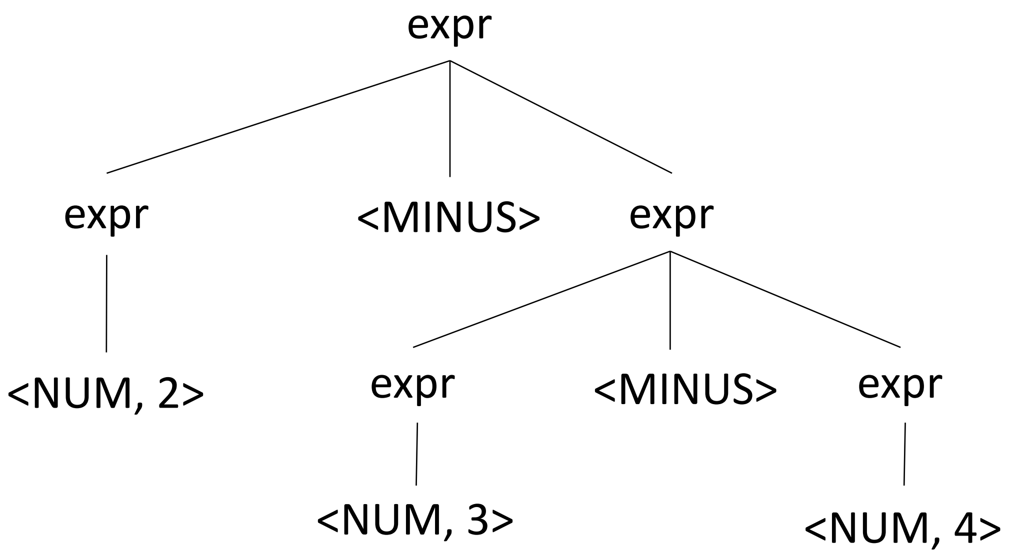


We want to disallow this parse tree

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>

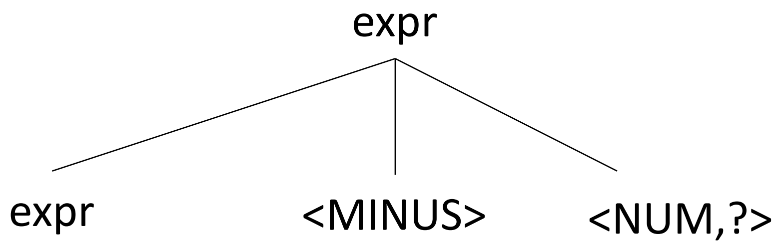
`input: 2-3-4`



No longer allowed

Associativity for a single operator

`input: 2-3-4`

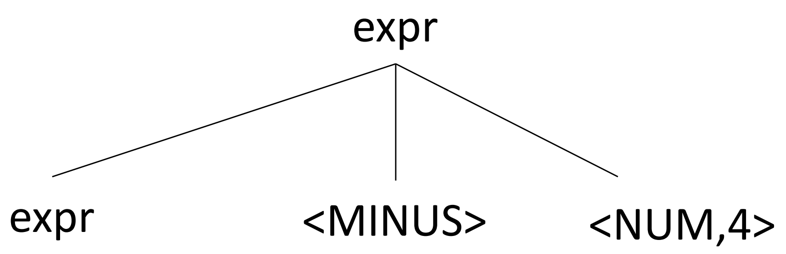


<table>
<tr><th>Operator</th><th>Name</th><th>Productions</th></tr>
<tr><td>-</td><td>expr</td><td>: expr MINUS NUM
| NUM</td></tr>
</table>

Lets start over

Associativity for a single operator

`input: 2-3-4`

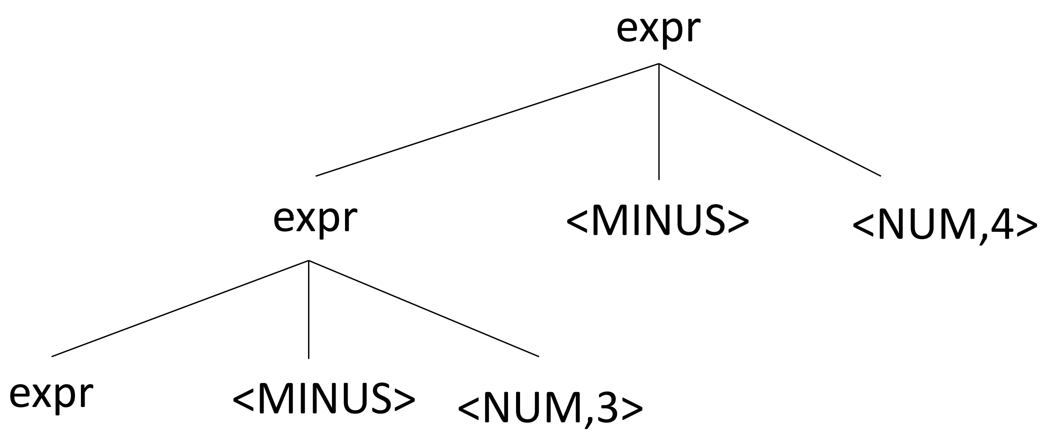


Operator	Name	Productions
-	`expr`	`: expr MINUS NUM` `| NUM`

Associativity for a single operator

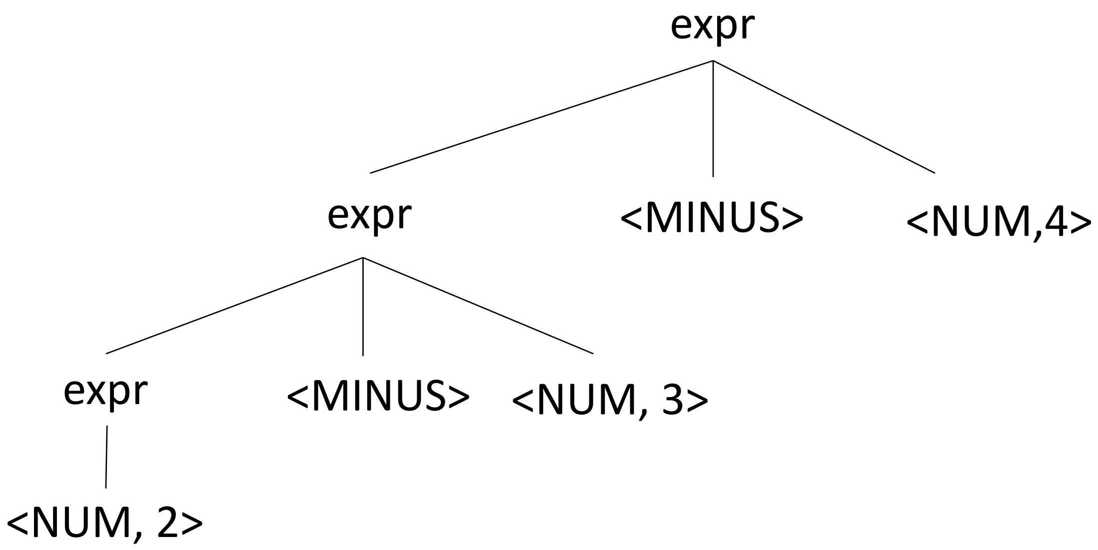
`input: 2-3-4`

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



Associativity for a single operator

`input: 2-3-4`



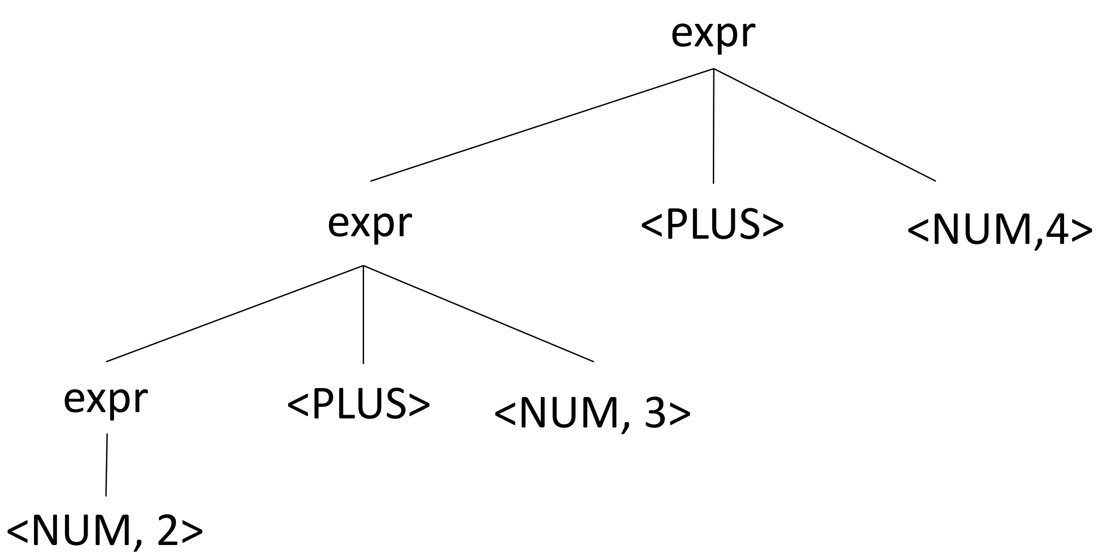
<table>
<tr><th>Operator</th><th>Name</th><th>Productions</th></tr>
<tr><td>-</td><td><code>expr</code></td><td><code>: expr MINUS NUM</code>
<code>| NUM</code></td></tr>
</table>

Should you have associativity when its not required?

`input: 2+3+4`

Benefits?
Drawbacks?

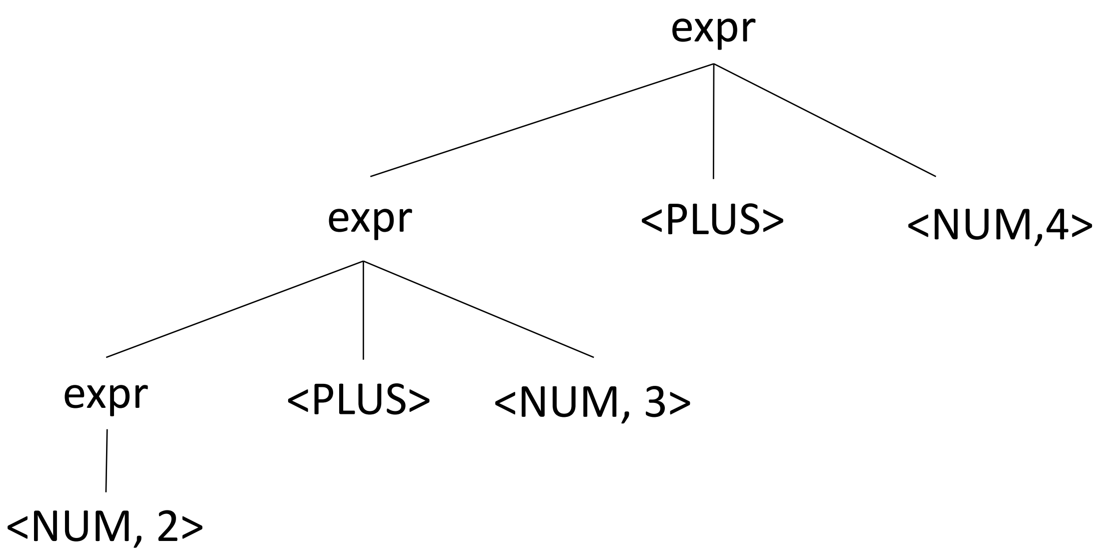
Operator	Name	Productions
+	`expr`	`: expr PLUS expr` `| NUM`



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 associativity do operators in C have?

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

New topic: Algorithms for parsing

New topic: Algorithms for parsing

One goal:

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

- We will do that be implicitly attempting to derive a parse tree for *s*

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

Top-down parsing

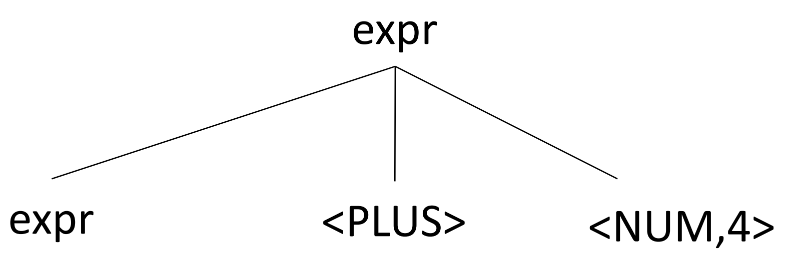
`input: 2+3+4`

expr

Operator	Name	Productions
+	expr	: expr PLUS NUM | NUM

Top-down parsing

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

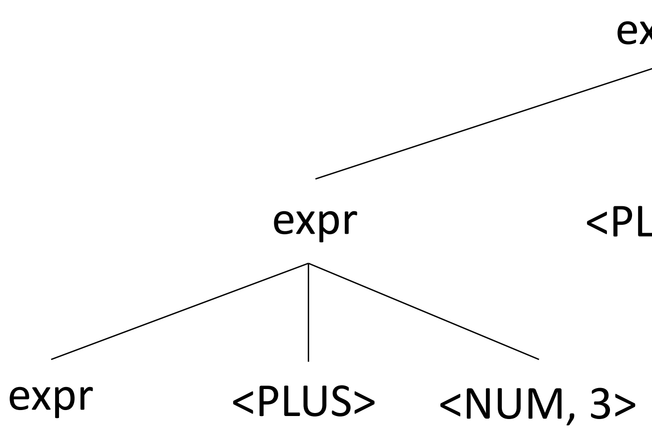


Operator	Name	Productions
+	`expr`	`: expr PLUS NUM` `| NUM`

Top-down parsing

`input: 2+3+4`

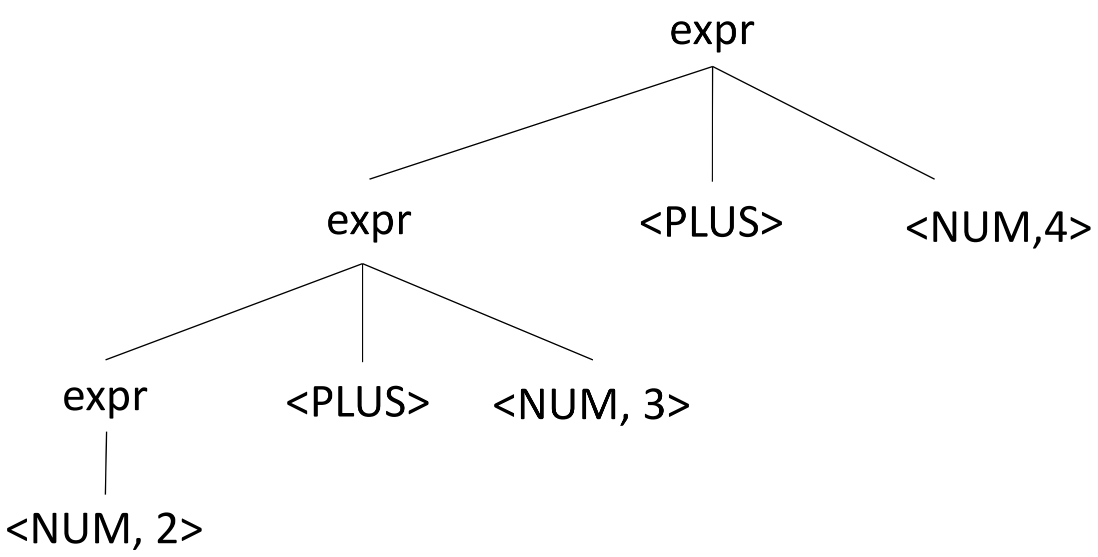
Operator	Name	Productions
+	`expr`	`: expr PLUS NUM` `| NUM`



Top-down parsing

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

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



Top-down parsing

Pros:

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

Cons:

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

Bottom-up parsing

`input: 2+3+4`

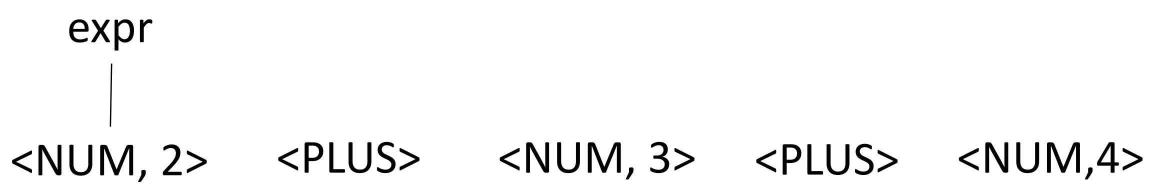
Operator	Name	Productions
+	`expr`	`: expr PLUS NUM` `| NUM`

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

Bottom-up parsing

`input: 2+3+4`

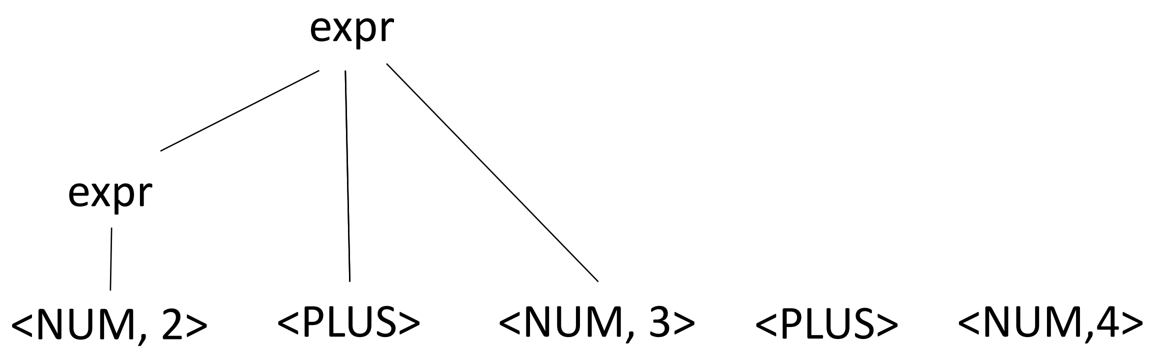
Operator	Name	Productions
+	`expr`	`: expr PLUS NUM` `| NUM`



Bottom-up parsing

`input: 2+3+4`

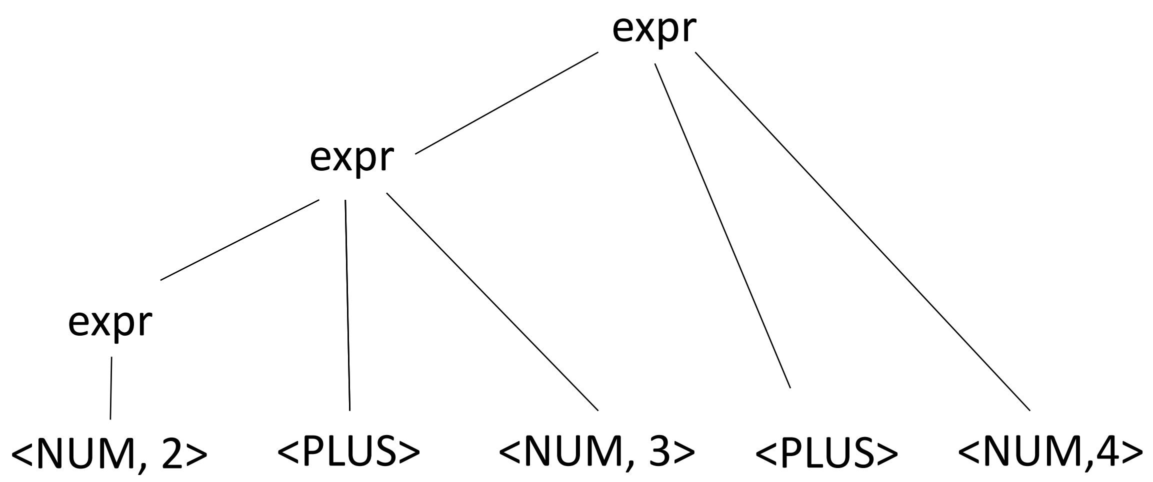
Operator	Name	Productions
+	`expr`	`: expr PLUS NUM` `| NUM`



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

Let’s start with top down

```
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
```

Variable	Value
focus	
to_match	
s.istring	
stack	

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

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

Expanded Rule	Sentential Form
start	Expr

```
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
```

Currently we assume this is magic and picks the right rule every time

Variable	Value
focus	
to_match	
s.istring	
stack	

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

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

Expanded Rule	Sentential Form
start	Expr

```
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
```

Currently we assume this is magic and picks the right rule every time

```
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	Op
to_match	‘+’
s.istring	*b)*c*
stack	Unit ‘)’ Op, Expr, None

Expanded Rule	Sentential Form
start	Expr
1	Expr Op Unit
2	Unit Op Unit
3	'(’ Expr ‘)’ Op Unit
1	‘(‘ Expr Op Unit ‘)’ Op Unit
2	‘(‘ Unit Op Unit ‘)’ Op Unit
4	‘(‘ ID Op Unit ‘)’ Op Unit

And so on…

```
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 if we don't have a magic choice

```
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

```
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

Top down parsing does not handle left recursion

- In general, any CFG can be re-written without left recursion

Eliminating direct left recursion

```
Fee ::= Fee “a”
     |   “b”
```

What does this grammar describe?

Eliminating direct left recursion

```
Fee ::= Fee “a”
    |   “b”
```

The grammar can be rewritten as

```
Fee  ::= “b” Fee2

Fee2 ::= “a” Fee2
     |    “”
```

Eliminating direct left recursion

In general, A and B can be any sequence of non-terminals and terminals

```
Fee ::= Fee A
     |   B
```

```
Fee  ::= B Fee2

Fee2 ::= A Fee2
      |   “”
```

Eliminating direct left recursion

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

Lets do this one as an example:

```
Fee ::= Fee A
     |    B
```

→

```
Fee   ::= B Fee2

Fee2  ::= A Fee2
      |     ""
```

Eliminating direct left recursion

```
A = Op Unit
B = Unit
```

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

```
1: Expr  ::= Unit Expr2
2: Expr2 ::= Op Unit Expr2
3:        | ""
4: Unit  ::= '(' Expr ')'
5:        |    ID
6: Op    ::= '+'
7:        |   '*'
```

Lets do this one as an example:

```
Fee ::= Fee A
     |   B
```

```
Fee   ::= B Fee2

Fee2  ::= A Fee2
       |     ""
```

```
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
```

Variable	Value
focus	
to_match	
s.istring	
stack	

```
1: Expr   ::= Unit Expr2
2: Expr2  ::= Op Unit Expr2
3:          | ""
4: Unit   ::= '(' Expr ')'
5:          |   ID
6: Op     ::= '+'
7:          |   '*'
```

Expanded Rule	Sentential Form
start	Expr

```
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
```

Variable	Value
focus	
to_match	
s.istring	
stack	

How to handle this case?

```
1: Expr   ::= Unit Expr2
2: Expr2  ::= Op Unit Expr2
3:          | ""
4: Unit   ::= '(' Expr ')'
5:          |  ID
6: Op     ::= '+'
7:          |  '*'
```

Expanded Rule	Sentential Form
start	Expr

```
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);
    if A == "": focus=pop(); continue;
    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
```

How to handle this case?

```
1: Expr   ::= Unit Expr2
2: Expr2  ::= Op Unit Expr2
3:          | ""
4: Unit   ::= '(' Expr ')'
5:          |  ID
6: Op     ::= '+'
7:          | '*'
```

Variable	Value
focus	
to_match	
s.istring	
stack	

Expanded Rule	Sentential Form
start	Expr

How about indirect 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

How about indirect 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:            |   '*'
```

Identify indirect left left recursion

$$Expr\_base \rightarrow_{lhs} Expr\_op \rightarrow_{lhs} Expr\_base$$

How about indirect 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:           |   '*'
```

Identify indirect left left recursion

$$Expr\_base \rightarrow_{lhs} Expr\_op \rightarrow_{lhs} Expr\_base$$

Substitute indirect non-terminal closer to initial non-terminal

How about indirect 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:           |   '*'
```

Identify indirect left left recursion

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

What to do with production rule 3?

$$Expr\_base \rightarrow_{lhs} Expr\_op \rightarrow_{lhs} Expr\_base$$

Substitute indirect non-terminal closer to initial non-terminal

How about indirect 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:           |   ‘*’
```

Identify indirect left left recursion

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

What to do with production rule 3?
It may need to stay if another production rule references it!

$$Expr\_base \rightarrow_{lhs} Expr\_op \rightarrow_{lhs} Expr\_base$$

Substitute indirect non-terminal closer to initial non-terminal

Next time: algorithms for syntactic analysis

- Continue with our top down parser.
    - Backtracking
    - Lookahead sets