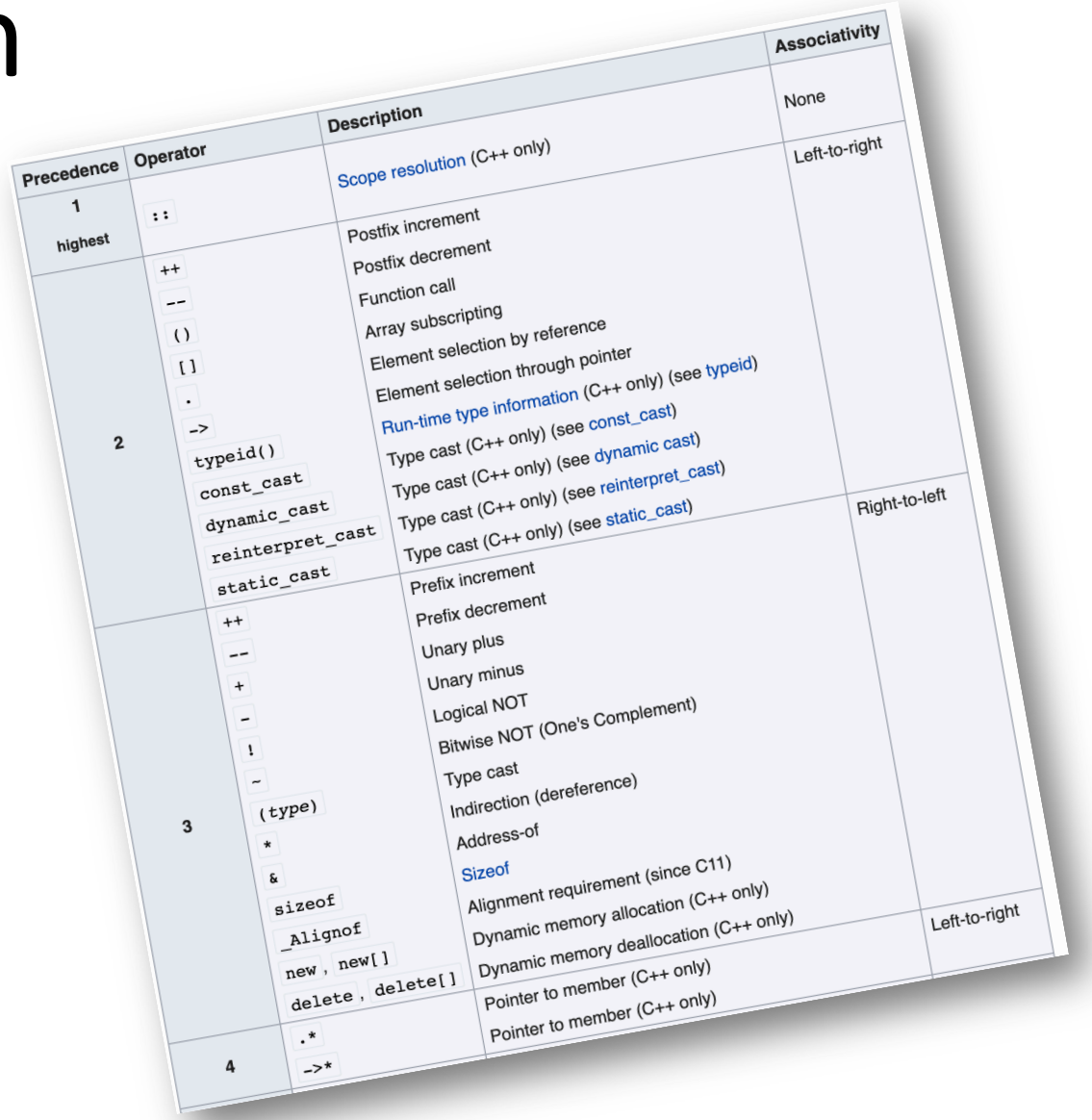
CSE211: Compiler Design

Oct. 8, 2020

- **Topic**: Parsing

- **Questions**:
  - *What is operator precedence? What is it in your favorite languages*
  - *What is operator associativity? What operators are associative and what are not?*



Precedence	Operator	Description	Associativity
1 highest	`::`	Scope resolution (C++ only)	None
2	`++`	Postfix increment	Left-to-right
	`--`	Postfix decrement	
	`()`	Function call	
	`[]`	Array subscripting	
	`.`	Element selection by reference	
	`->`	Element selection through pointer	
	`typeid()`	Run-time type information (C++ only) (see typeid)	
	`const_cast`	Type cast (C++ only) (see const_cast)	
	`dynamic_cast`	Type cast (C++ only) (see dynamic cast)	
	`reinterpret_cast`	Type cast (C++ only) (see reinterpret_cast)	
	`static_cast`	Type cast (C++ only) (see static_cast)	
3	`++`	Prefix increment	Right-to-left
	`--`	Prefix decrement	
	`+`	Unary plus	
	`-`	Unary minus	
	`!`	Logical NOT	
	`~`	Bitwise NOT (One's Complement)	
	`(type)`	Type cast	
	`*`	Indirection (dereference)	
	`&`	Address-of	
	`sizeof`	Sizeof	
	`_Alignof`	Alignment requirement (since C11)	
	`new`, `new[]`	Dynamic memory allocation (C++ only)	
	`delete`, `delete[]`	Dynamic memory deallocation (C++ only)	
4	`.*`	Pointer to member (C++ only)	Left-to-right
	`->*`	Pointer to member (C++ only)	

Announcements

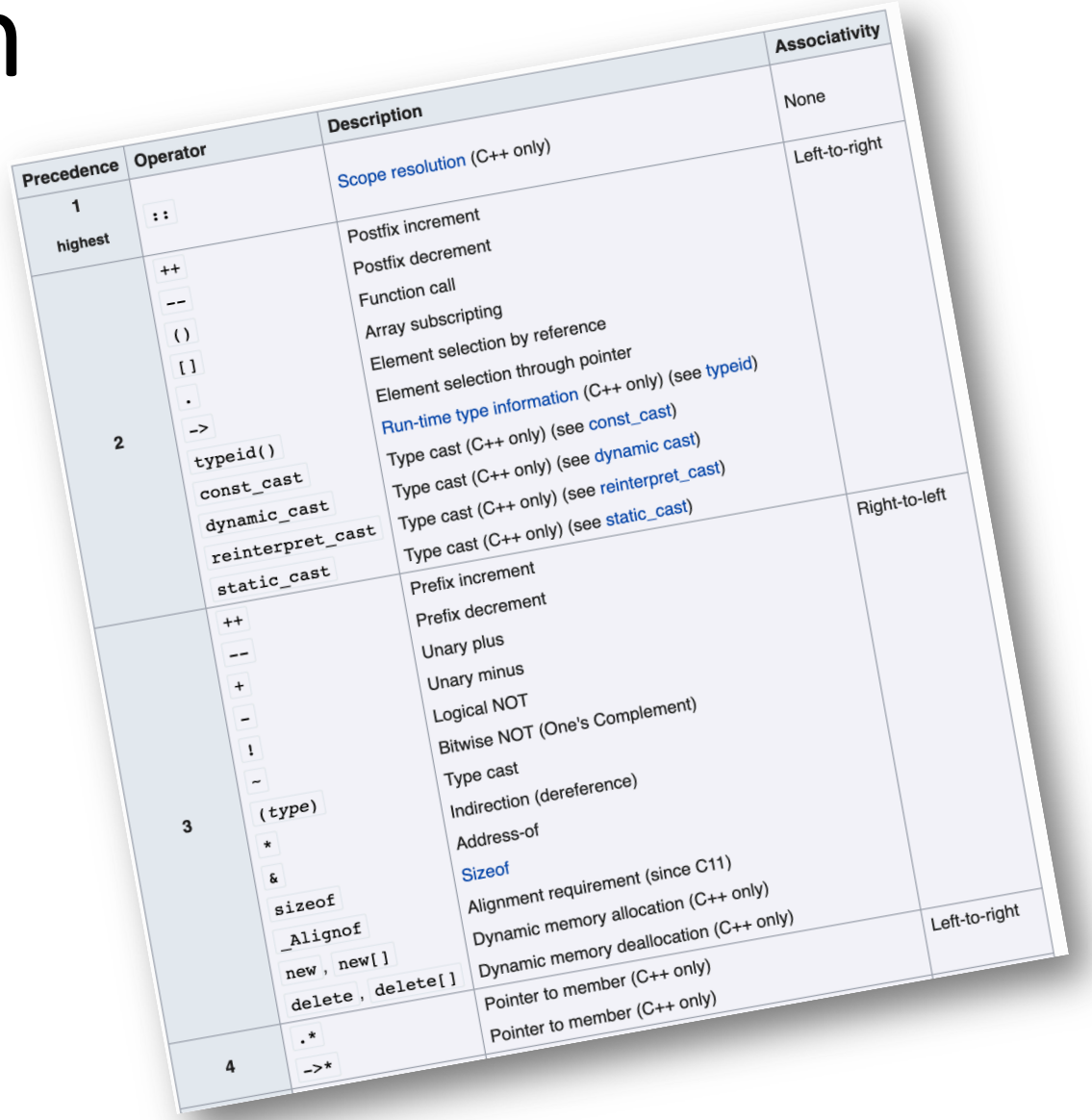
- Homework released! Have a look but don’t panic
  - Remember, due dates pushed back 1 week
  - We will go over parsing with derivatives this/next week
  - We will go over PLY this/next week

CSE211: Compiler Design

Oct. 8, 2020

- **Topic**: Parsing

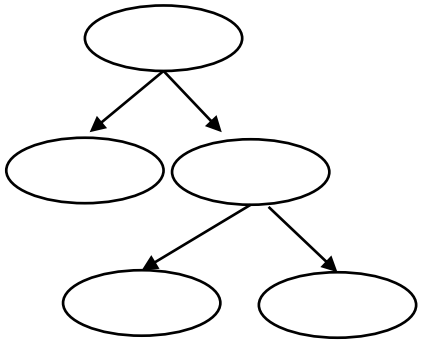
- **Questions**:
  - *What is operator precedence? What is it in your favorite languages*
  - *What is operator associativity? What operators are associative and what are not?*

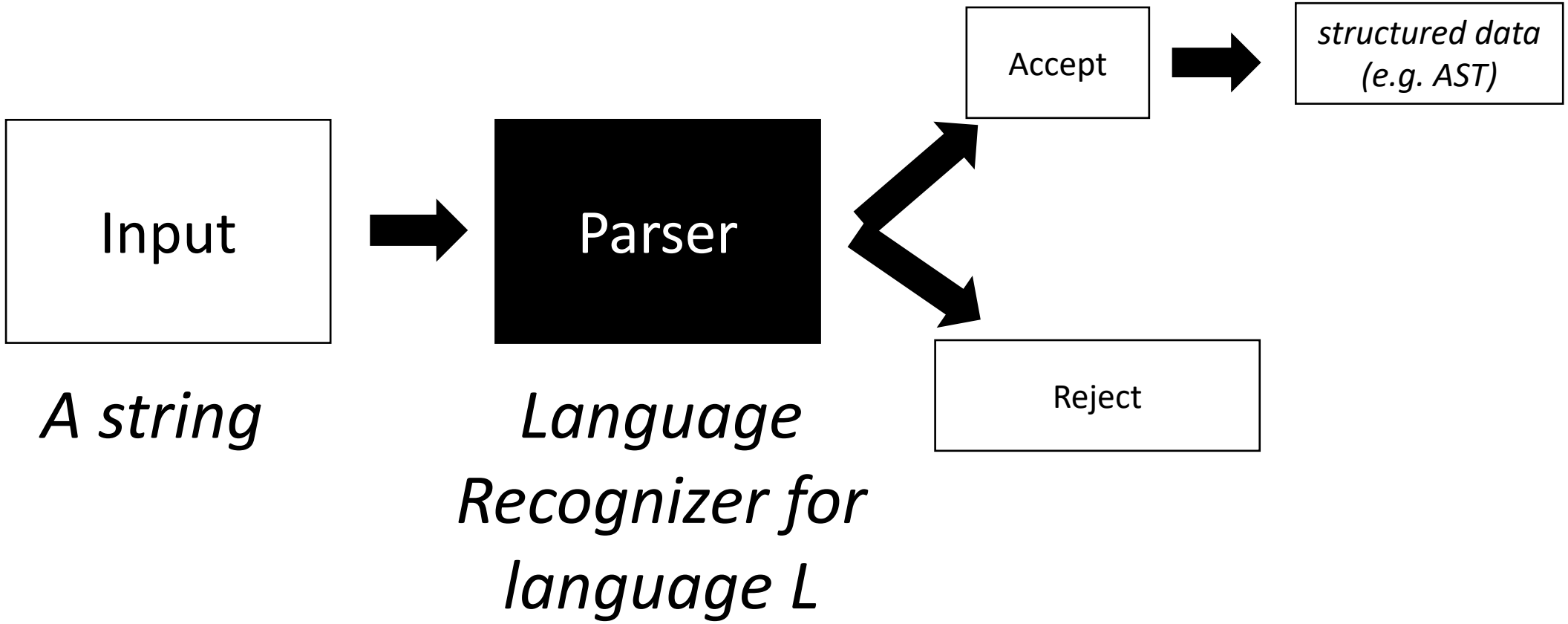


Precedence	Operator	Description	Associativity
1 highest	`::`	Scope resolution (C++ only)	None
2	`++`	Postfix increment	Left-to-right
	`--`	Postfix decrement	
	`()`	Function call	
	`[]`	Array subscripting	
	`.`	Element selection by reference	
	`->`	Element selection through pointer	
	`typeid()`	Run-time type information (C++ only) (see typeid)	
	`const_cast`	Type cast (C++ only) (see const_cast)	
	`dynamic_cast`	Type cast (C++ only) (see dynamic cast)	
	`reinterpret_cast`	Type cast (C++ only) (see reinterpret_cast)	
	`static_cast`	Type cast (C++ only) (see static_cast)	
3	`++`	Prefix increment	Right-to-left
	`--`	Prefix decrement	
	`+`	Unary plus	
	`-`	Unary minus	
	`!`	Logical NOT	
	`~`	Bitwise NOT (One's Complement)	
	`(type)`	Type cast	
	`*`	Indirection (dereference)	
	`&`	Address-of	
	`sizeof`	Sizeof	
	`_Alignof`	Alignment requirement (since C11)	
	`new`, `new[]`	Dynamic memory allocation (C++ only)	
	`delete`, `delete[]`	Dynamic memory deallocation (C++ only)	
4	`.*`	Pointer to member (C++ only)	Left-to-right
	`->*`	Pointer to member (C++ only)	

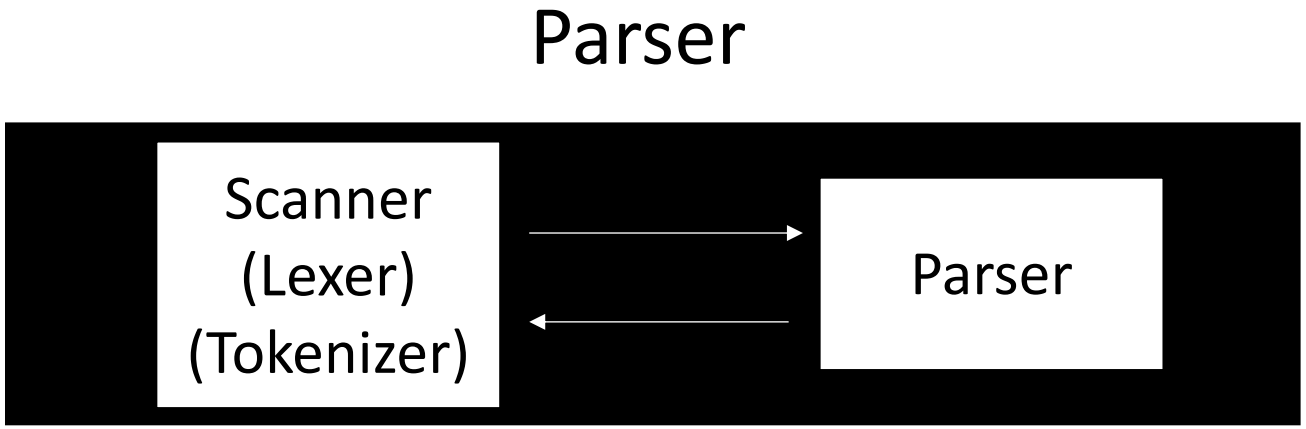
High-level parser

continue to the rest of compilation





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)

BNF specification of context-free grammar

- Sentence:
  - ARTICLE ADJECTIVE* NOUN VERB

The big red fox jumped

The fox jumped

BNF specification of context-free grammar

- Production rules:

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

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

  sentence: ARTICLE ADJECTIVE NOUN VERB
  | ARTICLE NOUN VERB

BNF specification of context-free grammar

- Production rules can reference other production rules

sentence: adjective_sentence
| non_adjective_sentence

adjective_sentence: ARTICLE ADJECTIVE NOUN VERB

non_adjective_sentence: ARTICLE NOUN VERB

BNF specification of context-free grammar

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

sentence: ARTICLE adjective_list NOUN VERB

BNF specification of context-free grammar

- 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>

BNF specification of context-free grammar

- Sentence:
  - ARTICLE ADJECTIVE? NOUN VERB

- **What about a mathematical sentence (expression)?**

  - NUM
  - NUM PLUS NUM
  - NUM TIMES NUM
  - NUM PLUS NUM TIMES NUM
  - ….

Production rules for expressions

expression : NUM

| NUM PLUS NUM

| NUM TIMES NUM

Production rules for expressions

expression : NUM

| NUM PLUS NUM

| NUM TIMES NUM

expression : NUM

| expression PLUS expression

| expression TIMES expression

Why not just use regular expressions?

- NUM = “[0-9]+”

Why not just use regular expressions?

- NUM = “[0-9]+”
- OP = “+|*”

Why not just use regular expressions?

- NUM = “[0-9]+”
- OP = “+|*”
- expression = “NUM (OP NUM)*”

Why not just use regular expressions?

- NUM = “[0-9]+”
- OP = “+|*”
- expression = “NUM (OP NUM)*”

For example, this matches: “1+2+5600+6*7”

What about ()’s

- there is a formal proof available that regex CANNOT match ()’s: pumping lemma
- Informal argument:
  - Try matching ((^n)()^n) using Kleene star
- What about production rules?

For proof sketch:

Why we can’t match () with regex:

can we match: “((()))” “ \(*\)*”

BNF for expressions

expression : NUM

| expression PLUS expression

| expression TIMES expression

BNF for expressions

expression : NUM

| expression PLUS expression

| expression TIMES expression

| LPAREN expression RPAREN

BNF for expressions

expression : NUM

| expression PLUS expression

| expression TIMES expression

| LPAREN expression RPAREN

Where else in programming languages are matching constructs used?

Exercise

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
```

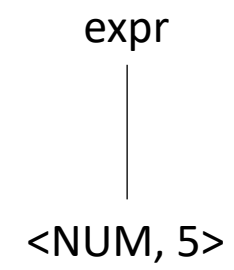
expr

Parse trees

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

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

```
input: 5
```



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
```

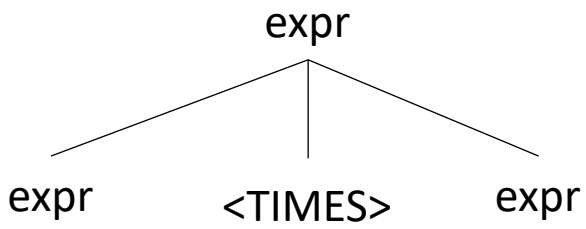
expr

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
```

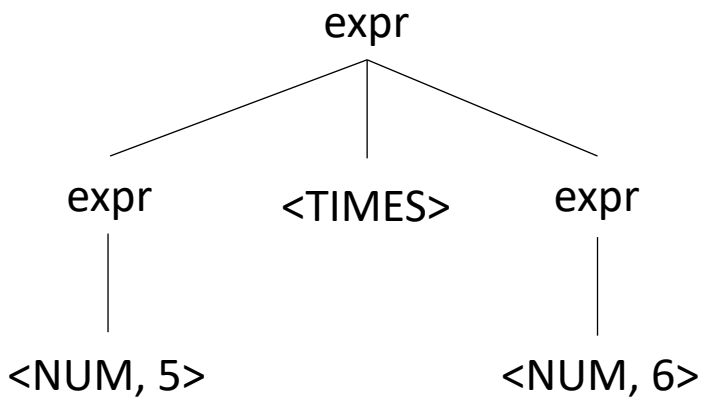


Parse trees

- 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

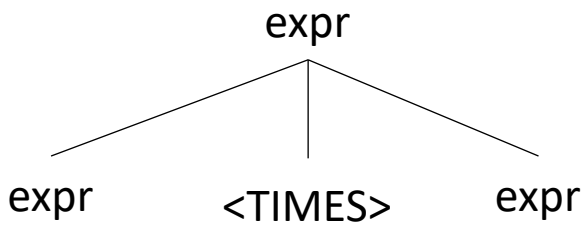
Parse trees

- 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?



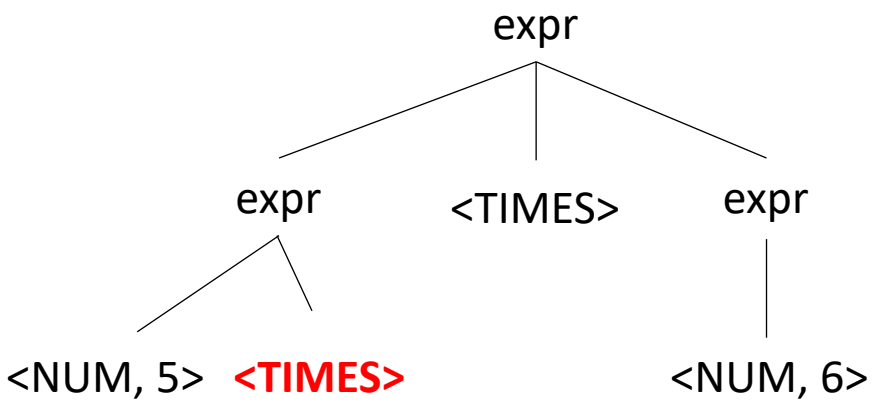
Parse trees

- 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

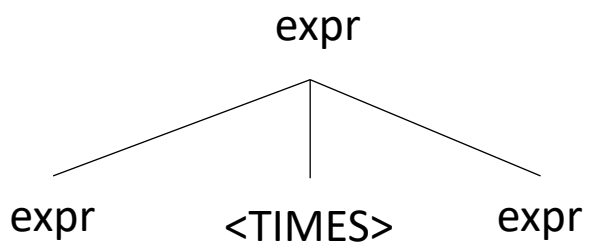
expr

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
```

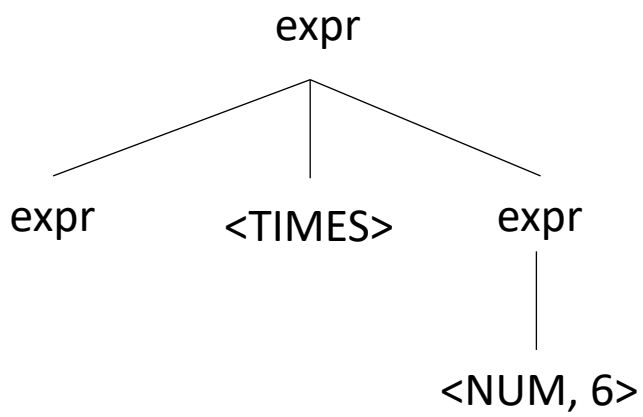


Parse trees

- 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
```

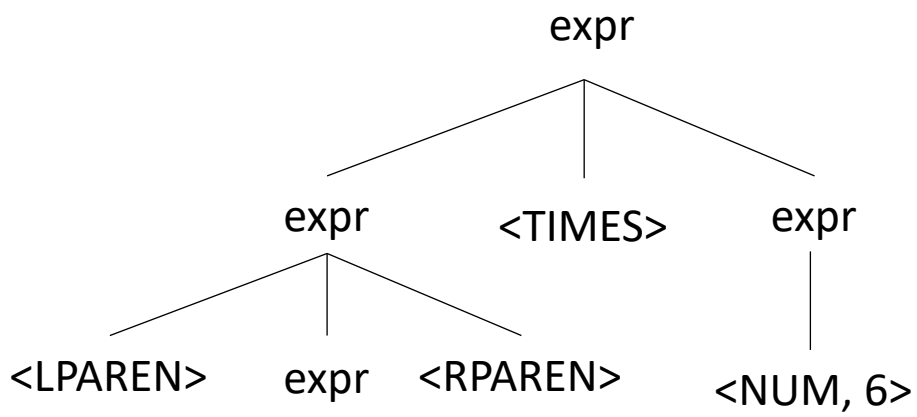


Parse trees

- 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
```

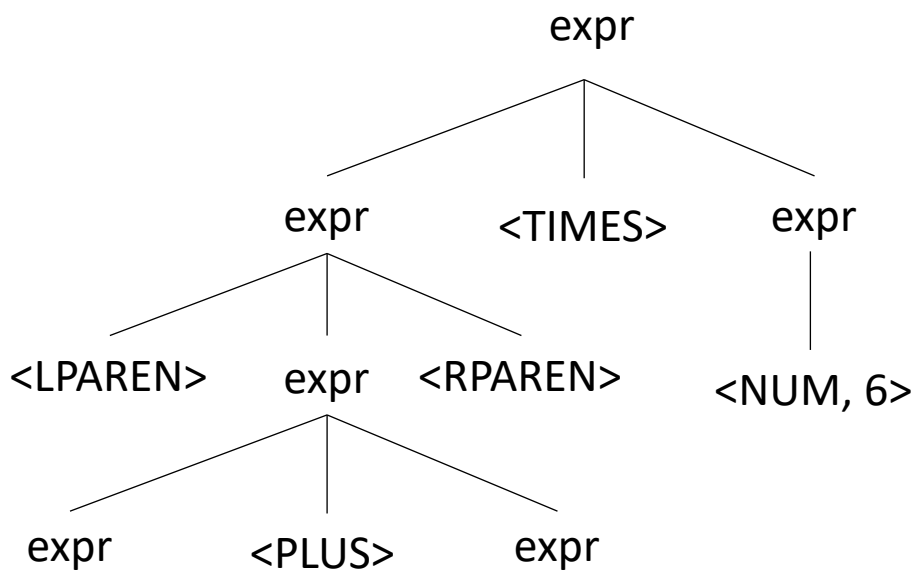


Parse trees

- 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
```

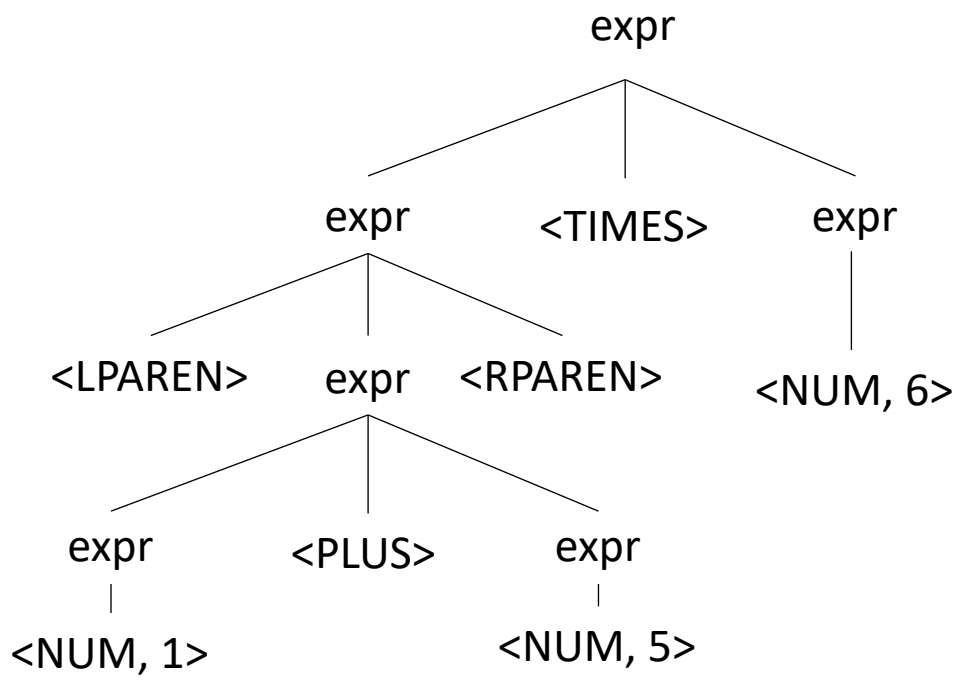


Parse trees

- 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
```

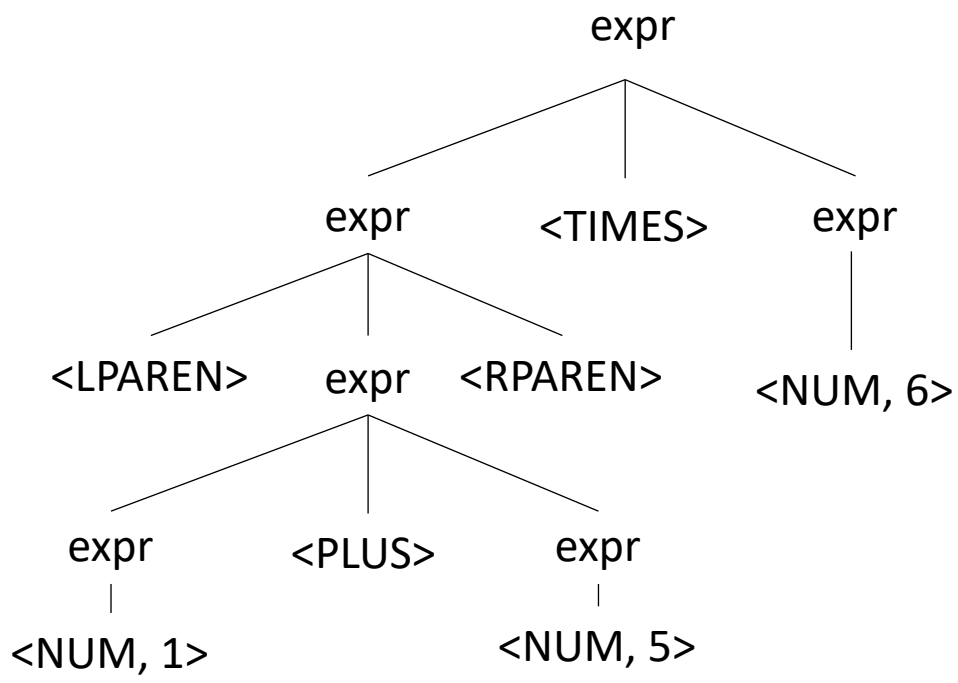


Parse trees

- How to create a parse tree from a string?

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

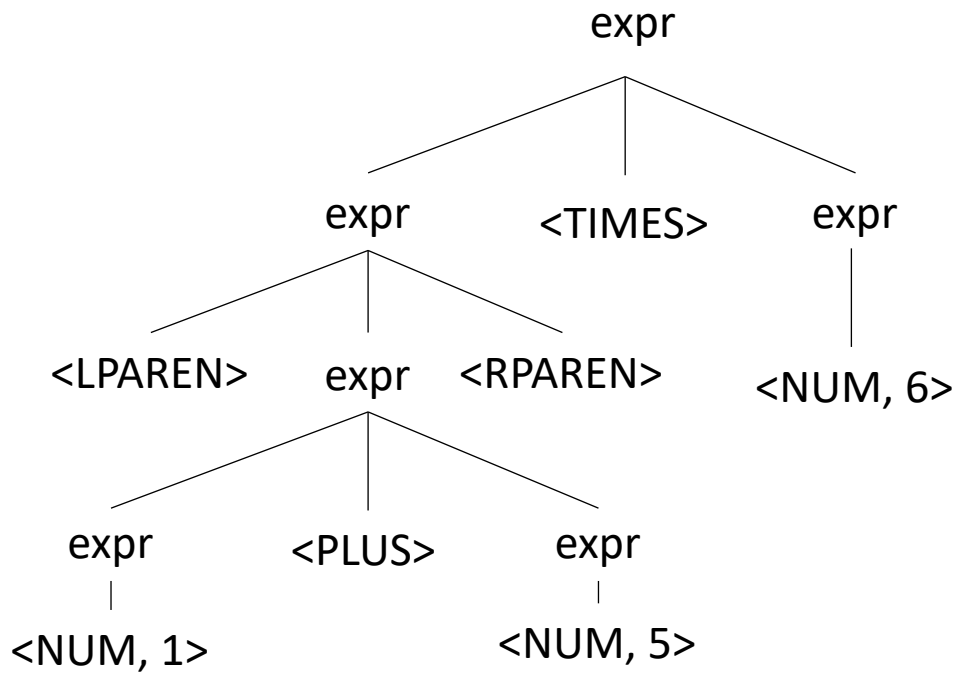
```
expr : NUM
     | expr PLUS expr
     | expr TIMES expr
     | LPAREN expr RPAREN
```



Parse trees

- How to create a string from a parse tree?

```
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

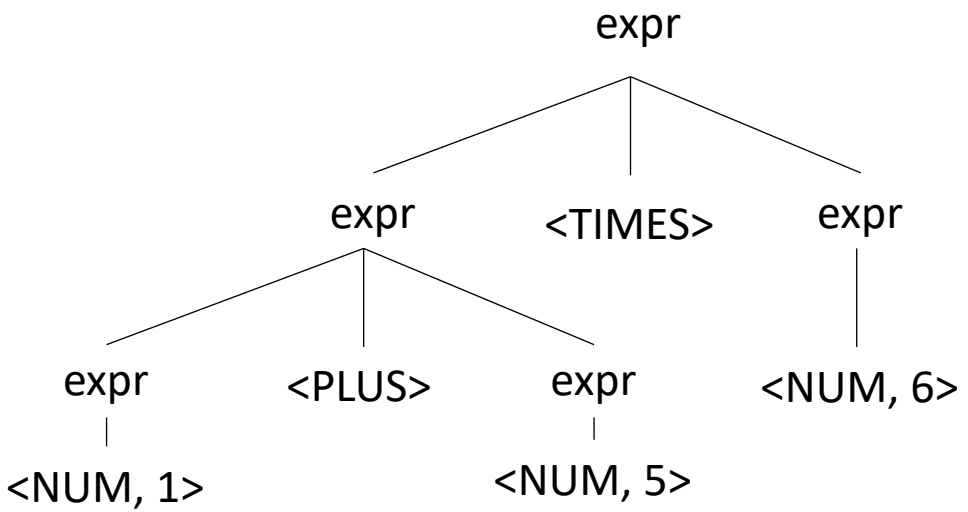
| expr TIMES expr

| LPAREN expr RPAREN

Parse trees

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

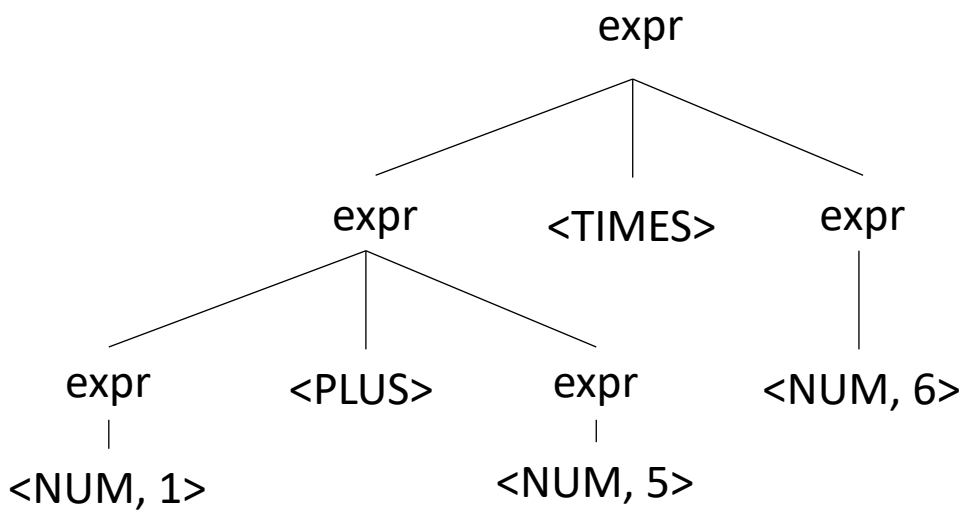
```
expr : NUM
     | expr PLUS expr
     | expr TIMES expr
     | LPAREN expr RPAREN
```



Ambiguous grammars

- `input: 1 + 5 * 6`

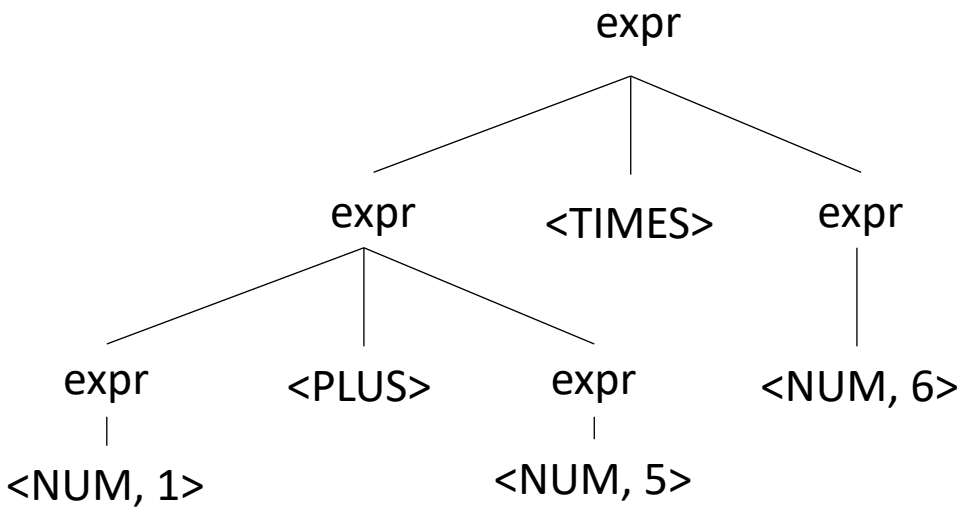
```
expr : NUM
     | expr PLUS expr
     | expr TIMES expr
     | LPAREN expr RPAREN
```

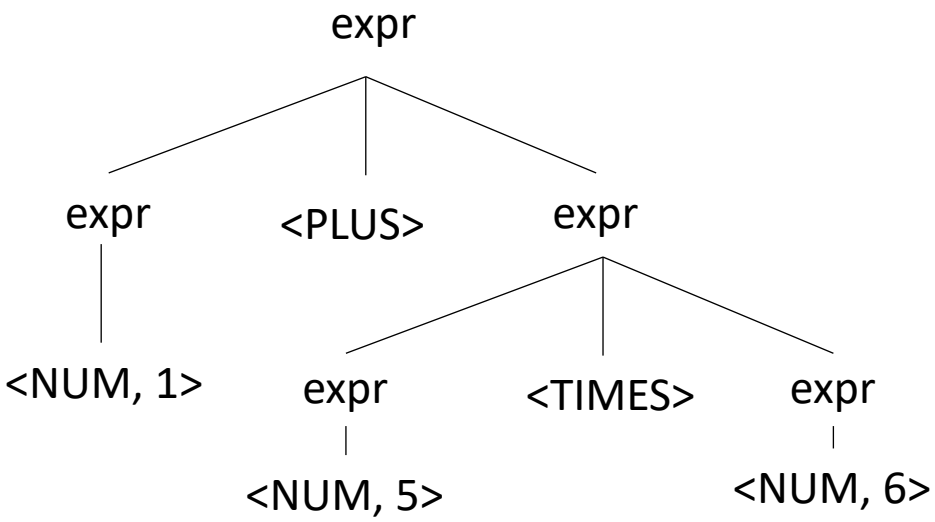


Ambiguous grammars

- `input: 1 + 5 * 6`

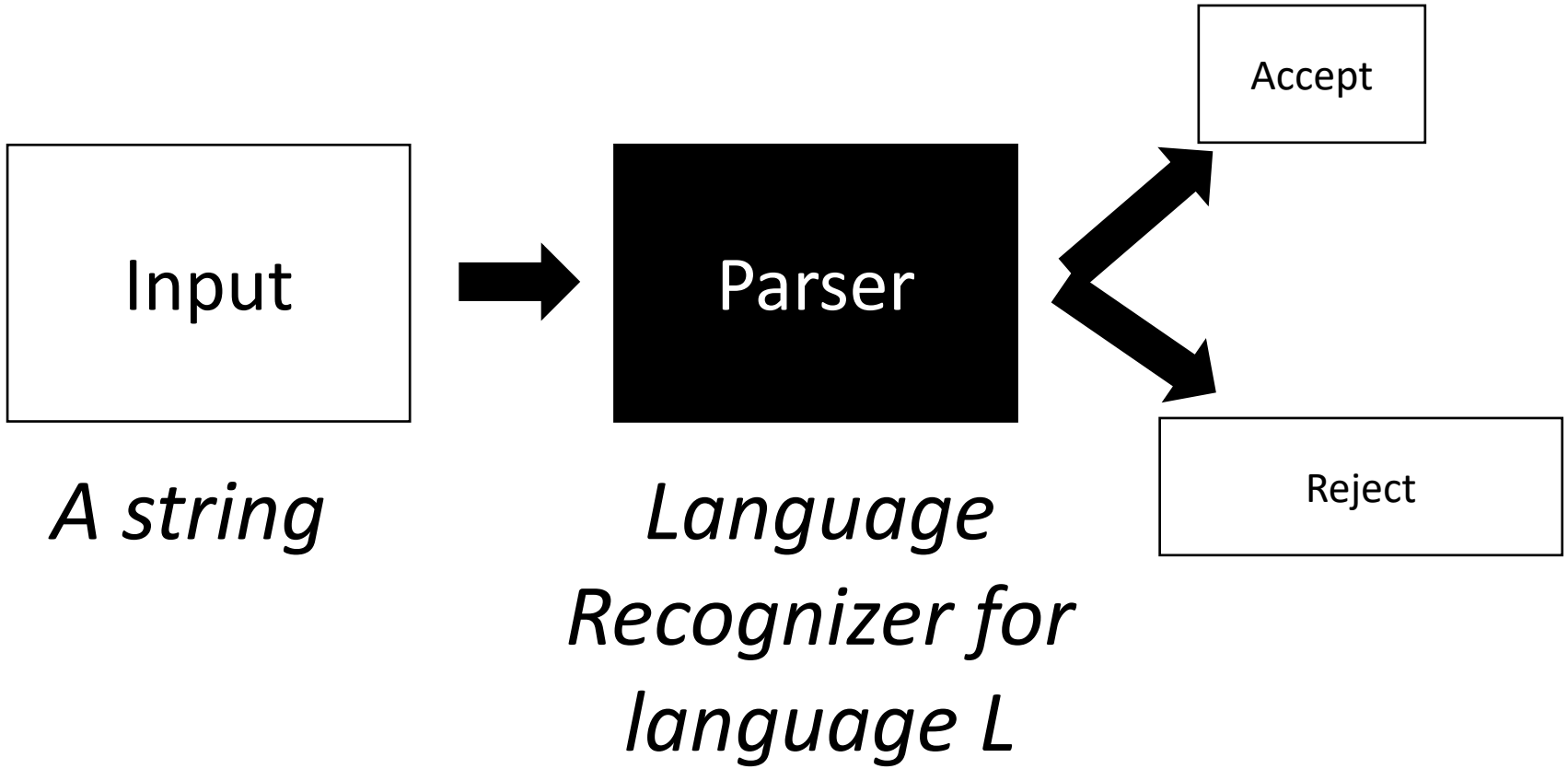
```
expr : NUM
     | expr PLUS expr
     | expr TIMES expr
     | LPAREN expr RPAREN
```





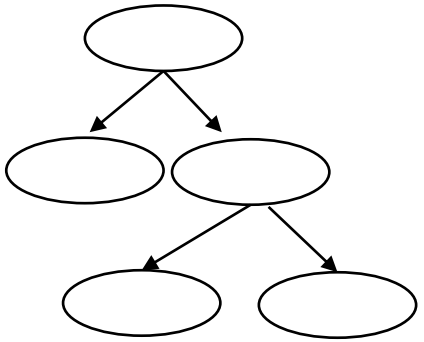
Ambiguous grammars

- What's the issue?

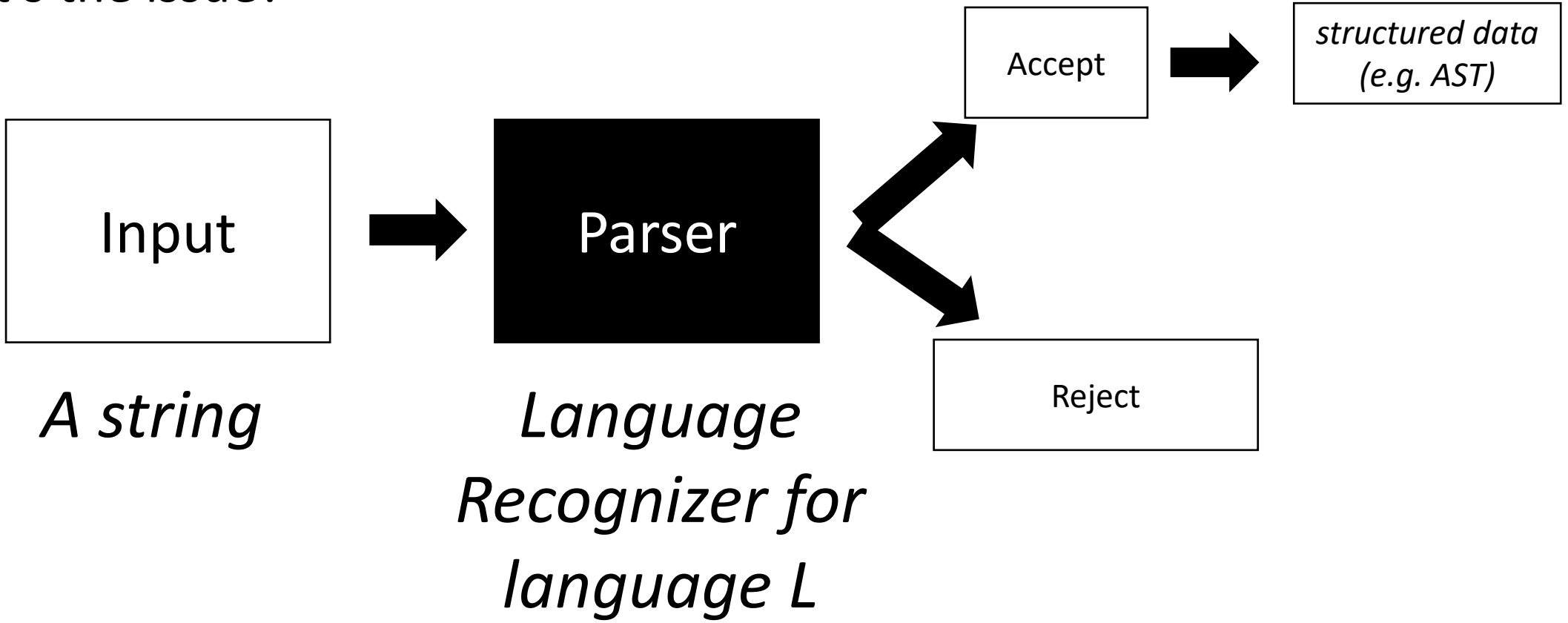


Ambiguous grammars

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

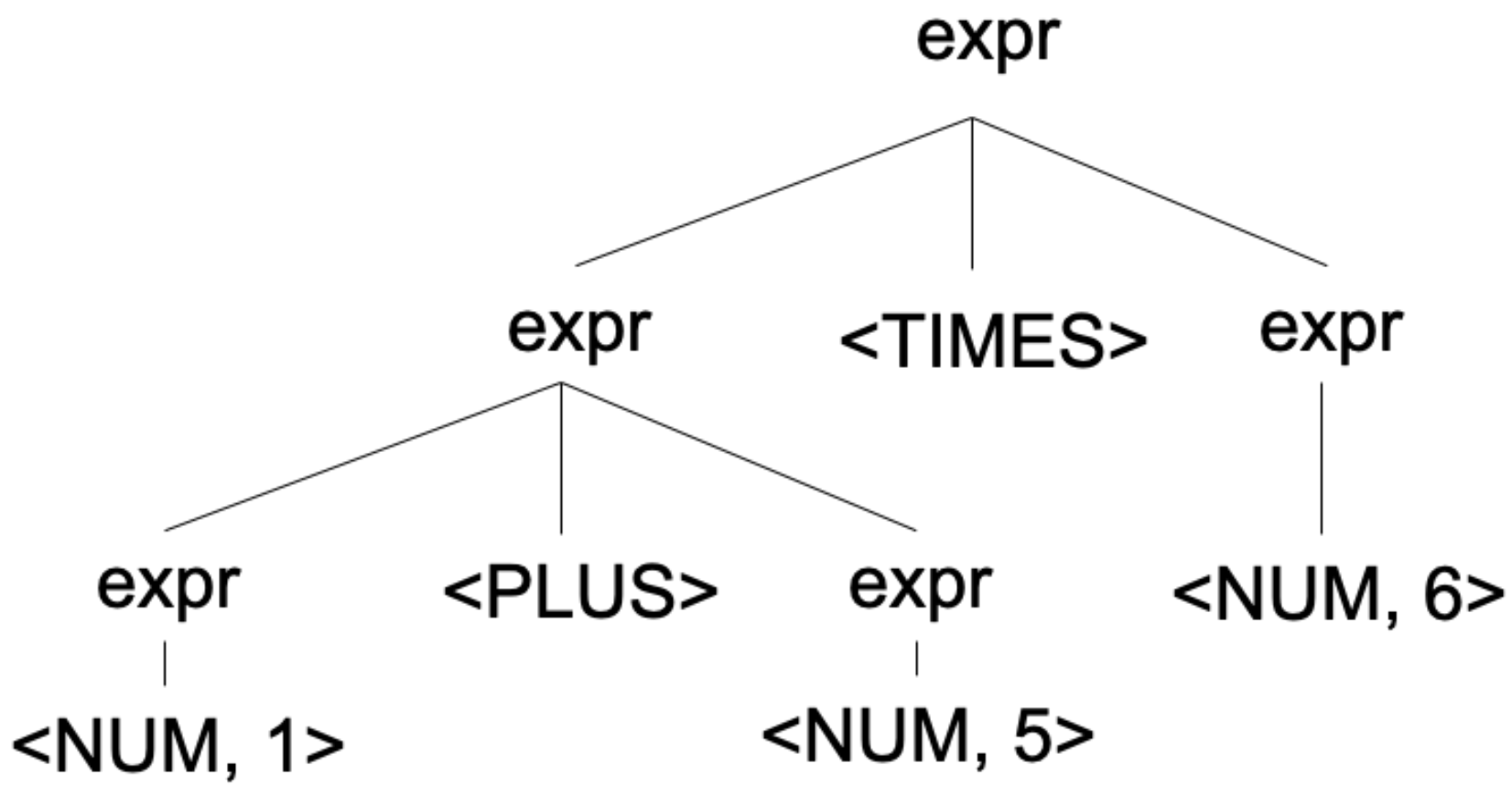
Meaning into structure

- 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
- Traditionally encodes the order of operation

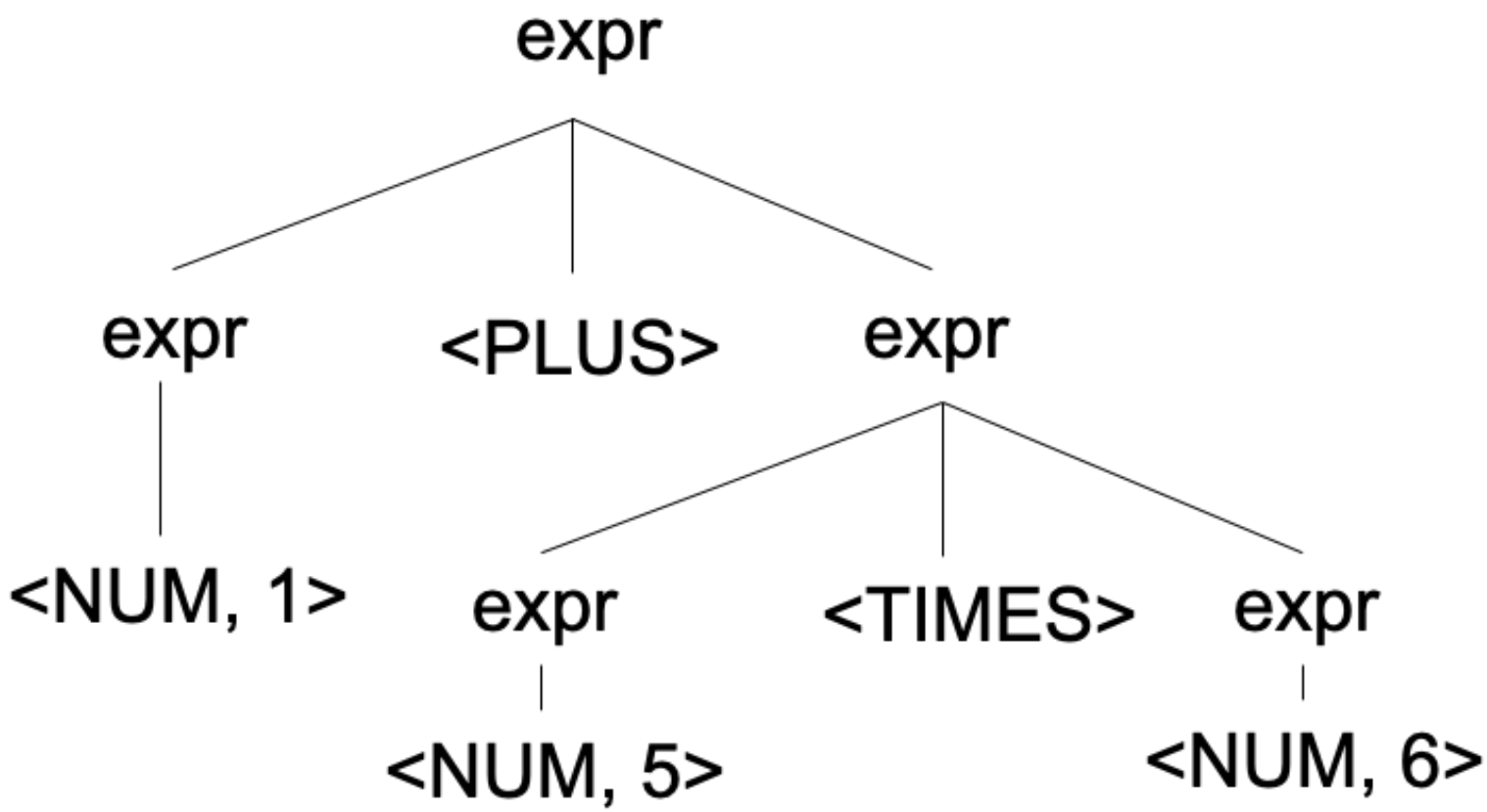
Work through examples

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



Work through examples

`input: 1+5*6`



Meaning into structure

- 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

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

Meaning into structure

- 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 example

- 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 example

- 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 + Expr | Term
*	Term	: Term * Term | Factor
()	Factor	: (Expr) | NUM

Precedence example

- 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 + Expr | Term
*	Term	: Term * Term | Factor
()	Factor	: (Expr) | NUM

Now lets create a parse tree

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

Operator	Name	Productions
+	Expr	: Expr+Expr | Term
*	Term	: Term * Term | Factor
()	Factor	: (Expr) | NUM

Now lets create a parse tree

`input: 1+5*6`

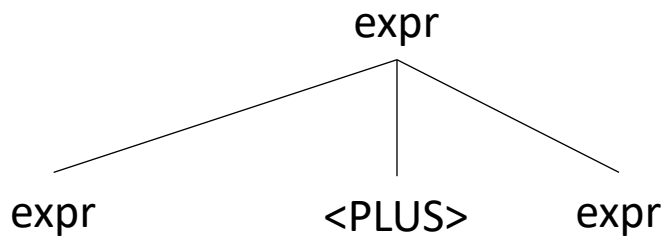
Operator	Name	Productions
+	Expr	: Expr+Expr | Term
*	Term	: Term * Term | Factor
()	Factor	: (Expr) | NUM

expr

Now lets create a parse tree

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

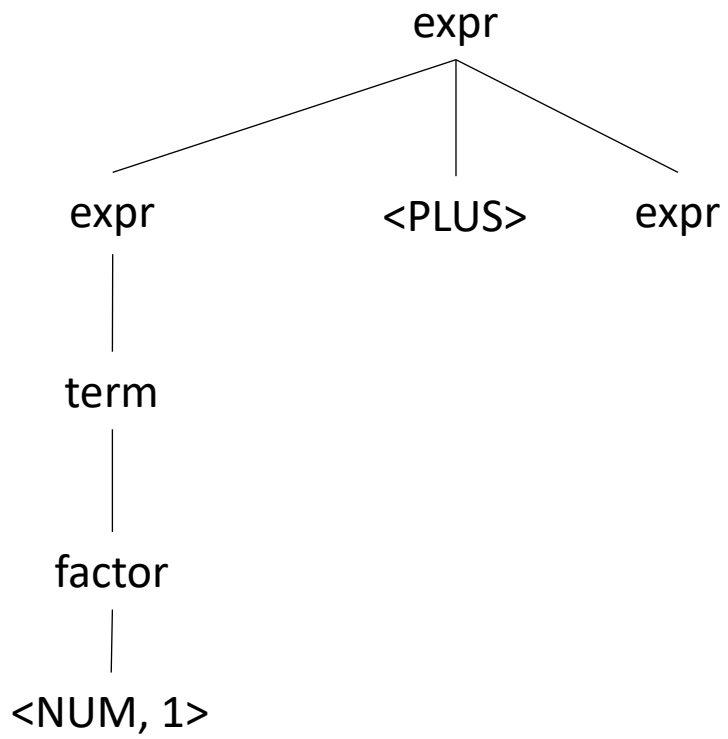
Operator	Name	Productions
+	Expr	: Expr+Expr | Term
*	Term	: Term * Term | Factor
()	Factor	: (Expr) | NUM



Now lets create a parse tree

`input: 1+5*6`

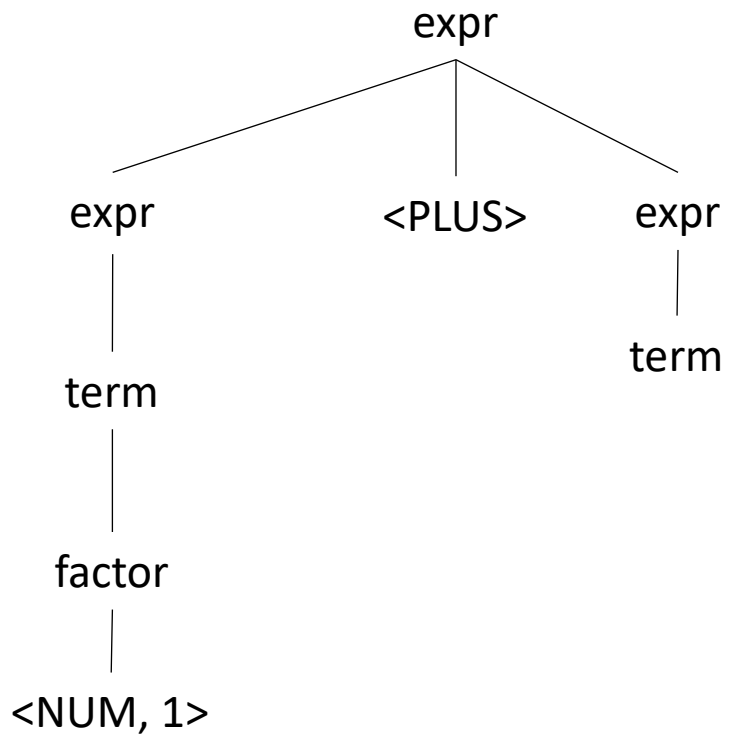
Operator	Name	Productions
+	Expr	: Expr+Expr | Term
*	Term	: Term * Term | Factor
()	Factor	: (Expr) | NUM



Now lets create a parse tree

`input: 1+5*6`

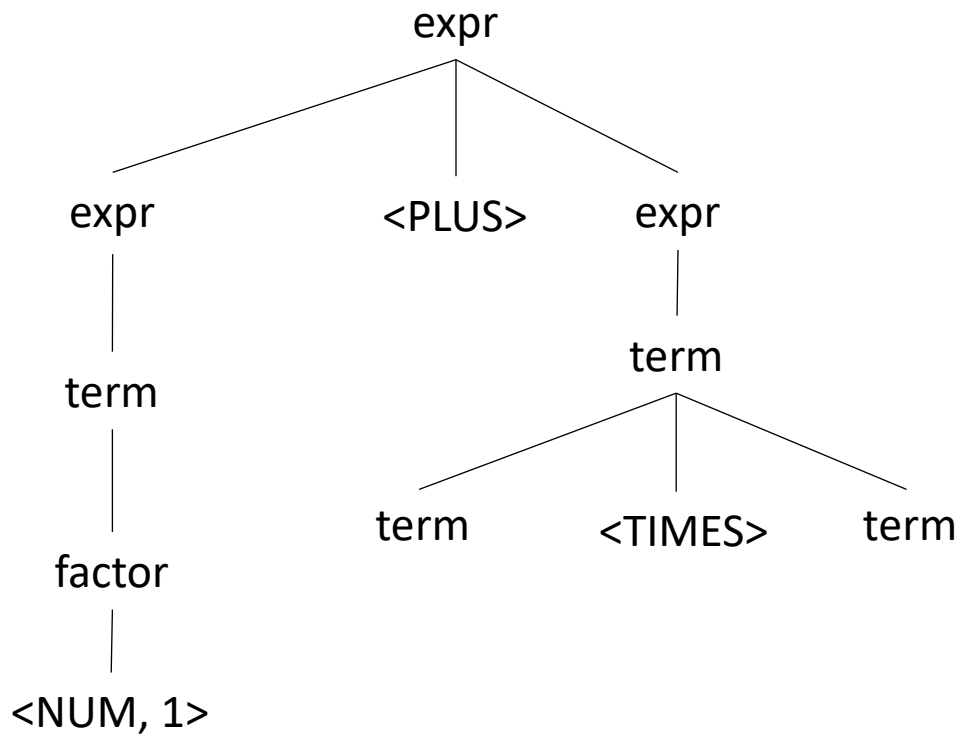
Operator	Name	Productions
+	Expr	: Expr+Expr | Term
*	Term	: Term * Term | Factor
()	Factor	: (Expr) | NUM



Now lets create a parse tree

`input: 1+5*6`

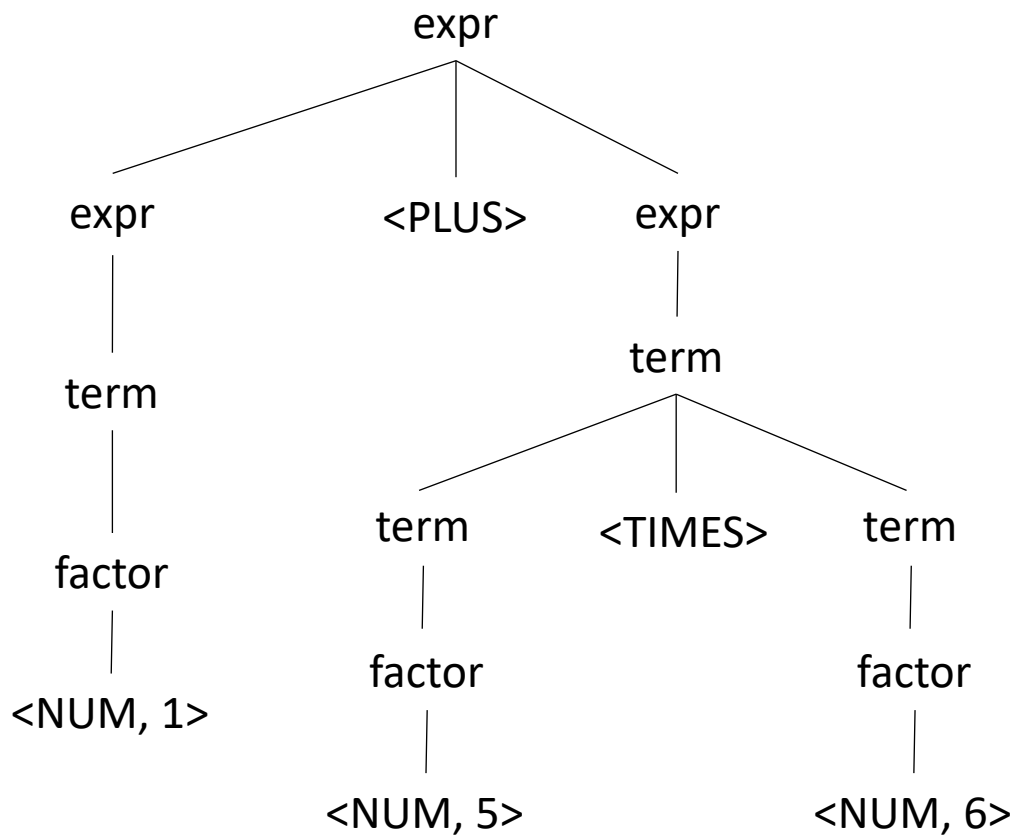
Operator	Name	Productions
+	Expr	: Expr+Expr | Term
*	Term	: Term * Term | Factor
()	Factor	: (Expr) | NUM



Now lets create a parse tree

`input: 1+5*6`

Operator	Name	Productions
+	Expr	: Expr+Expr | Term
*	Term	: Term * Term | Factor
()	Factor	: (Expr) | NUM



Regular expression example

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

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

Regular expression example

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

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

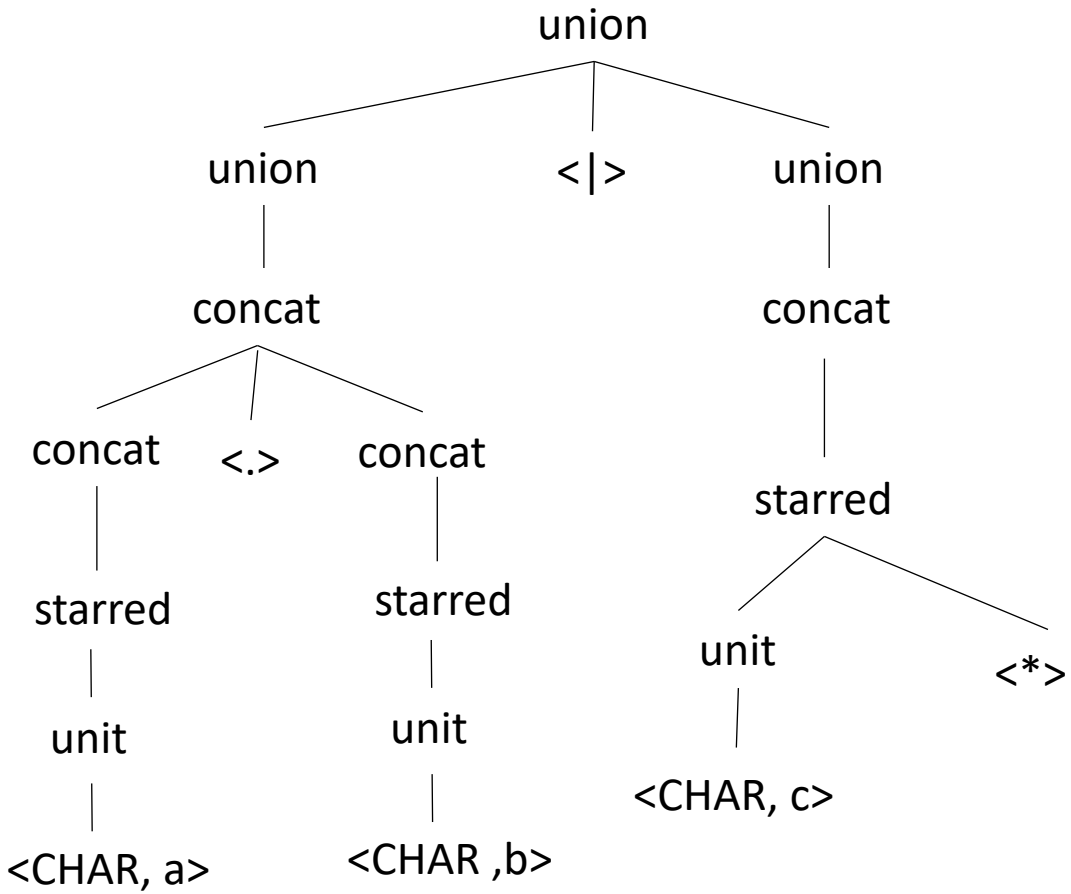
Operator	Name	Productions
|	union	: union | union | concat
.	concat	: concat . concat | starred
*	starred	: starred * | unit
()	unit	: (union) | CHAR

Regular expression example

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

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

`input: a.b | c*`



Are we finished?

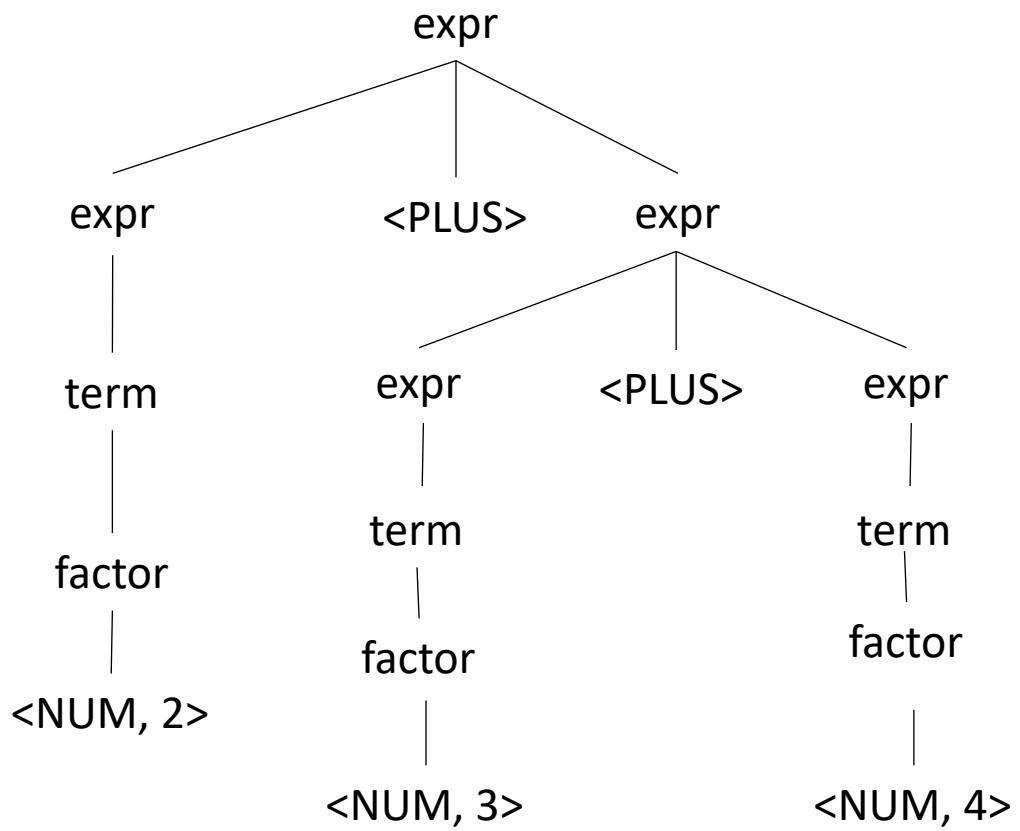
```
input: 2+3+4
```

Operator	Name	Productions
+	Expr	: Expr+Expr | Term
*	Term	: Term * Term | Factor
()	Factor	: (Expr) | NUM

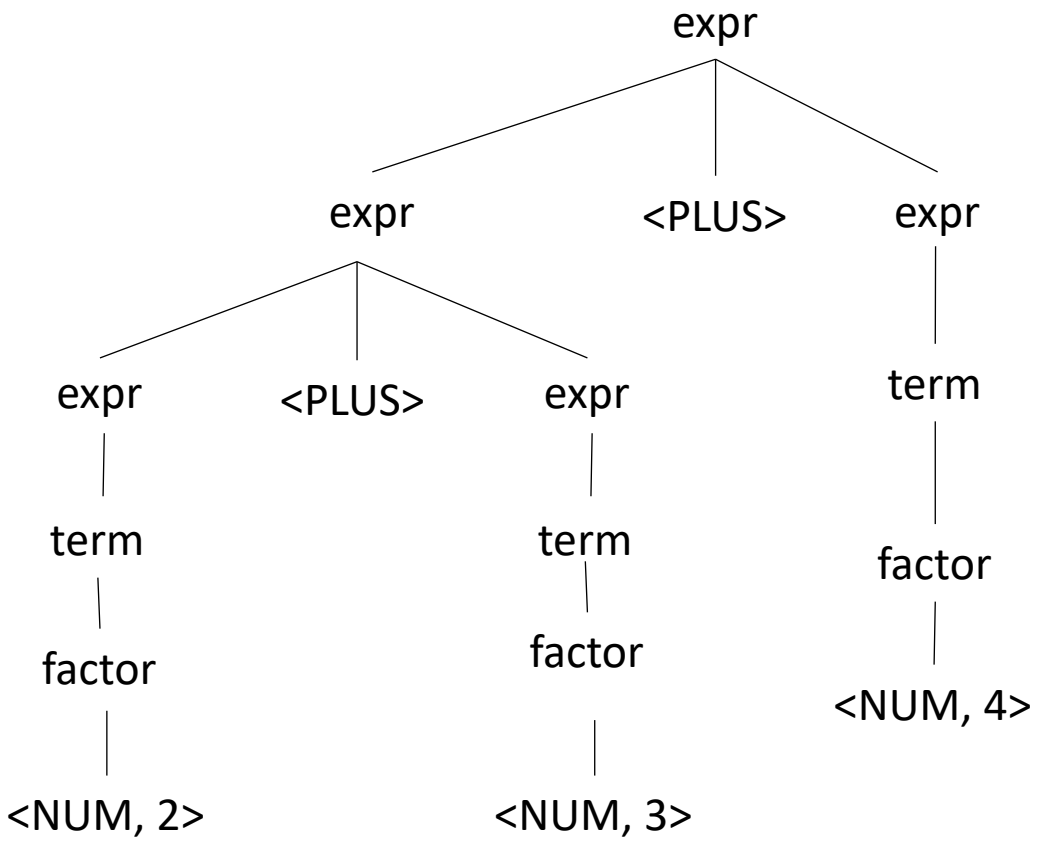
Are we finished?

Operator	Name	Productions
+	Expr	: Expr+Expr | Term
*	Term	: Term * Term | Factor
()	Factor	: (Expr) | NUM

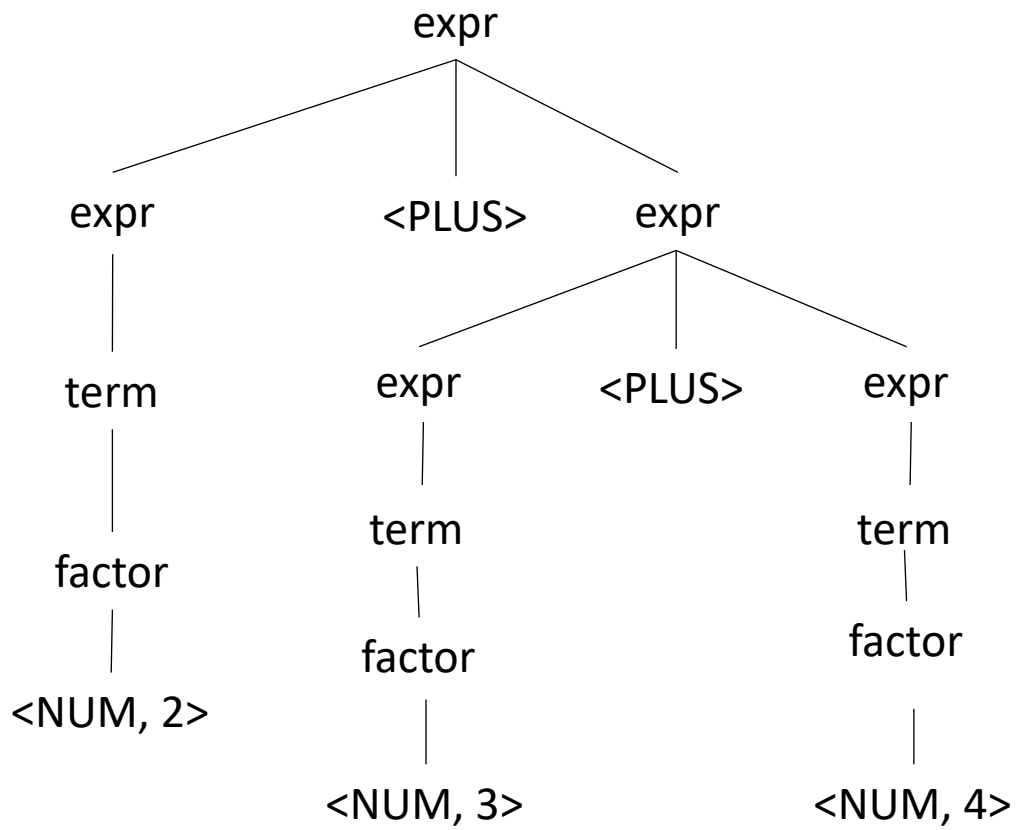
```
input: 2+3+4
```



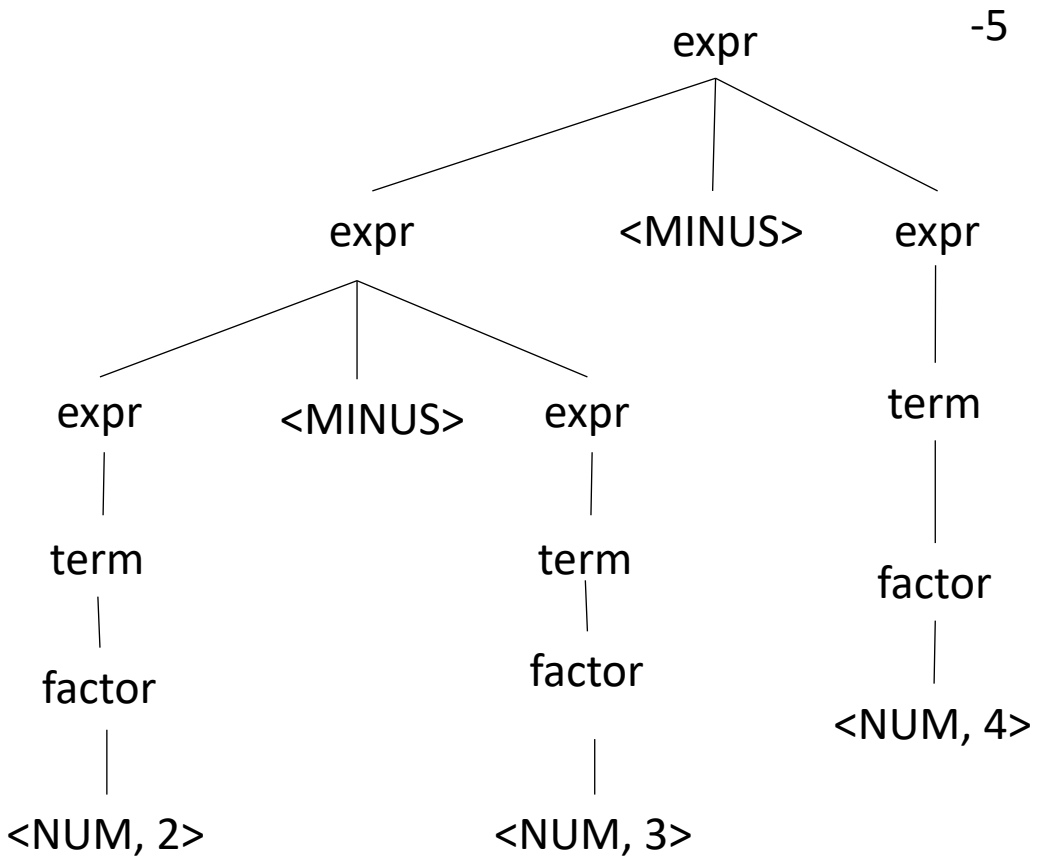
Are we finished?



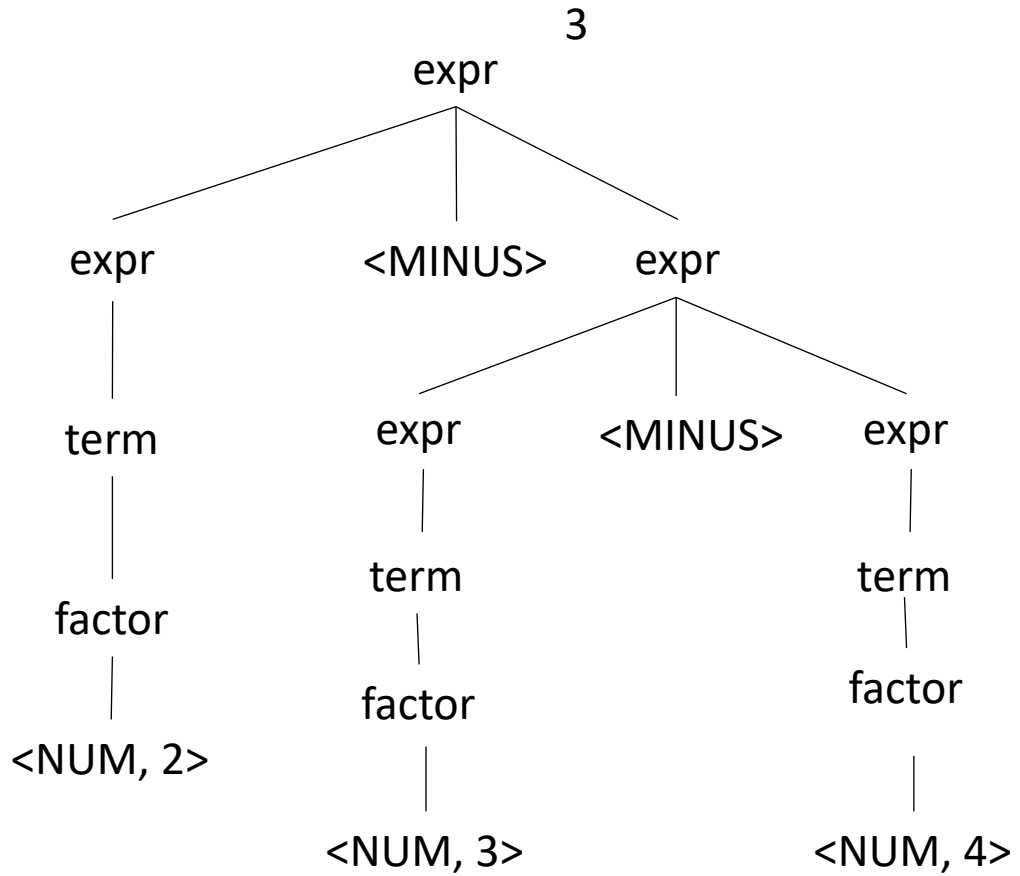

```
input: 2+3+4
```



Are we finished?



`input: 2-3-4`



Associativity

- Some operators are associative (+,*).
  - You should define associativity anyways! Avoid nondeterminism!
- Some are left associative (-, /)
- Are any right associative? ^

Meaning into structure

- How to avoid ambiguity related to precedence?
- Some parser generators allow you to specify it, e.g. Yacc
- You can also modify production rules:
  - left associative has recursion on the left
  - right associative has recursion on the right

Putting it together

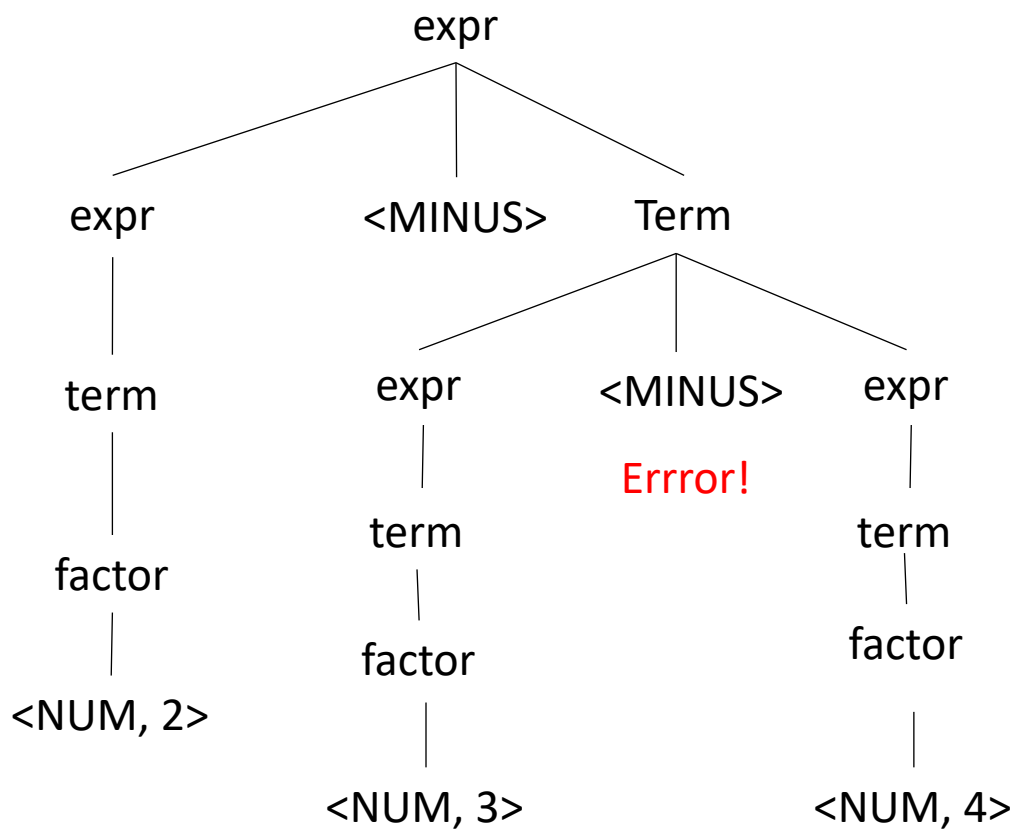
- expressions with {+ - * / ^}

Operator	Name	Productions	
+,-	Expr	: Expr + Term | Expr - Term | Term	left associative
*,/	Term	: Term * Pow : Term / Pow | Pow	left associative
^	Pow	: Factor ^ Pow | Factor	right associative

Are we finished?

Operator	Name	Productions
+,-	Expr	: Expr + Term | Expr - Term | Term
*,/	Term	: Term * Factor : Term / Factor | Factor
()	Factor	: (Expr) | NUM

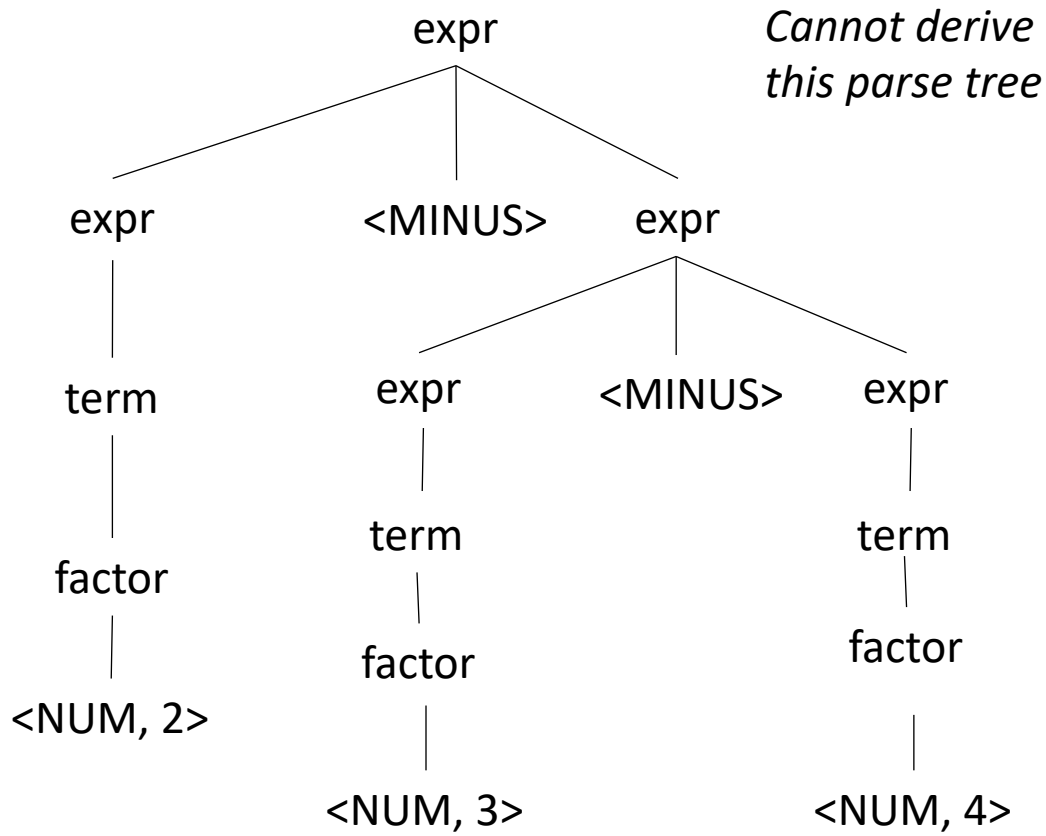
`input: 2-3-4`



Are we finished?

Operator	Name	Productions
+,-	Expr	: Expr + Term | Expr - Term | Term
*,/	Term	: Term * Factor : Term / Factor | Factor
()	Factor	: (Expr) | NUM

`input: 2-3-4`

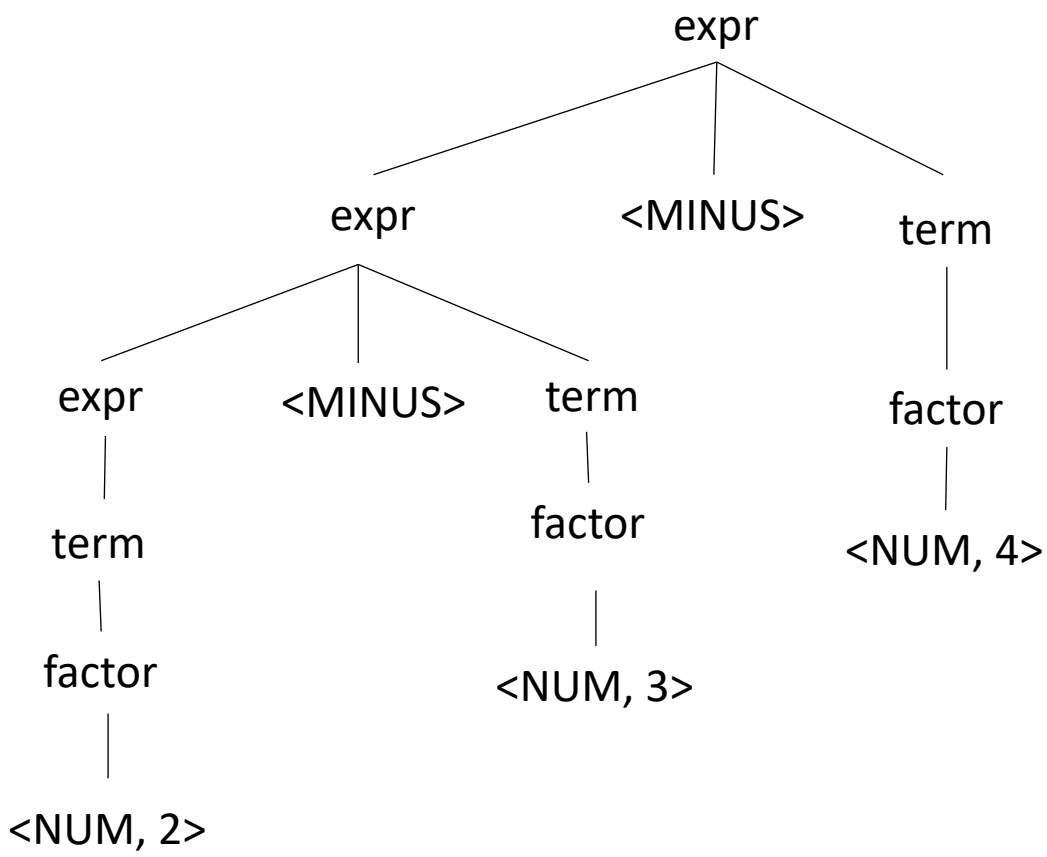


Cannot derive this parse tree!

Are we finished?

Operator	Name	Productions
+,-	Expr	: Expr + Term | Expr - Term | Term
*,/	Term	: Term * Factor : Term / Factor | Factor
()	Factor	: (Expr) | NUM

`input: 2-3-4`

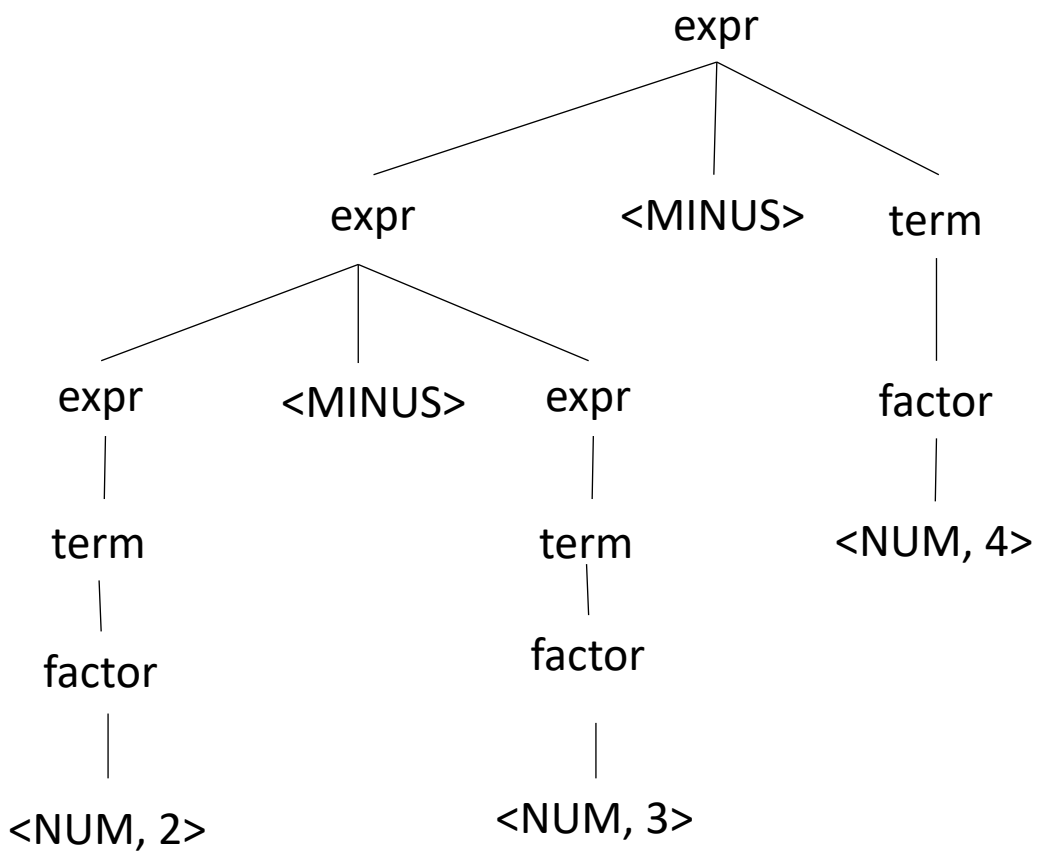


Are we finished?

Operator	Name	Productions
+,-	Expr	: Expr + Term | Expr - Term | Term
*,/	Term	: Term * Factor : Term / Factor | Factor
()	Factor	: (Expr) | NUM

`input: 2-3-4`

This parse tree works!



How are parsers implemented?

- Many different ways: read chapter 3 in EAC

- Most likely you can use a parser generator
  - write production rules and tokens in a DSL or decorator
  - generator automatically creates a parser for you

Parsing actions

- Each production rule gets an action.
- actions are executed in a post-order traversal
- actions can return a value to their parent
  - actions can assume their children have executed
- Sometimes called “ad hoc syntax-directed translation”
  - Chapter 4 of EAC

Building an interpreter

- Consider the production rules for expressions:

Building an interpreter

Operator	Name	Productions	Action

+,-	Expr	: Expr + Term
		| Expr - Term
		| Term
			`{return e[0] + e[2]}`
			`{return e[0] – e[2]}`
			`{return e[0]}`

*,/	Term	: Term * Factor
		: Term / Factor
		| Factor
			`{return e[0] * e[2]}`
			`{return e[0] / e[2]}`
			`{return e[0]}`

()	Factor	: (Expr) | NUM	`{return e[1]}` `{return int(e[0])}`

Consider a struct `e` that contains elements for each of the children

Next week

- Homework overview
- Implementing a parser in PLY
  - Implementing a simple interpreter
- Parsing regular expressions with derivatives
- Have a good weekend!