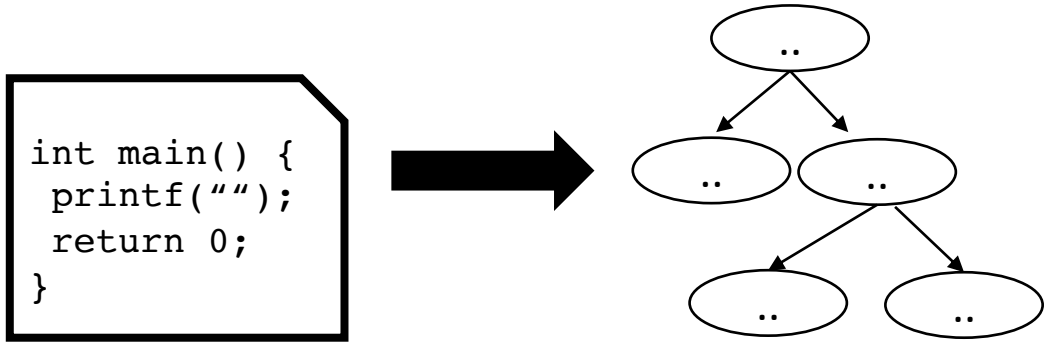
CSE211: Compiler Design

Sept. 29, 2022



- **Topic**: Parsing overview 2 (production rules)

- **Questions**:
  - *What are the limitations of tokens for parsing?*
  - *What is a context free grammar? Is it more or less powerful than a regular expression?*

Logistics

- Everyone should be on canvas
  - No one should be on a waitlist
- Everyone should be on the class Piazza
  - according to my math, we have about 7 people missing
  - please sign up
- Please make sure to record attendance for today!

Logistics

- Assignment 1 will be released next Tuesday by the end of the day
  - You will have 2 weeks to do it

- Two parts:
  - A very simple interpreter for a very simple language
  - A regular expression matcher using parsing with derivatives

- We will use PLY as our parser generator
  - If you want to use something different, e.g. Antlr, lex, yacc, let me me know!

Logistics

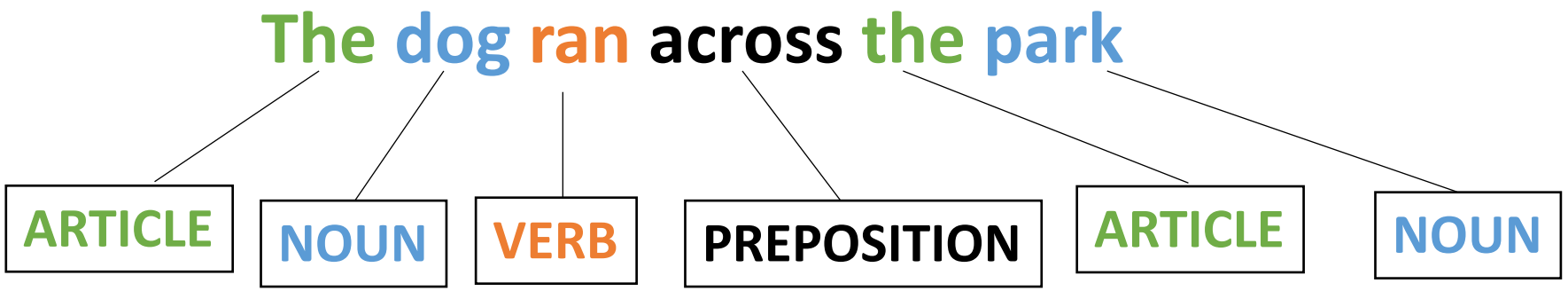
- First office hours will be tomorrow 3-5 pm
- Sign up sheet will be released at noon tomorrow
  - Look for a canvas announcement

Review

- What is a scanner?

Scanner

- splits an input into tokens (e.g. parts of speech)



Scanner

My Old Computer Crashed

Scanner

```
[(ARTICLE, “my”) (ADJECTIVE, “old”) (NOUN, “Computer”) (VERB, “Crashed”)]
```

Splits an input sentence it into lexemes

Scanner

(5 + 4) * 8

Scanner

[[(LPAR, “(”) (NUM, “5”) (PLUS, “+”) (NUM, “4”) (RPAR, “)”) (TIMES, “*”) (NUM, “

Splits an input sentence it into lexemes

Scanner

(5 + 4) * 8

What if we wanted to tokenize this arithmetic sentence?

Scanner

(5 + 4) * 8

What if we wanted to tokenize this arithmetic sentence?

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
  - What about using *?

Dealing with subset

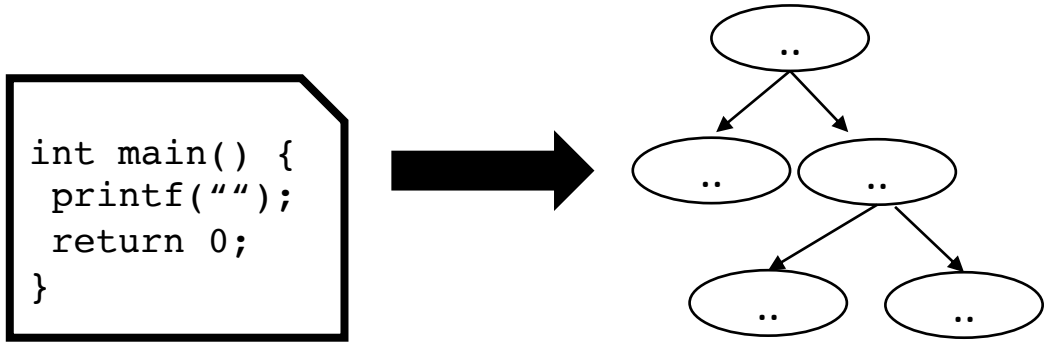
how to tokenize this input?

if

Tokens:
ID = “[a-z]+”
IF = “if”

CSE211: Compiler Design

Sept. 29, 2022



- **Topic**: Parsing overview 2 (production rules)

- **Questions**:
  - *What are the limitations of tokens for parsing?*
  - *What is a context free grammar? Is it more or less powerful than a regular expression?*

Define a full language using tokens?

- What about a mathematical sentence (expression)?

limited to non-negative integers and just using + and *

Define a full language using tokens?

limited to non-negative integers and just using + and *

- What about a mathematical sentence (expression)?

- First lets define tokens:

Define a full language using tokens?

limited to non-negative integers and just using + and *

- What about a mathematical sentence (expression)?

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

Define a full language using tokens?

limited to non-negative integers and just using + and *

- What about a mathematical sentence (expression)?

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

- What should our language look like?

Define a full language using tokens?

- Where are we going to run into issues?

Matching () using a regular expression

- there is a formal proof available that regex CANNOT match ()'s: pumping lemma

- Informal argument:
  - Try matching $(^n)^n$ using Kleene star
  - Impossible!

- We are going to need a more powerful language description framework!

Matching () using a regular expression

- there is a formal proof available that regex CANNOT match ()'s: pumping lemma

- Informal argument:
  - Try matching $(^n)^n$ using Kleene star
  - Impossible!

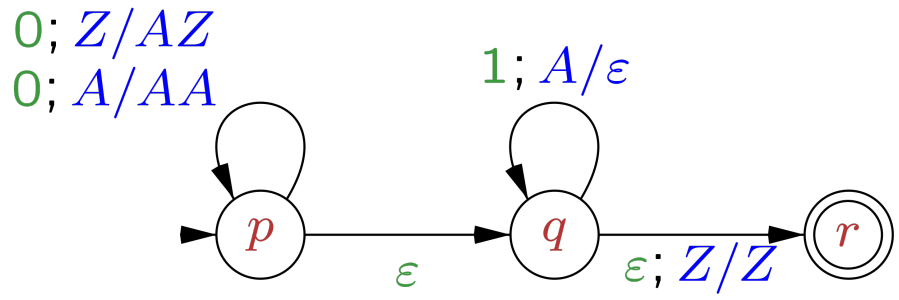
https://stackoverflow.com/questions/1732348/regex-match-open-tags-except-xhtml-self-contained-tags

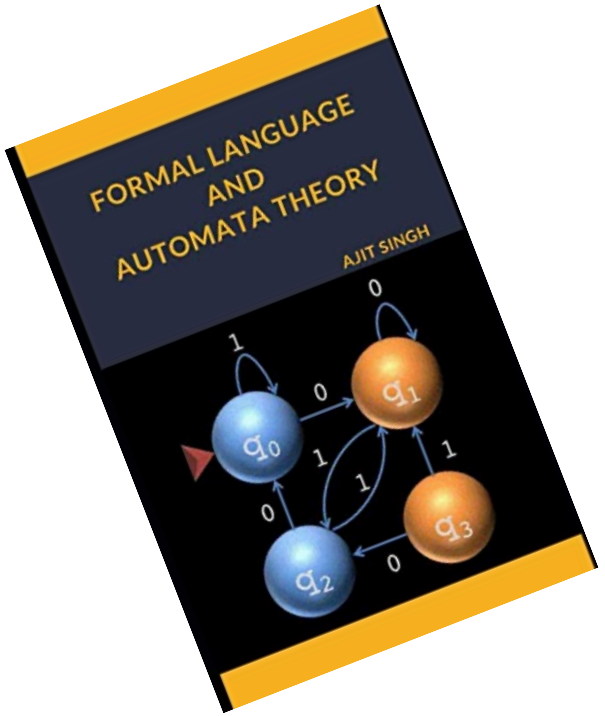
(previously) 2nd most upvoted post on stackoverflow

- We are going to need a more powerful language description framework!

Context Free Grammars

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





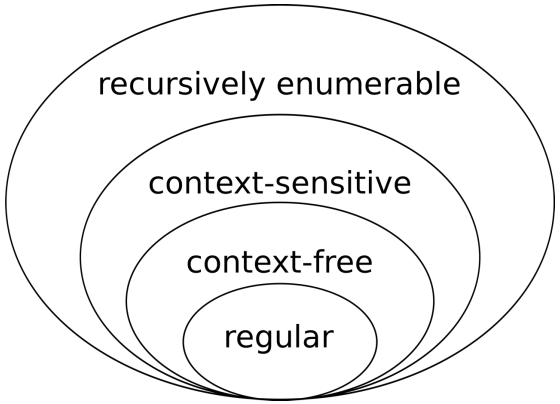


Image Credit:
By Jochgem - Own work, CC BY-SA 3.0,
https://commons.wikimedia.org/w/index.php?curid=5036988

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>

Let’s go back to mathematical sentences (expressions)

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

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 = ‘\(‘
  - RPAREN = ‘\)’

What other syntax like () are used in programming languages?

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

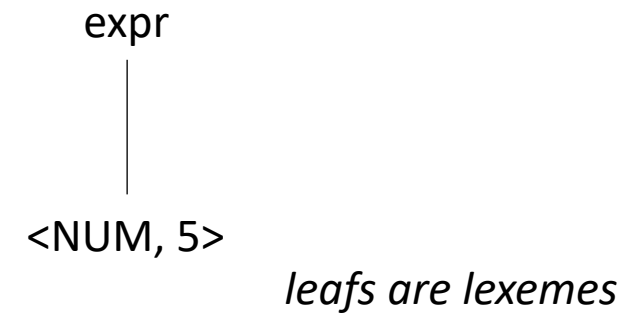
root of the tree is the entry production

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
```

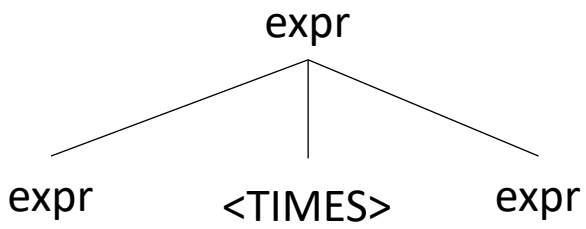
expr

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
```

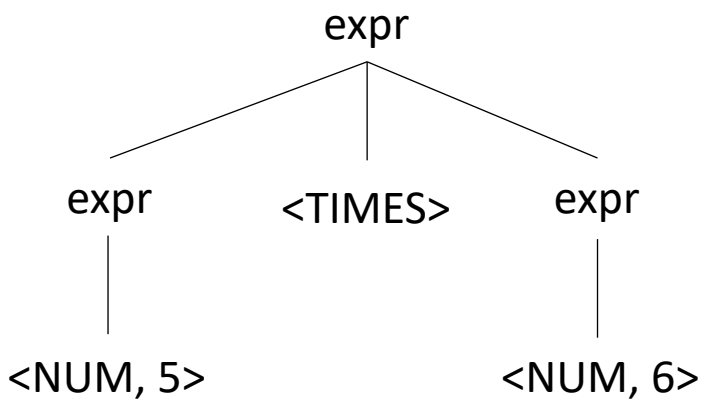


Parse trees

- 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.

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

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

What happens in an error?

expr

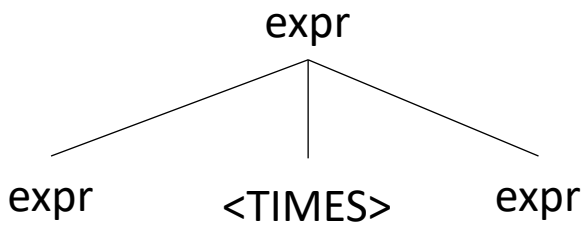
Parse trees

- 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?



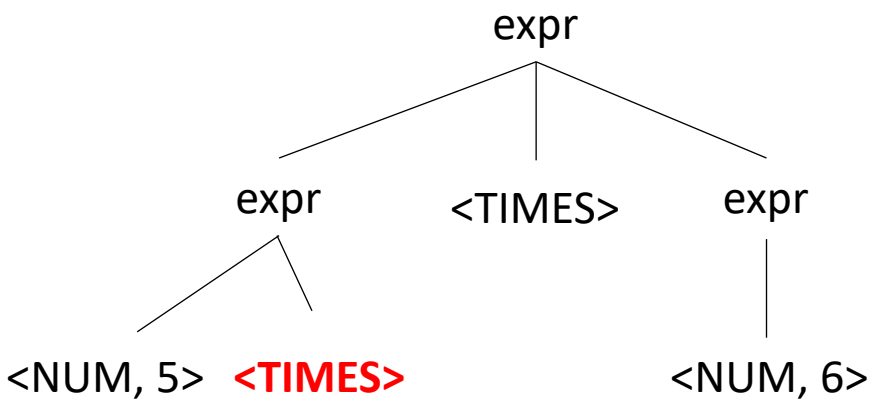
Parse trees

- 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

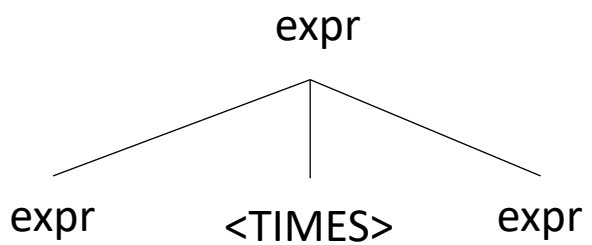
expr

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
```

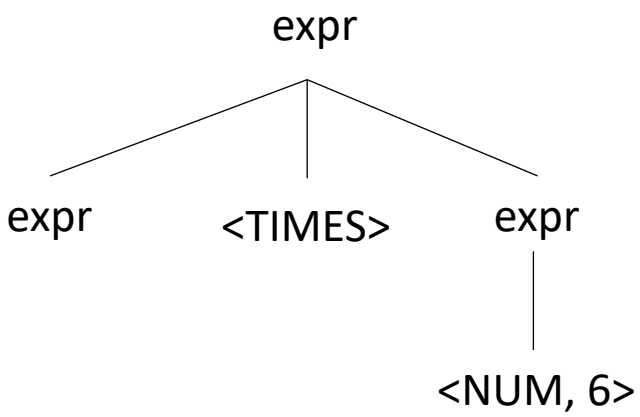


Parse trees

- 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
```

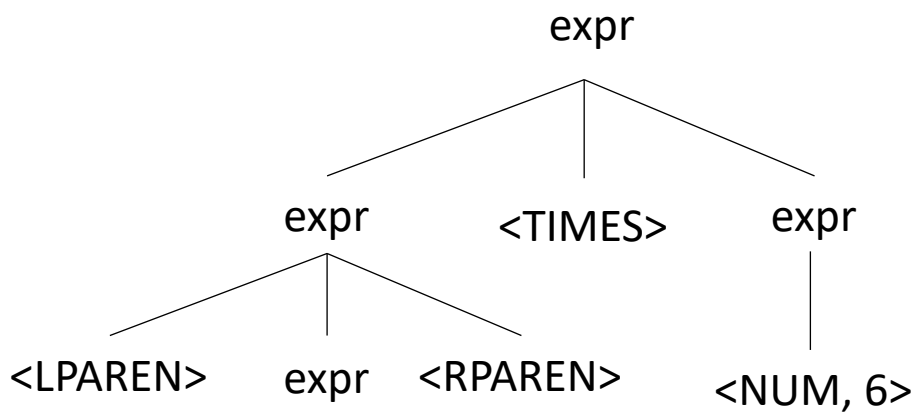


Parse trees

- 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
```

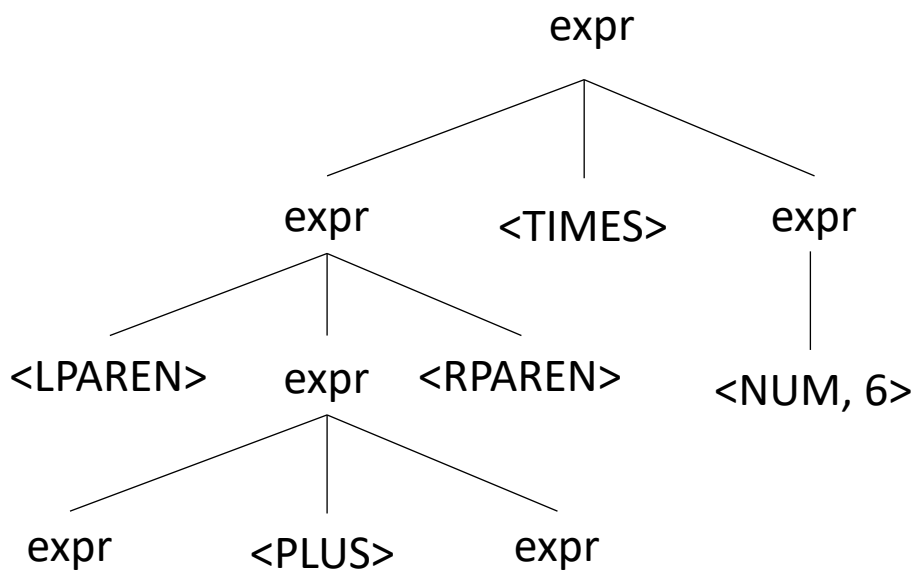


Parse trees

- 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
```

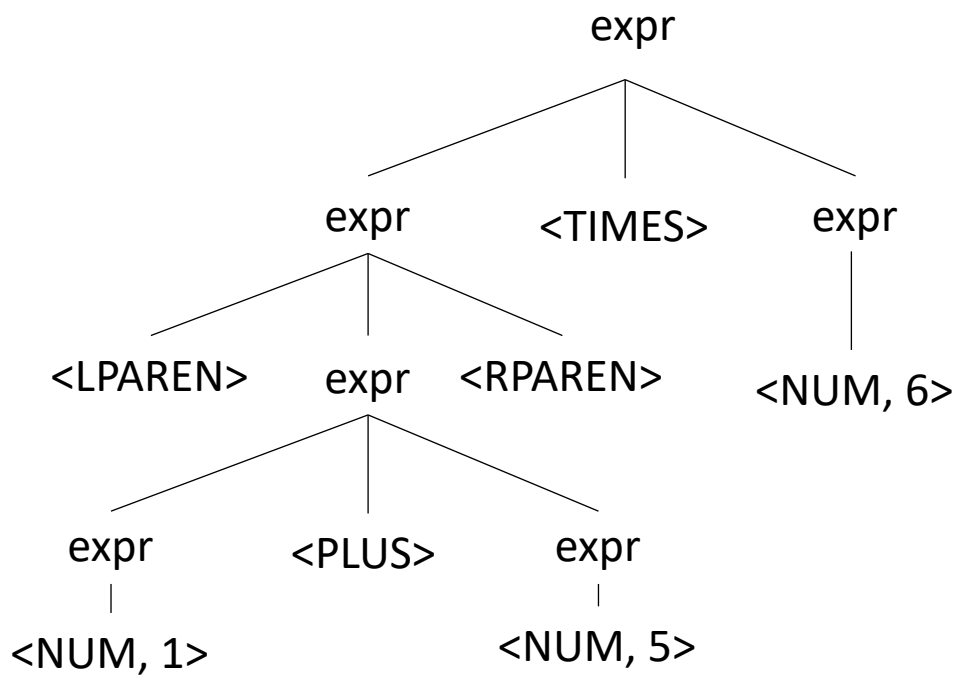


Parse trees

- 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
```

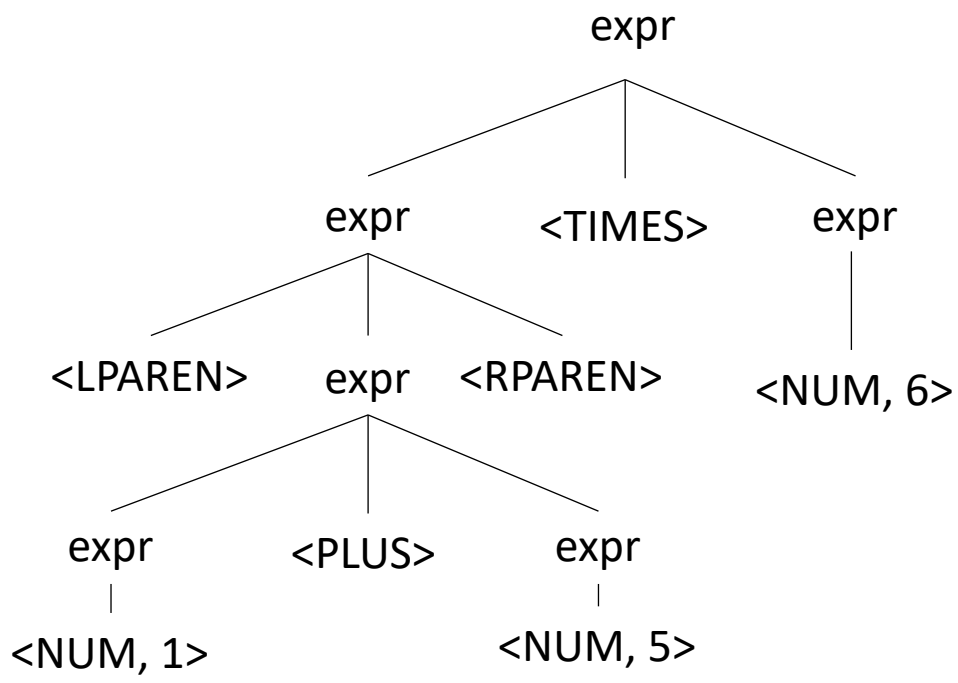


Parse trees

- 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

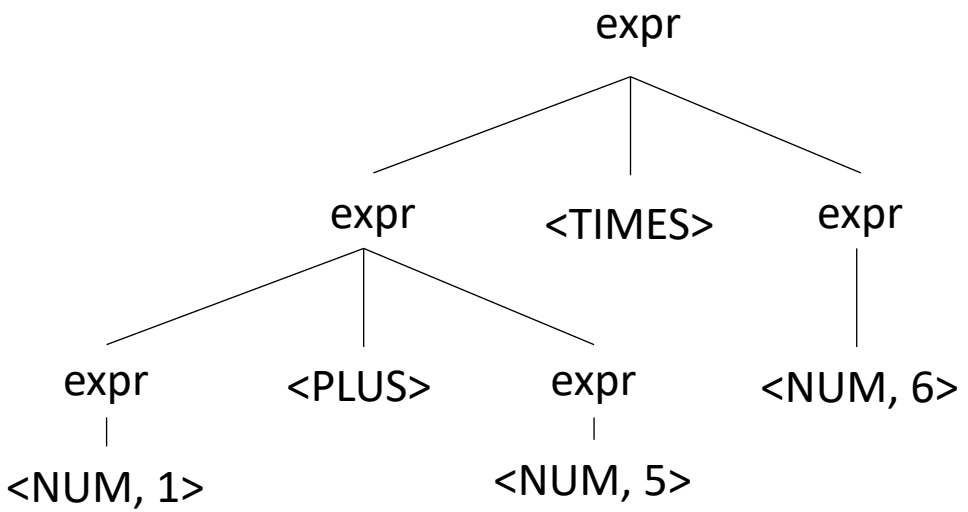
| expr TIMES expr

| LPAREN expr RPAREN

Parse trees

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

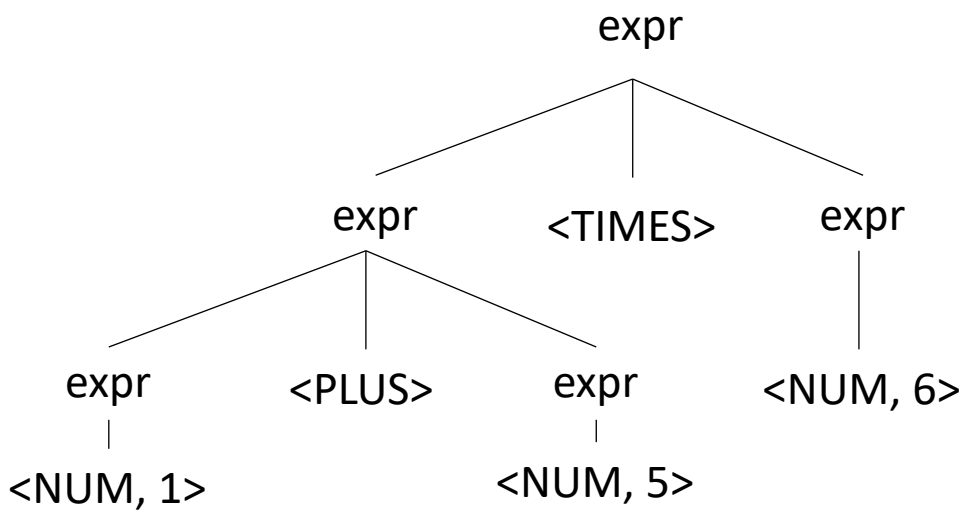
```
expr : NUM
     | expr PLUS expr
     | expr TIMES expr
     | LPAREN expr RPAREN
```



Parse trees

- `input: 1 + 5 * 6`

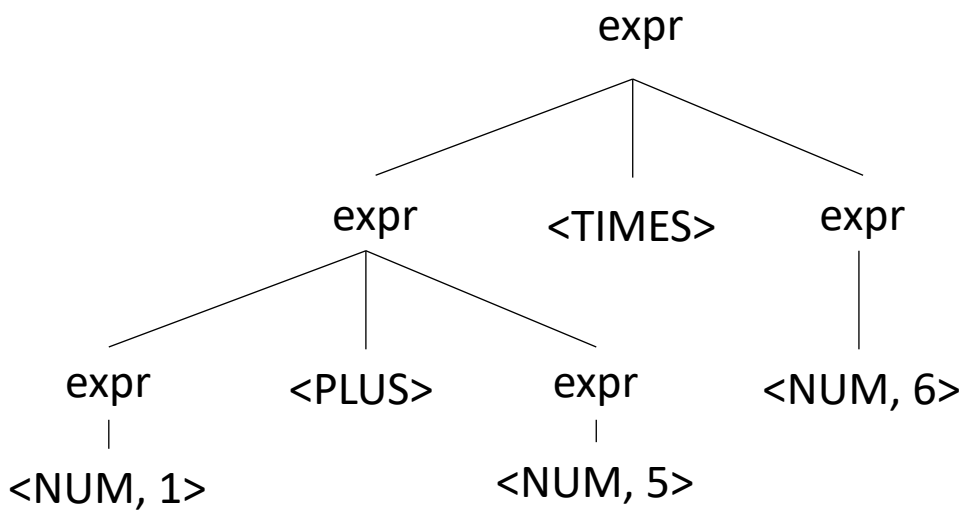
```
expr : NUM
     | expr PLUS expr
     | expr TIMES expr
     | LPAREN expr RPAREN
```

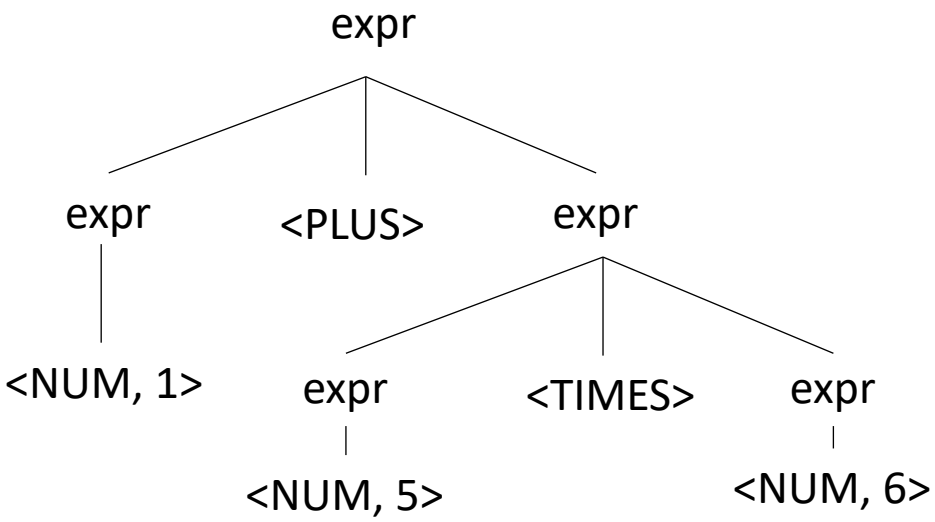


Parse trees

- `input: 1 + 5 * 6`

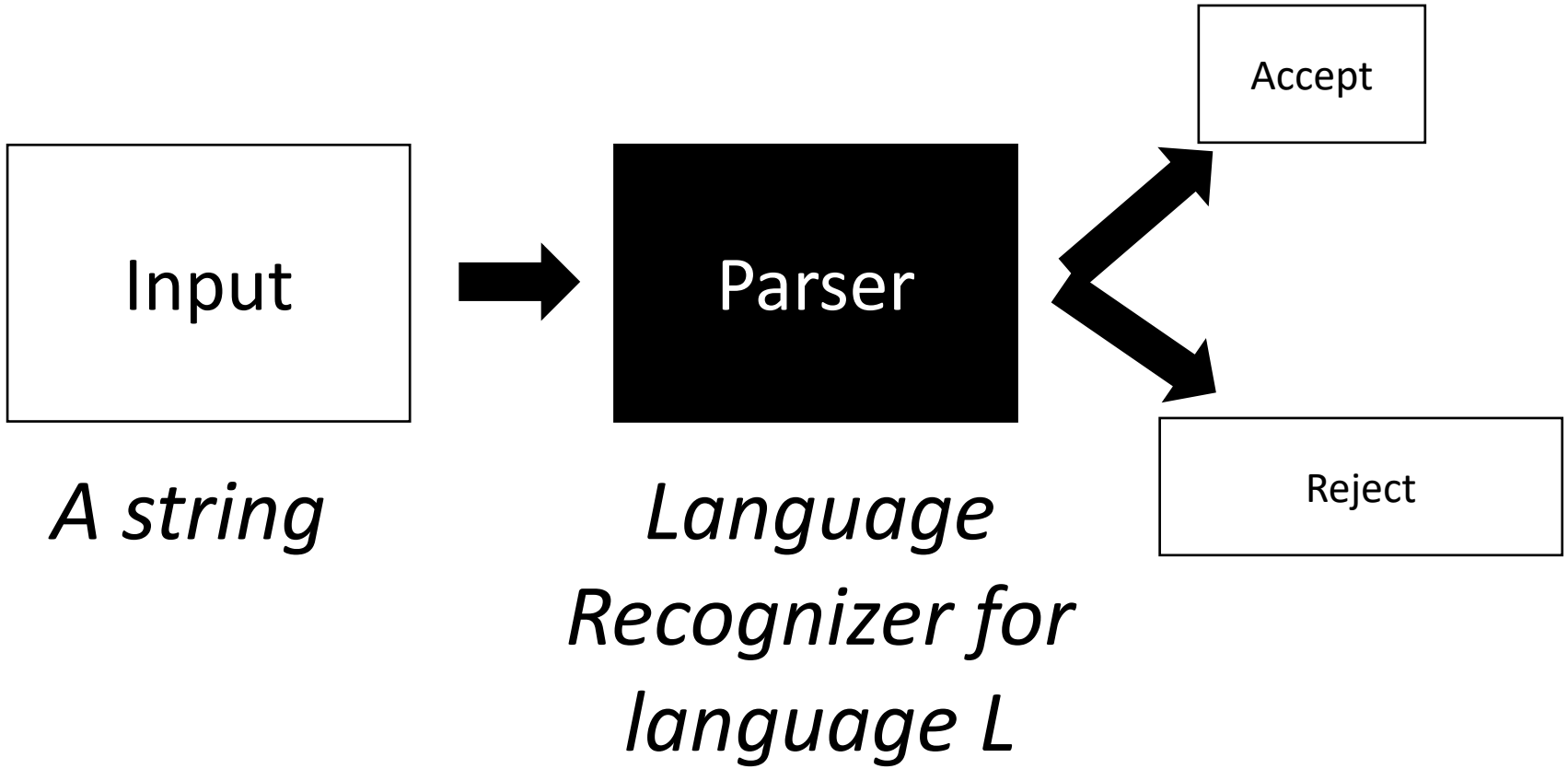
```
expr : NUM
     | expr PLUS expr
     | expr TIMES expr
     | LPAREN expr RPAREN
```





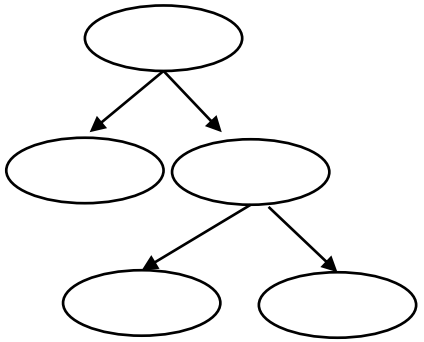
Ambiguous grammars

- What's the issue?

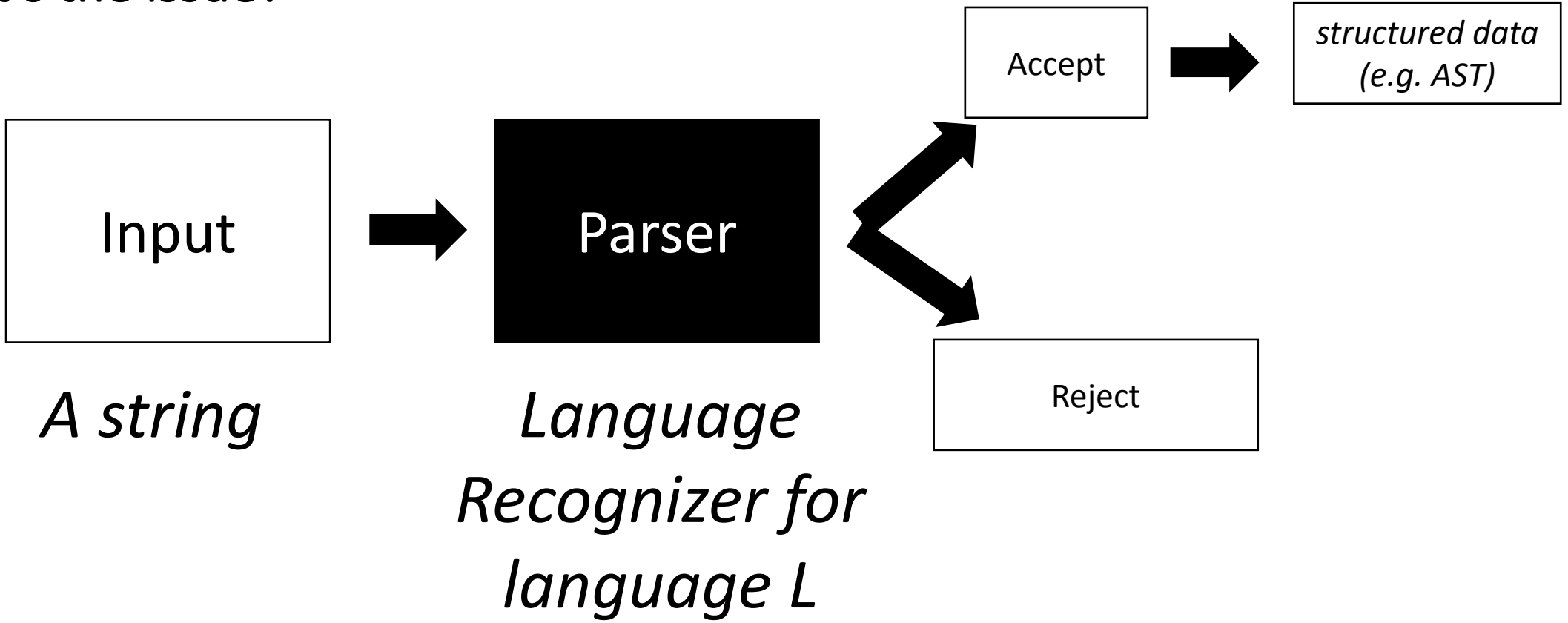


Ambiguous grammars

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

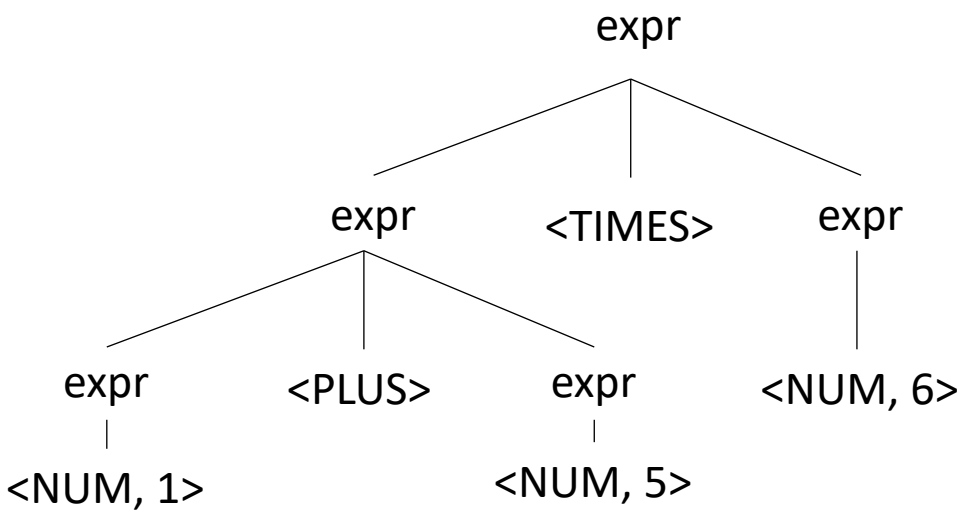
- Structural meaning defined to be a post-order traversal
  - Children return values to their parent
  - Nodes are only evaluated once all their children have been evaluated
  - Evaluated from left to right
  - Also called “Natural Order”

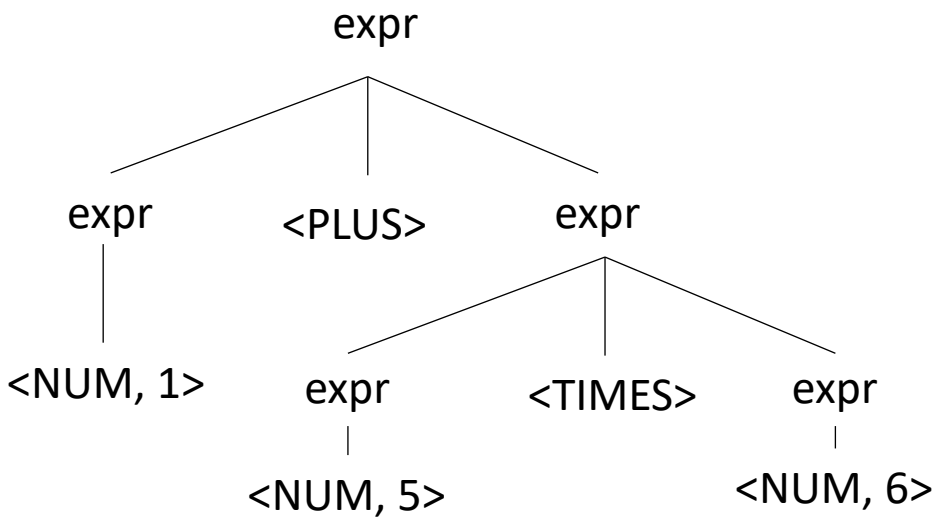
- Can also encode the order of operation

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
```

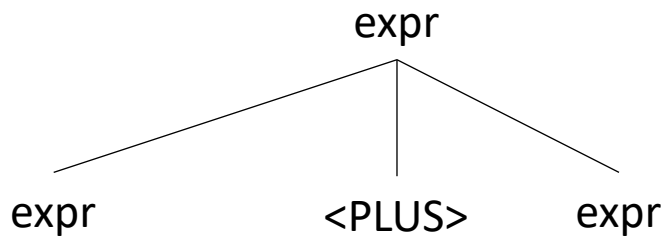
expr

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
```

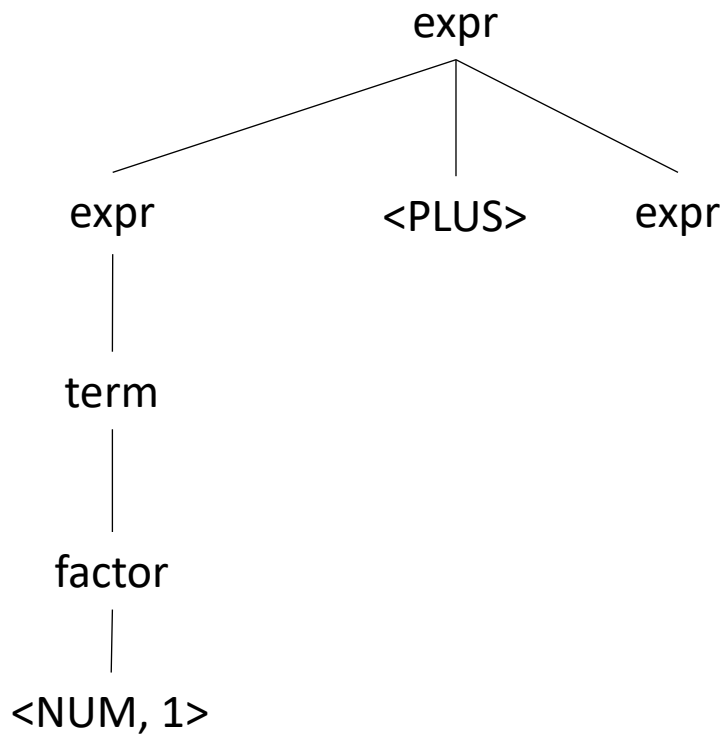
Operator	Name	Productions
+	expr	: expr PLUS expr | term
*	term	: term TIMES term | factor
()	factor	: LPAREN expr RPAREN | NUM



Now lets create a parse tree

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

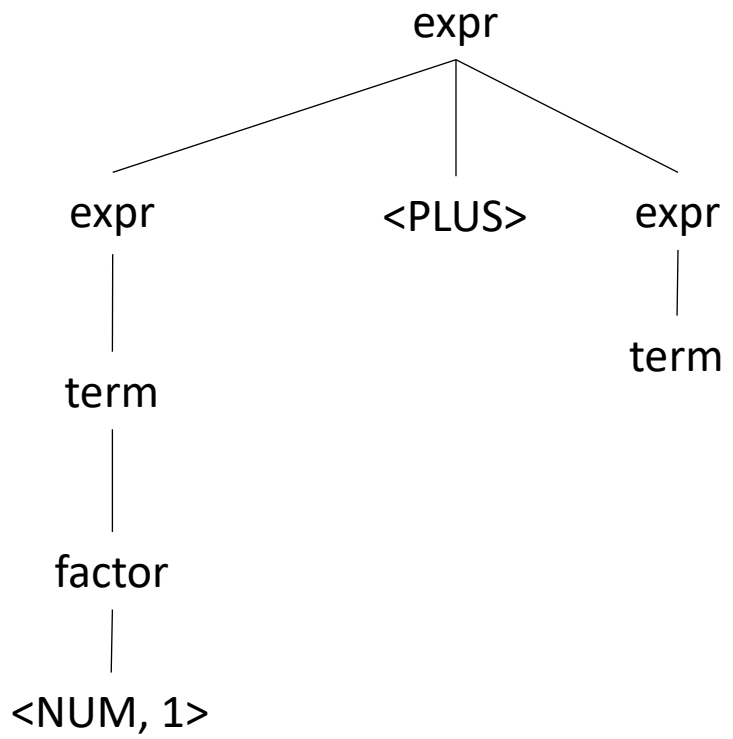
Operator	Name	Productions
+	expr	: expr PLUS expr | term
*	term	: term TIMES term | factor
()	factor	: LPAREN expr RPAREN | NUM



Now lets create a parse tree

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

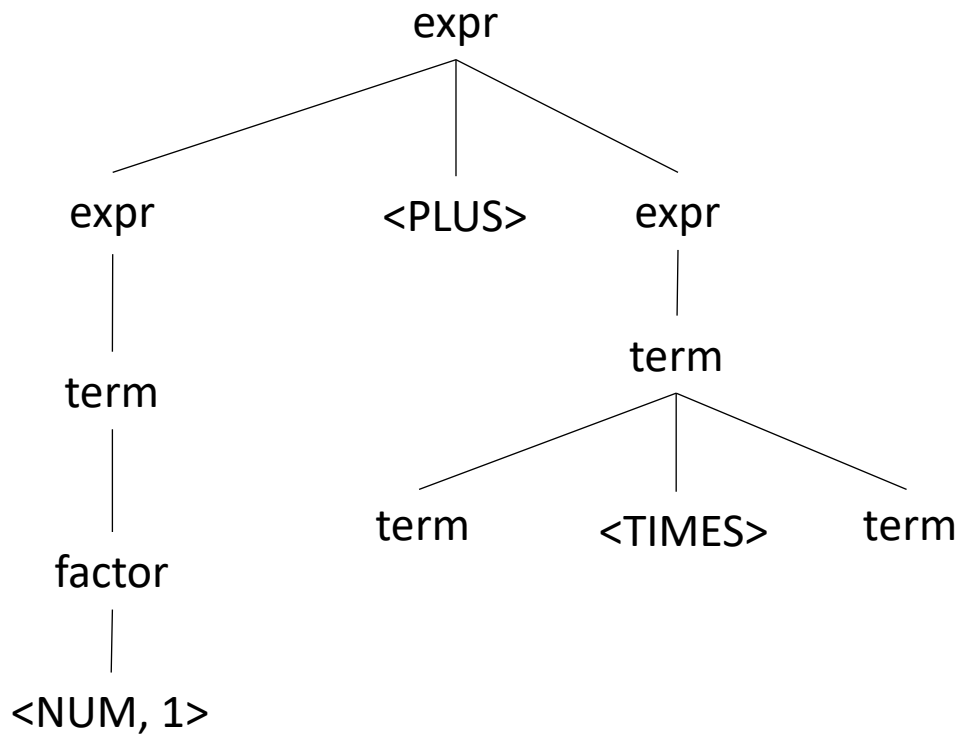
Operator	Name	Productions
+	expr	: expr PLUS expr | term
*	term	: term TIMES term | factor
()	factor	: LPAREN expr RPAREN | NUM



Now lets create a parse tree

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

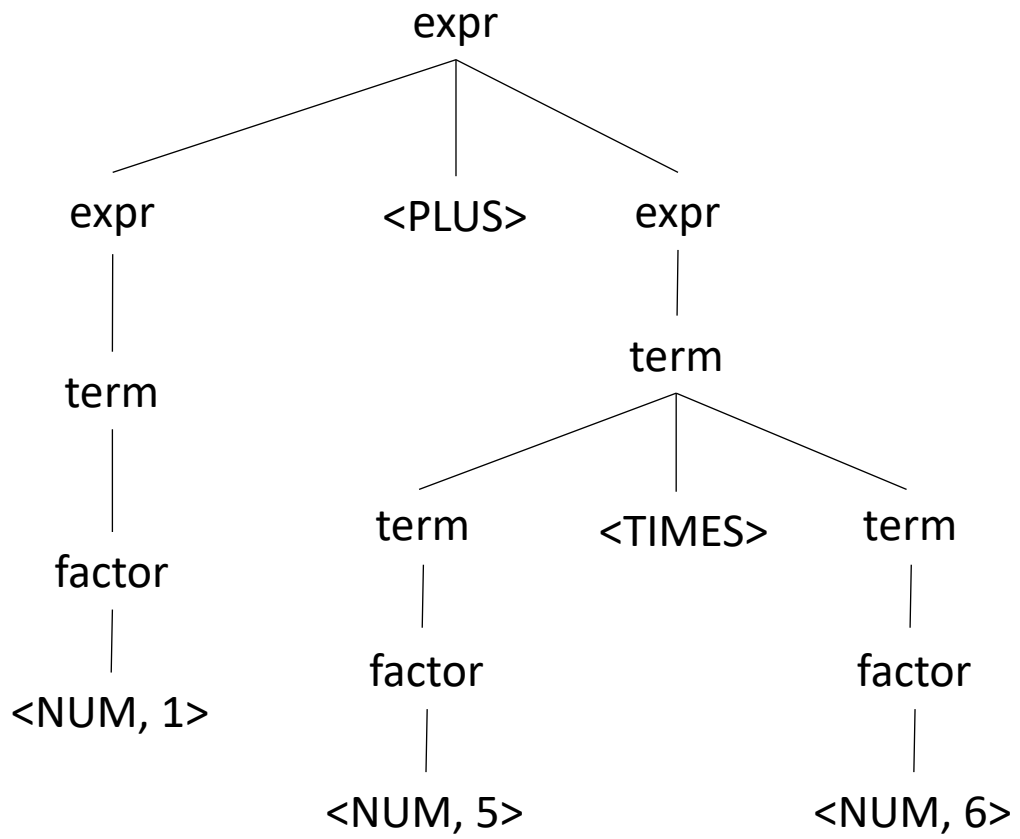
Operator	Name	Productions
+	expr	: expr PLUS expr | term
*	term	: term TIMES term | factor
()	factor	: LPAREN expr RPAREN | NUM



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



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

Parsing REs

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

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

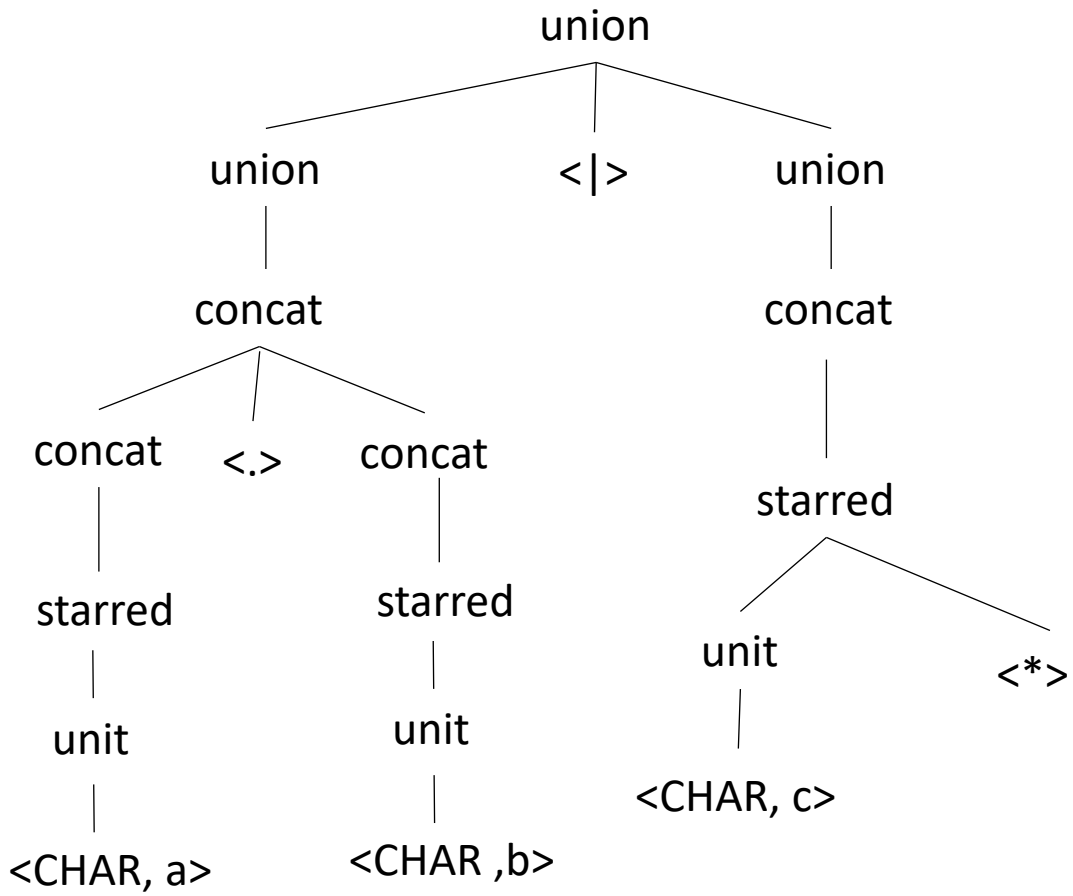
Operator	Name	Productions
|	union	: union PIPE union | concat
.	concat	: concat DOT concat | starred
*	starred	: starred STAR | unit
()	unit	: LPAREN union RPAREN | CHAR

Parsing REs

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

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

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



Let’s make some more parse trees

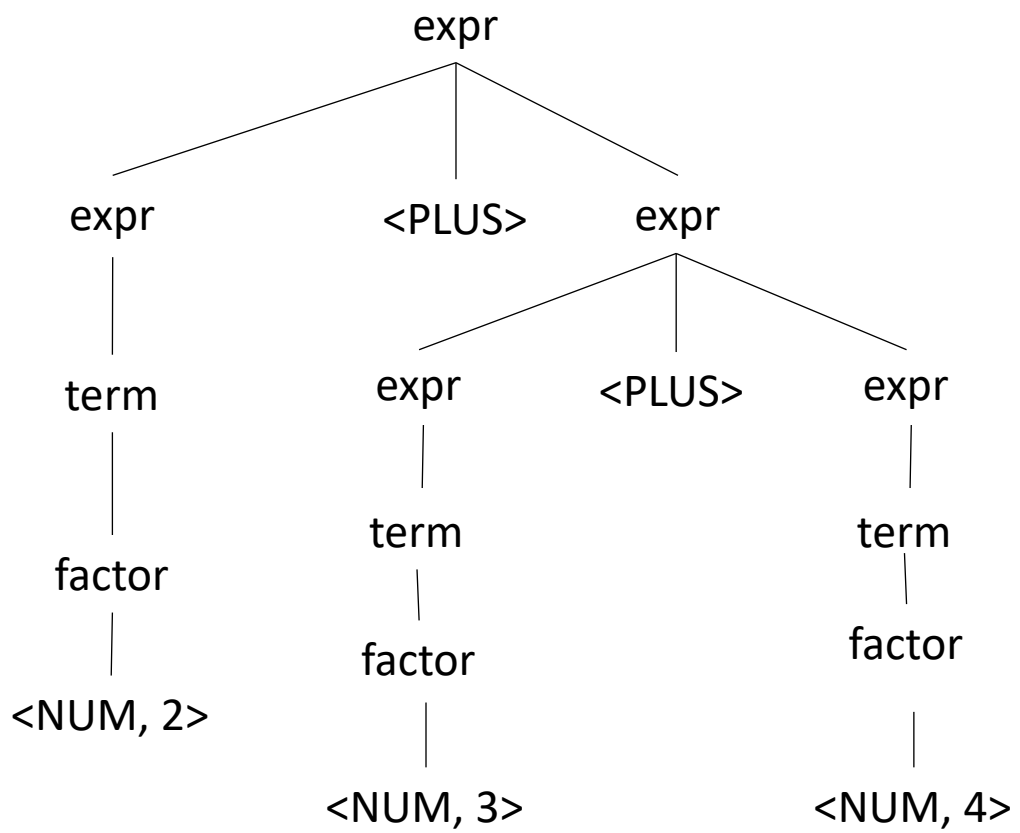
```
input: 2+3+4
```

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

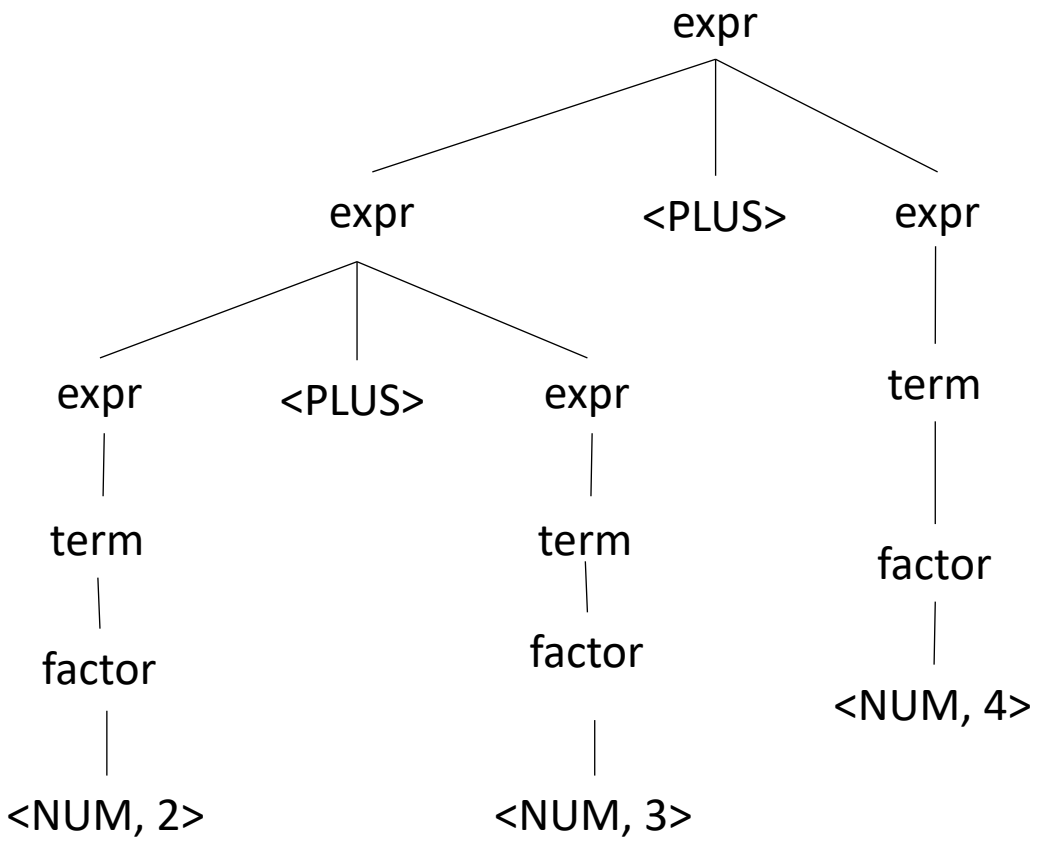
Operator	Name	Productions
+	expr	: expr PLUS expr | term
*	term	: term TIMES term | factor
()	factor	: LP expr RP | NUM

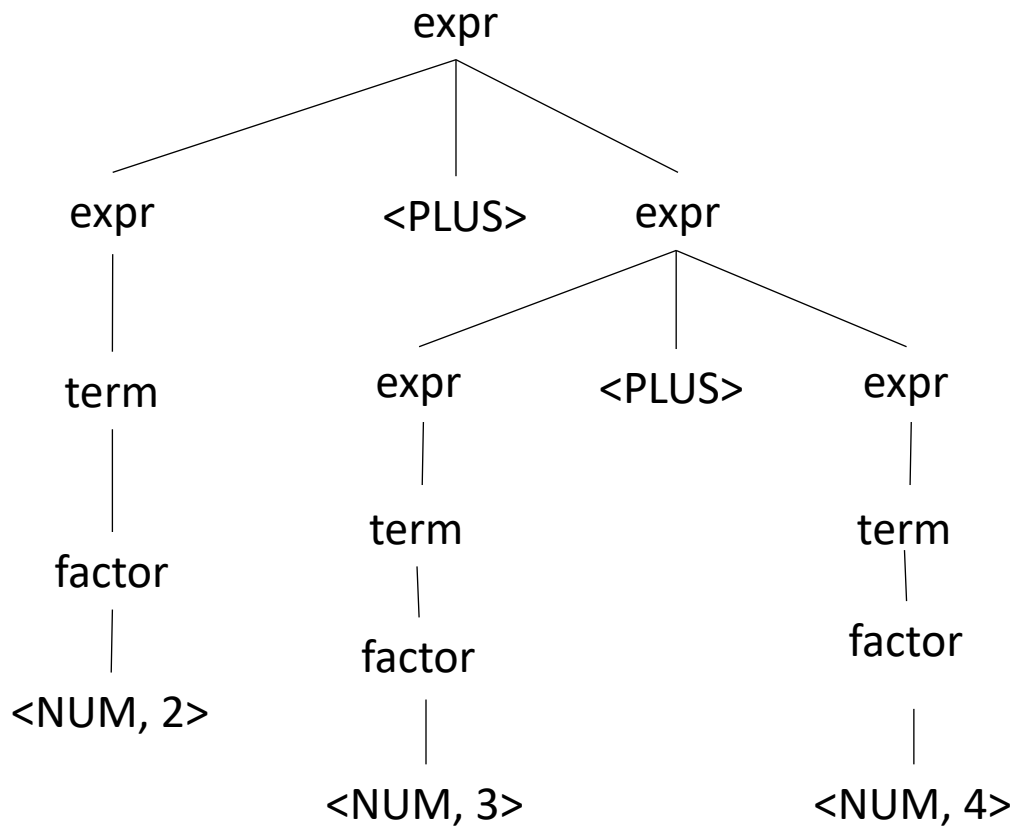
`input: 2+3+4`



This is ambiguous, is it an issue?

`input: 2+3+4`



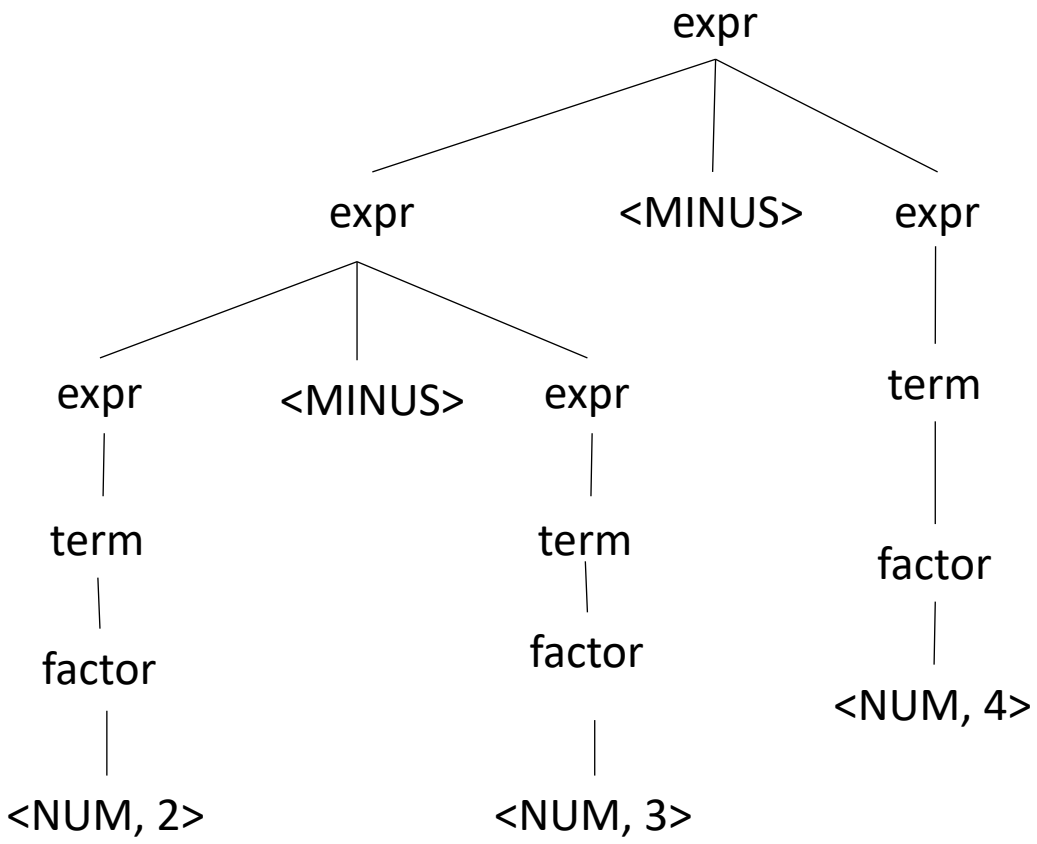


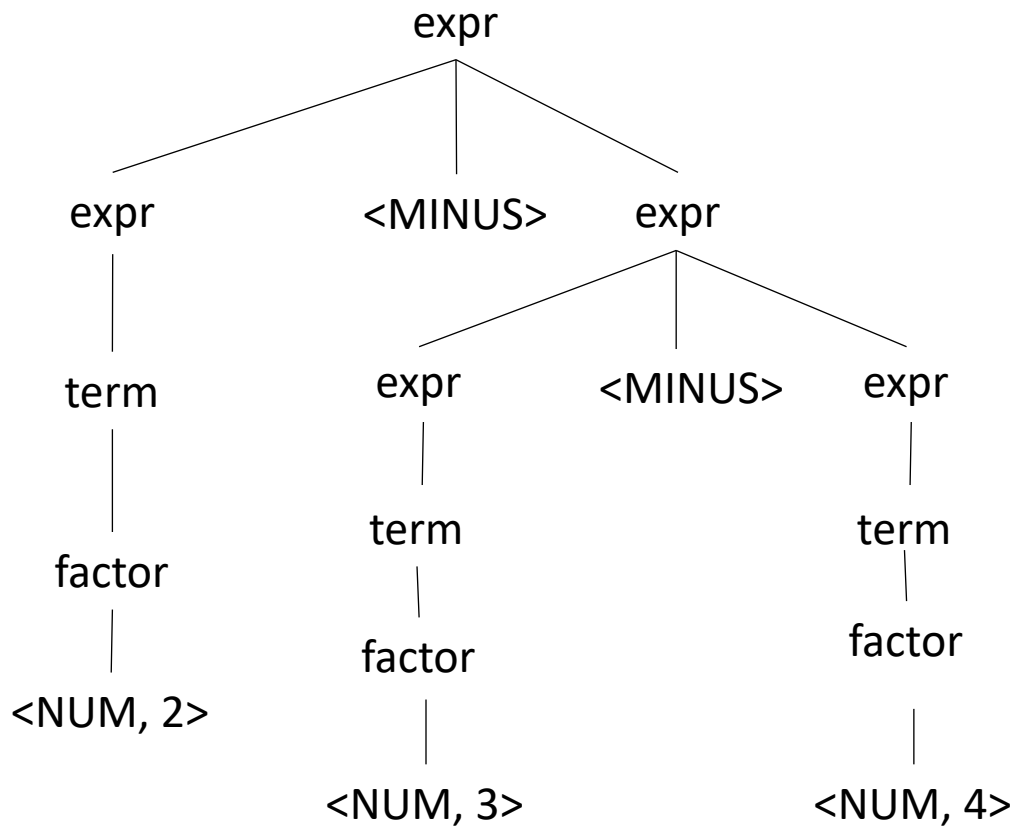
What about for a different operator?

```
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?

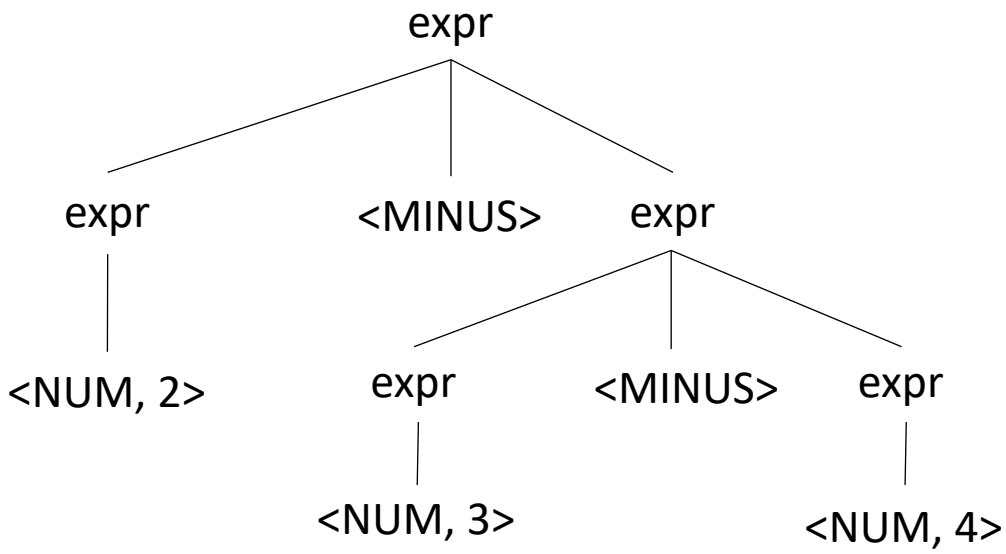
- Like precedence, some tools (e.g. YACC) allow associativity specification through keywords:
  - “+”: left, “^”: right

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

Associativity for a single operator

Operator	Name	Productions
-	expr	: expr MINUS expr | NUM

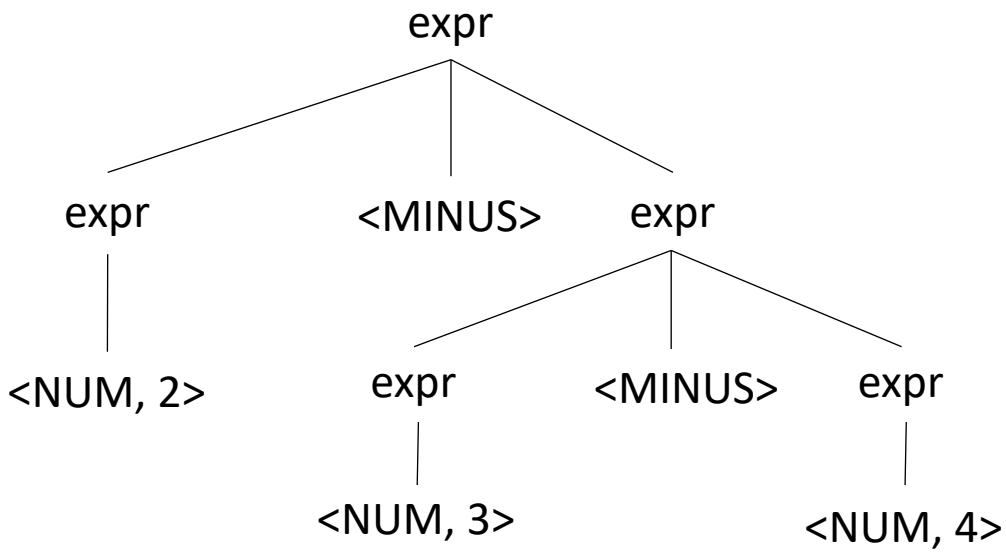
```
input: 2-3-4
```



Associativity for a single operator

Operator	Name	Productions
-	expr	: expr MINUS NUM | NUM

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

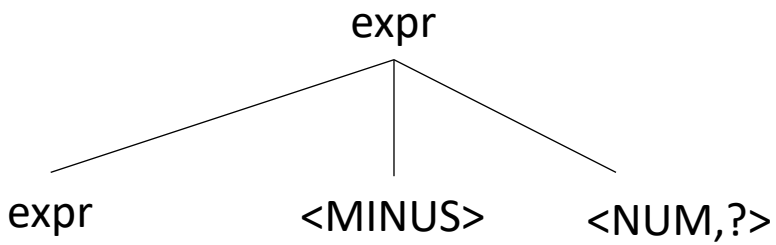


No longer allowed

Associativity for a single operator

Operator	Name	Productions
-	expr	: expr MINUS NUM | NUM

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

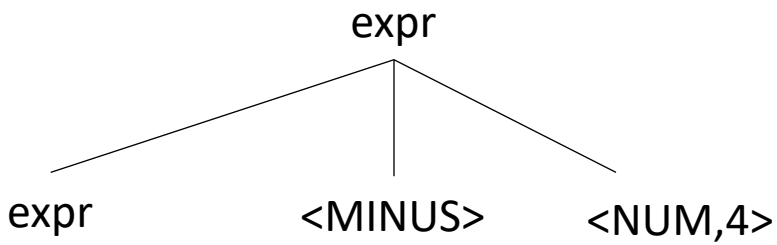


Lets start over

Associativity for a single operator

Operator	Name	Productions
-	expr	: expr MINUS NUM | NUM

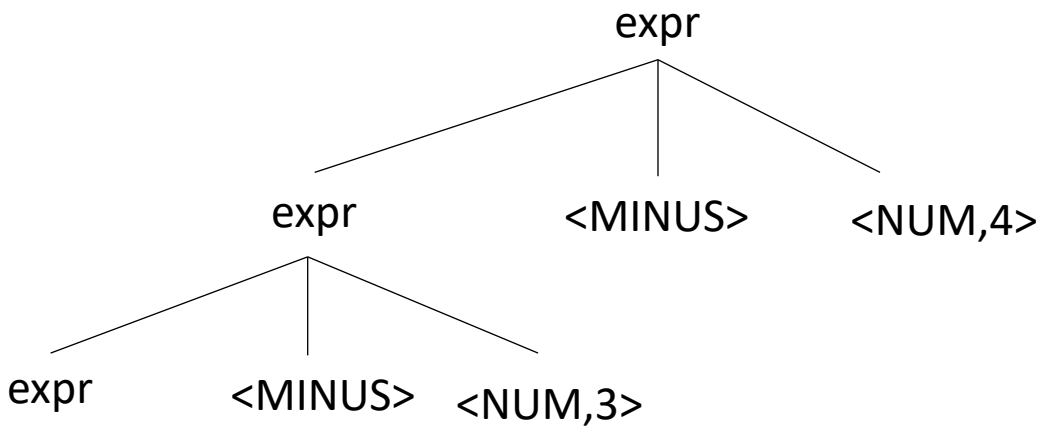
`input: 2-3-4`



Associativity for a single operator

Operator	Name	Productions
-	expr	: expr MINUS NUM | NUM

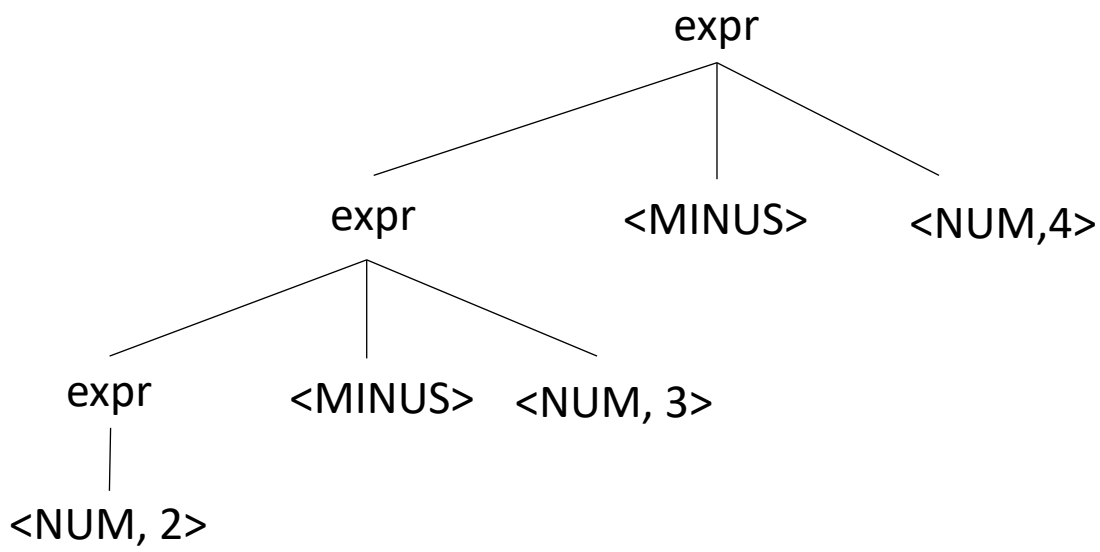
```
input: 2-3-4
```



Associativity for a single operator

Operator	Name	Productions
-	expr	: expr MINUS NUM | NUM

`input: 2-3-4`

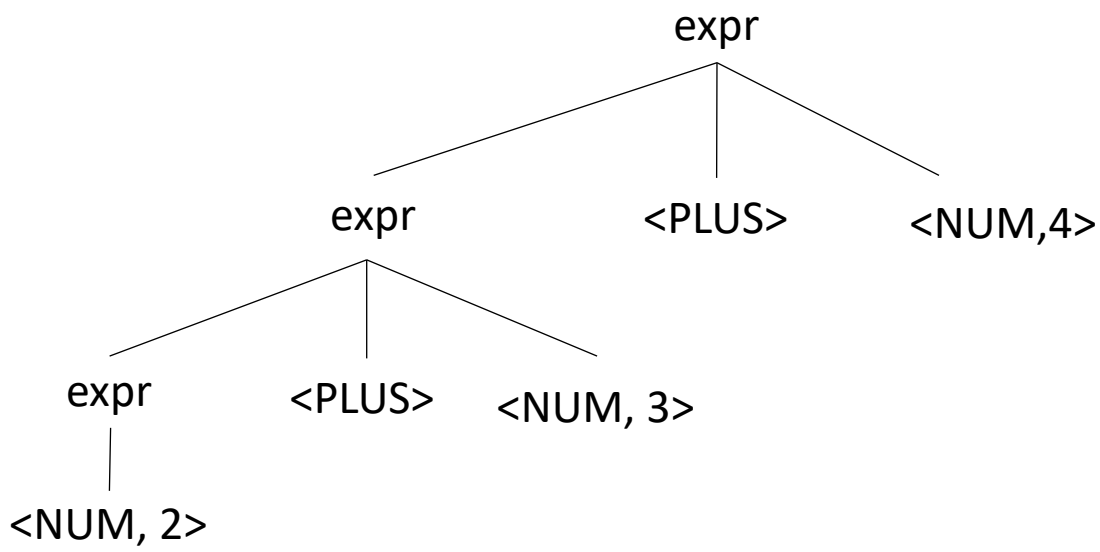


Should you have associativity when its not required?

Benefits?
Drawbacks?

Operator	Name	Productions
+	expr	: expr PLUS NUM | NUM

`input: 2+3+4`

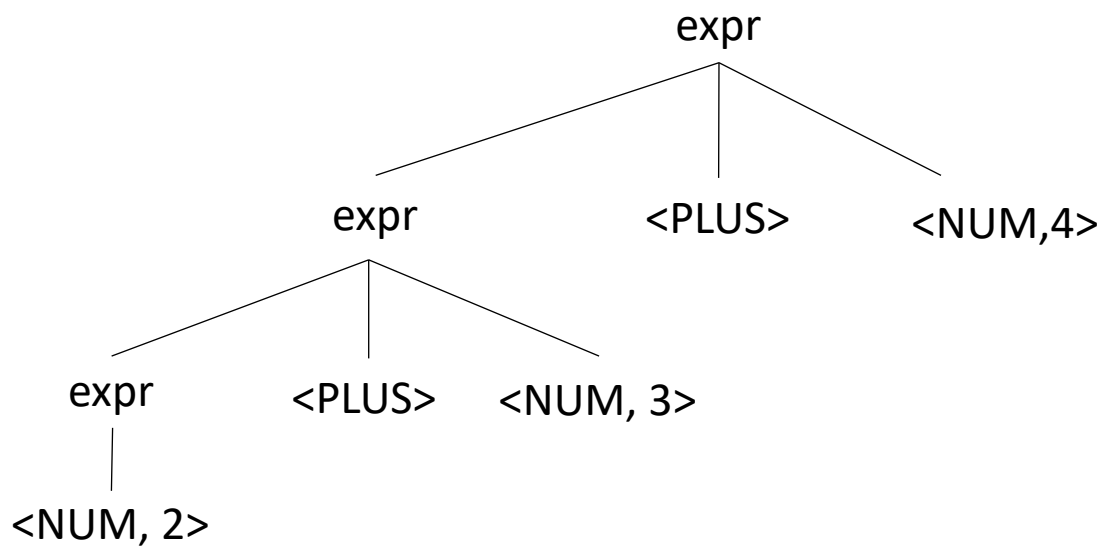


Should you have associativity when its not required?

Benefits?
Drawbacks?

Operator	Name	Productions
+	expr	: expr PLUS NUM | NUM

`input: 2+3+4`



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

Helps with debugging, etc. etc.

Many tools will warn if it detects ambiguity

Let’s make a richer 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

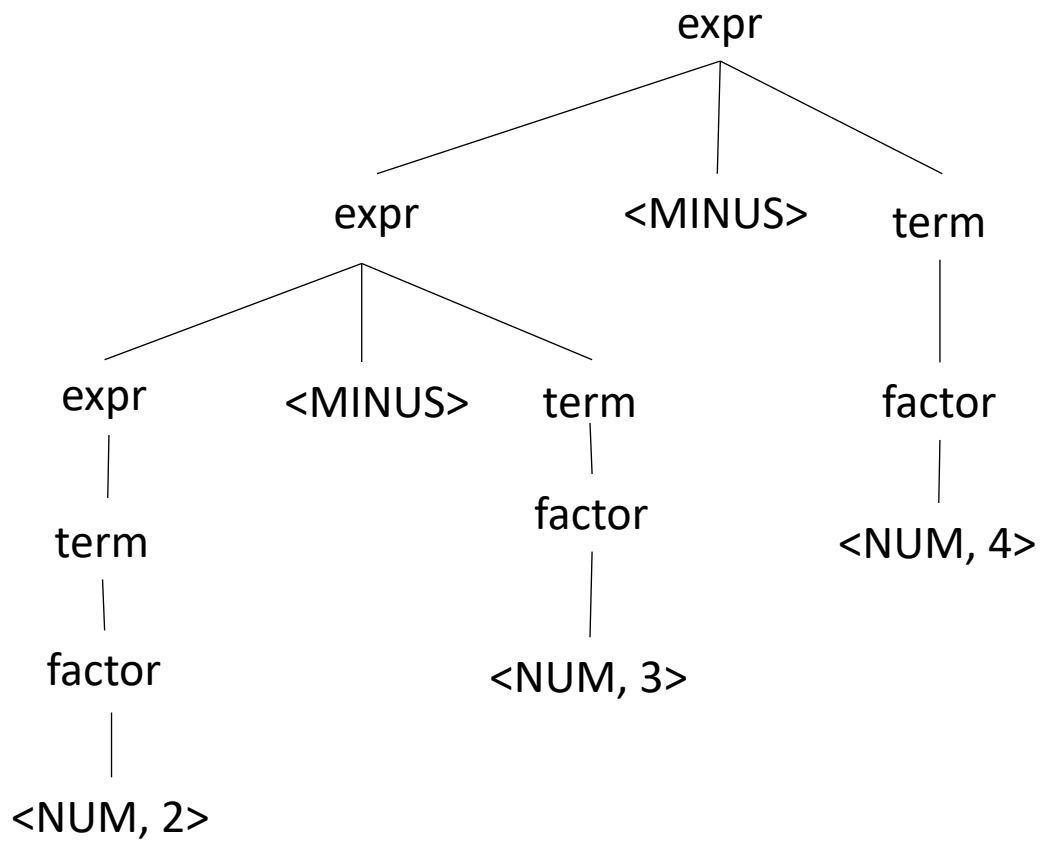
Tokens:

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

  https://www.geeksforgeeks.org/precedence-and-associativity-of-operators-in-python/

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)

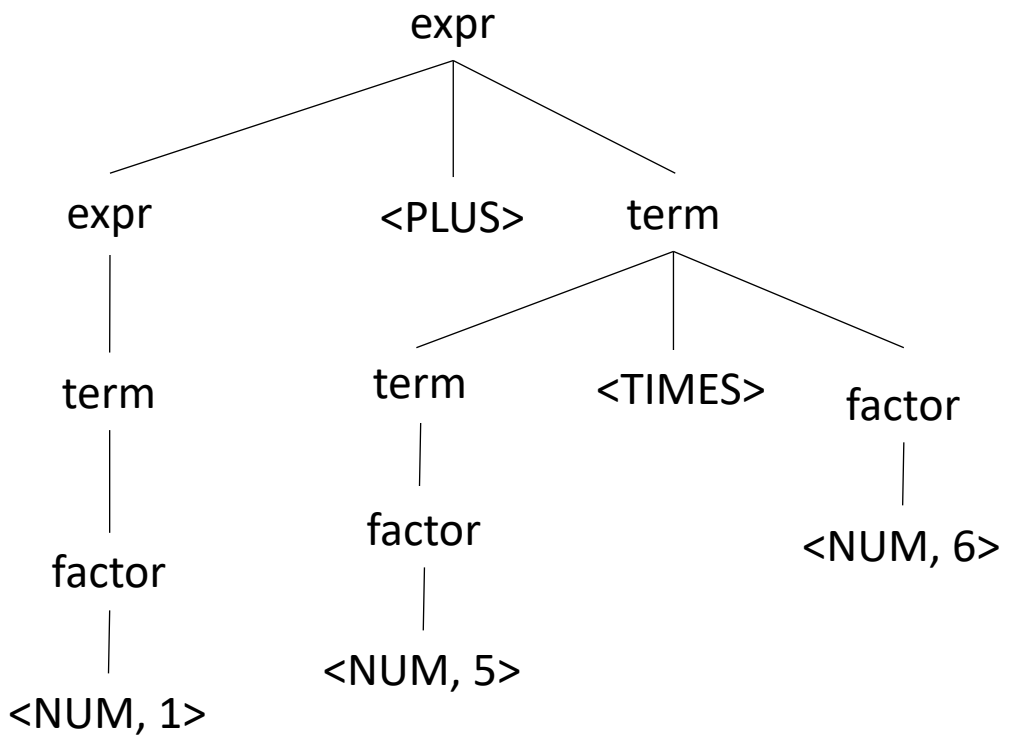
Production actions

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

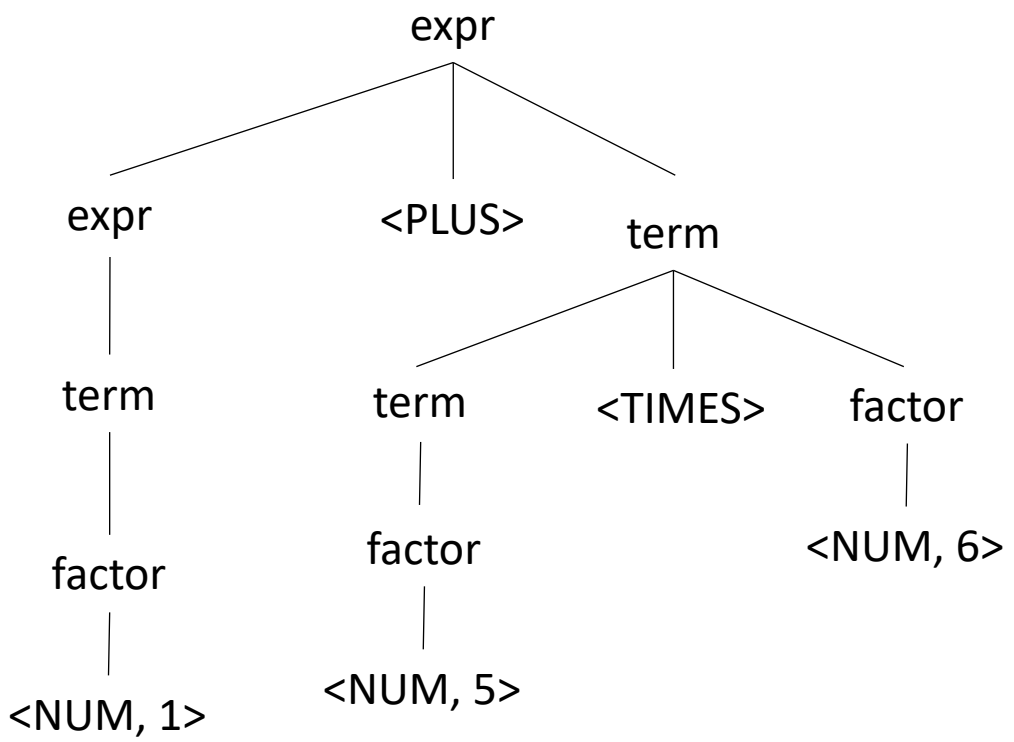
Operator	Name	Productions	Actions

+,-	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 Tuesday
- Enjoy your weekend!