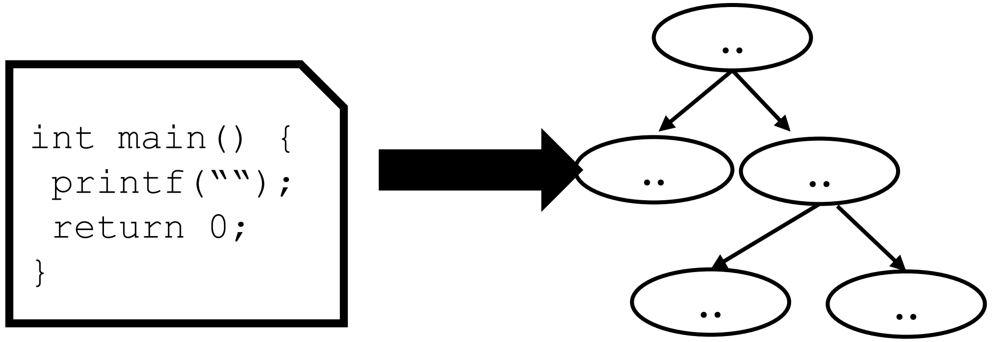
CSE110A: Compilers

Topics: Midterm Review of Parsing



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?

$\texttt{if}\ \mathit{Expr}_1\ \texttt{then}\ \texttt{if}\ \mathit{Expr}_2\ \texttt{then}\ \mathit{Assignment}_1\ \texttt{else}\ \mathit{Assignment}_2$

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

Ambiguous grammars

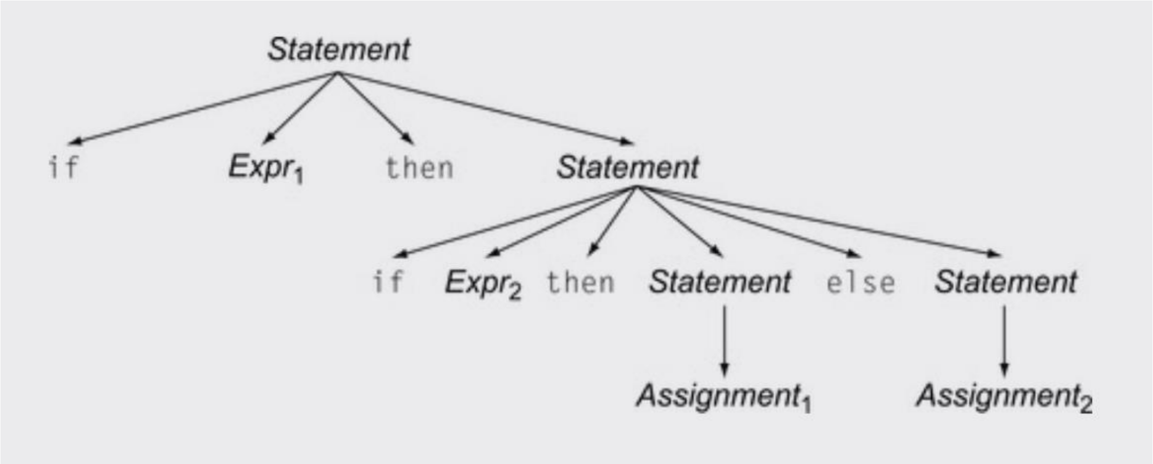
```
1: Statement ::= "if" Expr "then" Statement "else" Statement
2:           |   "if" Expr "then" Statement
3:           |    Assignment
4:           |    ....
```

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 *Expr₁* **then** **if** *Expr₂* **then** *Assignment₁* **else** *Assignment₂*

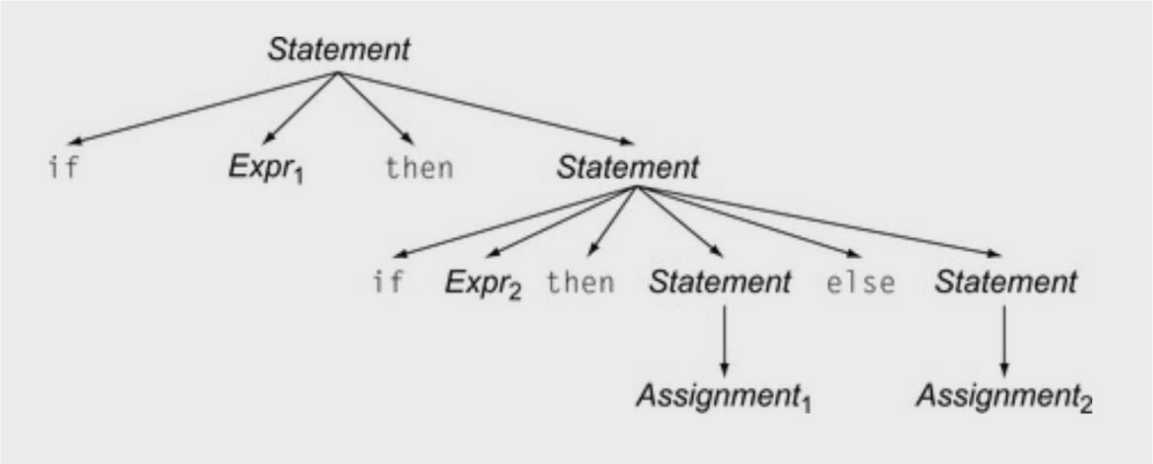


Valid derivation

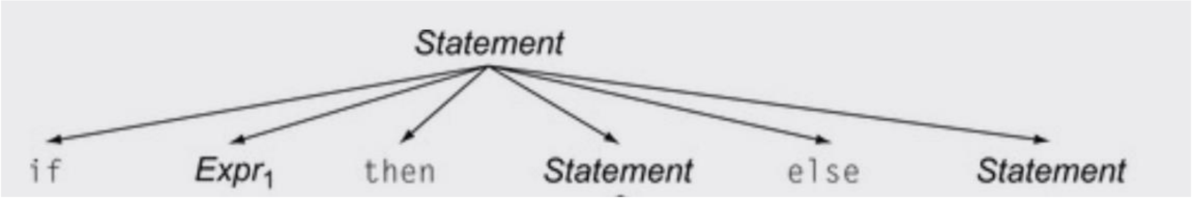
Ambiguous grammars

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

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



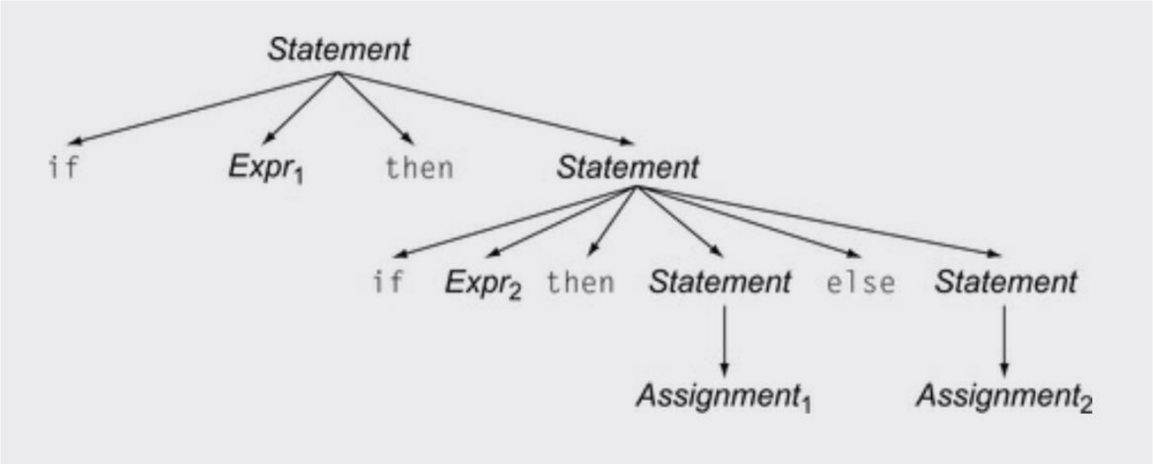
Valid derivation



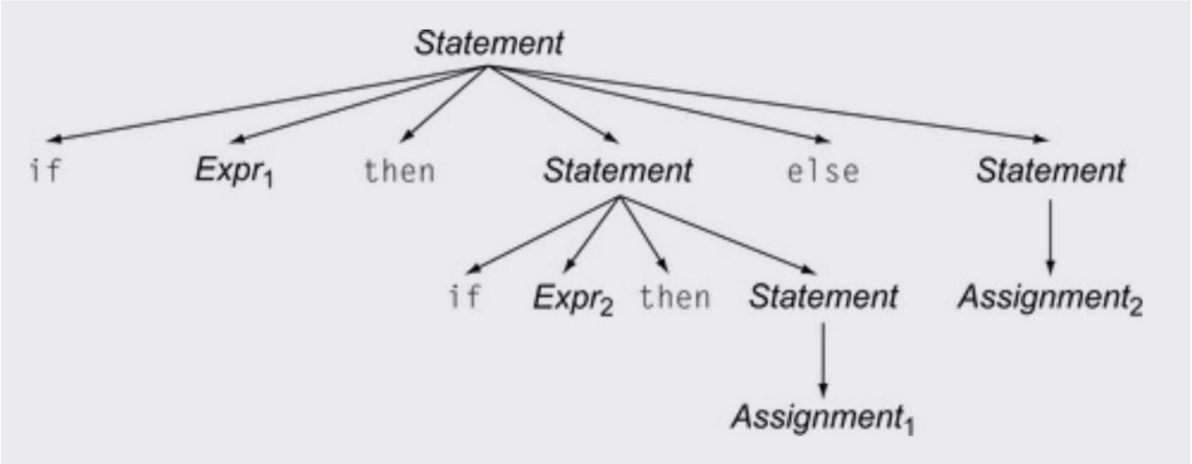
Ambiguous grammars

```
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** $Assignment_2$



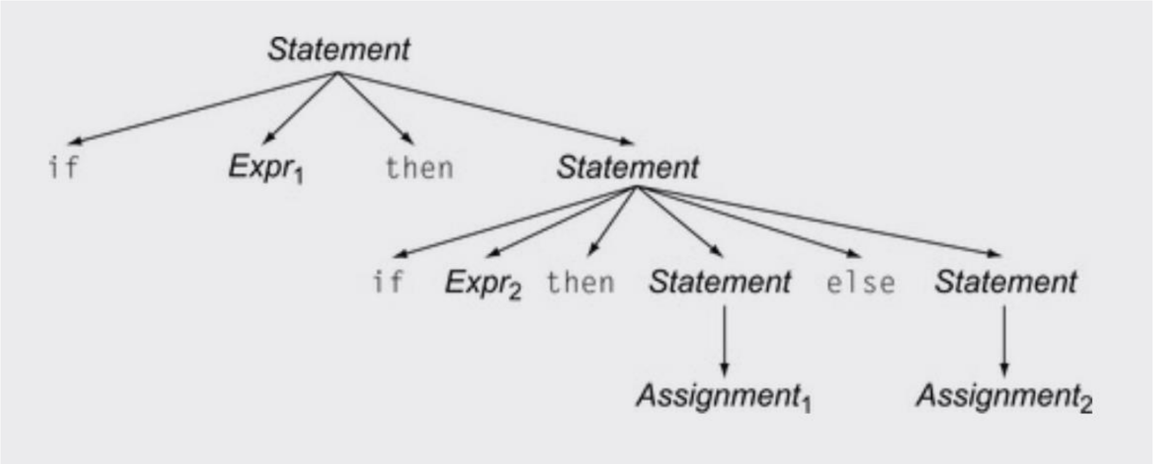
Valid derivation



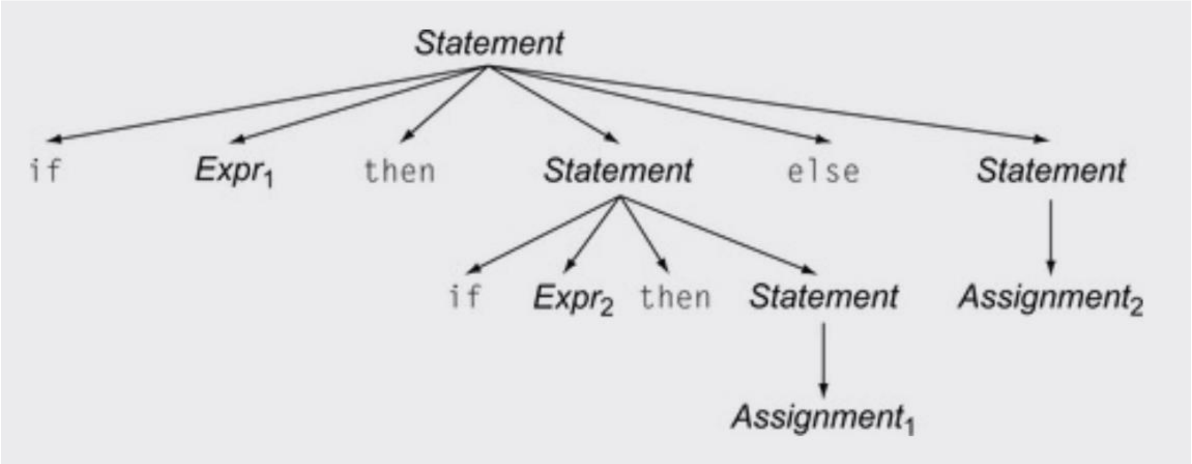
And another valid derivation

Ambiguous grammars

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



Valid derivation



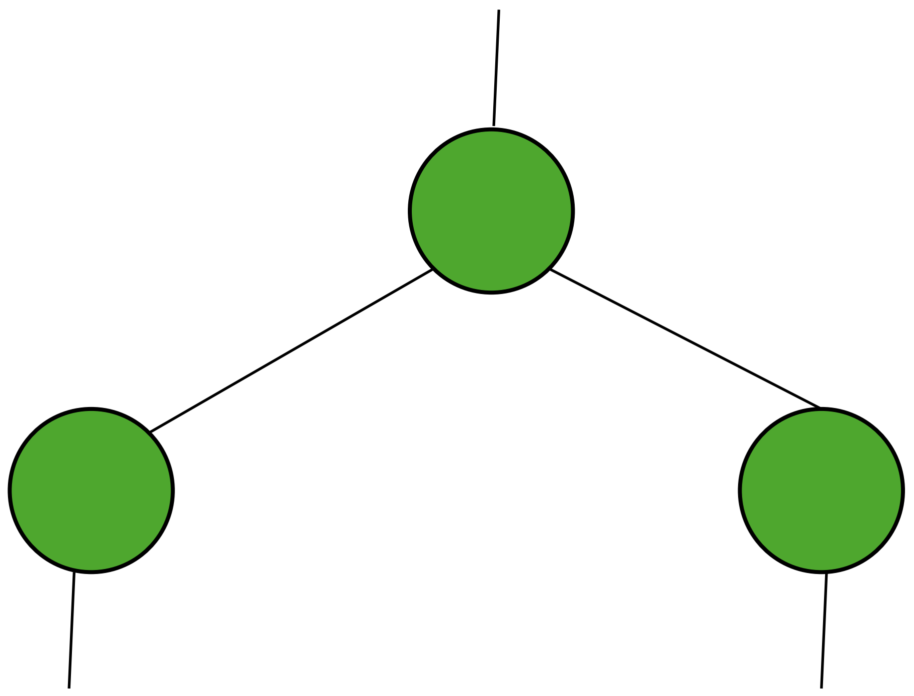
And another valid derivation

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)

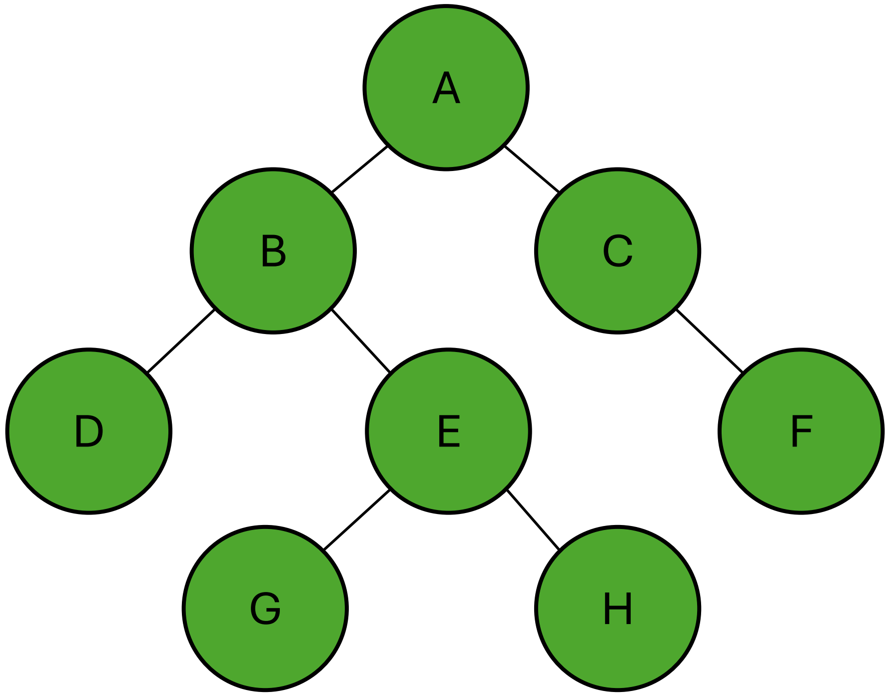
Post order traversal



visiting for for different types
of traversals:

pre order?
in order?
post order?

Review: Possible Orders of Traversal



Traversal Order	Order Visited	Example Output
pre-order	Top-Root->Left-Child->Right-Child	A B D E G H C F
in-order	Left/Bottom->Its-Root->Right/Bottom	D B G E H A C F
post-order	Left/Bottom->Right/Bottom->Its-Root	D G H E B F C A

Traversals never visit the same node twice.

Pre-order Traversal (Root-Left-Right. Top to Bottom)

Prioritizes visits from top to bottom, and left to right

Useful for serializing and copying trees.

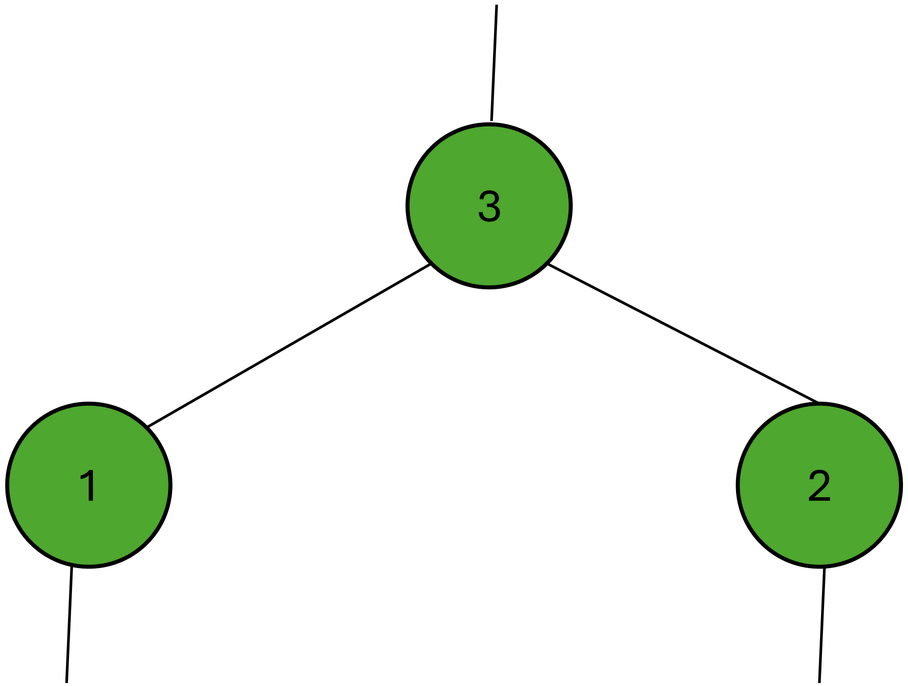
In-order Traversal (Left-Root-Right, Bottom to Top)

Prioritizes visits from bottom left to right visiting parent nodes on the way. Sometimes used for sorting.

Post-order (or natural) Traversal (Left-Right-Root)

Prioritizes traversing subtrees by visiting lowest nodes first, and parent nodes later. Useful for evaluating expression trees (e.g. parsing)

Post order traversal

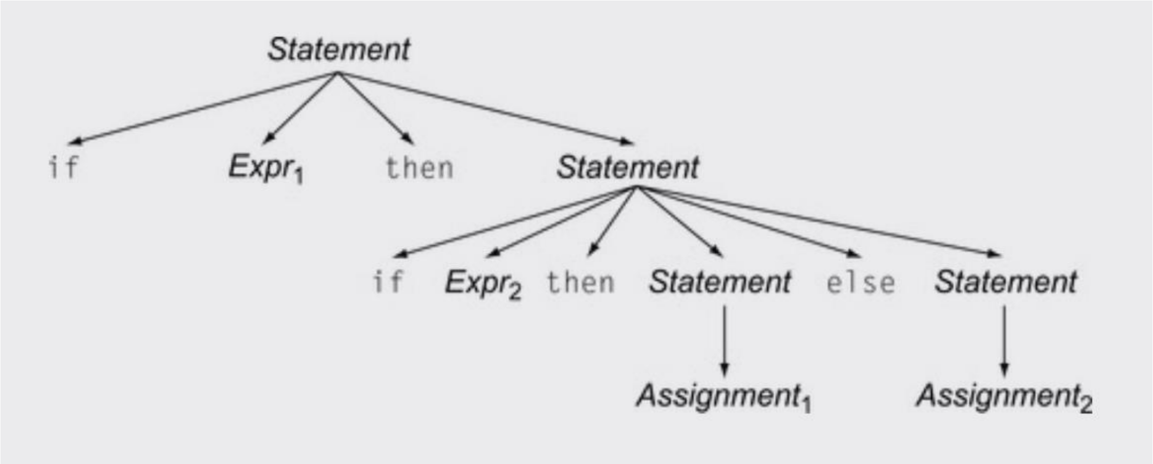


visiting for different types of traversals:

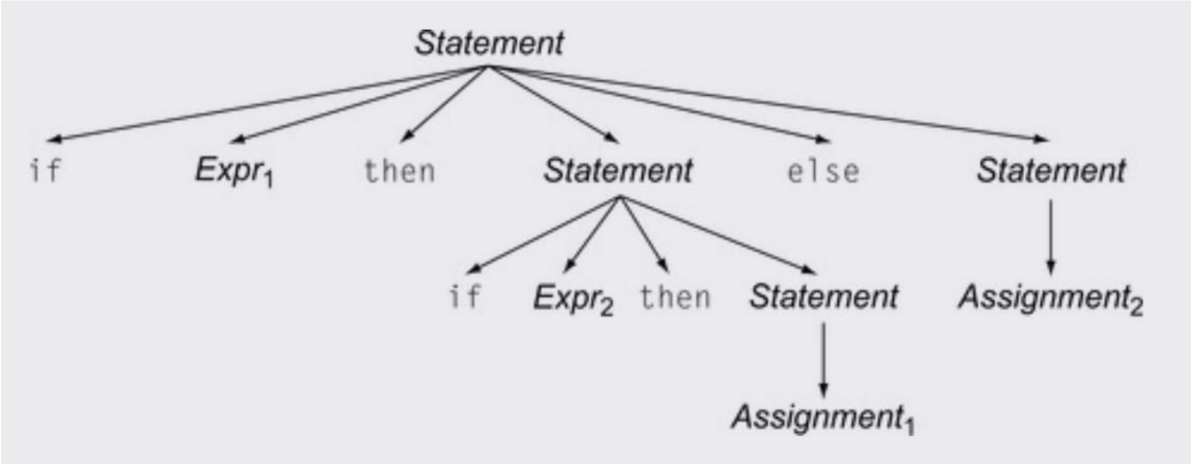
post order

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)
```

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


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


print(check0)
```

How does Python handle this?

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

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

```
input: 5
```

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

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

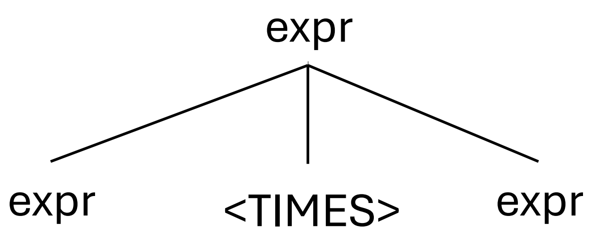
```
input: 5*6
```

expr

Parse trees examples

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

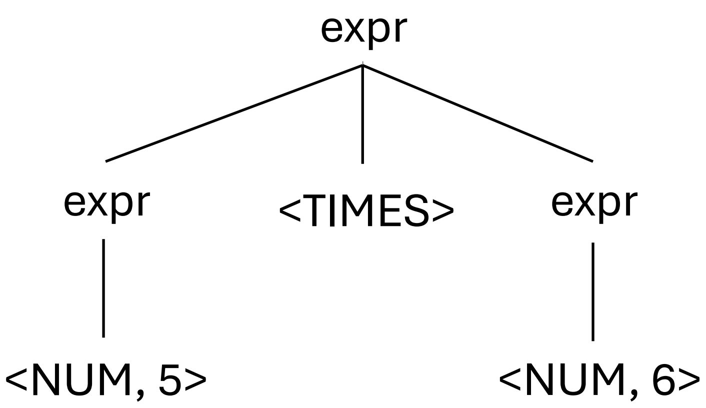
`input: 5*6`



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?

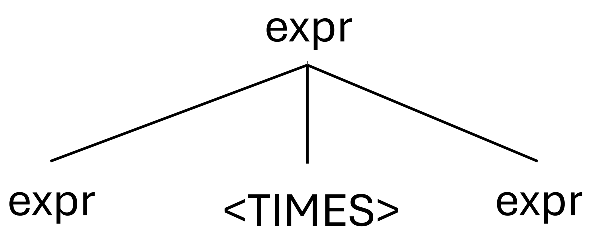
expr

Parse trees examples

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

`input: 5**6`

What happens in an error?

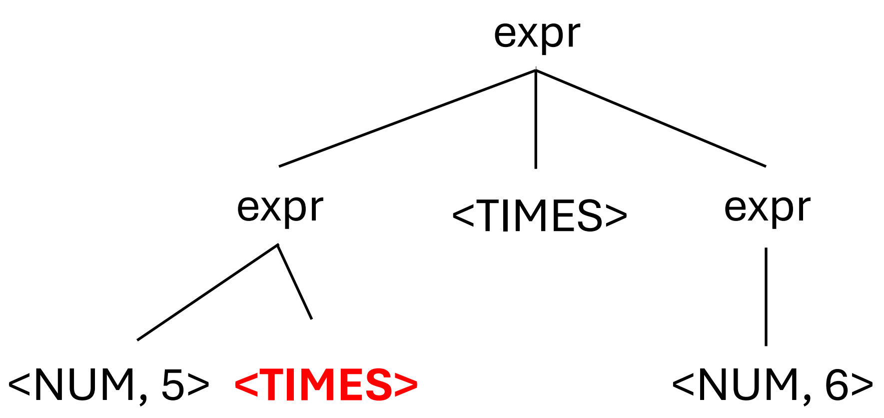


Parse trees examples

```
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

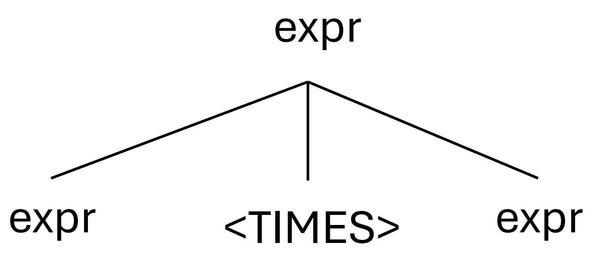
```
expr ::= NUM
      | expr PLUS expr
      | expr TIMES expr
      | LPAREN expr RPAREN
```

expr

Parse trees examples

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

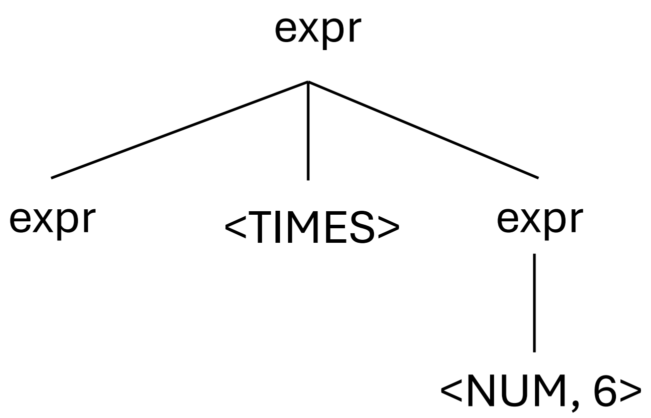
```
expr ::= NUM
      | expr PLUS expr
      | expr TIMES expr
      | LPAREN expr RPAREN
```



Parse trees examples

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

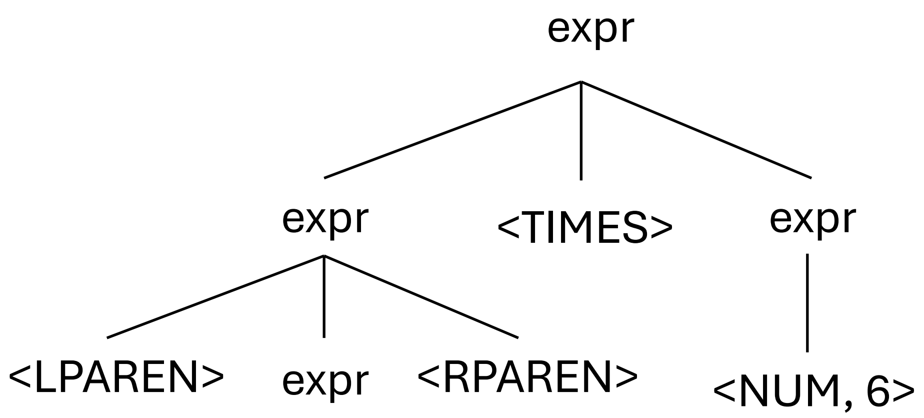
```
input: (1+5)*6
```



Parse trees examples

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

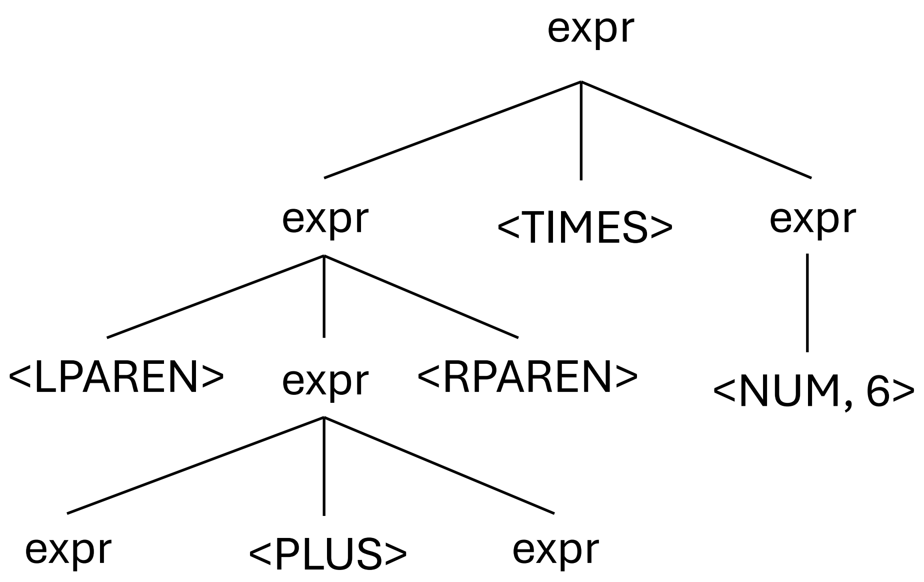
`input: (1+5)*6`



Parse trees examples

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

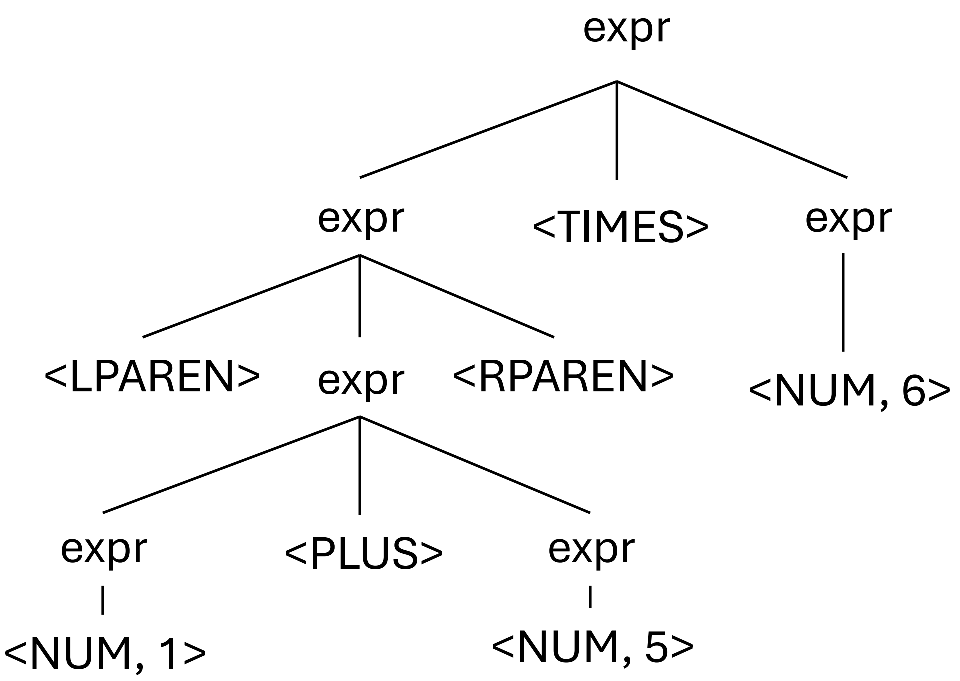
```
expr ::= NUM
      | expr PLUS expr
      | expr TIMES expr
      | LPAREN expr RPAREN
```



Parse trees examples

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

`input: (1+5)*6`

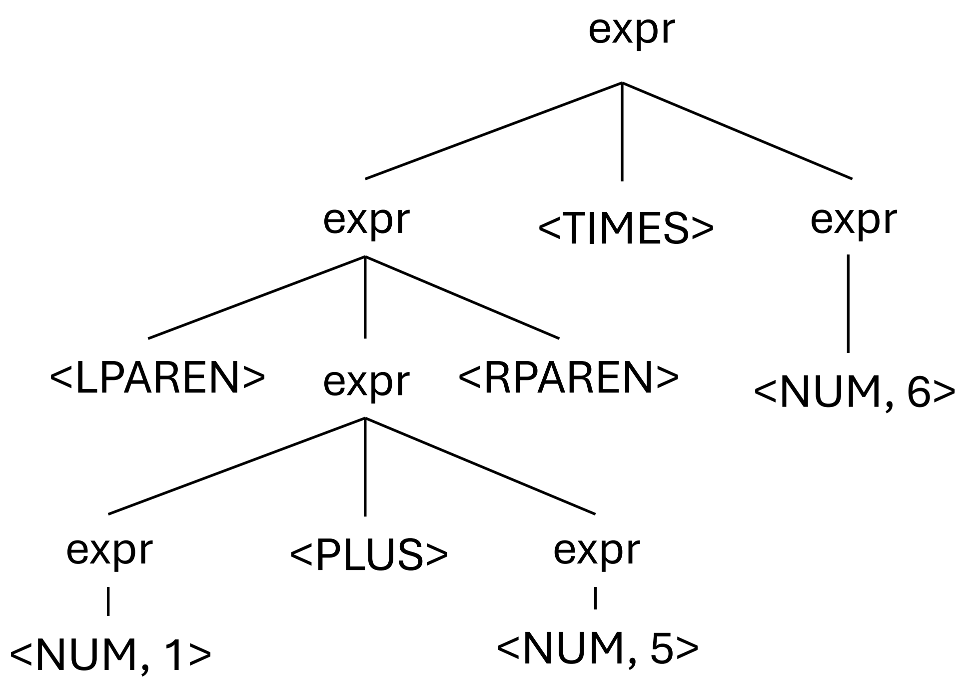


Parse trees examples

Does this parse tree capture the structure we want?

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

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



Parse trees

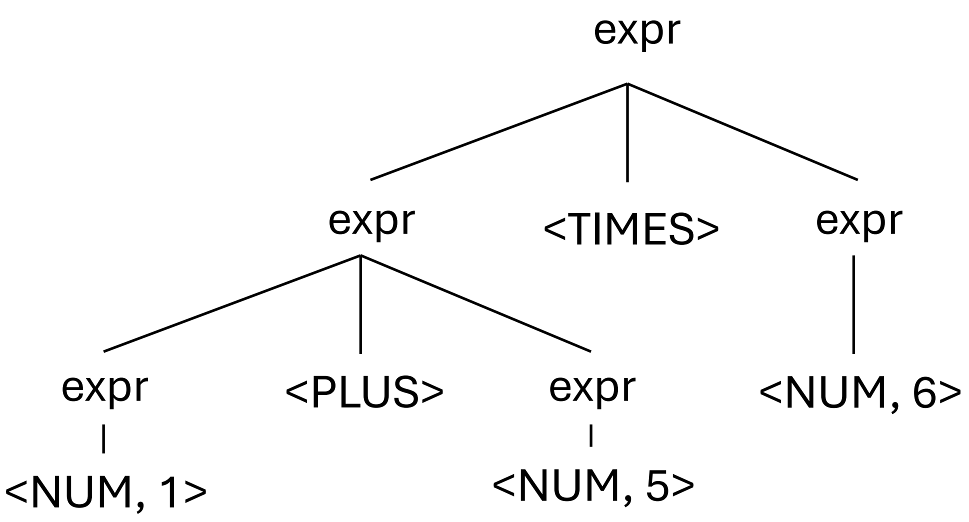
- How about : `1 + 5 * 6`

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

Parse trees

- How about : `1 + 5 * 6`

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

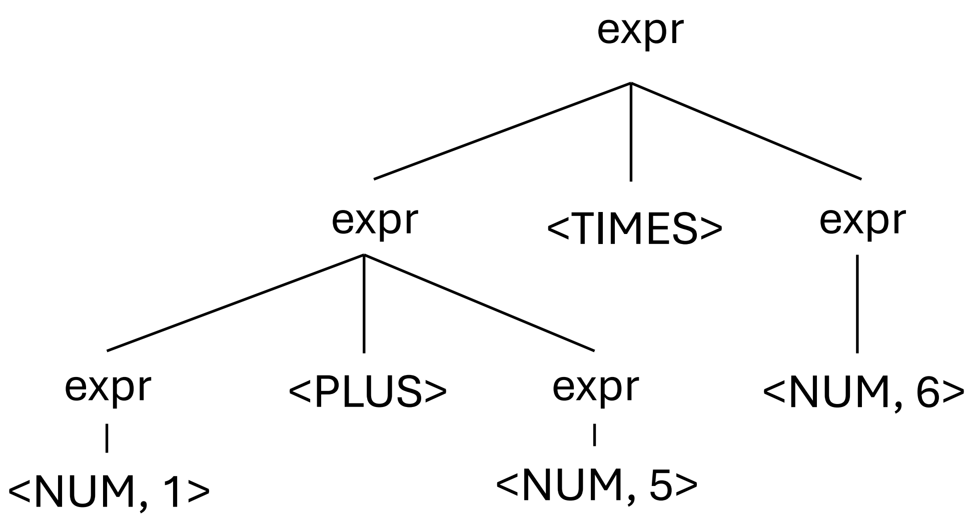


AMBIGUOUS GRAMMARS AND PRECEDENCE

Ambiguous Grammars

- `input: 1 + 5 * 6`

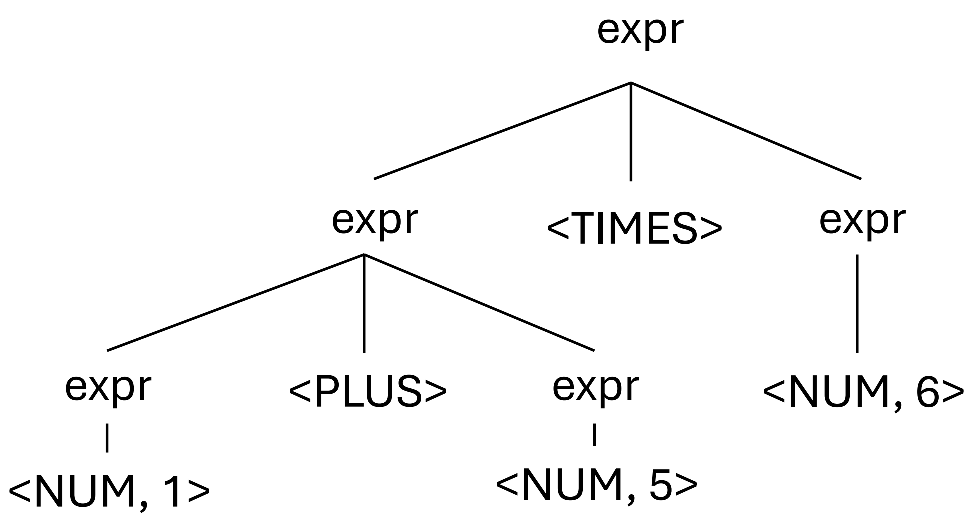
```
expr ::= NUM
       | expr PLUS expr
       | expr TIMES expr
       | LPAREN expr RPAREN
```

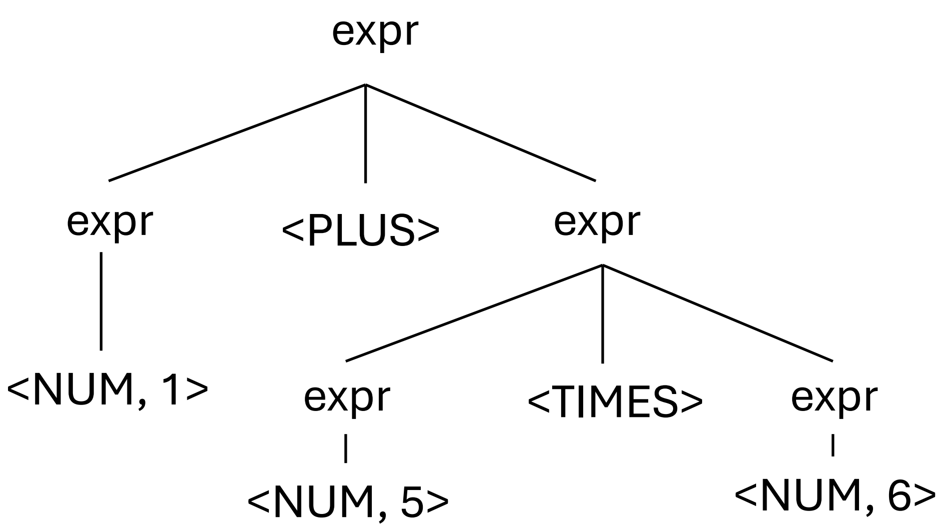


Ambiguous Grammars

- `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 Precedence Ambiguity

- How to avoid ambiguity related to precedence?

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

- Lets try with expressions and the following:

  \+ * ()

Avoiding Precedence Ambiguity

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`

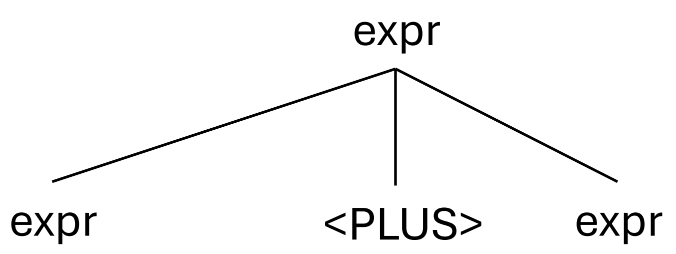
expr

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`

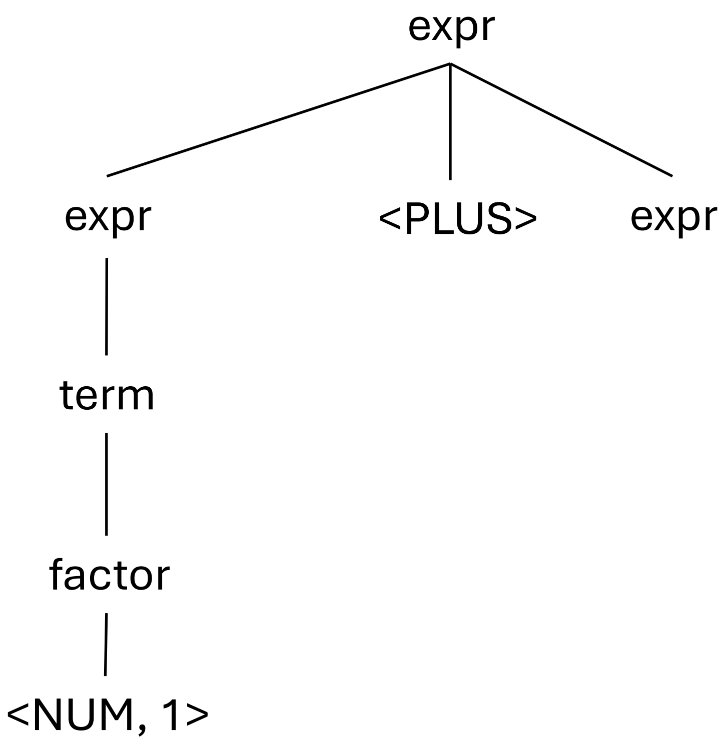
Operator	Name	Productions
+	`expr`	`: expr PLUS expr` `| term`
*	`term`	`: term TIMES term` `| factor`
()	`factor`	`: LPAREN expr RPAREN` `| NUM`



Now lets create a parse tree

input: 1+5*6

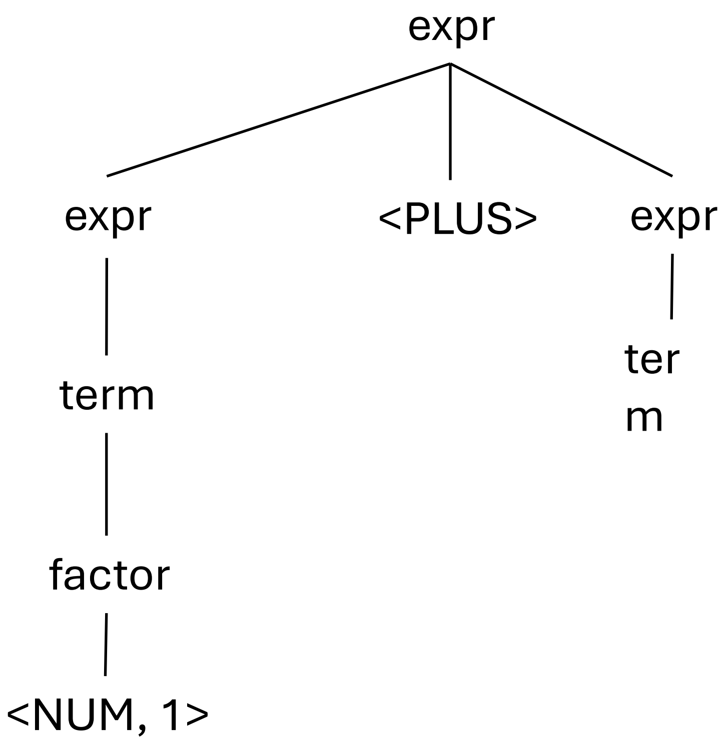
Operator	Name	Productions
+	expr	: expr PLUS expr | term
*	term	: term TIMES term | factor
()	factor	: LPAREN expr RPAREN | NUM



Now lets create a parse tree

`input: 1+5*6`

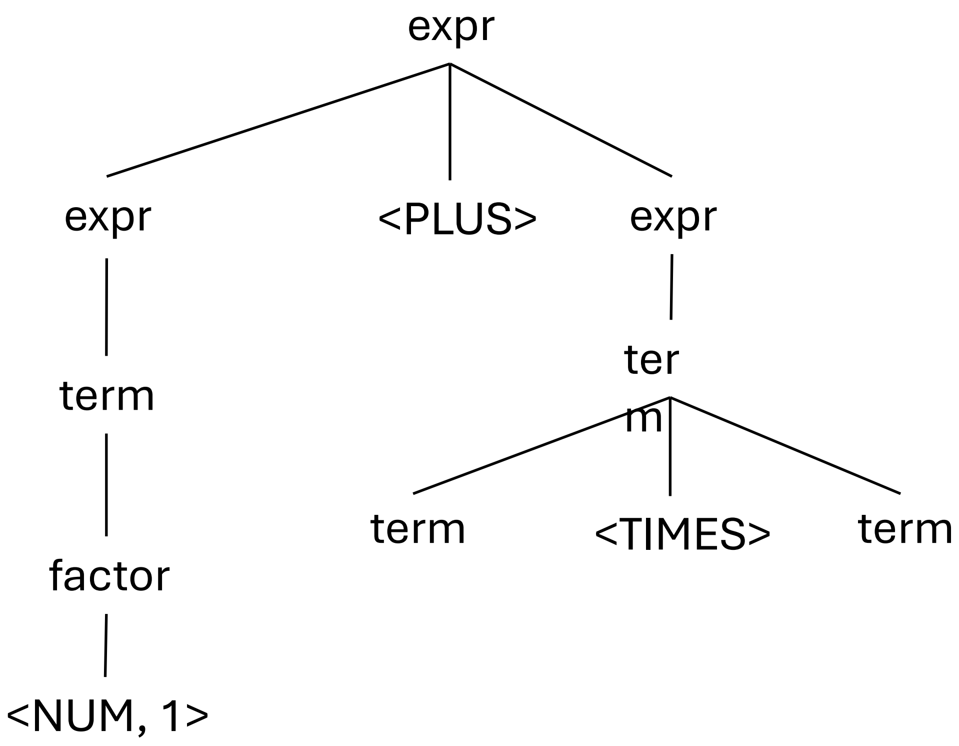
Operator	Name	Productions
+	`expr`	`: expr PLUS expr` `| term`
*	`term`	`: term TIMES term` `| factor`
()	`factor`	`: LPAREN expr RPAREN` `| NUM`



Now lets create a parse tree

input: 1+5*6

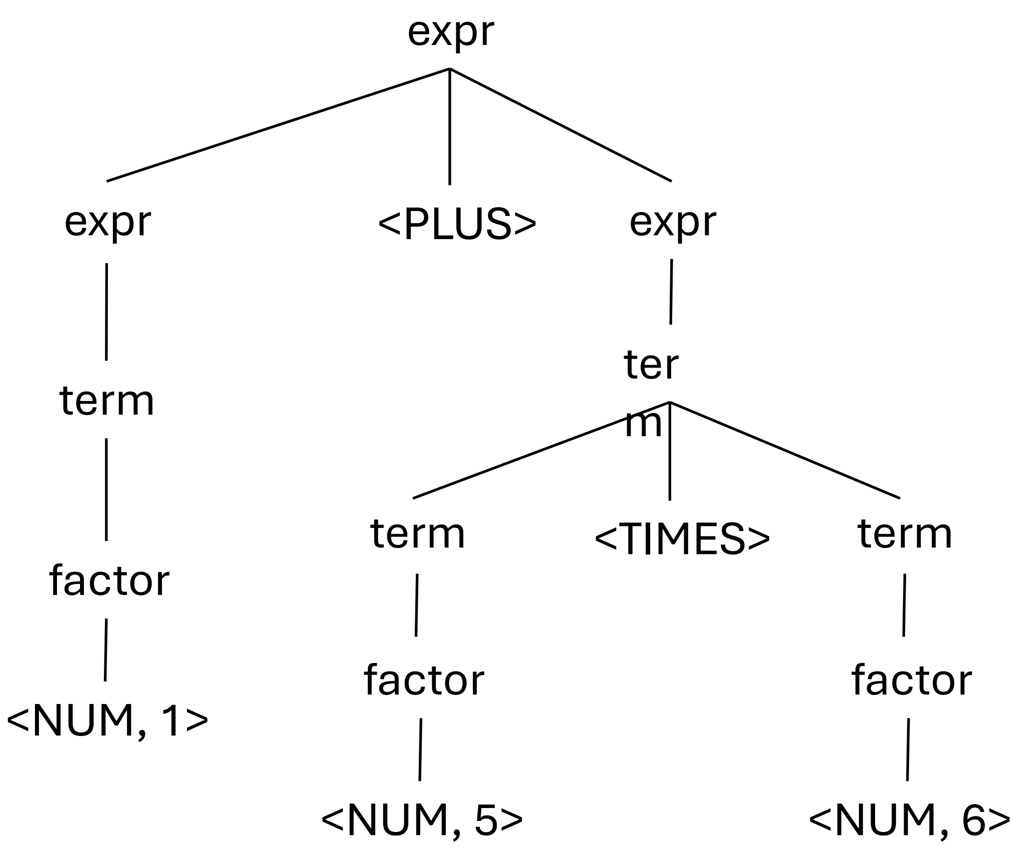
Operator	Name	Productions
+	`expr`	`: expr PLUS expr` `| term`
*	`term`	`: term TIMES term` `| factor`
()	`factor`	`: LPAREN expr RPAREN` `| NUM`



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

<table>
<tr><th>Operator</th><th>Name</th><th>Productions</th></tr>
<tr><td>|</td><td>choice</td><td>: choice PIPE choice
| concat</td></tr>
<tr><td>.</td><td>concat</td><td>: concat DOT concat
| starred</td></tr>
<tr><td>*</td><td>starred</td><td>: starred STAR
| unit</td></tr>
<tr><td>()</td><td>unit</td><td>: LPAREN choice RPAREN
| CHAR</td></tr>
</table>

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*`

<table>
<tr><th>Operator</th><th>Name</th><th>Productions</th></tr>
<tr><td>|</td><td><code>choice</code></td><td><code>: choice PIPE choice</code>
<code>| concat</code></td></tr>
<tr><td>.</td><td><code>concat</code></td><td><code>: concat DOT concat</code>
<code>| starred</code></td></tr>
<tr><td>*</td><td><code>starred</code></td><td><code>: starred STAR</code>
<code>| unit</code></td></tr>
<tr><td>()</td><td><code>unit</code></td><td><code>: LPAREN choice RPAREN</code>
<code>| CHAR</code></td></tr>
</table>

Parsing Regular Expressions

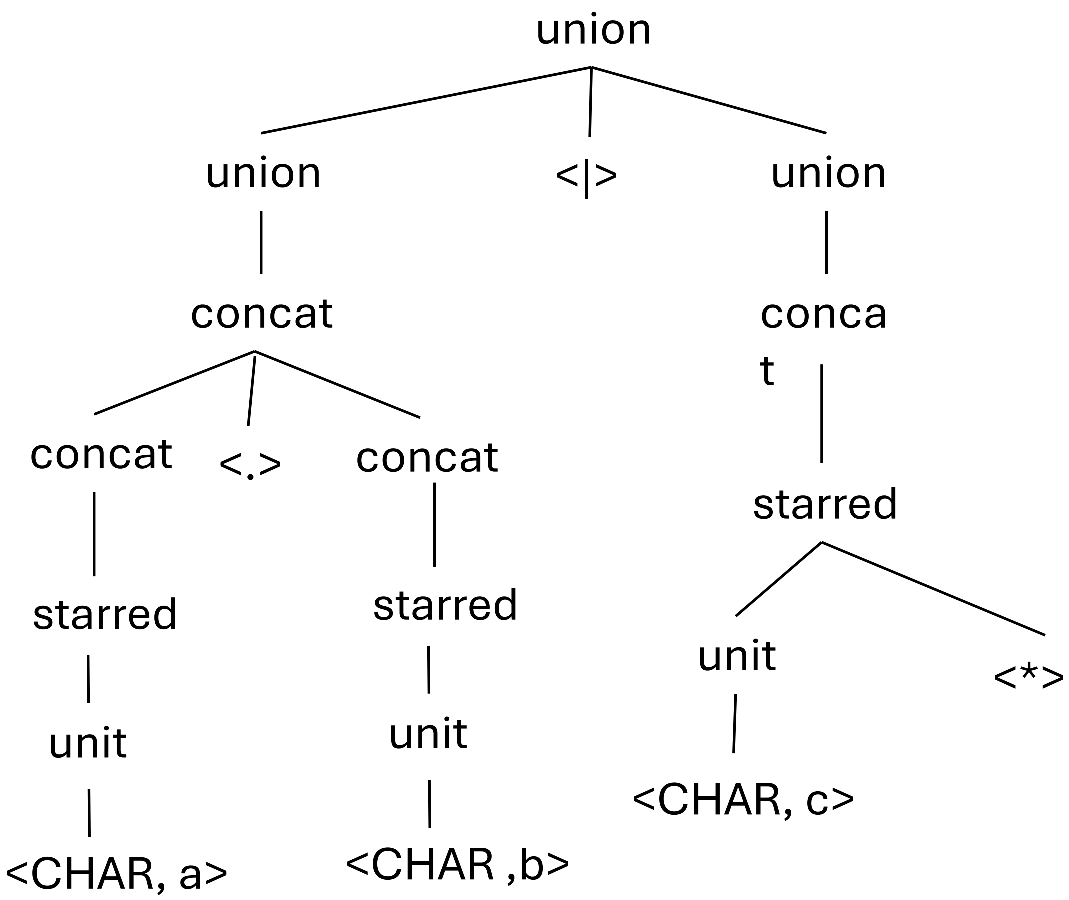
(and considering precedence)

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

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

<table>
<tr><th>Operator</th><th>Name</th><th>Productions</th></tr>
<tr><td>|</td><td>choice</td><td>: choice PIPE choice
| concat</td></tr>
<tr><td>.</td><td>concat</td><td>: concat DOT concat
| starred</td></tr>
<tr><td>*</td><td>starred</td><td>: starred STAR
| unit</td></tr>
<tr><td>()</td><td>unit</td><td>: LPAREN choice RPAREN
| CHAR</td></tr>
</table>

`input: a.b | c*`



How many levels of precedence does C have?

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

Fixing Grammar for Associativity

Let's make some more parse trees

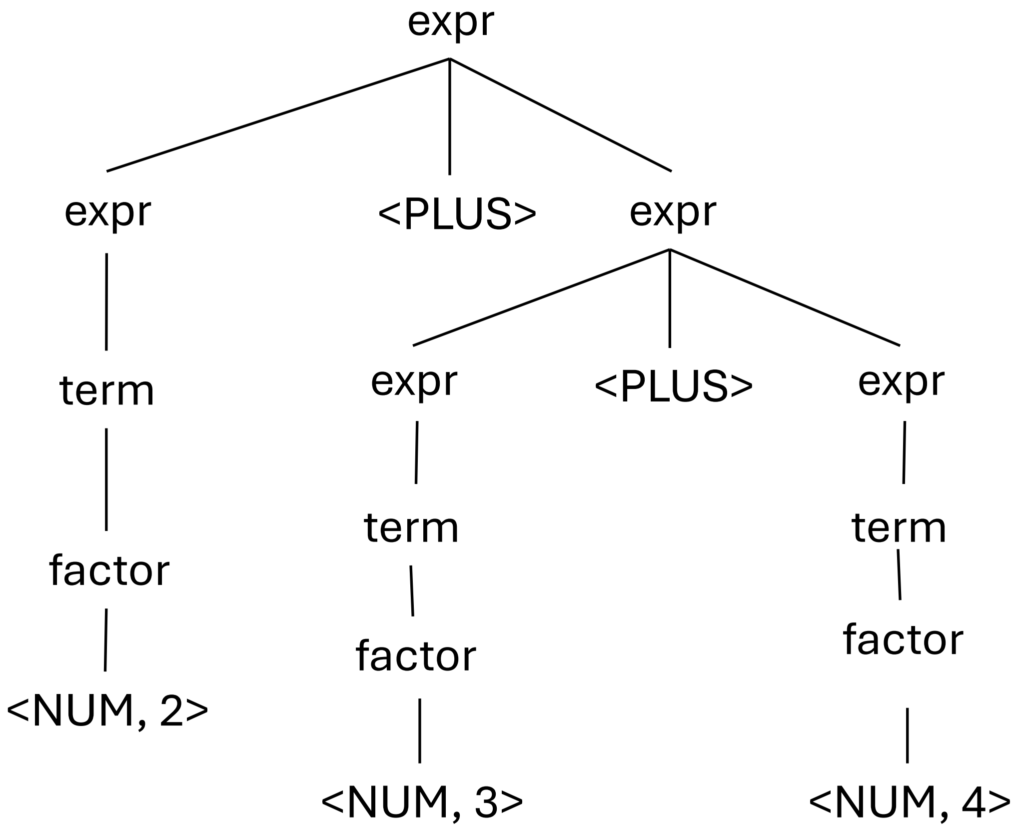
`input: 2+3+4`

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

Let’s make some more parse trees

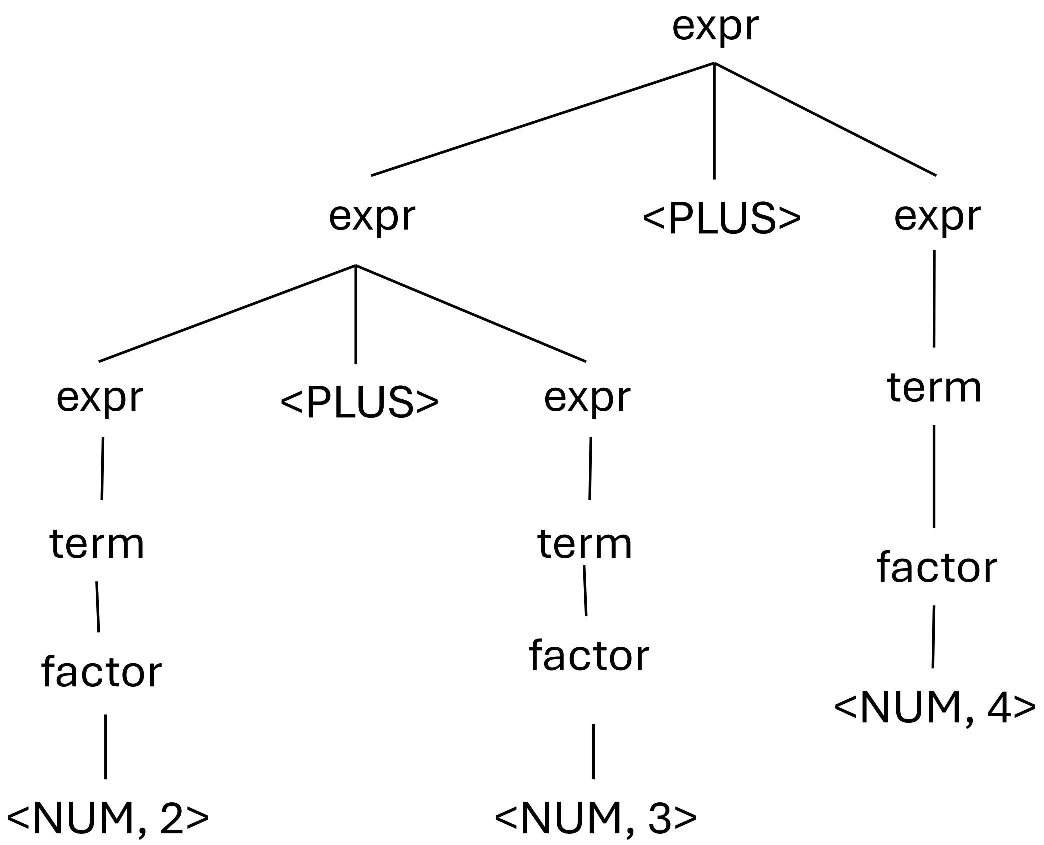
Operator	Name	Productions
+	`expr`	`: expr PLUS expr` `| term`
*	`term`	`: term TIMES term` `| factor`
()	`factor`	`: LP expr RP` `| NUM`

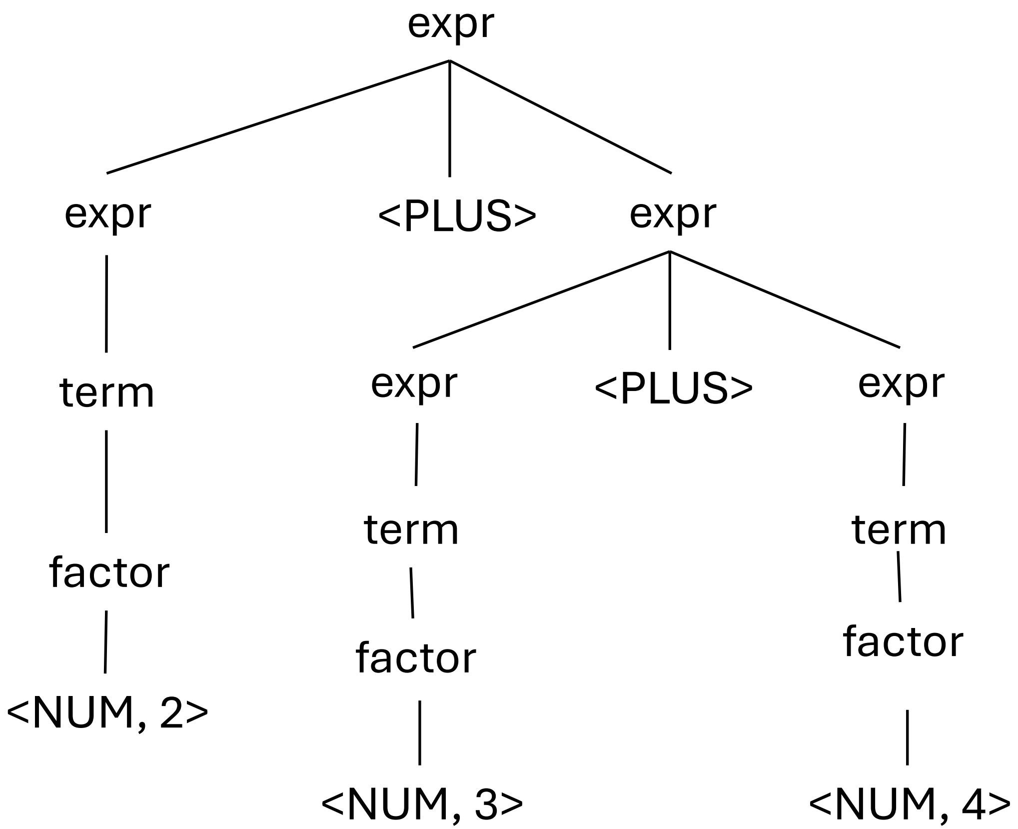
`input: 2+3+4`



This is ambiguous, is it an issue?

`input: 2+3+4`



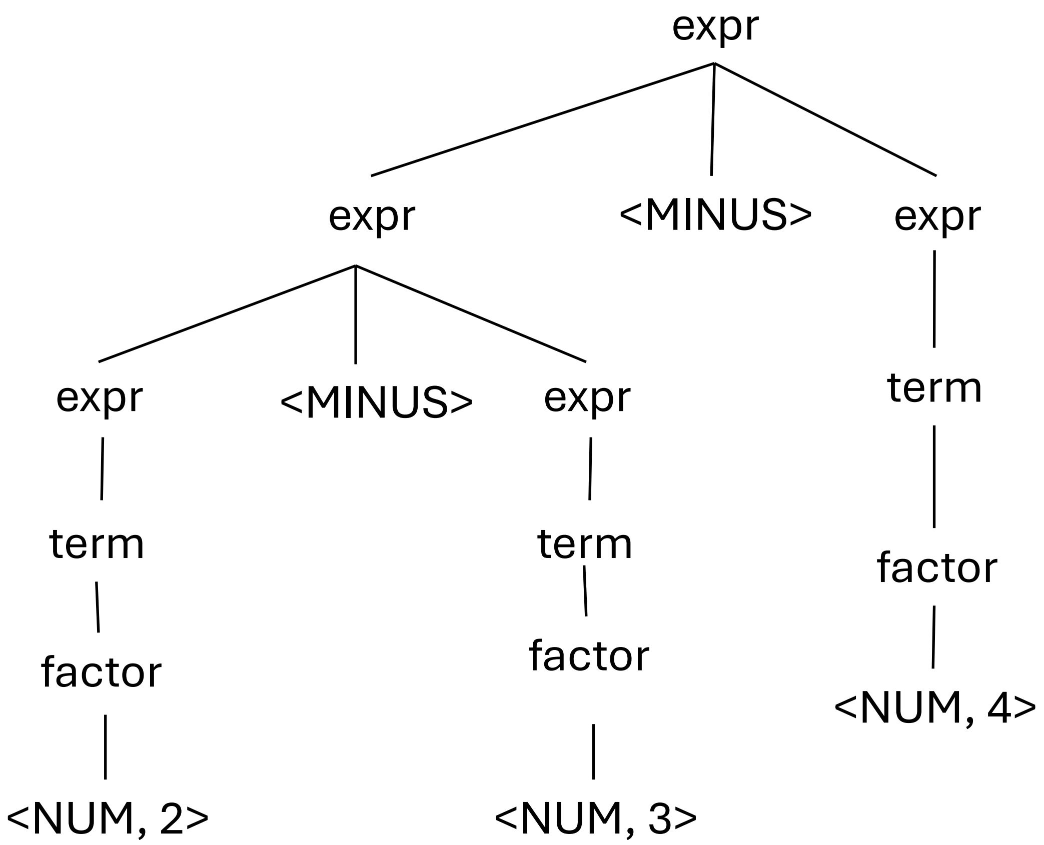


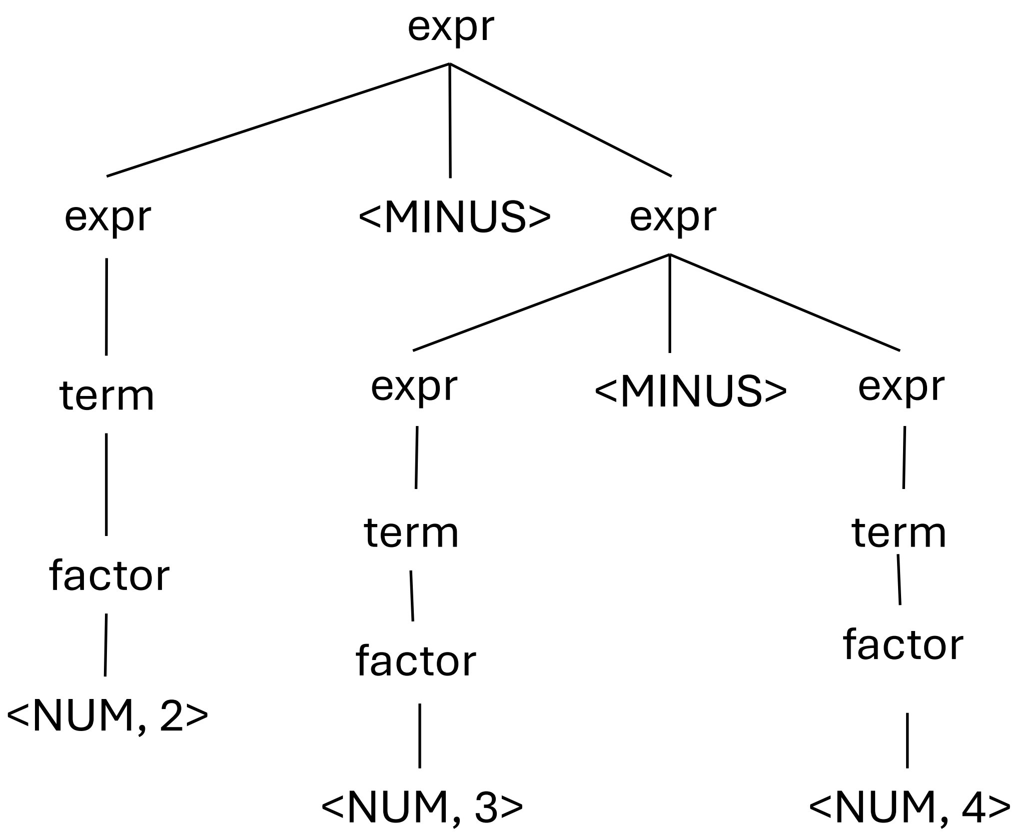
What about for a different operator?

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

What about for a different operator?

`input: 2-3-4`



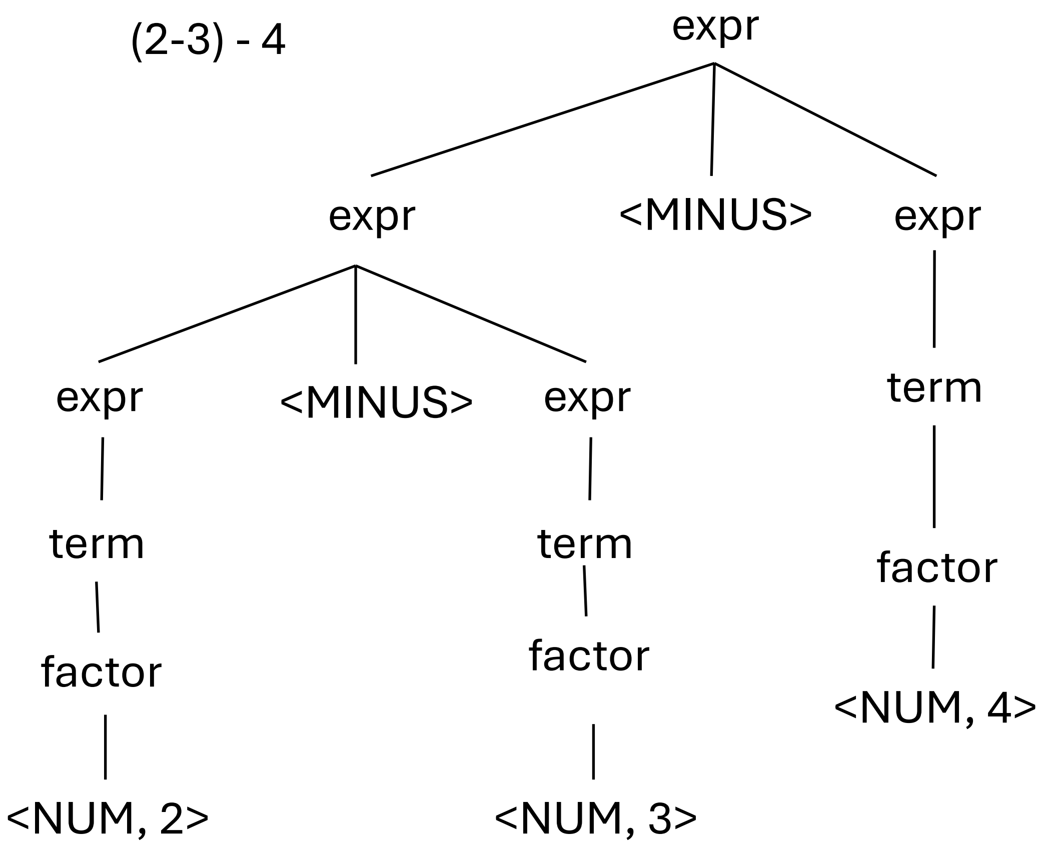


Which one is right?

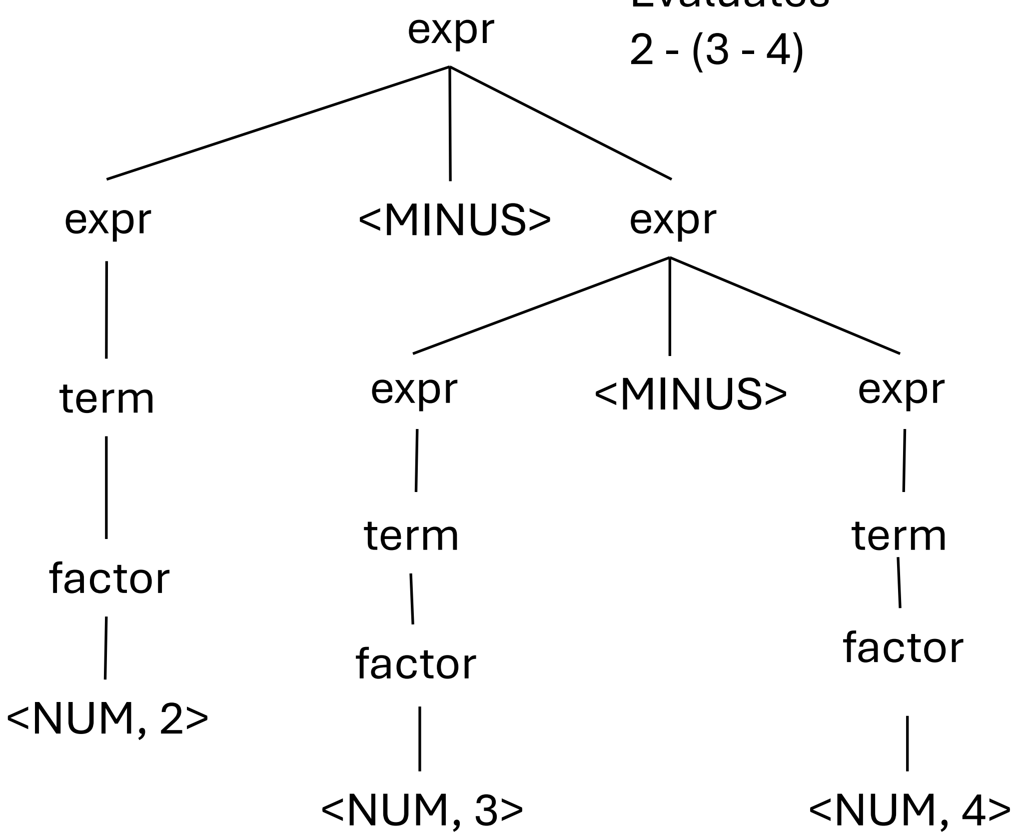
What about for a different operator?

`input: 2-3-4`

Evaluates
(2-3) - 4



Evaluates
2 - (3 - 4)



Which one is right?

Associativity

If an operator is not associative then we define

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

Associativity

If an operator is not associative then we define

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

How to encode associativity?

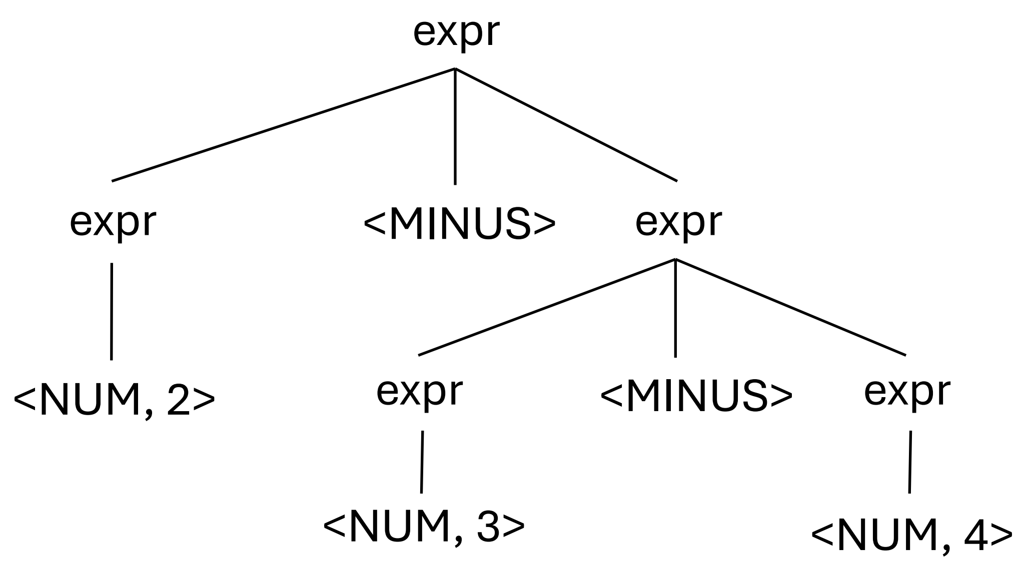
- Like precedence, some tools (e.g. YACC/Bison) allow associativity specification through keywords:
  - “+”: left, “^”: right

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

Associativity for a single operator

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

`input: 2-3-4`

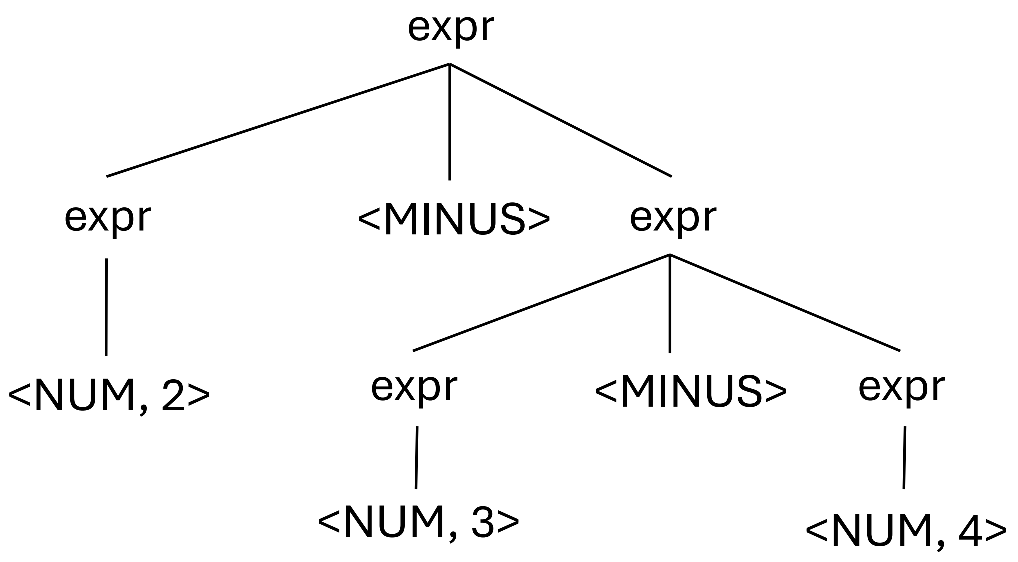


We want to disallow this parse tree

Associativity for a single operator

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

`input: 2-3-4`

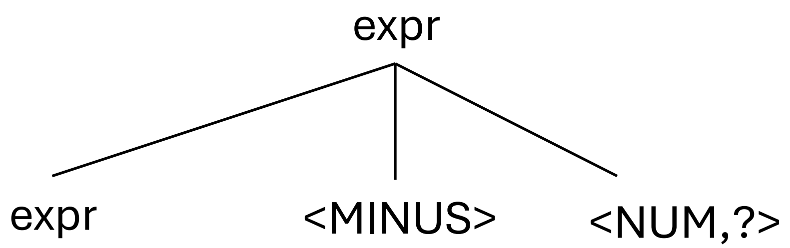


No longer allowed

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>

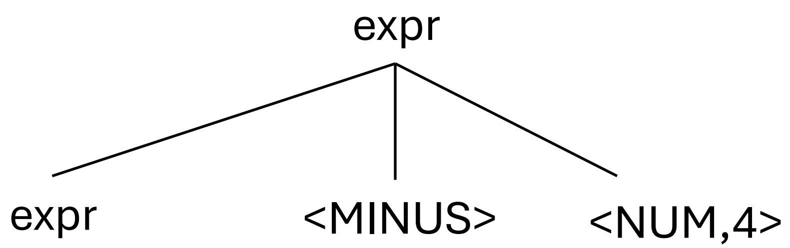
`input: 2-3-4`



Lets start over

Associativity for a single operator

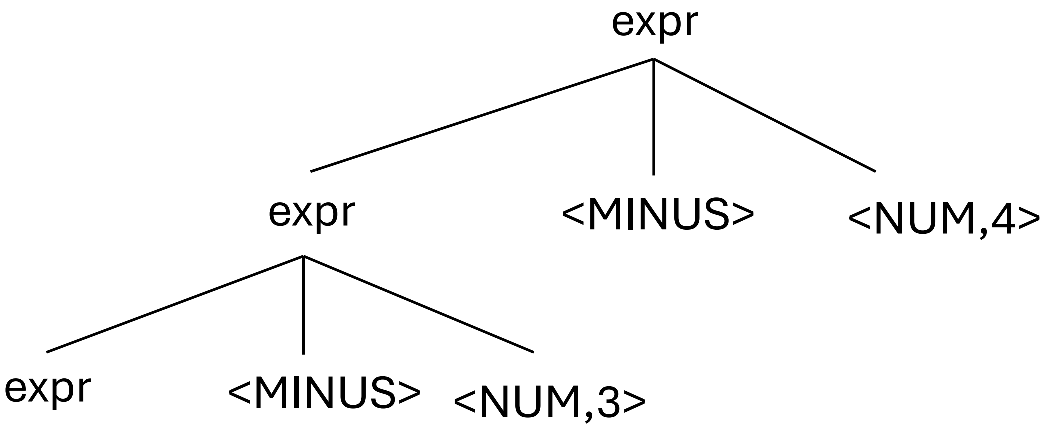
`input: 2-3-4`



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

Associativity for a single operator

`input: 2-3-4`

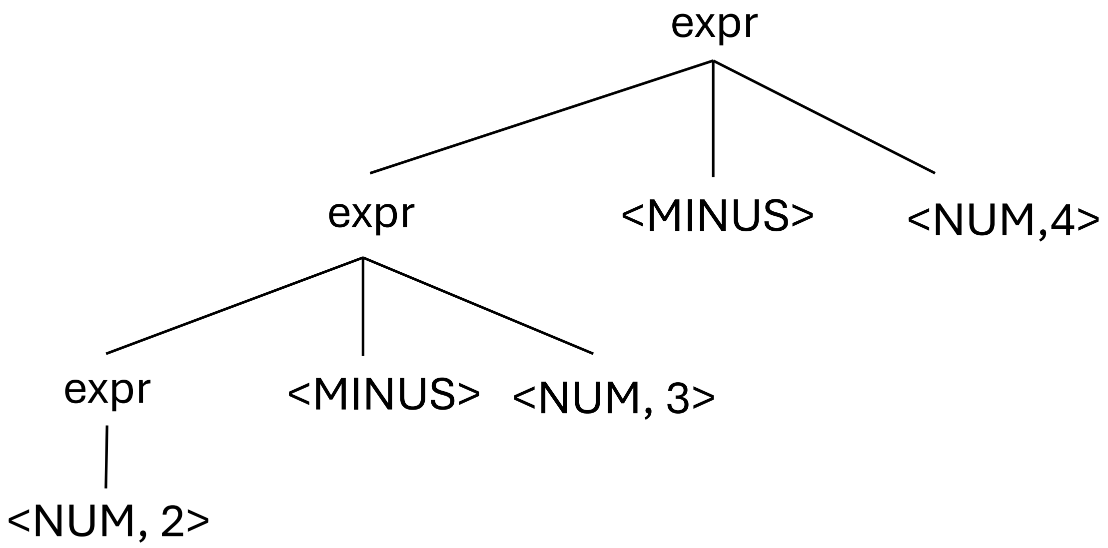


Operator	**Name**	**Productions**
-	`expr`	`: expr MINUS NUM` `| NUM`

Associativity for a single operator

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

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

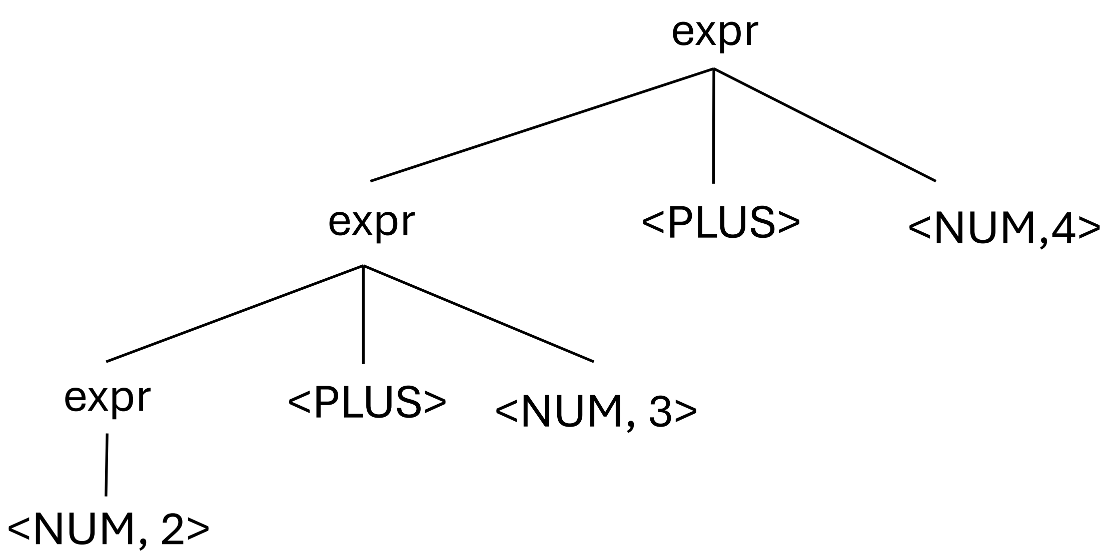


Should you have associativity when its not required?

Benefits?
Drawbacks?

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

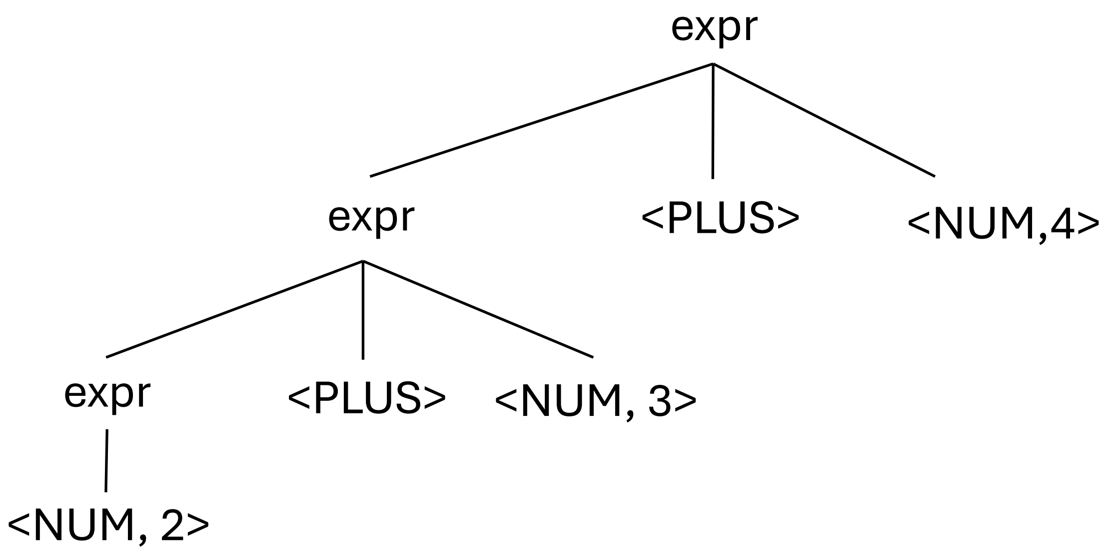
`input: 2+3+4`



Should you have associativity when its not required?

Benefits?
Drawbacks?

`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* `()`

<table>
<tr><th>Operator</th><th>Name</th><th>Productions</th></tr>
<tr><td>+,-</td><td>expr</td><td>: expr PLUS term
| expr MINUS term
| term</td></tr>
<tr><td>*,/</td><td>term</td><td>: term TIMES pow
| term DIV pow
| pow</td></tr>
<tr><td>^</td><td>pow</td><td>: factor CARROT pow
| factor</td></tr>
<tr><td>()</td><td>factor</td><td>: LPAR expr RPAR
| NUM</td></tr>
</table>

```
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

One goal:

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

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

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

Top-down parsing

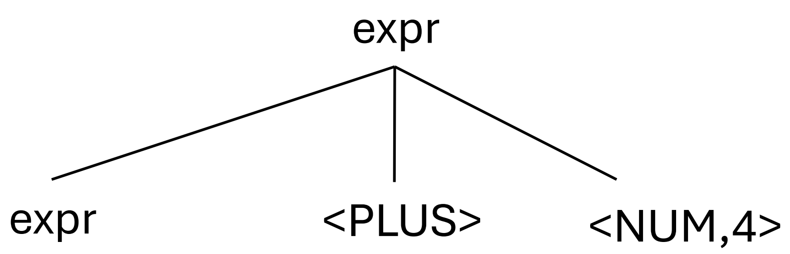
`input: 2+3+4`

expr

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

Top-down parsing

`input: 2+3+4`

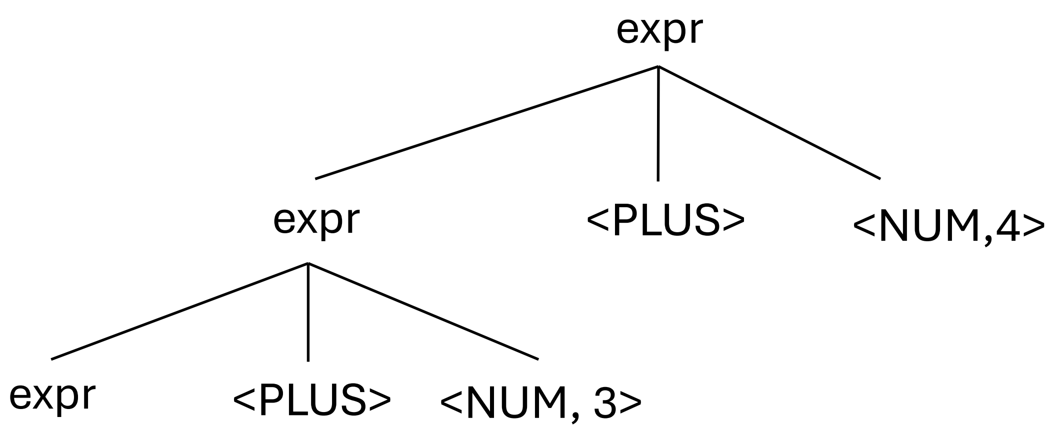


Operator	Name	Productions
+	`expr`	`: expr PLUS NUM` `| NUM`

Top-down parsing

`input: 2+3+4`

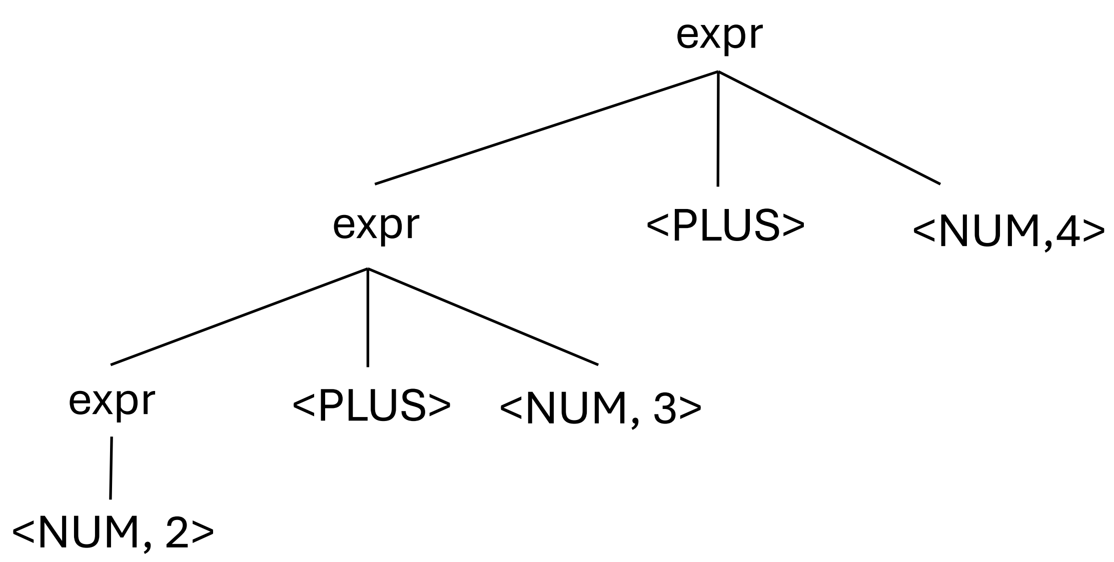
Operator	Name	Productions
+	`expr`	`: expr PLUS NUM` `| NUM`



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
- Many grammars need to be re-written

Bottom-up parsing

`input: 2+3+4`

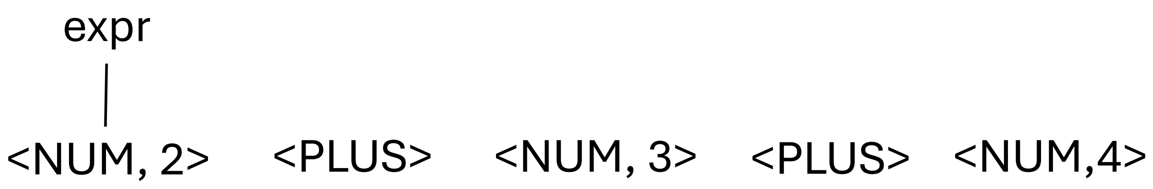
Operator	Name	Productions
+	`expr`	`: expr PLUS NUM` `| NUM`

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

Bottom-up parsing

`input: 2+3+4`

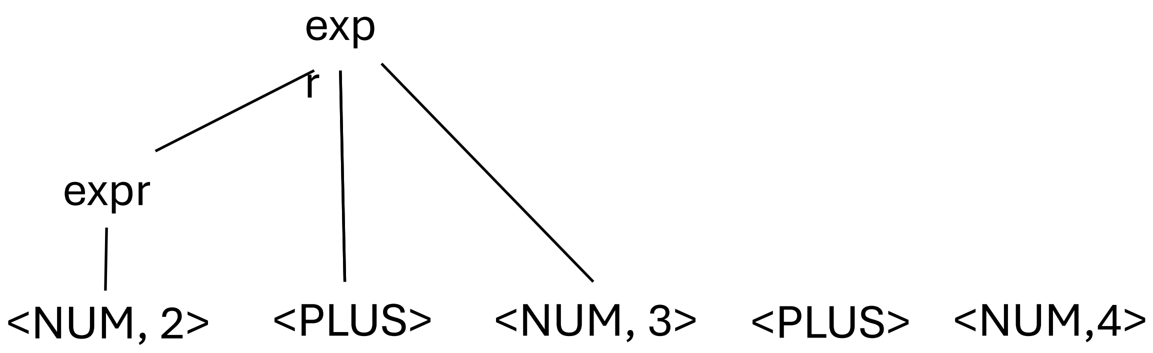
Operator	Name	Productions
+	`expr`	`: expr PLUS NUM` `| NUM`



Bottom-up parsing

`input: 2+3+4`

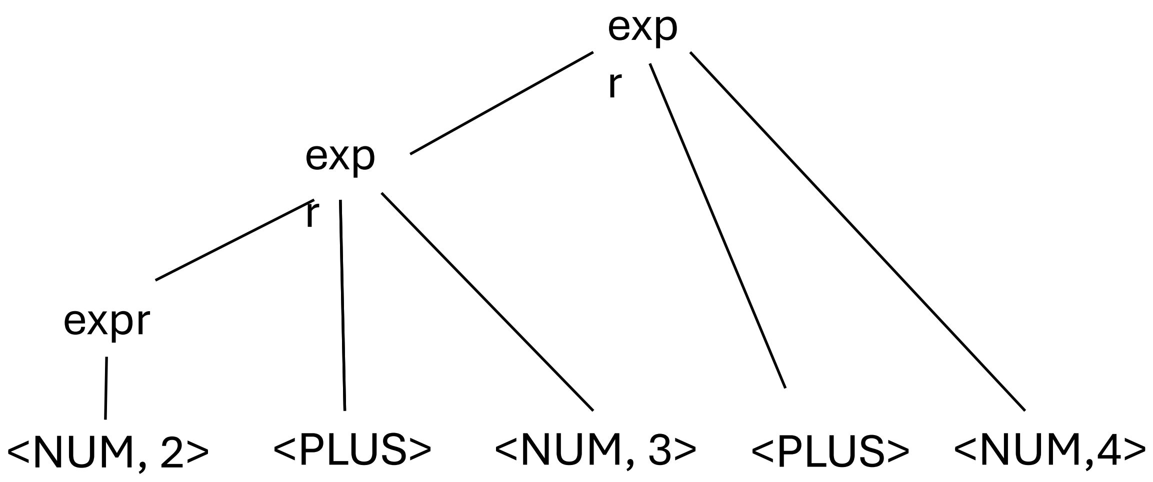
Operator	Name	Productions
+	`expr`	`: expr PLUS NUM` `| NUM`



Bottom-up parsing

`input: 2+3+4`

Operator	Name	Productions
+	expr	: expr PLUS NUM | NUM



Bottom up

Pros:

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

Cons:

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

Let’s start with top-down parsing:

Algorithm: LL(1) Parsing
Left to Right with Look Ahead of 1

```
root = start symbol;    # Expr
focus = root;
push(None);
to_match = s.token();   # Read first token

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

  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

```
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
```

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 And so on…	‘(‘ ID Op Unit ‘)’ Op Unit

```
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

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
2	Expr Op Unit
2	Expr Op Unit Op Unit
2	Expr Op Unit Op Unit Op Unit
2	Expr Op Unit ....

Infinite recursion!

Top down parsing does not handle left recursion

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

direct left recursion

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

indirect left recursion

Top down parsing cannot handle either of these

Top down parsing does not handle left recursion

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

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 = ?
B = ?
```

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

```
1: Expr  ::= ?
2: 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
      |     ""
```

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; # ignore it
    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

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

TOP-DOWN PARSERS

ELIMINATING LEFT-RECURSION

ORACULARITY: MAKING A BACKTRACK FREE LL(1) PARSER USING FIRST, FOLLOW and FIRST+ SETS

```
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 B1 == "": 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
```

Could we make a smarter choice here?

```
1: Expr  ::= ID Expr2
2: Expr2 ::= '+' Expr2
3:         |  ""
```

Can we match: "a"?

Variable	Value
focus	Expr2
to_match	None
s.istring	""
stack	None

Expanded Rule	Sentential Form
start	Expr
1	ID Expr2

The First Set

For each production choice, find the set of tokens that each production can start with

```
1: Expr  ::= Unit Expr2
2: Expr2 ::= Op Unit Expr2
3:         | “”
4: Unit  ::= ‘(‘ Expr ‘)’
5:         |    ID
6: Op    ::= ‘+’
7:         |   ‘*’
```

```
First sets:
1: {}
2: {}
3: {}
4: {}
5: {}
6: {}
7: {}
```

The First Set

For each production choice, find the set of tokens that each production can start with

```
1: Expr  ::= Unit Expr2
2: Expr2 ::= Op Unit Expr2
3:         | “”
4: Unit  ::= ‘(‘ Expr ‘)’
5:         |    ID
6: Op    ::= ‘+’
7:         |   ‘*’
```

```
First sets:
1: {‘(‘, ID}
2: {‘+’, ‘*’}
3: {“”}
4: {‘(‘}
5: {ID}
6: {‘+’}
7: {‘*’}
```

We can use first sets to decide which rule to pick!

```
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:          |   '*'
```

```
First sets:
1: {'(', ID}
2: {'+', '*'}
3: {""}
4: {'('}
5: {ID}
6: {'+'}
7: {'*'}
```

We simply use to_match and compare it to the first sets for each choice

For example, Op and Unit

The Follow Set

Rules with “” in their First set need special attention

```
1: Expr  ::= Unit Expr2
2: Expr2 ::= Op Unit Expr2
3:         | “”
4: Unit  ::= ‘(‘ Expr ‘)’
5:         |    ID
6: Op    ::= ‘+’
7:         |   ‘*’
```

```
First sets:
1: {‘(‘, ID}
2: {‘+’, ‘*’}
3: {“”}
4: {‘(‘}
5: {ID}
6: {‘+’}
7: {‘*’}
```

```
Follow sets:
1: NA
2: NA
3: {}
4: NA
5: NA
6: NA
7: NA
```

We need to find the tokens that any string that follows the production can start with.

The Follow Set

Rules with “” in their First set need special attention

```
1: Expr  ::= Unit Expr2
2: Expr2 ::= Op Unit Expr2
3:        | “”
4: Unit  ::= ‘(‘ Expr ‘)’
5:        |    ID
6: Op    ::= ‘+’
7:        |   ‘*’
```

```
First sets:
1: {‘(‘, ID}
2: {‘+’, ‘*’}
3: {“”}
4: {‘(‘}
5: {ID}
6: {‘+’}
7: {‘*’}
```

```
Follow sets:
1: NA
2: NA
3: {}
4: NA
5: NA
6: NA
7: NA
```

We need to find the tokens that any string that follows the production can start with.

The First+ Set

The First+ set is conditional combination of First and Follow sets

$$\mathrm{FIRST}^{+}(A \rightarrow \beta) = \begin{cases} \mathrm{FIRST}(\beta) & \text{if } \epsilon \notin \mathrm{FIRST}(\beta) \\ \mathrm{FIRST}(\beta) \cup \mathrm{FOLLOW}(A) & \textit{otherwise} \end{cases}$$

Calculating Follow(A) Set

```
1: Expr  ::= Unit Expr2
2: Expr2 ::= Op Unit Expr2
3:         | “”
4: Unit  ::= ‘(‘ Expr ‘)’
5:         |    ID
6: Op    ::= ‘+’
7:         |    ‘*’
```

Rules with “” in their First set need special attention

We need to find the tokens that any string that follows the production can start with.

```
First sets:     Follow sets:
1: {‘(‘, ID}    1: NA
2: {‘+’, ‘*’}   2: NA
3: {“”}         3: {}
4: {‘(‘}        4: NA
5: {ID}         5: NA
6: {‘+’}        6: NA
7: {‘*’}        7: NA
```

Follow Set

	ID	+	*	)	(	e	$
Expr				v			v
Expr2				v			v
Unit		v	v	v			v
Op	v				v		

To compute FOLLOW(A) for all nonterminals A, apply the following rules until nothing can be added to any FOLLOW set:

1. Place \$ in FOLLOW(S), where S is the start symbol and \$ is the input right endmarker.
2. If there is a production $A \Rightarrow \alpha B \beta$, then everything in FIRST(β), except for ε, is placed in FOLLOW(B).
3. If there is a production $A \Rightarrow \alpha B$, or a production $A \Rightarrow \alpha B \beta$ where FIRST(β) contains ε (i.e., $\beta \Rightarrow \varepsilon$), then everything in FOLLOW(A) is in FOLLOW(B).

The Follow Set

Rules with "" in their First set need special attention

```
1: Expr  ::= Unit Expr2
2: Expr2 ::= Op Unit Expr2
3:         | “”
4: Unit  ::= ‘(‘ Expr ‘)’
5:         |   ID
6: Op    ::= ‘+’
7:         |  ‘*’
```

```
First sets:          Follow sets:
1: {‘(‘, ID}         1: NA
2: {‘+’, ‘*’}        2: NA
3: {“”}              3: {None, ’)’}
4: {‘(‘}             4: NA
5: {ID}              5: NA
6: {‘+’}             6: NA
7: {‘*’}             7: NA
```

Follow Set

	ID	+	*	)	(	e	$
Expr				v			v
Expr2				v			v
Unit		v	v	v			v
Op	v				v		

We need to find the tokens that any string that follows the production can start with. $ and None in Follow/First+ sets are the same. The None/$ terminal refer to the end of a valid language sentence.

The First+ Set

The First+ set is the combination of First and Follow sets

```
                                 First sets:    Follow sets:     First+ sets:
1: Expr  ::= Unit Expr2          1: {‘(‘, ID}   1: NA            1: {‘(‘, ID}
2: Expr2 ::= Op Unit Expr2       2: {‘+’, ‘*’}  2: NA            2: {‘+’, ‘*’}
3:        | “”                   3: {“”}        3: {None, ’)’}   3: {None, ’)’}
4: Unit  ::= ‘(‘ Expr ‘)’        4: {‘(‘}       4: NA            4: {‘(‘}
5:        |    ID                5: {ID}        5: NA            5: {ID}
6: Op    ::= ‘+’                 6: {‘+’}       6: NA            6: {‘+’}
7:        |    ‘*’               7: {‘*’}       7: NA            7: {‘*’}
```

Do we need backtracking?

The First+ set is the combination of First and Follow sets

```
1: Expr  ::= Unit Expr2
2: Expr2 ::= Op Unit Expr2
3:         | “”
4: Unit  ::= ‘(‘ Expr ‘)’
5:         |    ID
6: Op    ::= ‘+’
7:         |    ‘*’
```

```
First+ sets:
1: {‘(‘, ID}
2: {‘+’, ‘*’}
3: {None, ’)’}
4: {‘(‘}
5: {ID}
6: {‘+’}
7: {‘*’}
```

For each non-terminal: if every production has a disjoint First+ set then we do not need any backtracking!

Do we need backtracking?

The First+ set is the combination of First and Follow sets

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

```
First+ sets:
1: {'(', ID}
2: {'+', '*'}
3: {None, ')'}
4: {'('}
5: {ID}
6: {'+'}
7: {'*'}
```

For each non-terminal: if every production has a disjoint First+ set then we do not need any backtracking!

Do we need backtracking?

The First+ set is the combination of First and Follow sets

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

```
First+ sets:
1:  {'(', ID}
2:  {'+', '*'}
3:  {None, ')'}
4:  {'('}
5:  {ID}
6:  {'+'}
7:  {'*'}
```

These grammars are called LL(1)

- L - scanning the input left to right
- L - left derivation
- 1 - how many look ahead symbols

They are also called predictive grammars

Many programming languages are LL(1)

For each non-terminal: if every production has a disjoint First+ set then we do not need any backtracking!

Recursive Descent allows for easy predictive lookahead of two or more.

LEFT-FACTORING A GRAMMAR

Sometimes the grammar needs to be refactored

```
Factor ::= ID
       |   ID ‘[‘ Args ‘]’
       |   ID ‘(‘ Args ‘)’
...
```

Sometimes the grammar needs to be refactored

```
1: Factor ::= ID
2:         |   ID '[' Args ']'
3:         |   ID '(' Args ')'
...
```

```
First
1: {}
2: {}
3: {}
...
```

Left Factoring

Left Factoring: the process of extracting and isolating common prefixes in a set of productions

$$A ::= a\ B_1 \mid a\ B_2 \mid a\ B_3 \mid c_1 \mid c_2 \mid c_3$$

So, define new Non-Terminal for Common Suffixes

Left Factor this way:

$$A ::= a\ B$$

$$B ::= B_1 \mid B_2 \mid B_3$$

$$C ::= c_1 \mid c_2 \mid c_3$$

Often permits a disjoint First Set for every production (RHS) avoiding backtracking.

11	*Factor*	→	name
12		|	name [*ArgList*]
13		|	name (*ArgList*)

11	*Factor*	→	name *Arguments*
12	*Arguments*	→	[*ArgList*]
13		|	(*ArgList*)
14		|	ϵ

Sometimes the grammar needs to be refactored

```
1: Factor ::= ID
2:         |   ID '[' Args ']'
3:         |   ID '(' Args ')'
...
```

```
First
1: {ID}
2: {ID}
3: {ID}
...
```

We cannot select the next rule based on a single look ahead token!

Sometimes the grammar needs to be refactored

```
1: Factor ::= ID
2:        |   ID '[' Args ']'
3:        |   ID '(' Args ')'
...
```

```
First
1: {ID}
2: {ID}
3: {ID}
...
```

We can refactor

```
1: Factor      ::= ID Option_args
2: Option_args ::= '[' Args ']'
3:             |   '(' Args ')'
4:             |   ""
```

```
First
1: {}
2: {}
3: {}
4: {}
```

Sometimes the grammar needs to be refactored

```
1: Factor ::= ID
2:          |   ID '[' Args ']'
3:          |   ID '(' Args ')'
...
```

```
First
1: {ID}
2: {ID}
3: {ID}
...
```

We can refactor

```
1: Factor       ::= ID Option_args
2: Option_args  ::= '[' Args ']'
3:              |   '(' Args ')'
4:              |   ""
```

```
First
1: {ID}
2: {'['}
3: {'('}
4: {""}
```

// We will need to compute the follow set

Sometimes the grammar needs to be refactored

```
1: Factor ::= ID
2:         |   ID '[' Args ']'
3:         |   ID '(' Args ')'
...
```

```
First
1: {ID}
2: {ID}
3: {ID}
...
```

It is not always possible to rewrite grammars into a predictive form, but many programming languages can be.

We can refactor

```
1: Factor       ::= ID Option_args
2: Option_args ::= '[' Args ']'
3:              |   '(' Args ')'
4:              |   ""
```

```
First
1: {ID}
2: {'['}
3: {'('}
4: {""}
```

// We will need to compute the follow set

A Pattern for Left Factoring

A ::= $a\ B_1 \mid a\ B_2 \mid a\ B_3 \mid c_1 \mid c_2 \mid c_3$

Left Factor this way:

A ::= $a\ B$

B ::= $B_1 \mid B_2 \mid B_3$

C ::= $c_1 \mid c_2 \mid c_3$

S : Expr
1. Expr : Expr + Term
2. | Expr - Term
3. | Term
4. Term : Term * Factor
5. | term / Factor
6. | Factor
7. Factor : (Expr)
8. | num
9. | name

We now have a full top-down LL(1) parsing algorithm!

```
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
```

```
First+ sets:
1: {'(', ID}
2: {'+', '*'}
3: {None, ')'}
4: {'('}
5: {ID}
6: {'+'}
7: {'*'}
```

First+ sets for each production rule

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

input grammar, refactored to remove left recursion

To pick the next rule, compare `to_match` with the possible `first+` sets. Pick the rule whose `first+` set contains `to_match`.

If there is no such rule then it is a parsing error.

Moving on to a simpler implementation:

Recursive Descent Parser

Let’s look at the grammar

```
1: Expr  ::= Unit Expr2
2: Expr2 ::= Op Unit Expr2
3:        | “”
4: Unit  ::= ‘(‘ Expr ‘)’
5:        |    ID
6: Op    ::= ‘+’
7:        |   ‘*’
```

Let’s look at the grammar

```
1: Expr   ::= Unit Expr2
2: Expr2  ::= Op Unit Expr2
3:          | “”
4: Unit   ::= ‘(‘ Expr ‘)’
5:          |   ID
6: Op     ::= ‘+’
7:          |  ‘*’
```

How do we parse an Expr?

Let’s look at the grammar

```
1: Expr  ::= Unit Expr2
2: Expr2 ::= Op Unit Expr2
3:         | “”
4: Unit  ::= ‘(‘ Expr ‘)’
5:         |   ID
6: Op    ::= ‘+’
7:         |   ‘*’
```

How do we parse an Expr?
We parse a Unit followed by an Expr2

Let’s look at the grammar

```
1: Expr   ::= Unit Expr2
2: Expr2  ::= Op Unit Expr2
3:          | “”
4: Unit   ::= ‘(‘ Expr ‘)’
5:          |    ID
6: Op     ::= ‘+’
7:          |   ‘*’
```

How do we parse an Expr?
We parse a Unit followed by an Expr2

We can just write exactly that!

```
def parse_Expr(self):
    self.parse_Unit();
    self.parse_Expr2();
    return
```

Let’s look at the grammar

```
1: Expr   ::= Unit Expr2
2: Expr2  ::= Op Unit Expr2
3:          | “”
4: Unit   ::= ‘(‘ Expr ‘)’
5:          |    ID
6: Op     ::= ‘+’
7:          |   ‘*’
```

How do we parse an Expr2?

Let’s look at the grammar

```
1: Expr  ::= Unit Expr2
2: Expr2 ::= Op Unit Expr2
3:          | “”
4: Unit  ::= ‘(‘ Expr ‘)’
5:          |   ID
6: Op    ::= ‘+’
7:          |   ‘*’
```

How do we parse an Expr2?

```
First+ sets:
1: {‘(‘, ID}
2: {‘+’, ‘*’}
3: {None, ’)’}
4: {‘(‘}
5: {ID}
6: {‘+’}
7: {‘*’}
```

Let’s look at the grammar

```
1: Expr  ::= Unit Expr2
2: Expr2 ::= Op Unit Expr2
3:         | “”
4: Unit  ::= ‘(‘ Expr ‘)’
5:         |   ID
6: Op    ::= ‘+’
7:         |   ‘*’
```

How do we parse an Expr2?

```
def parse_Expr2(self):

    token_id = get_token_id(self.to_match)

    # Expr2 ::= Op Unit Expr2
    if token_id in ["PLUS", "MULT"]:
        self.parse_Op()
        self.parse_Unit()
        self.parse_Expr2()
        return

    # Expr2 ::= ""
    if token_id in [None, ”RPAR"]:
        return
    else:
        raise ParserException(self.linennumber    # line number (for you to do)
                    self.to_match,                # observed token
                    ["PLUS", "MULT", ”RPAR"])     # expected token
```

```
First+ sets:
1: {‘(‘, ID}
2: {‘+’, ‘*’}
3: {None, ’)’}
4: {‘(‘}
5: {ID}
6: {‘+’}
7: {‘*’}
```

Let’s look at the grammar

```
1: Expr   ::= Unit Expr2
2: Expr2  ::= Op Unit Expr2
3:          | “”
4: Unit   ::= ‘(‘ Expr ‘)’
5:          |    ID
6: Op     ::= ‘+’
7:          |   ‘*’
```

How do we parse a Unit?

```
First+ sets:
1: {‘(‘, ID}
2: {‘+’, ‘*’}
3: {None, ’)’}
4: {‘(‘}
5: {ID}
6: {‘+’}
7: {‘*’}
```

Let’s look at the grammar

```
1: Expr   ::= Unit Expr2
2: Expr2  ::= Op Unit Expr2
3:          | “”
4: Unit   ::= ‘(‘ Expr ‘)’
5:          |    ID
6: Op     ::= ‘+’
7:          |   ‘*’
```

```
First+ sets:
1: {‘(‘, ID}
2: {‘+’, ‘*’}
3: {None, ’)’}
4: {‘(‘}
5: {ID}
6: {‘+’}
7: {‘*’}
```

How do we parse a Unit?

```
def parse_Unit(self):

    token_id = get_token_id(self.to_match)

    # Unit  ::= ‘(‘ Expr ‘)’
    if token_id == "LPAR":
        self.eat("LPAR")
        self.parse_Expr()
        self.eat("RPAR")
        return

    # Unit :: = ID
    if token_id == "ID":
        self.eat("ID")
        return
```

```
def eat(self, expected_token):
  if self.to_match == expected_token:
    self.advance()  # move to the next token
  else:
    raise ParserException(
          self.current_line,    # line# (for you to do)
          self.to_match,        # observed token
          [expected_token])  #   i.e. PAR or RPAR
```

Let's look at the grammar

How do we parse a Unit?

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

```
First+ sets:
1: {'(', ID}
2: {'+', '*'}
3: {None, ')'}
4: {'('}
5: {ID}
6: {'+'}
7: {'*'}
```

```
def parse_Unit(self):
    token_id = get_token_id(self.to_match)

    # Unit  ::= '(' Expr ')'
    if token_id == "LPAR":
        self.eat("LPAR")
        self.parse_Expr()
        self.eat("RPAR")
        return

    # Unit :: = ID
    if token_id == "ID":
        self.eat("ID")
        return
```

Note: function eat must ensure that to_match has the expected token ID and advances to the next token, i.e. something like this:

```
def eat(self, expected_token):
    if self.to_match == expected_token:
        self.advance()  # move to the next token
    else:
        raise ParserException(
                 self.current_line,     # line# (for you to do)
                 self.to_match,         # observed token
                 [expected_token])      # LPAR or RPAR or ID)
```

```
class ParserException(Exception):
    def __init__(self, line_number, found_token, expected_tokens):
        self.line_number = line_number
        self.found_token = found_token
        self.expected_tokens = expected_tokens
        # Create a readable error message
        message = (f"Parse error on line {line_number}: found 
'{found_token}', "
                   f"expected one of {expected_tokens}")
        super().__init__(message)
```

Let’s look at the grammar

```
1: Expr   ::= Unit Expr2
2: Expr2  ::= Op Unit Expr2
3:          | “”
4: Unit   ::= ‘(‘ Expr ‘)’
5:          |    ID
6: Op     ::= ‘+’
7:          |    ‘*’
```

How do we parse an Op?

```
First+ sets:
1: {‘(‘, ID}
2: {‘+’, ‘*’}
3: {None, ’)’}
4: {‘(‘}
5: {ID}
6: {‘+’}
7: {‘*’}
```

Let’s look at the grammar

```
1: Expr  ::= Unit Expr2
2: Expr2 ::= Op Unit Expr2
3:         | “”
4: Unit  ::= ‘(‘ Expr ‘)’
5:         |    ID
6: Op    ::=  ‘+’
7:         |    ‘*’
```

```
First+ sets:
1: {‘(‘, ID}
2: {‘+’, ‘*’}
3: {None, ’)’}
4: {‘(‘}
5: {ID}
6: {‘+’}
7: {‘*’}
```

How do we parse an Op?

```
def parse_Op(self):

    token_id = get_token_id(self.to_match)

    # Op  ::= '+'
    if token_id == "PLUS":
        self.eat("PLUS")
        return

    # Op  ::= '*'
    if token_id == "MULT":
        self.eat("MULT")
        return
```

```
def eat(self, expected_token):
    …
    raise ParserException(
            self.current_line,
            self.to_match,        # observed token
            [expected_token]) # expected token
    …
```

Recursive Descent

IS THAT SIMPLE

and allows lookahead > 1