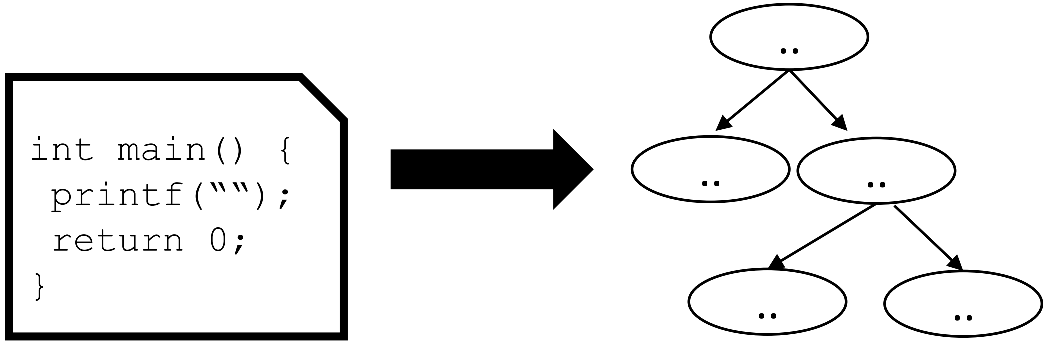
CSE211: Compiler Design

Oct. 4, 2023



- **Topic**: Parsing overview 2

- **Questions**:
  - *What is a scanner?*
  - *What are regular expressions? What are some use-cases for them?*

Announcements

- Piazza is up! Please enroll. It should be considered required!
- My office hours will be on Thursday 3 – 5 PM
  - No hours this week though
- Occasional technical issues with recordings
  - Will not be re-recording classes

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 it
- Any remaining undergrads should get a permission code ASAP
- If anyone isn’t on Canvas, please let me know

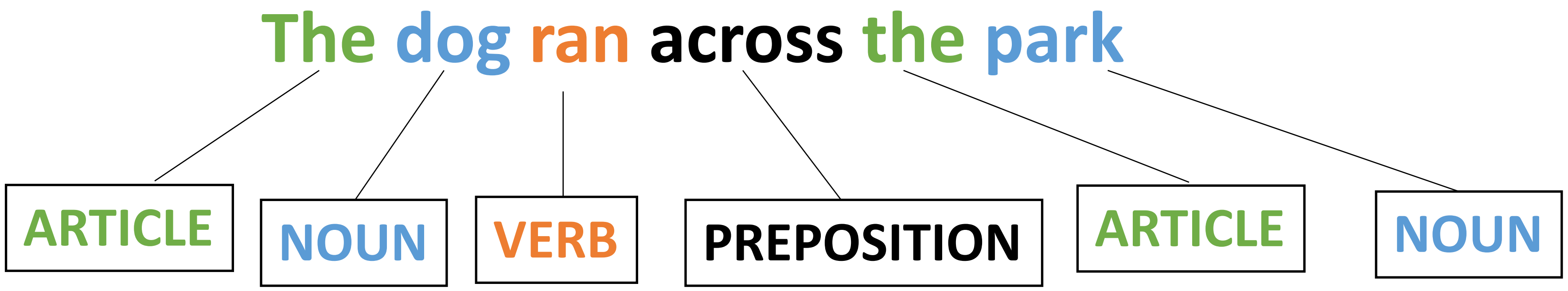
Announcements

- 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

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

- Lets write tokens for arithmetic expression:

(5 + 4) * 3

ideas?

Scanner

(5 + 4) * 3

Scanner

```
[[(LPAR, “(”)  (NUM, “5”) (PLUS, “+”) (NUM, “4”) (RPAR, “)”)
  (TIMES, “*”) (NUM, “3”)]
```

Splits an input sentence it into lexemes

Defining tokens

- Literal – single character:
  - PLUS = ‘+’, TIMES = ‘*’
- Keyword – single string:
  - IF = “if”, INT = “int”
- Sets of words:
  - NOUN = {“Cat”, “Dog”, “Car”}
- Numbers
  - NUM = {“0”, “1” …}

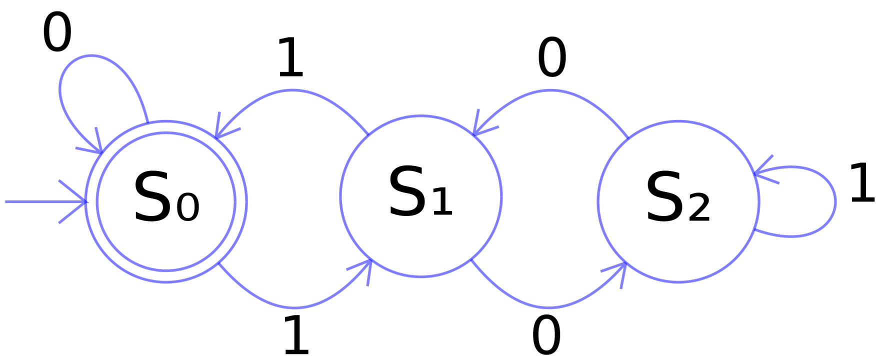
Defining tokens

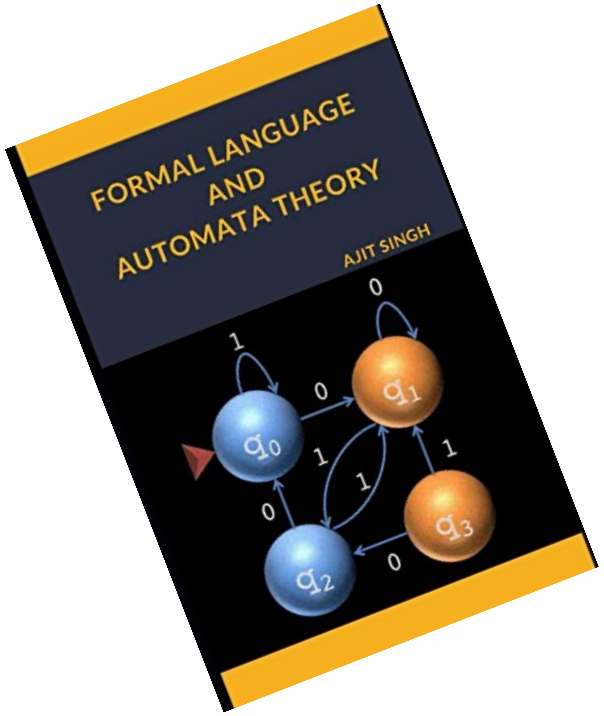
- ~~Literal – single character:~~
  - ~~PLUS = ‘+’, TIMES = ‘*’~~
  -
- ~~Keyword – single string:~~
  - ~~IF = “if”, INT = “int”~~
  -
- ~~Sets of words:~~
  - ~~NOUN = {“Cat”, “Dog”, “Car”}~~
  -
- ~~Numbers~~
  - ~~NUM = {“0”, “1” …}~~

- Regular expressions!

Regular Expressions

- Lots of literature!
  - Simplest grammar in the Chomsky language hierarchy
  - abstract machine definition (finite automata)
  - Many implementations (e.g. Python standard library)





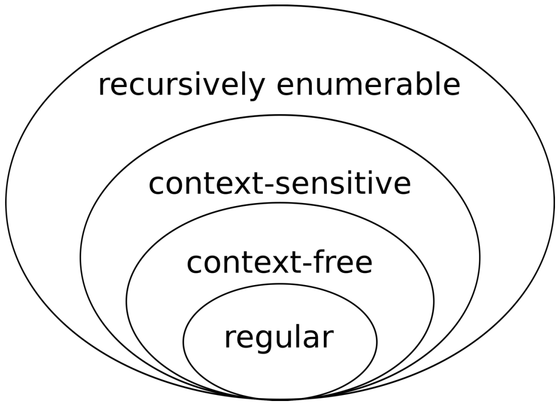


image source: wikipedia

Regular Expressions

We will define RE's recursively:

Input:

- Regular Expression *R*
- String *S*

Output:

- Does the Regular Expression *R* match the string *S*

Regular Expressions

We will define RE’s recursively:

The base case: a character literal

- The RE for a character ‘x’ is given by ‘x’. It matches only the character ‘x’

Regular Expressions

We will define RE’s recursively:

Regular expressions are closed under concatenation:

- The concatenation of two REs x and y is given by xy and matches the strings of RE x concatenated with the strings of RE y

Regular Expressions

We will define RE’s recursively:

Regular expressions are closed under union:

- The union of two REs x and y is given by x|y and matches the strings of RE x **OR** the strings of RE y

Regular Expressions

We will define RE's recursively:

Regular expressions are closed under Kleene star:

- The Kleene star of an RE x is given by x* and matches the strings of RE x **REPEATED** 0 or more times

Regular Expressions

Examples

Regular Expressions

- Use ()’s to force precedence!

- Just like in math:
  - 3 + 4 * 5

- what is the precedence of concatenation, union, and star?
  - “x | yw”
    - Is it “(x | y)w” or “x | (yw)”
  - “xy*”
    - is it (xy)* or x(y*)

Regular Expressions

- Use ()'s to force precedence!

- Just like in math:
  - 3 + 4 * 5

- what is the precedence of concatenation, union, and star?
  - "x | yw"
    - Is it "(x | y)w" or "x | (yw)"
  - "xy*"
    - is it (xy)* or x(y*)

How can we determine precedence?

Regular Expressions

- Use ()’s to force precedence!

- Just like in math:
  - 3 + 4 * 5

- what is the precedence of concatenation, union, and star?
  - Star > Concat > Union
  - use () to avoid mistakes!

Regular Expressions

Most RE implementations provide syntactic sugar:

- Ranges:
  - [0-9]: any number between 0 and 9
  - [a-z]: any lower case character
  - [A-Z]: any upper case character

- Optional(?)
  - Matches 0 or 1 instances:
  - ab?c matches ”abc” or ”ac”
  - can be implemented as: (abc | ac)

Defining tokens using REs

- Literal – single character:
  - PLUS = ‘\+’, TIMES = ‘*’

Why the backslash characters?

- Keyword – single string:
  - IF = “if”, INT = “int”
- Sets of words:
  - NOUN = “(Cat)|(Dog)|(Car)”
- Numbers
  - SINGLE_NUM = [0-9]
  - how to do INT?
  - how to do FLOAT?

Defining tokens using REs

- Literal – single character:
  - PLUS = ‘\+’, TIMES = ‘*’
- Keyword – single string:
  - IF = “if”, INT = “int”
- Sets of words:
  - NOUN = “(Cat)|(Dog)|(Car)”
- Numbers
  - SINGLE_NUM = [0-9]
  - INT = -?([1-9][0-9]*) | 0
  - FLOAT =?

Scanner

- Takes in a list of tokens and a string and tokenizes the input

Scanner

Tokens

- ARTICLE = ”The|A|My|Your”
- NOUN = “Dog|Car|Computer”
- VERB = ”Ran|Crashed|Accelerated”
- ADJECTIVE = “Purple|Spotted|Old”

Input

“My Old Computer Crashed”

Scanner

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

Tokens are defined with Regular expressions, which are used to split up the input stream into lexemes

`re.match`

- A streaming API supported by most RE libraries
  - Only has to match part the beginning part of the string, not the entire string

re.match

- A streaming API supported by most RE libraries
  - Only has to match part the beginning part of the string, not the entire string
- CLASS_TOKEN = {“cse |211|cse211”}
- What would get matched here?: “cse211”
- (CLASS_TOKEN, ?)

Scanners should provide the longest possible match

- Important for operators, e.g. in C
- ++, +=,

how would we parse "x++;"

```
(ID, "x") (ADD, "+") (ADD, "+") (SEMI, ";")

(ID, "x") (INCREMENT, "++") (SEMI, ";")
```

We can experiment in Godbolt using the clang args:: -fsyntax-only -Xclang -dump-tokens

Subtle differences here

- RE definitions are not guaranteed to give you the longest possible match
  - OP = “+|++”, ID = ”[a-z]”
  - What will this return for “x++”
- Scanners will tokenize the string according to the token with the longest match
  - PLUS = “+”, PP = “++”, ID = ”[a-z]”
  - What will this return for “x++”
- What does this mean for you?
  - If you are implementing a scanner?
  - If you are writing tokens?

Scanner Summary

- Tokens are defined using regular expressions
- A scanner uses tokens to split a string into lexemes
- Regular expressions are good for splitting up a program into numbers, variables, operators, and structure (e.g. parenthesis and braces)
- You will get more practice using them in the homework
- Chapter 2 in EAC goes into detail on regular expression parsing
  - Finite automata etc.

Define a full language using tokens?

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

- What about a mathematical sentence (expression)?

Define a full language using tokens?

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

- What about a mathematical sentence (expression)?

- First let's define tokens:

Define a full language using tokens?

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

- What about a mathematical sentence (expression)?

- First let’s 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 let's define tokens:
  - NUM = [0-9]+
  - PLUS = '\+'
  - TIMES = '*'
- What should our language look like?

Define a full language using tokens?

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

- What about a mathematical sentence (expression)?

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

- What should our language look like?
  - NUM

Define a full language using tokens?

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

- What about a mathematical sentence (expression)?

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

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

Define a full language using tokens?

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

- What about a mathematical sentence (expression)?

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

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

Define a full language using tokens?

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

- What about a mathematical sentence (expression)?

- First let’s 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
  - …

Define a full language using 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!

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!

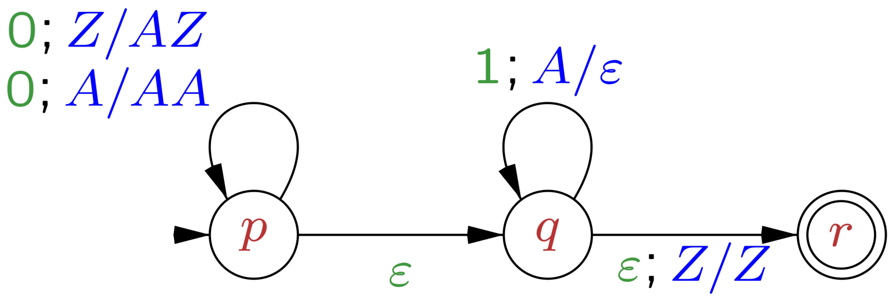
- We are going to need a more powerful language description framework!

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

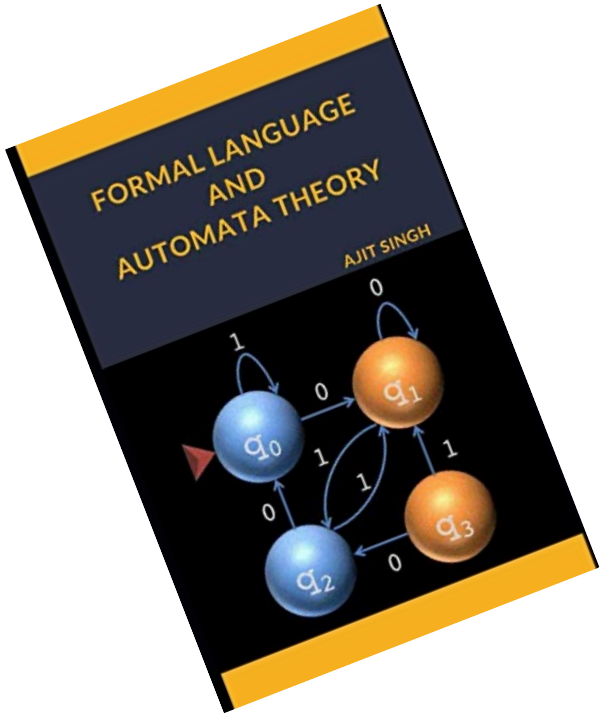
https://stackoverflow.com/questions/1732348/regex-match-open-tags-except-xhtml-self-contained-tags

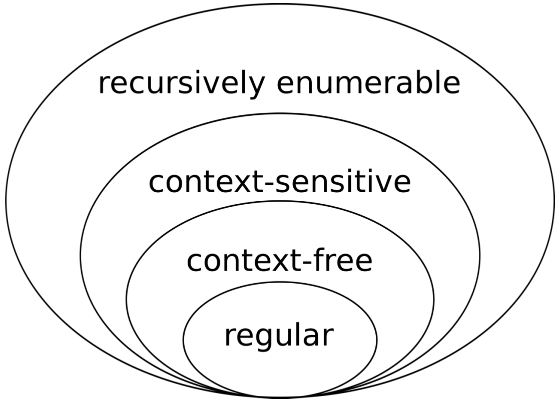
(previously) 2nd most upvoted post on stackoverflow

Context Free Grammars



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





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

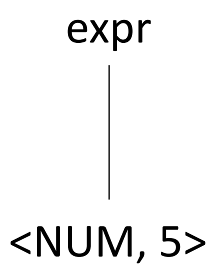
root of the tree is the entry production

Parse trees

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

`input: 5`

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



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.

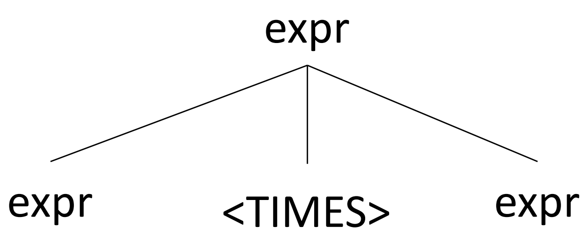
```
input: 5*6
```

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.

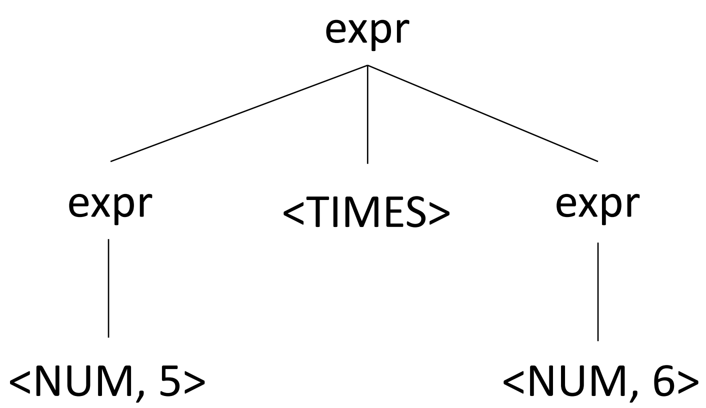
```
input: 5*6
```

expr : NUM

  | expr PLUS expr

  | expr TIMES expr

  | LPAREN expr RPAREN



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

What happens in an error?

Parse trees

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

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

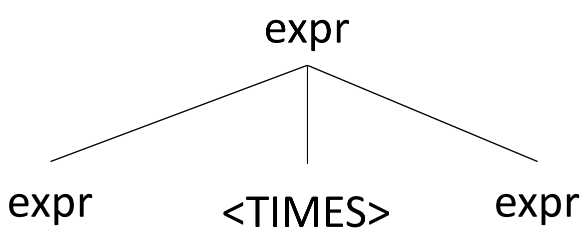
What happens in an error?

expr : NUM

| expr PLUS expr

| expr TIMES expr

| LPAREN expr RPAREN



Parse trees

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

expr : NUM

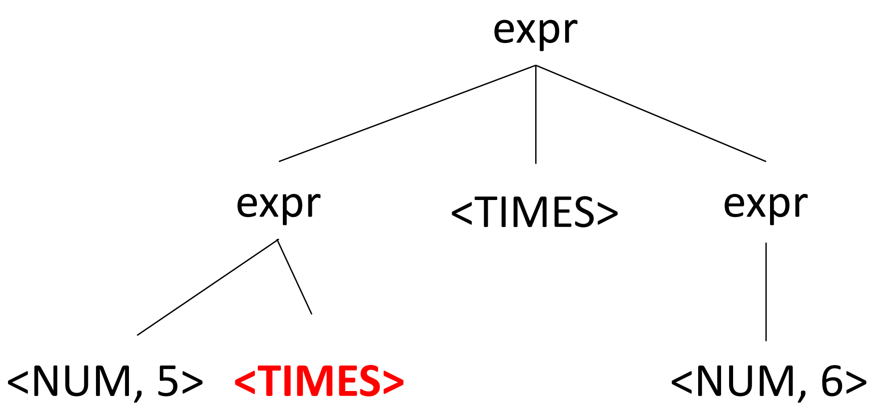
| expr PLUS expr

| 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

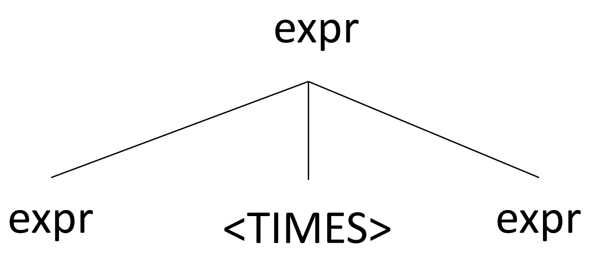
expr

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
```

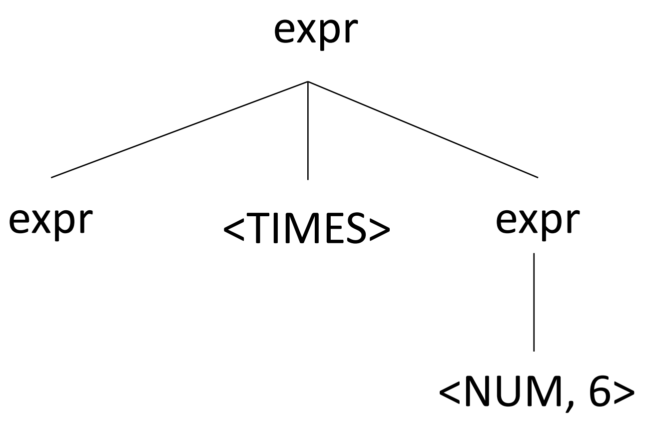


Parse trees

- 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.

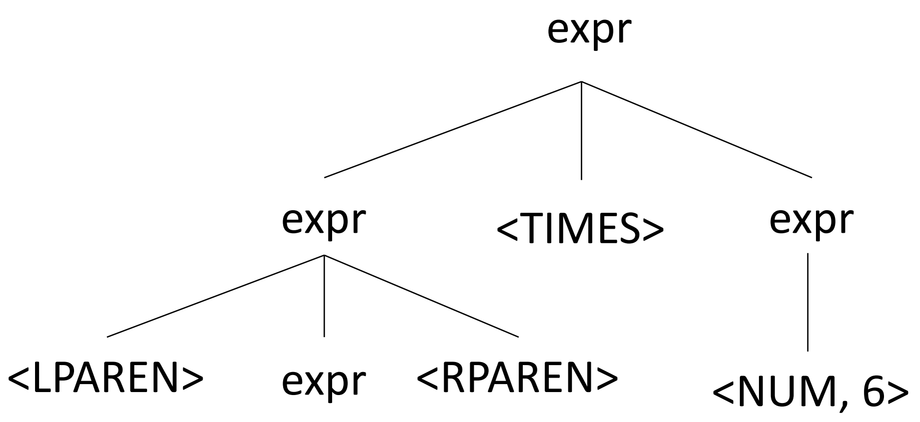
`input: (1+5)*6`

expr : NUM

| expr PLUS expr

| expr TIMES expr

| LPAREN expr RPAREN

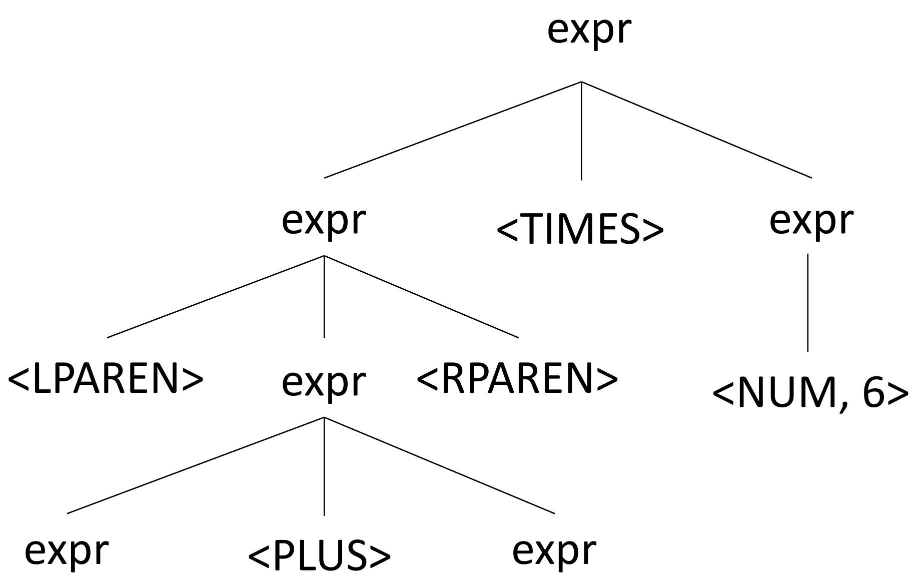


Parse trees

- 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.

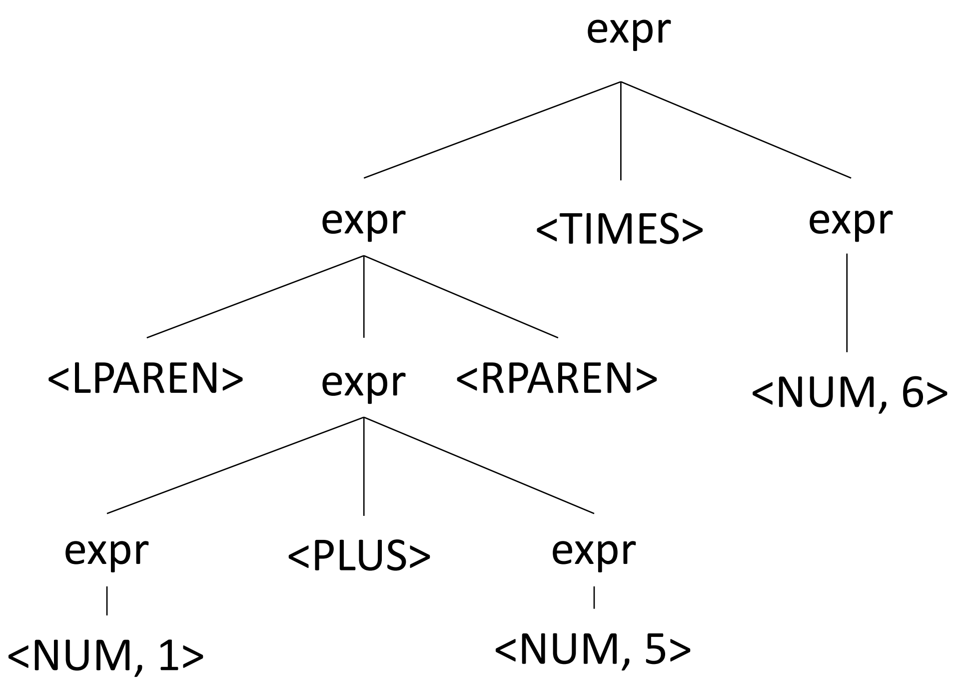
```
input: (1+5)*6
```

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?

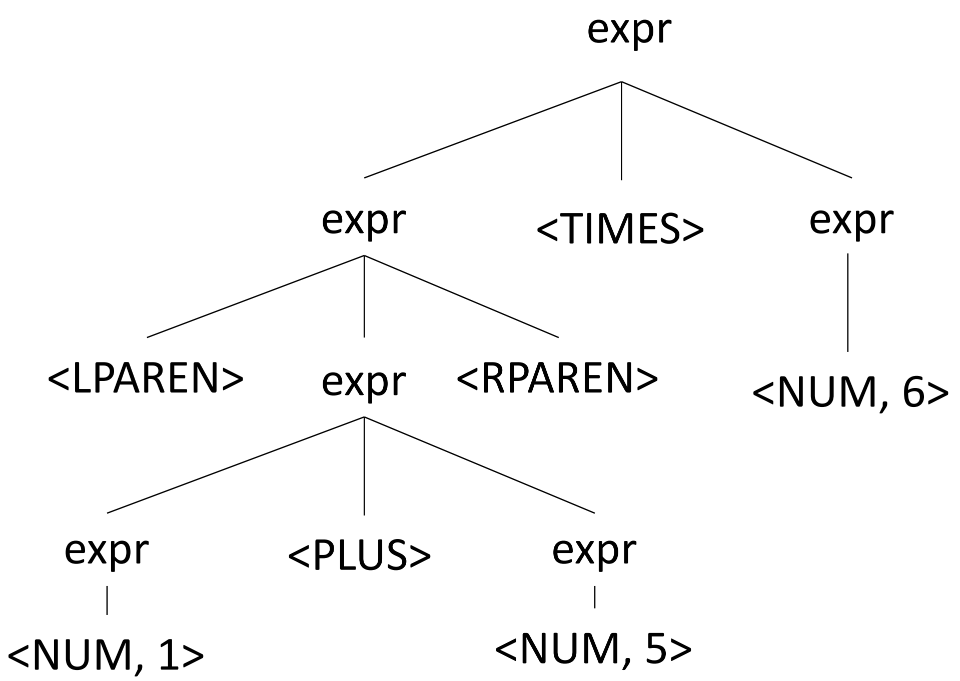
`input: ?`

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

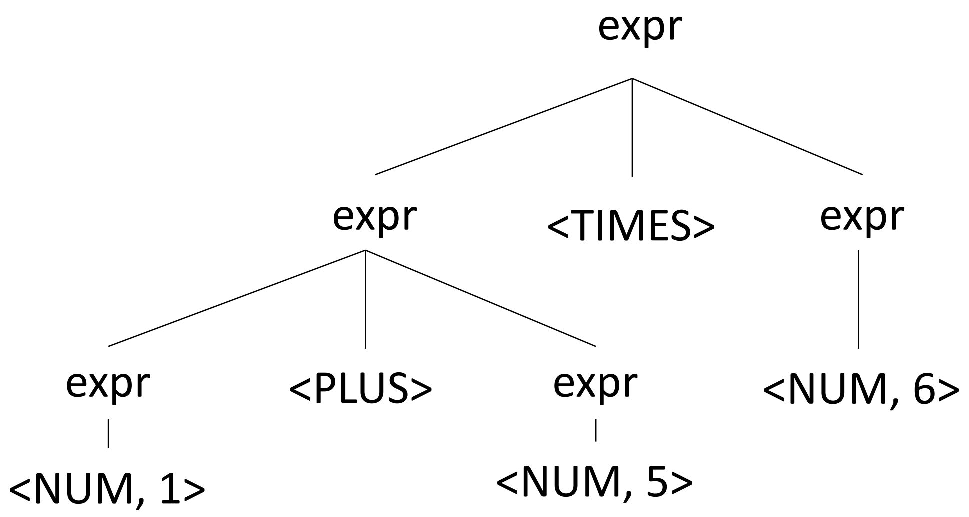
- Try making a parse tree from: `1 + 5 * 6`

```
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

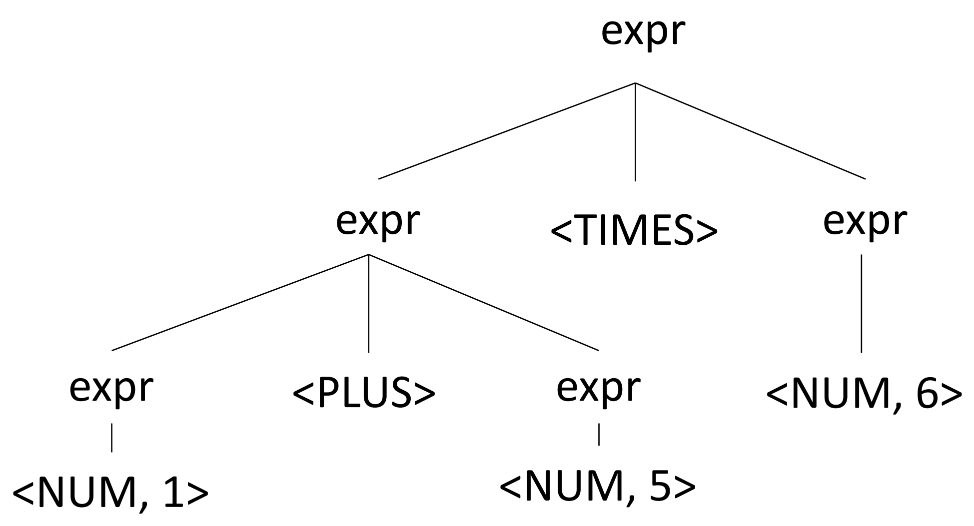
expr : NUM

| expr PLUS expr

| expr TIMES expr

| LPAREN expr RPAREN

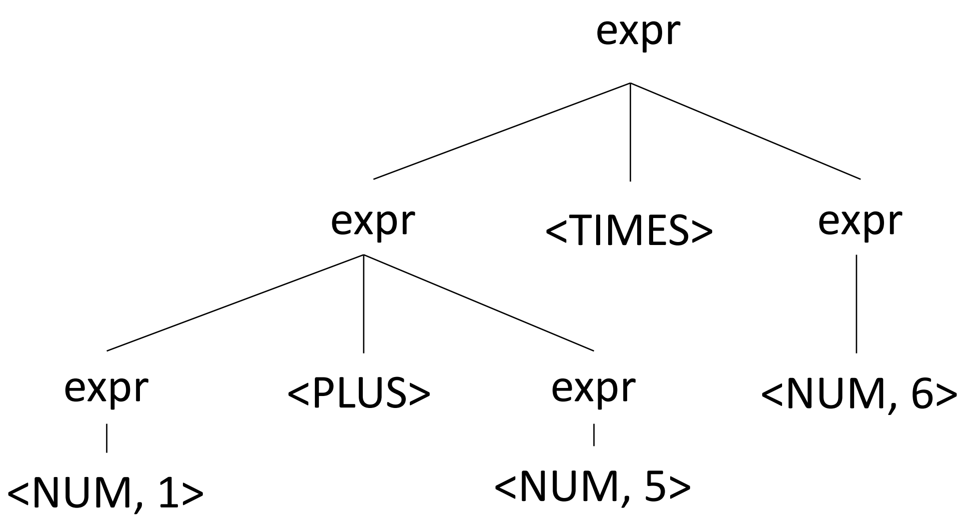
- `input: 1 + 5 * 6`

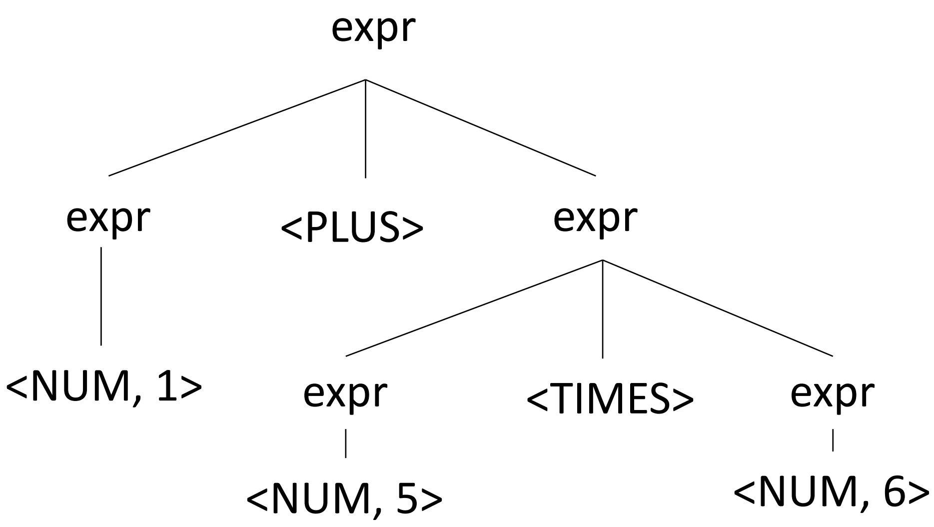


Parse trees

- `input: 1 + 5 * 6`

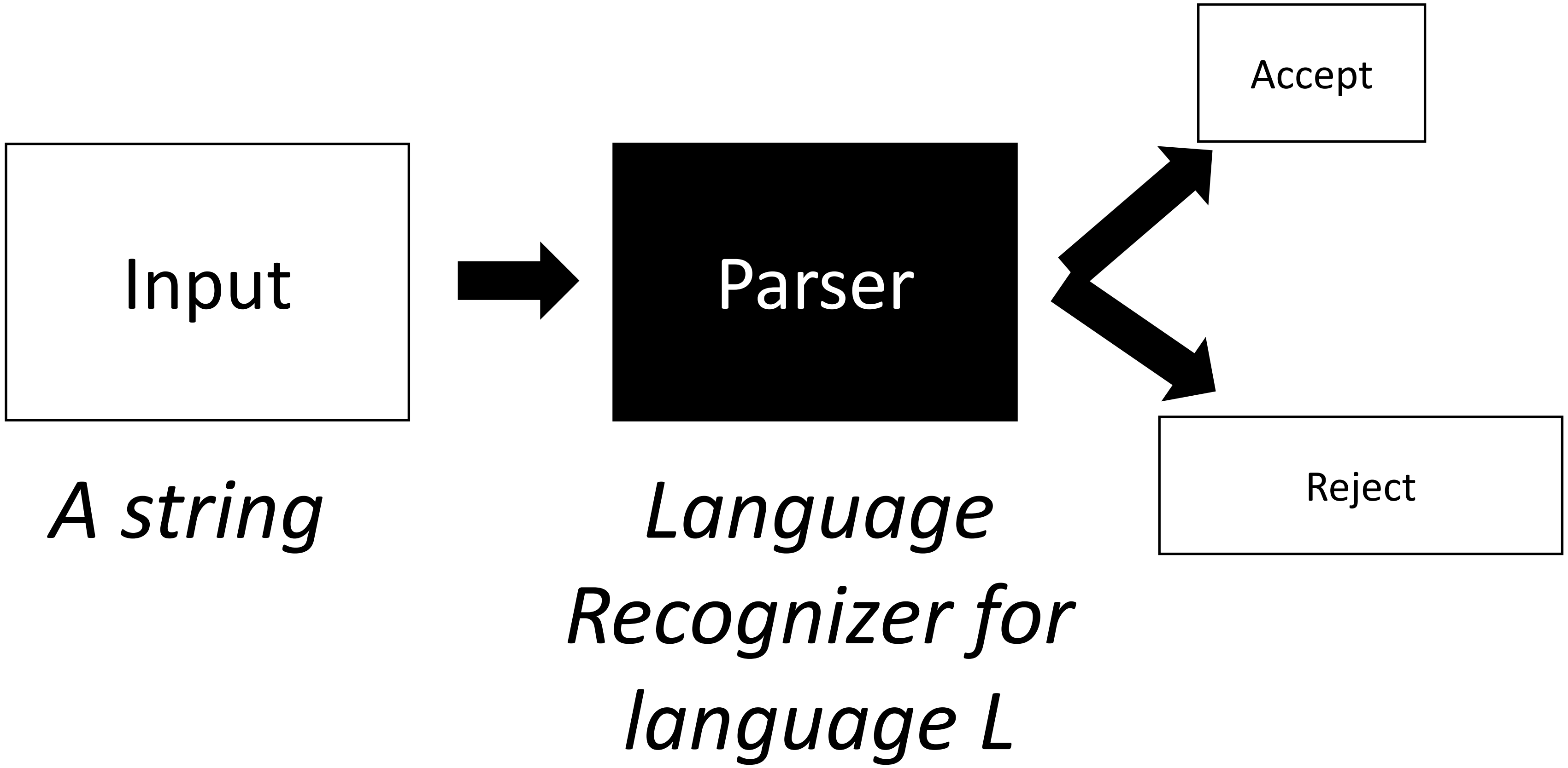
expr : NUM
| expr PLUS expr
| expr TIMES expr
| LPAREN expr RPAREN





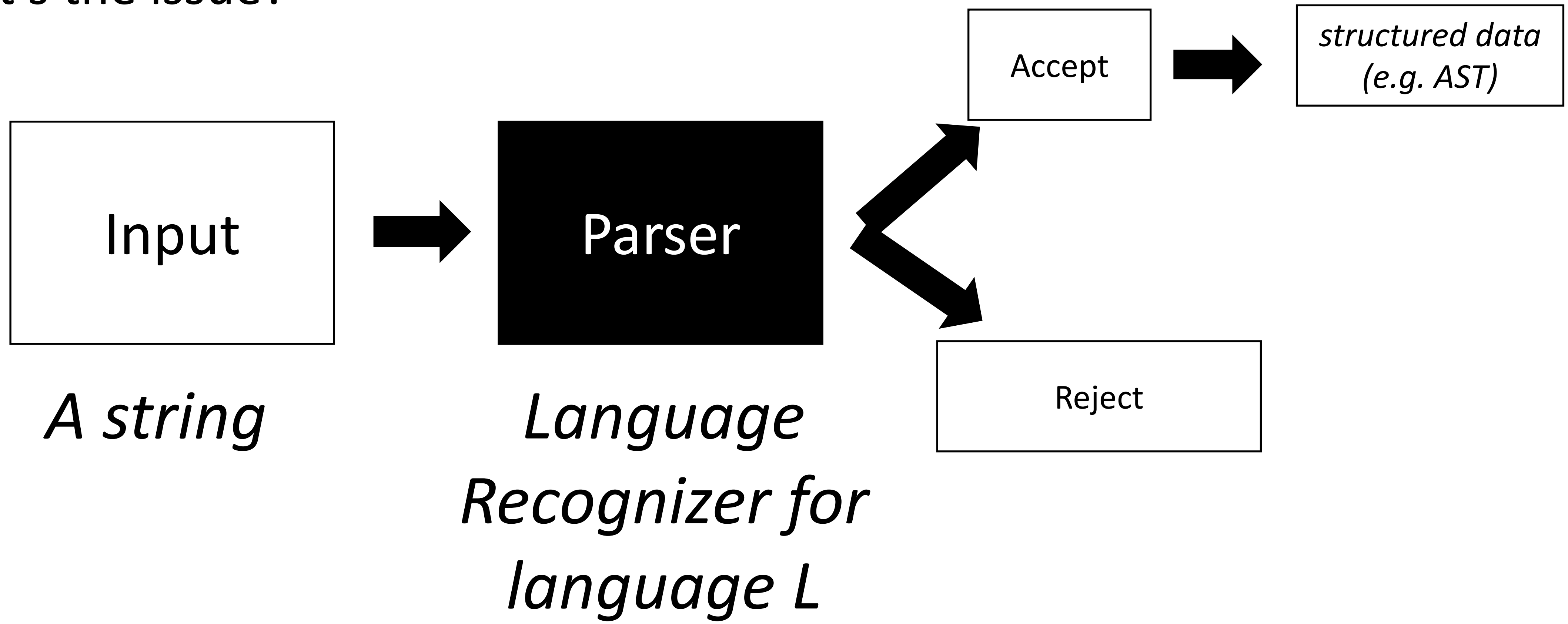
Ambiguous grammars

- What’s the issue?



Ambiguous grammars

- 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

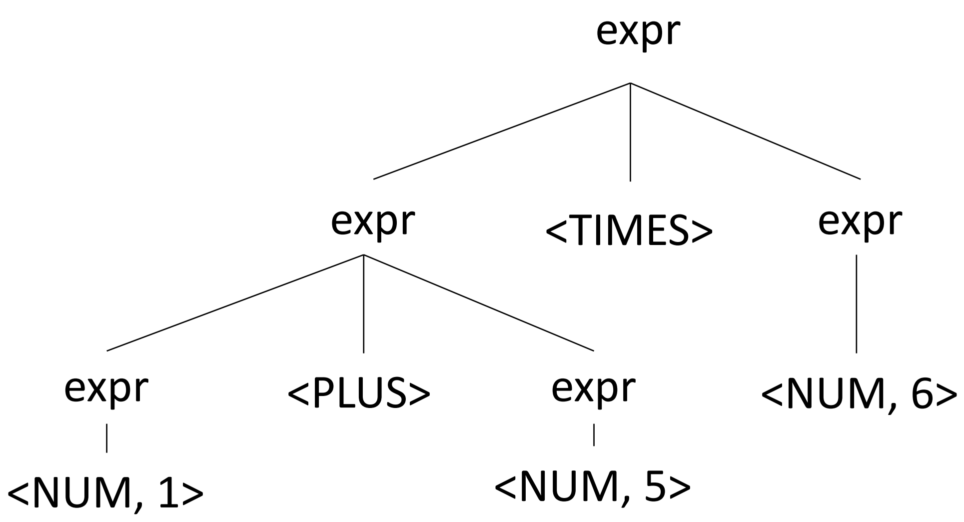
expr : NUM

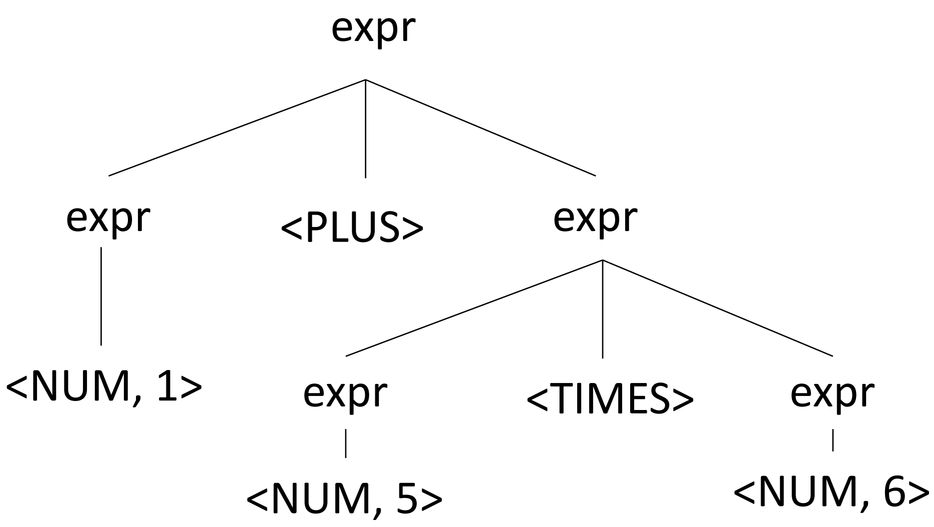
| expr PLUS expr

| expr TIMES expr

| LPAREN expr RPAREN

- `input: 1 + 5 * 6`





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`

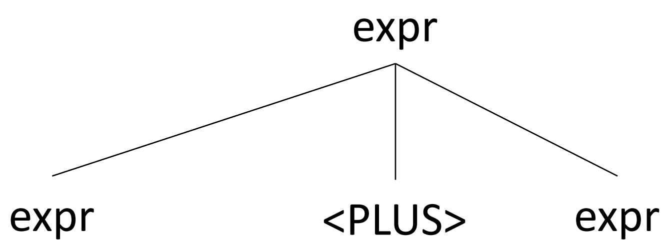
expr

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
```

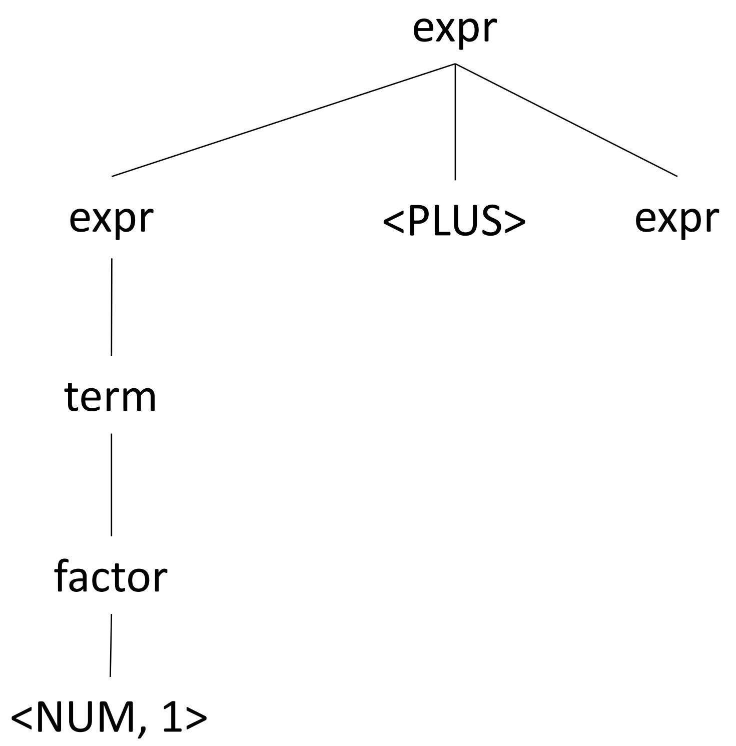
Operator	Name	Productions
+	expr	: expr PLUS expr | term
*	term	: term TIMES term | factor
()	factor	: LPAREN expr RPAREN | NUM



Now lets create a parse tree

input: 1+5*6

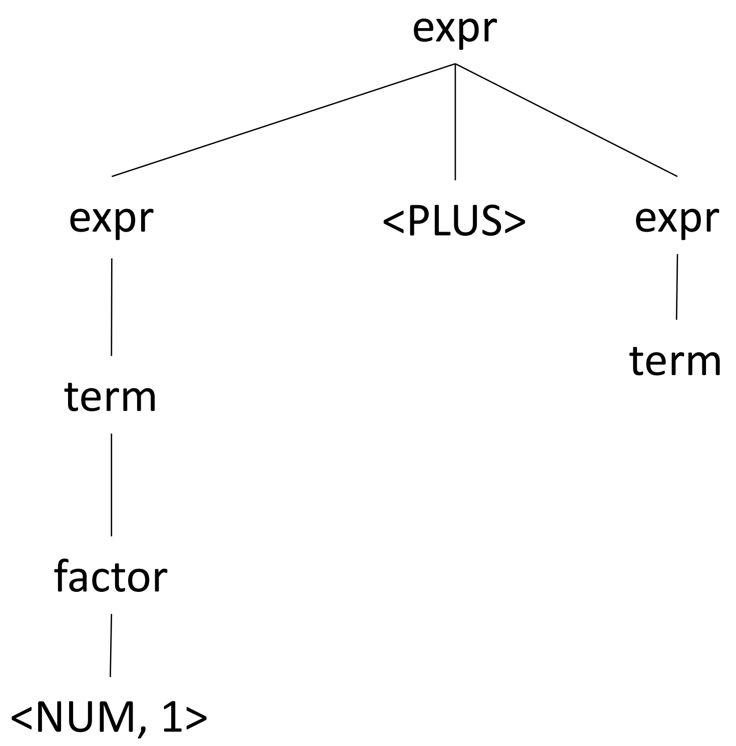
Operator	Name	Productions
+	expr	: expr PLUS expr | term
*	term	: term TIMES term | factor
()	factor	: LPAREN expr RPAREN | NUM



Now lets create a parse tree

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

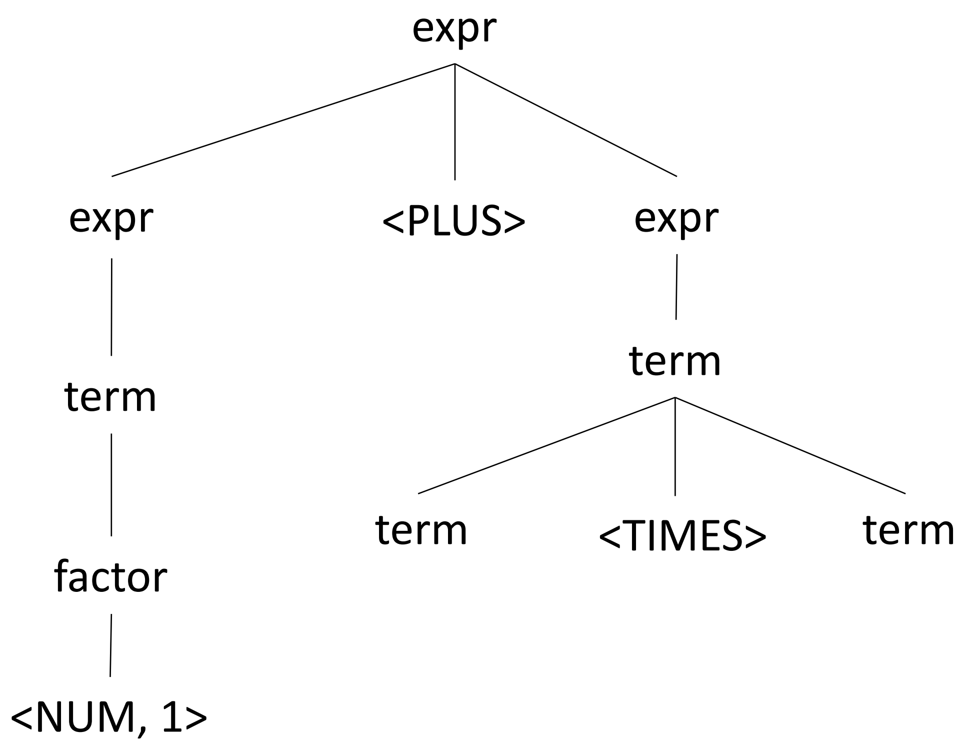
Operator	Name	Productions
+	expr	: expr PLUS expr | term
*	term	: term TIMES term | factor
()	factor	: LPAREN expr RPAREN | NUM



Now lets create a parse tree

`input: 1+5*6`

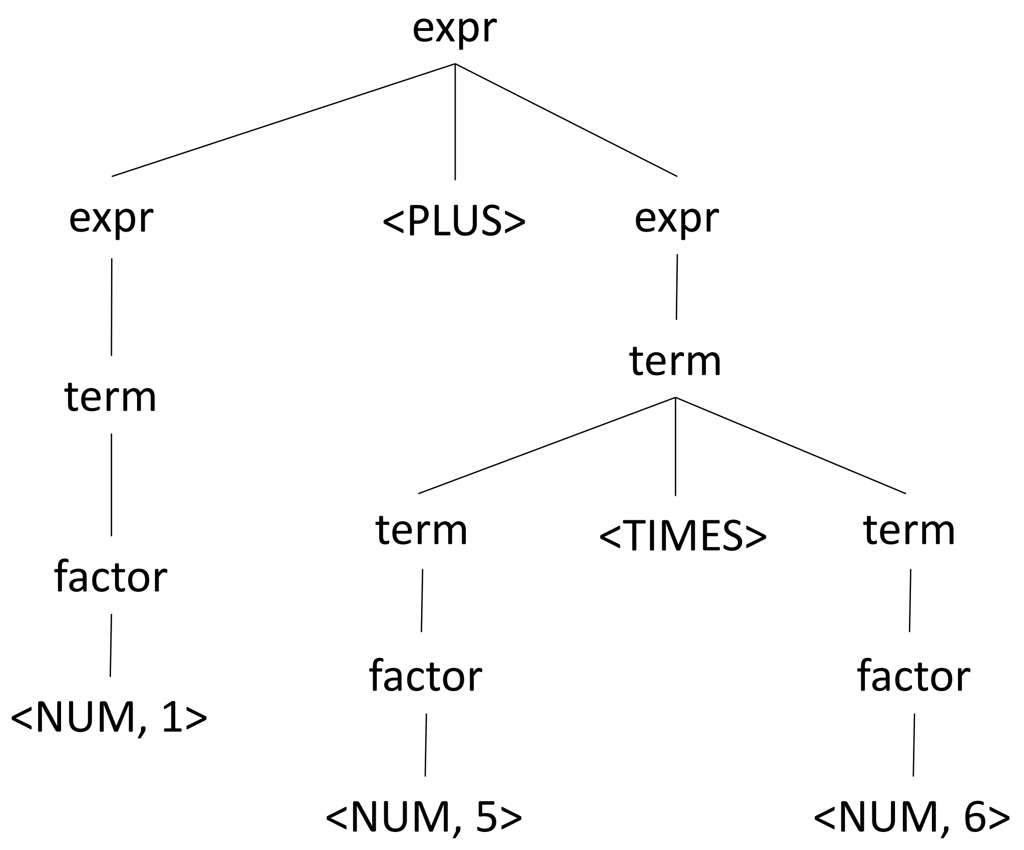
Operator	Name	Productions
+	expr	: expr PLUS expr | term
*	term	: term TIMES term | factor
()	factor	: LPAREN expr RPAREN | NUM



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
|		
.		
*		
()		

Parsing REs

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

<table>
<tr><th>Operator</th><th>Name</th><th>Productions</th></tr>
<tr><td>|</td><td>union</td><td></td></tr>
<tr><td>.</td><td>concat</td><td></td></tr>
<tr><td>*</td><td>starred</td><td></td></tr>
<tr><td>()</td><td>unit</td><td></td></tr>
</table>

Parsing REs

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

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

Parsing REs

`input: a.b | c*`

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

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

Parsing REs

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

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

`input: a.b | c*`