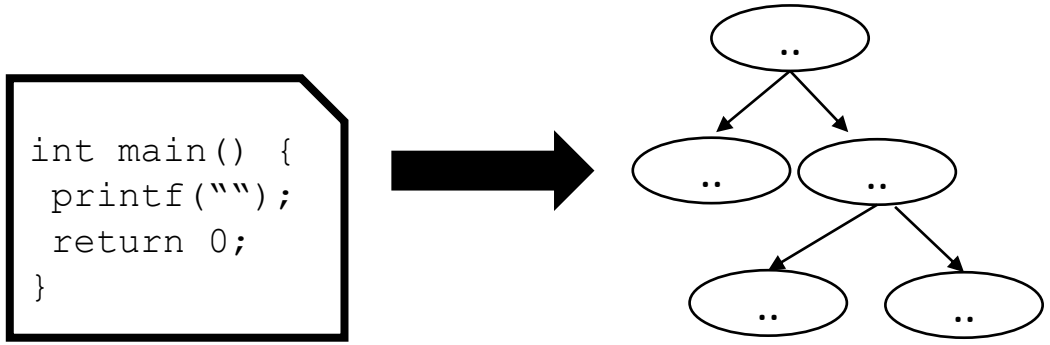
CSE211: Compiler Design

Oct. 6, 2023



- **Topic**: Parsing overview 3

- **Questions**:
  - *How are regular expressions used in a compiler?*
  - *What are limitations of regular expressions?*

Announcements

- Piazza is up! Please enroll. It should be considered required!

- My office hours will be on Thursday 3 – 5 PM
  - Starting next week
  - I'll send out a message around noon on Thursday with a signup sheet

- Rithik will be having his office hours too, just getting room reservations sorted out.

Announcements

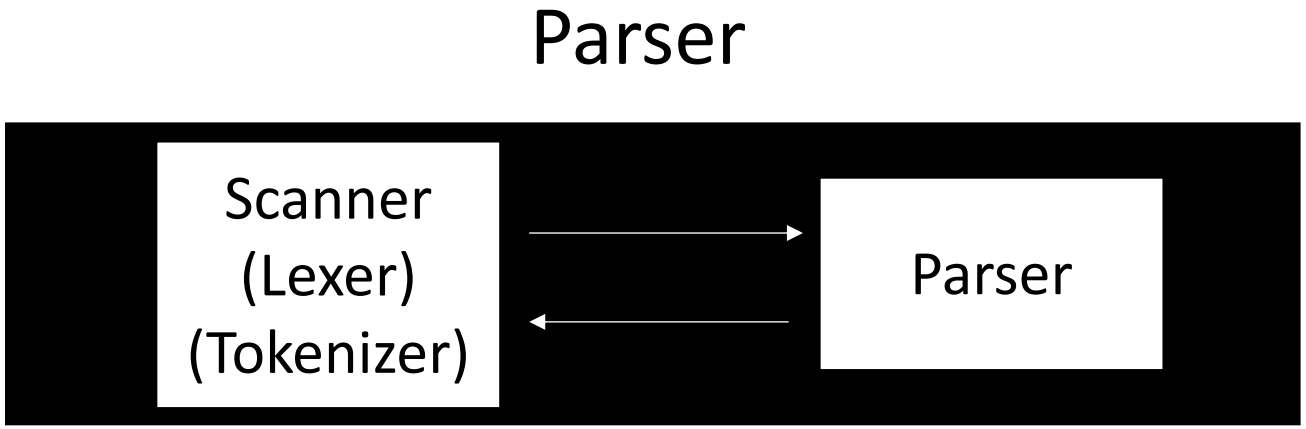
- Homework 1 is planned for release on Monday by Midnight
  - Please start thinking about partners
  - Please self organize (use Piazza)
  - You will have 2 weeks to do the homework
- Any remaining undergrads should get a permission code ASAP
- If anyone isn’t on Canvas, please let me know

Announcements

- Think about paper review
  - You will need to approve a paper with me by Oct. 23
  - First review is due Oct. 30
  - You should probably not wait until these due dates because the midterm is also on Oct. 30.
  - I give this time for you to organize, not as a guidance!
  - You can discuss papers on piazza or ask me for suggestions

Review

Parser architecture



First level of abstraction. Transforms a string of characters into a string of tokens

***Language**:*
Regular Expressions (REs)

Second level: transforms a string of tokens in a tree of tokens.

***Language**:*
Context-Free Grammars (CFGs)

Scanner

(5 + 4) * 8

Scanner

```
[[(LPAR, "(") (NUM, "5") (PLUS, "+") (NUM, "4") (RPAR, ")")
 (TIMES, "*") (NUM, "8")]
```

Splits an input sentence it into lexemes

Dealing with a stream of input

How does this input get tokenized?

x++;

Tokens:
ID = “[a-z]”
OP = “+|++”

Dealing with a stream of input

How to fix it?

x++;

Tokens:
ID = “[a-z]”
OP = “+|++”

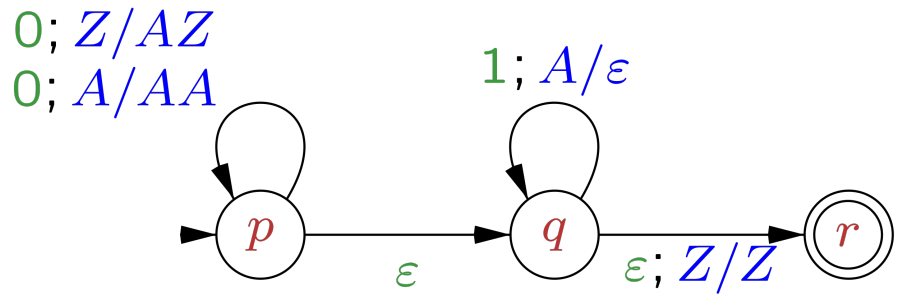
Dealing with streams

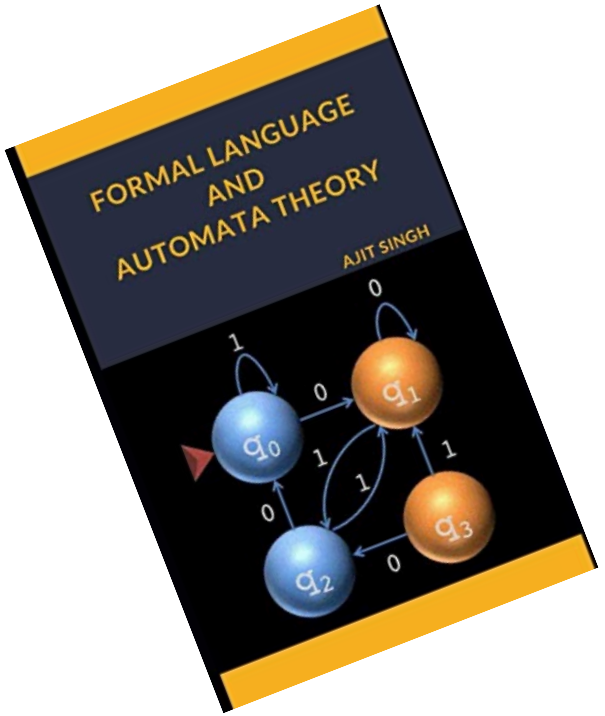
- Scanners will always return the token with the longest match
  - If you are implementing a scanner, you need to ensure this!
  - If you are using a scanner, you can depend on this!

- Streaming RE matchers (e.g. `re.match`) are not guaranteed to return the longest match when using a union

Context Free Grammars

- Backus–Naur form (BNF)
  - A syntax for representing context free grammars
  - Naturally creates tree-like structures
- More powerful than regular expressions





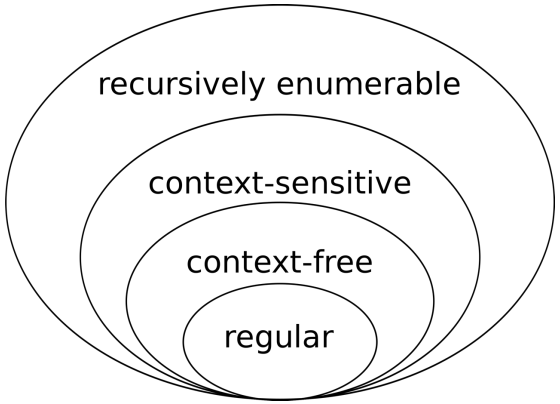


Image Credit:
By Jochgem - Own work, CC BY-SA 3.0,
https://commons.wikimedia.org/w/index.php?curid=5036988

BNF Production Rules

- <production name> : <token list>
  - Example:
    sentence: ARTICLE NOUN VERB

- <production name> : <token list> | <token list>
  - Example:

  sentence: ARTICLE ADJECTIVE NOUN VERB
  | ARTICLE NOUN VERB

Convention: Tokens in all caps, production rules in lower case

BNF Production Rules

- Production rules can reference other production rules

sentence: non_adjective_sentence
| adjective_sentence

non_adjective_sentence: ARTICLE NOUN VERB

adjective_sentence: ARTICLE ADJECTIVE NOUN VERB

BNF Production Rules

sentence: ARTICLE ADJECTIVE NOUN VERB*

BNF Production Rules

sentence: ARTICLE ADJECTIVE NOUN VERB*

We cannot do the star in production rules

BNF Production Rules

- Production rules can be recursive
  - Imagine a list of adjectives:
    “The small brown energetic dog barked”

sentence: ARTICLE adjective_list NOUN VERB

adjective_list: ADJECTIVE adjective_list

| <empty>

New material

Let’s go back to mathematical sentences (expressions)

- First lets define tokens:
  - NUM =
  - PLUS =
  - TIMES =

Let’s limit ourselves to non-negative numbers and +,*.

How can we make BNF production rules for this?

Let’s go back to mathematical sentences (expressions)

- First lets define tokens:
  - NUM = [0-9]+
  - PLUS = ‘\+’
  - TIMES = ’*’

expression : NUM

| expression PLUS expression

| expression TIMES expression

Let’s go back to mathematical sentences (expressions)

- First lets define tokens:
  - NUM = [0-9]+
  - PLUS = ‘\+’
  - TIMES = ’*’

Let’s add () to the language!

expression : NUM

| expression PLUS expression

| expression TIMES expression

Let’s go back to mathematical sentences (expressions)

- First lets define tokens:
  - NUM = [0-9]+
  - PLUS = ‘\+’
  - TIMES = ’*’
  - LPAREN = ‘\(‘
  - RPAREN = ‘\)’

expression : NUM
| expression PLUS expression
| expression TIMES expression
| LPAREN expression RPAREN

How to determine if a string matches a CFG?

Parse trees

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

```
input: 5
```

expr : NUM

| expr PLUS expr

| expr TIMES expr

| LPAREN expr RPAREN

Parse trees

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

```
input: 5
```

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

expr

Parse trees

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

```
input: 5
```

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

expr

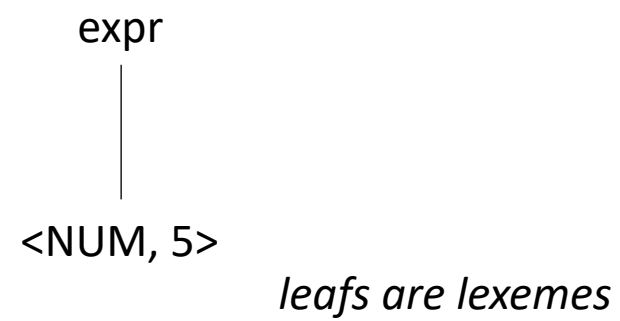
root of the tree is the entry production

Parse trees

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

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

```
input: 5
```



leafs are lexemes

Parse trees

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

```
input: 5*6
```

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

Parse trees

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

```
input: 5*6
```

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

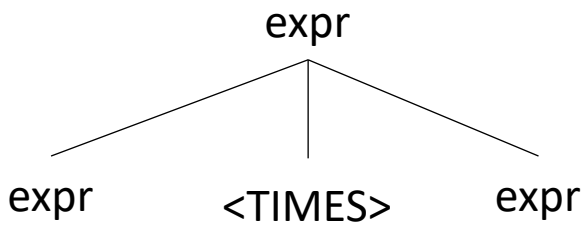
expr

Parse trees

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

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

```
input: 5*6
```

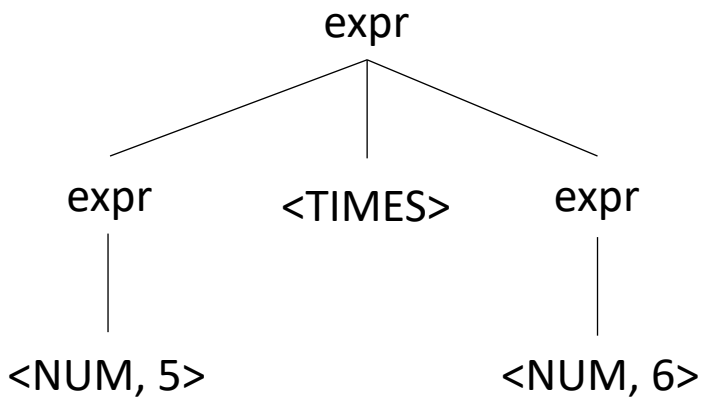


Parse trees

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

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

```
input: 5*6
```



Parse trees

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

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

What happens in an error?

expr : NUM

| expr PLUS expr

| expr TIMES expr

| LPAREN expr RPAREN

expr

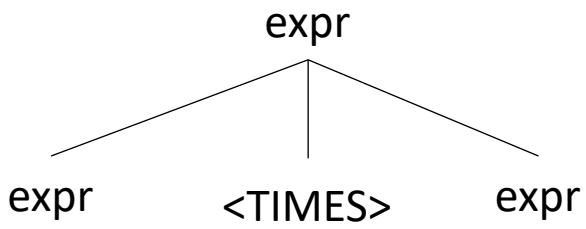
Parse trees

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

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

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

What happens in an error?



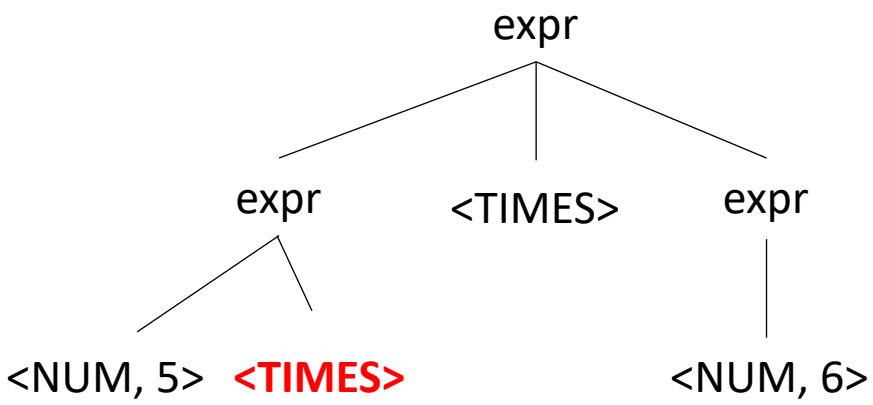
Parse trees

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

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

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

What happens in an error?



Not possible!

Parse trees

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

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

expr : NUM

| expr PLUS expr

| expr TIMES expr

| LPAREN expr RPAREN

Parse trees

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

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

expr : NUM

| expr PLUS expr

| expr TIMES expr

| LPAREN expr RPAREN

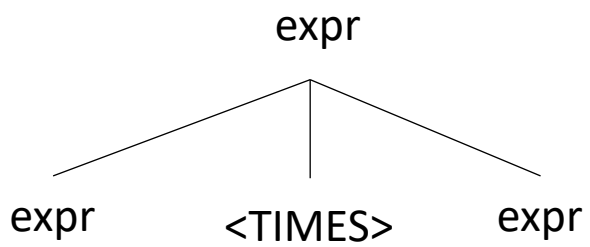
expr

Parse trees

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

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

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

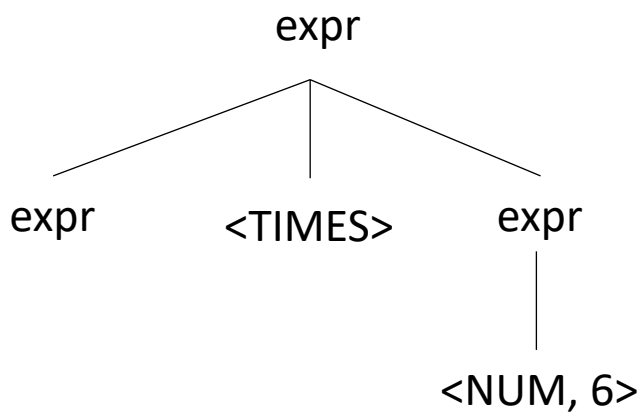


Parse trees

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

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

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

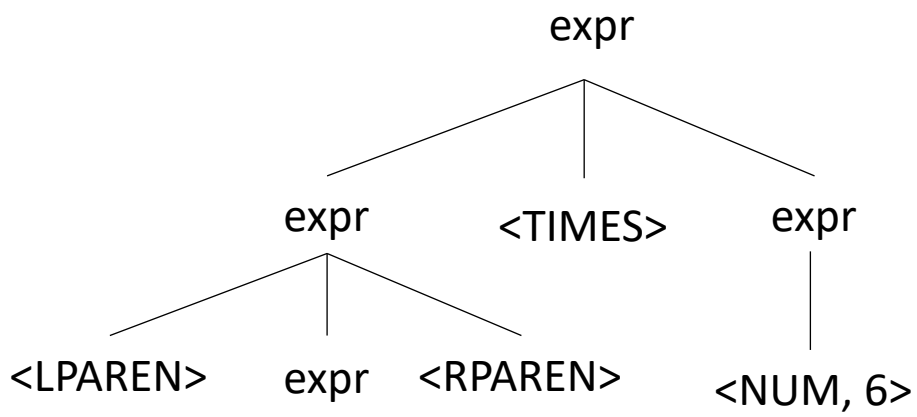


Parse trees

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

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

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

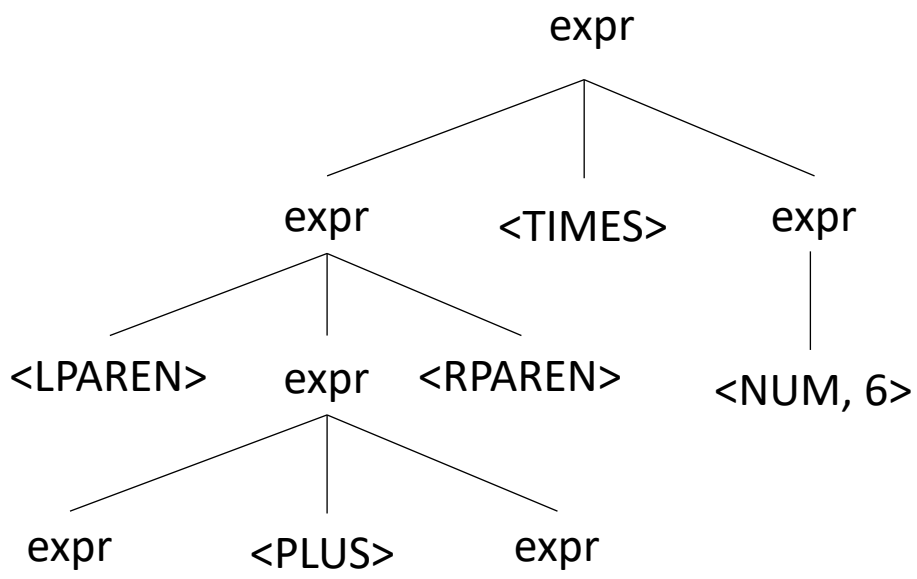


Parse trees

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

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

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

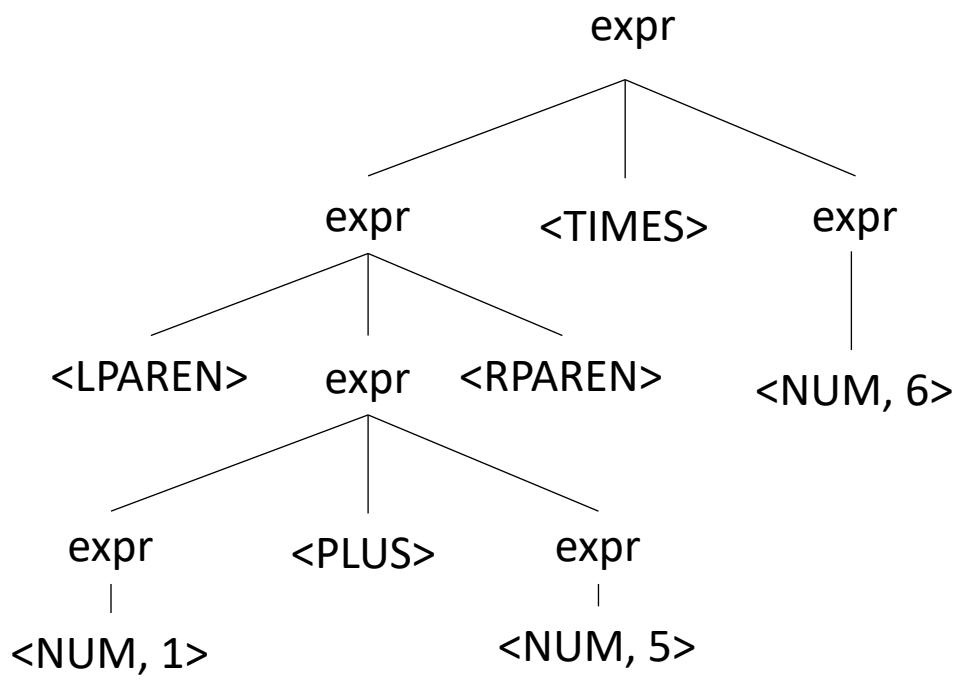


Parse trees

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

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

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

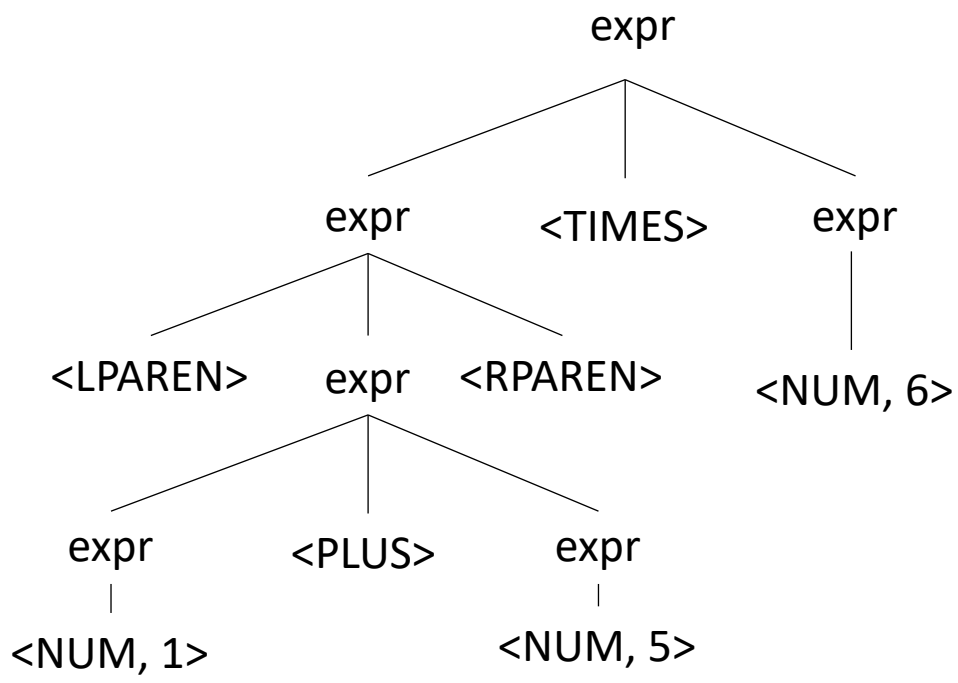


Parse trees

- Reverse question: given a parse tree: how do you create a string?

```
input: ?
```

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



Ambiguous grammars

“I saw a person on a hill with a telescope.”

What does it mean??

https://www.quora.com/What-are-some-examples-of-ambiguous-sentences

Parse trees

- Try making a parse tree from: `1 + 5 * 6`

expr : NUM

| expr PLUS expr

| expr TIMES expr

| LPAREN expr RPAREN

Parse trees

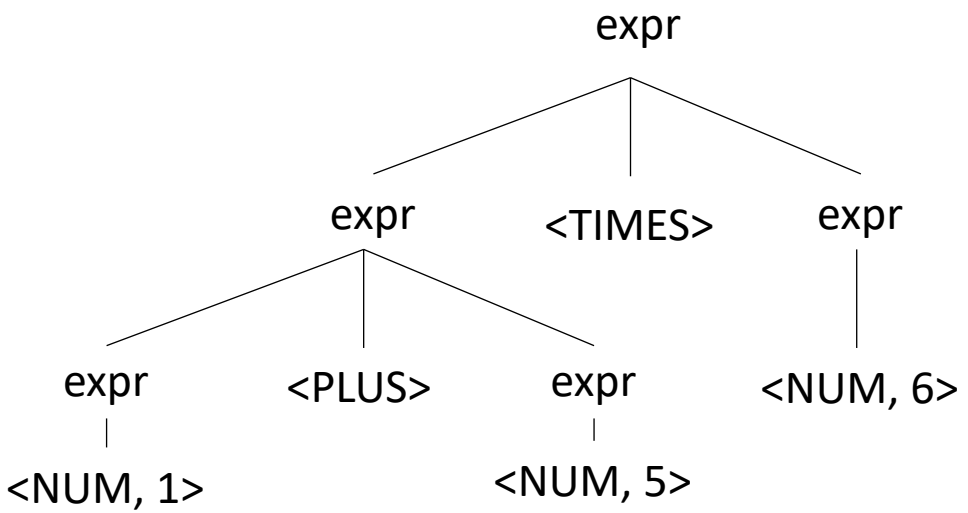
- Try making a parse tree from: `1 + 5 * 6`

expr : NUM

| expr PLUS expr

| expr TIMES expr

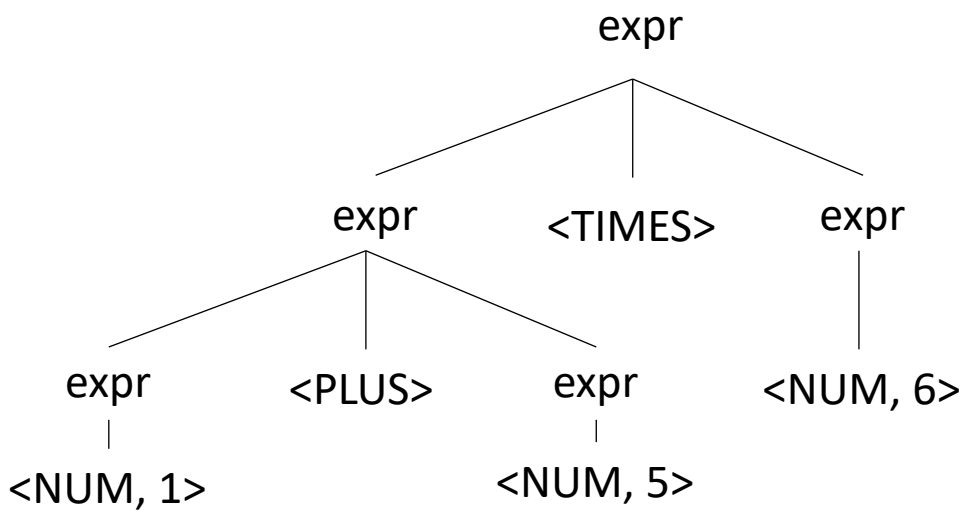
| LPAREN expr RPAREN



Parse trees

- `input: 1 + 5 * 6`

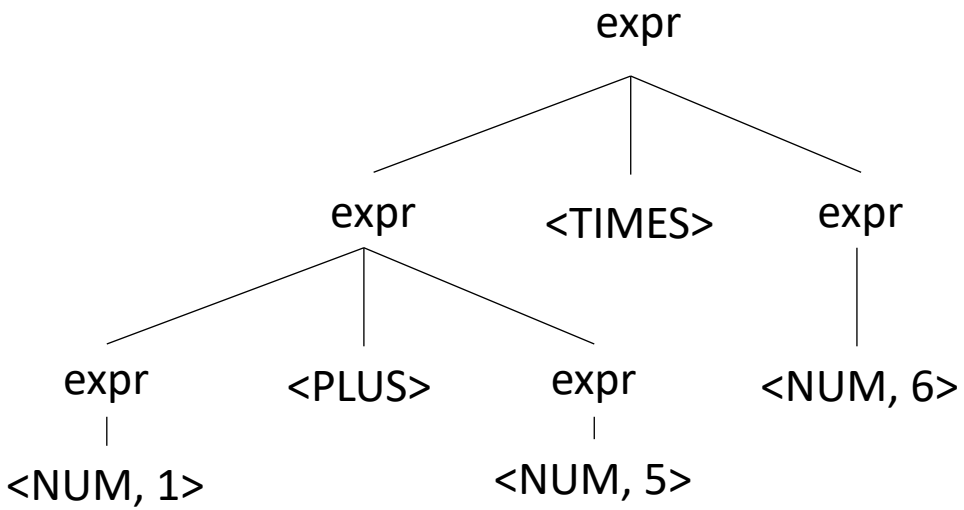
```
expr : NUM
     | expr PLUS expr
     | expr TIMES expr
     | LPAREN expr RPAREN
```

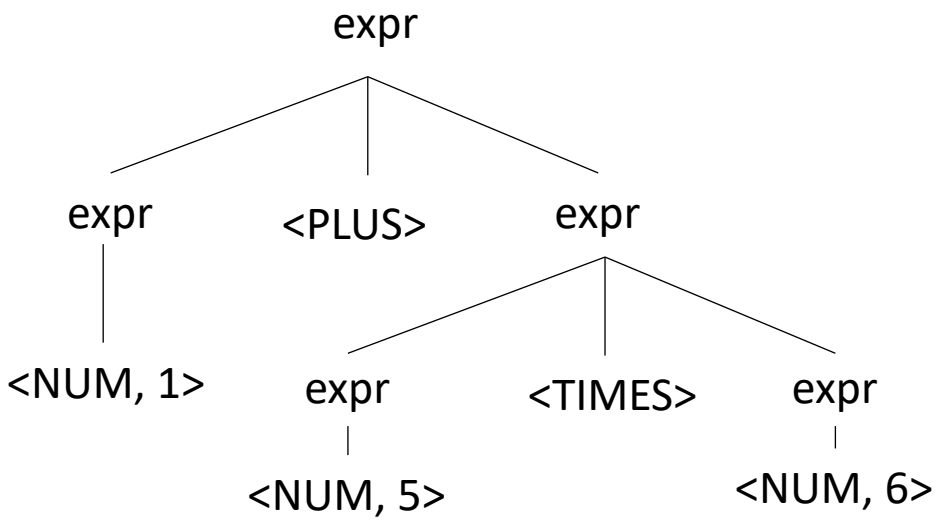


Parse trees

- `input: 1 + 5 * 6`

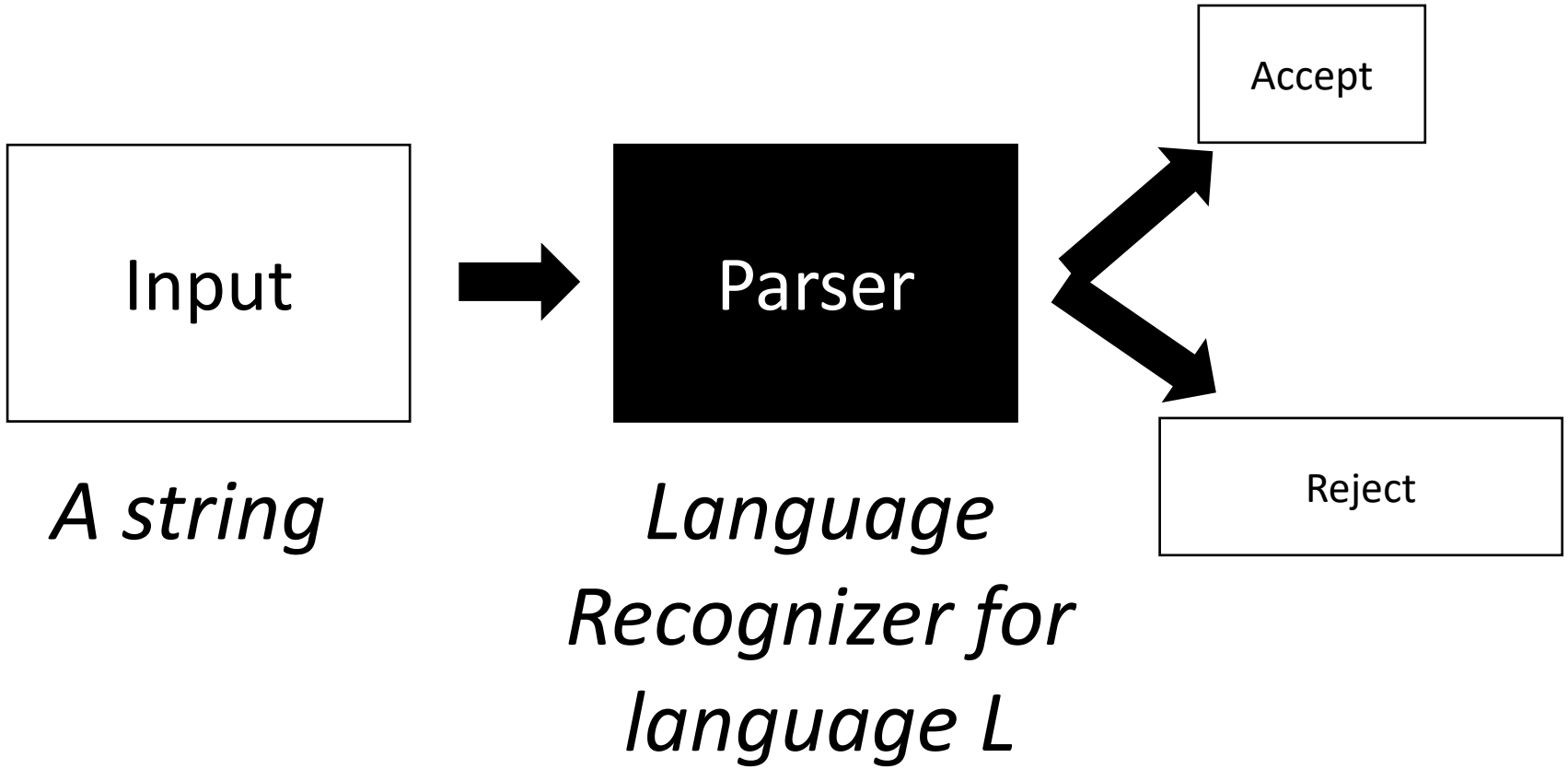
```
expr : NUM
     | expr PLUS expr
     | expr TIMES expr
     | LPAREN expr RPAREN
```





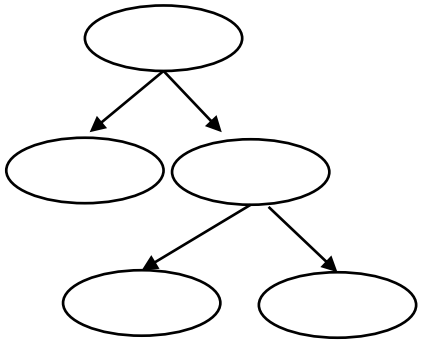
Ambiguous grammars

- What's the issue?

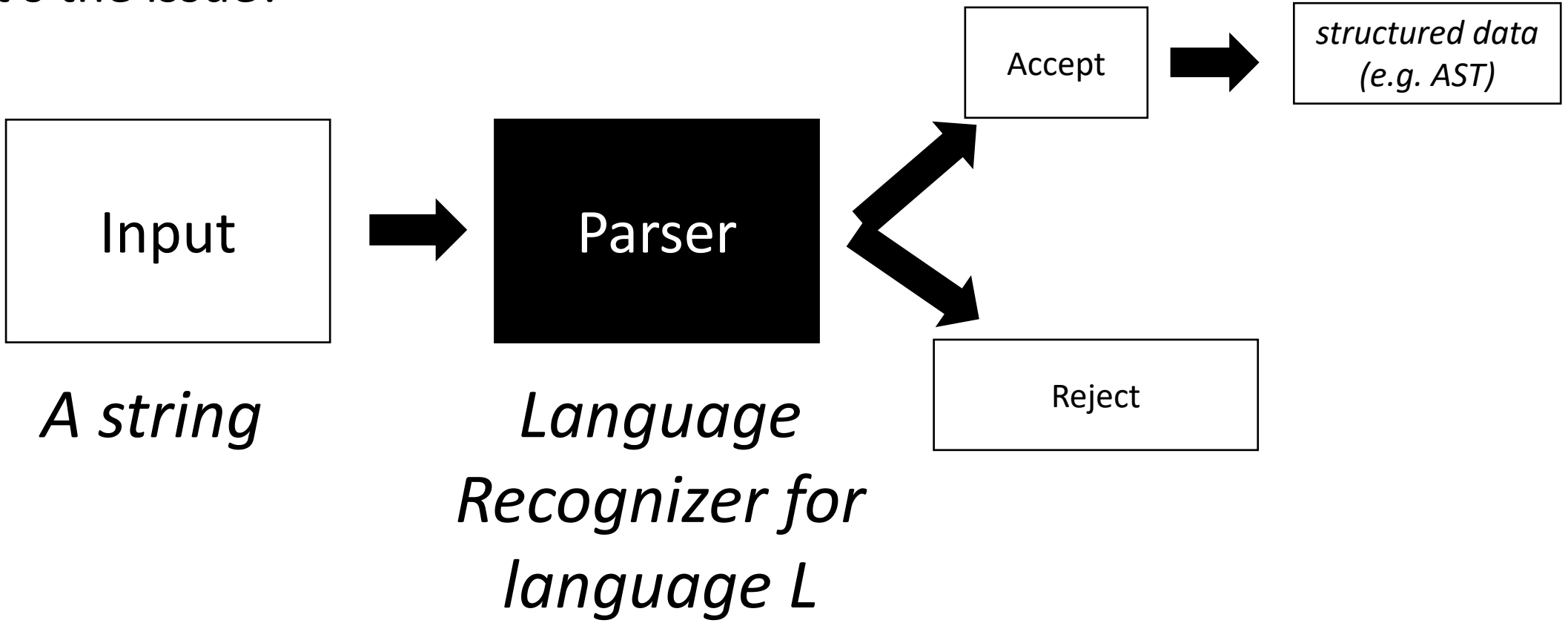


Ambiguous grammars

continue to the rest of compilation



- What's the issue?



Meaning into structure

- Structural meaning defined to be a post-order traversal

Meaning into structure

- Structural meaning defined to be a post-order traversal
  - Children return values to their parent
  - Nodes are only evaluated once all their children have been evaluated
  - Evaluated from left to right
  - Also called “Natural Order”

Meaning into structure

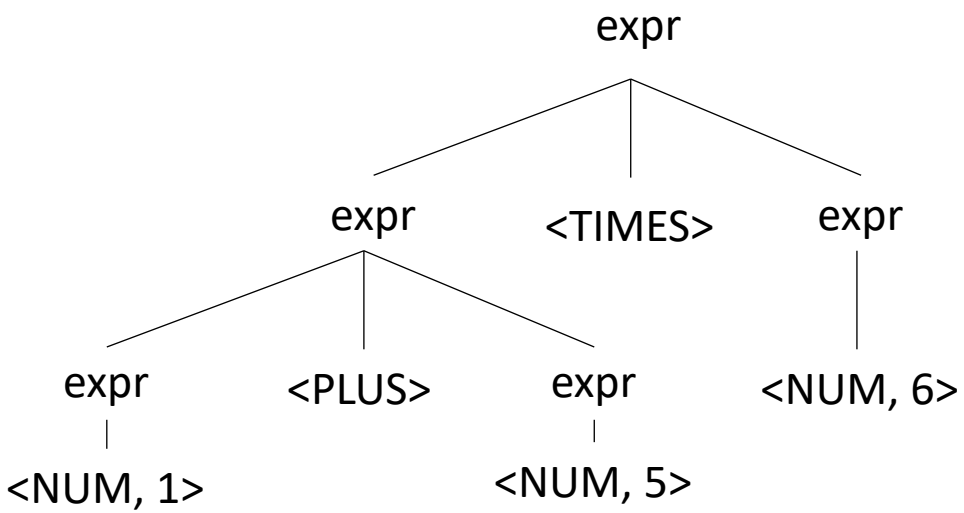
- Structural meaning defined to be a post-order traversal
  - Children return values to their parent
  - Nodes are only evaluated once all their children have been evaluated
  - Evaluated from left to right

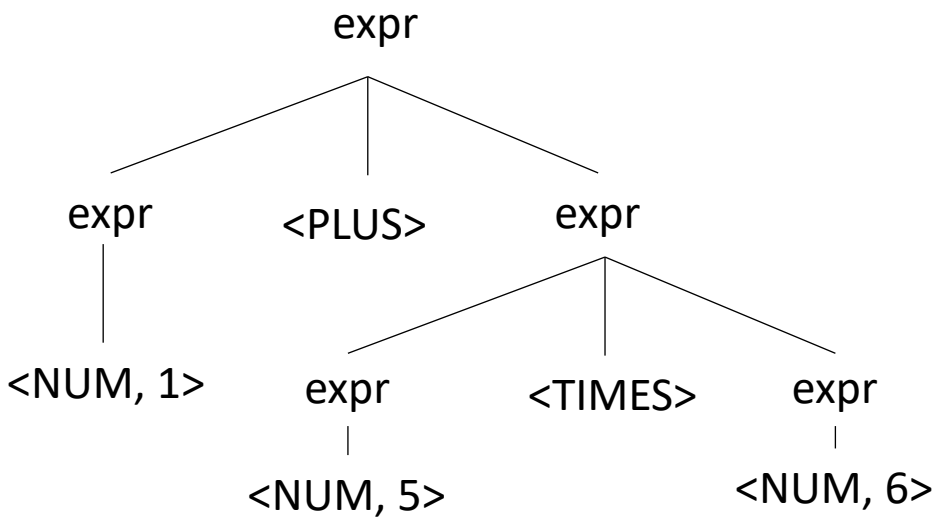
- Can also encode the order of operation

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 rule 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 rule 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
```

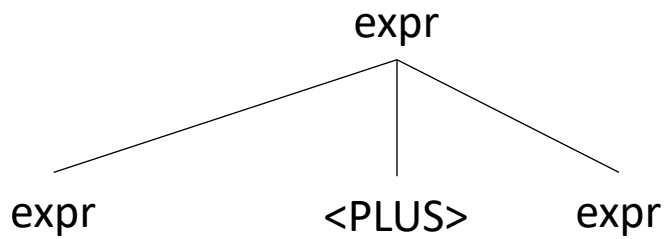
expr

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
```

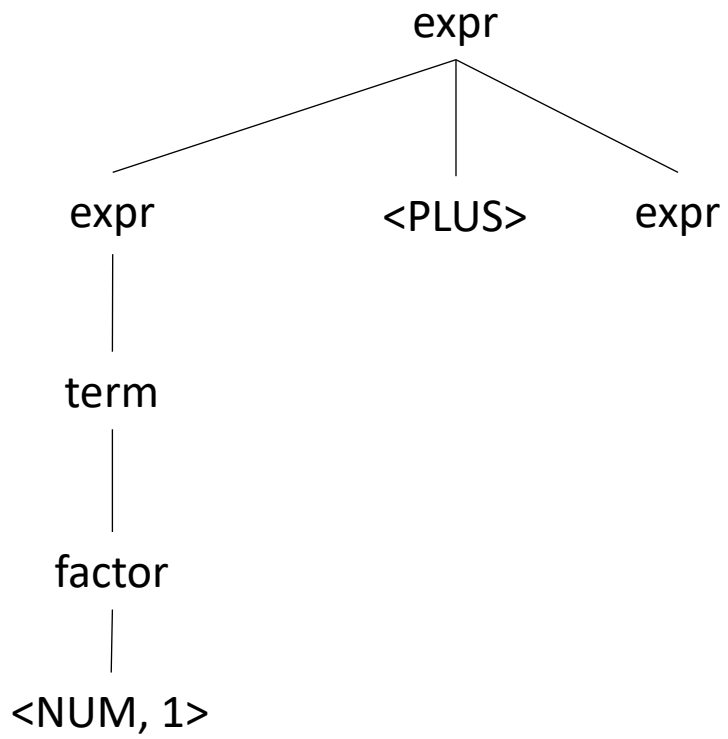
Operator	Name	Productions
+	expr	: expr PLUS expr | term
*	term	: term TIMES term | factor
()	factor	: LPAREN expr RPAREN | NUM



Now lets create a parse tree

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

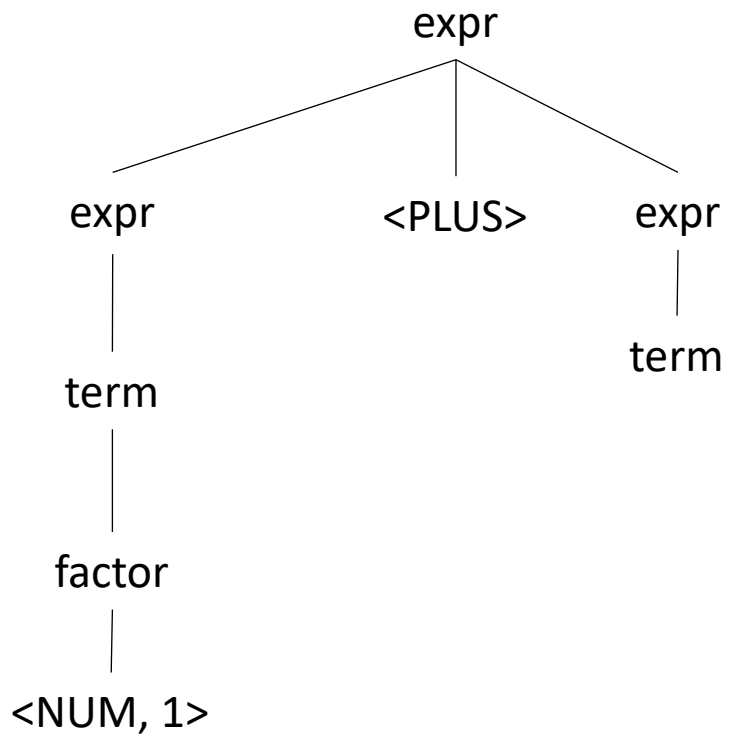
Operator	Name	Productions
+	expr	: expr PLUS expr | term
*	term	: term TIMES term | factor
()	factor	: LPAREN expr RPAREN | NUM



Now lets create a parse tree

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

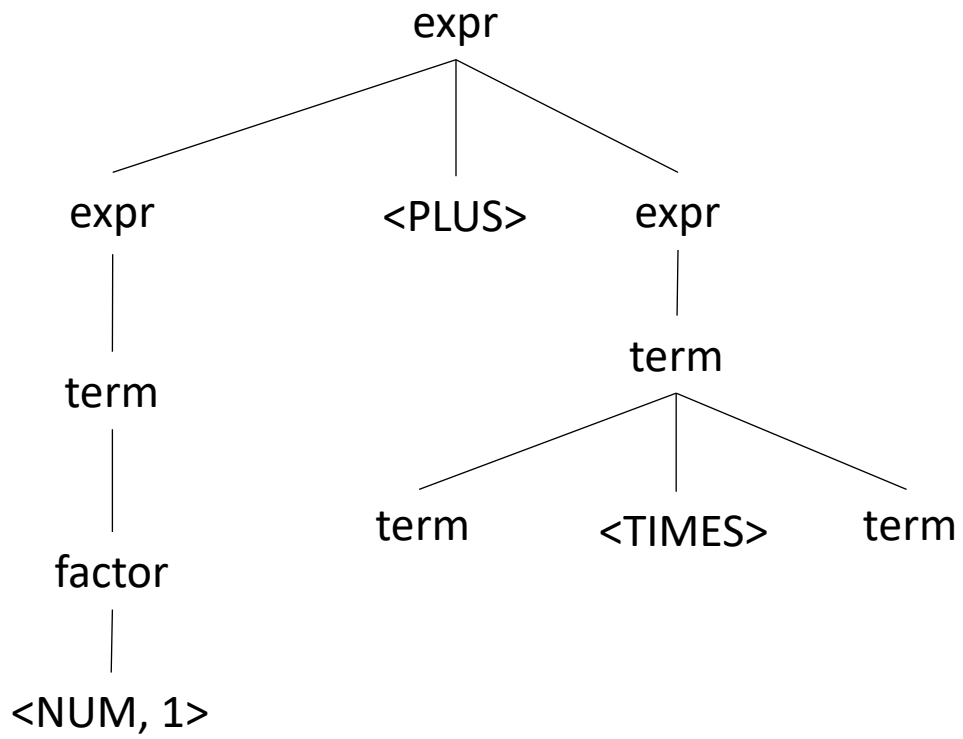
Operator	Name	Productions
+	expr	: expr PLUS expr | term
*	term	: term TIMES term | factor
()	factor	: LPAREN expr RPAREN | NUM



Now lets create a parse tree

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

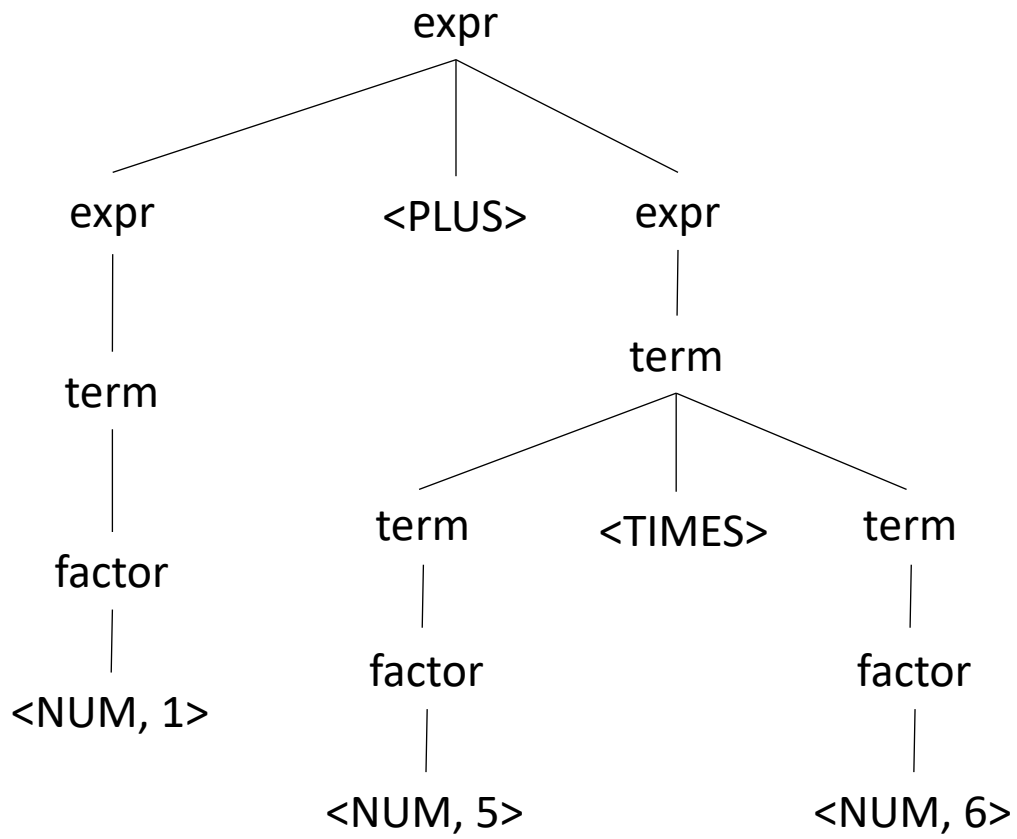
Operator	Name	Productions
+	expr	: expr PLUS expr | term
*	term	: term TIMES term | factor
()	factor	: LPAREN expr RPAREN | NUM



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



What other sources of ambiguity?

Let’s make some more parse trees

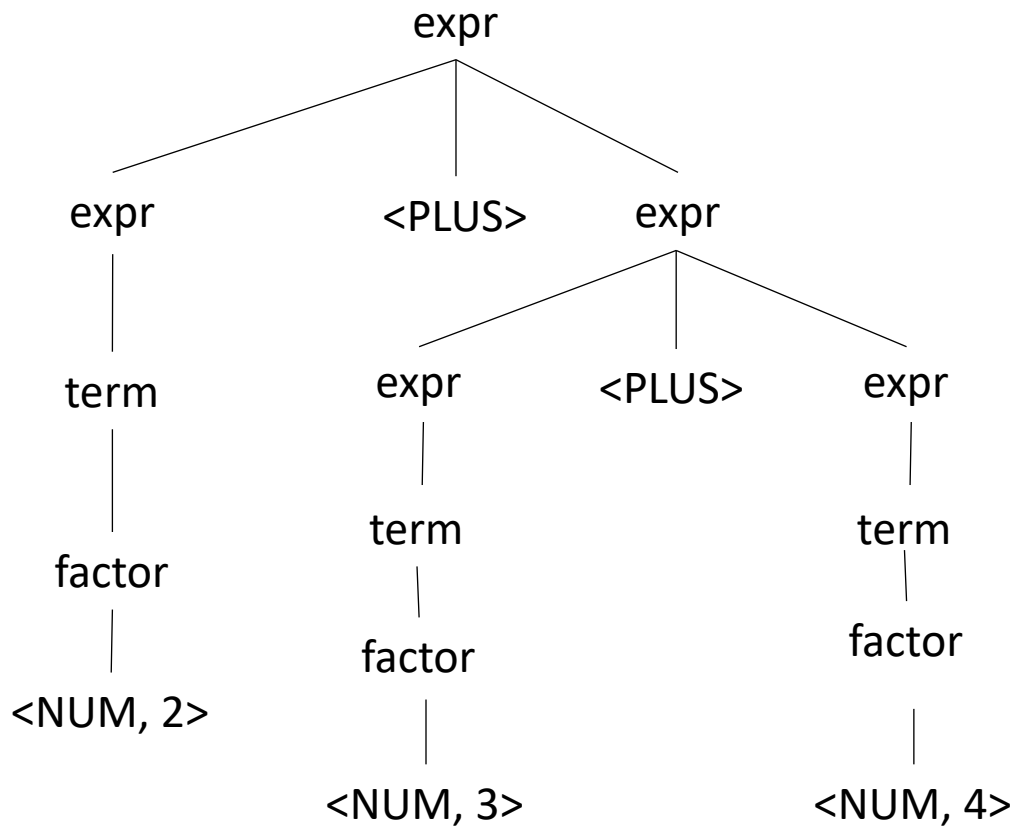
```
input: 2+3+4
```

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

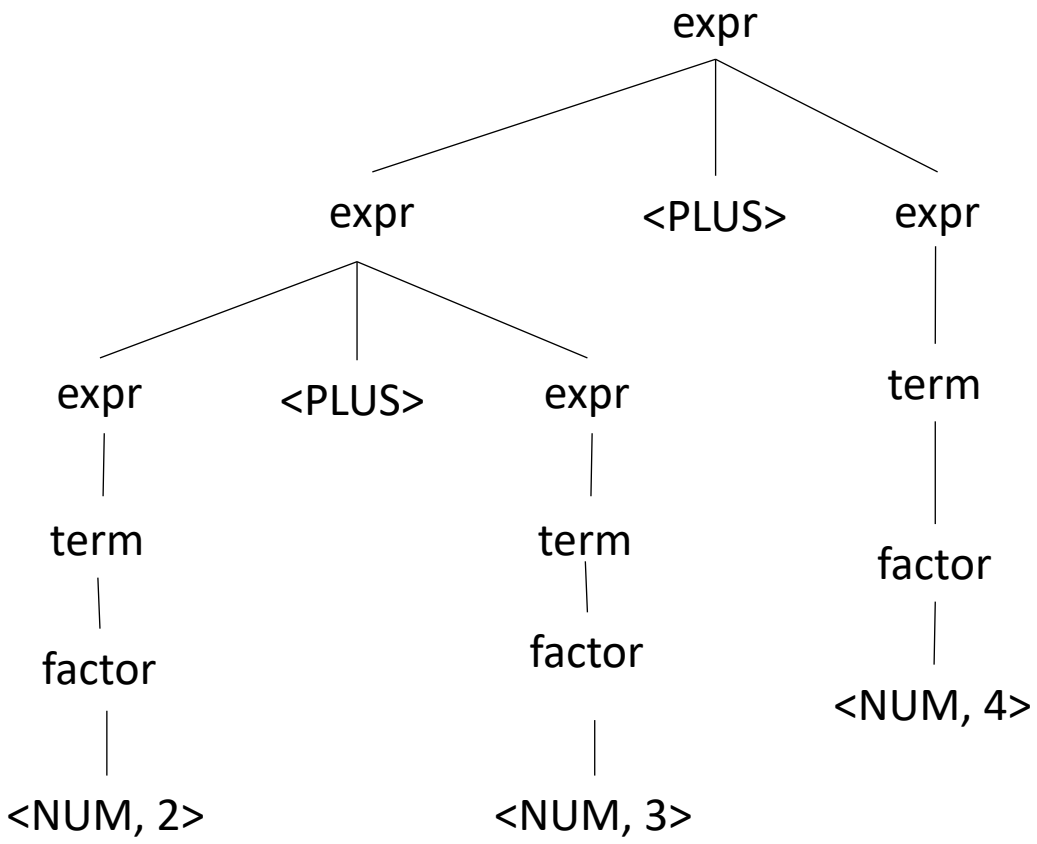
Operator	Name	Productions
+	expr	: expr PLUS expr | term
*	term	: term TIMES term | factor
()	factor	: LP expr RP | NUM

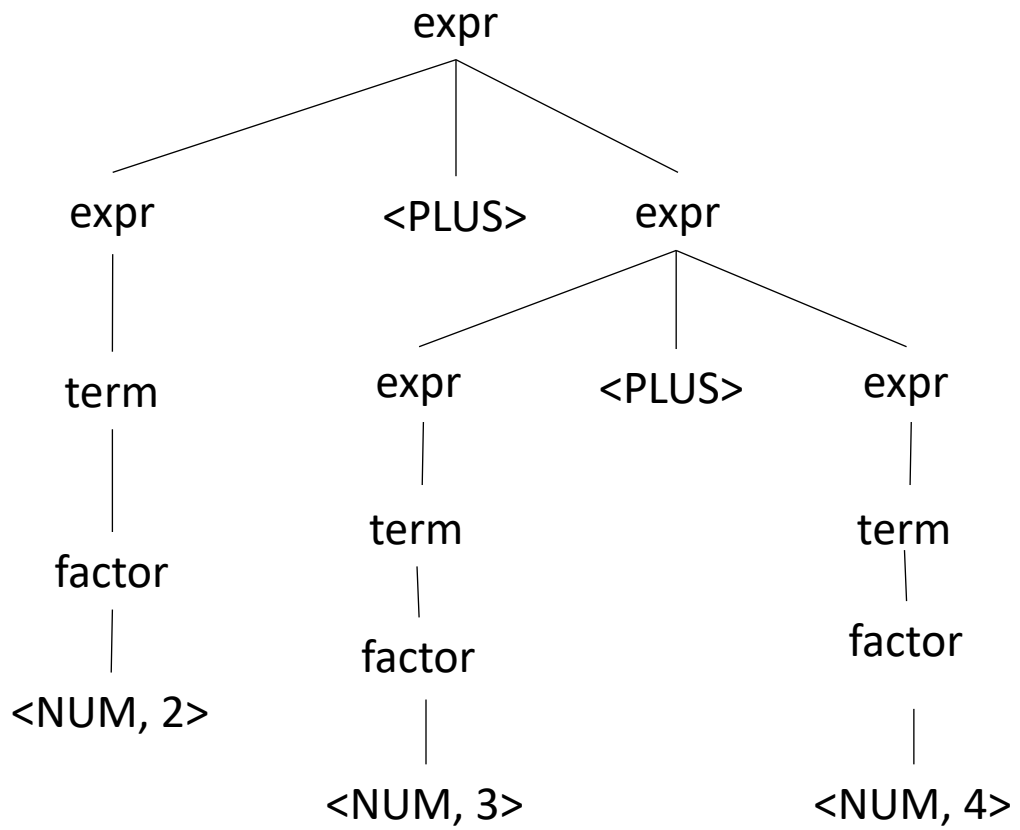
`input: 2+3+4`



This is ambiguous, is it an issue?

`input: 2+3+4`



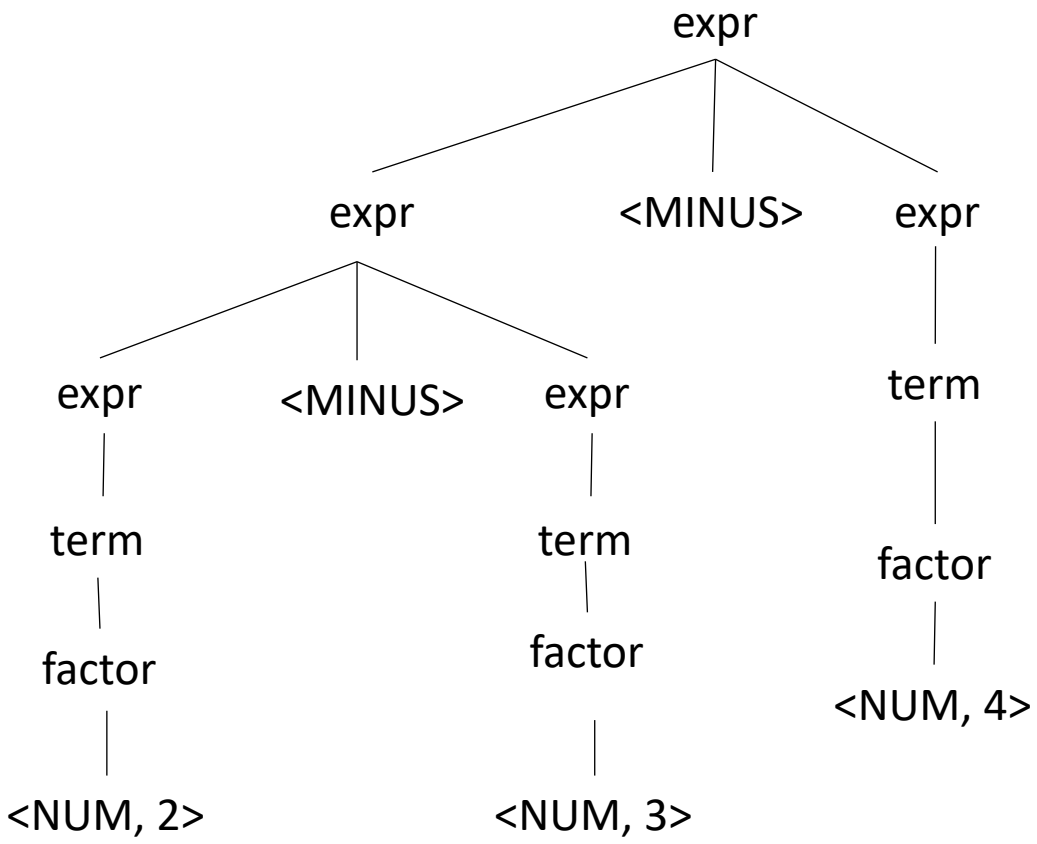


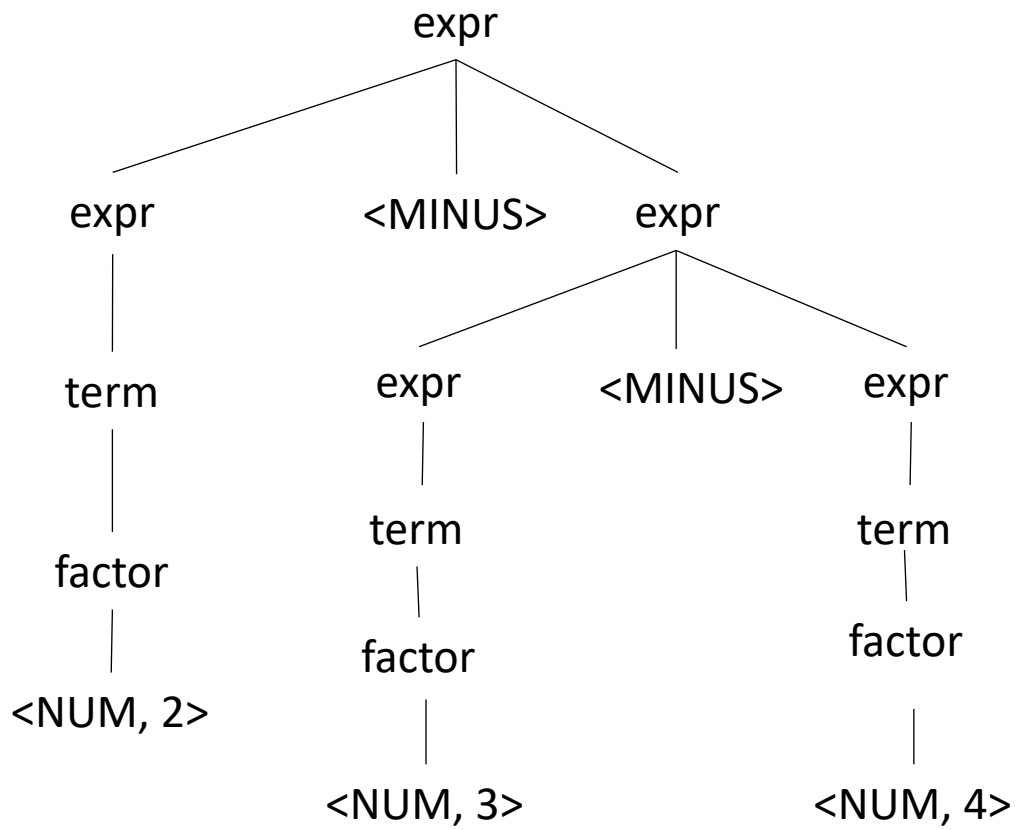
What about for a different operator?

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

What about for a different operator?

`input: 2-3-4`





Which one is right?

Associativity

The order in which we evaluate the same operator

Sometimes it doesn't matter:

- Integer arithmetic
- Integer multiplication
- What else?

Good test:

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

What about floating point arithmetic?

Associativity

The order in which we evaluate the same operator

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

- right-to-left (right-associative)
  - Any operators you can think of?

Associativity

The order in which we evaluate the same operator

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

- right-to-left (right-associative)
  - Any operators you can think of?
  - Assignment, power operator

How to encode associativity?

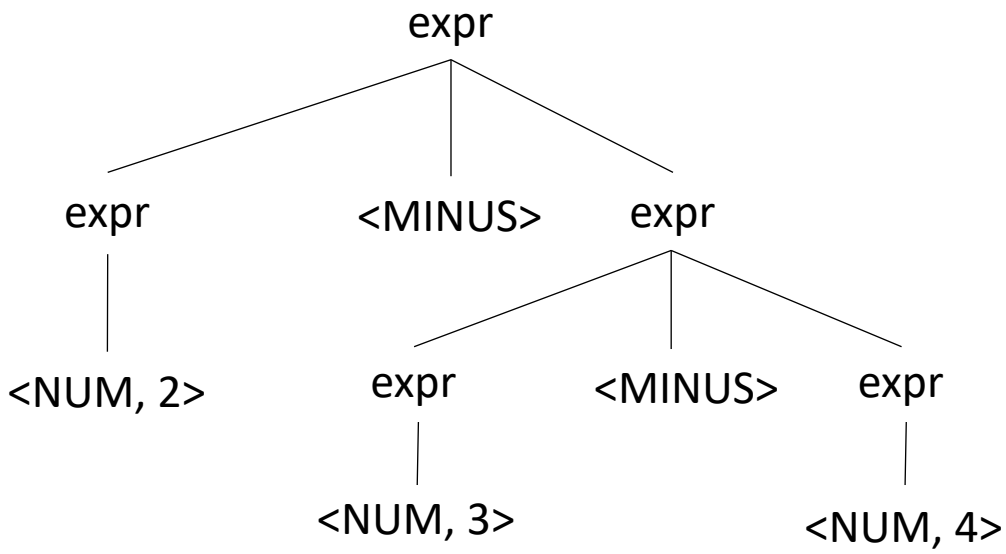
- Like precedence, some tools (e.g. YACC) allow associativity specification through keywords:
  - “+”: left, “^”: right

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

Associativity for a single operator

Operator	Name	Productions
-	expr	: expr MINUS NUM | NUM

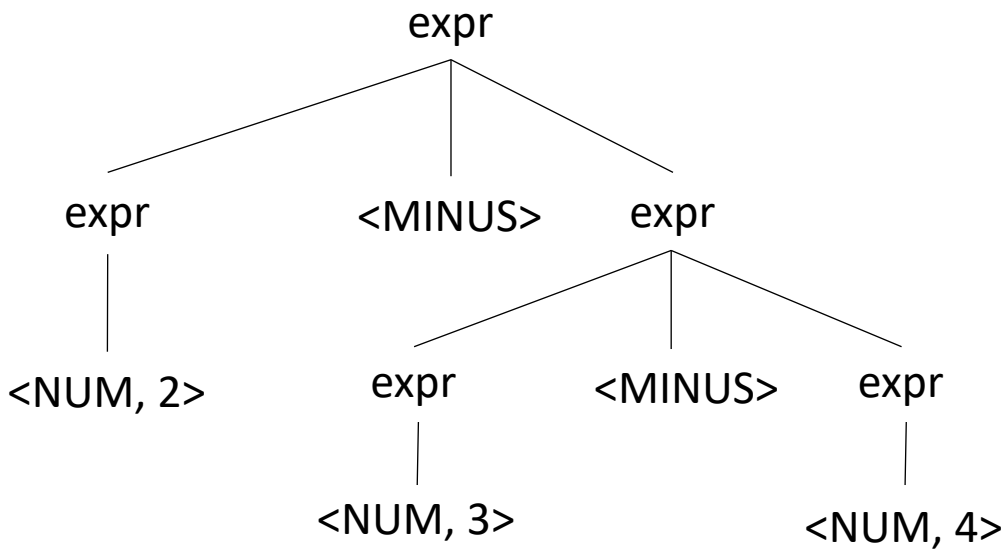
```
input: 2-3-4
```



Associativity for a single operator

Operator	Name	Productions
-	expr	: expr MINUS NUM | NUM

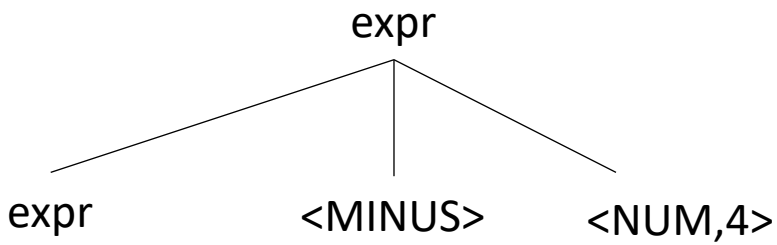
`input: 2-3-4`



No longer allowed

Associativity for a single operator

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

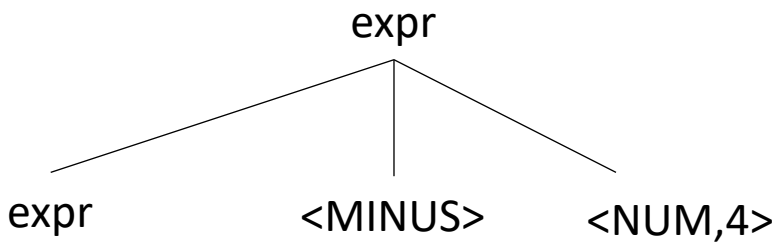


Operator	Name	Productions
-	expr	: expr MINUS NUM | NUM

Lets start over

Associativity for a single operator

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

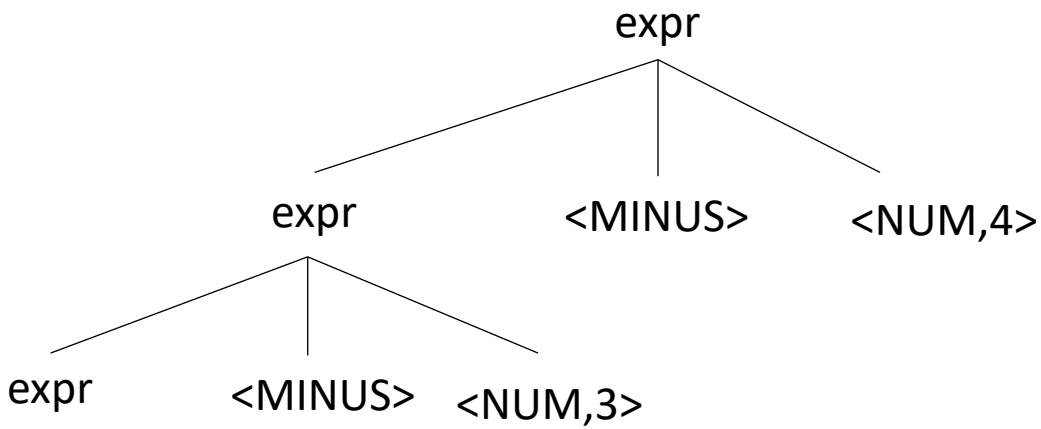


Operator	Name	Productions
-	expr	: expr MINUS NUM | NUM

Associativity for a single operator

Operator	Name	Productions
-	expr	: expr MINUS NUM | NUM

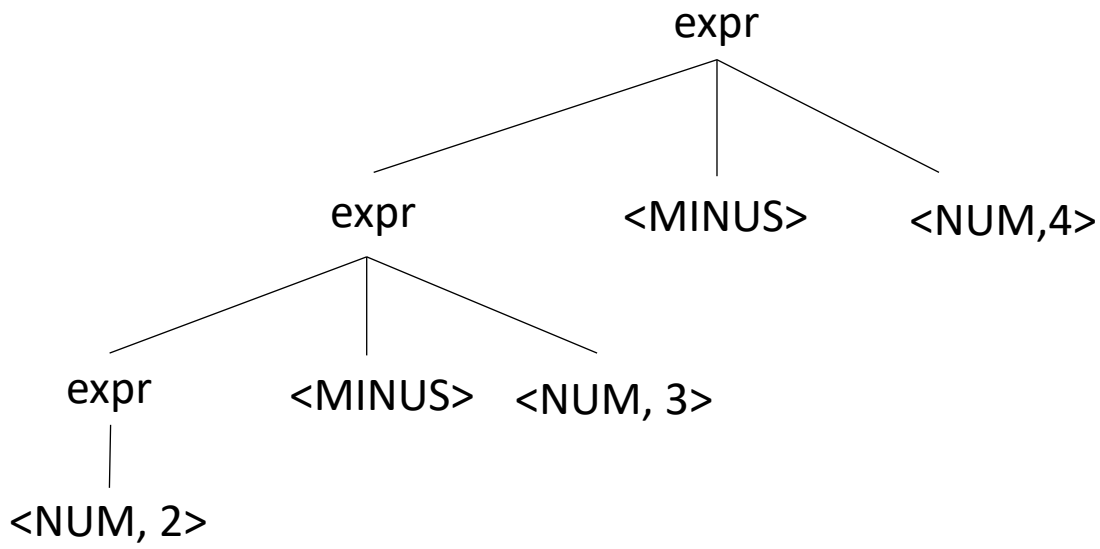
```
input: 2-3-4
```



Associativity for a single operator

Operator	Name	Productions
-	expr	: expr MINUS NUM | NUM

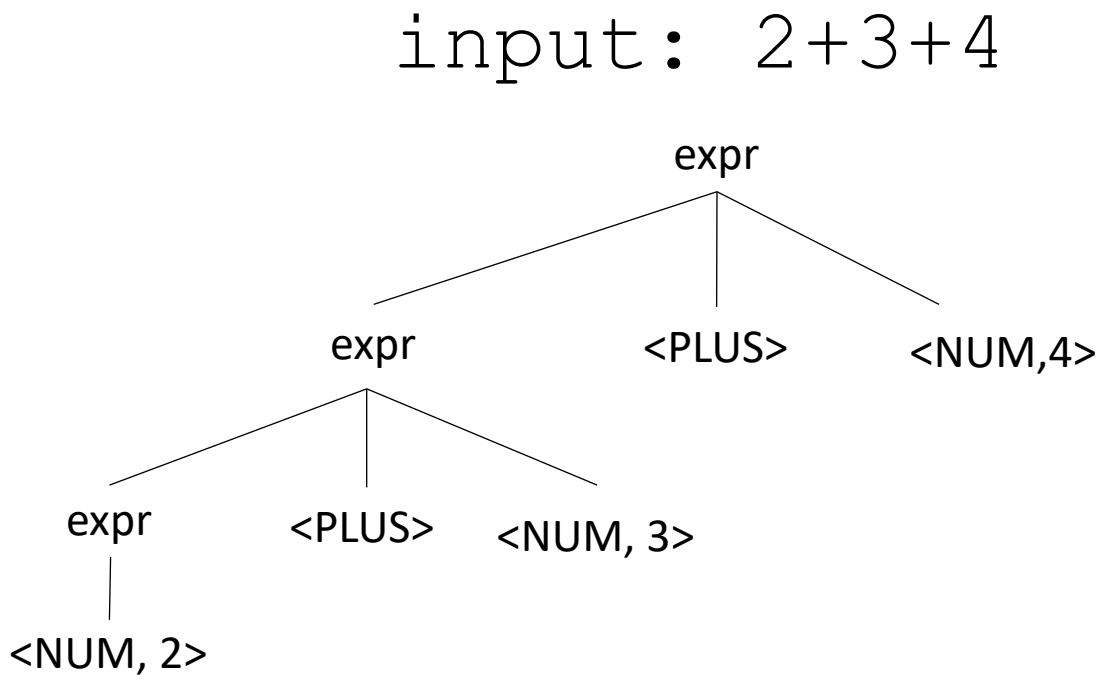
```
input: 2-3-4
```



Should you have associativity when its not required?

Benefits?
Drawbacks?

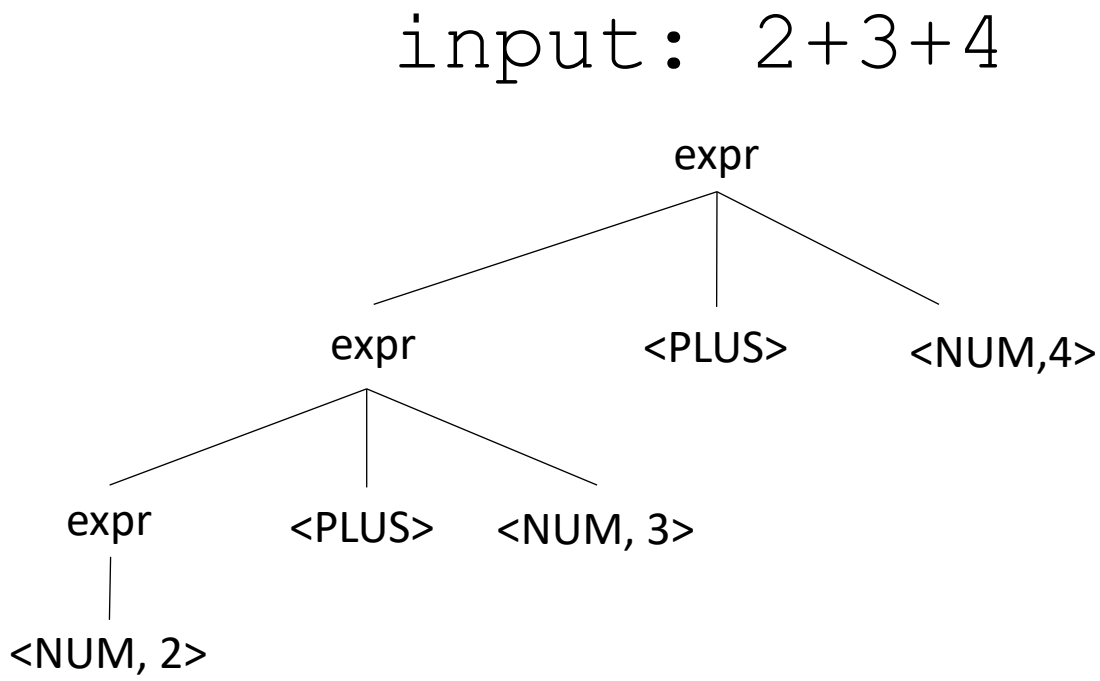
Operator	Name	Productions
+	expr	: expr PLUS NUM | NUM



Should you have associativity when its not required?

Benefits?
Drawbacks?

Operator	Name	Productions
+	expr	: expr PLUS NUM | NUM



Good design principle to avoid ambiguous grammars, even when strictly not required too.

Helps with debugging, etc. etc.

Many tools will warn if it detects ambiguity

Let’s make a richer grammar

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

Operator	Name	Productions

Tokens:

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

Let’s make a richer grammar

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

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

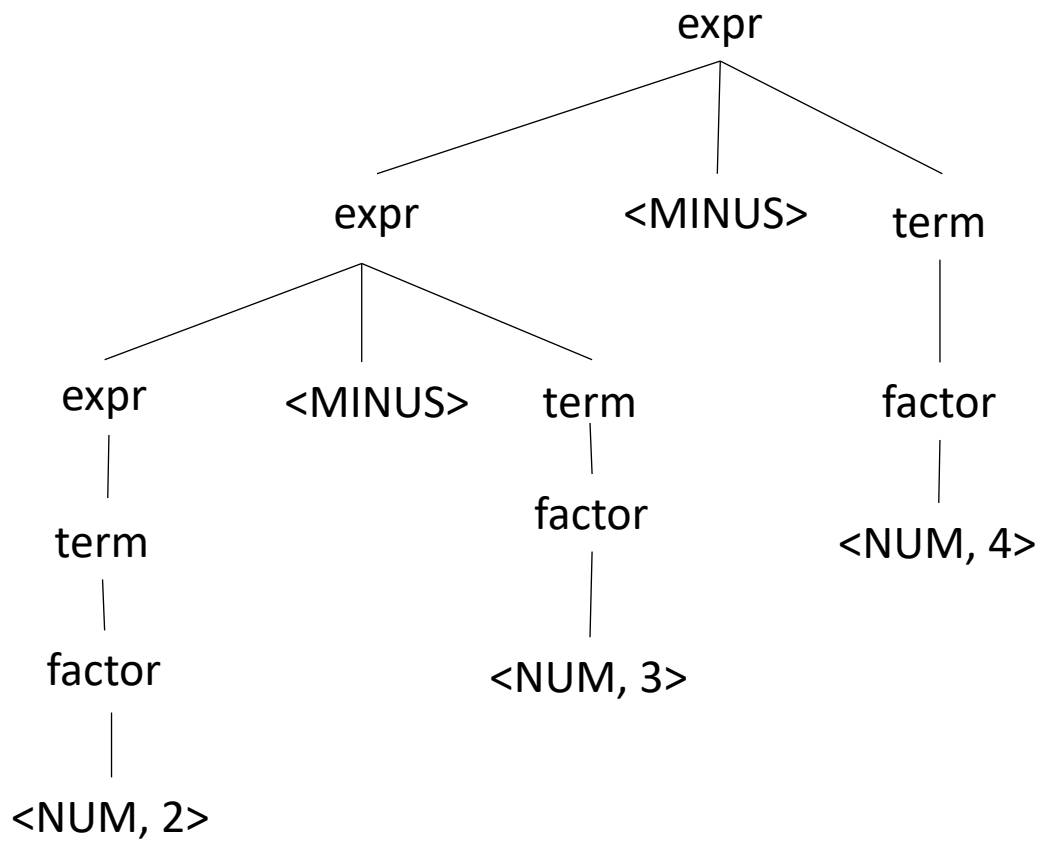
Tokens:

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

Let’s make a richer grammar

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

`input: 2-3-4`



What do these look like in real-world languages?

- C++ : https://en.cppreference.com/w/cpp/language/operator_precedence

- Python: https://docs.python.org/3/reference/expressions.html#operator-precedence

Production rules in a compiler

- Great to check if a string is grammatically correct
- But can the production rules actually help us with compilation??

Production actions

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

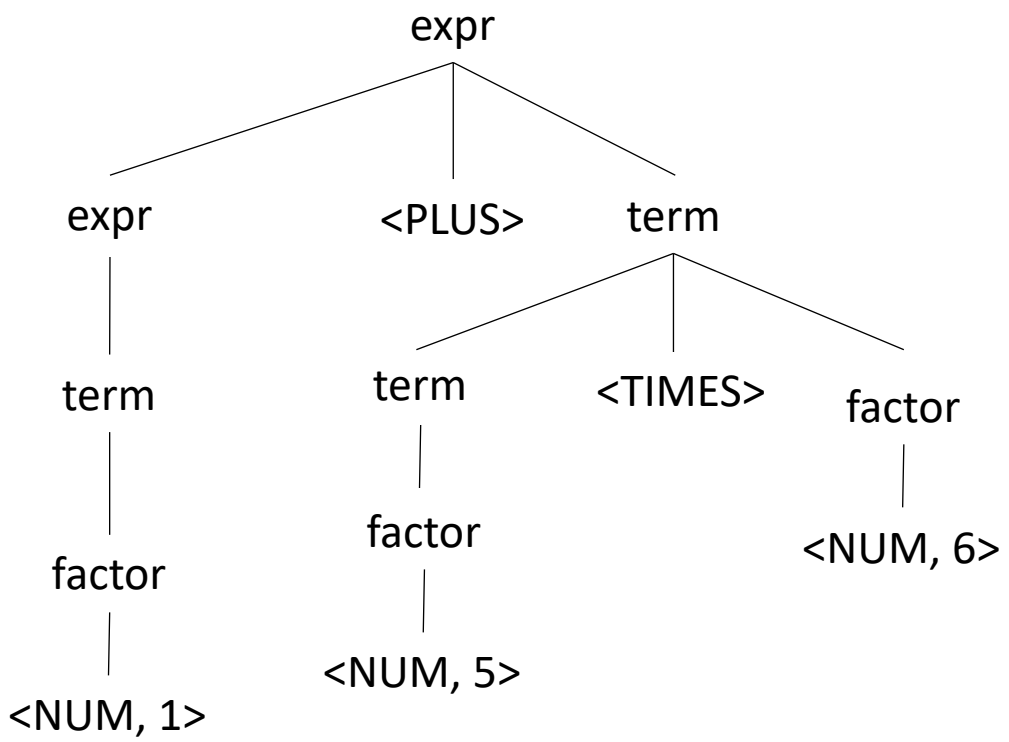
Production actions

Example: executing a mathematical expression during parsing

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

Operator	Name	Productions	Actions
+,-	expr	: expr PLUS term | expr MINUS term | term	{ } { } { }
*,/	term	: term TIMES factor : term DIV factor | factor	{ } { } { }
()	factor	: LPAR expr RPAR | NUM	{ } { }

`input: 1+5*6`



Production actions

Example: executing a mathematical expression during parsing

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

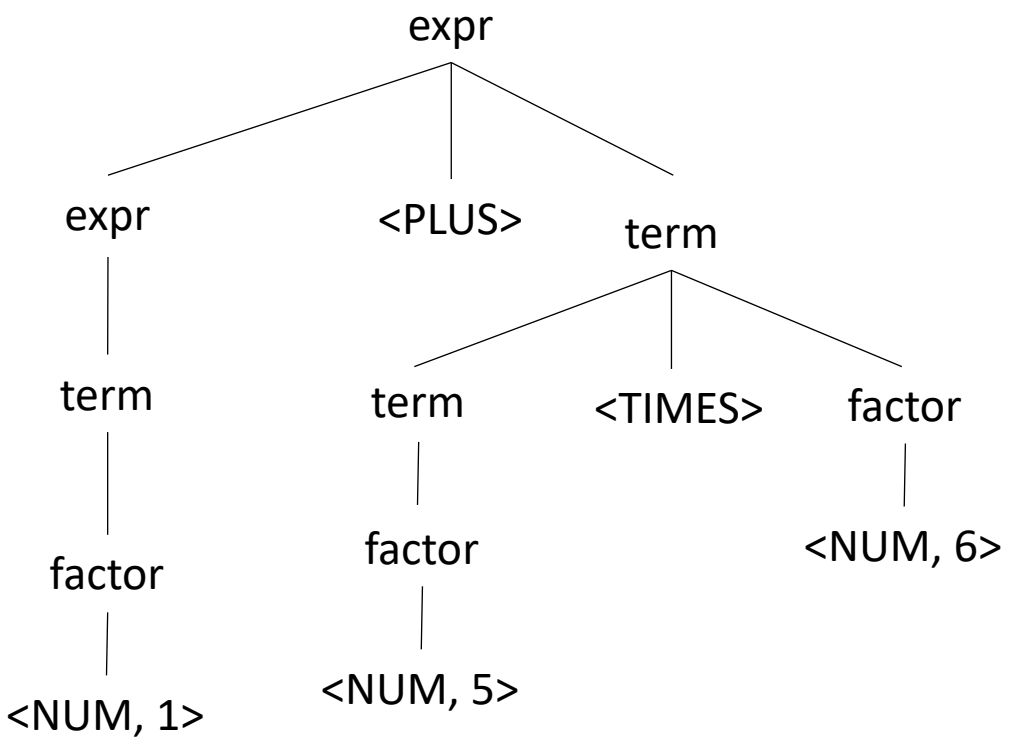
Operator	Name	Productions	Actions

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

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

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

`input: 1+5*6`



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

Next week

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

If time

Parsing REs

Let’s try it for regular expressions, {| . * ()} (where . is concat)

Operator	Name	Productions

Parsing REs

Let’s try it for regular expressions, {| . * ()} (where . is concat)

Operator	Name	Productions
|		
.		
*		
()		

Parsing REs

Let’s try it for regular expressions, {| . * ()} (where . is concat)

Operator	Name	Productions
|	union	
.	concat	
*	starred	
()	unit	

Parsing REs

Let’s try it for regular expressions, {| . * ()} (where . is concat)

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

Parsing REs

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

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

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

Parsing REs

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

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

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