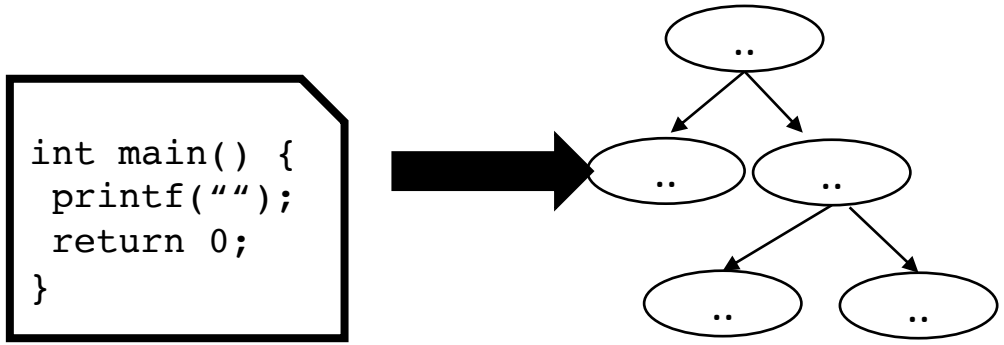
CSE211: Compiler Design

Sept. 29, 2021



- **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?*

Announcements:

- Quiz results are in!
- Slack wins, there is a link to join
  - Official communication will occur through canvas
  - Private communication will occur through canvas
  - Discussions can happen on slack
  - Keep it open during class?
- Any issues so far?
  - Accessing text book
  - Slides
- Homework 1 will be assigned in 1 week!
  - In the meantime, make sure you can get docker working and let me know if you want any software installed

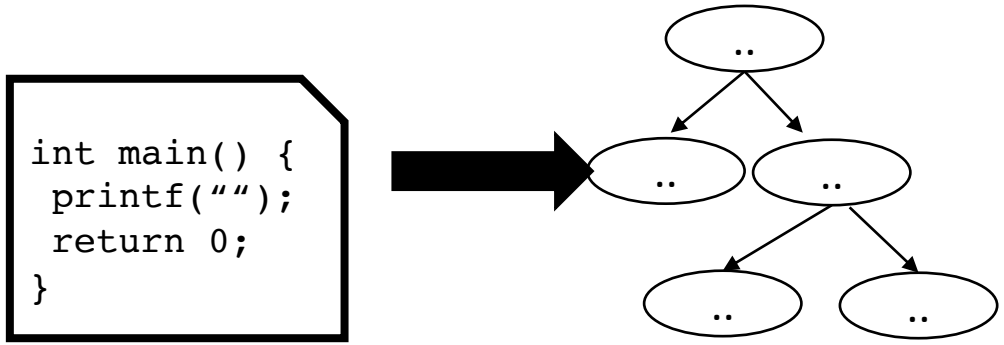
Announcements:

- I think there is a class in here afterwards
  - I can stay after class outside

- Office hours tomorrow:
  - 2 - 3pm
  - E2 - 233 (no name tag yet!)

CSE211: Compiler Design

Sept. 29, 2021



- **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?*

Refresher

Regular expressions:

- 3 primitive operations
  - union
  - concat
  - Kleene star

- Precedence?

- Common additional operators?

Refresher

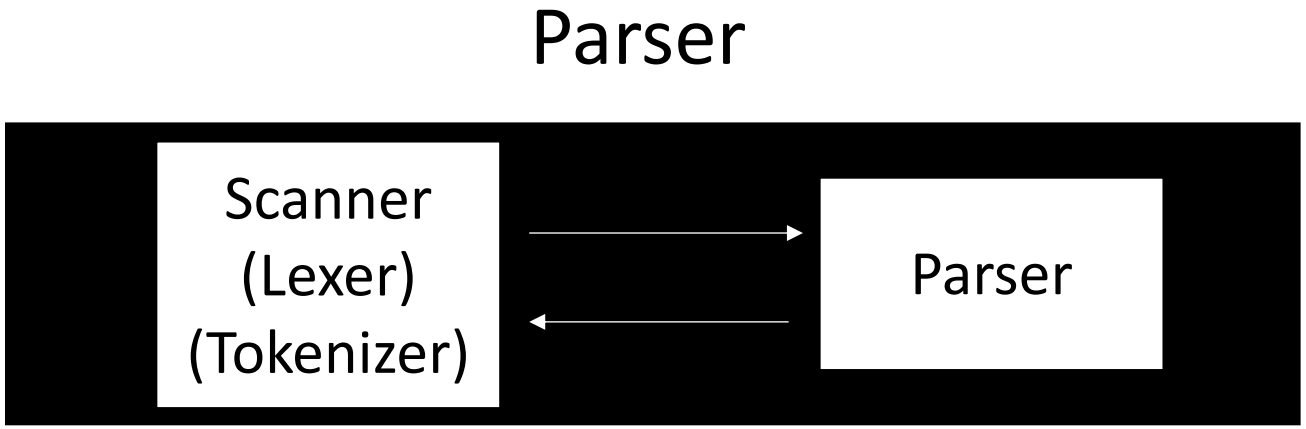
Exercise:

C-style ids

floating point number

Email addresses

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)

Lists of Tokens

- Main idea:
  - We can construct languages out of tokens

ARTICLE ADJECTIVE NOUN VERB

Lists of Tokens

- Main idea:
  - We can construct languages out of tokens

ARTICLE ADJECTIVE NOUN VERB

My Old Computer Crashed

Scanner

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

Lists of Tokens

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

- What about a mathematical sentence (expression)?

Lists of Tokens

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

- What about a mathematical sentence (expression)?

- First lets define tokens:

Lists of 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 = ’*’

Lists of 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?

Lists of 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?
  - NUM

Lists of 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?
  - NUM
  - NUM PLUS NUM

Lists of 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?
  - NUM
  - NUM PLUS NUM
  - …

Lists of 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 = ’*’

Why not just use regular expressions?

What would the expression look like?

- What should our language look like?
  - NUM
  - NUM PLUS NUM
  - …

Lists of Tokens

- Where are we going to run into issues?

What about ()'s

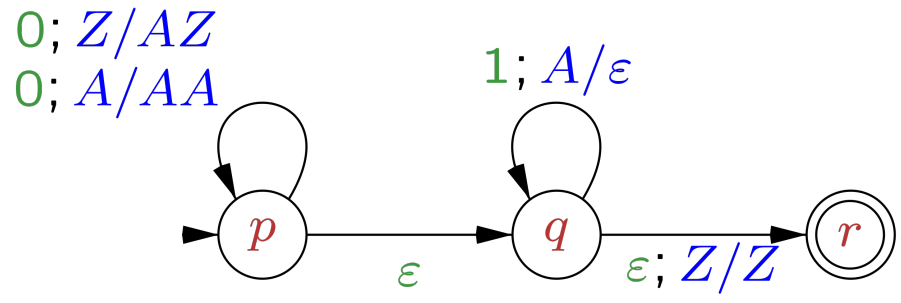
- there is a formal proof available that regex CANNOT match ()'s: pumping lemma

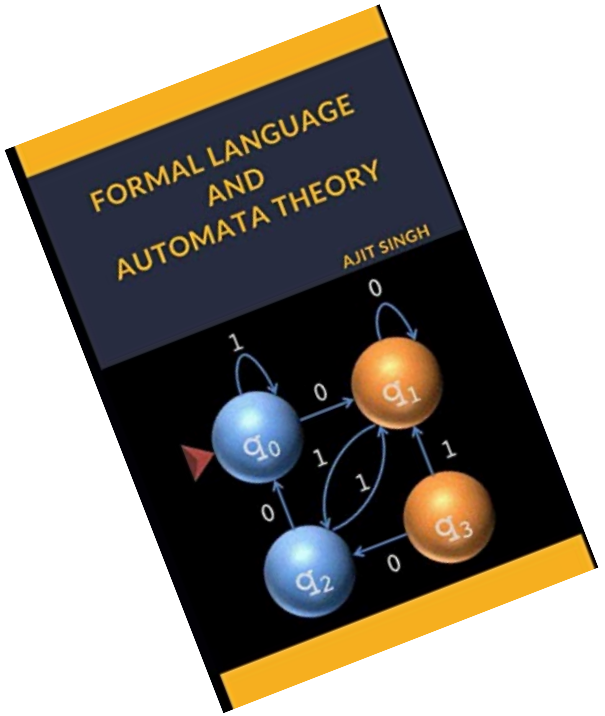
- Informal argument:
  - Try matching $(^n)^n$ using Kleene star
  - Impossible!

- 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





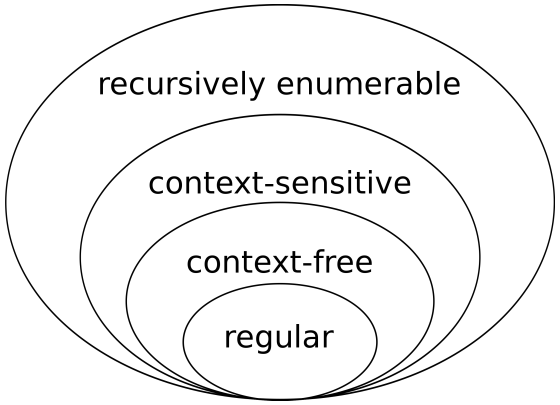


Image Credit:
By Jochgem - Own work, CC BY-SA 3.0,
https://commons.wikimedia.org/w/index.php?curid=5036988

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?

```
expr : NUM
       | NUM bin_op expr

bin_op : PLUS | TIMES
```

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 = ’*’
  - LP = ‘\(’
  - RP = \)’

Let’s add () to the language!

expression : NUM
| expression PLUS expression
| expression TIMES expression
| LP expression RP

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

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

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

2nd most upvoted post on stackoverflow

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

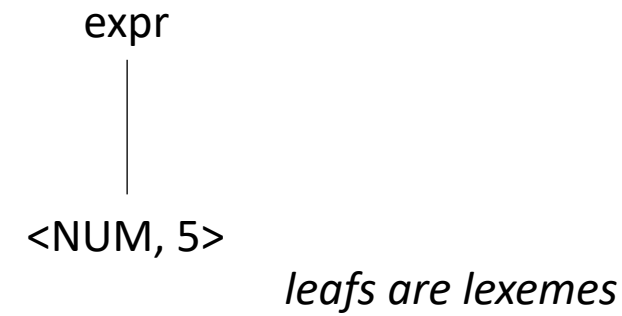
root of the tree is the entry production

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
```

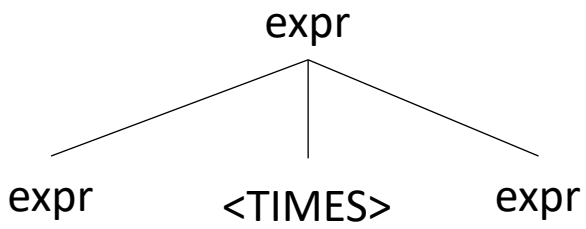
expr

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
```

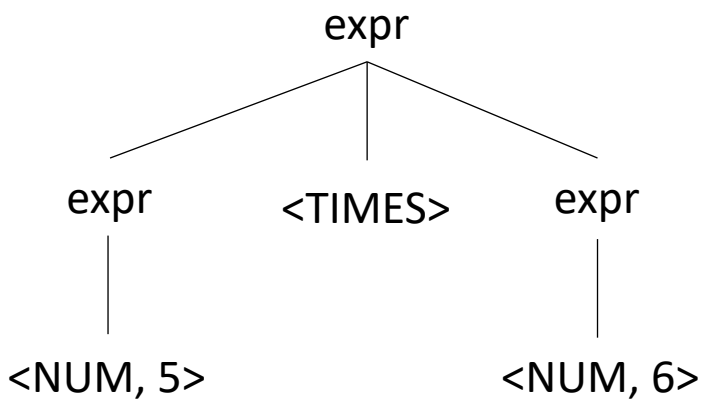


Parse trees

- 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

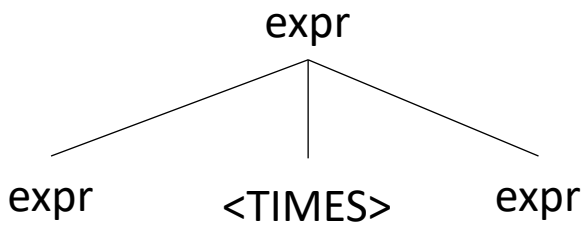
Parse trees

- 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?



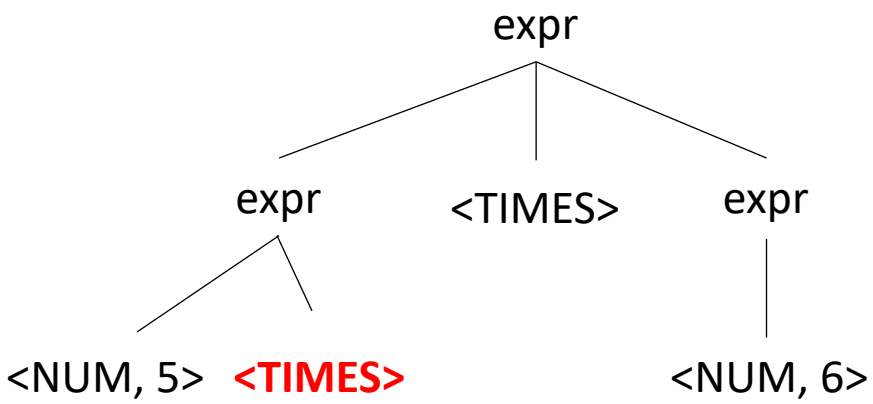
Parse trees

- 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

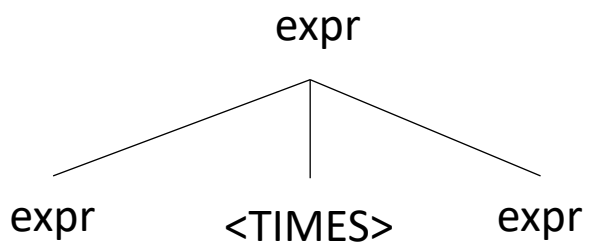
expr

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
```

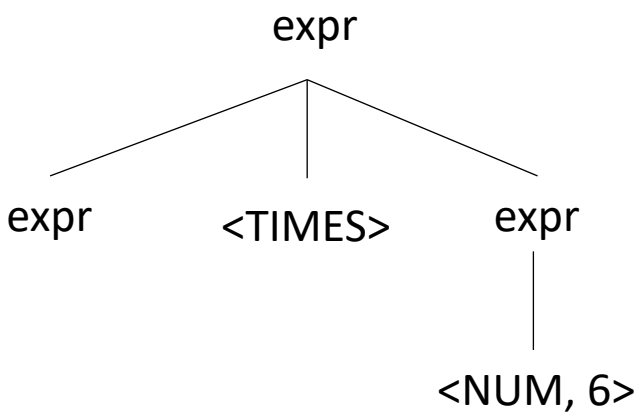


Parse trees

- 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
```

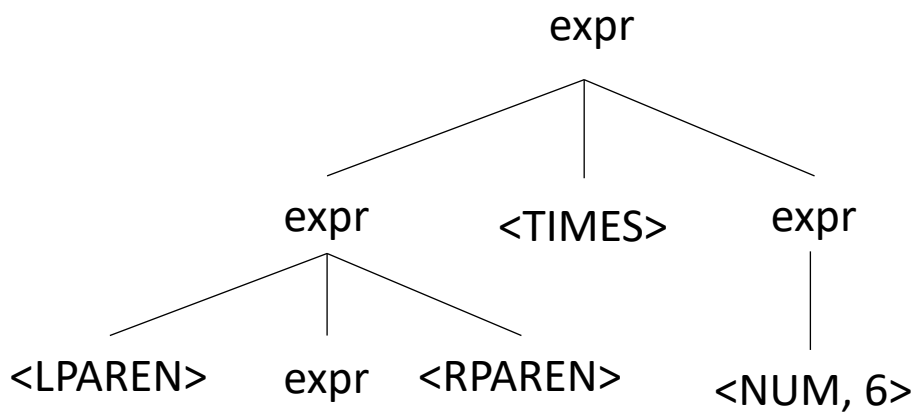


Parse trees

- 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
```

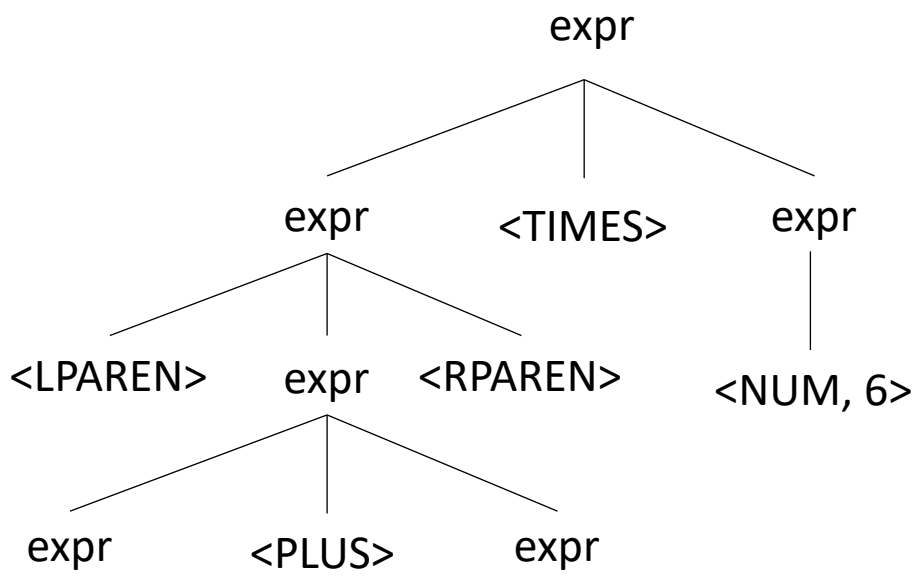


Parse trees

- 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
```

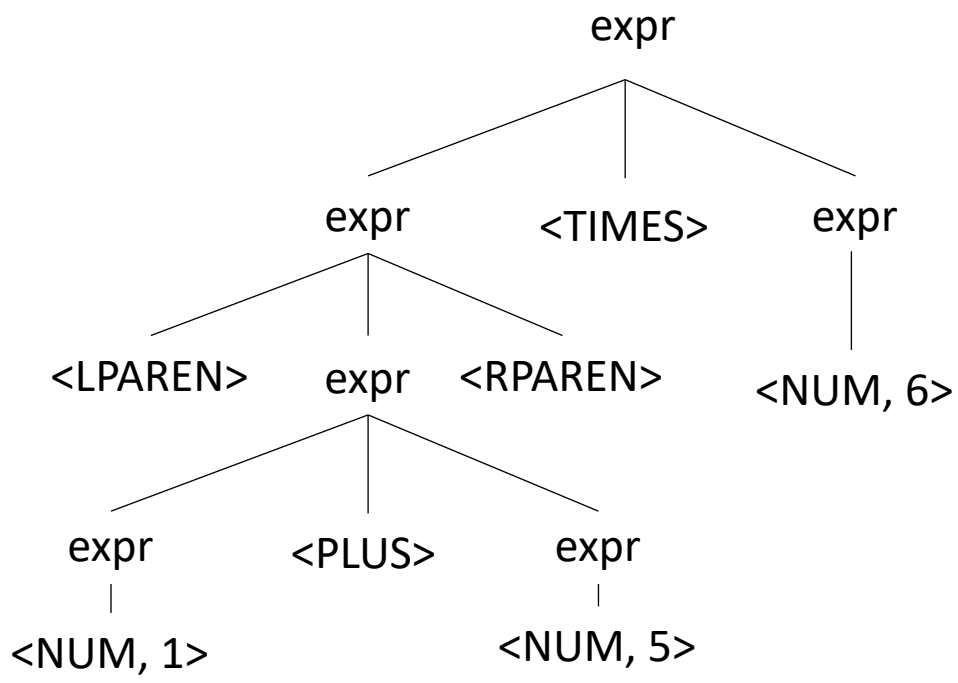


Parse trees

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

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

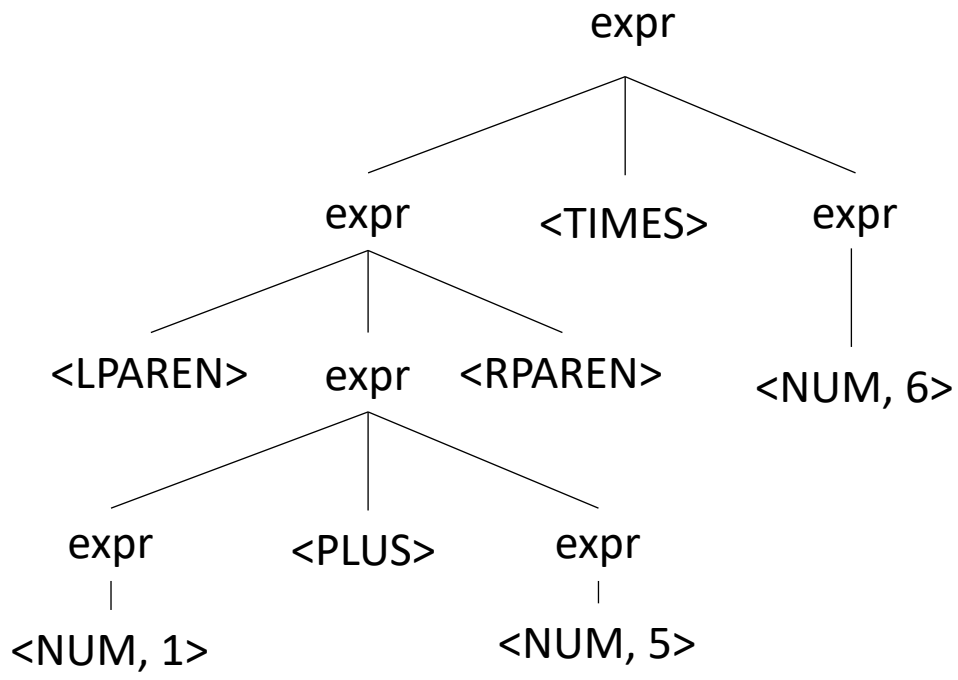
```
expr : NUM
     | expr PLUS expr
     | expr TIMES expr
     | LPAREN expr RPAREN
```



Parse trees

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

```
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

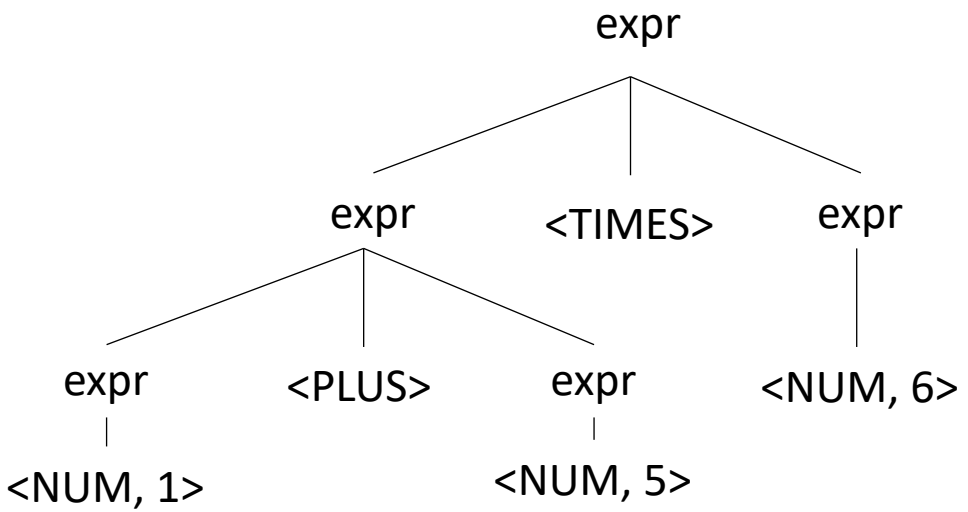
| expr TIMES expr

| LPAREN expr RPAREN

Parse trees

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

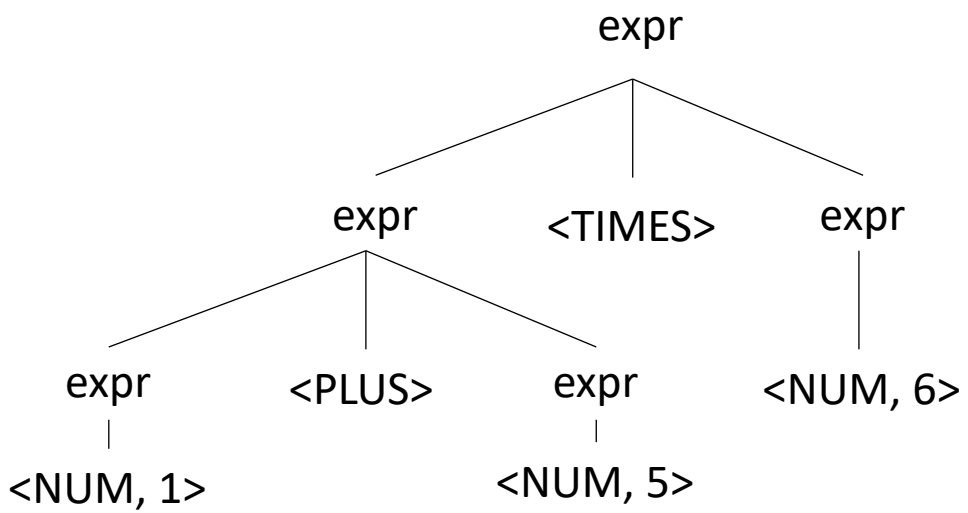
```
expr : NUM
     | expr PLUS expr
     | expr TIMES expr
     | LPAREN expr RPAREN
```



Ambiguous grammars

- `input: 1 + 5 * 6`

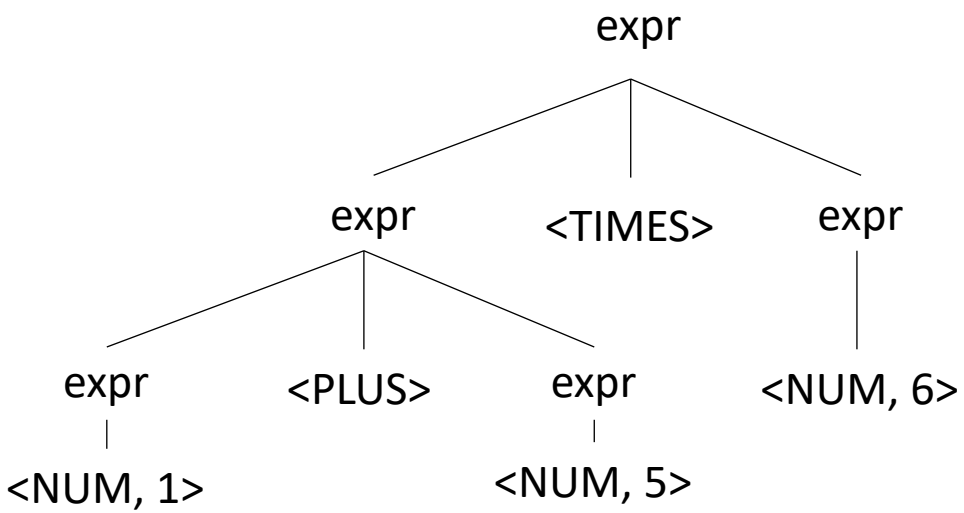
```
expr : NUM
     | expr PLUS expr
     | expr TIMES expr
     | LPAREN expr RPAREN
```

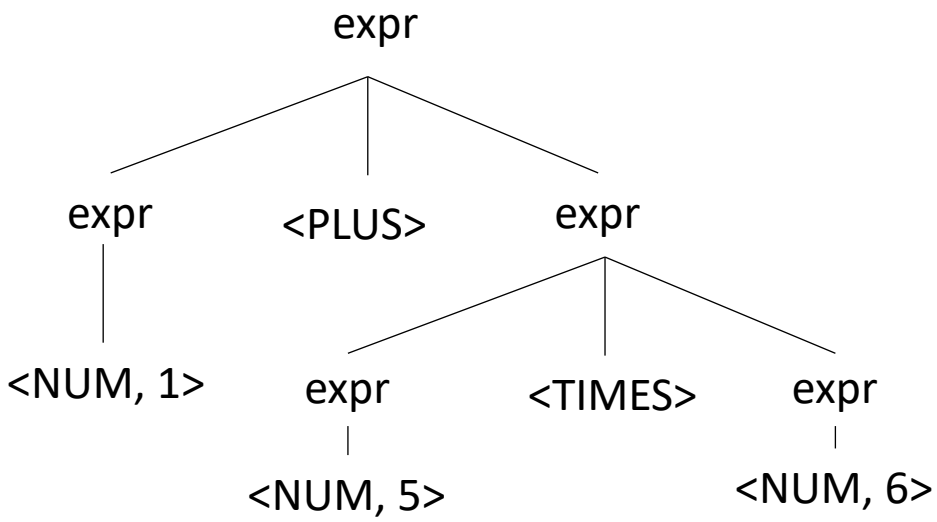


Ambiguous grammars

- `input: 1 + 5 * 6`

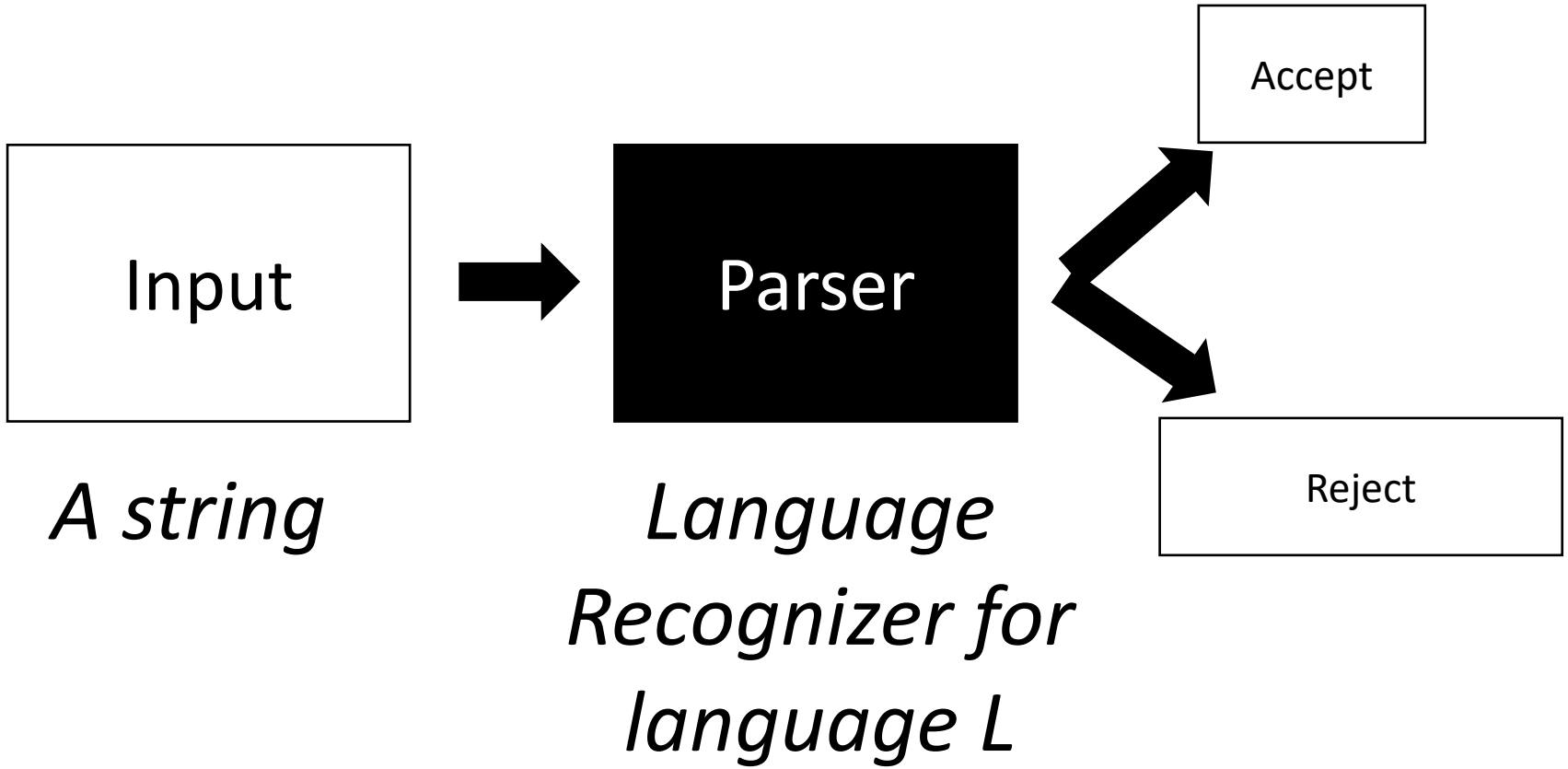
```
expr : NUM
     | expr PLUS expr
     | expr TIMES expr
     | LPAREN expr RPAREN
```





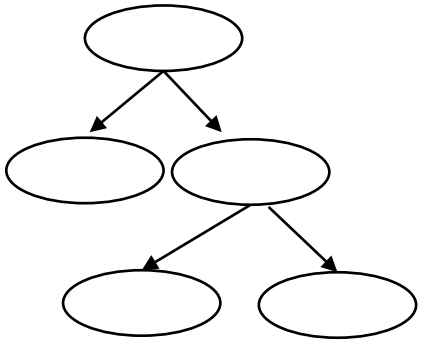
Ambiguous grammars

- What's the issue?

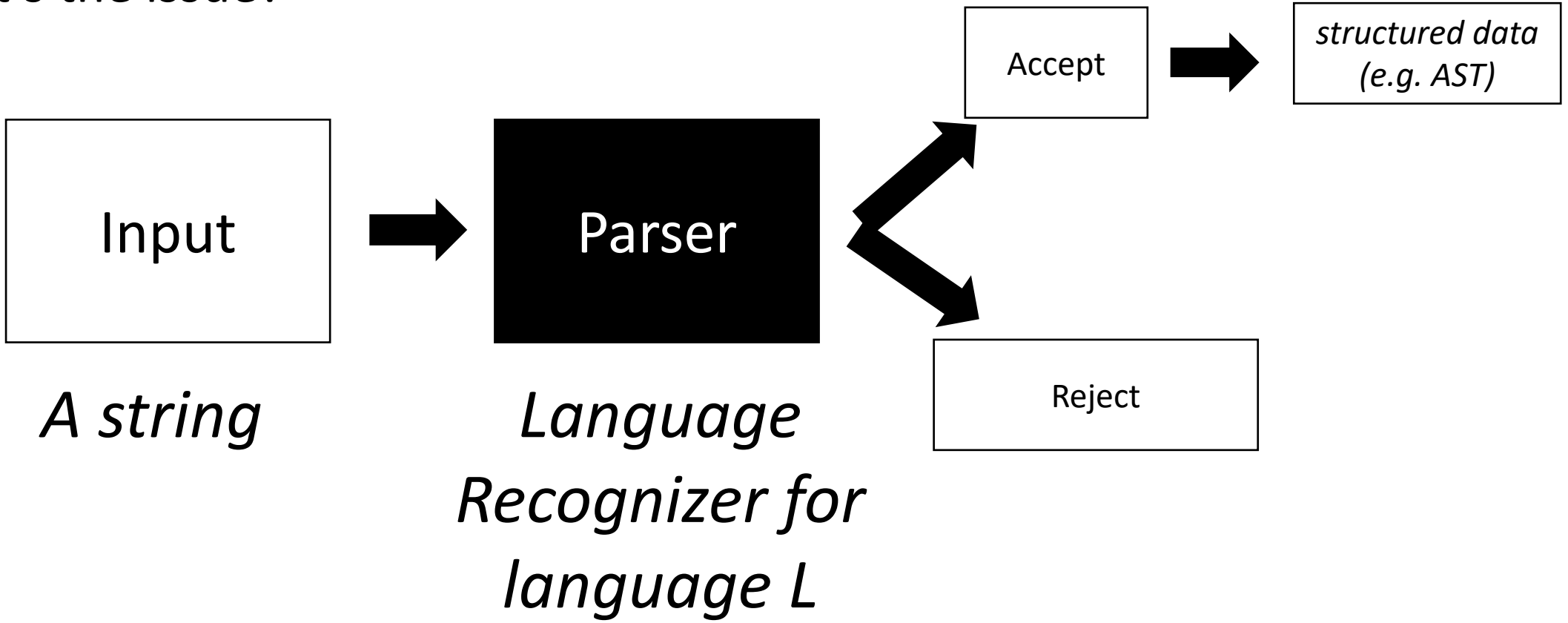


Ambiguous grammars

continue to the rest of compilation



- What's the issue?



Meaning into structure

- Structural meaning defined to be a post-order traversal

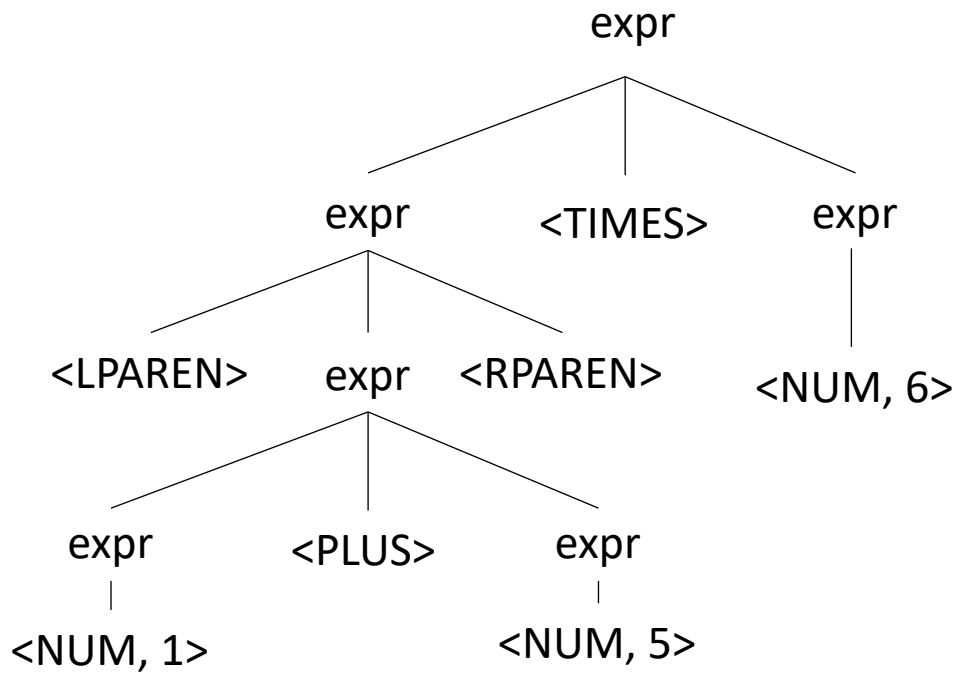
Meaning into structure

- 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
  - Also called “Natural Order”

Parse trees

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

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



Meaning into structure

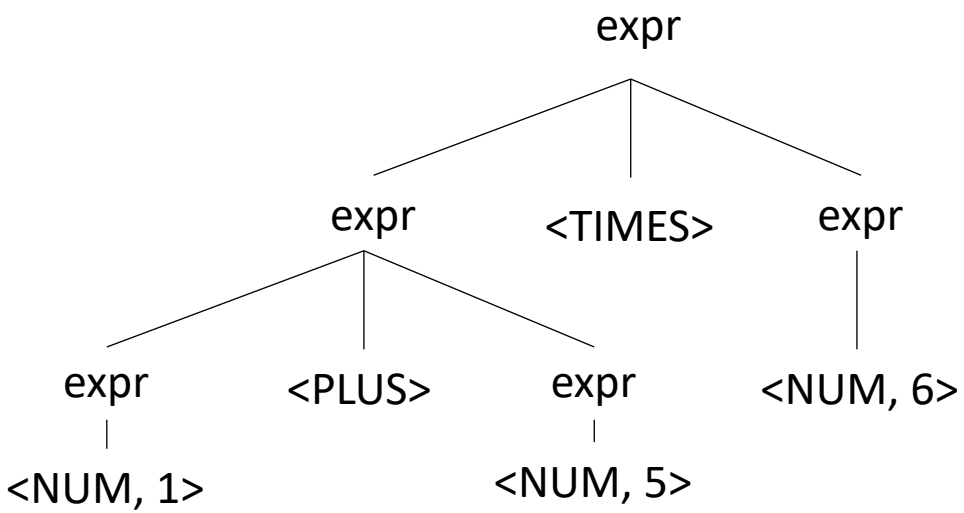
- Structural meaning defined to be a post-order traversal
  - Children return values to their parent
  - Nodes are only evaluated once all their children have been evaluated
  - Evaluated from left to right

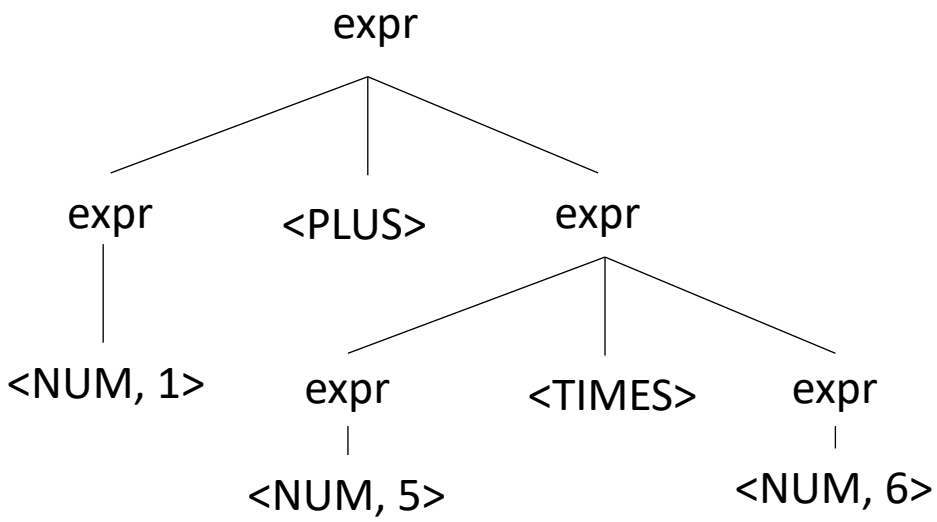
- Can also encode the order of operation

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
```

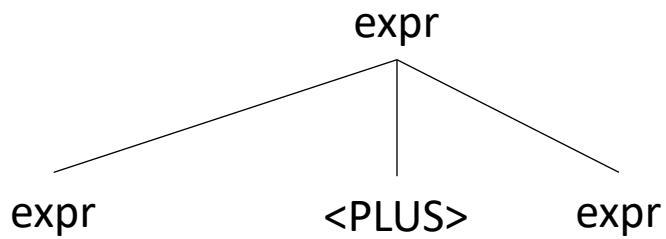
expr

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
```

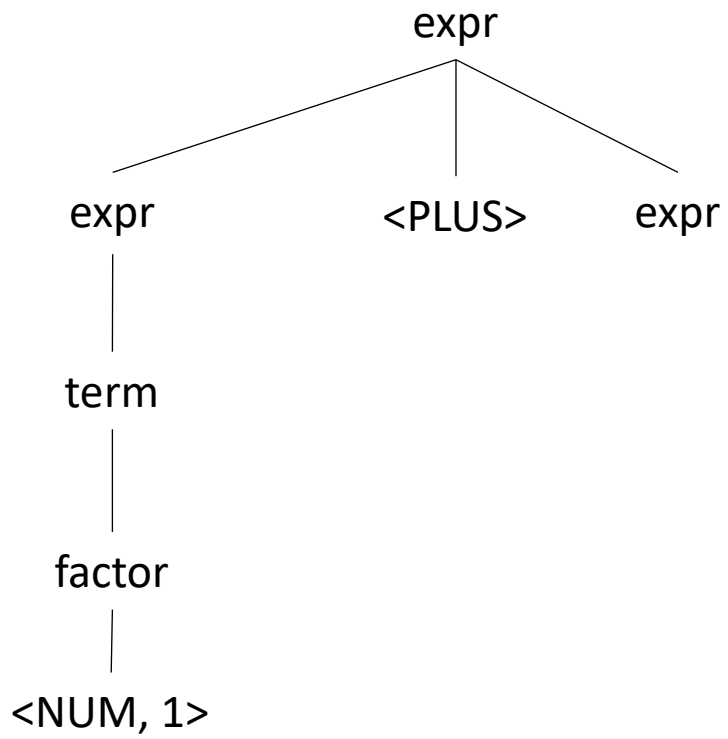
Operator	Name	Productions
+	expr	: expr PLUS expr | term
*	term	: term TIMES term | factor
()	factor	: LPAREN expr RPAREN | NUM



Now lets create a parse tree

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

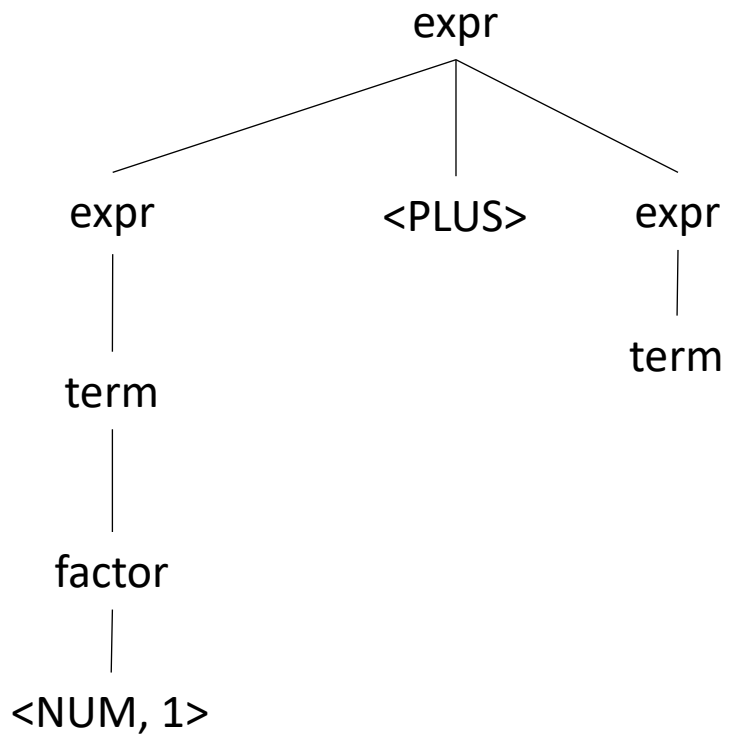
Operator	Name	Productions
+	expr	: expr PLUS expr | term
*	term	: term TIMES term | factor
()	factor	: LPAREN expr RPAREN | NUM



Now lets create a parse tree

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

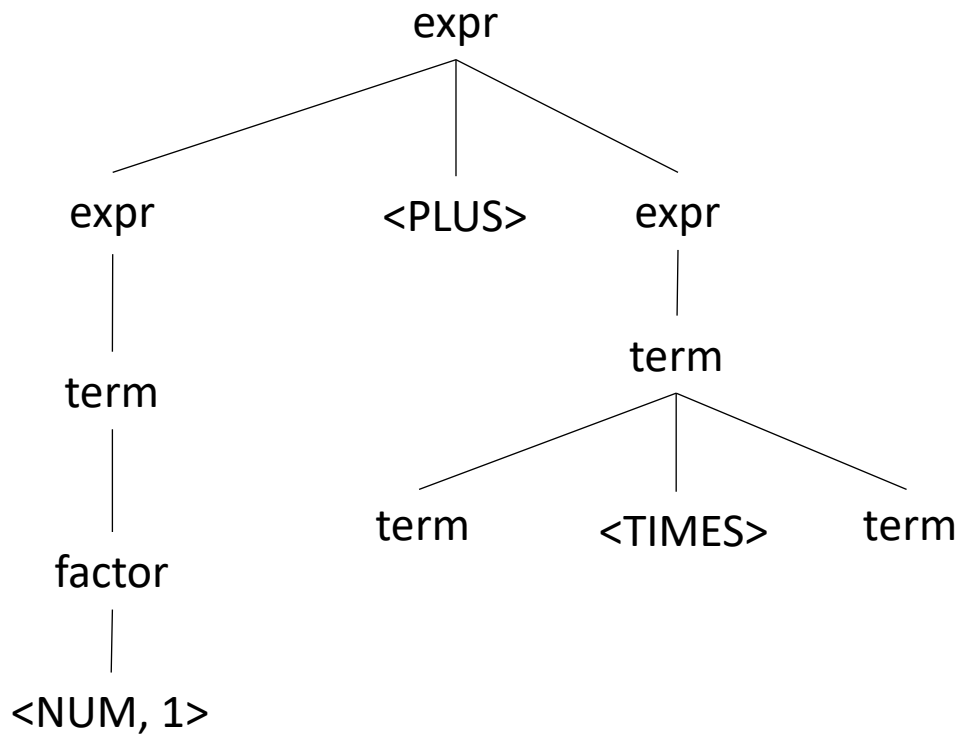
Operator	Name	Productions
+	expr	: expr PLUS expr | term
*	term	: term TIMES term | factor
()	factor	: LPAREN expr RPAREN | NUM



Now lets create a parse tree

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

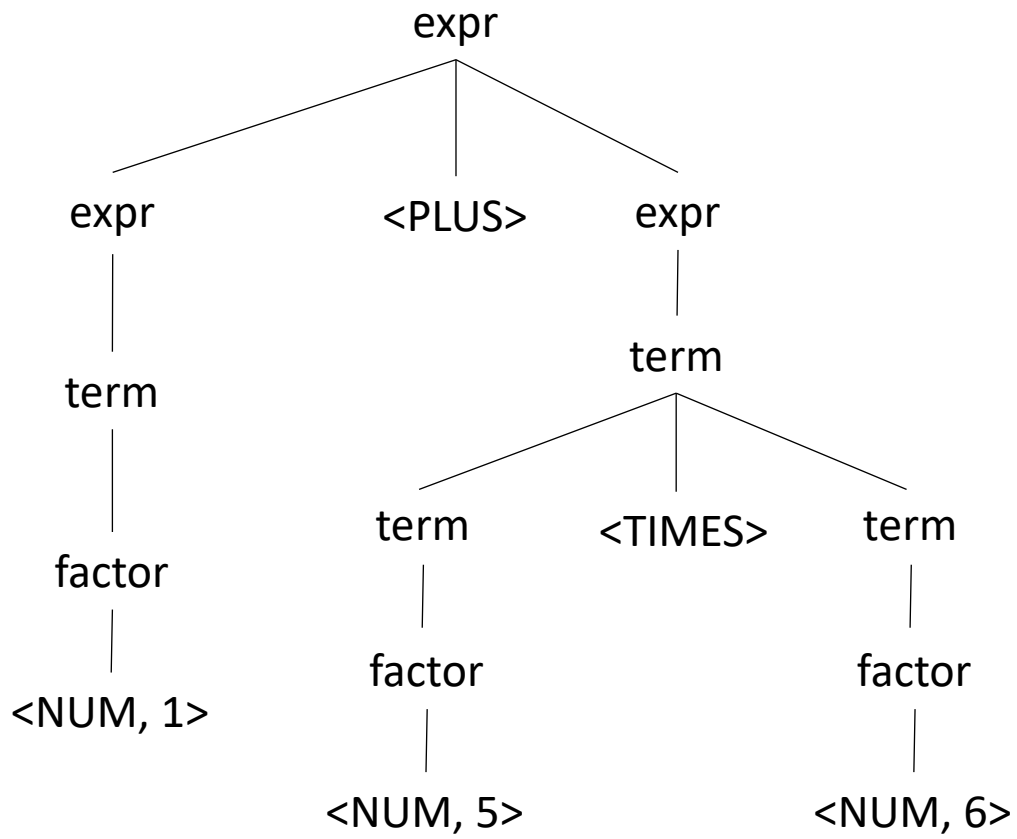
Operator	Name	Productions
+	expr	: expr PLUS expr | term
*	term	: term TIMES term | factor
()	factor	: LPAREN expr RPAREN | NUM



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



Parsing REs

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

Operator	Name	Productions
|	p0	p0 PIPE p0 p1
.	p1	p1 DOT p1 p2
*	p2	p2 STAR p3
()	p3	LPAR p0 RPAR 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

Parsing REs

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

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

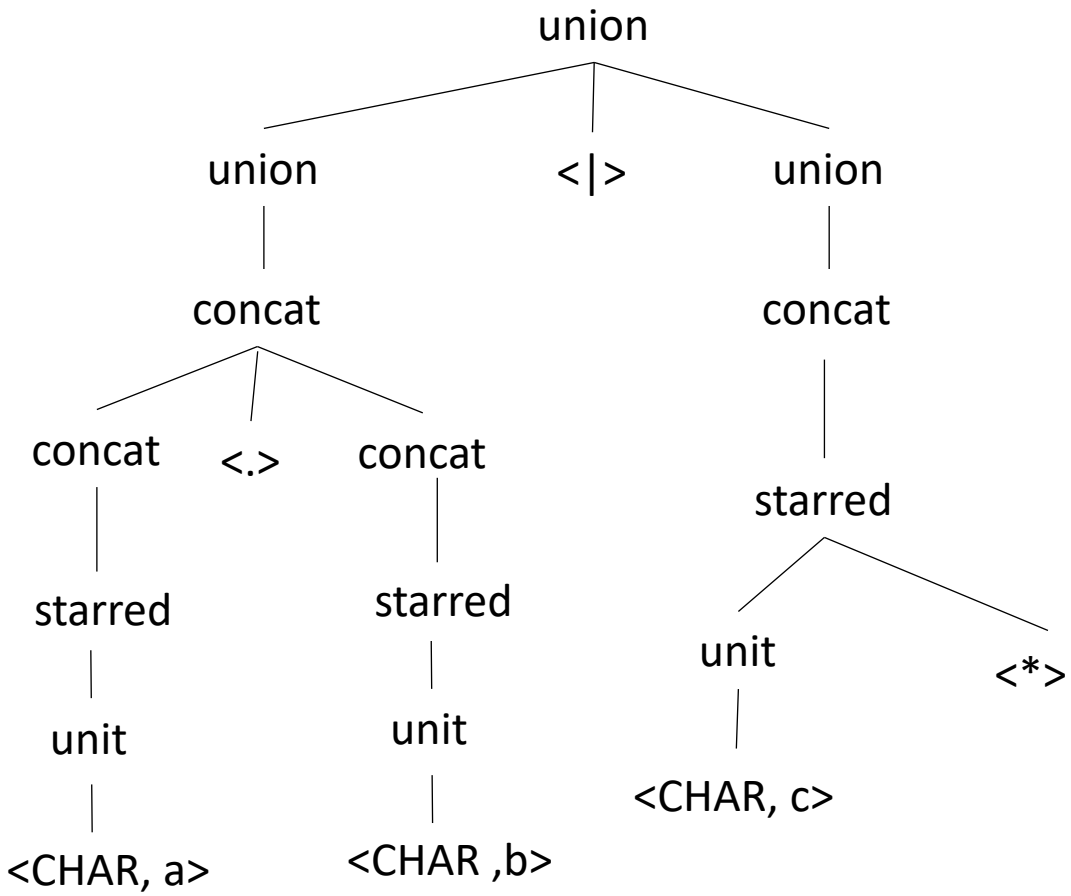
Operator	Name	Productions
|	union	: union PIPE union | concat
.	concat	: concat DOT concat | starred
*	starred	: starred STAR | unit
()	unit	: LPAREN union RPAREN | CHAR

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*
```



Next class

- Chapter 3 in EAC goes into detail on parsers
  - Some parsing algorithms, ambiguous grammars, etc.
- Encoding associativity into production rules
- For you:
  - Try out docker instructions!
  - Join slack for discussions!
  - Homework is released in 1 week!
- See you on Friday!